

Reference number of working document: **ISO/IEC JTC1/SC22/WG5 N1578**

Date: 2003-10-8

Reference number of document: **ISO/IEC FCD 1539-1:2004(E)**

Committee identification: ISO/IEC JTC1/SC22

Secretariat: ANSI

**Information technology - Programming languages - Fortran -  
Part 1: Base Language**

*Technologies de l'information - Langages de programmation - Fortran -  
Partie 1: Langage de base*



# Contents

1	Overview . . . . .	1
1.1	Scope . . . . .	1
1.2	Processor . . . . .	1
1.3	Inclusions . . . . .	1
1.4	Exclusions . . . . .	1
1.5	Conformance . . . . .	2
1.6	Compatibility . . . . .	3
1.6.1	Fortran 95 compatibility . . . . .	3
1.6.2	Fortran 90 compatibility . . . . .	3
1.6.3	FORTTRAN 77 compatibility . . . . .	4
1.7	Notation used in this standard . . . . .	4
1.7.1	Informative notes . . . . .	4
1.7.2	Syntax rules . . . . .	4
1.7.3	Constraints . . . . .	5
1.7.4	Assumed syntax rules . . . . .	6
1.7.5	Syntax conventions and characteristics . . . . .	6
1.7.6	Text conventions . . . . .	6
1.8	Deleted and obsolescent features . . . . .	7
1.8.1	Nature of deleted features . . . . .	7
1.8.2	Nature of obsolescent features . . . . .	7
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 . . . . .	15
2.4	Data concepts . . . . .	15
2.4.1	Type . . . . .	15
2.4.2	Data value . . . . .	16
2.4.3	Data entity . . . . .	16
2.4.4	Scalar . . . . .	17
2.4.5	Array . . . . .	18
2.4.6	Pointer . . . . .	18
2.4.7	Storage . . . . .	18
2.5	Fundamental terms . . . . .	18
2.5.1	Name and designator . . . . .	19
2.5.2	Keyword . . . . .	19
2.5.3	Association . . . . .	19
2.5.4	Declaration . . . . .	19

2.5.5	Definition . . . . .	19
2.5.6	Reference . . . . .	20
2.5.7	Intrinsic . . . . .	20
2.5.8	Operator . . . . .	20
2.5.9	Sequence . . . . .	20
2.5.10	Companion processors . . . . .	21
3	Characters, lexical tokens, and source form . . . . .	23
3.1	Processor character set . . . . .	23
3.1.1	Letters . . . . .	23
3.1.2	Digits . . . . .	24
3.1.3	Underscore . . . . .	24
3.1.4	Special characters . . . . .	24
3.1.5	Other characters . . . . .	24
3.2	Low-level syntax . . . . .	25
3.2.1	Names . . . . .	25
3.2.2	Constants . . . . .	25
3.2.3	Operators . . . . .	25
3.2.4	Statement labels . . . . .	26
3.2.5	Delimiters . . . . .	27
3.3	Source form . . . . .	27
3.3.1	Free source form . . . . .	27
3.3.2	Fixed source form . . . . .	29
3.4	Including source text . . . . .	30
4	Types . . . . .	33
4.1	The concept of type . . . . .	33
4.1.1	Set of values . . . . .	33
4.1.2	Constants . . . . .	33
4.1.3	Operations . . . . .	34
4.2	Type parameters . . . . .	34
4.3	Relationship of types and values to objects . . . . .	35
4.4	Intrinsic types . . . . .	35
4.4.1	Integer type . . . . .	36
4.4.2	Real type . . . . .	37
4.4.3	Complex type . . . . .	39
4.4.4	Character type . . . . .	40
4.4.5	Logical type . . . . .	43
4.5	Derived types . . . . .	44
4.5.1	Derived-type definition . . . . .	45
4.5.2	Derived-type parameters . . . . .	48
4.5.3	Components . . . . .	49
4.5.4	Type-bound procedures . . . . .	56
4.5.5	Final subroutines . . . . .	58
4.5.6	Type extension . . . . .	60
4.5.7	Derived-type values . . . . .	62
4.5.8	Derived-type specifier . . . . .	62
4.5.9	Construction of derived-type values . . . . .	63
4.5.10	Derived-type operations and assignment . . . . .	65
4.6	Enumerations and enumerators . . . . .	66
4.7	Construction of array values . . . . .	67
5	Data object declarations and specifications . . . . .	71
5.1	Type declaration statements . . . . .	71

5.1.1	Declaration type specifiers . . . . .	75
5.1.2	Attributes . . . . .	76
5.2	Attribute specification statements . . . . .	85
5.2.1	Accessibility statements . . . . .	86
5.2.2	ALLOCATABLE statement . . . . .	86
5.2.3	ASYNCHRONOUS statement . . . . .	86
5.2.4	BIND statement . . . . .	87
5.2.5	DATA statement . . . . .	87
5.2.6	DIMENSION statement . . . . .	90
5.2.7	INTENT statement . . . . .	90
5.2.8	OPTIONAL statement . . . . .	90
5.2.9	PARAMETER statement . . . . .	90
5.2.10	POINTER statement . . . . .	91
5.2.11	PROTECTED statement . . . . .	91
5.2.12	SAVE statement . . . . .	91
5.2.13	TARGET statement . . . . .	92
5.2.14	VALUE statement . . . . .	92
5.2.15	VOLATILE statement . . . . .	92
5.3	IMPLICIT statement . . . . .	92
5.4	NAMELIST statement . . . . .	95
5.5	Storage association of data objects . . . . .	95
5.5.1	EQUIVALENCE statement . . . . .	95
5.5.2	COMMON statement . . . . .	98
6	Use of data objects . . . . .	103
6.1	Scalars . . . . .	104
6.1.1	Substrings . . . . .	104
6.1.2	Structure components . . . . .	104
6.1.3	Type parameter inquiry . . . . .	106
6.2	Arrays . . . . .	106
6.2.1	Whole arrays . . . . .	107
6.2.2	Array elements and array sections . . . . .	107
6.3	Dynamic association . . . . .	110
6.3.1	ALLOCATE statement . . . . .	110
6.3.2	NULLIFY statement . . . . .	113
6.3.3	DEALLOCATE statement . . . . .	114
7	Expressions and assignment . . . . .	117
7.1	Expressions . . . . .	117
7.1.1	Form of an expression . . . . .	117
7.1.2	Intrinsic operations . . . . .	121
7.1.3	Defined operations . . . . .	122
7.1.4	Type, type parameters, and shape of an expression . . . . .	123
7.1.5	Conformability rules for elemental operations . . . . .	125
7.1.6	Specification expression . . . . .	125
7.1.7	Initialization expression . . . . .	126
7.1.8	Evaluation of operations . . . . .	128
7.2	Interpretation of operations . . . . .	132
7.2.1	Numeric intrinsic operations . . . . .	133
7.2.2	Character intrinsic operation . . . . .	134
7.2.3	Relational intrinsic operations . . . . .	134
7.2.4	Logical intrinsic operations . . . . .	135
7.3	Precedence of operators . . . . .	136
7.4	Assignment . . . . .	138

7.4.1	Assignment statement . . . . .	138
7.4.2	Pointer assignment . . . . .	143
7.4.3	Masked array assignment – WHERE . . . . .	145
7.4.4	FORALL . . . . .	148
8	Execution control . . . . .	155
8.1	Executable constructs containing blocks . . . . .	155
8.1.1	Rules governing blocks . . . . .	155
8.1.2	IF construct . . . . .	156
8.1.3	CASE construct . . . . .	158
8.1.4	ASSOCIATE construct . . . . .	160
8.1.5	SELECT TYPE construct . . . . .	162
8.1.6	DO construct . . . . .	164
8.2	Branching . . . . .	169
8.2.1	GO TO statement . . . . .	169
8.2.2	Computed GO TO statement . . . . .	170
8.2.3	Arithmetic IF statement . . . . .	170
8.3	CONTINUE statement . . . . .	170
8.4	STOP statement . . . . .	170
9	Input/output statements . . . . .	171
9.1	Records . . . . .	171
9.1.1	Formatted record . . . . .	171
9.1.2	Unformatted record . . . . .	172
9.1.3	Endfile record . . . . .	172
9.2	External files . . . . .	172
9.2.1	File existence . . . . .	173
9.2.2	File access . . . . .	173
9.2.3	File position . . . . .	175
9.2.4	File storage units . . . . .	177
9.3	Internal files . . . . .	177
9.4	File connection . . . . .	178
9.4.1	Connection modes . . . . .	179
9.4.2	Unit existence . . . . .	179
9.4.3	Connection of a file to a unit . . . . .	179
9.4.4	Preconnection . . . . .	180
9.4.5	The OPEN statement . . . . .	180
9.4.6	The CLOSE statement . . . . .	185
9.5	Data transfer statements . . . . .	186
9.5.1	Control information list . . . . .	186
9.5.2	Data transfer input/output list . . . . .	191
9.5.3	Execution of a data transfer input/output statement . . . . .	194
9.5.4	Termination of data transfer statements . . . . .	205
9.6	Waiting on pending data transfer . . . . .	205
9.6.1	WAIT statement . . . . .	206
9.6.2	Wait operation . . . . .	206
9.7	File positioning statements . . . . .	207
9.7.1	BACKSPACE statement . . . . .	207
9.7.2	ENDFILE statement . . . . .	208
9.7.3	REWIND statement . . . . .	208
9.8	FLUSH statement . . . . .	209
9.9	File inquiry . . . . .	209
9.9.1	Inquiry specifiers . . . . .	210
9.9.2	Restrictions on inquiry specifiers . . . . .	216

9.9.3	Inquire by output list . . . . .	216
9.10	Error, end-of-record, and end-of-file conditions . . . . .	216
9.10.1	Error conditions and the ERR= specifier . . . . .	217
9.10.2	End-of-file condition and the END= specifier . . . . .	217
9.10.3	End-of-record condition and the EOR= specifier . . . . .	218
9.10.4	IOSTAT= specifier . . . . .	218
9.10.5	IOMSG= specifier . . . . .	219
9.11	Restrictions on input/output statements . . . . .	219
10	Input/output editing . . . . .	221
10.1	Explicit format specification methods . . . . .	221
10.1.1	FORMAT statement . . . . .	221
10.1.2	Character format specification . . . . .	221
10.2	Form of a format item list . . . . .	222
10.2.1	Edit descriptors . . . . .	222
10.2.2	Fields . . . . .	224
10.3	Interaction between input/output list and format . . . . .	224
10.4	Positioning by format control . . . . .	225
10.5	Decimal symbol . . . . .	226
10.6	Data edit descriptors . . . . .	226
10.6.1	Numeric editing . . . . .	226
10.6.2	Logical editing . . . . .	232
10.6.3	Character editing . . . . .	232
10.6.4	Generalized editing . . . . .	233
10.6.5	User-defined derived-type editing . . . . .	235
10.7	Control edit descriptors . . . . .	235
10.7.1	Position editing . . . . .	235
10.7.2	Slash editing . . . . .	236
10.7.3	Colon editing . . . . .	236
10.7.4	SS, SP, and S editing . . . . .	237
10.7.5	P editing . . . . .	237
10.7.6	BN and BZ editing . . . . .	237
10.7.7	RU, RD, RZ, RN, RC, and RP editing . . . . .	238
10.7.8	DC and DP editing . . . . .	238
10.8	Character string edit descriptors . . . . .	238
10.9	List-directed formatting . . . . .	238
10.9.1	List-directed input . . . . .	239
10.9.2	List-directed output . . . . .	241
10.10	Namelist formatting . . . . .	242
10.10.1	Namelist input . . . . .	243
10.10.2	Namelist output . . . . .	246
11	Program units . . . . .	249
11.1	Main program . . . . .	249
11.2	Modules . . . . .	250
11.2.1	The USE statement and use association . . . . .	251
11.3	Block data program units . . . . .	253
12	Procedures . . . . .	255
12.1	Procedure classifications . . . . .	255
12.1.1	Procedure classification by reference . . . . .	255
12.1.2	Procedure classification by means of definition . . . . .	255
12.2	Characteristics of procedures . . . . .	256
12.2.1	Characteristics of dummy arguments . . . . .	256

12.2.2	Characteristics of function results . . . . .	257
12.3	Procedure interface . . . . .	257
12.3.1	Implicit and explicit interfaces . . . . .	257
12.3.2	Specification of the procedure interface . . . . .	258
12.4	Procedure reference . . . . .	266
12.4.1	Actual arguments, dummy arguments, and argument association . . . . .	268
12.4.2	Function reference . . . . .	276
12.4.3	Subroutine reference . . . . .	276
12.4.4	Resolving named procedure references . . . . .	276
12.4.5	Resolving type-bound procedure references . . . . .	278
12.5	Procedure definition . . . . .	279
12.5.1	Intrinsic procedure definition . . . . .	279
12.5.2	Procedures defined by subprograms . . . . .	279
12.5.3	Definition and invocation of procedures by means other than Fortran . . . . .	285
12.5.4	Statement function . . . . .	285
12.6	Pure procedures . . . . .	286
12.7	Elemental procedures . . . . .	287
12.7.1	Elemental procedure declaration and interface . . . . .	287
12.7.2	Elemental function actual arguments and results . . . . .	288
12.7.3	Elemental subroutine actual arguments . . . . .	289
13	Intrinsic procedures and modules . . . . .	291
13.1	Classes of intrinsic procedures . . . . .	291
13.2	Arguments to intrinsic procedures . . . . .	291
13.2.1	The shape of array arguments . . . . .	292
13.2.2	Mask arguments . . . . .	292
13.3	Bit model . . . . .	292
13.4	Numeric models . . . . .	293
13.5	Standard generic intrinsic procedures . . . . .	294
13.5.1	Numeric functions . . . . .	294
13.5.2	Mathematical functions . . . . .	294
13.5.3	Character functions . . . . .	295
13.5.4	Kind functions . . . . .	295
13.5.5	Miscellaneous type conversion functions . . . . .	295
13.5.6	Numeric inquiry functions . . . . .	295
13.5.7	Array inquiry functions . . . . .	296
13.5.8	Other inquiry functions . . . . .	296
13.5.9	Bit manipulation procedures . . . . .	296
13.5.10	Floating-point manipulation functions . . . . .	296
13.5.11	Vector and matrix multiply functions . . . . .	297
13.5.12	Array reduction functions . . . . .	297
13.5.13	Array construction functions . . . . .	297
13.5.14	Array location functions . . . . .	297
13.5.15	Null function . . . . .	297
13.5.16	Allocation transfer procedure . . . . .	297
13.5.17	Random number subroutines . . . . .	297
13.5.18	System environment procedures . . . . .	298
13.6	Specific names for standard intrinsic functions . . . . .	298
13.7	Specifications of the standard intrinsic procedures . . . . .	300
13.8	Standard intrinsic modules . . . . .	359
13.8.1	The IEEE modules . . . . .	359
13.8.2	The ISO_FORTRAN_ENV intrinsic module . . . . .	360
13.8.3	The ISO_C_BINDING module . . . . .	361

14	Exceptions and IEEE arithmetic . . . . .	363
14.1	Derived types and constants defined in the modules . . . . .	364
14.2	The exceptions . . . . .	365
14.3	The rounding modes . . . . .	367
14.4	Underflow mode . . . . .	367
14.5	Halting . . . . .	368
14.6	The floating point status . . . . .	368
14.7	Exceptional values . . . . .	368
14.8	IEEE arithmetic . . . . .	368
14.9	Tables of the procedures . . . . .	369
14.9.1	Inquiry functions . . . . .	369
14.9.2	Elemental functions . . . . .	370
14.9.3	Kind function . . . . .	370
14.9.4	Elemental subroutines . . . . .	370
14.9.5	Nonelemental subroutines . . . . .	371
14.10	Specifications of the procedures . . . . .	371
14.11	Examples . . . . .	386
15	Interoperability with C . . . . .	391
15.1	The ISO_C_BINDING intrinsic module . . . . .	391
15.1.1	Named constants and derived types in the module . . . . .	391
15.1.2	Procedures in the module . . . . .	392
15.2	Interoperability between Fortran and C entities . . . . .	395
15.2.1	Interoperability of intrinsic types . . . . .	396
15.2.2	Interoperability with C pointer types . . . . .	397
15.2.3	Interoperability of derived types and C struct types . . . . .	398
15.2.4	Interoperability of scalar variables . . . . .	399
15.2.5	Interoperability of array variables . . . . .	399
15.2.6	Interoperability of procedures and procedure interfaces . . . . .	400
15.3	Interoperation with C global variables . . . . .	402
15.3.1	Binding labels for common blocks and variables . . . . .	403
15.4	Interoperation with C functions . . . . .	403
15.4.1	Binding labels for procedures . . . . .	403
15.4.2	Exceptions and IEEE arithmetic procedures . . . . .	404
16	Scope, association, and definition . . . . .	405
16.1	Scope of global identifiers . . . . .	405
16.2	Scope of local identifiers . . . . .	406
16.2.1	Local identifiers that are the same as common block names . . . . .	407
16.2.2	Function results . . . . .	407
16.2.3	Restrictions on generic declarations . . . . .	407
16.2.4	Components, type parameters, and bindings . . . . .	408
16.2.5	Argument keywords . . . . .	409
16.3	Statement and construct entities . . . . .	409
16.4	Association . . . . .	410
16.4.1	Name association . . . . .	410
16.4.2	Pointer association . . . . .	414
16.4.3	Storage association . . . . .	415
16.4.4	Inheritance association . . . . .	418
16.4.5	Establishing associations . . . . .	418
16.5	Definition and undefinition of variables . . . . .	419
16.5.1	Definition of objects and subobjects . . . . .	419
16.5.2	Variables that are always defined . . . . .	419
16.5.3	Variables that are initially defined . . . . .	419

16.5.4	Variables that are initially undefined . . . . .	420
16.5.5	Events that cause variables to become defined . . . . .	420
16.5.6	Events that cause variables to become undefined . . . . .	421
16.5.7	Variable definition context . . . . .	423
A	Glossary of technical terms . . . . .	425
B	Decremental features . . . . .	437
B.1	Deleted features . . . . .	437
B.2	Obsolescent features . . . . .	438
B.2.1	Alternate return . . . . .	438
B.2.2	Computed GO TO statement . . . . .	438
B.2.3	Statement functions . . . . .	438
B.2.4	DATA statements among executables . . . . .	439
B.2.5	Assumed character length functions . . . . .	439
B.2.6	Fixed form source . . . . .	439
B.2.7	CHARACTER* form of CHARACTER declaration . . . . .	439
C	Extended notes . . . . .	441
C.1	Section 4 notes . . . . .	441
C.1.1	Intrinsic and derived types (4.4, 4.5) . . . . .	441
C.1.2	Selection of the approximation methods (4.4.2) . . . . .	442
C.1.3	Type extension and component accessibility (4.5.1.1, 4.5.3) . . . . .	442
C.1.4	Abstract types . . . . .	443
C.1.5	Pointers (4.5.1) . . . . .	444
C.1.6	Structure constructors and generic names . . . . .	445
C.1.7	Generic type-bound procedures . . . . .	447
C.1.8	Final subroutines (4.5.5, 4.5.5.1, 4.5.5.2, 4.5.5.3) . . . . .	448
C.2	Section 5 notes . . . . .	450
C.2.1	The POINTER attribute (5.1.2.11) . . . . .	450
C.2.2	The TARGET attribute (5.1.2.14) . . . . .	451
C.2.3	The VOLATILE attribute (5.1.2.16) . . . . .	451
C.3	Section 6 notes . . . . .	452
C.3.1	Structure components (6.1.2) . . . . .	452
C.3.2	Allocation with dynamic type (6.3.1) . . . . .	453
C.3.3	Pointer allocation and association . . . . .	454
C.4	Section 7 notes . . . . .	455
C.4.1	Character assignment . . . . .	455
C.4.2	Evaluation of function references . . . . .	455
C.4.3	Pointers in expressions . . . . .	455
C.4.4	Pointers on the left side of an assignment . . . . .	455
C.4.5	An example of a FORALL construct containing a WHERE construct . . . . .	456
C.4.6	Examples of FORALL statements . . . . .	457
C.5	Section 8 notes . . . . .	458
C.5.1	Loop control . . . . .	458
C.5.2	The CASE construct . . . . .	458
C.5.3	Examples of DO constructs . . . . .	458
C.5.4	Examples of invalid DO constructs . . . . .	460
C.6	Section 9 notes . . . . .	461
C.6.1	External files (9.2) . . . . .	461
C.6.2	Nonadvancing input/output (9.2.3.1) . . . . .	463
C.6.3	Asynchronous input/output . . . . .	464
C.6.4	OPEN statement (9.4.5) . . . . .	465
C.6.5	Connection properties (9.4.3) . . . . .	466

C.6.6	CLOSE statement (9.4.6) . . . . .	467
C.7	Section 10 notes . . . . .	467
C.7.1	Number of records (10.3, 10.4, 10.7.2) . . . . .	467
C.7.2	List-directed input (10.9.1) . . . . .	468
C.8	Section 11 notes . . . . .	468
C.8.1	Main program and block data program unit (11.1, 11.3) . . . . .	468
C.8.2	Dependent compilation (11.2) . . . . .	469
C.8.3	Examples of the use of modules . . . . .	471
C.9	Section 12 notes . . . . .	477
C.9.1	Portability problems with external procedures (12.3.2.2) . . . . .	477
C.9.2	Procedures defined by means other than Fortran (12.5.3) . . . . .	478
C.9.3	Procedure interfaces (12.3) . . . . .	478
C.9.4	Abstract interfaces (12.3) and procedure pointer components (4.5) . . . . .	478
C.9.5	Argument association and evaluation (12.4.1.2) . . . . .	480
C.9.6	Pointers and targets as arguments (12.4.1.2) . . . . .	481
C.9.7	Polymorphic Argument Association (12.4.1.3) . . . . .	482
C.10	Section 15 notes . . . . .	484
C.10.1	Runtime environments . . . . .	484
C.10.2	Examples of Interoperation between Fortran and C Functions . . . . .	484
C.11	Section 16 notes . . . . .	489
C.11.1	Examples of host association (16.4.1.3) . . . . .	489
C.11.2	Rules ensuring unambiguous generics (16.2.3) . . . . .	490
C.12	Array feature notes . . . . .	495
C.12.1	Summary of features . . . . .	495
C.12.2	Examples . . . . .	497
C.12.3	FORMula TRANslation and array processing . . . . .	501
C.12.4	Sum of squared residuals . . . . .	502
C.12.5	Vector norms: infinity-norm and one-norm . . . . .	502
C.12.6	Matrix norms: infinity-norm and one-norm . . . . .	502
C.12.7	Logical queries . . . . .	503
C.12.8	Parallel computations . . . . .	503
C.12.9	Example of element-by-element computation . . . . .	504
C.12.10	Bit manipulation and inquiry procedures . . . . .	504
D	Syntax rules . . . . .	505
D.1	Extract of all syntax rules . . . . .	505
D.2	Syntax rule cross-reference . . . . .	543
E	Index . . . . .	555



## List of Tables

2.1	Requirements on statement ordering . . . . .	14
2.2	Statements allowed in scoping units . . . . .	14
3.1	Special characters . . . . .	24
6.1	Subscript order value . . . . .	108
7.1	Type of operands and results for intrinsic operators . . . . .	121
7.2	Interpretation of the numeric intrinsic operators . . . . .	133
7.3	Interpretation of the character intrinsic operator // . . . . .	134
7.4	Interpretation of the relational intrinsic operators . . . . .	134
7.5	Interpretation of the logical intrinsic operators . . . . .	135
7.6	The values of operations involving logical intrinsic operators . . . . .	136
7.7	Categories of operations and relative precedence . . . . .	136
7.8	Type conformance for the intrinsic assignment statement . . . . .	139
7.9	Numeric conversion and the assignment statement . . . . .	141
13.1	Characteristics of the result of NULL ( ) . . . . .	341
15.1	Names of C characters with special semantics . . . . .	392
15.2	Interoperability between Fortran and C types . . . . .	396

## 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 nongovernmental, 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 fourth edition cancels and replaces the third edition (ISO/IEC 1539-1:1997), 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*
- *Part 3: Conditional Compilation*

The annexes 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, informally known as Fortran 2003. With the limitations noted in 1.6.2, the syntax and semantics of Fortran 95 are contained entirely within Fortran 2003. Therefore, any standard-conforming Fortran 95 program not affected by such limitations is a standard conforming Fortran 2003 program. New features of Fortran 2003 can be compatibly incorporated into such Fortran 95 programs, with any exceptions indicated in the text of this part of the standard.

Fortran 2003 contains several extensions to Fortran 95; among them are:

- (1) Derived-type enhancements: parameterized derived types (allows the kind, length, or shape of a derived type's components to be chosen when the derived type is used), mixed component accessibility (allows different components to have different accessibility), public entities of private type, improved structure constructors, and finalizers.
- (2) Object oriented programming support: enhanced data abstraction (allows one type to extend the definition of another type), polymorphism (allows the type of a variable to vary at runtime), dynamic type allocation, SELECT TYPE construct (allows a choice of execution flow depending upon the type a polymorphic object currently has), and type-bound procedures.
- (3) The ASSOCIATE construct (allows a complex expression or object to be denoted by a simple symbol).
- (4) Data manipulation enhancements: allocatable components, deferred type parameters, VOLATILE attribute, explicit type specification in array constructors, INTENT specification of pointer arguments, specified lower bounds of pointer assignment and pointer rank remapping, extended initialization expressions, MAX and MIN intrinsics for character type, and enhanced complex constants.
- (5) Input/output enhancements: asynchronous transfer operations (allows a program to continue to process data while an input/output transfer occurs), stream access (allows access to a file without reference to any record structure), user specified transfer operations for derived types, user specified control of rounding during format conversions, the FLUSH statement, named constants for preconnected units, regularization of input/output keywords, and access to input/output error messages.
- (6) Procedure pointers.
- (7) Scoping enhancements: the ability to rename defined operators (supports greater data abstraction) and control of host association into interface bodies.
- (8) Support for IEC 60559 (IEEE 754) exceptions and arithmetic (to the extent a processor's arithmetic supports the IEC standard).
- (9) Interoperability with the C programming language (allows portable access to many libraries and the low-level facilities provided by C and allows the portable use of Fortran libraries by programs written in C).
- (10) Support for international usage: (ISO 10646) and choice of decimal or comma in numeric formatted input/output.
- (11) Enhanced integration with the host operating system: access to command line arguments, environment variables, and the processor's error messages.

## Organization of this part of ISO/IEC 1539

This part of ISO/IEC 1539 is organized in 16 sections, dealing with 8 conceptual areas. These 8 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, 14
Execution control	Section 8
Input/output	Sections 9, 10
Program units	Sections 11, 12
Interoperability with C	Section 15
Scoping and association rules	Section 16

It also contains the following nonnormative material:

Glossary	A
Decremental features	B
Extended notes	C
Syntax rules	D
Index	E

# 1 Information technology — Programming languages —

## 2 Fortran —

### 3 Part 1:

#### 4 Base Language

## 5 Section 1: Overview

### 6 1.1 Scope

7 ISO/IEC 1539 is a multipart International Standard; the parts are published separately. This publi-  
8 cation, ISO/IEC 1539-1, which is the first part, specifies the form and establishes the interpretation  
9 of programs expressed in the base Fortran language. The purpose of this part of ISO/IEC 1539 is to  
10 promote portability, reliability, maintainability, and efficient execution of Fortran programs for use on  
11 a variety of computing systems. The second part, ISO/IEC 1539-2, defines additional facilities for the  
12 manipulation of character strings of variable length. The third part, ISO/IEC 1539-3, defines a stan-  
13 dard conditional compilation facility for Fortran. A processor conforming to part 1 need not conform to  
14 ISO/IEC 1539-2 or ISO/IEC 1539-3; however, conformance to either assumes conformance to this part.  
15 Throughout this publication, the term “this standard” refers to ISO/IEC 1539-1.

### 16 1.2 Processor

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

### 19 1.3 Inclusions

20 This standard specifies

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

### 25 1.4 Exclusions

26 This standard does not specify

- 27 (1) The mechanism by which programs are transformed for use on computing systems,
- 28 (2) The operations required for setup and control of the use of programs on computing systems,
- 29 (3) The method of transcription of programs or their input or output data to or from a storage  
30 medium,
- 31 (4) The program and processor behavior when this standard fails to establish an interpretation  
32 except for the processor detection and reporting requirements in items (2) through (8) of

- 1           1.5,  
2           (5) The size or complexity of a program and its data that will exceed the capacity of any  
3           particular computing system or the capability of a particular processor,  
4           (6) The physical properties of the representation of quantities and the method of rounding,  
5           approximating, or computing numeric values on a particular processor,  
6           (7) The physical properties of input/output records, files, and units, or  
7           (8) The physical properties and implementation of storage.

## 8 1.5 Conformance

9 A program (2.2.1) is a **standard-conforming program** if it uses only those forms and relationships  
10 described herein and if the program has an interpretation according to this standard. A program unit  
11 (2.2) conforms to this standard if it can be included in a program in a manner that allows the program  
12 to be standard conforming.

13 A processor conforms to this standard if

- 14           (1) It executes any standard-conforming program in a manner that fulfills the interpretations  
15           herein, subject to any limits that the processor may impose on the size and complexity of  
16           the program;  
17           (2) It contains the capability to detect and report the use within a submitted program unit of  
18           a form designated herein as obsolescent, insofar as such use can be detected by reference to  
19           the numbered syntax rules and constraints;  
20           (3) It contains the capability to detect and report the use within a submitted program unit of  
21           an additional form or relationship that is not permitted by the numbered syntax rules or  
22           constraints, including the deleted features described in Annex B;  
23           (4) It contains the capability to detect and report the use within a submitted program unit of  
24           an intrinsic type with a kind type parameter value not supported by the processor (4.4);  
25           (5) It contains the capability to detect and report the use within a submitted program unit of  
26           source form or characters not permitted by Section 3;  
27           (6) It contains the capability to detect and report the use within a submitted program of name  
28           usage not consistent with the scope rules for names, labels, operators, and assignment  
29           symbols in Section 16;  
30           (7) It contains the capability to detect and report the use within a submitted program unit of  
31           intrinsic procedures whose names are not defined in Section 13; and  
32           (8) It contains the capability to detect and report the reason for rejecting a submitted program.

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

35 A standard-conforming processor may allow additional forms and relationships provided that such ad-  
36 ditions do not conflict with the standard forms and relationships. However, a standard-conforming  
37 processor may allow additional intrinsic procedures even though this could cause a conflict with the  
38 name of a procedure in a standard-conforming program. If such a conflict occurs and involves the name  
39 of an external procedure, the processor is permitted to use the intrinsic procedure unless the name is  
40 given the EXTERNAL attribute (5.1.2.6) in the scoping unit (16). A standard-conforming program  
41 shall not use nonstandard intrinsic procedures or modules that have been added by the processor.

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

1 In some cases, this standard allows the provision of facilities that are not completely specified in the  
2 standard. These facilities are identified as **processor dependent**. They shall be provided, with methods  
3 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.

## 4 1.6 Compatibility

5 Each standard since ISO 1539:1980 (informally referred to as FORTRAN 77), defines more intrinsic  
6 procedures than the previous one. Therefore, a Fortran program conforming to an older standard may  
7 have a different interpretation under a newer standard if it invokes an external procedure having the  
8 same name as one of the new standard intrinsic procedures, unless that procedure is specified to have  
9 the EXTERNAL attribute.

### 10 1.6.1 Fortran 95 compatibility

11 Except as identified in this section, this standard is an upward compatible extension to the preceding  
12 Fortran International Standard, ISO/IEC 1539-1:1997 (Fortran 95). Any standard-conforming Fortran  
13 95 program remains standard-conforming under this standard. The following Fortran 95 features may  
14 have different interpretations in this standard:

- 15 (1) Earlier Fortran standards had the concept of printing, meaning that column one of format-  
16 ted output had special meaning for a processor-dependent (possibly empty) set of logical  
17 units. This could be neither detected nor specified by a standard-specified means. The  
18 interpretation of the first column is not specified by this standard.
- 19 (2) This standard specifies a different output format for real zero values in list-directed and  
20 namelist output.
- 21 (3) If the processor can distinguish between positive and negative real zero, this standard re-  
22 quires different returned values for ATAN2(Y,X) when  $X < 0$  and Y is negative real zero  
23 and for LOG(X) and SQRT(X) when X is complex with  $\text{REAL}(X) < 0$  and negative zero  
24 imaginary part.

### 25 1.6.2 Fortran 90 compatibility

26 Except for the deleted features noted in Annex B.1, and except as identified in this section, this stan-  
27 dard is an upward compatible extension to ISO/IEC 1539:1991 (Fortran 90). Any standard-conforming  
28 Fortran 90 program that does not use one of the deleted features remains standard-conforming under  
29 this standard.

30 The PAD= specifier in the INQUIRE statement in this standard returns the value UNDEFINED if there  
31 is no connection or the connection is for unformatted input/output. Fortran 90 specified YES.

32 Fortran 90 specified that if the second argument to MOD or MODULO was zero, the result was processor  
33 dependent. This standard specifies that the second argument shall not be zero.

### 1 1.6.3 FORTRAN 77 compatibility

2 Except for the deleted features noted in Annex B.1, and except as identified in this section, this standard  
3 is an upward compatible extension to ISO 1539:1980 (FORTRAN 77). Any standard-conforming FOR-  
4 TRAN 77 program that does not use one of the deleted features noted in Annex B.1 and that does not  
5 depend on the differences specified here remains standard conforming under this standard. This stan-  
6 dard restricts the behavior for some features that were processor dependent in FORTRAN 77. Therefore,  
7 a standard-conforming FORTRAN 77 program that uses one of these processor-dependent features may  
8 have a different interpretation under this standard, yet remain a standard-conforming program. The  
9 following FORTRAN 77 features may have different interpretations in this standard:

- 10 (1) FORTRAN 77 permitted a processor to supply more precision derived from a real constant  
11 than can be represented in a real datum when the constant is used to initialize a data  
12 object of type double precision real in a DATA statement. This standard does not permit  
13 a processor this option.
- 14 (2) If a named variable that was not in a common block was initialized in a DATA statement and  
15 did not have the SAVE attribute specified, FORTRAN 77 left its SAVE attribute processor  
16 dependent. This standard specifies (5.2.5) that this named variable has the SAVE attribute.
- 17 (3) FORTRAN 77 specified that the number of characters required by the input list was to be  
18 less than or equal to the number of characters in the record during formatted input. This  
19 standard specifies (9.5.3.4.2) that the input record is logically padded with blanks if there  
20 are not enough characters in the record, unless the PAD= specifier with the value 'NO' is  
21 specified in an appropriate OPEN or READ statement.
- 22 (4) A value of 0 for a list item in a formatted output statement will be formatted in a differ-  
23 ent form for some G edit descriptors. In addition, this standard specifies how rounding of  
24 values will affect the output field form, but FORTRAN 77 did not address this issue. There-  
25 fore, some FORTRAN 77 processors may produce an output form different from the output  
26 form produced by Fortran 2003 processors for certain combinations of values and G edit  
27 descriptors.
- 28 (5) If the processor can distinguish between positive and negative real zero, the behavior of the  
29 SIGN intrinsic function when the second argument is negative real zero is changed by this  
30 standard.

## 31 1.7 Notation used in this standard

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

### 35 1.7.1 Informative notes

36 Informative notes of explanation, rationale, examples, and other material are interspersed with the  
37 normative body of this publication. The informative material is nonnormative; it is identified by being  
38 in shaded, framed boxes that have numbered headings beginning with “NOTE.”

### 39 1.7.2 Syntax rules

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

- 42 (1) Characters from the Fortran character set (3.1) are interpreted literally as shown, except  
43 where otherwise noted.

1 (2) Lower-case italicized letters and words (often hyphenated and abbreviated) represent gen-  
 2 eral syntactic classes for which particular syntactic entities shall be substituted in actual  
 3 statements.

4 Common abbreviations used in syntactic terms are:

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

5 (3) The syntactic metasymbols used are:

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

6 (4) Each syntax rule is given a unique identifying number of the form R<sub>snn</sub>, where s is a one-  
 7 or two-digit section number and nn is a two-digit sequence number within that section.  
 8 The syntax rules are distributed as appropriate throughout the text, and are referenced by  
 9 number as needed. Some rules in Sections 2 and 3 are more fully described in later sections;  
 10 in such cases, the section number s is the number of the later section where the rule is  
 11 repeated.

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

#### NOTE 1.2

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*

If particular entities are substituted for *digit*, actual digit strings might be:

4  
 67  
 1999  
 10243852

### 15 1.7.3 Constraints

16 Each constraint is given a unique identifying number of the form C<sub>snn</sub>, where s is a one- or two-digit  
 17 section number and nn is a two-digit sequence number within that section.

1 Often a constraint is associated with a particular syntax rule. Where that is the case, the constraint is  
 2 annotated with the syntax rule number in parentheses. A constraint that is associated with a syntax  
 3 rule constitutes part of the definition of the syntax term defined by the rule. It thus applies in all places  
 4 where the syntax term appears.

5 Some constraints are not associated with particular syntax rules. The effect of such a constraint is similar  
 6 to that of a restriction stated in the text, except that a processor is required to have the capability to  
 7 detect and report violations of constraints (1.5). In some cases, a broad requirement is stated in text  
 8 and a subset of the same requirement is also stated as a constraint. This indicates that a standard-  
 9 conforming program is required to adhere to the broad requirement, but that a standard-conforming  
 10 processor is required only to have the capability of diagnosing violations of the constraint.

#### 11 1.7.4 Assumed syntax rules

12 In order to minimize the number of additional syntax rules and convey appropriate constraint informa-  
 13 tion, the following rules are assumed; an explicit syntax rule for a term overrides an assumed rule. The  
 14 letters “*xyz*” stand for any syntactic class phrase:

15 R101 *xyz-list* is *xyz* [ , *xyz* ] ...  
 16 R102 *xyz-name* is *name*  
 17 R103 *scalar-xyz* is *xyz*

18 C101 (R103) *scalar-xyz* shall be scalar.

#### 19 1.7.5 Syntax conventions and characteristics

- 20 (1) Any syntactic class name ending in “-*stmt*” follows the source form statement rules: it shall  
 21 be delimited by end-of-line or semicolon, and may be labeled unless it forms part of another  
 22 statement (such as an IF or WHERE statement). Conversely, everything considered to be  
 23 a source form statement is given a “-*stmt*” ending in the syntax rules.
- 24 (2) The rules on statement ordering are described rigorously in the definition of *program-unit*  
 25 (R202). Expression hierarchy is described rigorously in the definition of *expr* (R722).
- 26 (3) The suffix “-*spec*” is used consistently for specifiers, such as input/output statement speci-  
 27 fiers. It also is used for type declaration attribute specifications (for example, “*array-spec*”  
 28 in R510), and in a few other cases.
- 29 (4) Where reference is made to a type parameter, including the surrounding parentheses, the  
 30 suffix “-*selector*” is used. See, for example, “*kind-selector*” (R404) and “*length-selector*”  
 31 (R425).
- 32 (5) The term “*subscript*” (for example, R618, R619, and R620) is used consistently in array  
 33 definitions.

#### 34 1.7.6 Text conventions

35 In the descriptive text, an equivalent English word is frequently used in place of a syntactic term.  
 36 Particular statements and attributes are identified in the text by an upper-case keyword, e.g., “END  
 37 statement”. Boldface words are used in the text where they are first defined with a specialized meaning.  
 38 The descriptions of obsolescent features appear in a smaller type size.

##### NOTE 1.3

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

## 1.8 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 and Fortran 95, but are still frequently used.

### 1.8.1 Nature of deleted features

- (1) Better methods existed in FORTRAN 77.
- (2) These features are not included in Fortran 95 or this revision of Fortran.

### 1.8.2 Nature of obsolescent features

- (1) Better methods existed in Fortran 90 and Fortran 95.
- (2) It is recommended that programmers use these better methods in new programs and convert existing code to these methods.
- (3) These features are identified in the text of this document by a distinguishing type font (1.7.6).
- (4) If the use of these features becomes insignificant, future Fortran standards committees should consider deleting them.
- (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.

## 1.9 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/IEC 646:1991, *Information technology—ISO 7-bit coded character set for information interchange*. ISO/IEC 646:1991 (International Reference Version) is the international equivalent of ANSI X3.4-1986, commonly known as ASCII.

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

ISO/IEC 9899:1999, *Information technology—Programming languages—C*. This standard refers to ISO/IEC 9899:1999 as the C standard.

ISO/IEC 10646-1:2000, *Information technology—Universal multiple-octet coded character set (UCS)—Part 1: Architecture and basic multilingual plane*.

IEC 60559 (1989-01), *Binary floating-point arithmetic for microprocessor systems*.

Because IEC 60559 (1989-01) was originally IEEE 754-1985, *Standard for binary floating-point arithmetic*, and is widely known by this name, this standard refers to it as the IEEE standard.



## 1 Section 2: Fortran terms and concepts

### 2 2.1 High level syntax

3 This section introduces the terms associated with program units and other Fortran concepts above the  
 4 construct, statement, and expression levels and illustrates their relationships. The notation used in this  
 5 standard is described in 1.7.

#### NOTE 2.1

Constraints and other information related to the rules that do not begin with R2 appear in the appropriate section.

6	R201	<i>program</i>	<b>is</b>	<i>program-unit</i> [ <i>program-unit</i> ] ...
7				
8				A <i>program</i> shall contain exactly one <i>main-program program-unit</i> or a main program defined by means
9				other than Fortran, but not both.
10	R202	<i>program-unit</i>	<b>is</b>	<i>main-program</i>
11			<b>or</b>	<i>external-subprogram</i>
12			<b>or</b>	<i>module</i>
13			<b>or</b>	<i>block-data</i>
14	R1101	<i>main-program</i>	<b>is</b>	[ <i>program-stmt</i> ] [ <i>specification-part</i> ] [ <i>execution-part</i> ] [ <i>internal-subprogram-part</i> ] <i>end-program-stmt</i>
15				
16				
17				
18				
19	R203	<i>external-subprogram</i>	<b>is</b>	<i>function-subprogram</i>
20			<b>or</b>	<i>subroutine-subprogram</i>
21	R1223	<i>function-subprogram</i>	<b>is</b>	<i>function-stmt</i> [ <i>specification-part</i> ] [ <i>execution-part</i> ] [ <i>internal-subprogram-part</i> ] <i>end-function-stmt</i>
22				
23				
24				
25				
26	R1231	<i>subroutine-subprogram</i>	<b>is</b>	<i>subroutine-stmt</i> [ <i>specification-part</i> ] [ <i>execution-part</i> ] [ <i>internal-subprogram-part</i> ] <i>end-subroutine-stmt</i>
27				
28				
29				
30				
31	R1104	<i>module</i>	<b>is</b>	<i>module-stmt</i> [ <i>specification-part</i> ] [ <i>module-subprogram-part</i> ] <i>end-module-stmt</i>
32				
33				
34				
35	R1116	<i>block-data</i>	<b>is</b>	<i>block-data-stmt</i> [ <i>specification-part</i> ] <i>end-block-data-stmt</i>
36				
37				
38	R204	<i>specification-part</i>	<b>is</b>	[ <i>use-stmt</i> ] ... [ <i>import-stmt</i> ] ... [ <i>implicit-part</i> ] [ <i>declaration-construct</i> ] ...
39				
40				
41				

1	R205	<i>implicit-part</i>	<b>is</b>	[ <i>implicit-part-stmt</i> ] ...
2				<i>implicit-stmt</i>
3	R206	<i>implicit-part-stmt</i>	<b>is</b>	<i>implicit-stmt</i>
4			<b>or</b>	<i>parameter-stmt</i>
5			<b>or</b>	<i>format-stmt</i>
6			<b>or</b>	<i>entry-stmt</i>
7	R207	<i>declaration-construct</i>	<b>is</b>	<i>derived-type-def</i>
8			<b>or</b>	<i>entry-stmt</i>
9			<b>or</b>	<i>enum-def</i>
10			<b>or</b>	<i>format-stmt</i>
11			<b>or</b>	<i>interface-block</i>
12			<b>or</b>	<i>parameter-stmt</i>
13			<b>or</b>	<i>procedure-declaration-stmt</i>
14			<b>or</b>	<i>specification-stmt</i>
15			<b>or</b>	<i>type-declaration-stmt</i>
16			<b>or</b>	<i>stmt-function-stmt</i>
17	R208	<i>execution-part</i>	<b>is</b>	<i>executable-construct</i>
18				[ <i>execution-part-construct</i> ] ...
19	R209	<i>execution-part-construct</i>	<b>is</b>	<i>executable-construct</i>
20			<b>or</b>	<i>format-stmt</i>
21			<b>or</b>	<i>entry-stmt</i>
22			<b>or</b>	<i>data-stmt</i>
23	R210	<i>internal-subprogram-part</i>	<b>is</b>	<i>contains-stmt</i>
24				<i>internal-subprogram</i>
25				[ <i>internal-subprogram</i> ] ...
26	R211	<i>internal-subprogram</i>	<b>is</b>	<i>function-subprogram</i>
27			<b>or</b>	<i>subroutine-subprogram</i>
28	R1107	<i>module-subprogram-part</i>	<b>is</b>	<i>contains-stmt</i>
29				<i>module-subprogram</i>
30				[ <i>module-subprogram</i> ] ...
31	R1108	<i>module-subprogram</i>	<b>is</b>	<i>function-subprogram</i>
32			<b>or</b>	<i>subroutine-subprogram</i>
33	R212	<i>specification-stmt</i>	<b>is</b>	<i>access-stmt</i>
34			<b>or</b>	<i>allocatable-stmt</i>
35			<b>or</b>	<i>asynchronous-stmt</i>
36			<b>or</b>	<i>bind-stmt</i>
37			<b>or</b>	<i>common-stmt</i>
38			<b>or</b>	<i>data-stmt</i>
39			<b>or</b>	<i>dimension-stmt</i>
40			<b>or</b>	<i>equivalence-stmt</i>
41			<b>or</b>	<i>external-stmt</i>
42			<b>or</b>	<i>intent-stmt</i>
43			<b>or</b>	<i>intrinsic-stmt</i>
44			<b>or</b>	<i>namelist-stmt</i>
45			<b>or</b>	<i>optional-stmt</i>
46			<b>or</b>	<i>pointer-stmt</i>
47			<b>or</b>	<i>protected-stmt</i>
48			<b>or</b>	<i>save-stmt</i>
49			<b>or</b>	<i>target-stmt</i>
50			<b>or</b>	<i>volatile-stmt</i>
51			<b>or</b>	<i>value-stmt</i>
52	R213	<i>executable-construct</i>	<b>is</b>	<i>action-stmt</i>
53			<b>or</b>	<i>associate-construct</i>
54			<b>or</b>	<i>case-construct</i>

1		<b>or</b> <i>do-construct</i>
2		<b>or</b> <i>forall-construct</i>
3		<b>or</b> <i>if-construct</i>
4		<b>or</b> <i>select-type-construct</i>
5		<b>or</b> <i>where-construct</i>
6	R214	<b>is</b> <i>allocate-stmt</i>
7		<b>or</b> <i>assignment-stmt</i>
8		<b>or</b> <i>backspace-stmt</i>
9		<b>or</b> <i>call-stmt</i>
10		<b>or</b> <i>close-stmt</i>
11		<b>or</b> <i>continue-stmt</i>
12		<b>or</b> <i>cycle-stmt</i>
13		<b>or</b> <i>deallocate-stmt</i>
14		<b>or</b> <i>endfile-stmt</i>
15		<b>or</b> <i>end-function-stmt</i>
16		<b>or</b> <i>end-program-stmt</i>
17		<b>or</b> <i>end-subroutine-stmt</i>
18		<b>or</b> <i>exit-stmt</i>
19		<b>or</b> <i>flush-stmt</i>
20		<b>or</b> <i>forall-stmt</i>
21		<b>or</b> <i>goto-stmt</i>
22		<b>or</b> <i>if-stmt</i>
23		<b>or</b> <i>inquire-stmt</i>
24		<b>or</b> <i>nullify-stmt</i>
25		<b>or</b> <i>open-stmt</i>
26		<b>or</b> <i>pointer-assignment-stmt</i>
27		<b>or</b> <i>print-stmt</i>
28		<b>or</b> <i>read-stmt</i>
29		<b>or</b> <i>return-stmt</i>
30		<b>or</b> <i>rewind-stmt</i>
31		<b>or</b> <i>stop-stmt</i>
32		<b>or</b> <i>wait-stmt</i>
33		<b>or</b> <i>where-stmt</i>
34		<b>or</b> <i>write-stmt</i>
35		<b>or</b> <i>arithmetic-if-stmt</i>
36		<b>or</b> <i>computed-goto-stmt</i>
37	C201	(R208) An <i>execution-part</i> shall not contain an <i>end-function-stmt</i> , <i>end-program-stmt</i> , or <i>end-</i>
38		<i>subroutine-stmt</i> .

## 39 2.2 Program unit concepts

40 Program units are the fundamental components of a Fortran program. A **program unit** may be  
 41 a main program, an external subprogram, a module, or a block data program unit. A subprogram  
 42 may be a function subprogram or a subroutine subprogram. A module contains definitions that are  
 43 to be made accessible to other program units. A block data program unit is used to specify initial  
 44 values for data objects in named common blocks. Each type of program unit is described in Section  
 45 11 or 12. An **external subprogram** is a subprogram that is not in a main program, a module, or  
 46 another subprogram. An **internal subprogram** is a subprogram that is in a main program or another  
 47 subprogram. A **module subprogram** is a subprogram that is in a module but is not an internal  
 48 subprogram.

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

- 1 (1) A program unit or subprogram, excluding any scoping units in it,
- 2 (2) A derived-type definition (4.5.1), or
- 3 (3) An interface body, excluding any scoping units in it.

4 A scoping unit that immediately surrounds another scoping unit is called the **host scoping unit** (often  
5 abbreviated to **host**).

## 6 2.2.1 Program

7 A **program** consists of exactly one main program, any number (including zero) of other kinds of program  
8 units, and any number (including zero) of external procedures and other entities defined by means other  
9 than Fortran.

### NOTE 2.2

There is a restriction that there shall be no more than one unnamed block data program unit (11.3).

This standard places no ordering requirement on the program units that constitute a program, but because the public portions of a module are required to be available by the time a module reference (11.2.1) is processed, a processor may require a particular order of processing of the program units.

## 10 2.2.2 Main program

11 The Fortran main program is described in 11.1.

## 12 2.2.3 Procedure

13 A **procedure** encapsulates an arbitrary sequence of actions that may be invoked directly during program  
14 execution. Procedures are either functions or subroutines. A **function** is a procedure that is invoked  
15 in an expression; its invocation causes a value to be computed which is then used in evaluating the  
16 expression. The variable that returns the value of a function is called the **result variable**. A **subroutine**  
17 is a procedure that is invoked in a CALL statement, by a defined assignment statement, or by some  
18 operations on derived-type entities. Unless it is a pure procedure, a subroutine may be used to change  
19 the program state by changing the values of any of the data objects accessible to the subroutine; unless  
20 it is a pure procedure, a function may do this in addition to computing the function value.

21 Procedures are described further in Section 12.

### 22 2.2.3.1 External procedure

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

### 26 2.2.3.2 Module procedure

27 A **module procedure** is a procedure that is defined by a module subprogram (R1108). The module  
28 containing the subprogram is the **host** scoping unit of the module procedure.

### 29 2.2.3.3 Internal procedure

30 An **internal procedure** is a procedure that is defined by an internal subprogram (R211). The containing  
31 main program or subprogram is the **host** scoping unit of the internal procedure. An internal procedure

1 is local to its host in the sense that the internal procedure is accessible within the host scoping unit and  
2 all its other internal procedures but is not accessible elsewhere.

#### 3 **2.2.3.4 Interface block**

4 An **interface body** describes an abstract interface or the interface of a dummy procedure, external  
5 procedure, procedure pointer, or type-bound procedure.

6 An **interface block** is a specific interface block, an abstract interface block, or a generic interface block.  
7 A specific interface block is a collection of interface bodies. A generic interface block may also be used  
8 to specify that procedures may be invoked

- 9 (1) By using a generic name,
- 10 (2) By using a defined operator,
- 11 (3) By using a defined assignment, or
- 12 (4) For derived-type input/output.

#### 13 **2.2.4 Module**

14 A **module** contains (or accesses from other modules) definitions that are to be made accessible to other  
15 program units. These definitions include data object declarations, type definitions, procedure definitions,  
16 and interface blocks. A scoping unit in another program unit may access the definitions in a module.  
17 Modules are further described in Section 11.

### 18 **2.3 Execution concepts**

19 Each Fortran statement is classified as either an executable statement or a nonexecutable statement.  
20 There are restrictions on the order in which statements may appear in a program unit, and not all  
21 executable statements may appear in all contexts.

#### 22 **2.3.1 Executable/nonexecutable statements**

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

27 **Nonexecutable statements** do not specify actions; they are used to configure the program environment  
28 in which actions take place. The nonexecutable statements are all those not classified as executable.

#### 29 **2.3.2 Statement order**

30 The syntax rules of subclause 2.1 specify the statement order within program units and subprograms.  
31 These rules are illustrated in Table 2.1 and Table 2.2. Table 2.1 shows the ordering rules for statements  
32 and applies to all program units, subprograms, and interface bodies. Vertical lines delineate varieties of  
33 statements that may be interspersed and horizontal lines delineate varieties of statements that shall not  
34 be interspersed. Internal or module subprograms shall follow a CONTAINS statement. Between USE  
35 and CONTAINS statements in a subprogram, nonexecutable statements generally precede executable  
36 statements, although the ENTRY statement, FORMAT statement, and DATA statement may appear  
37 among the executable statements. Table 2.2 shows which statements are allowed in a scoping unit.

Table 2.1: Requirements on statement ordering

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

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
IMPORT statement	No	No	No	No	No	No	Yes
ENTRY statement	No	No	No	Yes	Yes	No	No
FORMAT statement	Yes	No	No	Yes	Yes	Yes	No
Misc. decls (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:

- Misc. declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, enum statements, procedure declaration statements, and specification statements.
- The scoping unit of a module does not include any module subprograms that the module contains.

### 1 2.3.3 The END statement

- 2 An *end-program-stmt*, *end-function-stmt*, *end-subroutine-stmt*, *end-module-stmt*, or *end-block-data-stmt*  
3 is an **END statement**. Each program unit, module subprogram, and internal subprogram shall have  
4 exactly one END statement. The *end-program-stmt*, *end-function-stmt*, and *end-subroutine-stmt* state-  
5 ments are executable, and may be branch target statements (8.2). Executing an *end-program-stmt* causes  
6 normal termination of execution of the program. Executing an *end-function-stmt* or *end-subroutine-stmt*

1 is equivalent to executing a *return-stmt* with no *scalar-int-expr*.

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

### 3 **2.3.4 Execution sequence**

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

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

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

20 Normal termination of execution of the program occurs if a STOP statement or *end-program-stmt* is  
21 executed. Normal termination of execution of a program also may occur during execution of a procedure  
22 defined by a companion processor (C standard 5.1.2.2.3 and 7.20.4.3). If normal termination of execution  
23 occurs within a Fortran program unit and the program incorporates procedures defined by a companion  
24 processor, the process of execution termination shall include the effect of executing the C exit() function  
25 (C standard 7.20.4.3).

## 26 **2.4 Data concepts**

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

### 29 **2.4.1 Type**

30 A **type** is a named category of data that is characterized by a set of values, a syntax for denoting  
31 these values, and a set of operations that interpret and manipulate the values. This central concept is  
32 described in 4.1.

33 A type may be parameterized, in which case the set of data values, the syntax for denoting them, and  
34 the set of operations depend on the values of one or more parameters. Such a parameter is called a **type**  
35 **parameter** (4.2).

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

#### 37 **2.4.1.1 Intrinsic type**

38 An **intrinsic type** is a type that is defined by the language, along with operations, and is always  
39 accessible. The intrinsic types are integer, real, complex, character, and logical. The properties of  
40 intrinsic types are described in 4.4. The intrinsic type parameters are KIND and LEN.

1 The **kind type parameter** indicates the decimal exponent range for the integer type (4.4.1), the  
 2 decimal precision and exponent range for the real and complex types (4.4.2, 4.4.3), and the representation  
 3 methods for the character and logical types (4.4.4, 4.4.5). The **character length parameter** specifies  
 4 the number of characters for the character type.

#### 5 2.4.1.2 Derived type

6 A **derived type** is a type that is not defined by the language but requires a type definition to declare its  
 7 components. A scalar object of such a derived type is called a **structure** (5.1.1.1). Derived types may  
 8 be parameterized. Assignment of structures is defined intrinsically (7.4.1.3), but there are no intrinsic  
 9 operations for structures. For each derived type, a structure constructor is available to provide values  
 10 (4.5.9). In addition, data objects of derived type may be used as procedure arguments and function  
 11 results, and may appear in input/output lists. If additional operations are needed for a derived type,  
 12 they shall be supplied as procedure definitions.

13 Derived types are described further in 4.5.

#### 14 2.4.2 Data value

15 Each intrinsic type has associated with it a set of values that a datum of that type may take, depending  
 16 on the values of the type parameters. The values for each intrinsic type are described in 4.4. The values  
 17 that objects of a derived type may assume are determined by the type definition, type parameter values,  
 18 and the sets of values of its components.

#### 19 2.4.3 Data entity

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

##### 24 2.4.3.1 Data object

25 A **data object** (often abbreviated to **object**) is a constant (4.1.2), a variable (6), or a subobject of a  
 26 constant. The type and type parameters of a named data object may be specified explicitly (5.1) or  
 27 implicitly (5.3).

28 **Subobjects** are portions of certain objects that may be referenced and defined (variables only) inde-  
 29 pendently of the other portions. These include portions of arrays (array elements and array sections),  
 30 portions of character strings (substrings), portions of complex objects (real and imaginary parts), and  
 31 portions of structures (components). Subobjects are themselves data objects, but subobjects are refer-  
 32 enced only by object designators or intrinsic functions. A subobject of a variable is a variable. Subobjects  
 33 are described in Section 6.

34 Objects referenced by a name are:

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

36 Subobjects referenced by an object designator are:

37	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)

1 Subobjects of complex objects may also be referenced by intrinsic functions.

#### 2 **2.4.3.1.1 Variable**

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

4 A named **local variable** of the scoping unit of a module, main program, or subprogram, is a named  
5 variable that is a local entity of the scoping unit, is not a dummy argument, is not in COMMON, does  
6 not have the BIND attribute, and is not accessed by use or host association. A subobject of a named  
7 local variable is also a local variable.

#### 8 **2.4.3.1.2 Constant**

9 A **constant** has a value and cannot become defined, redefined, or undefined during execution of a  
10 program. A constant with a name is called a **named constant** and has the PARAMETER attribute  
11 (5.1.2.10). A constant without a name is called a **literal constant** (4.4).

#### 12 **2.4.3.1.3 Subobject of a constant**

13 A **subobject of a constant** is a portion of a constant. The portion referenced may depend on the  
14 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.

#### 15 **2.4.3.2 Expression**

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

#### 19 **2.4.3.3 Function reference**

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

#### 23 **2.4.4 Scalar**

24 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.

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

## 1 2.4.5 Array

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

5 An array may have up to seven dimensions, and any **extent** (number of elements) in any dimension.  
6 The **rank** of the array is the number of dimensions; its **size** is the total number of elements, which is  
7 equal to the product of the extents. An array may have zero size. The **shape** of an array is determined  
8 by its rank and its extent in each dimension, and may be represented as a rank-one array whose elements  
9 are the extents. All named arrays shall be declared, and the rank of a named array is specified in its  
10 declaration. The rank of a named array, once declared, is constant; the extents may be constant or may  
11 vary during execution.

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

18 A rank-one array may be constructed from scalars and other arrays and may be reshaped into any  
19 allowable array shape (4.7).

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

## 21 2.4.6 Pointer

22 A **data pointer** is a data entity that has the POINTER attribute. A **procedure pointer** is a procedure  
23 entity that has the POINTER attribute. A **pointer** is either a data pointer or a procedure pointer.

24 A pointer is **associated** with a **target** by pointer assignment (7.4.2). A data pointer may also be  
25 associated with a target by allocation (6.3.1). A pointer is **disassociated** following execution of a  
26 NULLIFY statement, following pointer assignment with a disassociated pointer, by default initialization,  
27 or by explicit initialization. A data pointer may also be disassociated by execution of a DEALLOCATE  
28 statement. A disassociated pointer is not associated with a target (16.4.2).

29 A pointer that is not associated shall not be referenced or defined.

30 If a data pointer is an array, the rank is declared, but the extents are determined when the pointer is  
31 associated with a target.

## 32 2.4.7 Storage

33 Many of the facilities of this standard make no assumptions about the physical storage characteristics of  
34 data objects. However, program units that include storage association dependent features shall observe  
35 the storage restrictions described in 16.4.3.

## 36 2.5 Fundamental terms

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

**1 2.5.1 Name and designator**

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

6 An **object designator** is a designator for a data object. A **procedure designator** is a designator for  
 7 a procedure.

**NOTE 2.5**

An object name is a special case of an object designator.

**8 2.5.2 Keyword**

9 The term **keyword** is used in two ways.

- 10 (1) It is used to describe a word that is part of the syntax of a statement. These keywords are  
 11 not reserved words; that is, names with the same spellings are allowed. In the syntax rules,  
 12 such keywords appear literally. In descriptive text, this meaning is denoted by the term  
 13 “keyword” without any modifier. Examples of statement keywords are: IF, READ, UNIT,  
 14 KIND, and INTEGER.
- 15 (2) It is used to denote names that identify items in a list. In actual argument lists, type  
 16 parameter lists, and structure constructors, items may be identified by a preceding *keyword*=  
 17 rather than their position within the list. An **argument keyword** is the name of a dummy  
 18 argument in the interface for the procedure being referenced, a **type parameter keyword**  
 19 is the name of a type parameter in the type being specified, and a **component keyword**  
 20 is the name of a component in a structure constructor.

21 R215 *keyword*                                    **is** *name*

**NOTE 2.6**

Use of keywords rather than position to identify items in a list can make such lists more readable  
 and allows them to be reordered. This facilitates specification of a list in cases where optional  
 items are omitted.

**22 2.5.3 Association**

23 **Association** is name association (16.4.1), pointer association (16.4.2), storage association (16.4.3),  
 24 or inheritance association (16.4.4). Name association is argument association, host association, use  
 25 association, linkage association, or construct association.

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

**29 2.5.4 Declaration**

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

**32 2.5.5 Definition**

33 The term **definition** is used in two ways.

- 1 (1) It refers to the specification of derived types and procedures.  
2 (2) When an object is given a valid value during program execution, it is said to become  
3 **defined**. This is often accomplished by execution of an assignment or input statement.  
4 When a variable does not have a predictable value, it is said to be **undefined**. Similarly,  
5 when a pointer is associated with a target or nullified, its pointer association status is said  
6 to become **defined**. When the association status of a pointer is not predictable, its pointer  
7 association status is said to be **undefined**.
- 8 Section 16 describes the ways in which variables may become defined and undefined.

### 9 2.5.6 Reference

- 10 A **data object reference** is the appearance of the data object designator in a context requiring its  
11 value at that point during execution.
- 12 A **procedure reference** is the appearance of the procedure designator, operator symbol, or assignment  
13 symbol in a context requiring execution of the procedure at that point. An occurrence of user-defined  
14 derived-type input/output (10.6.5) or derived-type finalization (4.5.5.1) is also a procedure reference.
- 15 The appearance of a data object designator or procedure designator in an actual argument list does not  
16 constitute a reference to that data object or procedure unless such a reference is necessary to complete  
17 the specification of the actual argument.
- 18 A **module reference** is the appearance of a module name in a USE statement (11.2.1).

### 19 2.5.7 Intrinsic

- 20 The qualifier **intrinsic** has two meanings.
- 21 (1) The qualifier signifies that the term to which it is applied is defined in this standard. In-  
22 trinsic applies to types, procedures, modules, assignment statements, and operators. All  
23 intrinsic types, procedures, assignments, and operators may be used in any scoping unit  
24 without further definition or specification. Intrinsic modules may be accessed by use as-  
25 sociation. Intrinsic procedures and modules defined in this standard are called standard  
26 intrinsic procedures and standard intrinsic modules, respectively.
- 27 (2) The qualifier applies to procedures or modules that are provided by a processor but are not  
28 defined in this standard (13, 14, 15.1). Such procedures and modules are called nonstandard  
29 intrinsic procedures and nonstandard intrinsic modules, respectively.

### 30 2.5.8 Operator

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

### 35 2.5.9 Sequence

- 36 A **sequence** is a set ordered by a one-to-one correspondence with the numbers 1, 2, through  $n$ . The  
37 number of elements in the sequence is  $n$ . A sequence may be empty, in which case it contains no elements.
- 38 The elements of a nonempty sequence are referred to as the first element, second element, etc. The  
39  $n$ th element, where  $n$  is the number of elements in the sequence, is called the last element. An empty  
40 sequence has no first or last element.

## 1 2.5.10 Companion processors

2 A processor has one or more companion processors. A **companion processor** is a processor-dependent  
3 mechanism by which global data and procedures may be referenced or defined. A companion processor  
4 may be a mechanism that references and defines such entities by a means other than Fortran (12.5.3),  
5 it may be the Fortran processor itself, or it may be another Fortran processor. If there is more than  
6 one companion processor, the means by which the Fortran processor selects among them are processor  
7 dependent.

8 If a procedure is defined by means of a companion processor that is not the Fortran processor itself,  
9 this standard refers to the C function that defines the procedure, although the procedure need not be  
10 defined by means of the C programming language.

### NOTE 2.7

A companion processor might or might not be a mechanism that conforms to the requirements of the C standard.

For example, a processor may allow a procedure defined by some language other than Fortran or C to be linked (12.5.3) with a Fortran procedure if it can be described by a C prototype as defined in 6.5.5.3 of the C standard.



## 1 Section 3: Characters, lexical tokens, and source form

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

### 4 3.1 Processor character set

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

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

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

14 R301	<i>character</i>	<b>is</b>	<i>alphanumeric-character</i>
15		<b>or</b>	<i>special-character</i>
16 R302	<i>alphanumeric-character</i>	<b>is</b>	<i>letter</i>
17		<b>or</b>	<i>digit</i>
18		<b>or</b>	<i>underscore</i>

19 Except for the currency symbol, the graphics used for the characters shall be as given in 3.1.1, 3.1.2,  
20 3.1.3, and 3.1.4. However, the style of any graphic is not specified.

21 The default character type shall support a character set that includes the Fortran character set. By sup-  
22 plying nondefault character types, the processor may support additional character sets. The characters  
23 available in the ASCII and ISO 10646 character sets are specified by ISO/IEC 646:1991 (International  
24 Reference Version) and ISO/IEC 10646-1:2000 UCS-4, respectively; the characters available in other non-  
25 default character types are not specified by this standard, except that one character in each nondefault  
26 character type shall be designated as a blank character to be used as a padding character.

#### 27 3.1.1 Letters

28 The twenty-six **letters** are:

29 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

30 The set of letters defines the syntactic class *letter*. The processor character set shall include lower-  
31 case and upper-case letters. A lower-case letter is equivalent to the corresponding upper-case letter in  
32 program units except in a character context (3.3).

#### NOTE 3.1

The following statements are equivalent:

```
CALL BIG_COMPLEX_OPERATION (NDATE)
call big_complex_operation (ndate)
```

**NOTE 3.1 (cont.)**

Call Big_Complex_Operation (NDate)
------------------------------------

**1 3.1.2 Digits**

2 The ten **digits** are:

3     0 1 2 3 4 5 6 7 8 9

4 The ten digits define the syntactic class *digit*.

**5 3.1.3 Underscore**

6 R303   *underscore*                    **is**   \_

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

**8 3.1.4 Special characters**

9 The **special characters** are shown in Table 3.1.

Table 3.1: **Special characters**

Character	Name of character	Character	Name of character
	Blank	;	Semicolon
=	Equals	!	Exclamation point
+	Plus	"	Quotation mark or quote
-	Minus	%	Percent
*	Asterisk	&	Ampersand
/	Slash	~	Tilde
\	Backslash	<	Less than
(	Left parenthesis	>	Greater than
)	Right parenthesis	?	Question mark
[	Left square bracket	'	Apostrophe
]	Right square bracket	`	Grave accent
{	Left curly bracket	^	Circumflex accent
}	Right curly bracket		Vertical line
,	Comma	\$	Currency symbol
.	Decimal point or period	#	Number sign
:	Colon	@	Commercial at

10 The special characters define the syntactic class *special-character*. Some of the special characters are  
 11 used for operator symbols, bracketing, and various forms of separating and delimiting other lexical  
 12 tokens.

**13 3.1.5 Other characters**

14 Additional characters may be representable in the processor, but may appear only in comments (3.3.1.1,  
 15 3.3.2.1), character constants (4.4.4), input/output records (9.1.1), and character string edit descriptors  
 16 (10.2.1).

## 1 3.2 Low-level syntax

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

### 6 3.2.1 Names

7 **Names** are used for various entities such as variables, program units, dummy arguments, named con-  
 8 stants, and derived types.

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

10 C301 (R304) The maximum length of a *name* is 63 characters.

#### NOTE 3.2

Examples of names:

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

#### NOTE 3.3

The word “name” always denotes this particular syntactic form. The word “identifier” is used where entities may be identified by other syntactic forms or by values; its particular meaning depends on the context in which it is used.

### 11 3.2.2 Constants

12	R305	<i>constant</i>	<b>is</b>	<i>literal-constant</i>
13			<b>or</b>	<i>named-constant</i>
14	R306	<i>literal-constant</i>	<b>is</b>	<i>int-literal-constant</i>
15			<b>or</b>	<i>real-literal-constant</i>
16			<b>or</b>	<i>complex-literal-constant</i>
17			<b>or</b>	<i>logical-literal-constant</i>
18			<b>or</b>	<i>char-literal-constant</i>
19			<b>or</b>	<i>boz-literal-constant</i>
20	R307	<i>named-constant</i>	<b>is</b>	<i>name</i>
21	R308	<i>int-constant</i>	<b>is</b>	<i>constant</i>

22 C302 (R308) *int-constant* shall be of type integer.

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

24 C303 (R309) *char-constant* shall be of type character.

### 25 3.2.3 Operators

26	R310	<i>intrinsic-operator</i>	<b>is</b>	<i>power-op</i>
27			<b>or</b>	<i>mult-op</i>
28			<b>or</b>	<i>add-op</i>
29			<b>or</b>	<i>concat-op</i>

1		<b>or</b> <i>rel-op</i>
2		<b>or</b> <i>not-op</i>
3		<b>or</b> <i>and-op</i>
4		<b>or</b> <i>or-op</i>
5		<b>or</b> <i>equiv-op</i>
6	R707 <i>power-op</i>	<b>is</b> **
7	R708 <i>mult-op</i>	<b>is</b> *
8		<b>or</b> /
9	R709 <i>add-op</i>	<b>is</b> +
10		<b>or</b> -
11	R711 <i>concat-op</i>	<b>is</b> //
12	R713 <i>rel-op</i>	<b>is</b> .EQ.
13		<b>or</b> .NE.
14		<b>or</b> .LT.
15		<b>or</b> .LE.
16		<b>or</b> .GT.
17		<b>or</b> .GE.
18		<b>or</b> ==
19		<b>or</b> /=
20		<b>or</b> <
21		<b>or</b> <=
22		<b>or</b> >
23		<b>or</b> >=
24	R718 <i>not-op</i>	<b>is</b> .NOT.
25	R719 <i>and-op</i>	<b>is</b> .AND.
26	R720 <i>or-op</i>	<b>is</b> .OR.
27	R721 <i>equiv-op</i>	<b>is</b> .EQV.
28		<b>or</b> .NEQV.
29	R311 <i>defined-operator</i>	<b>is</b> <i>defined-unary-op</i>
30		<b>or</b> <i>defined-binary-op</i>
31		<b>or</b> <i>extended-intrinsic-op</i>
32	R703 <i>defined-unary-op</i>	<b>is</b> . letter [ letter ] ... .
33	R723 <i>defined-binary-op</i>	<b>is</b> . letter [ letter ] ... .
34	R312 <i>extended-intrinsic-op</i>	<b>is</b> <i>intrinsic-operator</i>

### 35 3.2.4 Statement labels

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

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

38 C304 (R313) At least one digit in a *label* shall be nonzero.

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

#### NOTE 3.4

For example:

```
99999
10
010
```

**NOTE 3.4 (cont.)**

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.

1 Any statement may have a statement label, but the labels are used only in the following ways:

- 2 (1) The label on a branch target statement (8.2) is used to identify that statement as the  
3 possible destination of a branch.
- 4 (2) The label on a FORMAT statement (10.1.1) is used to identify that statement as the format  
5 specification for a data transfer statement (9.5).
- 6 (3) In some forms of the DO construct (8.1.6), the range of the DO construct is identified by  
7 the label on the last statement in that range.

**8 3.2.5 Delimiters**

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

- 10 ( ... )  
11 / ... /  
12 [ ... ]  
13 (/ ... /)

**14 3.3 Source form**

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

19 A **character context** means characters within a character literal constant (4.4.4) or within a character  
20 string edit descriptor (10.2.1).

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

22 There are two **source forms**: free and fixed. Free form and fixed form shall not be mixed in the same program unit.

23 The means for specifying the source form of a program unit are processor dependent.

**24 3.3.1 Free source form**

25 In **free source form** there are no restrictions on where a statement (or portion of a statement) may  
26 appear within a line. A line may contain zero characters. If a line consists entirely of characters of  
27 default kind (4.4.4), it may contain at most 132 characters. If a line contains any character that is not  
28 of default kind, the maximum number of characters allowed on the line is processor dependent.

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

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

**NOTE 3.5**

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

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

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

Adjacent keywords where separating blanks are optional

BLOCK DATA	DOUBLE PRECISION
ELSE IF	ELSE WHERE
END ASSOCIATE	END BLOCK DATA
END DO	END ENUM
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
SELECT TYPE	

5 **3.3.1.1 Free form commentary**

- 6 The character “!” initiates a **comment** except where it appears within a character context. The  
7 comment extends to the end of the line. If the first nonblank character on a line is an “!”, the line  
8 is a comment line. Lines containing only blanks or containing no characters are also comment lines.  
9 Comments may appear anywhere in a program unit and may precede the first statement of a program  
10 unit or may follow the last statement of a program unit. Comments have no effect on the interpretation  
11 of the program unit.

**NOTE 3.6**

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

12 **3.3.1.2 Free form statement continuation**

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

- 18 If a noncharacter context is to be continued, an “&” shall be the last nonblank character on the line,  
19 or the last nonblank character before an “!”. There shall be a later line that is not a comment; the  
20 statement is continued on the next such line. If the first nonblank character on that line is an “&”, the  
21 statement continues at the next character position following that “&”; otherwise, it continues with the  
22 first character position of that line.

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

3 If a character context is to be continued, an “&” shall be the last nonblank character on the line and  
4 shall not be followed by commentary. There shall be a later line that is not a comment; an “&” shall be  
5 the first nonblank character on the next such line and the statement continues with the next character  
6 following that “&”.

### 7 **3.3.1.3 Free form statement termination**

8 If a statement is not continued, a comment or the end of the line terminates the statement.

9 A statement may alternatively be terminated by a “;” character that appears other than in a character  
10 context or in a comment. The “;” is not part of the statement. After a “;” terminator, another statement  
11 may appear on the same line, or begin on that line and be continued. A “;” shall not appear as the first  
12 nonblank character on a line. A sequence consisting only of zero or more blanks and one or more “;”  
13 terminators, in any order, is equivalent to a single “;” terminator.

### 14 **3.3.1.4 Free form statements**

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

#### **NOTE 3.7**

No Fortran statement begins with a digit.

16 A statement shall not have more than 255 continuation lines.

## 17 **3.3.2 Fixed source form**

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

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

### 22 **3.3.2.1 Fixed form commentary**

23 The character “!” initiates a **comment** except where it appears within a character context or in character position 6. The  
24 comment extends to the end of the line. If the first nonblank character on a line is an “!” in any character position other  
25 than character position 6, the line is a comment line. Lines beginning with a “C” or “\*” in character position 1 and lines  
26 containing only blanks are also comment lines. Comments may appear anywhere in a program unit and may precede the  
27 first statement of the program unit or may follow the last statement of a program unit. Comments have no effect on the  
28 interpretation of the program unit.

#### **NOTE 3.8**

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

### 29 **3.3.2.2 Fixed form statement continuation**

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

#### **NOTE 3.9**

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.

1 Comment lines cannot be continued. Comment lines may occur within a continued statement.

### 2 3.3.2.3 Fixed form statement termination

3 If a statement is not continued, a comment or the end of the line terminates the statement.

4 A statement may alternatively be terminated by a “;” character that appears other than in a character context, in a  
5 comment, or in character position 6. The “;” is not part of the statement. After a “;” terminator, another statement may  
6 begin on the same line, or begin on that line and be continued. A “;” shall not appear as the first nonblank character on  
7 a line, except in character position 6. A sequence consisting only of zero or more blanks and one or more “;” terminators,  
8 in any order, is equivalent to a single “;” terminator.

### 9 3.3.2.4 Fixed form statements

10 A label, if present, shall occur in character positions 1 through 5 of the first line of a statement; otherwise, positions 1  
11 through 5 shall be blank. Blanks may appear anywhere within a label. A statement following a “;” on the same line shall  
12 not be labeled. Character positions 1 through 5 of any continuation lines shall be blank. A statement shall not have more  
13 than 255 continuation lines. The program unit END statement shall not be continued. A statement whose initial line  
14 appears to be a program unit END statement shall not be continued.

## 15 3.4 Including source text

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

18 `INCLUDE char-literal-constant`

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

20 An INCLUDE line is not a Fortran statement.

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

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

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

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

#### NOTE 3.10

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;

**NOTE 3.10 (cont.)**

- (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.



## 1 Section 4: Types

2 Fortran provides an abstract means whereby data may be categorized without relying on a particular  
3 physical representation. This abstract means is the concept of type.

4 An intrinsic type is one that is defined by the language. The intrinsic types are integer, real, complex,  
5 character, and logical.

6 A derived type is one that is defined by a derived-type definition (4.5.1).

7 A derived type may be used only where its definition is accessible (4.5.1.1). An intrinsic type is always  
8 accessible.

9 A type is specified in several contexts by a **type specifier**.

10 R401 *type-spec*                               **is** *intrinsic-type-spec*  
11   **or** *derived-type-spec*

12 C401 (R401) The *derived-type-spec* shall not specify an abstract type (4.5.6).

### 13 4.1 The concept of type

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

#### NOTE 4.1

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

#### 16 4.1.1 Set of values

17 For each type, there is a set of valid values. The set of valid values may be completely determined, as is  
18 the case for logical, or may be determined by a processor-dependent method, as is the case for integer,  
19 character, and real. For complex, the set of valid values consists of the set of all the combinations of the  
20 values of the individual components. For derived types, the set of valid values is as defined in 4.5.7.

#### 21 4.1.2 Constants

22 The syntax for literal constants of each intrinsic type is specified in 4.4.

23 The syntax for denoting a value indicates the type, type parameters, and the particular value.

24 A constant value may be given a name (5.1.2.10, 5.2.9).

25 A structure constructor (4.5.9) may be used to construct a constant value of derived type from an  
26 appropriate sequence of initialization expressions (7.1.7). Such a constant value is considered to be a  
27 scalar even though the value may have components that are arrays.

### 1 4.1.3 Operations

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

6 For derived types, there are no intrinsic operations. Operations on derived types may be defined by the  
 7 program (4.5.10).

## 8 4.2 Type parameters

9 A type may be parameterized. In this case, the set of values, the syntax for denoting the values, and  
 10 the set of operations on the values of the type depend on the values of the parameters.

11 The intrinsic types are all parameterized. Derived types may be defined to be parameterized.

12 A type parameter is either a kind type parameter or a length type parameter.

13 A kind type parameter may be used in initialization and specification expressions within the derived-type  
 14 definition (4.5.1) for the type; it participates in generic resolution (16.2.3). Each of the intrinsic types  
 15 has a kind type parameter named KIND, which is used to distinguish multiple representations of the  
 16 intrinsic type.

#### NOTE 4.2

By design, the value of a kind type parameter is known at compile time. Some parameterizations that involve multiple representation forms need to be distinguished at compile time for practical implementation and performance. Examples include the multiple precisions of the intrinsic real type and the possible multiple character sets of the intrinsic character type.

A type parameter of a derived type may be specified to be a kind type parameter in order to allow generic resolution based on the parameter; that is to allow a single generic to include two specific procedures that have interfaces distinguished only by the value of a kind type parameter of a dummy argument. Generics are designed to be resolvable at compile time.

17 A length type parameter may be used in specification expressions within the derived-type definition for  
 18 the type, but it shall not be used in initialization expressions. The intrinsic character type has a length  
 19 type parameter named LEN, which is the length of the string.

#### NOTE 4.3

The adjective “length” is used for type parameters other than kind type parameters because they often specify a length, as for intrinsic character type. However, they may be used for other purposes. The important difference from kind type parameters is that their values need not be known at compile time and might change during execution.

20 A type parameter value may be specified with a type specification (4.4, 4.5.8).

21	R402	<i>type-param-value</i>	is	<i>scalar-int-expr</i>
22			or	*
23			or	:

24 C402 (R402) The *type-param-value* for a kind type parameter shall be an initialization expression.

25 C403 (R402) A colon may be used as a *type-param-value* only in the declaration of an entity or

1 component that has the POINTER or ALLOCATABLE attribute.

2 A **deferred type parameter** is a length type parameter whose value can change during execution of  
3 the program. A colon as a *type-param-value* specifies a deferred type parameter.

4 The values of the deferred type parameters of an object are determined by successful execution of an  
5 ALLOCATE statement (6.3.1), execution of an intrinsic assignment statement (7.4.1.3), execution of a  
6 pointer assignment statement (7.4.2), or by argument association (12.4.1.2).

#### NOTE 4.4

Deferred type parameters of functions, including function procedure pointers, have no values. Instead, they indicate that those type parameters of the function result will be determined by execution of the function, if it returns an allocated allocatable result or an associated pointer result.

7 An **assumed type parameter** is a length type parameter for a dummy argument that assumes the  
8 type parameter value from the corresponding actual argument; it is also used for an associate name in a  
9 SELECT TYPE construct that assumes the type parameter value from the corresponding selector. An  
10 asterisk as a *type-param-value* specifies an assumed type parameter.

### 11 4.3 Relationship of types and values to objects

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

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

19 A variable is a data object. The type and type parameters of a variable determine which values that  
20 variable may take. Assignment provides one means of defining or redefining the value of a variable of  
21 any type. Assignment is defined intrinsically for all types where the type, type parameters, and shape  
22 of both the variable and the value to be assigned to it are identical. Assignment between objects of  
23 certain differing intrinsic types, type parameters, and shapes is described in Section 7. A subroutine and  
24 a generic interface (4.5.1, 12.3.2.1) whose generic specifier is ASSIGNMENT (=) define an assignment  
25 that is not defined intrinsically or redefine an intrinsic derived-type assignment (7.4.1.4).

#### NOTE 4.5

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

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

### 27 4.4 Intrinsic types

28 The intrinsic types are:

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

30 The **numeric types** are provided for numerical computation. The normal operations of arithmetic,  
31 addition (+), subtraction (-), multiplication (\*), division (/), exponentiation (\*\*), identity (unary +),

1 and negation (unary  $-$ ), are defined intrinsically for the numeric types.

2 R403 *intrinsic-type-spec*      **is** INTEGER [ *kind-selector* ]  
 3                                      **or** REAL [ *kind-selector* ]  
 4                                      **or** DOUBLE PRECISION  
 5                                      **or** COMPLEX [ *kind-selector* ]  
 6                                      **or** CHARACTER [ *char-selector* ]  
 7                                      **or** LOGICAL [ *kind-selector* ]  
 8 R404 *kind-selector*            **is** ( [ KIND = ] *scalar-int-initialization-expr* )

9 C404 (R404) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a rep-  
 10 resentation method that exists on the processor.

#### 11 4.4.1 Integer type

12 The set of values for the **integer type** is a subset of the mathematical integers. A processor shall  
 13 provide one or more **representation methods** that define sets of values for data of type integer. Each  
 14 such method is characterized by a value for a type parameter called the **kind** type parameter. The kind  
 15 type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.7.59).  
 16 The decimal exponent range of a representation method is returned by the intrinsic function RANGE  
 17 (13.7.96). The intrinsic function SELECTED\_INT\_KIND (13.7.105) returns a kind value based on a  
 18 specified decimal range requirement. The integer type includes a zero value, which is considered neither  
 19 negative nor positive. The value of a signed integer zero is the same as the value of an unsigned integer  
 20 zero.

21 The type specifier for the integer type uses the keyword INTEGER.

22 If the kind type parameter is not specified, the default kind value is KIND (0) and the type specified is  
 23 **default integer**.

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

25 R405 *signed-int-literal-constant*    **is** [ *sign* ] *int-literal-constant*  
 26 R406 *int-literal-constant*            **is** *digit-string* [ *- kind-param* ]  
 27 R407 *kind-param*                      **is** *digit-string*  
 28    **or** *scalar-int-constant-name*  
 29 R408 *signed-digit-string*            **is** [ *sign* ] *digit-string*  
 30 R409 *digit-string*                    **is** *digit* [ *digit* ] ...  
 31 R410 *sign*                              **is** +  
 32    **or** -

33 C405 (R407) A *scalar-int-constant-name* shall be a named constant of type integer.

34 C406 (R407) The value of *kind-param* shall be nonnegative.

35 C407 (R406) The value of *kind-param* shall specify a representation method that exists on the pro-  
 36 cessor.

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

39 An integer constant is interpreted as a decimal value.

#### NOTE 4.6

Examples of signed integer literal constants are:
---

## NOTE 4.6 (cont.)

```

473
+56
-101
21_2
21_SHORT
1976354279568241_8

```

where SHORT is a scalar integer named constant.

- 1 R411 *boz-literal-constant*            **is** *binary-constant*  
2    **or** *octal-constant*  
3    **or** *hex-constant*  
4 R412 *binary-constant*                **is** B ' *digit* [ *digit* ] ... '  
5    **or** B " *digit* [ *digit* ] ... "
- 6 C408 (R412) *digit* shall have one of the values 0 or 1.
- 7 R413 *octal-constant*                **is** O ' *digit* [ *digit* ] ... '  
8    **or** O " *digit* [ *digit* ] ... "
- 9 C409 (R413) *digit* shall have one of the values 0 through 7.
- 10 R414 *hex-constant*                **is** Z ' *hex-digit* [ *hex-digit* ] ... '  
11    **or** Z " *hex-digit* [ *hex-digit* ] ... "
- 12 R415 *hex-digit*                    **is** *digit*  
13    **or** A  
14    **or** B  
15    **or** C  
16    **or** D  
17    **or** E  
18    **or** F

19 Binary, octal and hexadecimal constants are interpreted according to their respective number systems.  
20 The *hex-digits* A through F represent the numbers ten through fifteen, respectively; they may be repre-  
21 sented by their lower-case equivalents.

- 22 C410 (R411) A *boz-literal-constant* shall appear only as a *data-stmt-constant* in a DATA statement, as  
23 the actual argument associated with the dummy argument A of the numeric intrinsic functions  
24 DBLE, REAL or INT, or as the actual argument associated with the X or Y dummy argument  
25 of the intrinsic function CMPLX.

## 26 4.4.2 Real type

27 The **real type** has values that approximate the mathematical real numbers. A processor shall provide  
28 two or more **approximation methods** that define sets of values for data of type real. Each such method  
29 has a **representation method** and is characterized by a value for a type parameter called the **kind** type  
30 parameter. The kind type parameter of an approximation method is returned by the intrinsic inquiry  
31 function KIND (13.7.59). The decimal precision and decimal exponent range of an approximation  
32 method are returned by the intrinsic functions PRECISION (13.7.90) and RANGE (13.7.96). The  
33 intrinsic function SELECTED\_REAL\_KIND (13.7.106) returns a kind value based on specified precision  
34 and decimal range requirements.

**NOTE 4.7**

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

1 The real type includes a zero value. Processors that distinguish between positive and negative zeros  
2 shall treat them as equivalent

- 3 (1) in all relational operations,  
4 (2) as actual arguments to intrinsic procedures other than those for which it is explicitly specified  
5 that negative zero is distinguished, and  
6 (3) as the *scalar-numeric-expr* in an arithmetic IF.

**NOTE 4.8**

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

( X >= 0.0 )

evaluates to true if X = 0.0 or if X = -0.0,

( X < 0.0 )

evaluates to false for X = -0.0, and

IF (X) 1,2,3

causes a transfer of control to the branch target statement with the statement label "2" for both X = 0.0 and X = -0.0.

In order to distinguish between 0.0 and -0.0, a program should use the SIGN function. SIGN(1.0,X) will return -1.0 if X < 0.0 or if the processor distinguishes between 0.0 and -0.0 and X has the value -0.0.

7 The type specifier for the real type uses the keyword REAL. The keyword DOUBLE PRECISION is an  
8 alternate specifier for one kind of real type.

9 If the type keyword REAL is specified and the kind type parameter is not specified, the default kind  
10 value is KIND (0.0) and the type specified is **default real**. If the type keyword DOUBLE PRECISION  
11 is specified, the kind value is KIND (0.0D0) and the type specified is **double precision real**. The  
12 decimal precision of the double precision real approximation method shall be greater than that of the  
13 default real method.

14 R416 *signed-real-literal-constant* is [ *sign* ] *real-literal-constant*  
15 R417 *real-literal-constant* is *significand* [ *exponent-letter exponent* ] [ - *kind-param* ]  
16 or *digit-string exponent-letter exponent* [ - *kind-param* ]  
17 R418 *significand* is *digit-string* . [ *digit-string* ]  
18 or . *digit-string*  
19 R419 *exponent-letter* is E  
20 or D  
21 R420 *exponent* is *signed-digit-string*

22 C411 (R417) If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.

23 C412 (R417) The value of *kind-param* shall specify an approximation method that exists on the  
24 processor.

25 A real literal constant without a kind type parameter is a default real constant if it is without an

- 1 exponent part or has exponent letter E, and is a double precision real constant if it has exponent letter  
 2 D. A real literal constant written with a kind type parameter is a real constant with the specified kind  
 3 type parameter.
- 4 The exponent represents the power of ten scaling to be applied to the significand or digit string. The  
 5 meaning of these constants is as in decimal scientific notation.
- 6 The significand may be written with more digits than a processor will use to approximate the value of  
 7 the constant.

**NOTE 4.9**

Examples of signed real literal constants are:

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

where QUAD is a scalar integer named constant.

### 8 4.4.3 Complex type

- 9 The **complex type** has values that approximate the mathematical complex numbers. The values of a  
 10 complex type are ordered pairs of real values. The first real value is called the **real part**, and the second  
 11 real value is called the **imaginary part**.

12 Each approximation method used to represent data entities of type real shall be available for both the  
 13 real and imaginary parts of a data entity of type complex. A **kind** type parameter may be specified for  
 14 a complex entity and selects for both parts the real approximation method characterized by this kind  
 15 type parameter value. The kind type parameter of an approximation method is returned by the intrinsic  
 16 inquiry function KIND (13.7.59).

17 The type specifier for the complex type uses the keyword COMPLEX. There is no keyword for double  
 18 precision complex. If the type keyword COMPLEX is specified and the kind type parameter is not  
 19 specified, the default kind value is the same as that for default real, the type of both parts is default  
 20 real, and the type specified is **default complex**.

```
21 R421  complex-literal-constant    is  ( real-part , imag-part )
22 R422  real-part                  is  signed-int-literal-constant
23                                           or  signed-real-literal-constant
24                                           or  named-constant
25 R423  imag-part                  is  signed-int-literal-constant
26                                           or  signed-real-literal-constant
27                                           or  named-constant
```

- 28 C413 (R421) Each named constant in a complex literal constant shall be of type integer or real.

29 If the real part and the imaginary part of a complex literal constant are both real, the kind type  
 30 parameter value of the complex literal constant is the kind type parameter value of the part with the  
 31 greater decimal precision; if the precisions are the same, it is the kind type parameter value of one of the  
 32 parts as determined by the processor. If a part has a kind type parameter value different from that of

1 the complex literal constant, the part is converted to the approximation method of the complex literal  
2 constant.

3 If both the real and imaginary parts are integer, they are converted to the default real approximation  
4 method and the constant is of type default complex. If only one of the parts is an integer, it is converted  
5 to the approximation method selected for the part that is real and the kind type parameter value of the  
6 complex literal constant is that of the part that is real.

#### NOTE 4.10

Examples of complex literal constants are:

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

where PI is a previously declared named real constant.

### 7 4.4.4 Character type

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

12 A processor shall provide one or more **representation methods** that define sets of values for data of  
13 type character. Each such method is characterized by a value for a type parameter called the **kind** type  
14 parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry  
15 function KIND (13.7.59). The intrinsic function SELECTED\_CHAR\_KIND (13.7.104) returns a kind  
16 value based on the name of a character type. Any character of a particular representation method  
17 representable in the processor may occur in a character string of that representation method.

18 The character set defined by ISO/IEC 646:1991 (International Reference Version) is referred to as the  
19 **ASCII character set** or the **ASCII character type**. The character set defined by ISO/IEC 10646-  
20 1:2000 UCS-4 is referred to as the **ISO 10646 character set** or the **ISO 10646 character type**.

#### 21 4.4.4.1 Character type specifier

22 The type specifier for the character type uses the keyword CHARACTER.

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

25	R424	<i>char-selector</i>	<b>is</b>	<i>length-selector</i>
26			<b>or</b>	( LEN = <i>type-param-value</i> , ■
27				■ KIND = <i>scalar-int-initialization-expr</i> )
28			<b>or</b>	( <i>type-param-value</i> , ■
29				■ [ KIND = ] <i>scalar-int-initialization-expr</i> )
30			<b>or</b>	( KIND = <i>scalar-int-initialization-expr</i> ■
31				■ [ , LEN = <i>type-param-value</i> ] )
32	R425	<i>length-selector</i>	<b>is</b>	( [ LEN = ] <i>type-param-value</i> )
33			<b>or</b>	* <i>char-length</i> [ , ]
34	R426	<i>char-length</i>	<b>is</b>	( <i>type-param-value</i> )

1

or *scalar-int-literal-constant*

2

C414 (R424) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.

3

4

C415 (R426) The *scalar-int-literal-constant* shall not include a *kind-param*.

5

C416 (R424 R425 R426) A *type-param-value* of \* may be used only in the following ways:

6

(1) to declare a dummy argument,

7

(2) to declare a named constant,

8

(3) in the *type-spec* of an ALLOCATE statement wherein each *allocate-object* is a dummy argument of type CHARACTER with an assumed character length, or

9

(4) in an external function, to declare the character length parameter of the function result.

10

C417 A function name shall not be declared with an asterisk *type-param-value* unless it is of type CHARACTER and is the name of the result of an external function or the name of a dummy function.

11

12

C418 A function name declared with an asterisk *type-param-value* shall not be an array, a pointer, recursive, or pure.

13

C419 (R425) The optional comma in a *length-selector* is permitted only in a *declaration-type-spec* in a *type-declaration-stmt*.

14

15

C420 (R425) The optional comma in a *length-selector* is permitted only if no double-colon separator appears in the *type-declaration-stmt*.

16

17

C421 (R424) The length specified for a character statement function or for a statement function dummy argument of type character shall be an initialization expression.

18

19

The *char-selector* in a CHARACTER *intrinsic-type-spec* and the \* *char-length* in an *entity-decl* or in a *component-decl* of a type definition specify character length. The \* *char-length* in an *entity-decl* or a *component-decl* specifies an individual length and overrides the length specified in the *char-selector*, if any. If a \* *char-length* is not specified in an *entity-decl* or a *component-decl*, the *length-selector* or *type-param-value* specified in the *char-selector* is the character length. If the length is not specified in a *char-selector* or a \* *char-length*, the length is 1.

20

21

If the character length parameter value evaluates to a negative value, the length of character entities declared is zero. A character length parameter value of : indicates a deferred type parameter (4.2). A *char-length* type parameter value of \* has the following meaning:

22

23

24

(1) If used to declare a dummy argument of a procedure, the dummy argument assumes the length of the associated actual argument.

25

26

(2) If used to declare a named constant, the length is that of the constant value.

27

(3) If used in the *type-spec* of an ALLOCATE statement, each *allocate-object* assumes its length from the associated actual argument.

28

29

(4) If used to specify the character length parameter of a function result, any scoping unit invoking the function shall declare the function name with a character length parameter value other than \* or access such a definition by host or use association. When the function is invoked, the length of the result variable in the function is assumed from the value of this type parameter.

30

31

32

33

#### 4.4.4.2 Character literal constant

34

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

35

1 R427 *char-literal-constant*      **is** [ *kind-param* - ] ' [ *rep-char* ] ... '  
 2    **or** [ *kind-param* - ] " [ *rep-char* ] ... "

3 C422 (R427) The value of *kind-param* shall specify a representation method that exists on the pro-  
 4 cessor.

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

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

- 9       (1) In free source form, it is any graphic character in the processor-dependent character set.
- 10       (2) In fixed source form, it is any character in the processor-dependent character set. A processor may restrict  
 11 the occurrence of some or all of the control characters.

#### NOTE 4.11

FORTRAN 77 allowed any character to occur in a character context. This standard allows a source program to contain characters of more than one kind. Some processors may identify characters of nondefault kinds by control characters (called "escape" or "shift" characters). It is difficult, if not impossible, to process, edit, and print files where some instances of control characters have their intended meaning and some instances might not. Almost all control characters have uses or effects that effectively preclude their use in character contexts and this is why free source form allows only graphic characters as representable characters. Nevertheless, for compatibility with FORTRAN 77, control characters remain permitted in principle in fixed source form.

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

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

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

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

#### NOTE 4.12

Examples of character literal constants are:

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

both of which have the value DON'T and

```
''
```

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

**NOTE 4.13**

An example of a nondefault character literal constant, where the processor supports the corresponding character set, is:

```
NIHONGO_’彼女なしでは何もできない。’
```

where NIHONGO is a named constant whose value is the kind type parameter for Nihongo (Japanese) characters.

1 **4.4.4.3 Collating sequence**

2 Each implementation defines a collating sequence for the character set of each kind of character. A  
3 **collating sequence** is a one-to-one mapping of the characters into the nonnegative integers such that  
4 each character corresponds to a different nonnegative integer. The intrinsic functions CHAR (13.7.19)  
5 and ICHAR (13.7.50) provide conversions between the characters and the integers according to this  
6 mapping.

**NOTE 4.14**

For example:

```
ICHAR ( 'X' )
```

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

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

- 8 (1) ICHAR ('A') < ICHAR ('B') < ... < ICHAR ('Z') for the twenty-six upper-case letters.  
9 (2) ICHAR ('0') < ICHAR ('1') < ... < ICHAR ('9') for the ten digits.  
10 (3) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('A') or  
11 ICHAR (' ') < ICHAR ('A') < ICHAR ('Z') < ICHAR ('0').  
12 (4) ICHAR ('a') < ICHAR ('b') < ... < ICHAR ('z') for the twenty-six lower-case letters.  
13 (5) ICHAR (' ') < ICHAR ('0') < ICHAR ('9') < ICHAR ('a') or  
14 ICHAR (' ') < ICHAR ('a') < ICHAR ('z') < ICHAR ('0').

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

18 The collating sequence for the ASCII character type is as defined by ISO/IEC 646:1991 (International  
19 Reference Version); this collating sequence is called the **ASCII collating sequence** in this standard.  
20 The collating sequence for the ISO 10646 character type is as defined by ISO/IEC 10646-1:2000.

**NOTE 4.15**

The intrinsic functions ACHAR (13.7.2) and IACHAR (13.7.45) provide conversion between characters and corresponding integer values according to the ASCII collating sequence.

21 The intrinsic functions LGT, LGE, LLE, and LLT (13.7.63-13.7.66) provide comparisons between strings  
22 based on the ASCII collating sequence. International portability is guaranteed if the set of characters  
23 used is limited to the letters, digits, underscore, and special characters.

24 **4.4.5 Logical type**

25 The **logical type** has two values, which represent true and false.

1 A processor shall provide one or more **representation methods** for data of type logical. Each such  
2 method is characterized by a value for a type parameter called the **kind** type parameter. The kind type  
3 parameter of a representation method is returned by the intrinsic inquiry function KIND (13.7.59).

4 The type specifier for the logical type uses the keyword LOGICAL.

5 If the kind type parameter is not specified, the default kind value is KIND (.FALSE.) and the type  
6 specified is **default logical**.

7 R428 *logical-literal-constant* **is** .TRUE. [ *- kind-param* ]  
8 **or** .FALSE. [ *- kind-param* ]

9 C423 (R428) The value of *kind-param* shall specify a representation method that exists on the pro-  
10 cessor.

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

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

## 18 4.5 Derived types

19 Additional types may be derived from the intrinsic types and other derived types. A type definition is  
20 required to define the name of the type and the names and attributes of its components and type-bound  
21 procedures.

22 A derived type may be parameterized by multiple type parameters, each of which is defined to be either  
23 a kind or length type parameter and may have a default value.

24 The **ultimate components** of an object of derived type are the components that are of intrinsic type  
25 or have the POINTER or ALLOCATABLE attribute, plus the ultimate components of the components  
26 of the object that are of derived type and have neither the ALLOCATABLE nor POINTER attribute.

### NOTE 4.16

The ultimate components of objects of the derived type `kids` defined below are `name`, `age`, and `other_kids`.

```
type :: person
  character(len=20) :: name
  integer :: age
end type person

type :: kids
  type(person) :: oldest_child
  type(person), allocatable, dimension(:) :: other_kids
end type kids
```

27 By default, no storage sequence is implied by the order of the component definitions. However, a storage  
28 order is implied for a sequence type (4.5.1.2). If the derived type has the BIND attribute, the storage  
29 sequence is that required by the companion processor (2.5.10, 15.2.3).

1 **4.5.1 Derived-type definition**

- 2 R429 *derived-type-def*           **is** *derived-type-stmt*  
3    [ *type-param-def-stmt* ] ...  
4    [ *private-or-sequence* ] ...  
5    [ *component-part* ]  
6    [ *type-bound-procedure-part* ]  
7    *end-type-stmt*
- 8 R430 *derived-type-stmt*           **is** TYPE [ [ , *type-attr-spec-list* ] :: ] *type-name* ■  
9    ■ [ ( *type-param-name-list* ) ]
- 10 R431 *type-attr-spec*           **is** *access-spec*  
11    **or** EXTENDS ( *parent-type-name* )  
12    **or** ABSTRACT  
13    **or** BIND (C)
- 14 C424 (R430) A derived type *type-name* shall not be DOUBLEPRECISION or the same as the name  
15 of any intrinsic type defined in this standard.
- 16 C425 (R430) The same *type-attr-spec* shall not appear more than once in a given *derived-type-stmt*.
- 17 C426 (R431) A *parent-type-name* shall be the name of an accessible extensible type (4.5.6).
- 18 C427 (R429) If the type definition contains or inherits (4.5.6.1) a deferred binding (4.5.4), ABSTRACT  
19 shall appear.
- 20 C428 (R429) If ABSTRACT appears, the type shall be extensible.
- 21 C429 (R429) If EXTENDS appears, SEQUENCE shall not appear.
- 22 R432 *private-or-sequence*           **is** *private-components-stmt*  
23    **or** *sequence-stmt*
- 24 C430 (R429) The same *private-or-sequence* shall not appear more than once in a given *derived-type-*  
25 *def*.
- 26 R433 *end-type-stmt*               **is** END TYPE [ *type-name* ]
- 27 C431 (R433) If END TYPE is followed by a *type-name*, the *type-name* shall be the same as that in  
28 the corresponding *derived-type-stmt*.
- 29 Derived types with the BIND attribute are subject to additional constraints as specified in 15.2.3.

**NOTE 4.17**

An example of a derived-type definition is:

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

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

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

1 **4.5.1.1 Accessibility**

2 Types that are defined in a module or accessible in that module by use association have either the  
 3 PUBLIC or PRIVATE attribute. Types for which an *access-spec* is not explicitly specified in that  
 4 module have the default accessibility attribute for that module. The default accessibility attribute for a  
 5 module is PUBLIC unless it has been changed by a PRIVATE statement (5.2.1). Only types that have  
 6 the PUBLIC attribute in that module are available to be accessed from that module by use association.

7 The accessibility of a type does not affect, and is not affected by, the accessibility of its components and  
 8 bindings.

9 If a type definition is private, then the type name, and thus the structure constructor (4.5.9) for the  
 10 type, are accessible only within the module containing the definition.

**NOTE 4.18**

An example of a type with a private name is:

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

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

11 **4.5.1.2 Sequence type**

12 R434 *sequence-stmt* is SEQUENCE

13 C432 (R438) If SEQUENCE appears, all derived types specified in component definitions shall be  
 14 sequence types.

15 C433 (R429) If SEQUENCE appears, a *type-bound-procedure-part* shall not appear.

16 If the **SEQUENCE statement** appears, the type is a **sequence type**. The order of the component  
 17 definitions in a sequence type specifies a storage sequence for objects of that type. If there are no type  
 18 parameters and all of the ultimate components of objects of the type are of type default integer, default  
 19 real, double precision real, default complex, or default logical and are not pointers or allocatable, the  
 20 type is a **numeric sequence type**. If there are no type parameters and all of the ultimate components  
 21 of objects of the type are of type default character and are not pointers or allocatable, the type is a  
 22 **character sequence type**.

**NOTE 4.19**

An example of a numeric sequence type is:

```
TYPE NUMERIC_SEQ
  SEQUENCE
  INTEGER :: INT_VAL
  REAL    :: REAL_VAL
  LOGICAL :: LOG_VAL
END TYPE NUMERIC_SEQ
```

**NOTE 4.20**

A structure resolves into a sequence of components. Unless the structure includes a SEQUENCE statement, the use of this terminology in no way implies that these components are stored in

**NOTE 4.20 (cont.)**

this, or any other, order. Nor is there any requirement that contiguous storage be used. The sequence merely refers to the fact that in writing the definitions there will necessarily be an order in which the components appear, and this will define a sequence of components. This order is of limited significance because a component of an object of derived type will always be accessed by a component name except in the following contexts: the sequence of expressions in a derived-type value constructor, intrinsic assignment, the data values in namelist input data, and the inclusion of the structure in an input/output list of a formatted data transfer, where it is expanded to this sequence of components. Provided the processor adheres to the defined order in these cases, it is otherwise free to organize the storage of the components for any nonsequence structure in memory as best suited to the particular architecture.

**1 4.5.1.3 Determination of derived types**

- 2 Derived-type definitions with the same type name may appear in different scoping units, in which case  
3 they may be independent and describe different derived types or they may describe the same type.
- 4 Two data entities have the same type if they are declared with reference to the same derived-type  
5 definition. The definition may be accessed from a module or from a host scoping unit. Data entities in  
6 different scoping units also have the same type if they are declared with reference to different derived-type  
7 definitions that specify the same type name, all have the SEQUENCE property or all have the BIND  
8 attribute, have no components with PRIVATE accessibility, and have type parameters and components  
9 that agree in order, name, and attributes. Otherwise, they are of different derived types. A data entity  
10 declared using a type with the SEQUENCE property or with the BIND attribute is not of the same type  
11 as an entity of a type declared to be PRIVATE or that has any components that are PRIVATE.

**NOTE 4.21**

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

```

TYPE POINT
  REAL X, Y
END TYPE POINT
TYPE (POINT) :: X1
CALL SUB (X1)
...
CONTAINS
  SUBROUTINE SUB (A)
    TYPE (POINT) :: A
    ...
  END SUBROUTINE SUB

```

The definition of derived type POINT is known in subroutine SUB by host association. Because the declarations of X1 and A both reference the same derived-type definition, X1 and A have the same type. X1 and A also would have the same type if the derived-type definition were in a module and both SUB and its containing program unit accessed the module.

**NOTE 4.22**

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

```

PROGRAM PGM
  TYPE EMPLOYEE
  SEQUENCE

```

## NOTE 4.22 (cont.)

```

        INTEGER      ID_NUMBER
        CHARACTER (50) NAME
    END TYPE EMPLOYEE
    TYPE (EMPLOYEE) PROGRAMMER
    CALL SUB (PROGRAMMER)
        ...
END PROGRAM PGM
SUBROUTINE SUB (POSITION)
    TYPE EMPLOYEE
        SEQUENCE
        INTEGER      ID_NUMBER
        CHARACTER (50) NAME
    END TYPE EMPLOYEE
    TYPE (EMPLOYEE) POSITION
        ...
END SUBROUTINE SUB

```

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

Suppose the component name ID\_NUMBER was ID\_NUM in the subroutine. Because all the component names are not identical to the component names in derived type EMPLOYEE in the main program, the actual argument PROGRAMMER would not be of the same type as the dummy argument POSITION. Thus, the program would not be standard conforming.

## NOTE 4.23

The requirement that the two types have the same name applies to the *type-names* of the respective *derived-type-stmts*, not to local names introduced via renaming in USE statements.

## 1 4.5.2 Derived-type parameters

2 R435 *type-param-def-stmt* is INTEGER [ *kind-selector* ], *type-param-attr-spec* :: ■

3 ■ *type-param-decl-list*

4 R436 *type-param-decl* is *type-param-name* [ = *scalar-int-initialization-expr* ]

5 C434 (R435) A *type-param-name* in a *type-param-def-stmt* in a *derived-type-def* shall be one of the  
6 *type-param-names* in the *derived-type-stmt* of that *derived-type-def*.

7 C435 (R435) Each *type-param-name* in the *derived-type-stmt* in a *derived-type-def* shall appear as a  
8 *type-param-name* in a *type-param-def-stmt* in that *derived-type-def*.

9 R437 *type-param-attr-spec* is KIND  
10 or LEN

11 The derived type is parameterized if the *derived-type-stmt* has any *type-param-names*.

12 Each type parameter is itself of type integer. If its kind selector is omitted, the kind type parameter is  
13 default integer.

14 The *type-param-attr-spec* explicitly specifies whether a type parameter is a kind parameter or a length  
15 parameter.

- 1 If a *type-param-decl* has a *scalar-int-initialization-expr*, the type parameter has a default value which  
 2 is specified by the expression. If necessary, the value is converted according to the rules of intrinsic  
 3 assignment (7.4.1.3) to a value of the same kind as the type parameter.
- 4 A type parameter may be used as a primary in a specification expression (7.1.6) in the *derived-type-*  
 5 *def*. A kind type parameter may also be used as a primary in an initialization expression (7.1.7) in the  
 6 *derived-type-def*.

**NOTE 4.24**

The following example uses derived-type parameters.

```
TYPE humongous_matrix(k, d)
  INTEGER, KIND :: k = kind(0.0)
  INTEGER(selected_int_kind(12)), LEN :: d
  !-- Specify a nondefault kind for d.
  REAL(k) :: element(d,d)
END TYPE
```

In the following example, *dim* is declared to be a kind parameter, allowing generic overloading of  
 procedures distinguished only by *dim*.

```
TYPE general_point(dim)
  INTEGER, KIND :: dim
  REAL :: coordinates(dim)
END TYPE
```

7 **4.5.2.1 Type parameter order**

- 8 **Type parameter order** is an ordering of the type parameters of a derived type; it is used for derived-  
 9 type specifiers.

- 10 The type parameter order of a nonextended type is the order of the type parameter list in the derived-  
 11 type definition. The type parameter order of an extended type consists of the type parameter order of  
 12 its parent type followed by any additional type parameters in the order of the type parameter list in the  
 13 derived-type definition.

**NOTE 4.25**

Given

```
TYPE :: t1(k1,k2)
  INTEGER,KIND :: k1,k2
  REAL(k1) a(k2)
END TYPE
TYPE,EXTENDS(t1) :: t2(k3)
  INTEGER,KIND :: k3
  LOGICAL(k3) flag
END TYPE
```

the type parameter order for type T1 is K1 then K2, and the type parameter order for type T2 is  
 K1 then K2 then K3.

14 **4.5.3 Components**

- 15 R438 *component-part* is [ *component-def-stmt* ] ...

- 1 R439 *component-def-stmt* is *data-component-def-stmt*  
 2 or *proc-component-def-stmt*
- 3 R440 *data-component-def-stmt* is *declaration-type-spec* [ [ , *component-attr-spec-list* ] :: ] ■  
 4 ■ *component-decl-list*
- 5 R441 *component-attr-spec* is POINTER  
 6 or DIMENSION ( *component-array-spec* )  
 7 or ALLOCATABLE  
 8 or *access-spec*
- 9 R442 *component-decl* is *component-name* [ ( *component-array-spec* ) ] ■  
 10 ■ [ \* *char-length* ] [ *component-initialization* ]
- 11 R443 *component-array-spec* is *explicit-shape-spec-list*  
 12 or *deferred-shape-spec-list*
- 13 R444 *component-initialization* is = *initialization-expr*  
 14 or => *null-init*
- 15 C436 (R440) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.
- 16 C437 (R440) A component declared with the CLASS keyword (5.1.1.2) shall have the ALLOCATABLE  
 17 or POINTER attribute.
- 18 C438 (R440) If the POINTER attribute is not specified for a component, the *declaration-type-spec* in  
 19 the *component-def-stmt* shall specify an intrinsic type or a previously defined derived type.
- 20 C439 (R440) If the POINTER attribute is specified for a component, the *declaration-type-spec* in the  
 21 *component-def-stmt* shall specify an intrinsic type or any accessible derived type including the  
 22 type being defined.
- 23 C440 (R440) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec*  
 24 shall be a *deferred-shape-spec-list*.
- 25 C441 (R440) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each  
 26 *component-array-spec* shall be an *explicit-shape-spec-list*.
- 27 C442 (R443) Each bound in the *explicit-shape-spec* shall either be an initialization expression or be a  
 28 specification expression that does not contain references to specification functions or any object  
 29 designators other than named constants or subobjects thereof.
- 30 C443 (R440) A component shall not have both the ALLOCATABLE and the POINTER attribute.
- 31 C444 (R442) The \* *char-length* option is permitted only if the type specified is character.
- 32 C445 (R439) Each *type-param-value* within a *component-def-stmt* shall either be a colon, be an ini-  
 33 tialization expression, or be a specification expression that contains neither references to speci-  
 34 fication functions nor any object designators other than named constants or subobjects thereof.

**NOTE 4.26**

Because a type parameter is not an object, a *type-param-value* or a bound in an *explicit-shape-spec* may contain a *type-param-name*.

- 35 C446 (R440) If *component-initialization* appears, a double-colon separator shall appear before the  
 36 *component-decl-list*.
- 37 C447 (R440) If => appears in *component-initialization*, POINTER shall appear in the *component-*  
 38 *attr-spec-list*. If = appears in *component-initialization*, POINTER or ALLOCATABLE shall  
 39 not appear in the *component-attr-spec-list*.

- 1 R445 *proc-component-def-stmt* is PROCEDURE ( [ *proc-interface* ] ) , ■  
 2 ■ *proc-component-attr-spec-list* :: *proc-decl-list*

**NOTE 4.27**

See 12.3.2.3 for definitions of *proc-interface* and *proc-decl*.

- 3 R446 *proc-component-attr-spec* is POINTER  
 4 or PASS [ (*arg-name*) ]  
 5 or NOPASS  
 6 or *access-spec*
- 7 C448 (R445) The same *proc-component-attr-spec* shall not appear more than once in a given *proc-*  
 8 *component-def-stmt*.
- 9 C449 (R445) POINTER shall appear in each *proc-component-attr-spec-list*.
- 10 C450 (R445) If the procedure pointer component has an implicit interface or has no arguments,  
 11 NOPASS shall be specified.
- 12 C451 (R445) If PASS (*arg-name*) appears, the interface shall have a dummy argument named *arg-*  
 13 *name*.
- 14 C452 (R445) PASS and NOPASS shall not both appear in the same *proc-component-attr-spec-list*.

**15 4.5.3.1 Array components**

- 16 A data component is an array if its *component-decl* contains a *component-array-spec* or its *data-compo-*  
 17 *nent-def-stmt* contains the DIMENSION attribute. If the *component-decl* contains a *component-array-*  
 18 *spec*, it specifies the array rank, and if the array is explicit shape (5.1.2.5.1), the array bounds; otherwise,  
 19 the *component-array-spec* in the DIMENSION attribute specifies the array rank, and if the array is  
 20 explicit shape, the array bounds.

**NOTE 4.28**

An example of a derived type definition with an array component is:

```
TYPE LINE
  REAL, DIMENSION (2, 2) :: COORD      !
                                     ! COORD(:,1) has the value of (/X1, Y1/)
                                     ! COORD(:,2) has the value of (/X2, Y2/)
  REAL                          :: WIDTH ! Line width in centimeters
  INTEGER                       :: PATTERN ! 1 for solid, 2 for dash, 3 for dot
END TYPE LINE
```

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

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

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

**NOTE 4.29**

An example of a derived type definition with an allocatable component is:

**NOTE 4.29 (cont.)**

```

TYPE STACK
  INTEGER                :: INDEX
  INTEGER, ALLOCATABLE :: CONTENTS (:)
END TYPE STACK

```

For each scalar variable of type STACK, the shape of the component CONTENTS is determined by execution of an ALLOCATE statement or assignment statement, or by argument association.

**NOTE 4.30**

Default initialization of an explicit-shape array component may be specified by an initialization expression consisting of an array constructor (4.7), or of a single scalar that becomes the value of each array element.

**1 4.5.3.2 Pointer components**

- 2 A component is a pointer (2.4.6) if its *component-attr-spec-list* contains the POINTER attribute. A  
 3 pointer component may be a data pointer or a procedure pointer.

**NOTE 4.31**

An example of a derived type definition with a pointer component is:

```

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

```

Any object of type REFERENCE will have the four nonpointer components VOLUME, YEAR, PAGE, and TITLE, plus a pointer to an array of characters holding SYNOPSIS. The size of this target array will be determined by the length of the abstract. The space for the target may be allocated (6.3.1) or the pointer component may be associated with a target by a pointer assignment statement (7.4.2).

**4 4.5.3.3 The passed-object dummy argument**

- 5 A **passed-object dummy argument** is a distinguished dummy argument of a procedure pointer  
 6 component or type-bound procedure. It affects procedure overriding (4.5.6.2) and argument association  
 7 (12.4.1.1).
- 8 If NOPASS is specified, the procedure pointer component or type-bound procedure has no passed-object  
 9 dummy argument.
- 10 If neither PASS nor NOPASS is specified or PASS is specified without *arg-name*, the first dummy argu-  
 11 ment of a procedure pointer component or type-bound procedure is its passed-object dummy argument.
- 12 If PASS (*arg-name*) is specified, the dummy argument named *arg-name* is the passed-object dummy  
 13 argument of the procedure pointer component or named type-bound procedure.
- 14 C453 The passed-object dummy argument shall be a scalar, nonpointer, nonallocatable dummy data  
 15 object with the same declared type as the type being defined; all of its length type parameters  
 16 shall be assumed; it shall be polymorphic (5.1.1.2) if and only if the type being defined is

1 extensible (4.5.6).

**NOTE 4.32**

If a procedure is bound to several types as a type-bound procedure, different dummy arguments might be the passed-object dummy argument in different contexts.

2 **4.5.3.4 Default initialization for components**

3 Default initialization provides a means of automatically initializing pointer components to be disasso-  
4 ciated, and nonpointer nonallocatable components to have a particular value. Allocatable components  
5 are always initialized to not allocated.

6 If *null-init* appears for a pointer component, that component in any object of the type has an initial  
7 association status of disassociated (16.4.2.1) or becomes disassociated as specified in 16.4.2.1.2.

8 If *initialization-expr* appears for a nonpointer component, that component in any object of the type  
9 is initially defined (16.5.3) or becomes defined as specified in 16.5.5 with the value determined from  
10 *initialization-expr*. If necessary, the value is converted according to the rules of intrinsic assignment  
11 (7.4.1.3) to a value that agrees in type, type parameters, and shape with the component. If the compo-  
12 nent is of a type for which default initialization is specified for a component, the default initialization  
13 specified by *initialization-expr* overrides the default initialization specified for that component. When  
14 one initialization **overrides** another it is as if only the overriding initialization were specified (see Note  
15 4.34). Explicit initialization in a type declaration statement (5.1) overrides default initialization (see  
16 Note 4.33). Unlike explicit initialization, default initialization does not imply that the object has the  
17 SAVE attribute.

18 A subcomponent (6.1.2) is **default-initialized** if the type of the object of which it is a component  
19 specifies default initialization for that component, and the subcomponent is not a subobject of an object  
20 that is default-initialized or explicitly initialized.

**NOTE 4.33**

It is not required that initialization be specified for each component of a derived type. For example:

```
TYPE DATE
  INTEGER DAY
  CHARACTER (LEN = 5) MONTH
  INTEGER :: YEAR = 1994      ! Partial default initialization
END TYPE DATE
```

In the following example, the default initial value for the YEAR component of TODAY is overridden by explicit initialization in the type declaration statement:

```
TYPE (DATE), PARAMETER :: TODAY = DATE (21, "Feb.", 1995)
```

**NOTE 4.34**

The default initial value of a component of derived type may be overridden by default initialization specified in the definition of the type. Continuing the example of Note 4.33:

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

**NOTE 4.34 (cont.)**

```
END TYPE SINGLE_SCORE
TYPE(SINGLE_SCORE) SETUP
```

The PLAY\_DAY component of SETUP receives its initial value from TODAY, overriding the initialization for the YEAR component.

**NOTE 4.35**

Arrays of structures may be declared with elements that are partially or totally initialized by default. Continuing the example of Note 4.34 :

```
TYPE MEMBER (NAME_LEN)
  INTEGER, LEN :: NAME_LEN
  CHARACTER (LEN = NAME_LEN) NAME = ''
  INTEGER :: TEAM_NO, HANDICAP = 0
  TYPE (SINGLE_SCORE), POINTER :: HISTORY => NULL ( )
END TYPE MEMBER
TYPE (MEMBER(9)) LEAGUE (36)      ! Array of partially initialized elements
TYPE (MEMBER(9)) :: ORGANIZER = MEMBER ("I. Manage",1,5,NULL ( ))
```

ORGANIZER is explicitly initialized, overriding the default initialization for an object of type MEMBER.

Allocated objects may also be initialized partially or totally. For example:

```
ALLOCATE (ORGANIZER % HISTORY)   ! A partially initialized object of type
                                ! SINGLE_SCORE is created.
```

**NOTE 4.36**

A pointer component of a derived type may have as its target an object of that derived type. The type definition may specify that in objects declared to be of this type, such a pointer is default initialized to disassociated. For example:

```
TYPE NODE
  INTEGER          :: VALUE = 0
  TYPE (NODE), POINTER :: NEXT_NODE => NULL ( )
END TYPE
```

A type such as this may be used to construct linked lists of objects of type NODE. See C.1.5 for an example.

**1 4.5.3.5 Component order**

2 **Component order** is an ordering of the nonparent components of a derived type; it is used for intrinsic  
3 formatted input/output and structure constructors (where component keywords are not used). Parent  
4 components are excluded from the component order of an extended type (4.5.6).

5 The component order of a nonextended type is the order of the declarations of the components in the  
6 derived-type definition. The component order of an extended type consists of the component order of  
7 its parent type followed by any additional components in the order of their declarations in the extended  
8 derived-type definition.

**NOTE 4.37**

Given the same type definitions as in Note 4.5.2.1, the component order of type T1 is just A (there is only one component), and the component order of type T2 is A then FLAG. The parent component (T1) does not participate in the component order.

1 **4.5.3.6 Component accessibility**2 R447 *private-components-stmt* is PRIVATE3 C454 (R447) A *private-components-stmt* is permitted only if the type definition is within the speci-  
4 cation part of a module.5 The default accessibility for the components that are declared in a type's *component-part* is private  
6 if the type definition contains a *private-components-stmt*, and public otherwise. The accessibility of a  
7 component may be explicitly declared by an *access-spec*; otherwise its accessibility is the default for the  
8 type definition in which it is declared.9 If a component is private, that component name is accessible only within the module containing the  
10 definition.**NOTE 4.38**

Type parameters are not components. They are effectively always public.

**NOTE 4.39**

The accessibility of the components of a type is independent of the accessibility of the type name. It is possible to have all four combinations: a public type name with a public component, a private type name with a private component, a public type name with a private component, and a private type name with a public component.

**NOTE 4.40**

An example of a type with private components is:

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

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

**NOTE 4.41**

The following example illustrates the use of an individual component *access-spec* to override the default accessibility:

```
TYPE MIXED
  PRIVATE
  INTEGER :: I
  INTEGER, PUBLIC :: J
END TYPE MIXED
```

## NOTE 4.41 (cont.)

TYPE (MIXED) :: M

The component M%J is accessible in any scoping unit where M is accessible; M%I is accessible only within the module containing the TYPE MIXED definition.

1 **4.5.4 Type-bound procedures**

2 R448 *type-bound-procedure-part* is *contains-stmt*  
 3 [ *binding-private-stmt* ]  
 4 *proc-binding-stmt*  
 5 [ *proc-binding-stmt* ] ...

6 R449 *binding-private-stmt* is PRIVATE

7 C455 (R448) A *binding-private-stmt* is permitted only if the type definition is within the specification  
 8 part of a module.

9 R450 *proc-binding-stmt* is *specific-binding*  
 10 or *generic-binding*  
 11 or *final-binding*

12 R451 *specific-binding* is PROCEDURE [ (*interface-name*) ] ■  
 13 ■ [ [ , *binding-attr-list* ] :: ] ■  
 14 ■ *binding-name* [= > *procedure-name* ]

15 C456 (R451) If => *procedure-name* appears, the double-colon separator shall appear.

16 C457 (R451) If => *procedure-name* appears, *interface-name* shall not appear.

17 C458 (R451) The *procedure-name* shall be the name of an accessible module procedure or an external  
 18 procedure that has an explicit interface.

19 If neither => *procedure-name* nor *interface-name* appears, it is as though => *procedure-name* had  
 20 appeared with a procedure name the same as the binding name.

21 R452 *generic-binding* is GENERIC ■  
 22 ■ [ , *access-spec* ] :: *generic-spec* => *binding-name-list*

23 C459 (R452) Within the *specification-part* of a module, each *generic-binding* shall specify, either  
 24 implicitly or explicitly, the same accessibility as every other *generic-binding* with that *generic-*  
 25 *spec* in the same derived type.

26 C460 (R452) Each *binding-name* in *binding-name-list* shall be the name of a specific binding of the  
 27 type.

28 C461 (R452) If *generic-spec* is not *generic-name*, each of its specific bindings shall have a passed-object  
 29 dummy argument (4.5.3.3).

30 C462 (R452) If *generic-spec* is OPERATOR ( *defined-operator* ), the interface of each binding shall  
 31 be as specified in 12.3.2.1.1.

32 C463 (R452) If *generic-spec* is ASSIGNMENT ( = ), the interface of each binding shall be as specified  
 33 in 12.3.2.1.2.

34 C464 (R452) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in  
 35 9.5.3.7. The type of the *dtv* argument shall be *type-name*.

- 1 R453 *binding-attr* is PASS [ (*arg-name*) ]  
 2 or NOPASS  
 3 or NON\_OVERRIDABLE  
 4 or DEFERRED  
 5 or *access-spec*
- 6 C465 (R453) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.
- 7 C466 (R451) If the interface of the binding has no dummy argument of the type being defined,  
 8 NOPASS shall appear.
- 9 C467 (R451) If PASS (*arg-name*) appears, the interface of the binding shall have a dummy argument  
 10 named *arg-name*.
- 11 C468 (R453) PASS and NOPASS shall not both appear in the same *binding-attr-list*.
- 12 C469 (R453) NON\_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-*  
 13 *attr-list*.
- 14 C470 (R453) DEFERRED shall appear if and only if *interface-name* appears.
- 15 C471 (R451) An overriding binding (4.5.6.2) shall have the DEFERRED attribute only if the binding  
 16 it overrides is deferred.
- 17 C472 (R451) A binding shall not override an inherited binding (4.5.6.1) that has the NON-OVER-  
 18 RIDABLE attribute.
- 19 Each binding in a *proc-binding-stmt* specifies a **type-bound procedure**. A type-bound procedure  
 20 may have a passed-object dummy argument (4.5.3.3). A *generic-binding* specifies a type-bound generic  
 21 interface for its specific bindings. A binding that specifies the DEFERRED attribute is a **deferred**  
 22 **binding**. A deferred binding shall appear only in the definition of an abstract type.
- 23 A type-bound procedure may be identified by a **binding name** in the scope of the type definition.  
 24 This name is the *binding-name* for a specific binding, and the *generic-name* for a generic binding whose  
 25 *generic-spec* is *generic-name*. A final binding, or a generic binding whose *generic-spec* is not *generic-*  
 26 *name*, has no binding name.
- 27 The interface of a specific binding is that of the procedure specified by *procedure-name* or the interface  
 28 specified by *interface-name*.

**NOTE 4.42**

An example of a type and a type-bound procedure is:

```

TYPE POINT
  REAL :: X, Y
CONTAINS
  PROCEDURE, PASS :: LENGTH => POINT_LENGTH
END TYPE POINT
...

```

and in the *module-subprogram-part* of the same module:

```

REAL FUNCTION POINT_LENGTH (A, B)
  CLASS (POINT), INTENT (IN) :: A, B
  POINT_LENGTH = SQRT ( (A%X - B%X)**2 + (A%Y - B%Y)**2 )
END FUNCTION POINT_LENGTH

```

- 1 The same *generic-spec* may be used in several *generic-bindings* within a single derived-type definition.  
 2 Each additional *generic-binding* with the same *generic-spec* extends the generic interface.

**NOTE 4.43**

Unlike the situation with generic procedure names, a generic type-bound procedure name is not permitted to be the same as a specific type-bound procedure name in the same type (16.2).

- 3 The default accessibility for the procedure bindings of a type is private if the type definition contains a  
 4 *binding-private-stmt*, and public otherwise. The accessibility of a procedure binding may be explicitly  
 5 declared by an *access-spec*; otherwise its accessibility is the default for the type definition in which it is  
 6 declared.
- 7 A public type-bound procedure is accessible via any accessible object of the type. A private type-bound  
 8 procedure is accessible only within the module containing the type definition.

**NOTE 4.44**

The accessibility of a type-bound procedure is not affected by a PRIVATE statement in the *component-part*; the accessibility of a data component is not affected by a PRIVATE statement in the *type-bound-procedure-part*.

### 9 4.5.5 Final subroutines

- 10 R454 *final-binding* is FINAL [ :: ] *final-subroutine-name-list*
- 11 C473 (R454) A *final-subroutine-name* shall be the name of a module procedure with exactly one  
 12 dummy argument. That argument shall be nonoptional and shall be a nonpointer, nonallocat-  
 13 able, nonpolymorphic variable of the derived type being defined. All length type parameters of  
 14 the dummy argument shall be assumed. The dummy argument shall not be INTENT(OUT).
- 15 C474 (R454) A *final-subroutine-name* shall not be one previously specified as a final subroutine for  
 16 that type.
- 17 C475 (R454) A final subroutine shall not have a dummy argument with the same kind type parameters  
 18 and rank as the dummy argument of another final subroutine of that type.
- 19 The FINAL keyword specifies a list of **final subroutines**. A final subroutine might be executed when  
 20 a data entity of that type is finalized (4.5.5.1).
- 21 A derived type is **finalizable** if it has any final subroutines or if it has any nonpointer, nonallocatable  
 22 component whose type is finalizable. A nonpointer data entity is finalizable if its type is finalizable.

**NOTE 4.45**

Final subroutines are effectively always “accessible”. They are called for entity finalization regardless of the accessibility of the type, its other type-bound procedures, or the subroutine name itself.

**NOTE 4.46**

Final subroutines are not inherited through type extension and cannot be overridden. The final subroutines of the parent type are called after any additional final subroutines of an extended type are called.

#### 1 4.5.5.1 The finalization process

2 Only finalizable entities are finalized. When an entity is **finalized**, the following steps are carried out  
3 in sequence:

- 4 (1) If the dynamic type of the entity has a final subroutine whose dummy argument has the  
5 same kind type parameters and rank as the entity being finalized, it is called with the entity  
6 as an actual argument. Otherwise, if there is an elemental final subroutine whose dummy  
7 argument has the same kind type parameters as the entity being finalized, it is called with  
8 the entity as an actual argument. Otherwise, no subroutine is called at this point.
- 9 (2) Each finalizable component that appears in the type definition is finalized. If the entity  
10 being finalized is an array, each finalizable component of each element of that entity is  
11 finalized separately.
- 12 (3) If the entity is of extended type and the parent type is finalizable, the parent component is  
13 finalized.

14 If several entities are to be finalized as a consequence of an event specified in 4.5.5.2, the order in which  
15 they are finalized is processor-dependent. A final subroutine shall not reference or define an object that  
16 has already been finalized.

17 If an object is not finalized, it retains its definition status and does not become undefined.

#### 18 4.5.5.2 When finalization occurs

19 When a pointer is deallocated its target is finalized. When an allocatable entity is deallocated, it is  
20 finalized.

21 A nonpointer, nonallocatable object that is not a dummy argument or function result is finalized im-  
22 mediately before it would become undefined due to execution of a RETURN or END statement (16.5.6,  
23 item (3)). If the object is defined in a module and there are no longer any active procedures referencing  
24 the module, it is processor-dependent whether it is finalized.

25 If an executable construct references a function, the result is finalized after execution of the innermost  
26 executable construct containing the reference.

27 If an executable construct references a structure constructor, the entity created by the structure con-  
28 structor is finalized after execution of the innermost executable construct containing the reference.

29 If a specification expression in a scoping unit references a function, the result is finalized before execution  
30 of the first executable statement in the scoping unit.

31 When a procedure is invoked, a nonpointer, nonallocatable object that is an actual argument associated  
32 with an INTENT(OUT) dummy argument is finalized.

33 When an intrinsic assignment statement is executed, *variable* is finalized after evaluation of *expr* and  
34 before the definition of *variable*.

#### NOTE 4.47

If finalization is used for storage management, it often needs to be combined with defined assign- ment.
---

35 If an object is allocated via pointer allocation and later becomes unreachable due to all pointers to that  
36 object having their pointer association status changed, it is processor dependent whether it is finalized.  
37 If it is finalized, it is processor dependent as to when the final subroutines are called.

### 1 4.5.5.3 Entities that are not finalized

- 2 If program execution is terminated, either by an error (e.g. an allocation failure) or by execution of  
 3 a STOP or END PROGRAM statement, entities existing immediately prior to termination are not  
 4 finalized.

#### NOTE 4.48

A nonpointer, nonallocatable object that has the SAVE attribute or that occurs in the main program is never finalized as a direct consequence of the execution of a RETURN or END statement.

A variable in a module is not finalized if it retains its definition status and value, even when there is no active procedure referencing the module.

### 5 4.5.6 Type extension

- 6 A nonsequence derived type that does not have the BIND attribute is an **extensible type**.  
 7 An extensible type that does not have the EXTENDS attribute is a **base type**. A type that has the  
 8 EXTENDS attribute is an **extended type**. The **parent type** of an extended type is the type named  
 9 in the EXTENDS attribute specification.

#### NOTE 4.49

The name of the parent type might be a local name introduced via renaming in a USE statement.

- 10 A base type is an **extension type** of itself only. An extended type is an extension of itself and of all  
 11 types for which its parent type is an extension.  
 12 An **abstract type** is a type that has the ABSTRACT attribute.

#### NOTE 4.50

A deferred binding (4.5.4) defers the implementation of a type-bound procedure to extensions of the type; it may appear only in an abstract type. The dynamic type of an object cannot be abstract; therefore, a deferred binding cannot be invoked. An extension of an abstract type need not be abstract if it has no deferred bindings. A short example of an abstract type is:

```

TYPE, ABSTRACT :: FILE_HANDLE
CONTAINS
  PROCEDURE(OPEN_FILE), DEFERRED, PASS(HANDLE) :: OPEN
  ...
END TYPE

```

For a more elaborate example see C.1.4.

### 13 4.5.6.1 Inheritance

- 14 An extended type includes all of the type parameters, all of the components, and the nonoverridden  
 15 (4.5.6.2) nonfinal procedure bindings of its parent type. These are said to be **inherited** by the extended  
 16 type from the parent type. They retain all of the attributes that they had in the parent type. Additional  
 17 type parameters, components, and procedure bindings may be declared in the derived-type definition of  
 18 the extended type.

**NOTE 4.51**

Inaccessible components and bindings of the parent type are also inherited, but they remain inaccessible in the extended type. Inaccessible entities occur if the type being extended is accessed via use association and has a private entity.

**NOTE 4.52**

A base type is not required to have any components, bindings, or parameters; an extended type is not required to have more components, bindings, or parameters than its parent type.

- 1 An extended type has a scalar, nonpointer, nonallocatable, **parent component** with the type and  
 2 type parameters of the parent type. The name of this component is the parent type name. It has the  
 3 accessibility of the parent type. Components of the parent component are **inheritance associated**  
 4 (16.4.4) with the corresponding components inherited from the parent type. An **ancestor component**  
 5 of a type is the parent component of the type or an ancestor component of the parent component.

**NOTE 4.53**

A component or type parameter declared in an extended type shall not have the same name as any accessible component or type parameter of its parent type.

**NOTE 4.54**

Examples:

```

TYPE POINT                                ! A base type
  REAL :: X, Y
END TYPE POINT

TYPE, EXTENDS(POINT) :: COLOR_POINT      ! An extension of TYPE(POINT)
  ! Components X and Y, and component name POINT, inherited from parent
  INTEGER :: COLOR
END TYPE COLOR_POINT

```

### 6 4.5.6.2 Type-bound procedure overriding

- 7 If a nongeneric binding specified in a type definition has the same binding name as a binding from the  
 8 parent type then the binding specified in the type definition **overrides** the one from the parent type.
- 9 The overriding binding and the overridden binding shall satisfy the following conditions:
- 10 (1) Either both shall have a passed-object dummy argument or neither shall.
  - 11 (2) If the overridden binding is pure then the overriding binding shall also be pure.
  - 12 (3) Either both shall be elemental or neither shall.
  - 13 (4) They shall have the same number of dummy arguments.
  - 14 (5) Passed-object dummy arguments, if any, shall correspond by name and position.
  - 15 (6) Dummy arguments that correspond by position shall have the same names and characteris-  
 16 tics, except for the type of the passed-object dummy arguments.
  - 17 (7) Either both shall be subroutines or both shall be functions having the same result charac-  
 18 teristics (12.2.2).
  - 19 (8) If the overridden binding is PUBLIC then the overriding binding shall not be PRIVATE.

**NOTE 4.55**

The following is an example of procedure overriding, expanding on the example in Note 4.42.

```
TYPE, EXTENDS (POINT) :: POINT_3D
  REAL :: Z
CONTAINS
  PROCEDURE, PASS :: LENGTH => POINT_3D_LENGTH
END TYPE POINT_3D
...
```

and in the *module-subprogram-part* of the same module:

```
REAL FUNCTION POINT_3D_LENGTH ( A, B )
  CLASS (POINT_3D), INTENT (IN) :: A
  CLASS (POINT), INTENT (IN) :: B
  SELECT TYPE(B)
    CLASS IS(POINT_3D)
      POINT_3D_LENGTH = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 + (A%Z-B%Z)**2 )
    RETURN
  END SELECT
  PRINT *, 'In POINT_3D_LENGTH, dynamic type of argument is incorrect.'
  STOP
END FUNCTION POINT_3D_LENGTH
```

- 1 If a generic binding specified in a type definition has the same *generic-spec* as an inherited binding, it
- 2 extends the generic interface and shall satisfy the requirements specified in 16.2.3.
- 3 If a generic binding in a type definition has the same *dtio-generic-spec* as one inherited from the parent, it
- 4 extends the type-bound generic interface for *dtio-generic-spec* and shall satisfy the requirements specified
- 5 in 16.2.3.
- 6 A binding of a type and a binding of an extension of that type are said to correspond if the latter binding
- 7 is the same binding as the former, overrides a corresponding binding, or is an inherited corresponding
- 8 binding.

### 9 4.5.7 Derived-type values

10 The **component value** of

- 11 (1) a pointer component is its pointer association;
- 12 (2) an allocatable component is its allocation status and, if it is allocated, its dynamic type and
- 13 type parameters, bounds and value;
- 14 (3) a nonpointer nonallocatable component is its value.

15 The set of values of a particular derived type consists of all possible sequences of the component values

16 of its components.

### 17 4.5.8 Derived-type specifier

18 A derived-type specifier is used in several contexts to specify a particular derived type and type param-

19 eters.

- 1 R455 *derived-type-spec* is *type-name* [ ( *type-param-spec-list* ) ]  
 2 R456 *type-param-spec* is [ *keyword* = ] *type-param-value*
- 3 C476 (R455) *type-name* shall be the name of an accessible derived type.
- 4 C477 (R455) *type-param-spec-list* shall appear only if the type is parameterized.
- 5 C478 (R455) There shall be at most one *type-param-spec* corresponding to each parameter of the type.  
 6 If a type parameter does not have a default value, there shall be a *type-param-spec* corresponding  
 7 to that type parameter.
- 8 C479 (R456) The *keyword*= may be omitted from a *type-param-spec* only if the *keyword*= has been  
 9 omitted from each preceding *type-param-spec* in the *type-param-spec-list*.
- 10 C480 (R456) Each *keyword* shall be the name of a parameter of the type.
- 11 C481 (R456) An asterisk may be used as a *type-param-value* in a *type-param-spec* only in the decla-  
 12 ration of a dummy argument or associate name or in the allocation of a dummy argument.
- 13 Type parameter values that do not have type parameter keywords specified correspond to type param-  
 14 eters in type parameter order (4.5.2.1). If a type parameter keyword is present, the value is assigned to  
 15 the type parameter named by the keyword. If necessary, the value is converted according to the rules of  
 16 intrinsic assignment (7.4.1.3) to a value of the same kind as the type parameter.
- 17 The value of a type parameter for which no *type-param-value* has been specified is its default value.

#### 18 4.5.9 Construction of derived-type values

19 A derived-type definition implicitly defines a corresponding **structure constructor** that allows con-  
 20 struction of values of that derived type. The type and type parameters of a constructed value are  
 21 specified by a derived type specifier.

- 22 R457 *structure-constructor* is *derived-type-spec* ( [ *component-spec-list* ] )  
 23 R458 *component-spec* is [ *keyword* = ] *component-data-source*  
 24 R459 *component-data-source* is *expr*  
 25 or *data-target*  
 26 or *proc-target*

- 27 C482 (R457) The *derived-type-spec* shall not specify an abstract type (4.5.6).
- 28 C483 (R457) At most one *component-spec* shall be provided for a component.
- 29 C484 (R457) If a *component-spec* is provided for a component, no *component-spec* shall be provided  
 30 for any component with which it is inheritance associated.
- 31 C485 (R457) A *component-spec* shall be provided for a component unless it has default initialization  
 32 or is inheritance associated with another component for which a *component-spec* is provided or  
 33 that has default initialization.
- 34 C486 (R458) The *keyword*= may be omitted from a *component-spec* only if the *keyword*= has been  
 35 omitted from each preceding *component-spec* in the constructor.
- 36 C487 (R458) Each *keyword* shall be the name of a component of the type.
- 37 C488 (R457) The type name and all components of the type for which a *component-spec* appears shall  
 38 be accessible in the scoping unit containing the structure constructor.
- 39 C489 (R457) If *derived-type-spec* is a type name that is the same as a generic name, the *component-*

- 1            *spec-list* shall not be a valid *actual-arg-spec-list* for a function reference that is resolvable as a  
 2            generic reference (12.4.4.1).
- 3 C490    (R459) A *data-target* shall correspond to a nonprocedure pointer component; a *proc-target* shall  
 4            correspond to a procedure pointer component.
- 5 C491    (R459) A *data-target* shall have the same rank as its corresponding component.

**NOTE 4.56**

The form 'name(...)' is interpreted as a generic *function-reference* if possible; it is interpreted as a *structure-constructor* only if it cannot be interpreted as a generic *function-reference*.

- 6 In the absence of a component keyword, each *component-data-source* is assigned to the corresponding  
 7 component in component order (4.5.3.5). If a component keyword appears, the *expr* is assigned to the  
 8 component named by the keyword. For a nonpointer component, the declared type and type parameters  
 9 of the component and *expr* shall conform in the same way as for a *variable* and *expr* in an intrinsic  
 10 assignment statement (7.4.1.2), as specified in Table 7.8. If necessary, each value of intrinsic type is  
 11 converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type and type  
 12 parameters with the corresponding component of the derived type. For a nonpointer nonallocatable  
 13 component, the shape of the expression shall conform with the shape of the component.
- 14 If a component with default initialization has no corresponding *component-data-source*, then the default  
 15 initialization is applied to that component.

**NOTE 4.57**

Because no parent components appear in the defined component ordering, a value for a parent component may be specified only with a component keyword. Examples of equivalent values using types defined in Note 4.54:

```
! Create values with components x = 1.0, y = 2.0, color = 3.
TYPE(POINT) :: PV = POINT(1.0, 2.0)           ! Assume components of TYPE(POINT)
                                                ! are accessible here.
...
COLOR_POINT( point=point(1,2), color=3)      ! Value for parent component
COLOR_POINT( point=PV, color=3)              ! Available even if TYPE(point)
                                                ! has private components
COLOR_POINT( 1, 2, 3)                        ! All components of TYPE(point)
                                                ! need to be accessible.
```

- 16 A structure constructor shall not appear before the referenced type is defined.

**NOTE 4.58**

This example illustrates a derived-type constant expression using a derived type defined in Note 4.17:

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

This could also be written as

```
PERSON (NAME = 'JOHN SMITH', AGE = 21)
```

**NOTE 4.59**

An example constructor using the derived type GENERAL\_POINT defined in Note 4.24 is

```
general_point(dim=3) ( (/ 1., 2., 3. /) )
```

- 1 For a pointer component, the corresponding *component-data-source* shall be an allowable *data-target* or  
 2 *proc-target* for such a pointer in a pointer assignment statement (7.4.2). If the component data source is  
 3 a pointer, the association of the component is that of the pointer; otherwise, the component is pointer  
 4 associated with the component data source.

**NOTE 4.60**

For example, if the variable TEXT were declared (5.1) to be

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

and BIBLIO were declared using the derived-type definition REFERENCE in Note 4.31

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

the statement

```
BIBLIO = REFERENCE (1, 1987, 1, "This is the title of the referenced &  

&paper", TEXT)
```

is valid and associates the pointer component SYNOPSIS of the object BIBLIO with the target object TEXT.

- 5 If a component of a derived type is allocatable, the corresponding constructor expression shall either be  
 6 a reference to the intrinsic function NULL with no arguments, an allocatable entity of the same rank,  
 7 or shall evaluate to an entity of the same rank. If the expression is a reference to the intrinsic function  
 8 NULL, the corresponding component of the constructor has a status of unallocated. If the expression  
 9 is an allocatable entity, the corresponding component of the constructor has the same allocation status  
 10 as that allocatable entity and, if it is allocated, the same dynamic type, bounds, and value; if a length  
 11 parameter of the component is deferred, its value is the same as the corresponding parameter of the  
 12 expression. Otherwise the corresponding component of the constructor has an allocation status of  
 13 allocated and has the same bounds and value as the expression.

**NOTE 4.61**

When the constructor is an actual argument, the allocation status of the allocatable component is available through the associated dummy argument.

#### 14 4.5.10 Derived-type operations and assignment

- 15 Intrinsic assignment of derived-type entities is described in 7.4.1. This standard does not specify any  
 16 intrinsic operations on derived-type entities. Any operation on derived-type entities or defined assign-  
 17 ment (7.4.1.4) for derived-type entities shall be defined explicitly by a function or a subroutine, and a  
 18 generic interface (4.5.1, 12.3.2.1).

## 1 4.6 Enumerations and enumerators

2 An enumeration is a set of enumerators. An enumerator is a named integer constant. An enumeration  
3 definition specifies the enumeration and its set of enumerators of the corresponding integer kind.

```
4 R460  enum-def           is  enum-def-stmt
5                                enumerator-def-stmt
6                                [ enumerator-def-stmt ] ...
7                                end-enum-stmt
8 R461  enum-def-stmt     is  ENUM, BIND(C)
9 R462  enumerator-def-stmt is  ENUMERATOR [ :: ] enumerator-list
10 R463  enumerator       is  named-constant [ = scalar-int-initialization-expr ]
11 R464  end-enum-stmt    is  END ENUM
```

12 C492 (R462) If = appears in an *enumerator*, a double-colon separator shall appear before the *enu-*  
13 *merator-list*.

14 For an enumeration, the kind is selected such that an integer type with that kind is interoperable (15.2.1)  
15 with the corresponding C enumeration type. The corresponding C enumeration type is the type that  
16 would be declared by a C enumeration specifier (6.7.2.2 of the C standard) that specified C enumeration  
17 constants with the same values as those specified by the *enum-def*, in the same order as specified by the  
18 *enum-def*.

19 The companion processor (2.5.10) shall be one that uses the same representation for the types declared  
20 by all C enumeration specifiers that specify the same values in the same order.

### NOTE 4.62

If a companion processor uses an unsigned type to represent a given enumeration type, the Fortran processor will use the signed integer type of the same width for the enumeration, even though some of the values of the enumerators cannot be represented in this signed integer type. The values of any such enumerators will be interoperable with the values declared in the C enumeration.

### NOTE 4.63

The C standard guarantees the enumeration constants fit in a C int (6.7.2.2 of the C standard). Therefore, the Fortran processor can evaluate all enumerator values using the integer type with kind parameter C.INT, and then determine the kind parameter of the integer type that is interoperable with the corresponding C enumerated type.

### NOTE 4.64

The C standard specifies that two enumeration types are compatible only if they specify enumeration constants with the same names and same values in the same order. This standard further requires that a C processor that is to be a companion processor of a Fortran processor use the same representation for two enumeration types if they both specify enumeration constants with the same values in the same order, even if the names are different.

21 An enumerator is treated as if it were explicitly declared with the PARAMETER attribute. The enu-  
22 merator is defined in accordance with the rules of intrinsic assignment (7.4) with the value determined  
23 as follows:

- 24 (1) If *scalar-int-initialization-expr* is specified, the value of the enumerator is the result of  
25 *scalar-int-initialization-expr*.
- 26 (2) If *scalar-int-initialization-expr* is not specified and the enumerator is the first enumerator  
27 in *enum-def*, the enumerator has the value 0.

- 1 (3) If *scalar-int-initialization-expr* is not specified and the enumerator is not the first enumer-  
 2 ator in *enum-def*, its value is the result of adding 1 to the value of the enumerator that  
 3 immediately precedes it in the *enum-def*.

**NOTE 4.65**

Example of an enumeration definition:

```
ENUM, BIND(C)
  ENUMERATOR :: RED = 4, BLUE = 9
  ENUMERATOR YELLOW
END ENUM
```

The kind type parameter for this enumeration is processor dependent, but the processor is required to select a kind sufficient to represent the values 4, 9, and 10, which are the values of its enumerators. The following declaration might be equivalent to the above enumeration definition.

```
INTEGER(SELECTED_INT_KIND(2)), PARAMETER :: RED = 4, BLUE = 9, YELLOW = 10
```

An entity of the same kind type parameter value can be declared using the intrinsic function KIND with one of the enumerators as its argument, for example

```
INTEGER(KIND(RED)) :: X
```

**NOTE 4.66**

There is no difference in the effect of declaring the enumerators in multiple ENUMERATOR statements or in a single ENUMERATOR statement. The order in which the enumerators in an enumeration definition are declared is significant, but the number of ENUMERATOR statements is not.

## 4 4.7 Construction of array values

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

7	R465	<i>array-constructor</i>	<b>is</b> (/ <i>ac-spec</i> /)
8			<b>or</b> <i>left-square-bracket ac-spec right-square-bracket</i>
9	R466	<i>ac-spec</i>	<b>is</b> <i>type-spec</i> ::
10			<b>or</b> [ <i>type-spec</i> ::] <i>ac-value-list</i>
11	R467	<i>left-square-bracket</i>	<b>is</b> [
12	R468	<i>right-square-bracket</i>	<b>is</b> ]
13	R469	<i>ac-value</i>	<b>is</b> <i>expr</i>
14			<b>or</b> <i>ac-implied-do</i>
15	R470	<i>ac-implied-do</i>	<b>is</b> ( <i>ac-value-list</i> , <i>ac-implied-do-control</i> )
16	R471	<i>ac-implied-do-control</i>	<b>is</b> <i>ac-do-variable</i> = <i>scalar-int-expr</i> , <i>scalar-int-expr</i> ■
17			■ [ , <i>scalar-int-expr</i> ]
18	R472	<i>ac-do-variable</i>	<b>is</b> <i>scalar-int-variable</i>

- 19 C493 (R472) *ac-do-variable* shall be a named variable.
- 20 C494 (R466) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the  
 21 same type and kind type parameters.
- 22 C495 (R466) If *type-spec* specifies an intrinsic type, each *ac-value* expression in the *array-constructor*

- 1 shall be of an intrinsic type that is in type conformance with a variable of type *type-spec* as  
 2 specified in Table 7.8.
- 3 C496 (R466) If *type-spec* specifies a derived type, all *ac-value* expressions in the *array-constructor*  
 4 shall be of that derived type and shall have the same kind type parameter values as specified by  
 5 *type-spec*.
- 6 C497 (R470) The *ac-do-variable* of an *ac-implied-do* that is in another *ac-implied-do* shall not appear  
 7 as the *ac-do-variable* of the containing *ac-implied-do*.
- 8 If *type-spec* is omitted, the type and type parameters of the array constructor are those of the *ac-value*  
 9 expressions.
- 10 If *type-spec* appears, it specifies the type and type parameters of the array constructor. Each *ac-value*  
 11 expression in the *array-constructor* shall be compatible with intrinsic assignment to a variable of this  
 12 type and type parameters. Each value is converted to the type parameters of the *array-constructor* in  
 13 accordance with the rules of intrinsic assignment (7.4.1.3).
- 14 The character length of an *ac-value* in an *ac-implied-do* whose iteration count is zero shall not depend  
 15 on the value of the *ac-do-variable* and shall not depend on the value of an expression that is not an  
 16 initialization expression.
- 17 If an *ac-value* is a scalar expression, its value specifies an element of the array constructor. If an *ac-*  
 18 *value* is an array expression, the values of the elements of the expression, in array element order (6.2.2.2),  
 19 specify the corresponding sequence of elements of the array constructor. If an *ac-value* is an *ac-implied-*  
 20 *do*, it is expanded to form a sequence of elements under the control of the *ac-do-variable*, as in the DO  
 21 construct (8.1.6.4).
- 22 For an *ac-implied-do*, the loop initialization and execution is the same as for a DO construct.
- 23 An empty sequence forms a zero-sized rank-one array.

**NOTE 4.67**

A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE intrinsic function (13.7.99). An example is:

```
X = (/ 3.2, 4.01, 6.5 /)
Y = RESHAPE (SOURCE = [ 2.0, [ 4.5, 4.5 ], X ], SHAPE = [ 3, 2 ])
```

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

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

**NOTE 4.68**

Examples of array constructors containing an implied DO are:

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

and

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

**NOTE 4.69**

Using the type definition for PERSON in Note 4.17, an example of the construction of a derived-type array value is:

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

**NOTE 4.70**

Using the type definition for LINE in Note 4.28, an example of the construction of a derived-type scalar value with a rank-2 array component is:

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

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

**NOTE 4.71**

Examples of zero-size array constructors are:

```
(/ INTEGER :: /)
(/ ( I, I = 1, 0) /)
```

**NOTE 4.72**

An example of an array constructor that specifies a length type parameter:

```
(/ CHARACTER(LEN=7) :: 'Takata', 'Tanaka', 'Hayashi' /)
```

In this constructor, without the type specification, it would have been necessary to specify all of the constants with the same character length.



## 1 Section 5: Data object declarations and specifications

2 Every data object has a type and rank and may have type parameters and other attributes that determine  
 3 the uses of the object. Collectively, these properties are the **attributes** of the object. The type of a  
 4 named data object is either specified explicitly in a type declaration statement or determined implicitly  
 5 by the first letter of its name (5.3). All of its attributes may be included in a type declaration statement  
 6 or may be specified individually in separate specification statements.

### NOTE 5.1

For example:

```
INTEGER :: INCOME, EXPENDITURE
```

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

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

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

### 7 5.1 Type declaration statements

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

9 R502 *declaration-type-spec* is *intrinsic-type-spec*

10 or TYPE ( *derived-type-spec* )

11 or CLASS ( *derived-type-spec* )

12 or CLASS ( \* )

13 C501 (R502) In a *declaration-type-spec*, every *type-param-value* that is not a colon or an asterisk shall  
 14 be a *specification-expr*.

15 C502 (R502) In a *declaration-type-spec* that uses the CLASS keyword, *derived-type-spec* shall specify  
 16 an extensible type.

### NOTE 5.2

A *declaration-type-spec* is used in a nonexecutable statement; a *type-spec* is used in an array constructor, a SELECT TYPE construct, or an ALLOCATE statement.

17 C503 (R502) The TYPE(*derived-type-spec*) shall not specify an abstract type (4.5.6).

18 R503 *attr-spec* is *access-spec*

19 or ALLOCATABLE

20 or ASYNCHRONOUS

21 or DIMENSION ( *array-spec* )

22 or EXTERNAL

23 or INTENT ( *intent-spec* )

24 or INTRINSIC

25 or *language-binding-spec*

1			or OPTIONAL
2			or PARAMETER
3			or POINTER
4			or PROTECTED
5			or SAVE
6			or TARGET
7			or VALUE
8			or VOLATILE
9	R504	<i>entity-decl</i>	is <i>object-name</i> [( <i>array-spec</i> )] [ * <i>char-length</i> ] [ <i>initialization</i> ]
10			or <i>function-name</i> [ * <i>char-length</i> ]
11	C504		(R504) If a <i>type-param-value</i> in a <i>char-length</i> in an <i>entity-decl</i> is not a colon or an asterisk, it
12			shall be a <i>specification-expr</i> .
13	R505	<i>object-name</i>	is <i>name</i>
14	C505		(R505) The <i>object-name</i> shall be the name of a data object.
15	R506	<i>initialization</i>	is = <i>initialization-expr</i>
16			or => <i>null-init</i>
17	R507	<i>null-init</i>	is <i>function-reference</i>
18	C506		(R507) The <i>function-reference</i> shall be a reference to the NULL intrinsic function with no
19			arguments.
20	C507		(R501) The same <i>attr-spec</i> shall not appear more than once in a given <i>type-declaration-stmt</i> .
21	C508		An entity shall not be explicitly given any attribute more than once in a scoping unit.
22	C509		(R501) An entity declared with the CLASS keyword shall be a dummy argument or have the
23			ALLOCATABLE or POINTER attribute.
24	C510		(R501) An array that has the POINTER or ALLOCATABLE attribute shall be specified with
25			an <i>array-spec</i> that is a <i>deferred-shape-spec-list</i> (5.1.2.5.3).
26	C511		(R501) An <i>array-spec</i> for an <i>object-name</i> that is a function result that does not have the AL-
27			LOCATABLE or POINTER attribute shall be an <i>explicit-shape-spec-list</i> .
28	C512		(R501) If the POINTER attribute is specified, the ALLOCATABLE, TARGET, EXTERNAL,
29			or INTRINSIC attribute shall not be specified.
30	C513		(R501) If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or
31			PARAMETER attribute shall not be specified.
32	C514		(R501) The PARAMETER attribute shall not be specified for a dummy argument, a pointer,
33			an allocatable entity, a function, or an object in a common block.
34	C515		(R501) The INTENT, VALUE, and OPTIONAL attributes may be specified only for dummy
35			arguments.
36	C516		(R501) The INTENT attribute shall not be specified for a dummy procedure without the
37			POINTER attribute.
38	C517		(R501) The SAVE attribute shall not be specified for an object that is in a common block, a
39			dummy argument, a procedure, a function result, an automatic data object, or an object with

- 1 the PARAMETER attribute.
- 2 C518 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.
- 3 C519 (R501) An entity in an *entity-decl-list* shall not have the EXTERNAL or INTRINSIC attribute  
4 specified unless it is a function.
- 5 C520 (R504) The \* *char-length* option is permitted only if the type specified is character.
- 6 C521 (R504) The *function-name* shall be the name of an external function, an intrinsic function, a  
7 function dummy procedure, or a statement function.
- 8 C522 (R501) The *initialization* shall appear if the statement contains a PARAMETER attribute  
9 (5.1.2.10).
- 10 C523 (R501) If *initialization* appears, a double-colon separator shall appear before the *entity-decl-list*.
- 11 C524 (R504) *initialization* shall not appear if *object-name* is a dummy argument, a function result, an  
12 object in a named common block unless the type declaration is in a block data program unit,  
13 an object in blank common, an allocatable variable, an external name, an intrinsic name, or an  
14 automatic object.
- 15 C525 (R504) If => appears in *initialization*, the object shall have the POINTER attribute. If =  
16 appears in *initialization*, the object shall not have the POINTER attribute.
- 17 C526 (R501) If the VOLATILE attribute is specified, the PARAMETER, INTRINSIC, EXTERNAL,  
18 or INTENT(IN) attribute shall not be specified.
- 19 C527 (R501) If the VALUE attribute is specified, the PARAMETER, EXTERNAL, POINTER,  
20 ALLOCATABLE, DIMENSION, VOLATILE, INTENT(INOUT), or INTENT(OUT) attribute  
21 shall not be specified.
- 22 C528 (R501) If the VALUE attribute is specified for a dummy argument of type character, the length  
23 parameter shall be omitted or shall be specified by an initialization expression with the value  
24 one.
- 25 C529 (R501) The VALUE attribute shall not be specified for a dummy procedure.
- 26 C530 (R501) The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a  
27 dummy argument of a procedure that has a *proc-language-binding-spec*.
- 28 C531 (R503) A *language-binding-spec* shall appear only in the specification part of a module.
- 29 C532 (R501) If a *language-binding-spec* is specified, the entity declared shall be an interoperable  
30 variable (15.2).
- 31 C533 (R501) If a *language-binding-spec* with a NAME= specifier appears, the *entity-decl-list* shall  
32 consist of a single *entity-decl*.
- 33 C534 (R503) The PROTECTED attribute is permitted only in the specification part of a module.
- 34 C535 (R501) The PROTECTED attribute is permitted only for a procedure pointer or named variable  
35 that is not in a common block.
- 36 C536 (R501) If the PROTECTED attribute is specified, the EXTERNAL, INTRINSIC, or PARAM-  
37 ETER attribute shall not be specified.
- 38 C537 A nonpointer object that has the PROTECTED attribute and is accessed by use association  
39 shall not appear in a variable definition context (16.5.7) or as the *data-target* or *proc-target* in

1           a *pointer-assignment-stmt*.

2 C538    A pointer object that has the PROTECTED attribute and is accessed by use association shall  
3           not appear as

4           (1)   A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,

5           (2)   An *allocate-object* in an *allocate-stmt* or *deallocate-stmt*, or

6           (3)   An actual argument in a reference to a procedure if the associated dummy argument is a  
7           pointer with the INTENT(OUT) or INTENT(INOUT) attribute.

8   A name that identifies a specific intrinsic function in a scoping unit has a type as specified in 13.6. An  
9   explicit type declaration statement is not required; however, it is permitted. Specifying a type for a  
10   generic intrinsic function name in a type declaration statement is not sufficient, by itself, to remove the  
11   generic properties from that function.

12   A function result may be declared to have the POINTER or ALLOCATABLE attribute.

13   A *specification-expr* in an *array-spec*, in a *type-param-value* in a *declaration-type-spec* corresponding to  
14   a length type parameter, or in a *char-length* in an *entity-decl* shall be an initialization expression unless  
15   it is in an interface body (12.3.2.1), the specification part of a subprogram, or the *declaration-type-spec*  
16   of a FUNCTION statement (12.5.2.1). If the data object being declared depends on the value of a  
17   *specification-expr* that is not an initialization expression, and it is not a dummy argument, such an  
18   object is called an **automatic data object**.

#### NOTE 5.3

An automatic object shall neither appear in a SAVE or DATA statement nor be declared with a SAVE attribute nor be initially defined by an *initialization*.

19   If a type parameter in a *declaration-type-spec* or in a *char-length* in an *entity-decl* is defined by an  
20   expression that is not an initialization expression, the type parameter value is established on entry to  
21   the procedure and is not affected by any redefinition or undefinition of the variables in the specification  
22   expression during execution of the procedure.

23   If an *entity-decl* contains *initialization* and the *object-name* does not have the PARAMETER attribute,  
24   the entity is a variable with **explicit initialization**. Explicit initialization alternatively may be specified  
25   in a DATA statement unless the variable is of a derived type for which default initialization is specified.  
26   If *initialization* is =*initialization-expr*, the *object-name* is initially defined with the value specified by  
27   the *initialization-expr*; if necessary, the value is converted according to the rules of intrinsic assignment  
28   (7.4.1.3) to a value that agrees in type, type parameters, and shape with the *object-name*. A variable,  
29   or part of a variable, shall not be explicitly initialized more than once in a program. If the variable is an  
30   array, it shall have its shape specified in either the type declaration statement or a previous attribute  
31   specification statement in the same scoping unit.

32   If *initialization* is =>*null-init*, *object-name* shall be a pointer, and its initial association status is disas-  
33   sociated.

34   The presence of *initialization* implies that *object-name* is saved, except for an *object-name* in a named  
35   common block or an *object-name* with the PARAMETER attribute. The implied SAVE attribute may  
36   be reaffirmed by explicit use of the SAVE attribute in the type declaration statement, by inclusion of  
37   the *object-name* in a SAVE statement (5.2.12), or by the appearance of a SAVE statement without a  
38   *saved-entity-list* in the same scoping unit.

#### NOTE 5.4

Examples of type declaration statements are:

## NOTE 5.4 (cont.)

```

REAL A (10)
LOGICAL, DIMENSION (5, 5) :: MASK1, MASK2
COMPLEX :: CUBE_ROOT = (-0.5, 0.866)
INTEGER, PARAMETER :: SHORT = SELECTED_INT_KIND (4)
INTEGER (SHORT) K      ! Range at least -9999 to 9999.
REAL (KIND (0.0D0)) A
REAL (KIND = 2) B
COMPLEX (KIND = KIND (0.0D0)) :: C
CHARACTER (LEN = 10, KIND = 2) A
CHARACTER B, C *20
TYPE (PERSON) :: CHAIRMAN
TYPE(NODE), POINTER :: HEAD => NULL ( )
TYPE (humongous_matrix (k=8, d=1000)) :: mat

```

(The last line above uses a type definition from Note 4.24.)

1 **5.1.1 Declaration type specifiers**

2 The *declaration-type-spec* in a type declaration statement specifies the type of the entities in the entity  
3 declaration list. This explicit type declaration may override or confirm the implicit type that could  
4 otherwise be indicated by the first letter of an entity name (5.3).

5 An *intrinsic-type-spec* in a type declaration statement is used to declare entities of intrinsic type.

6 **5.1.1.1 TYPE**

7 A TYPE type specifier is used to declare entities of a derived type.

8 Where a data entity is declared explicitly using the TYPE type specifier, the specified derived type shall  
9 have been defined previously in the scoping unit or be accessible there by use or host association. If  
10 the data entity is a function result, the derived type may be specified in the FUNCTION statement  
11 provided the derived type is defined within the body of the function or is accessible there by use or host  
12 association. If the derived type is specified in the FUNCTION statement and is defined within the body  
13 of the function, it is as if the function result variable was declared with that derived type immediately  
14 following the *derived-type-def* of the specified derived type.

15 A scalar entity of derived type is a **structure**. If a derived type has the SEQUENCE property, a scalar  
16 entity of the type is a **sequence structure**.

17 **5.1.1.2 CLASS**

18 A **polymorphic** entity is a data entity that is able to be of differing types during program execution.  
19 The type of a data entity at a particular point during execution of a program is its **dynamic type**. The  
20 **declared type** of a data entity is the type that it is declared to have, either explicitly or implicitly.

21 A CLASS type specifier is used to declare polymorphic objects. The declared type of a polymorphic  
22 object is the specified type if the CLASS type specifier contains a type name.

23 An object declared with the CLASS(\*) specifier is an **unlimited polymorphic** object. An unlimited  
24 polymorphic entity is not declared to have a type. It is not considered to have the same declared type  
25 as any other entity, including another unlimited polymorphic entity.

26 A nonpolymorphic entity is **type compatible** only with entities of the same type. For a polymorphic  
27 entity, type compatibility is based on its declared type. A polymorphic entity that is not an unlimited

- 1 polymorphic entity is type compatible with entities of the same type or any of its extensions. Even  
 2 though an unlimited polymorphic entity is not considered to have a declared type, it is type compatible  
 3 with all entities. An entity is said to be type compatible with a type if it is type compatible with entities  
 4 of that type.
- 5 Two entities are **type incompatible** if neither is type compatible with the other.
- 6 An entity is type, kind, and rank compatible, or **TKR compatible**, with another entity if the first  
 7 entity is type compatible with the second, the kind type parameters of the first entity have the same  
 8 values as corresponding kind type parameters of the second, and both entities have the same rank.
- 9 A polymorphic allocatable object may be allocated to be of any type with which it is type compatible.  
 10 A polymorphic pointer or dummy argument may, during program execution, be associated with objects  
 11 with which it is type compatible.
- 12 The dynamic type of an allocated allocatable polymorphic object is the type with which it was allocated.  
 13 The dynamic type of an associated polymorphic pointer is the dynamic type of its target. The dynamic  
 14 type of a nonallocatable nonpointer polymorphic dummy argument is the dynamic type of its associated  
 15 actual argument. The dynamic type of an unallocated allocatable or a disassociated pointer is the same  
 16 as its declared type. The dynamic type of an entity identified by an associate name (8.1.4) is the dynamic  
 17 type of the selector with which it is associated. The dynamic type of an object that is not polymorphic  
 18 is its declared type.

**NOTE 5.5**

Only components of the declared type of a polymorphic object may be designated by component selection (6.1.2).

19 **5.1.2 Attributes**

- 20 The additional attributes that may appear in the attribute specification of a type declaration statement  
 21 further specify the nature of the entities being declared or specify restrictions on their use in the program.

22 **5.1.2.1 Accessibility attribute**

- 23 The **accessibility attribute** specifies the accessibility of an entity via a particular identifier.

24 R508    *access-spec*                            **is** PUBLIC  
 25    **or** PRIVATE

- 26 C539    (R508) An *access-spec* shall appear only in the *specification-part* of a module.

- 27 Identifiers that are specified in a module or accessible in that module by use association have either  
 28 the PUBLIC or PRIVATE attribute. Identifiers for which an *access-spec* is not explicitly specified in  
 29 that module have the default accessibility attribute for that module. The default accessibility attribute  
 30 for a module is PUBLIC unless it has been changed by a PRIVATE statement (5.2.1). Only identifiers  
 31 that have the PUBLIC attribute in that module are available to be accessed from that module by use  
 32 association.

**NOTE 5.6**

In order for an identifier to be accessed by use association, it must have the PUBLIC attribute in the module from which it is accessed. It can nonetheless have the PRIVATE attribute in a module in which it is accessed by use association, and therefore not be available for use association from a module where it is PRIVATE.

**NOTE 5.7**

An example of an accessibility specification is:

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

1 **5.1.2.2 ALLOCATABLE attribute**

2 An object with the **ALLOCATABLE attribute** is one for which space is allocated by an **ALLOCATE**  
3 statement (6.3.1) or by a derived-type intrinsic assignment statement (7.4.1.3).

4 **5.1.2.3 ASYNCHRONOUS attribute**

5 The **ASYNCHRONOUS attribute** specifies that a variable may be subject to asynchronous in-  
6 put/output.

7 The base object of a variable shall have the **ASYNCHRONOUS** attribute in a scoping unit if:

- 8 (1) the variable appears in an executable statement or specification expression in that scoping  
9 unit and  
10 (2) any statement of the scoping unit is executed while the variable is a pending I/O storage  
11 sequence affector (9.5.1.4)

12 The **ASYNCHRONOUS** attribute may be conferred implicitly by the use of a variable in an asynchronous  
13 input/output statement (9.5.1.4).

14 An object may have the **ASYNCHRONOUS** attribute in a particular scoping unit without necessarily  
15 having it in other scoping units. If an object has the **ASYNCHRONOUS** attribute, then all of its  
16 subobjects also have the **ASYNCHRONOUS** attribute.

**NOTE 5.8**

The **ASYNCHRONOUS** attribute specifies the variables that might be associated with a pending input/output storage sequence (the actual memory locations on which asynchronous input/output is being performed) while the scoping unit is in execution. This information could be used by the compiler to disable certain code motion optimizations.

The **ASYNCHRONOUS** attribute is similar to the **VOLATILE** attribute. It is intended to facilitate traditional code motion optimizations in the presence of asynchronous input/output.

17 **5.1.2.4 BIND attribute for data entities**

18 The **BIND** attribute for a variable or common block specifies that it is capable of interoperating with a  
19 C variable that has external linkage (15.3).

20 R509 *language-binding-spec* **is** BIND (C [, NAME = *scalar-char-initialization-expr* ])

21 C540 (R509) The *scalar-char-initialization-expr* shall be of default character kind.

**NOTE 5.9**

The C standard provides a facility for creating C identifiers whose characters are not restricted to the C basic character set. Such a C identifier is referred to as a universal character name (6.4.3 of the C standard). The name of such a C identifier may include characters that are not part of the representation method used by the processor for type default character. If so, the C entity cannot be linked (12.5.3, 15.3.1) with a Fortran entity.

This standard does not require a processor to provide a means of linking Fortran entities with C

**NOTE 5.9 (cont.)**

entities whose names are specified using the universal character name facility.

- 1 The BIND attribute implies the SAVE attribute, which may be confirmed by explicit specification.

**NOTE 5.10**

Specifying the BIND attribute for an entity might have no discernable effect for a processor that is its own companion processor.

2 **5.1.2.5 DIMENSION attribute**

- 3 The **DIMENSION attribute** specifies that an entity is an array. The rank or rank and shape is  
 4 specified by the *array-spec*, if there is one, in the *entity-decl*, or by the *array-spec* in the DIMENSION  
 5 *attr-spec* otherwise. To specify that an entity is an array in a type declaration statement, either the  
 6 DIMENSION *attr-spec* shall appear, or an *array-spec* shall appear in the *entity-decl*. The appearance  
 7 of an *array-spec* in an *entity-decl* specifies the DIMENSION attribute for the entity. The DIMENSION  
 8 attribute alternatively may be specified in the specification statements DIMENSION, ALLOCATABLE,  
 9 POINTER, TARGET, or COMMON.

10 R510 *array-spec*                                **is** *explicit-shape-spec-list*  
 11    **or** *assumed-shape-spec-list*  
 12    **or** *deferred-shape-spec-list*  
 13    **or** *assumed-size-spec*

- 14 C541 (R510)The maximum rank is seven.

**NOTE 5.11**

Examples of DIMENSION attribute specifications are:

```

SUBROUTINE EX (N, A, B)
  REAL, DIMENSION (N, 10) :: W           ! Automatic explicit-shape array
  REAL A (:), B (0:)                   ! Assumed-shape arrays
  REAL, POINTER :: D (:, :)             ! Array pointer
  REAL, DIMENSION (:), POINTER :: P     ! Array pointer
  REAL, ALLOCATABLE, DIMENSION (:) :: E ! Allocatable array

```

15 **5.1.2.5.1 Explicit-shape array**

- 16 An **explicit-shape array** is a named array that is declared with an *explicit-shape-spec-list*. This specifies  
 17 explicit values for the bounds in each dimension of the array.

18 R511 *explicit-shape-spec*                    **is** [ *lower-bound* : ] *upper-bound*  
 19 R512 *lower-bound*                            **is** *specification-expr*  
 20 R513 *upper-bound*                           **is** *specification-expr*

- 21 C542 (R511) An explicit-shape array whose bounds are not initialization expressions shall be a dummy  
 22 argument, a function result, or an automatic array of a procedure.

- 23 An **automatic array** is an explicit-shape array that is declared in a subprogram, is not a dummy  
 24 argument, and has bounds that are not initialization expressions.

- 25 If an explicit-shape array has bounds that are not initialization expressions, the bounds, and hence  
 26 shape, are determined at entry to the procedure by evaluating the bounds expressions. The bounds of

1 such an array are unaffected by the redefinition or undefinition of any variable during execution of the  
2 procedure.

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

#### 10 5.1.2.5.2 Assumed-shape array

11 An **assumed-shape array** is a nonpointer dummy argument array that takes its shape from the asso-  
12 ciated actual argument array.

13 R514 *assumed-shape-spec*            **is** [ *lower-bound* ] :

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

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

#### 19 5.1.2.5.3 Deferred-shape array

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

21 An **allocatable array** is an array that has the ALLOCATABLE attribute and a specified rank, but its  
22 bounds, and hence shape, are determined by allocation or argument association.

23 An array with the ALLOCATABLE attribute shall be declared with a *deferred-shape-spec-list*.

24 An **array pointer** is an array with the POINTER attribute and a specified rank. Its bounds, and hence  
25 shape, are determined when it is associated with a target. An array with the POINTER attribute shall  
26 be declared with a *deferred-shape-spec-list*.

27 R515 *deferred-shape-spec*            **is** :

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

29 The size, bounds, and shape of an unallocated allocatable array or a disassociated array pointer are  
30 undefined. No part of such an array shall be referenced or defined; however, the array may appear as an  
31 argument to an intrinsic inquiry function as specified in 13.1.

32 The bounds of each dimension of an allocatable array are those specified when the array is allocated.

33 The bounds of each dimension of an array pointer may be specified in two ways:

- 34       (1) in an ALLOCATE statement (6.3.1) when the target is allocated, or
- 35       (2) by pointer assignment (7.4.2).

36 The bounds of the array target or allocatable array are unaffected by any subsequent redefinition or  
37 undefinition of variables involved in the bounds' specification expressions.

#### 1 5.1.2.5.4 Assumed-size array

2 An **assumed-size array** is a dummy argument array whose size is assumed from that of an associated  
 3 actual argument. The rank and extents may differ for the actual and dummy arrays; only the size of the  
 4 actual array is assumed by the dummy array. An assumed-size array is declared with an *assumed-size-*  
 5 *spec*.

6 R516 *assumed-size-spec* is [ *explicit-shape-spec-list* , ] [ *lower-bound* : ] \*

7 C543 An *assumed-size-spec* shall not appear except as the declaration of the array bounds of a dummy  
 8 data argument.

9 C544 An assumed-size array with INTENT (OUT) shall not be of a type for which default initialization  
 10 is specified.

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

- 12 (1) If the actual argument associated with the assumed-size dummy array is an array of any  
 13 type other than default character, the size is that of the actual array.
- 14 (2) If the actual argument associated with the assumed-size dummy array is an array element  
 15 of any type other than default character with a subscript order value of  $r$  (6.2.2.2) in an  
 16 array of size  $x$ , the size of the dummy array is  $x - r + 1$ .
- 17 (3) If the actual argument is a default character array, default character array element, or a  
 18 default character array element substring (6.1.1), and if it begins at character storage unit  $t$   
 19 of an array with  $c$  character storage units, the size of the dummy array is  $\text{MAX}(\text{INT}((c -$   
 20  $t + 1)/e), 0)$ , where  $e$  is the length of an element in the dummy character array.
- 21 (4) If the actual argument is of type default character and is a scalar that is not an array element  
 22 or array element substring designator, the size of the dummy array is  $\text{MAX}(\text{INT}(l/e), 0)$ ,  
 23 where  $e$  is the length of an element in the dummy character array and  $l$  is the length of the  
 24 actual argument.

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

26 An assumed-size array has no upper bound in its last dimension and therefore has no extent in its last  
 27 dimension and no shape. An assumed-size array name shall not be written as a whole array reference  
 28 except as an actual argument in a procedure reference for which the shape is not required.

29 The bounds of the first  $n - 1$  dimensions are those specified by the *explicit-shape-spec-list*, if present, in  
 30 the *assumed-size-spec*. The lower bound of the last dimension is *lower-bound*, if present, and 1 otherwise.  
 31 An assumed-size array may be subscripted or sectioned (6.2.2.3). The upper bound shall not be omitted  
 32 from a subscript triplet in the last dimension.

33 If an assumed-size array has bounds that are not initialization expressions, the bounds are determined  
 34 at entry to the procedure. The bounds of such an array are unaffected by the redefinition or undefinition  
 35 of any variable during execution of the procedure.

#### 36 5.1.2.6 EXTERNAL attribute

37 The **EXTERNAL attribute** specifies that an entity is an external procedure, dummy procedure,  
 38 procedure pointer, or block data subprogram. This attribute may also be specified by an EXTER-  
 39 NAL statement (12.3.2.2), a *procedure-declaration-stmt* (12.3.2.3) or an interface body that is not in an  
 40 abstract interface block (12.3.2.1).

41 If an external procedure or dummy procedure is used as an actual argument or is the target of a procedure  
 42 pointer assignment, it shall be declared to have the EXTERNAL attribute.

43 A procedure that has both the EXTERNAL and POINTER attributes is a procedure pointer.

1 **5.1.2.7 INTENT attribute**

2 The **INTENT attribute** specifies the intended use of a dummy argument.

3 R517 *intent-spec*                           **is** IN  
 4   **or** OUT  
 5   **or** INOUT

6 C545 (R517) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable  
 7 definition context (16.5.7).

8 C546 (R517) A pointer object with the INTENT (IN) attribute shall not appear as

- 9       (1) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,  
 10       (2) An *allocate-object* in an *allocate-stmt* or *deallocate-stmt*, or  
 11       (3) An actual argument in a reference to a procedure if the associated dummy argument is a  
 12 pointer with the INTENT (OUT) or INTENT (INOUT) attribute.

13 The INTENT (IN) attribute for a nonpointer dummy argument specifies that it shall neither be de-  
 14 fined nor become undefined during the execution of the procedure. The INTENT (IN) attribute for a  
 15 pointer dummy argument specifies that during the execution of the procedure its association shall not  
 16 be changed except that it may become undefined if the target is deallocated other than through the  
 17 pointer (16.4.2.1.3).

18 The INTENT (OUT) attribute for a nonpointer dummy argument specifies that it shall be defined  
 19 before a reference to the dummy argument is made within the procedure and any actual argument that  
 20 becomes associated with such a dummy argument shall be definable. On invocation of the procedure,  
 21 such a dummy argument becomes undefined except for components of an object of derived type for  
 22 which default initialization has been specified. The INTENT (OUT) attribute for a pointer dummy  
 23 argument specifies that on invocation of the procedure the pointer association status of the dummy  
 24 argument becomes undefined. Any actual argument associated with such a pointer dummy shall be a  
 25 pointer variable.

26 The INTENT (INOUT) attribute for a nonpointer dummy argument specifies that it is intended for use  
 27 both to receive data from and to return data to the invoking scoping unit. Such a dummy argument may  
 28 be referenced or defined. Any actual argument that becomes associated with such a dummy argument  
 29 shall be definable. The INTENT (INOUT) attribute for a pointer dummy argument specifies that it is  
 30 intended for use both to receive a pointer association from and to return a pointer association to the  
 31 invoking scoping unit. Any actual argument associated with such a pointer dummy shall be a pointer  
 32 variable.

33 If no INTENT attribute is specified for a dummy argument, its use is subject to the limitations of the  
 34 associated actual argument (12.4.1.2, 12.4.1.3, 12.4.1.4).

**NOTE 5.12**

An example of INTENT specification is:

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

35 If an object has an INTENT attribute, then all of its subobjects have the same INTENT attribute.

**NOTE 5.13**

If a dummy argument is a derived-type object with a pointer component, then the pointer as a pointer is a subobject of the dummy argument, but the target of the pointer is not. Therefore, the restrictions on subobjects of the dummy object apply to the pointer in contexts where it is used as a pointer, but not in contexts where it is dereferenced to indicate its target. For example, if X is a dummy argument of derived type with an integer pointer component P, and X has INTENT(IN), then the statement

```
X%P => NEW_TARGET
```

is prohibited, but

```
X%P = 0
```

is allowed (provided that X%P is associated with a definable target).

Similarly, the INTENT restrictions on pointer dummy arguments apply only to the association of the dummy argument; they do not restrict the operations allowed on its target.

**NOTE 5.14**

Argument intent specifications serve several purposes in addition to documenting the intended use of dummy arguments. A processor can check whether an INTENT (IN) dummy argument is used in a way that could redefine it. A slightly more sophisticated processor could check to see whether an INTENT (OUT) dummy argument could possibly be referenced before it is defined. If the procedure's interface is explicit, the processor can also verify that actual arguments corresponding to INTENT (OUT) or INTENT (INOUT) dummy arguments are definable. A more sophisticated processor could use this information to optimize the translation of the referencing scoping unit by taking advantage of the fact that actual arguments corresponding to INTENT (IN) dummy arguments will not be changed and that any prior value of an actual argument corresponding to an INTENT (OUT) dummy argument will not be referenced and could thus be discarded.

INTENT (OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If there is any possibility that an argument should retain its current value rather than being redefined, INTENT (INOUT) should be used rather than INTENT (OUT), even if there is no explicit reference to the value of the dummy argument. Because an INTENT(OUT) variable is considered undefined on entry to the procedure, any default initialization specified for its type will be applied.

INTENT (INOUT) is not equivalent to omitting the INTENT attribute. The argument corresponding to an INTENT (INOUT) dummy argument always shall be definable, while an argument corresponding to a dummy argument without an INTENT attribute need be definable only if the dummy argument is actually redefined.

1 **5.1.2.8 INTRINSIC attribute**

2 The **INTRINSIC attribute** confirms that a name is the specific name (13.6) or generic name (13.5)  
 3 of an intrinsic procedure. The INTRINSIC attribute allows the specific name of an intrinsic procedure  
 4 that is listed in 13.6 and not marked with a bullet (•) to be used as an actual argument (12.4).

5 Declaring explicitly that a generic intrinsic procedure name has the INTRINSIC attribute does not cause  
 6 that name to lose its generic property.

7 If the specific name of an intrinsic procedure (13.6) is used as an actual argument, the name shall be

1 explicitly specified to have the INTRINSIC attribute.

2 C547 (R503) (R1216) If the name of a generic intrinsic procedure is explicitly declared to have the  
 3 INTRINSIC attribute, and it is also the generic name in one or more generic interfaces (12.3.2.1)  
 4 accessible in the same scoping unit, the procedures in the interfaces and the specific intrinsic  
 5 procedures shall all be functions or all be subroutines, and the characteristics of the specific  
 6 intrinsic procedures and the procedures in the interfaces shall differ as specified in 16.2.3.

#### 7 5.1.2.9 OPTIONAL attribute

8 The **OPTIONAL attribute** specifies that the dummy argument need not be associated with an actual  
 9 argument in a reference to the procedure (12.4.1.6). The PRESENT intrinsic function may be used  
 10 to determine whether an actual argument has been associated with a dummy argument having the  
 11 OPTIONAL attribute.

#### 12 5.1.2.10 PARAMETER attribute

13 The **PARAMETER attribute** specifies entities that are named constants. The *object-name* has the  
 14 value specified by the *initialization-expr* that appears on the right of the equals; if necessary, the value  
 15 is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that agrees in type, type  
 16 parameters, and shape with the *object-name*.

17 A named constant shall not be referenced unless it has been defined previously in the same statement,  
 18 defined in a prior statement, or made accessible by use or host association.

#### NOTE 5.15

Examples of declarations with a PARAMETER attribute are:

```
REAL, PARAMETER :: ONE = 1.0, Y = 4.1 / 3.0
INTEGER, DIMENSION (3), PARAMETER :: ORDER = (/ 1, 2, 3 /)
TYPE(NODE), PARAMETER :: DEFAULT = NODE(0, NULL ( ))
```

#### 19 5.1.2.11 POINTER attribute

20 Entities with the **POINTER attribute** can be associated with different data objects or procedures  
 21 during execution of a program. A pointer is either a data pointer or a procedure pointer. Procedure  
 22 pointers are described in 12.3.2.3.

23 A data pointer shall neither be referenced nor defined unless it is pointer associated with a target object  
 24 that may be referenced or defined.

25 If a data pointer is associated, the values of its deferred type parameters are the same as the values of  
 26 the corresponding type parameters of its target.

27 A procedure pointer shall not be referenced unless it is pointer associated with a target procedure.

#### NOTE 5.16

Examples of POINTER attribute specifications are:

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

For a more elaborate example see C.2.1.

### 1 5.1.2.12 PROTECTED attribute

- 2 The **PROTECTED attribute** imposes limitations on the usage of module entities.
- 3 Other than within the module in which an entity is given the PROTECTED attribute,
- 4 (1) if it is a nonpointer object, it is not definable, and
- 5 (2) if it is a pointer, its association status shall not be changed except that it may become
- 6 undefined if its target is deallocated other than through the pointer (16.4.2.1.3) or if its
- 7 target becomes undefined by execution of a RETURN or END statement.
- 8 If an object has the PROTECTED attribute, all of its subobjects have the PROTECTED attribute.

#### NOTE 5.17

An example of the PROTECTED attribute:

```

MODULE temperature
  REAL, PROTECTED :: temp_c, temp_f
CONTAINS
  SUBROUTINE set_temperature_c(c)
    REAL, INTENT(IN) :: c
    temp_c = c
    temp_f = temp_c*(9.0/5.0) + 32
  END SUBROUTINE
END MODULE

```

The PROTECTED attribute ensures that the variables `temp_c` and `temp_f` cannot be modified other than via the `set_temperature_c` procedure, thus keeping them consistent with each other.

### 9 5.1.2.13 SAVE attribute

- 10 An entity with the **SAVE attribute**, in the scoping unit of a subprogram, retains its association status,
- 11 allocation status, definition status, and value after execution of a RETURN or END statement unless it
- 12 is a pointer and its target becomes undefined (16.4.2.1.3(4)). It is shared by all instances (12.5.2.3) of
- 13 the subprogram.
- 14 An entity with the SAVE attribute, declared in the scoping unit of a module, retains its association
- 15 status, allocation status, definition status, and value after a RETURN or END statement is executed in
- 16 a procedure that accesses the module unless it is a pointer and its target becomes undefined.
- 17 A **saved** entity is an entity that has the SAVE attribute. An **unsaved** entity is an entity that does not
- 18 have the SAVE attribute.
- 19 The SAVE attribute may appear in declarations in a main program and has no effect.

### 20 5.1.2.14 TARGET attribute

- 21 An object with the **TARGET attribute** may have a pointer associated with it (7.4.2). An object
- 22 without the TARGET attribute shall not have an accessible pointer associated with it.

#### NOTE 5.18

In addition to variables explicitly declared to have the TARGET attribute, the objects created by allocation of pointers (6.3.1.2) have the TARGET attribute.

- 23 If an object has the TARGET attribute, then all of its nonpointer subobjects also have the TARGET

1 attribute.

**NOTE 5.19**

Examples of TARGET attribute specifications are:

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

For a more elaborate example see [C.2.2](#).

**NOTE 5.20**

Every object designator that starts from a target object will have either the TARGET or POINTER attribute. If pointers are involved, the designator might not necessarily be a subobject of the original target object, but because pointers may point only to targets, there is no way to end up at a nonpointer that is not a target.

**2 5.1.2.15 VALUE attribute**

3 The **VALUE attribute** specifies a type of argument association ([12.4.1.2](#)) for a dummy argument.

**4 5.1.2.16 VOLATILE attribute**

5 The **VOLATILE attribute** specifies that an object may be referenced, defined, or become undefined,  
6 by means not specified by the program.

7 An object may have the VOLATILE attribute in a particular scoping unit without necessarily having  
8 it in other scoping units. If an object has the VOLATILE attribute, then all of its subobjects also have  
9 the VOLATILE attribute.

**NOTE 5.21**

The Fortran processor should use the most recent definition of a volatile object when a value is required. Likewise, it should make the most recent Fortran definition available. It is the programmer's responsibility to manage the interactions with the non-Fortran processes.

10 A pointer with the VOLATILE attribute may additionally have its association status and array bounds  
11 changed by means not specified by the program.

**NOTE 5.22**

If the target of a pointer is referenced, defined, or becomes undefined, by means not specified by the program, while the pointer is associated with the target, then the pointer shall have the VOLATILE attribute. Usually a pointer should have the VOLATILE attribute if its target has the VOLATILE attribute. Similarly, all members of an EQUIVALENCE group should have the VOLATILE attribute if one member has the VOLATILE attribute.

12 An allocatable object with the VOLATILE attribute may additionally have its allocation status, dynamic  
13 type and type parameters, and array bounds changed by means not specified by the program.

**14 5.2 Attribute specification statements**

15 All attributes (other than type) may be specified for entities, independently of type, by separate at-  
16 tribute specification statements. The combination of attributes that may be specified for a particular  
17 entity is subject to the same restrictions as for type declaration statements regardless of the method of

1 specification. This also applies to PROCEDURE, EXTERNAL, and INTRINSIC statements.

## 2 5.2.1 Accessibility statements

3 R518 *access-stmt* is *access-spec* [ [ :: ] *access-id-list* ]  
 4 R519 *access-id* is *use-name*  
 5 or *generic-spec*

6 C548 (R518) An *access-stmt* shall appear only in the *specification-part* of a module. Only one ac-  
 7 cessibility statement with an omitted *access-id-list* is permitted in the *specification-part* of a  
 8 module.

9 C549 (R519) Each *use-name* shall be the name of a named variable, procedure, derived type, named  
 10 constant, or namelist group.

11 An *access-stmt* with an *access-id-list* specifies the accessibility attribute (5.1.2.1), PUBLIC or PRIVATE,  
 12 of each *access-id* in the list. An *access-stmt* without an *access-id* list specifies the default accessibility  
 13 that applies to all potentially accessible identifiers in the *specification-part* of the module. The  
 14 statement

15 PUBLIC

16 specifies a default of public accessibility. The statement

17 PRIVATE

18 specifies a default of private accessibility. If no such statement appears in a module, the default is public  
 19 accessibility.

### NOTE 5.23

Examples of accessibility statements are:

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

## 20 5.2.2 ALLOCATABLE statement

21 R520 *allocatable-stmt* is ALLOCATABLE [ :: ] ■  
 22 ■ *object-name* [ ( *deferred-shape-spec-list* ) ] ■  
 23 ■ [ , *object-name* [ ( *deferred-shape-spec-list* ) ] ] ...

24 This statement specifies the ALLOCATABLE attribute (5.1.2.2) for a list of objects.

### NOTE 5.24

An example of an ALLOCATABLE statement is:

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

## 25 5.2.3 ASYNCHRONOUS statement

26 R521 *asynchronous-stmt* is ASYNCHRONOUS [ :: ] *object-name-list*

27 The ASYNCHRONOUS statement specifies the ASYNCHRONOUS attribute (5.1.2.3) for a list of ob-

1 jects.

## 2 5.2.4 BIND statement

3 R522 *bind-stmt* is *language-binding-spec* [ *::* ] *bind-entity-list*

4 R523 *bind-entity* is *entity-name*

5 or / *common-block-name* /

6 C550 (R522) If any *bind-entity* in a *bind-stmt* is an *entity-name*, the *bind-stmt* shall appear in the  
7 specification part of a module and the entity shall be an interoperable variable (15.2.4, 15.2.5).

8 C551 (R522) If the *language-binding-spec* has a NAME= specifier, the *bind-entity-list* shall consist of  
9 a single *bind-entity*.

10 C552 (R522) If a *bind-entity* is a common block, each variable of the common block shall be interoperable  
11 (15.2.4, 15.2.5).

12 The BIND statement specifies the BIND attribute (5.1.2.4) for a list of variables and common blocks.

## 13 5.2.5 DATA statement

14 R524 *data-stmt* is DATA *data-stmt-set* [ [ , ] *data-stmt-set* ] ...

15 This statement is used to specify explicit initialization (5.1).

16 A variable, or part of a variable, shall not be explicitly initialized more than once in a program. If a  
17 nonpointer object has been specified with default initialization in a type definition, it shall not appear  
18 in a *data-stmt-object-list*.

19 A variable that appears in a DATA statement and has not been typed previously may appear in a  
20 subsequent type declaration only if that declaration confirms the implicit typing. An array name,  
21 array section, or array element that appears in a DATA statement shall have had its array properties  
22 established by a previous specification statement.

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

26 R525 *data-stmt-set* is *data-stmt-object-list* / *data-stmt-value-list* /

27 R526 *data-stmt-object* is *variable*

28 or *data-implied-do*

29 R527 *data-implied-do* is ( *data-i-do-object-list* , *data-i-do-variable* = ■  
30 ■ *scalar-int-expr* , *scalar-int-expr* [ , *scalar-int-expr* ] )

31 R528 *data-i-do-object* is *array-element*

32 or *scalar-structure-component*

33 or *data-implied-do*

34 R529 *data-i-do-variable* is *scalar-int-variable*

35 C553 (R526) In a *variable* that is a *data-stmt-object*, any subscript, section subscript, substring start-  
36 ing point, and substring ending point shall be an initialization expression.

37 C554 (R526) A variable whose designator is included in a *data-stmt-object-list* or a *data-i-do-object-*  
38 *list* shall not be: a dummy argument, made accessible by use association or host association, in  
39 a named common block unless the DATA statement is in a block data program unit, in a blank  
40 common block, a function name, a function result name, an automatic object, or an allocatable

- 1 variable.
- 2 C555 (R526) A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be an object  
3 designator in which a pointer appears other than as the entire rightmost *part-ref*.
- 4 C556 (R529) The *data-i-do-variable* shall be a named variable.
- 5 C557 (R527) A *scalar-int-expr* of a *data-implied-do* shall involve as primaries only constants, subob-  
6 jects of constants, or DO variables of the containing *data-implied-dos*, and each operation shall  
7 be intrinsic.
- 8 C558 (R528) The *array-element* shall be a variable.
- 9 C559 (R528) The *scalar-structure-component* shall be a variable.
- 10 C560 (R528) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *sub-*  
11 *script-list*.
- 12 C561 (R528) In an *array-element* or a *scalar-structure-component* that is a *data-i-do-object*, any sub-  
13 script shall be an expression whose primaries are either constants, subobjects of constants, or  
14 DO variables of this *data-implied-do* or the containing *data-implied-dos*, and each operation shall  
15 be intrinsic.
- 16 R530 *data-stmt-value* is [ *data-stmt-repeat* \* ] *data-stmt-constant*  
17 R531 *data-stmt-repeat* is *scalar-int-constant*  
18 or *scalar-int-constant-subobject*
- 19 C562 (R531) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a named con-  
20 stant, it shall have been declared previously in the scoping unit or made accessible by use  
21 association or host association.
- 22 R532 *data-stmt-constant* is *scalar-constant*  
23 or *scalar-constant-subobject*  
24 or *signed-int-literal-constant*  
25 or *signed-real-literal-constant*  
26 or *null-init*  
27 or *structure-constructor*
- 28 C563 (R532) If a DATA statement constant value is a named constant or a structure constructor, the  
29 named constant or derived type shall have been declared previously in the scoping unit or made  
30 accessible by use or host association.
- 31 C564 (R532) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.
- 32 R533 *int-constant-subobject* is *constant-subobject*
- 33 C565 (R533) *int-constant-subobject* shall be of type integer.
- 34 R534 *constant-subobject* is *designator*
- 35 C566 (R534) *constant-subobject* shall be a subobject of a constant.
- 36 C567 (R534) Any subscript, substring starting point, or substring ending point shall be an initializa-  
37 tion expression.
- 38 The *data-stmt-object-list* is expanded to form a sequence of pointers and scalar variables, referred to as  
39 “sequence of variables” in subsequent text. A nonpointer array whose unqualified name appears in a  
40 *data-stmt-object-list* is equivalent to a complete sequence of its array elements in array element order

- 1 (6.2.2.2). An array section is equivalent to the sequence of its array elements in array element order. A  
 2 *data-implied-do* is expanded to form a sequence of array elements and structure components, under the  
 3 control of the *data-i-do-variable*, as in the DO construct (8.1.6.4).
- 4 The *data-stmt-value-list* is expanded to form a sequence of *data-stmt-constants*. A *data-stmt-repeat*  
 5 indicates the number of times the following *data-stmt-constant* is to be included in the sequence; omission  
 6 of a *data-stmt-repeat* has the effect of a repeat factor of 1.
- 7 A zero-sized array or a *data-implied-do* with an iteration count of zero contributes no variables to the  
 8 expanded sequence of variables, but a zero-length scalar character variable does contribute a variable  
 9 to the expanded sequence. A *data-stmt-constant* with a repeat factor of zero contributes no *data-stmt-*  
 10 *constants* to the expanded sequence of scalar *data-stmt-constants*.
- 11 The expanded sequences of variables and *data-stmt-constants* are in one-to-one correspondence. Each  
 12 *data-stmt-constant* specifies the initial value or *null-init* for the corresponding variable. The lengths of  
 13 the two expanded sequences shall be the same.
- 14 A *data-stmt-constant* shall be *null-init* if and only if the corresponding *data-stmt-object* has the POINT-  
 15 ER attribute. The initial association status of a pointer *data-stmt-object* is disassociated.
- 16 A *data-stmt-constant* other than *null-init* shall be compatible with its corresponding variable according  
 17 to the rules of intrinsic assignment (7.4.1.2). The variable is initially defined with the value specified by  
 18 the *data-stmt-constant*; if necessary, the value is converted according to the rules of intrinsic assignment  
 19 (7.4.1.3) to a value that agrees in type, type parameters, and shape with the variable.
- 20 If a *data-stmt-constant* is a *boz-literal-constant*, the corresponding variable shall be of type integer. The  
 21 *boz-literal-constant* is treated as if it were an *int-literal-constant* with a *kind-param* that specifies the  
 22 representation method with the largest decimal exponent range supported by the processor.

**NOTE 5.25**

Examples of DATA statements are:

```

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

```

The character variable NAME is initialized with the value JOHN DOE with padding on the right because the length of the constant is less than the length of the variable. All ten elements of the integer array MILES are initialized to zero. The two-dimensional array SKEW is initialized so that the lower triangle of SKEW is zero and the strict upper triangle is one. The structures MYNAME and YOURNAME are declared using the derived type PERSON from Note 4.17. The pointer HEAD\_OF\_LIST is declared using the derived type NODE from Note 4.36; it is initially disassociated. MYNAME is initialized by a structure constructor. YOURNAME is initialized by supplying a separate value for each component.

### 1 5.2.6 DIMENSION statement

2 R535 *dimension-stmt* is DIMENSION [ :: ] *array-name* ( *array-spec* ) ■  
 3 ■ [ , *array-name* ( *array-spec* ) ] ...

4 This statement specifies the DIMENSION attribute (5.1.2.5) and the array properties for each object  
 5 named.

#### NOTE 5.26

An example of a DIMENSION statement is:

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

### 6 5.2.7 INTENT statement

7 R536 *intent-stmt* is INTENT ( *intent-spec* ) [ :: ] *dummy-arg-name-list*

8 This statement specifies the INTENT attribute (5.1.2.7) for the dummy arguments in the list.

#### NOTE 5.27

An example of an INTENT statement is:

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

### 9 5.2.8 OPTIONAL statement

10 R537 *optional-stmt* is OPTIONAL [ :: ] *dummy-arg-name-list*

11 This statement specifies the OPTIONAL attribute (5.1.2.9) for the dummy arguments in the list.

#### NOTE 5.28

An example of an OPTIONAL statement is:

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

### 12 5.2.9 PARAMETER statement

13 The **PARAMETER statement** specifies the PARAMETER attribute (5.1.2.10) and the values for  
 14 the named constants in the list.

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

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

17 The named constant shall have its type, type parameters, and shape specified in a prior specification of  
 18 the *specification-part* or declared implicitly (5.3). If the named constant is typed by the implicit typing  
 19 rules, its appearance in any subsequent specification of the *specification-part* shall confirm this implied  
 20 type and the values of any implied type parameters.

21 The value of each named constant is that specified by the corresponding initialization expression; if  
 22 necessary, the value is converted according to the rules of intrinsic assignment (7.4.1.3) to a value that  
 23 agrees in type, type parameters, and shape with the named constant.

**NOTE 5.29**

An example of a PARAMETER statement is:

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

**1 5.2.10 POINTER statement**

2 R540 *pointer-stmt*                    **is** POINTER [ :: ] *pointer-decl-list*  
 3 R541 *pointer-decl*                    **is** *object-name* [ ( *deferred-shape-spec-list* ) ]  
 4    **or** *proc-entity-name*

5 C568 (R541) A *proc-entity-name* shall also be declared in a *procedure-declaration-stmt*.

6 This statement specifies the POINTER attribute (5.1.2.11) for a list of objects and procedure entities.

**NOTE 5.30**

An example of a POINTER statement is:

```
TYPE (NODE) :: CURRENT  

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

**7 5.2.11 PROTECTED statement**

8 R542 *protected-stmt*                    **is** PROTECTED [ :: ] *entity-name-list*

9 The **PROTECTED statement** specifies the PROTECTED attribute (5.1.2.12) for a list of entities.

**10 5.2.12 SAVE statement**

11 R543 *save-stmt*                        **is** SAVE [ [ :: ] *saved-entity-list* ]  
 12 R544 *saved-entity*                    **is** *object-name*  
 13    **or** *proc-pointer-name*  
 14    **or** / *common-block-name* /  
 15 R545 *proc-pointer-name*                **is** *name*

16 C569 (R545) A *proc-pointer-name* shall be the name of a procedure pointer.

17 C570 (R543) If a SAVE statement with an omitted saved entity list occurs in a scoping unit, no other  
 18 explicit occurrence of the SAVE attribute or SAVE statement is permitted in the same scoping  
 19 unit.

20 A SAVE statement with a saved entity list specifies the SAVE attribute (5.1.2.13) for all entities named  
 21 in the list or included within a common block named in the list. A SAVE statement without a saved  
 22 entity list is treated as though it contained the names of all allowed items in the same scoping unit.

23 If a particular common block name is specified in a SAVE statement in any scoping unit of a program  
 24 other than the main program, it shall be specified in a SAVE statement in every scoping unit in which  
 25 that common block appears except in the scoping unit of the main program. For a common block  
 26 declared in a SAVE statement, the values in the common block storage sequence (5.5.2.1) at the time a  
 27 RETURN or END statement is executed are made available to the next scoping unit in the execution  
 28 sequence of the program that specifies the common block name or accesses the common block. If a  
 29 named common block is specified in the scoping unit of the main program, the current values of the  
 30 common block storage sequence are made available to each scoping unit that specifies the named common  
 31 block. The definition status of each object in the named common block storage sequence depends on

- 1 the association that has been established for the common block storage sequence.
- 2 A SAVE statement may appear in the specification part of a main program and has no effect.

**NOTE 5.31**

An example of a SAVE statement is:

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

### 3 5.2.13 TARGET statement

- 4 R546 *target-stmt* is TARGET [ :: ] *object-name* [ ( *array-spec* ) ] ■  
 5 ■ [ , *object-name* [ ( *array-spec* ) ] ] ...

- 6 This statement specifies the TARGET attribute (5.1.2.14) for a list of objects.

**NOTE 5.32**

An example of a TARGET statement is:

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

### 7 5.2.14 VALUE statement

- 8 R547 *value-stmt* is VALUE [ :: ] *dummy-arg-name-list*

- 9 The VALUE statement specifies the VALUE attribute (5.1.2.15) for a list of dummy arguments.

### 10 5.2.15 VOLATILE statement

- 11 R548 *volatile-stmt* is VOLATILE [ :: ] *object-name-list*

- 12 The VOLATILE statement specifies the VOLATILE attribute (5.1.2.16) for a list of objects.

## 13 5.3 IMPLICIT statement

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

- 17 R549 *implicit-stmt* is IMPLICIT *implicit-spec-list*  
 18 or IMPLICIT NONE  
 19 R550 *implicit-spec* is *declaration-type-spec* ( *letter-spec-list* )  
 20 R551 *letter-spec* is *letter* [ - *letter* ]

- 21 C571 (R549) If IMPLICIT NONE is specified in a scoping unit, it shall precede any PARAMETER  
 22 statements that appear in the scoping unit and there shall be no other IMPLICIT statements  
 23 in the scoping unit.

- 24 C572 (R551) If the minus and second letter appear, the second letter shall follow the first letter  
 25 alphabetically.

- 26 A *letter-spec* consisting of two letters separated by a minus is equivalent to writing a list containing all  
 27 of the letters in alphabetical order in the alphabetic sequence from the first letter through the second  
 28 letter. For example, A-C is equivalent to A, B, C. The same letter shall not appear as a single letter, or

- 1 be included in a range of letters, more than once in all of the IMPLICIT statements in a scoping unit.
- 2 In each scoping unit, there is a mapping, which may be null, between each of the letters A, B, ..., Z  
 3 and a type (and type parameters). An IMPLICIT statement specifies the mapping for the letters in  
 4 its *letter-spec-list*. IMPLICIT NONE specifies the null mapping for all the letters. If a mapping is not  
 5 specified for a letter, the default for a program unit or an interface body is default integer if the letter  
 6 is I, J, ..., or N and default real otherwise, and the default for an internal or module procedure is the  
 7 mapping in the host scoping unit.
- 8 Any data entity that is not explicitly declared by a type declaration statement, is not an intrinsic  
 9 function, and is not made accessible by use association or host association is declared implicitly to be of  
 10 the type (and type parameters) mapped from the first letter of its name, provided the mapping is not  
 11 null. The mapping for the first letter of the data entity shall either have been established by a prior  
 12 IMPLICIT statement or be the default mapping for the letter. The mapping may be to a derived type  
 13 that is inaccessible in the local scope if the derived type is accessible to the host scope. The data entity  
 14 is treated as if it were declared in an explicit type declaration in the outermost scoping unit in which it  
 15 appears. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement  
 16 for the name of the result variable of that function subprogram.

**NOTE 5.33**

The following are examples of the use of IMPLICIT statements:

```

MODULE EXAMPLE_MODULE
  IMPLICIT NONE
  ...
  INTERFACE
    FUNCTION FUN (I)      ! Not all data entities need to
      INTEGER FUN        ! be declared explicitly
    END FUNCTION FUN
  END INTERFACE
CONTAINS
  FUNCTION JFUN (J)      ! All data entities need to
    INTEGER JFUN, J     ! be declared explicitly.
    ...
  END FUNCTION JFUN
END MODULE EXAMPLE_MODULE
SUBROUTINE SUB
  IMPLICIT COMPLEX (C)
  C = (3.0, 2.0)        ! C is implicitly declared COMPLEX
  ...
CONTAINS
  SUBROUTINE SUB1
    IMPLICIT INTEGER (A, C)
    C = (0.0, 0.0)      ! C is host associated and of
                       ! type complex
    Z = 1.0             ! Z is implicitly declared REAL
    A = 2               ! A is implicitly declared INTEGER
    CC = 1              ! CC is implicitly declared INTEGER
    ...
  END SUBROUTINE SUB1
  SUBROUTINE SUB2
    Z = 2.0             ! Z is implicitly declared REAL and
                       ! is different from the variable of
                       ! the same name in SUB1

```

**NOTE 5.33 (cont.)**

```

...
END SUBROUTINE SUB2
SUBROUTINE SUB3
  USE EXAMPLE_MODULE ! Accesses integer function FUN
                    ! by use association
  Q = FUN (K)        ! Q is implicitly declared REAL and
                    ! K is implicitly declared INTEGER
  ...
END SUBROUTINE SUB3
END SUBROUTINE SUB

```

**NOTE 5.34**

An IMPLICIT statement may specify a *declaration-type-spec* of derived type.

For example, given an IMPLICIT statement and a type defined as follows:

```

IMPLICIT TYPE (POSN) (A-B, W-Z), INTEGER (C-V)
TYPE POSN
  REAL X, Y
  INTEGER Z
END TYPE POSN

```

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

**NOTE 5.35**

The following is an example of a mapping to a derived type that is inaccessible in the local scope:

```

PROGRAM MAIN
  IMPLICIT TYPE(BLOB) (A)
  TYPE BLOB
    INTEGER :: I
  END TYPE BLOB
  TYPE(BLOB) :: B
  CALL STEVE
CONTAINS
  SUBROUTINE STEVE
    INTEGER :: BLOB
    ..
    AA = B
    ..
  END SUBROUTINE STEVE
END PROGRAM MAIN

```

In the subroutine STEVE, it is not possible to explicitly declare a variable to be of type BLOB because BLOB has been given a different meaning, but implicit mapping for the letter A still maps to type BLOB, so AA is of type BLOB.

## 1 5.4 NAMELIST statement

2 A **NAMELIST statement** specifies a group of named data objects, which may be referred to by a  
3 single name for the purpose of data transfer (9.5, 10.10).

4 R552 *namelist-stmt* is NAMELIST ■  
5 ■ / *namelist-group-name* / *namelist-group-object-list* ■  
6 ■ [ [ , ] / *namelist-group-name* / ■  
7 ■ *namelist-group-object-list* ] ...

8 C573 (R552) The *namelist-group-name* shall not be a name made accessible by use association.

9 R553 *namelist-group-object* is *variable-name*

10 C574 (R553) A *namelist-group-object* shall not be an assumed-size array.

11 C575 (R552) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-*  
12 *name* has the PUBLIC attribute.

13 The order in which the variables are specified in the NAMELIST statement determines the order in  
14 which the values appear on output.

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

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

19 A namelist group object shall either be accessed by use or host association or shall have its type, type  
20 parameters, and shape specified by previous specification statements or the procedure heading in the  
21 same scoping unit or by the implicit typing rules in effect for the scoping unit. If a namelist group object  
22 is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall  
23 confirm the implied type and type parameters.

### NOTE 5.36

An example of a NAMELIST statement is:

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

## 24 5.5 Storage association of data objects

25 In general, the physical storage units or storage order for data objects is not specifiable. However,  
26 the EQUIVALENCE, COMMON, and SEQUENCE statements and the BIND(C) *type-attr-spec* provide  
27 for control of the order and layout of storage units. The general mechanism of storage association is  
28 described in 16.4.3.

### 29 5.5.1 EQUIVALENCE statement

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

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

- 1 R554 *equivalence-stmt*            **is** EQUIVALENCE *equivalence-set-list*  
 2 R555 *equivalence-set*            **is** ( *equivalence-object* , *equivalence-object-list* )  
 3 R556 *equivalence-object*        **is** *variable-name*  
 4                                        **or** *array-element*  
 5                                        **or** *substring*
- 6 C576 (R556) An *equivalence-object* shall not be a designator with a base object that is a dummy  
 7 argument, a pointer, an allocatable variable, a derived-type object that has an allocatable ultimate  
 8 component, an object of a nonsequence derived type, an object of a derived type that has  
 9 a pointer at any level of component selection, an automatic object, a function name, an entry  
 10 name, a result name, a variable with the BIND attribute, a variable in a common block that  
 11 has the BIND attribute, or a named constant.
- 12 C577 (R556) An *equivalence-object* shall not be a designator that has more than one *part-ref*.
- 13 C578 (R556) An *equivalence-object* shall not have the TARGET attribute.
- 14 C579 (R556) Each subscript or substring range expression in an *equivalence-object* shall be an integer  
 15 initialization expression (7.1.7).
- 16 C580 (R555) If an *equivalence-object* is of type default integer, default real, double precision real,  
 17 default complex, default logical, or numeric sequence type, all of the objects in the equivalence  
 18 set shall be of these types.
- 19 C581 (R555) If an *equivalence-object* is of type default character or character sequence type, all of the  
 20 objects in the equivalence set shall be of these types.
- 21 C582 (R555) If an *equivalence-object* is of a sequence derived type that is not a numeric sequence or  
 22 character sequence type, all of the objects in the equivalence set shall be of the same type with  
 23 the same type parameter values.
- 24 C583 (R555) If an *equivalence-object* is of an intrinsic type other than default integer, default real,  
 25 double precision real, default complex, default logical, or default character, all of the objects in  
 26 the equivalence set shall be of the same type with the same kind type parameter value.
- 27 C584 (R556) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equiv-  
 28 alence set shall have the PROTECTED attribute.
- 29 C585 (R556) The name of an *equivalence-object* shall not be a name made accessible by use association.
- 30 C586 (R556) A *substring* shall not have length zero.

**NOTE 5.37**

The EQUIVALENCE statement allows the equivalencing of sequence structures and the equivalencing of objects of intrinsic type with nondefault type parameters, but there are strict rules regarding the appearance of these objects in an EQUIVALENCE statement.

A structure that appears in an EQUIVALENCE statement shall be a sequence structure. If a sequence structure is not of numeric sequence type or of character sequence type, it shall be equivalenced only to objects of the same type with the same type parameter values.

A structure of a numeric sequence type may be equivalenced to another structure of a numeric sequence type, an object of default integer type, default real type, double precision real type, default complex type, or default logical type such that components of the structure ultimately become associated only with objects of these types.

A structure of a character sequence type may be equivalenced to an object of default character

**NOTE 5.37 (cont.)**

type or another structure of a character sequence type.

An object of intrinsic type with nondefault kind type parameters may be equivalenced only to objects of the same type and kind type parameters.

Further rules on the interaction of EQUIVALENCE statements and default initialization are given in [16.4.3.3](#).

**1 5.5.1.1 Equivalence association**

2 An EQUIVALENCE statement specifies that the storage sequences ([16.4.3.1](#)) of the data objects specified  
 3 in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the *equivalence-set*, if  
 4 any, have the same first storage unit, and all of the zero-sized sequences in the *equivalence-set*, if any,  
 5 are storage associated with one another and with the first storage unit of any nonzero-sized sequences.  
 6 This causes the storage association of the data objects in the *equivalence-set* and may cause storage  
 7 association of other data objects.

**8 5.5.1.2 Equivalence of default character objects**

9 A data object of type default character may be equivalenced only with other objects of type default  
 10 character. The lengths of the equivalenced objects need not be the same.

11 An EQUIVALENCE statement specifies that the storage sequences of all the default character data  
 12 objects specified in an *equivalence-set* are storage associated. All of the nonzero-sized sequences in the  
 13 *equivalence-set*, if any, have the same first character storage unit, and all of the zero-sized sequences in  
 14 the *equivalence-set*, if any, are storage associated with one another and with the first character storage  
 15 unit of any nonzero-sized sequences. This causes the storage association of the data objects in the  
 16 *equivalence-set* and may cause storage association of other data objects.

**NOTE 5.38**

For example, using the declarations:

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

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

```

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

**17 5.5.1.3 Array names and array element designators**

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

#### 1 5.5.1.4 Restrictions on EQUIVALENCE statements

- 2 An EQUIVALENCE statement shall not specify that the same storage unit is to occur more than once  
3 in a storage sequence.

##### NOTE 5.39

For example:

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

is prohibited, because it would specify the same storage unit for A (1) and A (2).

- 4 An EQUIVALENCE statement shall not specify that consecutive storage units are to be nonconsecutive.  
5

##### NOTE 5.40

For example, the following is prohibited:

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

#### 6 5.5.2 COMMON statement

- 7 The **COMMON statement** specifies blocks of physical storage, called **common blocks**, that may be  
8 accessed by any of the scoping units in a program. Thus, the COMMON statement provides a global  
9 data facility based on storage association (16.4.3).

- 10 The common blocks specified by the COMMON statement may be named and are called **named com-**  
11 **mon blocks**, or may be unnamed and are called **blank common**.

12 R557 *common-stmt* is COMMON ■  
13 ■ [ / [ *common-block-name* ] / ] *common-block-object-list* ■  
14 ■ [ [ , ] / [ *common-block-name* ] / ■  
15 ■ *common-block-object-list* ] ...

16 R558 *common-block-object* is *variable-name* [ ( *explicit-shape-spec-list* ) ]  
17 or *proc-pointer-name*

- 18 C587 (R558) Only one appearance of a given *variable-name* or *proc-pointer-name* is permitted in all  
19 *common-block-object-lists* within a scoping unit.

- 20 C588 (R558) A *common-block-object* shall not be a dummy argument, an allocatable variable, a  
21 derived-type object with an ultimate component that is allocatable, an automatic object, a  
22 function name, an entry name, a variable with the BIND attribute, or a result name.

- 23 C589 (R558) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.1) or a  
24 type with the BIND attribute and it shall have no default initialization.

- 25 C590 (R558) A *variable-name* or *proc-pointer-name* shall not be a name made accessible by use  
26 association.

- 27 In each COMMON statement, the data objects whose names appear in a common block object list  
28 following a common block name are declared to be in that common block. If the first common block

1 name is omitted, all data objects whose names appear in the first common block object list are specified to  
2 be in blank common. Alternatively, the appearance of two slashes with no common block name between  
3 them declares the data objects whose names appear in the common block object list that follows to be  
4 in blank common.

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

10 The form *variable-name (explicit-shape-spec-list)* declares *variable-name* to have the DIMENSION at-  
11 tribute and specifies the array properties that apply. If derived-type objects of numeric sequence type  
12 (4.5.1) or character sequence type (4.5.1) appear in common, it is as if the individual components were  
13 enumerated directly in the common list.

#### NOTE 5.41

Examples of COMMON statements are:

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

#### 14 5.5.2.1 Common block storage sequence

15 For each common block in a scoping unit, a **common block storage sequence** is formed as follows:

- 16 (1) A storage sequence is formed consisting of the sequence of storage units in the storage  
17 sequences (16.4.3.1) of all data objects in the common block object lists for the common  
18 block. The order of the storage sequences is the same as the order of the appearance of the  
19 common block object lists in the scoping unit.
- 20 (2) The storage sequence formed in (1) is extended to include all storage units of any storage  
21 sequence associated with it by equivalence association. The sequence may be extended only  
22 by adding storage units beyond the last storage unit. Data objects associated with an entity  
23 in a common block are considered to be in that common block.

24 Only COMMON statements and EQUIVALENCE statements appearing in the scoping unit contribute  
25 to common block storage sequences formed in that unit.

#### 26 5.5.2.2 Size of a common block

27 The **size of a common block** is the size of its common block storage sequence, including any extensions  
28 of the sequence resulting from equivalence association.

#### 29 5.5.2.3 Common association

30 Within a program, the common block storage sequences of all nonzero-sized common blocks with the  
31 same name have the same first storage unit, and the common block storage sequences of all zero-sized  
32 common blocks with the same name are storage associated with one another. Within a program, the  
33 common block storage sequences of all nonzero-sized blank common blocks have the same first storage  
34 unit and the storage sequences of all zero-sized blank common blocks are associated with one another and  
35 with the first storage unit of any nonzero-sized blank common blocks. This results in the association of  
36 objects in different scoping units. Use association or host association may cause these associated objects  
37 to be accessible in the same scoping unit.

38 A nonpointer object of default integer type, default real type, double precision real type, default complex

- 1 type, default logical type, or numeric sequence type shall be associated only with nonpointer objects of  
2 these types.
- 3 A nonpointer object of type default character or character sequence type shall be associated only with  
4 nonpointer objects of these types.
- 5 A nonpointer object of a derived type that is not a numeric sequence or character sequence type shall  
6 be associated only with nonpointer objects of the same type with the same type parameter values.
- 7 A nonpointer object of intrinsic type other than default integer, default real, double precision real,  
8 default complex, default logical, or default character shall be associated only with nonpointer objects of  
9 the same type and type parameters.
- 10 A data pointer shall be storage associated only with data pointers of the same type and rank. Data  
11 pointers that are storage associated shall have deferred the same type parameters; corresponding non-  
12 deferred type parameters shall have the same value. A procedure pointer shall be storage associated  
13 only with another procedure pointer; either both interfaces shall be explicit or both interfaces shall be  
14 implicit. If the interfaces are explicit, the characteristics shall be the same. If the interfaces are implicit,  
15 either both shall be subroutines or both shall be functions with the same type and type parameters.
- 16 An object with the TARGET attribute may be storage associated only with another object that has the  
17 TARGET attribute and the same type and type parameters.

**NOTE 5.42**

A common block is permitted to contain sequences of different storage units, provided each scoping unit that accesses the common block specifies an identical sequence of storage units for the common block. For example, this allows a single common block to contain both numeric and character storage units.

Association in different scoping units between objects of default type, objects of double precision real type, and sequence structures is permitted according to the rules for equivalence objects (5.5.1).

**18 5.5.2.4 Differences between named common and blank common**

19 A blank common block has the same properties as a named common block, except for the following:

- 20 (1) Execution of a RETURN or END statement may cause data objects in a named common  
21 block to become undefined unless the common block name has been declared in a SAVE  
22 statement, but never causes data objects in blank common to become undefined (16.5.6).
- 23 (2) Named common blocks of the same name shall be of the same size in all scoping units of a  
24 program in which they appear, but blank common blocks may be of different sizes.
- 25 (3) A data object in a named common block may be initially defined by means of a DATA  
26 statement or type declaration statement in a block data program unit (11.3), but objects in  
27 blank common shall not be initially defined.

**28 5.5.2.5 Restrictions on common and equivalence**

29 An EQUIVALENCE statement shall not cause the storage sequences of two different common blocks to  
30 be associated.

31 Equivalence association shall not cause a common block storage sequence to be extended by adding  
32 storage units preceding the first storage unit of the first object specified in a COMMON statement for  
33 the common block.

**NOTE 5.43**

For example, the following is not permitted:

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

- 1 Equivalence association shall not cause a derived-type object with default initialization to be associated
- 2 with an object in a common block.



## 1 Section 6: Use of data objects

2 The appearance of a data object designator in a context that requires its value is termed a reference. A  
 3 reference is permitted only if the data object is defined. A reference to a pointer is permitted only if the  
 4 pointer is associated with a target object that is defined. A data object becomes defined with a value  
 5 when events described in 16.5.5 occur.

6 R601 *variable* **is** *designator*

7 C601 (R601) *designator* shall not be a constant or a subobject of a constant.

8 R602 *variable-name* **is** *name*

9 C602 (R602) A *variable-name* shall be the name of a variable.

10 R603 *designator* **is** *object-name*  
 11 **or** *array-element*  
 12 **or** *array-section*  
 13 **or** *structure-component*  
 14 **or** *substring*  
 15 R604 *logical-variable* **is** *variable*

16 C603 (R604) *logical-variable* shall be of type logical.

17 R605 *default-logical-variable* **is** *variable*

18 C604 (R605) *default-logical-variable* shall be of type default logical.

19 R606 *char-variable* **is** *variable*

20 C605 (R606) *char-variable* shall be of type character.

21 R607 *default-char-variable* **is** *variable*

22 C606 (R607) *default-char-variable* shall be of type default character.

23 R608 *int-variable* **is** *variable*

24 C607 (R608) *int-variable* shall be of type integer.

### NOTE 6.1

For example, given the declarations:

```
CHARACTER (10) A, B (10)
TYPE (PERSON) P ! See Note 4.17
```

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

25 A constant (3.2.2) is a literal constant or a named constant. A literal constant is a scalar denoted by a  
 26 syntactic form, which indicates its type, type parameters, and value. A named constant is a constant  
 27 that has a name; the name has the PARAMETER attribute (5.1.2.10, 5.2.9). A reference to a constant  
 28 is always permitted; redefinition of a constant is never permitted.

## 1 6.1 Scalars

2 A **scalar** (2.4.4) is a data entity that can be represented by a single value of the type and that is not an  
 3 array (6.2). Its value, if defined, is a single element from the set of values that characterize its type.

### NOTE 6.2

A scalar object of derived type has a single value that consists of the values of its components (4.5.7).

4 A scalar has rank zero.

### 5 6.1.1 Substrings

6 A **substring** is a contiguous portion of a character string (4.4.4). The following rules define the forms  
 7 of a substring:

8 R609	<i>substring</i>	<b>is</b>	<i>parent-string</i> ( <i>substring-range</i> )
9 R610	<i>parent-string</i>	<b>is</b>	<i>scalar-variable-name</i>
10		<b>or</b>	<i>array-element</i>
11		<b>or</b>	<i>scalar-structure-component</i>
12		<b>or</b>	<i>scalar-constant</i>
13 R611	<i>substring-range</i>	<b>is</b>	[ <i>scalar-int-expr</i> ] : [ <i>scalar-int-expr</i> ]

14 C608 (R610) *parent-string* shall be of type character.

15 The value of the first *scalar-int-expr* in *substring-range* is called the **starting point** and the value of  
 16 the second one is called the **ending point**. The length of a substring is the number of characters in the  
 17 substring and is  $\text{MAX}(l - f + 1, 0)$ , where  $f$  and  $l$  are the starting and ending points, respectively.

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

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

### NOTE 6.3

Examples of character substrings are:

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

### 25 6.1.2 Structure components

26 A **structure component** is part of an object of derived type; it may be referenced by an object  
 27 designator. A structure component may be a scalar or an array.

- 1 R612 *data-ref* is *part-ref* [ % *part-ref* ] ...  
 2 R613 *part-ref* is *part-name* [ ( *section-subscript-list* ) ]
- 3 C609 (R612) Each *part-name* except the rightmost shall be of derived type.
- 4 C610 (R612) Each *part-name* except the leftmost shall be the name of a component of the declared  
 5 type of the preceding *part-name*.
- 6 C611 (R612) If the rightmost *part-name* is of abstract type, *data-ref* shall be polymorphic.
- 7 C612 (R612) The leftmost *part-name* shall be the name of a data object.
- 8 C613 (R613) If a *section-subscript-list* appears, the number of *section-subscripts* shall equal the rank  
 9 of *part-name*.
- 10 The rank of a *part-ref* of the form *part-name* is the rank of *part-name*. The rank of a *part-ref* that has  
 11 a section subscript list is the number of subscript triplets and vector subscripts in the list.
- 12 C614 (R612) There shall not be more than one *part-ref* with nonzero rank. A *part-name* to the right  
 13 of a *part-ref* with nonzero rank shall not have the ALLOCATABLE or POINTER attribute.
- 14 The rank of a *data-ref* is the rank of the *part-ref* with nonzero rank, if any; otherwise, the rank is zero.  
 15 The **base object** of a *data-ref* is the data object whose name is the leftmost part name.
- 16 The type and type parameters, if any, of a *data-ref* are those of the rightmost part name.
- 17 A *data-ref* with more than one *part-ref* is a subobject of its base object if none of the *part-names*,  
 18 except for possibly the rightmost, are pointers. If the rightmost *part-name* is the only pointer, then the  
 19 *data-ref* is a subobject of its base object in contexts that pertain to its pointer association status but  
 20 not in any other contexts.

**NOTE 6.4**

If X is an object of derived type with a pointer component P, then the pointer X%P is a subobject of X when considered as a pointer – that is in contexts where it is not dereferenced.

However the target of X%P is not a subobject of X. Thus, in contexts where X%P is dereferenced to refer to the target, it is not a subobject of X.

- 21 R614 *structure-component* is *data-ref*
- 22 C615 (R614) There shall be more than one *part-ref* and the rightmost *part-ref* shall be of the form  
 23 *part-name*.
- 24 A structure component shall be neither referenced nor defined before the declaration of the base object.  
 25 A structure component is a pointer only if the rightmost part name is defined to have the POINTER  
 26 attribute.

**NOTE 6.5**

Examples of structure components are:

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

For a more elaborate example see [C.3.1](#).

**NOTE 6.6**

The syntax rules are structured such that a *data-ref* that ends in a component name without a following subscript list is a structure component, even when other component names in the *data-ref* are followed by a subscript list. A *data-ref* that ends in a component name with a following subscript list is either an array element or an array section. A *data-ref* of nonzero rank that ends with a *substring-range* is an array section. A *data-ref* of zero rank that ends with a *substring-range* is a substring.

1 A **subcomponent** of an object of derived type is a component of that object or of a subobject of that  
2 object.

### 3 **6.1.3 Type parameter inquiry**

4 A **type parameter inquiry** is used to inquire about a type parameter of a data object. It applies to  
5 both intrinsic and derived types.

6 R615 *type-param-inquiry*            **is** *designator* %<sub>0</sub> *type-param-name*

7 C616 (R615) The *type-param-name* shall be the name of a type parameter of the declared type of the  
8 object designated by the *designator*.

9 A deferred type parameter of a pointer that is not associated or of an unallocated allocatable variable  
10 shall not be inquired about.

**NOTE 6.7**

A *type-param-inquiry* has a syntax like that of a structure component reference, but it does not have the same semantics. It is not a variable and thus can never be assigned to. It may be used only as a primary in an expression. It is scalar even if *designator* is an array.

The intrinsic type parameters can also be inquired about by using the intrinsic functions KIND and LEN.

**NOTE 6.8**

The following are examples of type parameter inquiries:

```

a%kind      !-- A is real.  Same value as KIND(a).
s%len       !-- S is character.  Same value as LEN(s).
b(10)%kind  !-- Inquiry about an array element.
p%dim       !-- P is of the derived type general_point.
```

See Note 4.24 for the definition of the `general_point` type used in the last example above.

## 11 **6.2 Arrays**

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

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

### 1 6.2.1 Whole arrays

2 A **whole array** is a named array, which may be either a named constant (5.1.2.10, 5.2.9) or a variable;  
3 no subscript list is appended to the name.

4 The appearance of a whole array variable in an executable construct specifies all the elements of the  
5 array (2.4.5). An assumed-size array is permitted to appear as a whole array in an executable construct  
6 only as an actual argument in a procedure reference that does not require the shape.

7 The appearance of a whole array name in a nonexecutable statement specifies the entire array except  
8 for the appearance of a whole array name in an equivalence set (5.5.1.3).

### 9 6.2.2 Array elements and array sections

10 R616 *array-element* is *data-ref*

11 C617 (R616) Every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.

12 R617 *array-section* is *data-ref* [ ( *substring-range* ) ]

13 C618 (R617) Exactly one *part-ref* shall have nonzero rank, and either the final *part-ref* shall have a  
14 *section-subscript-list* with nonzero rank or another *part-ref* shall have nonzero rank.

15 C619 (R617) If a *substring-range* appears, the rightmost *part-name* shall be of type character.

16 R618 *subscript* is *scalar-int-expr*

17 R619 *section-subscript* is *subscript*

18 or *subscript-triplet*

19 or *vector-subscript*

20 R620 *subscript-triplet* is [ *subscript* ] : [ *subscript* ] [ : *stride* ]

21 R621 *stride* is *scalar-int-expr*

22 R622 *vector-subscript* is *int-expr*

23 C620 (R622) A *vector-subscript* shall be an integer array expression of rank one.

24 C621 (R620) The second subscript shall not be omitted from a *subscript-triplet* in the last dimension  
25 of an assumed-size array.

26 An array element is a scalar. An array section is an array. If a *substring-range* is present in an *array-*  
27 *section*, each element is the designated substring of the corresponding element of the array section.

#### NOTE 6.9

For example, with the declarations:

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

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

#### NOTE 6.10

Unless otherwise specified, an array element or array section does not have an attribute of the whole array. In particular, an array element or an array section does not have the POINTER or ALLOCATABLE attribute.

**NOTE 6.11**

Examples of array elements and array sections are:

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

1 **6.2.2.1 Array elements**

2 The value of a subscript in an array element shall be within the bounds for that dimension.

3 **6.2.2.2 Array element order**

4 The elements of an array form a sequence known as the **array element order**. The position of an array  
 5 element in this sequence is determined by the subscript order value of the subscript list designating the  
 6 element. The subscript order value is computed from the formulas in Table 6.1.

Table 6.1: **Subscript order value**

Rank	Subscript bounds	Subscript list	Subscript order value
1	$j_1:k_1$	$s_1$	$1 + (s_1 - j_1)$
2	$j_1:k_1, j_2:k_2$	$s_1, s_2$	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$
3	$j_1:k_1, j_2:k_2, j_3:k_3$	$s_1, s_2, s_3$	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$
.	.	.	.
.	.	.	.
.	.	.	.
7	$j_1:k_1, \dots, j_7:k_7$	$s_1, \dots, s_7$	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$ $+ \dots$ $+ (s_7 - j_7) \times d_6$ $\times d_5 \times \dots \times d_1$
Notes for Table 6.1:			
1) $d_i = \max(k_i - j_i + 1, 0)$ is the size of the $i$ th dimension.			
2) If the size of the array is nonzero, $j_i \leq s_i \leq k_i$ for all $i = 1, 2, \dots, 7$ .			

7 **6.2.2.3 Array sections**

8 An **array section** is an array subobject optionally followed by a substring range.

9 In an *array-section* having a *section-subscript-list*, each *subscript-triplet* and *vector-subscript* in the  
 10 section subscript list indicates a sequence of subscripts, which may be empty. Each subscript in such a  
 11 sequence shall be within the bounds for its dimension unless the sequence is empty. The array section is  
 12 the set of elements from the array determined by all possible subscript lists obtainable from the single  
 13 subscripts or sequences of subscripts specified by each section subscript.

14 In an *array-section* with no *section-subscript-list*, the rank and shape of the array is the rank and shape  
 15 of the *part-ref* with nonzero rank; otherwise, the rank of the array section is the number of subscript  
 16 triplets and vector subscripts in the section subscript list. The shape is the rank-one array whose  $i$ th

1 element is the number of integer values in the sequence indicated by the  $i$ th subscript triplet or vector  
 2 subscript. If any of these sequences is empty, the array section has size zero. The subscript order of the  
 3 elements of an array section is that of the array data object that the array section represents.

#### 4 **6.2.2.3.1 Subscript triplet**

5 A subscript triplet designates a regular sequence of subscripts consisting of zero or more subscript values.  
 6 The third expression in the subscript triplet is the increment between the subscript values and is called  
 7 the **stride**. The subscripts and stride of a subscript triplet are optional. An omitted first subscript in a  
 8 subscript triplet is equivalent to a subscript whose value is the lower bound for the array and an omitted  
 9 second subscript is equivalent to the upper bound. An omitted stride is equivalent to a stride of 1.

10 The stride shall not be zero.

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

##### **NOTE 6.12**

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

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

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

##### **NOTE 6.13**

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

##### **NOTE 6.14**

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

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

#### 18 **6.2.2.3.2 Vector subscript**

19 A **vector subscript** designates a sequence of subscripts corresponding to the values of the elements  
 20 of the expression. Each element of the expression shall be defined. A **many-one array section** is an  
 21 array section with a vector subscript having two or more elements with the same value. A many-one  
 22 array section shall appear neither on the left of the equals in an assignment statement nor as an input  
 23 item in a READ statement.

24 An array section with a vector subscript shall not be argument associated with a dummy array that  
 25 is defined or redefined. An array section with a vector subscript shall not be the target in a pointer

1 assignment statement. An array section with a vector subscript shall not be an internal file.

**NOTE 6.15**

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

U = (/ 1, 3, 2 /)  
V = (/ 2, 1, 1, 3 /)

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

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

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

Z (1, 2)    Z (3, 2)    Z (2, 2)

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

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

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

## 2 6.3 Dynamic association

3 Dynamic control over the allocation, association, and deallocation of pointer targets is provided by  
4 the ALLOCATE, NULLIFY, and DEALLOCATE statements and pointer assignment. ALLOCATE  
5 (6.3.1) creates targets for pointers; pointer assignment (7.4.2) associates pointers with existing targets;  
6 NULLIFY (6.3.2) disassociates pointers from targets, and DEALLOCATE (6.3.3) deallocates targets.  
7 Dynamic association applies to scalars and arrays of any type.

8 The ALLOCATE and DEALLOCATE statements also are used to create and deallocate variables with  
9 the ALLOCATABLE attribute.

**NOTE 6.16**

Detailed remarks regarding pointers and dynamic association are in C.3.3.

### 10 6.3.1 ALLOCATE statement

11 The **ALLOCATE statement** dynamically creates pointer targets and allocatable variables.

12 R623	<i>allocate-stmt</i>	<b>is</b> ALLOCATE ( [ <i>type-spec</i> :: ] <i>allocation-list</i> ■ ■ [ , <i>alloc-opt-list</i> ] )
13		
14 R624	<i>alloc-opt</i>	<b>is</b> STAT = <i>stat-variable</i> <b>or</b> ERRMSG = <i>errmsg-variable</i> <b>or</b> SOURCE = <i>source-expr</i>
15		
16		
17 R625	<i>stat-variable</i>	<b>is</b> <i>scalar-int-variable</i>
18 R626	<i>errmsg-variable</i>	<b>is</b> <i>scalar-default-char-variable</i>
19 R627	<i>source-expr</i>	<b>is</b> <i>expr</i>
20 R628	<i>allocation</i>	<b>is</b> <i>allocate-object</i> [ ( <i>allocate-shape-spec-list</i> ) ]

- 1 R629 *allocate-object* **is** *variable-name*  
 2 **or** *structure-component*
- 3 R630 *allocate-shape-spec* **is** [ *lower-bound-expr* : ] *upper-bound-expr*  
 4 R631 *lower-bound-expr* **is** *scalar-int-expr*  
 5 R632 *upper-bound-expr* **is** *scalar-int-expr*
- 6 C622 (R629) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.
- 7 C623 (R623) If any *allocate-object* in the statement has a deferred type parameter, either *type-spec* or  
 8 SOURCE= shall appear.
- 9 C624 (R623) If a *type-spec* appears, it shall specify a type with which each *allocate-object* is type  
 10 compatible.
- 11 C625 (R623) If any *allocate-object* is unlimited polymorphic, either *type-spec* or SOURCE= shall  
 12 appear.
- 13 C626 (R623) A *type-param-value* in a *type-spec* shall be an asterisk if and only if each *allocate-object*  
 14 is a dummy argument for which the corresponding type parameter is assumed.
- 15 C627 (R623) If a *type-spec* appears, the kind type parameter values of each *allocate-object* shall be  
 16 the same as the corresponding type parameter values of the *type-spec*.
- 17 C628 (R628) An *allocate-shape-spec-list* shall appear if and only if the *allocate-object* is an array.
- 18 C629 (R628) The number of *allocate-shape-specs* in an *allocate-shape-spec-list* shall be the same as  
 19 the rank of the *allocate-object*.
- 20 C630 (R624) No *alloc-opt* shall appear more than once in a given *alloc-opt-list*.
- 21 C631 (R623) If SOURCE= appears, *type-spec* shall not appear and *allocation-list* shall contain only  
 22 one *allocate-object*, which shall be type compatible (5.1.1.2) with *source-expr*.
- 23 C632 (R623) The *source-expr* shall be a scalar or have the same rank as *allocate-object*.
- 24 C633 (R623) Corresponding kind type parameters of *allocate-object* and *source-expr* shall have the  
 25 same values.
- 26 An *allocate-object* or a bound or type parameter of an *allocate-object* shall not depend on the value of  
 27 *stat-variable*, the value of *errmsg-variable*, or on the value, bounds, length type parameters, allocation  
 28 status, or association status of any *allocate-object* in the same ALLOCATE statement.
- 29 Neither *stat-variable*, *source-expr*, nor *errmsg-variable* shall be allocated within the ALLOCATE state-  
 30 ment in which it appears; nor shall they depend on the value, bounds, length type parameters, allocation  
 31 status, or association status of any *allocate-object* in the same ALLOCATE statement.
- 32 The optional *type-spec* specifies the dynamic type and type parameters of the objects to be allocated. If  
 33 a *type-spec* is specified, allocation of a polymorphic object allocates an object with the specified dynamic  
 34 type; if a *source-expr* is specified, the allocation allocates an object whose dynamic type and type  
 35 parameters are the same as those of the *source-expr*; otherwise it allocates an object with a dynamic  
 36 type the same as its declared type.
- 37 When an ALLOCATE statement having a *type-spec* is executed, any *type-param-values* in the *type-spec*  
 38 specify the type parameters. If the value specified for a type parameter differs from a corresponding  
 39 nondeferred value specified in the declaration of any of the *allocate-objects* then an error condition occurs.
- 40 If a *type-param-value* in a *type-spec* in an ALLOCATE statement is an asterisk, it denotes the current  
 41 value of that assumed type parameter. If it is an expression, subsequent redefinition or undefinition of

1 any entity in the expression does not affect the type parameter value.

**NOTE 6.17**

An example of an ALLOCATE statement is:

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

2 When an ALLOCATE statement is executed for an array, the values of the lower bound and upper  
3 bound expressions determine the bounds of the array. Subsequent redefinition or undefinition of any  
4 entities in the bound expressions do not affect the array bounds. If the lower bound is omitted, the  
5 default value is 1. If the upper bound is less than the lower bound, the extent in that dimension is zero  
6 and the array has zero size.

**NOTE 6.18**

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

7 If SOURCE= appears, *source-expr* shall be conformable (2.4.5) with *allocation*. If the value of a non-  
8 deferred length type parameter of *allocate-object* is different from the value of the corresponding type  
9 parameter of *source-expr*, an error condition occurs. If the allocation is successful, the value of *allocate-*  
10 *object* becomes that of *source-expr*.

**NOTE 6.19**

An example of an ALLOCATE statement in which the value and dynamic type are determined by  
reference to another object is:

```
CLASS(*), ALLOCATABLE :: NEW
CLASS(*), POINTER :: OLD
! ...
ALLOCATE (NEW, SOURCE=OLD) ! Allocate NEW with the value and dynamic type of OLD
```

A more extensive example is given in C.3.2.

11 If the STAT= specifier appears, successful execution of the ALLOCATE statement causes the *stat-*  
12 *variable* to become defined with a value of zero. If an error condition occurs during the execution  
13 of the ALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent positive  
14 integer value and each *allocate-object* will have a processor-dependent status; each *allocate-object* that  
15 was successfully allocated shall have an allocation status of allocated or a pointer association status of  
16 associated; each *allocate-object* that was not successfully allocated shall retain its previous allocation  
17 status or pointer association status.

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

20 The ERRMSG= specifier is described in 6.3.1.3.

21 **6.3.1.1 Allocation of allocatable variables**

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

- 24 (1) The status of an allocatable variable becomes **allocated** if it is allocated by an ALLOCATE  
25 statement, if it is allocated during assignment, or if it is given that status by the allocation  
26 transfer procedure (13.5.16). An allocatable variable with this status may be referenced,  
27 defined, or deallocated; allocating it causes an error condition in the ALLOCATE statement.

1           The intrinsic function ALLOCATED (13.7.9) returns true for such a variable.  
 2           (2) An allocatable variable has a status of **unallocated** if it is not allocated. The status of  
 3           an allocatable variable becomes unallocated if it is deallocated (6.3.3) or if it is given that  
 4           status by the allocation transfer procedure. An allocatable variable with this status shall  
 5           not be referenced or defined. It shall not be supplied as an actual argument corresponding  
 6           to a nonallocatable dummy argument, except to certain intrinsic inquiry functions. It may  
 7           be allocated with the ALLOCATE statement. Deallocating it causes an error condition in  
 8           the DEALLOCATE statement. The intrinsic function ALLOCATED (13.7.9) returns false  
 9           for such a variable.

10 At the beginning of execution of a program, allocatable variables are unallocated.

11 A saved allocatable object has an initial status of unallocated. The status may change during the  
 12 execution of the program.

13 When the allocation status of an allocatable variable changes, the allocation status of any associated  
 14 allocatable variable changes accordingly. Allocation of an allocatable variable establishes values for the  
 15 deferred type parameters of all associated allocatable variables.

16 An unsaved allocatable object that is a local variable of a procedure has a status of unallocated at the  
 17 beginning of each invocation of the procedure. The status may change during execution of the procedure.

18 An unsaved allocatable object that is a local variable of a module or a subobject thereof has an initial  
 19 status of unallocated. The status may change during execution of the program.

20 When an object of derived type is created by an ALLOCATE statement, any allocatable ultimate  
 21 components have an allocation status of unallocated.

#### 22 **6.3.1.2 Allocation of pointer targets**

23 Allocation of a pointer creates an object that implicitly has the TARGET attribute. Following successful  
 24 execution of an ALLOCATE statement for a pointer, the pointer is associated with the target and may  
 25 be used to reference or define the target. Additional pointers may become associated with the pointer  
 26 target or a part of the pointer target by pointer assignment. It is not an error to allocate a pointer  
 27 that is already associated with a target. In this case, a new pointer target is created as required by the  
 28 attributes of the pointer and any array bounds, type, and type parameters specified by the ALLOCATE  
 29 statement. The pointer is then associated with this new target. Any previous association of the pointer  
 30 with a target is broken. If the previous target had been created by allocation, it becomes inaccessible  
 31 unless other pointers are associated with it. The ASSOCIATED intrinsic function (13.7.13) may be used  
 32 to determine whether a pointer that does not have undefined association status is associated.

33 At the beginning of execution of a function whose result is a pointer, the association status of the result  
 34 pointer is undefined. Before such a function returns, it shall either associate a target with this pointer  
 35 or cause the association status of this pointer to become defined as disassociated.

#### 36 **6.3.1.3 ERRMSG= specifier**

37 If an error condition occurs during execution of an ALLOCATE or DEALLOCATE statement, the  
 38 processor shall assign an explanatory message to *errmsg-variable*. If no such condition occurs, the  
 39 processor shall not change the value of *errmsg-variable*.

#### 40 **6.3.2 NULLIFY statement**

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

42 R633    *nullify-stmt*                    **is** NULLIFY ( *pointer-object-list* )  
 43 R634    *pointer-object*                **is** *variable-name*

- 1                                    **or** *structure-component*  
 2                                    **or** *proc-pointer-name*

3 C634 (R634) Each *pointer-object* shall have the POINTER attribute.

4 A *pointer-object* shall not depend on the value, bounds, or association status of another *pointer-object*  
 5 in the same NULLIFY statement.

**NOTE 6.20**

When a NULLIFY statement is applied to a polymorphic pointer (5.1.1.2), its dynamic type becomes the declared type.

6 **6.3.3 DEALLOCATE statement**

7 The **DEALLOCATE statement** causes allocatable variables to be deallocated; it causes pointer tar-  
 8 gets to be deallocated and the pointers to be disassociated.

9 R635    *deallocate-stmt*                    **is** DEALLOCATE ( *allocate-object-list* [ , *dealloc-opt-list* ] )

10 C635 (R635) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.

11 R636    *dealloc-opt*                        **is** STAT = *stat-variable*  
 12    **or** ERRMSG = *errmsg-variable*

13 C636 (R636) No *dealloc-opt* shall appear more than once in a given *dealloc-opt-list*.

14 An *allocate-object* shall not depend on the value, bounds, allocation status, or association status of  
 15 another *allocate-object* in the same DEALLOCATE statement; it also shall not depend on the value of  
 16 the *stat-variable* or *errmsg-variable* in the same DEALLOCATE statement.

17 Neither *stat-variable* nor *errmsg-variable* shall be deallocated within the same DEALLOCATE state-  
 18 ment; they also shall not depend on the value, bounds, allocation status, or association status of any  
 19 *allocate-object* in the same DEALLOCATE statement.

20 If the STAT= specifier appears, successful execution of the DEALLOCATE statement causes the *stat-*  
 21 *variable* to become defined with a value of zero. If an error condition occurs during the execution of  
 22 the DEALLOCATE statement, the *stat-variable* becomes defined with a processor-dependent positive  
 23 integer value and each *allocate-object* that was successfully deallocated shall have an allocation status of  
 24 unallocated or a pointer association status of disassociated. Each *allocate-object* that was not successfully  
 25 deallocated shall retain its previous allocation status or pointer association status.

**NOTE 6.21**

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

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

28 The ERRMSG= specifier is described in 6.3.1.3.

**NOTE 6.22**

An example of a DEALLOCATE statement is:

DEALLOCATE (X, B)

### 1 6.3.3.1 Deallocation of allocatable variables

2 Deallocating an unallocated allocatable variable causes an error condition in the DEALLOCATE state-  
3 ment. Deallocating an allocatable variable with the TARGET attribute causes the pointer association  
4 status of any pointer associated with it to become undefined.

5 When the execution of a procedure is terminated by execution of a RETURN or END statement, an  
6 allocatable variable that is a named local variable of the procedure retains its allocation and definition  
7 status if it has the SAVE attribute or is a function result variable or a subobject thereof; otherwise, it  
8 is deallocated.

#### NOTE 6.23

The ALLOCATED intrinsic function may be used to determine whether a variable is allocated or unallocated.

9 If an unsaved allocatable object is a local variable of a module, and it is allocated when execution  
10 of a RETURN or END statement results in no active scoping unit having access to the module, it is  
11 processor-dependent whether the object retains its allocation status or is deallocated.

#### NOTE 6.24

The following example illustrates the effects of SAVE on allocation status.

```

MODULE MOD1
TYPE INITIALIZED_TYPE
  INTEGER :: I = 1 ! Default initialization
END TYPE INITIALIZED_TYPE
SAVE :: SAVED1, SAVED2
INTEGER :: SAVED1, UNSAVED1
TYPE(INITIALIZED_TYPE) :: SAVED2, UNSAVED2
ALLOCATABLE :: SAVED1(:), SAVED2(:), UNSAVED1(:), UNSAVED2(:)
END MODULE MOD1
PROGRAM MAIN
CALL SUB1 ! The values returned by the ALLOCATED intrinsic calls
          ! in the PRINT statement are:
          ! .FALSE., .FALSE., .FALSE., and .FALSE.
          ! Module MOD1 is used, and its variables are allocated.
          ! After return from the subroutine, whether the variables
          ! which were not specified with the SAVE attribute
          ! retain their allocation status is processor dependent.
CALL SUB1 ! The values returned by the first two ALLOCATED intrinsic
          ! calls in the PRINT statement are:
          ! .TRUE., .TRUE.
          ! The values returned by the second two ALLOCATED
          ! intrinsic calls in the PRINT statement are
          ! processor dependent and each could be either
          ! .TRUE. or .FALSE.
CONTAINS
SUBROUTINE SUB1
USE MOD1 ! Brings in saved and unsaved variables.
PRINT *, ALLOCATED(SAVED1), ALLOCATED(SAVED2), &
        ALLOCATED(UNSAVED1), ALLOCATED(UNSAVED2)
IF (.NOT. ALLOCATED(SAVED1)) ALLOCATE(SAVED1(10))
IF (.NOT. ALLOCATED(SAVED2)) ALLOCATE(SAVED2(10))
IF (.NOT. ALLOCATED(UNSAVED1)) ALLOCATE(UNSAVED1(10))

```

**NOTE 6.24 (cont.)**

```

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

```

- 1 If an executable construct references a function whose result is either allocatable or a structure with  
2 a subobject that is allocatable, and the function reference is executed, an allocatable result and any  
3 subobject that is an allocated allocatable entity in the result returned by the function is deallocated  
4 after execution of the innermost executable construct containing the reference.
- 5 If a specification expression in a scoping unit references a function whose result is either allocatable or  
6 a structure with a subobject that is allocatable, and the function reference is executed, an allocatable  
7 result and any subobject that is an allocated allocatable entity in the result returned by the function is  
8 deallocated before execution of the first executable statement in the scoping unit.
- 9 When a procedure is invoked, any allocated allocatable object that is an actual argument associated with  
10 an INTENT(OUT) allocatable dummy argument is deallocated; any allocated allocatable object that is  
11 a subobject of an actual argument associated with an INTENT(OUT) dummy argument is deallocated.
- 12 When an intrinsic assignment statement (7.4.1.3) is executed, any allocated allocatable subobject of the  
13 *variable* is deallocated before the assignment takes place.
- 14 When a variable of derived type is deallocated, any allocated allocatable subobject is deallocated.
- 15 If an allocatable component is a subobject of a finalizable object, that object is finalized before the  
16 component is automatically deallocated.
- 17 The effect of automatic deallocation is the same as that of a DEALLOCATE statement without a  
18 *dealloc-opt-list*.

**NOTE 6.25**

In the following example:

```

SUBROUTINE PROCESS
  REAL, ALLOCATABLE :: TEMP(:)
  REAL, ALLOCATABLE, SAVE :: X(:)
  ...
END SUBROUTINE PROCESS

```

on return from subroutine PROCESS, the allocation status of X is preserved because X has the SAVE attribute. TEMP does not have the SAVE attribute, so it will be deallocated. On the next invocation of PROCESS, TEMP will have an allocation status of unallocated.

19 **6.3.3.2 Deallocation of pointer targets**

- 20 If a pointer appears in a DEALLOCATE statement, its association status shall be defined. Deallocating  
21 a pointer that is disassociated or whose target was not created by an ALLOCATE statement causes an  
22 error condition in the DEALLOCATE statement. If a pointer is associated with an allocatable entity,  
23 the pointer shall not be deallocated.
- 24 If a pointer appears in a DEALLOCATE statement, it shall be associated with the whole of an object  
25 that was created by allocation. Deallocating a pointer target causes the pointer association status of  
26 any other pointer that is associated with the target or a portion of the target to become undefined.

## 1 Section 7: Expressions and assignment

2 This section describes the formation, interpretation, and evaluation rules for expressions, intrinsic and  
3 defined assignment, pointer assignment, masked array assignment (WHERE), and FORALL.

### 4 7.1 Expressions

5 An **expression** represents either a data reference or a computation, and its value is either a scalar or  
6 an array. An expression is formed from operands, operators, and parentheses.

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

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

#### 11 7.1.1 Form of an expression

12 An expression is defined in terms of several categories: primary, level-1 expression, level-2 expression,  
13 level-3 expression, level-4 expression, and level-5 expression.

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

##### 17 7.1.1.1 Primary

18 R701 *primary* is *constant*  
19 or *designator*  
20 or *array-constructor*  
21 or *structure-constructor*  
22 or *function-reference*  
23 or *type-param-inquiry*  
24 or *type-param-name*  
25 or ( *expr* )

26 C701 (R701) The *type-param-name* shall be the name of a type parameter.

27 C702 (R701) The *designator* shall not be a whole assumed-size array.

#### NOTE 7.1

Examples of a *primary* are:

<u>Example</u>	<u>Syntactic class</u>
1.0	<i>constant</i>
'ABCDEFGHIJKLMNQRSTUWXYZ' (I:I)	<i>constant-subobject</i>
A	<i>variable</i>
(/ 1.0, 2.0 /)	<i>array-constructor</i>
PERSON (12, 'Jones')	<i>structure-constructor</i>
F (X, Y)	<i>function-reference</i>

**NOTE 7.1 (cont.)**

(S + T)

*(expr)***1 7.1.1.2 Level-1 expressions**

2 Defined unary operators have the highest operator precedence (Table 7.7). Level-1 expressions are  
3 primaries optionally operated on by defined unary operators:

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

5 R703 *defined-unary-op* is . *letter* [ *letter* ] ... .

6 C703 (R703) A *defined-unary-op* shall not contain more than 63 letters and shall not be the same as  
7 any *intrinsic-operator* or *logical-literal-constant*.

**NOTE 7.2**

Simple examples of a level-1 expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>primary</i> (R701)
.INVERSE. B	<i>level-1-expr</i> (R702)

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

.INVERSE. (A + B)

**8 7.1.1.3 Level-2 expressions**

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

11 R704 *mult-operand* is *level-1-expr* [ *power-op mult-operand* ]

12 R705 *add-operand* is [ *add-operand mult-op* ] *mult-operand*

13 R706 *level-2-expr* is [ [ *level-2-expr* ] *add-op* ] *add-operand*

14 R707 *power-op* is \*\*

15 R708 *mult-op* is \*

16 or /

17 R709 *add-op* is +

18 or -

**NOTE 7.3**

Simple examples of a level-2 expression are:

<u>Example</u>	<u>Syntactic class</u>	<u>Remarks</u>
A	<i>level-1-expr</i>	A is a <i>primary</i> . (R702)
B ** C	<i>mult-operand</i>	B is a <i>level-1-expr</i> , ** is a <i>power-op</i> , and C is a <i>mult-operand</i> . (R704)
D * E	<i>add-operand</i>	D is an <i>add-operand</i> , * is a <i>mult-op</i> , and E is a <i>mult-operand</i> . (R705)
+1	<i>level-2-expr</i>	+ is an <i>add-op</i> and 1 is an <i>add-operand</i> . (R706)
F - I	<i>level-2-expr</i>	F is a <i>level-2-expr</i> , - is an <i>add-op</i> , and I is an <i>add-operand</i> . (R706)

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

**NOTE 7.3 (cont.)**

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

**1 7.1.1.4 Level-3 expressions**

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

3 R710 *level-3-expr* is [ *level-3-expr concat-op* ] *level-2-expr*

4 R711 *concat-op* is //

**NOTE 7.4**

Simple examples of a level-3 expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>level-2-expr</i> (R706)
B // C	<i>level-3-expr</i> (R710)

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

```
X // Y // 'ABCD'
```

**5 7.1.1.5 Level-4 expressions**

6 Level-4 expressions are level-3 expressions optionally involving the relational operators *rel-op*.

7 R712 *level-4-expr* is [ *level-3-expr rel-op* ] *level-3-expr*

8 R713 *rel-op* is .EQ.

9 or .NE.

10 or .LT.

11 or .LE.

12 or .GT.

13 or .GE.

14 or ==

15 or /=

16 or <

17 or <=

18 or >

19 or >=

**NOTE 7.5**

Simple examples of a level-4 expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>level-3-expr</i> (R710)
B == C	<i>level-4-expr</i> (R712)
D < E	<i>level-4-expr</i> (R712)

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

```
(A + B) /= C
```

1 **7.1.1.6 Level-5 expressions**

2 Level-5 expressions are level-4 expressions optionally involving the logical operators *not-op*, *and-op*,  
3 *or-op*, and *equiv-op*.

4	R714	<i>and-operand</i>	is	[ <i>not-op</i> ] <i>level-4-expr</i>
5	R715	<i>or-operand</i>	is	[ <i>or-operand and-op</i> ] <i>and-operand</i>
6	R716	<i>equiv-operand</i>	is	[ <i>equiv-operand or-op</i> ] <i>or-operand</i>
7	R717	<i>level-5-expr</i>	is	[ <i>level-5-expr equiv-op</i> ] <i>equiv-operand</i>
8	R718	<i>not-op</i>	is	.NOT.
9	R719	<i>and-op</i>	is	.AND.
10	R720	<i>or-op</i>	is	.OR.
11	R721	<i>equiv-op</i>	is	.EQV.
12			or	.NEQV.

**NOTE 7.6**

Simple examples of a level-5 expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>level-4-expr</i> (R712)
.NOT. B	<i>and-operand</i> (R714)
C .AND. D	<i>or-operand</i> (R715)
E .OR. F	<i>equiv-operand</i> (R716)
G .EQV. H	<i>level-5-expr</i> (R717)
S .NEQV. T	<i>level-5-expr</i> (R717)

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

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

13 **7.1.1.7 General form of an expression**

14 Expressions are level-5 expressions optionally involving defined binary operators. Defined binary oper-  
15 ators have the lowest operator precedence (Table 7.7).

16	R722	<i>expr</i>	is	[ <i>expr defined-binary-op</i> ] <i>level-5-expr</i>
17	R723	<i>defined-binary-op</i>	is	. <i>letter</i> [ <i>letter</i> ] ... .

18 C704 (R723) A *defined-binary-op* shall not contain more than 63 letters and shall not be the same as  
19 any *intrinsic-operator* or *logical-literal-constant*.

**NOTE 7.7**

Simple examples of an expression are:

<u>Example</u>	<u>Syntactic class</u>
A	<i>level-5-expr</i> (R717)
B.UNION.C	<i>expr</i> (R722)

More complicated examples of an expression are:

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

## 1 7.1.2 Intrinsic operations

2 An **intrinsic operation** is either an intrinsic unary operation or an intrinsic binary operation. An  
 3 **intrinsic unary operation** is an operation of the form *intrinsic-operator*  $x_2$  where  $x_2$  is of an intrinsic  
 4 type (4.4) listed in Table 7.1 for the unary intrinsic operator.

5 An **intrinsic binary operation** is an operation of the form  $x_1$  *intrinsic-operator*  $x_2$  where  $x_1$  and  
 6  $x_2$  are of the intrinsic types (4.4) listed in Table 7.1 for the binary intrinsic operator and are in shape  
 7 conformance (7.1.5).

Table 7.1: Type of operands and results for intrinsic operators

Intrinsic operator <i>op</i>	Type of $x_1$	Type of $x_2$	Type of $[x_1] op x_2$
Unary +, -		I, R, Z	I, R, Z
Binary +, -, *, /, **	I	I, R, Z	I, R, Z
	R	I, R, Z	R, R, Z
	Z	I, R, Z	Z, Z, Z
//	C	C	C
.EQ., .NE., ==, /=	I	I, R, Z	L, L, L
	R	I, R, Z	L, L, L
	Z	I, R, Z	L, L, L
	C	C	L
.GT., .GE., .LT., .LE. >, >=, <, <=	I	I, R	L, L
	R	I, R	L, L
	C	C	L
.NOT.		L	L
.AND., .OR., .EQV., .NEQV.	L	L	L
Note: The symbols I, R, Z, C, and L stand for the types integer, real, complex, character, and logical, respectively. Where more than one type for $x_2$ is given, the type of the result of the operation is given in the same relative position in the next column. For the intrinsic operators with operands of type character, the kind type parameters of the operands shall be the same.			

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

11 For numeric intrinsic binary operations, the two operands may be of different numeric types or different  
 12 kind type parameters. Except for a value raised to an integer power, if the operands have different types  
 13 or kind type parameters, the effect is as if each operand that differs in type or kind type parameter from  
 14 those of the result is converted to the type and kind type parameter of the result before the operation  
 15 is performed. When a value of type real or complex is raised to an integer power, the integer operand  
 16 need not be converted.

17 A **character intrinsic operation**, **relational intrinsic operation**, and **logical intrinsic operation**  
 18 are similarly defined in terms of a **character intrinsic operator** (//), **relational intrinsic operator**  
 19 (.EQ., .NE., .GT., .GE., .LT., .LE., ==, /=, >, >=, <, and <=), and **logical intrinsic operator**  
 20 (.AND., .OR., .NOT., .EQV., and .NEQV.), respectively. For the character intrinsic operator //, the  
 21 kind type parameters shall be the same. For the relational intrinsic operators with character operands,  
 22 the kind type parameters shall be the same.

23 A **numeric relational intrinsic operation** is a relational intrinsic operation where the operands are  
 24 of numeric type. A **character relational intrinsic operation** is a relational intrinsic operation where  
 25 the operands are of type character.

### 1 7.1.3 Defined operations

2 A **defined operation** is either a defined unary operation or a defined binary operation. A **defined**  
 3 **unary operation** is an operation that has the form *defined-unary-op*  $x_2$  or *intrinsic-operator*  $x_2$  and  
 4 that is defined by a function and a generic interface (4.5.1, 12.3.2.1).

5 A function defines the unary operation  $op$   $x_2$  if

- 6 (1) The function is specified with a FUNCTION (12.5.2.1) or ENTRY (12.5.2.4) statement that  
 7 specifies one dummy argument  $d_2$ ,
- 8 (2) Either
  - 9 (a) A generic interface (12.3.2.1) provides the function with a *generic-spec* of OPERA-  
 10 TOR ( $op$ ), or
  - 11 (b) There is a generic binding (4.5.1) in the declared type of  $x_2$  with a *generic-spec* of  
 12 OPERATOR ( $op$ ) and there is a corresponding binding to the function in the dynamic  
 13 type of  $x_2$ ,
- 14 (3) The type of  $d_2$  is compatible with the dynamic type of  $x_2$ ,
- 15 (4) The type parameters, if any, of  $d_2$  match the corresponding type parameters of  $x_2$ , and
- 16 (5) Either
  - 17 (a) The rank of  $x_2$  matches that of  $d_2$  or
  - 18 (b) The function is elemental and there is no other function that defines the operation.

19 If  $d_2$  is an array, the shape of  $x_2$  shall match the shape of  $d_2$ .

20 A **defined binary operation** is an operation that has the form  $x_1$  *defined-binary-op*  $x_2$  or  $x_1$  *intrinsic-*  
 21 *operator*  $x_2$  and that is defined by a function and a generic interface.

22 A function defines the binary operation  $x_1$   $op$   $x_2$  if

- 23 (1) The function is specified with a FUNCTION (12.5.2.1) or ENTRY (12.5.2.4) statement that  
 24 specifies two dummy arguments,  $d_1$  and  $d_2$ ,
- 25 (2) Either
  - 26 (a) A generic interface (12.3.2.1) provides the function with a *generic-spec* of OPERA-  
 27 TOR ( $op$ ), or
  - 28 (b) There is a generic binding (4.5.1) in the declared type of  $x_1$  or  $x_2$  with a *generic-*  
 29 *spec* of OPERATOR ( $op$ ) and there is a corresponding binding to the function in the  
 30 dynamic type of  $x_1$  or  $x_2$ , respectively,
- 31 (3) The types of  $d_1$  and  $d_2$  are compatible with the dynamic types of  $x_1$  and  $x_2$ , respectively,
- 32 (4) The type parameters, if any, of  $d_1$  and  $d_2$  match the corresponding type parameters of  $x_1$   
 33 and  $x_2$ , respectively, and
- 34 (5) Either
  - 35 (a) The ranks of  $x_1$  and  $x_2$  match those of  $d_1$  and  $d_2$  or
  - 36 (b) The function is elemental,  $x_1$  and  $x_2$  are conformable, and there is no other function  
 37 that defines the operation.

38 If  $d_1$  or  $d_2$  is an array, the shapes of  $x_1$  and  $x_2$  shall match the shapes of  $d_1$  and  $d_2$ , respectively.

#### NOTE 7.8

An intrinsic operator may be used as the operator in a defined operation. In such a case, the generic properties of the operator are extended.

1 An **extension operation** is a defined operation in which the operator is of the form *defined-unary-op*  
 2 or *defined-binary-op*. Such an operator is called an **extension operator**. The operator used in an  
 3 extension operation may be such that a generic interface for the operator may specify more than one  
 4 function.

5 A **defined elemental operation** is a defined operation for which the function is elemental (12.7).

#### 6 7.1.4 Type, type parameters, and shape of an expression

7 The type, type parameters, and shape of an expression depend on the operators and on the types, type  
 8 parameters, and shapes of the primaries used in the expression, and are determined recursively from  
 9 the syntactic form of the expression. The type of an expression is one of the intrinsic types (4.4) or a  
 10 derived type (4.5).

11 If an expression is a polymorphic primary or defined operation, the type parameters and the declared and  
 12 dynamic types of the expression are the same as those of the primary or defined operation. Otherwise  
 13 the type parameters and dynamic type of the expression are the same as its declared type and type  
 14 parameters; they are referred to simply as the type and type parameters of the expression.

15 R724 *logical-expr*                            **is** *expr*

16 C705 (R724) *logical-expr* shall be of type logical.

17 R725 *char-expr*                                **is** *expr*

18 C706 (R725) *char-expr* shall be of type character.

19 R726 *default-char-expr*                    **is** *expr*

20 C707 (R726) *default-char-expr* shall be of type default character.

21 R727 *int-expr*                                 **is** *expr*

22 C708 (R727) *int-expr* shall be of type integer.

23 R728 *numeric-expr*                         **is** *expr*

24 C709 (R728) *numeric-expr* shall be of type integer, real, or complex.

##### 25 7.1.4.1 Type, type parameters, and shape of a primary

26 The type, type parameters, and shape of a primary are determined according to whether the primary is a  
 27 constant, variable, array constructor, structure constructor, function reference, type parameter inquiry,  
 28 type parameter name, or parenthesized expression. If a primary is a constant, its type, type parameters,  
 29 and shape are those of the constant. If it is a structure constructor, it is scalar and its type and type  
 30 parameters are as described in 4.5.9. If it is an array constructor, its type, type parameters, and shape  
 31 are as described in 4.7. If it is a variable or function reference, its type, type parameters, and shape are  
 32 those of the variable (5.1.1, 5.1.2) or the function reference (12.4.2), respectively. If the function reference  
 33 is generic (12.3.2.1, 13.5) then its type, type parameters, and shape are those of the specific function  
 34 referenced, which is determined by the types, type parameters, and ranks of its actual arguments as  
 35 specified in 16.2.3. If it is a type parameter inquiry or type parameter name, it is a scalar integer with  
 36 the kind of the type parameter.

37 If a primary is a parenthesized expression, its type, type parameters, and shape are those of the expres-  
 38 sion.

39 If a pointer appears as one of the following, the associated target object is referenced:

- 1 (1) A primary in an intrinsic or defined operation,
- 2 (2) The *expr* of a parenthesized primary, or
- 3 (3) The only primary on the right-hand side of an intrinsic assignment statement.

4 The type, type parameters, and shape of the primary are those of the current target. If the pointer is  
5 not associated with a target, it may appear as a primary only as an actual argument in a reference to  
6 a procedure whose corresponding dummy argument is declared to be a pointer, or as the target in a  
7 pointer assignment statement.

8 A disassociated array pointer or an unallocated allocatable array has no shape but does have rank. The  
9 type, type parameters, and rank of the result of the NULL intrinsic function depend on context (13.7.88).

#### 10 7.1.4.2 Type, type parameters, and shape of the result of an operation

11 The type of the result of an intrinsic operation  $[x_1] \text{ op } x_2$  is specified by Table 7.1. The shape of the  
12 result of an intrinsic operation is the shape of  $x_2$  if *op* is unary or if  $x_1$  is scalar, and is the shape of  $x_1$   
13 otherwise.

14 The type, type parameters, and shape of the result of a defined operation  $[x_1] \text{ op } x_2$  is specified by the  
15 function defining the operation (7.2).

16 An expression of an intrinsic type has a kind type parameter. An expression of type character also has  
17 a character length parameter.

18 The type parameters of the result of an intrinsic operation are as follows:

- 19 (1) For an expression  $x_1 // x_2$  where *//* is the character intrinsic operator and  $x_1$  and  $x_2$  are  
20 of type character, the character length parameter is the sum of the lengths of the operands  
21 and the kind type parameter is the kind type parameter of  $x_1$ , which shall be the same as  
22 the kind type parameter of  $x_2$ .
- 23 (2) For an expression  $\text{op } x_2$  where *op* is an intrinsic unary operator and  $x_2$  is of type integer,  
24 real, complex, or logical, the kind type parameter of the expression is that of the operand.
- 25 (3) For an expression  $x_1 \text{ op } x_2$  where *op* is a numeric intrinsic binary operator with one operand  
26 of type integer and the other of type real or complex, the kind type parameter of the  
27 expression is that of the real or complex operand.
- 28 (4) For an expression  $x_1 \text{ op } x_2$  where *op* is a numeric intrinsic binary operator with both  
29 operands of the same type and kind type parameters, or with one real and one complex  
30 with the same kind type parameters, the kind type parameter of the expression is identical  
31 to that of each operand. In the case where both operands are integer with different kind type  
32 parameters, the kind type parameter of the expression is that of the operand with the greater  
33 decimal exponent range if the decimal exponent ranges are different; if the decimal exponent  
34 ranges are the same, the kind type parameter of the expression is processor dependent, but  
35 it is the same as that of one of the operands. In the case where both operands are any  
36 of type real or complex with different kind type parameters, the kind type parameter of  
37 the expression is that of the operand with the greater decimal precision if the decimal  
38 precisions are different; if the decimal precisions are the same, the kind type parameter of  
39 the expression is processor dependent, but it is the same as that of one of the operands.
- 40 (5) For an expression  $x_1 \text{ op } x_2$  where *op* is a logical intrinsic binary operator with both operands  
41 of the same kind type parameter, the kind type parameter of the expression is identical to  
42 that of each operand. In the case where both operands are of type logical with different  
43 kind type parameters, the kind type parameter of the expression is processor dependent,  
44 but it is the same as that of one of the operands.
- 45 (6) For an expression  $x_1 \text{ op } x_2$  where *op* is a relational intrinsic operator, the expression has  
46 the default logical kind type parameter.

### 1 7.1.5 Conformability rules for elemental operations

2 An **elemental operation** is an intrinsic operation or a defined elemental operation. Two entities are  
3 in **shape conformance** if both are arrays of the same shape, or one or both are scalars.

4 For all elemental binary operations, the two operands shall be in shape conformance. In the case where  
5 one is a scalar and the other an array, the scalar is treated as if it were an array of the same shape as  
6 the array operand with every element, if any, of the array equal to the value of the scalar.

### 7 7.1.6 Specification expression

8 A **specification expression** is an expression with limitations that make it suitable for use in speci-  
9 cations such as length type parameters (C501) and array bounds (R512, R513).

10 R729 *specification-expr*                    **is** *scalar-int-expr*

11 C710 (R729) The *scalar-int-expr* shall be a restricted expression.

12 A **restricted expression** is an expression in which each operation is intrinsic and each primary is

- 13 (1) A constant or subobject of a constant,
- 14 (2) An object designator with a base object that is a dummy argument that has neither the  
15 OPTIONAL nor the INTENT (OUT) attribute,
- 16 (3) An object designator with a base object that is in a common block,
- 17 (4) An object designator with a base object that is made accessible by use association or host  
18 association,
- 19 (5) An array constructor where each element and each *scalar-int-expr* of each *ac-implied-do-*  
20 *control* is a restricted expression,
- 21 (6) A structure constructor where each component is a restricted expression,
- 22 (7) A specification inquiry where each designator or function argument is
  - 23 (a) a restricted expression or
  - 24 (b) a variable whose properties inquired about are not
    - 25 (i) dependent on the upper bound of the last dimension of an assumed-size array,
    - 26 (ii) deferred, or
    - 27 (iii) defined by an expression that is not a restricted expression,
- 28 (8) A reference to any other standard intrinsic function where each argument is a restricted  
29 expression,
- 30 (9) A reference to a specification function where each argument is a restricted expression,
- 31 (10) A type parameter of the derived type being defined,
- 32 (11) An *ac-do-variable* within an array constructor where each *scalar-int-expr* of the correspond-  
33 ing *ac-implied-do-control* is a restricted expression, or
- 34 (12) A restricted expression enclosed in parentheses,

35 where each subscript, section subscript, substring starting point, substring ending point, and type pa-  
36 rameter value is a restricted expression, and where any final subroutine that is invoked is pure.

37 A **specification inquiry** is a reference to

- 38 (1) an array inquiry function (13.5.7),
- 39 (2) the bit inquiry function BIT.SIZE,
- 40 (3) the character inquiry function LEN,
- 41 (4) the kind inquiry function KIND,

- 1 (5) the character inquiry function `NEW_LINE`,
- 2 (6) a numeric inquiry function (13.5.6),
- 3 (7) a type parameter inquiry (6.1.3), or
- 4 (8) an IEEE inquiry function (14.9.1),

5 A function is a **specification function** if it is a pure function, is not a standard intrinsic function, is  
6 not an internal function, is not a statement function, and does not have a dummy procedure argument.

7 Evaluation of a specification expression shall not directly or indirectly cause a procedure defined by the  
8 subprogram in which it appears to be invoked.

#### NOTE 7.9

Specification functions are nonintrinsic functions that may be used in specification expressions to determine the attributes of data objects. The requirement that they be pure ensures that they cannot have side effects that could affect other objects being declared in the same *specification-part*. The requirement that they not be internal ensures that they cannot inquire, via host association, about other objects being declared in the same *specification-part*. The prohibition against recursion avoids the creation of a new instance of a procedure while construction of one is in progress.

9 A variable in a specification expression shall have its type and type parameters, if any, specified by a  
10 previous declaration in the same scoping unit, by the implicit typing rules in effect for the scoping unit,  
11 or by host or use association. If a variable in a specification expression is typed by the implicit typing  
12 rules, its appearance in any subsequent type declaration statement shall confirm the implied type and  
13 type parameters.

14 If a specification expression includes a specification inquiry that depends on a type parameter or an  
15 array bound of an entity specified in the same *specification-part*, the type parameter or array bound  
16 shall be specified in a prior specification of the *specification-part*. The prior specification may be to the  
17 left of the specification inquiry in the same statement, but shall not be within the same *entity-decl*. If a  
18 specification expression includes a reference to the value of an element of an array specified in the same  
19 *specification-part*, the array shall be completely specified in prior declarations.

#### NOTE 7.10

The following are examples of specification expressions:

```
LBOUND (B, 1) + 5 ! B is an assumed-shape dummy array
M + LEN (C)      ! M and C are dummy arguments
2 * PRECISION (A) ! A is a real variable made accessible
                  ! by a USE statement
```

### 20 7.1.7 Initialization expression

21 An **initialization expression** is an expression with limitations that make it suitable for use as a kind  
22 type parameter, initializer, or named constant. It is an expression in which each operation is intrinsic,  
23 and each primary is

- 24 (1) A constant or subobject of a constant,
- 25 (2) An array constructor where each element and each *scalar-int-expr* of each *ac-implicit-do-control* is an initialization expression,
- 26 (3) A structure constructor where each *component-spec* corresponding to an allocatable component is a reference to the transformational intrinsic function `NULL`, and each other  
27 *component-spec* is an initialization expression,
- 28 (4) A reference to an elemental standard intrinsic function, where each argument is an initial-  
29  
30

- 1            initialization expression,
- 2            (5) A reference to a transformational standard intrinsic function other than NULL, where each
- 3            argument is an initialization expression,
- 4            (6) A reference to the transformational intrinsic function NULL that does not have an argu-
- 5            ment with a type parameter that is assumed or is defined by an expression that is not an
- 6            initialization expression,
- 7            (7) A reference to the transformational function IEEE\_SELECTED\_REAL\_KIND from the in-
- 8            trinsic module IEEE\_ARITHMETIC (14), where each argument is an initialization expres-
- 9            sion.
- 10           (8) A specification inquiry where each designator or function argument is
- 11           (a) an initialization expression or
- 12           (b) a variable whose properties inquired about are not
- 13           (i) assumed,
- 14           (ii) deferred, or
- 15           (iii) defined by an expression that is not an initialization expression,
- 16           (9) A kind type parameter of the derived type being defined,
- 17           (10) An *ac-do-variable* within an array constructor where each *scalar-int-expr* of the correspond-
- 18           ing *ac-implied-do-control* is an initialization expression, or
- 19           (11) An initialization expression enclosed in parentheses,

20 and where each subscript, section subscript, substring starting point, substring ending point, and type

21 parameter value is an initialization expression.

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

23 C711 (R730) *initialization-expr* shall be an initialization expression.

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

25 C712 (R731) *char-initialization-expr* shall be an initialization expression.

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

27 C713 (R732) *int-initialization-expr* shall be an initialization expression.

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

29 C714 (R733) *logical-initialization-expr* shall be an initialization expression.

30 If an initialization expression includes a specification inquiry that depends on a type parameter or an

31 array bound of an entity specified in the same *specification-part*, the type parameter or array bound

32 shall be specified in a prior specification of the *specification-part*. The prior specification may be to the

33 left of the specification inquiry in the same statement, but shall not be within the same *entity-decl*.

#### NOTE 7.11

The following are examples of initialization expressions:

```

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

```

**NOTE 7.11 (cont.)**

```

4.0 * atan(1.0)
ceiling(number_of_decimal_digits / log10(radix(0.0)))

```

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

**1 7.1.8 Evaluation of operations**

2 An intrinsic operation requires the values of its operands.

3 The execution of any numeric operation whose result is not defined by the arithmetic used by the  
4 processor is prohibited. Raising a negative-valued primary of type real to a real power is prohibited.

5 The evaluation of a function reference shall neither affect nor be affected by the evaluation of any other  
6 entity within the statement. If a function reference causes definition or undefinition of an actual argument  
7 of the function, that argument or any associated entities shall not appear elsewhere in the same statement.  
8 However, execution of a function reference in the logical expression in an IF statement (8.1.2.4), the mask  
9 expression in a WHERE statement (7.4.3.1), or the subscripts and strides in a FORALL statement (7.4.4)  
10 is permitted to define variables in the statement that is conditionally executed.

**NOTE 7.12**

For example, the statements

```

A (I) = F (I)
Y = G (X) + X

```

are prohibited if the reference to F defines or undefines I or the reference to G defines or undefines X.

However, in the statements

```

IF (F (X)) A = X
WHERE (G (X)) B = X

```

F or G may define X.

11 The declared type of an expression in which a function reference appears does not affect, and is not  
12 affected by, the evaluation of the actual arguments of the function.

13 Execution of an array element reference requires the evaluation of its subscripts. The type of an expres-  
14 sion in which the array element reference appears does not affect, and is not affected by, the evaluation  
15 of its subscripts. Execution of an array section reference requires the evaluation of its section subscripts.  
16 The type of an expression in which an array section appears does not affect, and is not affected by, the  
17 evaluation of the array section subscripts. Execution of a substring reference requires the evaluation of  
18 its substring expressions. The type of an expression in which a substring appears does not affect, and  
19 is not affected by, the evaluation of the substring expressions. It is not necessary for a processor to  
20 evaluate any subscript expressions or substring expressions for an array of zero size or a character entity  
21 of zero length.

22 The appearance of an array constructor requires the evaluation of each *scalar-int-expr* of the *ac-implied-*  
23 *do-control* in any *ac-implied-do* it may contain. The type of an expression in which an array constructor  
24 appears does not affect, and is not affected by, the evaluation of such bounds and stride expressions.

25 When an elemental binary operation is applied to a scalar and an array or to two arrays of the same

- 1 shape, the operation is performed element-by-element on corresponding array elements of the array  
 2 operands. The processor may perform the element-by-element operations in any order.

**NOTE 7.13**

For example, the array expression

$$A + B$$

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

- 3 When an elemental unary operator operates on an array operand, the operation is performed element-  
 4 by-element, in any order, and the result is the same shape as the operand.

5 **7.1.8.1 Evaluation of operands**

- 6 It is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely  
 7 each operand, if the value of the expression can be determined otherwise.

**NOTE 7.14**

This principle is most often applicable to logical expressions, zero-sized arrays, and zero-length strings, but it applies to all expressions.

For example, in evaluating the expression

$$X > Y \text{ .OR. } L(Z)$$

where X, Y, and Z are real and L is a function of type logical, the function reference L(Z) need not be evaluated if X is greater than Y. Similarly, in the array expression

$$W(Z) + A$$

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

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

**NOTE 7.15**

In the examples in Note 7.14, if L or W defines its argument, evaluation of the expressions under the specified conditions causes Z to become undefined, no matter whether or not L(Z) or W(Z) is evaluated.

11 **7.1.8.2 Integrity of parentheses**

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

**NOTE 7.16**

For example, in evaluating the expression  $A + (B - C)$  where A, B, and C are of numeric types, the difference of B and C shall be evaluated before the addition operation is performed; the processor

**NOTE 7.16 (cont.)**

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

**1 7.1.8.3 Evaluation of numeric intrinsic operations**

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

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

**NOTE 7.17**

Any difference between the values of the expressions  $(1./3.)*3.$  and  $1.$  is a computational difference, not a mathematical difference. The difference between the values of the expressions  $5/2$  and  $5./2.$  is a mathematical difference, not a computational difference.

The mathematical definition of integer division is given in 7.2.1.1.

**NOTE 7.18**

The following are examples of expressions with allowable alternative forms that may be used by the processor in the evaluation of those expressions. A, B, and C represent arbitrary real or complex operands; I and J represent arbitrary integer operands; and X, Y, and Z represent arbitrary operands of numeric type.

<u>Expression</u>	<u>Allowable alternative form</u>
$X + Y$	$Y + X$
$X * Y$	$Y * X$
$-X + Y$	$Y - X$
$X + Y + Z$	$X + (Y + Z)$
$X - Y + Z$	$X - (Y - Z)$
$X * A / Z$	$X * (A / Z)$
$X * Y - X * Z$	$X * (Y - Z)$
$A / B / C$	$A / (B * C)$
$A / 5.0$	$0.2 * A$

The following are examples of expressions with forbidden alternative forms that shall not be used by a processor in the evaluation of those expressions.

<u>Expression</u>	<u>Forbidden alternative form</u>
$I / 2$	$0.5 * I$
$X * I / J$	$X * (I / J)$
$I / J / A$	$I / (J * A)$
$(X + Y) + Z$	$X + (Y + Z)$
$(X * Y) - (X * Z)$	$X * (Y - Z)$
$X * (Y - Z)$	$X * Y - X * Z$

8 In addition to the parentheses required to establish the desired interpretation, parentheses may be  
9 included to restrict the alternative forms that may be used by the processor in the actual evaluation  
10 of the expression. This is useful for controlling the magnitude and accuracy of intermediate values  
11 developed during the evaluation of an expression.

**NOTE 7.19**

For example, in the expression

$$A + (B - C)$$

the parenthesized expression  $(B - C)$  shall be evaluated and then added to  $A$ .

The inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions

$$A * I / J$$

$$A * (I / J)$$

may have different mathematical values if  $I$  and  $J$  are of type integer.

- 1 Each operand in a numeric intrinsic operation has a type that may depend on the order of evaluation
- 2 used by the processor.

**NOTE 7.20**

For example, in the evaluation of the expression

$$Z + R + I$$

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

### 3 7.1.8.4 Evaluation of the character intrinsic operation

- 4 The rules given in 7.2.2 specify the interpretation of the character intrinsic operation. A processor is
- 5 only required to evaluate as much of the character intrinsic operation as is required by the context in
- 6 which the expression appears.

**NOTE 7.21**

For example, the statements

```
CHARACTER (LEN = 2) C1, C2, C3, CF
C1 = C2 // CF (C3)
```

do not require the function  $CF$  to be evaluated, because only the value of  $C2$  is needed to determine the value of  $C1$  because  $C1$  and  $C2$  both have a length of 2.

### 7 7.1.8.5 Evaluation of relational intrinsic operations

- 8 The rules given in 7.2.3 specify the interpretation of relational intrinsic operations. Once the interpre-
- 9 tation of an expression has been established in accordance with those rules, the processor may evaluate
- 10 any other expression that is relationally equivalent, provided that the integrity of parentheses in any
- 11 expression is not violated.

**NOTE 7.22**

For example, the processor may choose to evaluate the expression

**NOTE 7.22 (cont.)**

$$I > J$$

where I and J are integer variables, as

$$J - I < 0$$

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

**3 7.1.8.6 Evaluation of logical intrinsic operations**

4 The rules given in 7.2.4 specify the interpretation of logical intrinsic operations. Once the interpretation  
5 of an expression has been established in accordance with those rules, the processor may evaluate any  
6 other expression that is logically equivalent, provided that the integrity of parentheses in any expression  
7 is not violated.

**NOTE 7.23**

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

$$L1 \text{ .AND. } L2 \text{ .AND. } L3$$

as

$$L1 \text{ .AND. } (L2 \text{ .AND. } L3)$$

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

**10 7.1.8.7 Evaluation of a defined operation**

11 The rules given in 7.2 specify the interpretation of a defined operation. Once the interpretation of an  
12 expression has been established in accordance with those rules, the processor may evaluate any other  
13 expression that is equivalent, provided that the integrity of parentheses is not violated.

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

**16 7.2 Interpretation of operations**

17 The intrinsic operations are those defined in 7.1.2. These operations are divided into the following  
18 categories: numeric, character, relational, and logical. The interpretations defined in the following  
19 sections apply to both scalars and arrays; the interpretation for arrays is obtained by applying the  
20 interpretation for scalars element by element.

21 The interpretation of a defined operation is provided by the function that defines the operation. The type,  
22 type parameters and interpretation of an expression that consists of an intrinsic or defined operation are  
23 independent of the type and type parameters of the context or any larger expression in which it appears.

**NOTE 7.24**

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

- 1 The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT.,  
 2 .LE., .GT., .GE., .EQ., and .NE., respectively.

### 3 **7.2.1 Numeric intrinsic operations**

- 4 A numeric operation is used to express a numeric computation. Evaluation of a numeric operation  
 5 produces a numeric value. The permitted data types for operands of the numeric intrinsic operations  
 6 are specified in 7.1.2.

- 7 The numeric operators and their interpretation in an expression are given in Table 7.2, where  $x_1$  denotes  
 8 the operand to the left of the operator and  $x_2$  denotes the operand to the right of the operator.

Table 7.2: Interpretation of the numeric intrinsic operators

Operator	Representing	Use of operator	Interpretation
**	Exponentiation	$x_1 ** x_2$	Raise $x_1$ to the power $x_2$
/	Division	$x_1 / x_2$	Divide $x_1$ by $x_2$
*	Multiplication	$x_1 * x_2$	Multiply $x_1$ by $x_2$
-	Subtraction	$x_1 - x_2$	Subtract $x_2$ from $x_1$
-	Negation	$- x_2$	Negate $x_2$
+	Addition	$x_1 + x_2$	Add $x_1$ and $x_2$
+	Identity	$+ x_2$	Same as $x_2$

- 9 The interpretation of a division operation depends on the types of the operands (7.2.1.1).  
 10 If  $x_1$  and  $x_2$  are of type integer and  $x_2$  has a negative value, the interpretation of  $x_1 ** x_2$  is the same  
 11 as the interpretation of  $1/(x_1 ** \text{ABS}(x_2))$ , which is subject to the rules of integer division (7.2.1.1).

**NOTE 7.25**

For example,  $2 ** (-3)$  has the value of  $1/(2 ** 3)$ , which is zero.

#### 12 **7.2.1.1 Integer division**

- 13 One operand of type integer may be divided by another operand of type integer. Although the math-  
 14 ematical quotient of two integers is not necessarily an integer, Table 7.1 specifies that an expression  
 15 involving the division operator with two operands of type integer is interpreted as an expression of type  
 16 integer. The result of such an operation is the integer closest to the mathematical quotient and between  
 17 zero and the mathematical quotient inclusively.

**NOTE 7.26**

For example, the expression  $(-8) / 3$  has the value  $(-2)$ .

#### 18 **7.2.1.2 Complex exponentiation**

- 19 In the case of a complex value raised to a complex power, the value of the operation  $x_1 ** x_2$  is the  
 20 principal value of  $x_1^{x_2}$ .

## 1 7.2.2 Character intrinsic operation

2 The character intrinsic operator // is used to concatenate two operands of type character with the same  
3 kind type parameter. Evaluation of the character intrinsic operation produces a result of type character.

4 The interpretation of the character intrinsic operator // when used to form an expression is given in  
5 Table 7.3, where  $x_1$  denotes the operand to the left of the operator and  $x_2$  denotes the operand to the  
6 right of the operator.

Table 7.3: Interpretation of the character intrinsic operator //

Operator	Representing	Use of operator	Interpretation
//	Concatenation	$x_1 // x_2$	Concatenate $x_1$ with $x_2$

7 The result of the character intrinsic operation // is a character string whose value is the value of  $x_1$   
8 concatenated on the right with the value of  $x_2$  and whose length is the sum of the lengths of  $x_1$  and  $x_2$ .  
9 Parentheses used to specify the order of evaluation have no effect on the value of a character expression.

### NOTE 7.27

For example, the value of ('AB' // 'CDE') // 'F' is the string 'ABCDEF'. Also, the value of 'AB' // ('CDE' // 'F') is the string 'ABCDEF'.

## 10 7.2.3 Relational intrinsic operations

11 A relational intrinsic operation is used to compare values of two operands using the relational intrinsic  
12 operators .LT., .LE., .GT., .GE., .EQ., .NE., <, <=, >, >=, ==, and /=. The permitted types for  
13 operands of the relational intrinsic operators are specified in 7.1.2.

### NOTE 7.28

As shown in Table 7.1, a relational intrinsic operator cannot be used to compare the value of an expression of a numeric type with one of type character or logical. Also, two operands of type logical cannot be compared, a complex operand may be compared with another numeric operand only when the operator is .EQ., .NE., ==, or /=, and two character operands cannot be compared unless they have the same kind type parameter value.

14 Evaluation of a relational intrinsic operation produces a result of type default logical.

15 The interpretation of the relational intrinsic operators is given in Table 7.4, where  $x_1$  denotes the operand  
16 to the left of the operator and  $x_2$  denotes the operand to the right of the operator.

Table 7.4: Interpretation of the relational intrinsic operators

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

**Interpretation of the relational intrinsic operators** (cont.)

Operator	Representing	Use of operator	Interpretation
/=	Not equal to	$x_1 \neq x_2$	$x_1$ not equal to $x_2$

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

5 In the numeric relational operation

6  $x_1 \text{ rel-op } x_2$

7 if the types or kind type parameters of  $x_1$  and  $x_2$  differ, their values are converted to the type and kind  
 8 type parameter of the expression  $x_1 + x_2$  before evaluation.

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

13 For a character relational intrinsic operation, the operands are compared one character at a time in  
 14 order, beginning with the first character of each character operand. If the operands are of unequal  
 15 length, the shorter operand is treated as if it were extended on the right with blanks to the length of  
 16 the longer operand. If both  $x_1$  and  $x_2$  are of zero length,  $x_1$  is equal to  $x_2$ ; if every character of  $x_1$   
 17 is the same as the character in the corresponding position in  $x_2$ ,  $x_1$  is equal to  $x_2$ . Otherwise, at the first  
 18 position where the character operands differ, the character operand  $x_1$  is considered to be less than  $x_2$   
 19 if the character value of  $x_1$  at this position precedes the value of  $x_2$  in the collating sequence (4.4.4.3);  
 20  $x_1$  is greater than  $x_2$  if the character value of  $x_1$  at this position follows the value of  $x_2$  in the collating  
 21 sequence.

**NOTE 7.29**

The collating sequence depends partially on the processor; however, the result of the use of the operators .EQ., .NE., ==, and /= does not depend on the collating sequence.

For nondefault character types, the blank padding character is processor dependent.

## 22 7.2.4 Logical intrinsic operations

23 A logical operation is used to express a logical computation. Evaluation of a logical operation produces  
 24 a result of type logical. The permitted types for operands of the logical intrinsic operations are specified  
 25 in 7.1.2.

26 The logical operators and their interpretation when used to form an expression are given in Table 7.5,  
 27 where  $x_1$  denotes the operand to the left of the operator and  $x_2$  denotes the operand to the right of the  
 28 operator.

Table 7.5: **Interpretation of the logical intrinsic operators**

Operator	Representing	Use of operator	Interpretation
.NOT.	Logical negation	.NOT. $x_2$	True if $x_2$ is false
.AND.	Logical conjunction	$x_1$ .AND. $x_2$	True if $x_1$ and $x_2$ are both true
.OR.	Logical inclusive disjunction	$x_1$ .OR. $x_2$	True if $x_1$ and/or $x_2$ is true
.NEQV.	Logical nonequivalence	$x_1$ .NEQV. $x_2$	True if either $x_1$ or $x_2$ is true, but not both

## Interpretation of the logical intrinsic operators

(cont.)

Operator	Representing	Use of operator	Interpretation
.EQV.	Logical equivalence	$x_1$ .EQV. $x_2$	True if both $x_1$ and $x_2$ are true or both are false

- 1 The values of the logical intrinsic operations are shown in Table 7.6.

Table 7.6: The values of operations involving logical intrinsic operators

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

## 2 7.3 Precedence of operators

- 3 There is a precedence among the intrinsic and extension operations corresponding to the form of expressions specified in 7.1.1, which determines the order in which the operands are combined unless the order is changed by the use of parentheses. This precedence order is summarized in Table 7.7.

Table 7.7: Categories of operations and relative precedence

Category of operation	Operators	Precedence
Extension	<i>defined-unary-op</i>	Highest
Numeric	**	.
Numeric	* or /	.
Numeric	unary + or -	.
Numeric	binary + or -	.
Character	//	.
Relational	.EQ., .NE., .LT., .LE., .GT., .GE., ==, /=, <, <=, >, >=	.
Logical	.NOT.	.
Logical	.AND.	.
Logical	.OR.	.
Logical	.EQV. or .NEQV.	.
Extension	defined-binary-op	Lowest

- 6 The precedence of a defined operation is that of its operator.

**NOTE 7.30**

For example, in the expression

```
-A ** 2
```

the exponentiation operator (\*\*) has precedence over the negation operator (-); therefore, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. The interpretation of the above expression is the same as the interpretation of the expression

```
-(A ** 2)
```

- 1 The general form of an expression (7.1.1) also establishes a precedence among operators in the same
- 2 syntactic class. This precedence determines the order in which the operands are to be combined in
- 3 determining the interpretation of the expression unless the order is changed by the use of parentheses.

**NOTE 7.31**

In interpreting a *level-2-expr* containing two or more binary operators + or −, each operand (*add-operand*) is combined from left to right. Similarly, the same left-to-right interpretation for a *mult-operand* in *add-operand*, as well as for other kinds of expressions, is a consequence of the general form. However, for interpreting a *mult-operand* expression when two or more exponentiation operators \*\* combine *level-1-expr* operands, each *level-1-expr* is combined from right to left.

For example, the expressions

```
2.1 + 3.4 + 4.9
2.1 * 3.4 * 4.9
2.1 / 3.4 / 4.9
2 ** 3 ** 4
'AB' // 'CD' // 'EF'
```

have the same interpretations as the expressions

```
(2.1 + 3.4) + 4.9
(2.1 * 3.4) * 4.9
(2.1 / 3.4) / 4.9
2 ** (3 ** 4)
('AB' // 'CD') // 'EF'
```

As a consequence of the general form (7.1.1), only the first *add-operand* of a *level-2-expr* may be preceded by the identity (+) or negation (−) operator. These formation rules do not permit expressions containing two consecutive numeric operators, such as A \*\* −B or A + −B. However, expressions such as A \*\* (−B) and A + (−B) are permitted. The rules do allow a binary operator or an intrinsic unary operator to be followed by a defined unary operator, such as:

```
A * .INVERSE. B
- .INVERSE. (B)
```

As another example, in the expression

```
A .OR. B .AND. C
```

the general form implies a higher precedence for the .AND. operator than for the .OR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

```
A .OR. (B .AND. C)
```

**NOTE 7.32**

An expression may contain more than one category of operator. The logical expression

```
L .OR. A + B >= C
```

where A, B, and C are of type real, and L is of type logical, contains a numeric operator, a relational operator, and a logical operator. This expression would be interpreted the same as the expression

**NOTE 7.32 (cont.)**

L .OR. ((A + B) >= C)
-----------------------

**NOTE 7.33**

If

- (1) The operator \*\* is extended to type logical,
- (2) The operator .STARSTAR. is defined to duplicate the function of \*\* on type real,
- (3) .MINUS. is defined to duplicate the unary operator -, and
- (4) L1 and L2 are type logical and X and Y are type real,

then in precedence: L1 \*\* L2 is higher than X \* Y; X \* Y is higher than X .STARSTAR. Y; and .MINUS. X is higher than -X.

## 1 7.4 Assignment

2 Execution of an assignment statement causes a variable to become defined or redefined. Execution of a  
 3 pointer assignment statement causes a pointer to become associated with a target or causes its pointer  
 4 association status to become disassociated or undefined. Execution of a WHERE statement or WHERE  
 5 construct masks the evaluation of expressions and assignment of values in array assignment statements  
 6 according to the value of a logical array expression. Execution of a FORALL statement or FORALL  
 7 construct controls the assignment to elements of arrays by using a set of index variables and a mask  
 8 expression.

### 9 7.4.1 Assignment statement

10 A variable may be defined or redefined by execution of an assignment statement.

#### 11 7.4.1.1 General form

12 R734 *assignment-stmt*                    **is** *variable* = *expr*

13 C715 (R734) The *variable* in an *assignment-stmt* shall not be a whole assumed-size array.

**NOTE 7.34**

Examples of an assignment statement are:
--

<pre>A = 3.5 + X * Y I = INT (A)</pre>
--

14 An *assignment-stmt* shall meet the requirements of either a defined assignment statement or an intrinsic  
 15 assignment statement.

#### 16 7.4.1.2 Intrinsic assignment statement

17 An **intrinsic assignment statement** is an assignment statement that is not a defined assignment  
 18 statement (7.4.1.4). In an intrinsic assignment statement, *variable* shall not be polymorphic, and

- 19 (1) If *expr* is an array then *variable* shall also be an array,
- 20 (2) Either *variable* shall be an allocatable array of the same rank as *expr* or the shapes of  
 21 *variable* and *expr* shall conform, and

1 (3) The declared types of *variable* and *expr* shall conform as specified in Table 7.8.

2 Table 7.8: **Type conformance for the intrinsic assignment statement**

Type of <i>variable</i>	Type of <i>expr</i>
integer	integer, real, complex
real	integer, real, complex
complex	integer, real, complex
ISO 10646, ASCII, or default character	ISO 10646, ASCII, or default character
other character	character of the same kind type parameter as <i>variable</i>
logical	logical
derived type	same derived type and kind type parameters as <i>variable</i> ; each length type parameter value shall be the same unless <i>variable</i> is allocatable and its corresponding type parameter is deferred

3 A **numeric intrinsic assignment statement** is an intrinsic assignment statement for which *variable*  
4 and *expr* are of numeric type. A **character intrinsic assignment statement** is an intrinsic assign-  
5 ment statement for which *variable* and *expr* are of type character. A **logical intrinsic assignment**  
6 **statement** is an intrinsic assignment statement for which *variable* and *expr* are of type logical. A  
7 **derived-type intrinsic assignment statement** is an intrinsic assignment statement for which *vari-*  
8 *able* and *expr* are of derived type.

9 An **array intrinsic assignment statement** is an intrinsic assignment statement for which *variable* is  
10 an array. The *variable* shall not be a many-one array section (6.2.2.3.2).

11 If *variable* is a pointer, it shall be associated with a definable target such that the type, type parameters,  
12 and shape of the target and *expr* conform.

### 13 7.4.1.3 Interpretation of intrinsic assignments

14 Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all  
15 expressions within *variable* (7.1.8), the possible conversion of *expr* to the type and type parameters  
16 of *variable* (Table 7.9), and the definition of *variable* with the resulting value. The execution of the  
17 assignment shall have the same effect as if the evaluation of all operations in *expr* and *variable* occurred  
18 before any portion of *variable* is defined by the assignment. The evaluation of expressions within *variable*  
19 shall neither affect nor be affected by the evaluation of *expr*. No value is assigned to *variable* if *variable*  
20 is of type character and zero length, or is an array of size zero.

21 If *variable* is a pointer, the value of *expr* is assigned to the target of *variable*.

22 If *variable* is an allocated allocatable variable, it is deallocated if *expr* is an array of different shape  
23 or any of the corresponding length type parameter values of *variable* and *expr* differ. If *variable* is or  
24 becomes an unallocated allocatable variable, then it is allocated with each deferred type parameter equal  
25 to the corresponding type parameters of *expr*, with the shape of *expr*, and with each lower bound equal  
26 to the corresponding element of LBOUND(*expr*).

#### NOTE 7.35

For example, given the declaration

```
CHARACTER(:), ALLOCATABLE :: NAME
```

then after the assignment statement

```
NAME = 'Dr. '//FIRST_NAME//' '//SURNAME
```

**NOTE 7.35 (cont.)**

NAME will have the length  $\text{LEN}(\text{FIRST\_NAME})+\text{LEN}(\text{SURNAME})+5$ , even if it had previously been unallocated, or allocated with a different length. However, for the assignment statement

```
NAME(:) = 'Dr. '//FIRST_NAME//' '//SURNAME
```

NAME must already be allocated at the time of the assignment; the assigned value is truncated or blank padded to the previously allocated length of NAME.

- 1 Both *variable* and *expr* may contain references to any portion of *variable*.

**NOTE 7.36**

For example, in the character intrinsic assignment statement:

```
STRING (2:5) = STRING (1:4)
```

the assignment of the first character of STRING to the second character does not affect the evaluation of STRING (1:4). If the value of STRING prior to the assignment was 'ABCDEF', the value following the assignment is 'AABCDF'.

- 2 If *expr* is a scalar and *variable* is an array, the *expr* is treated as if it were an array of the same shape  
 3 as *variable* with every element of the array equal to the scalar value of *expr*.
- 4 If *variable* is an array, the assignment is performed element-by-element on corresponding array elements  
 5 of *variable* and *expr*.

**NOTE 7.37**

For example, if A and B are arrays of the same shape, the array intrinsic assignment

```
A = B
```

assigns the corresponding elements of B to those of A; that is, the first element of B is assigned to the first element of A, the second element of B is assigned to the second element of A, etc.

If C is an allocatable array of rank 1, then

```
C = PACK(ARRAY,ARRAY>0)
```

will cause C to contain all the positive elements of ARRAY in array element order; if C is not allocated or is allocated with the wrong size, it will be re-allocated to be of the correct size to hold the result of PACK.

- 6 The processor may perform the element-by-element assignment in any order.

**NOTE 7.38**

For example, the following program segment results in the values of the elements of array X being reversed:

```
REAL X (10)
...
X (1:10) = X (10:1:-1)
```

- 1 For a numeric intrinsic assignment statement, *variable* and *expr* may have different numeric types or  
 2 different kind type parameters, in which case the value of *expr* is converted to the type and kind type  
 3 parameter of *variable* according to the rules of Table 7.9.

Table 7.9: Numeric conversion and the assignment statement

Type of <i>variable</i>	Value Assigned
integer	INT ( <i>expr</i> , KIND = KIND ( <i>variable</i> ))
real	REAL ( <i>expr</i> , KIND = KIND ( <i>variable</i> ))
complex	CMPLX ( <i>expr</i> , KIND = KIND ( <i>variable</i> ))
Note: The functions INT, REAL, CMPLX, and KIND are the generic functions defined in 13.7.	

- 4 For a logical intrinsic assignment statement, *variable* and *expr* may have different kind type parameters,  
 5 in which case the value of *expr* is converted to the kind type parameter of *variable*.
- 6 For a character intrinsic assignment statement, *variable* and *expr* may have different character length  
 7 parameters in which case the conversion of *expr* to the length of *variable* is as follows:
- 8 (1) If the length of *variable* is less than that of *expr*, the value of *expr* is truncated from the  
 9 right until it is the same length as *variable*.
- 10 (2) If the length of *variable* is greater than that of *expr*, the value of *expr* is extended on the  
 11 right with blanks until it is the same length as *variable*.
- 12 If *variable* and *expr* have different kind type parameters, each character *c* in *expr* is converted to the  
 13 kind type parameter of *variable* by ACHAR(IACHAR(*c*),KIND(*variable*)).

**NOTE 7.39**

For nondefault character types, the blank padding character is processor dependent. When assigning a character expression to a variable of a different kind, each character of the expression that is not representable in the kind of the variable is replaced by a processor-dependent character.

- 14 A derived-type intrinsic assignment is performed as if each component of *variable* were assigned from  
 15 the corresponding component of *expr* using pointer assignment (7.4.2) for each pointer component,  
 16 defined assignment for each nonpointer nonallocatable component of a type that has a type-bound  
 17 defined assignment consistent with the component, and intrinsic assignment for each other nonpointer  
 18 nonallocatable component. For an allocatable component the following sequence of operations is applied:
- 19 (1) If the component of *variable* is allocated, it is deallocated.
- 20 (2) If the component of *expr* is allocated, the corresponding component of *variable* is allocated  
 21 with the same dynamic type and type parameters as the component of *expr*. If it is an array,  
 22 it is allocated with the same bounds. The value of the component of *expr* is then assigned  
 23 to the corresponding component of *variable* using defined assignment if the declared type  
 24 of the component has a type-bound defined assignment consistent with the component, and  
 25 intrinsic assignment for the dynamic type of that component otherwise.
- 26 The processor may perform the component-by-component assignment in any order or by any means that  
 27 has the same effect.

**NOTE 7.40**

For an example of a derived-type intrinsic assignment statement, if C and D are of the same derived type with a pointer component P and nonpointer components S, T, U, and V of type integer, logical, character, and another derived type, respectively, the intrinsic

C = D

**NOTE 7.40 (cont.)**

pointer assigns D % P to C % P. It assigns D % S to C % S, D % T to C % T, and D % U to C % U using intrinsic assignment. It assigns D % V to C % V using defined assignment if objects of that type have a compatible type-bound defined assignment, and intrinsic assignment otherwise.

**NOTE 7.41**

If an allocatable component of *expr* is unallocated, the corresponding component of *variable* has an allocation status of unallocated after execution of the assignment.

- 1 When *variable* is a subobject, the assignment does not affect the definition status or value of other parts  
 2 of the object. For example, if *variable* is an array section, the assignment does not affect the definition  
 3 status or value of the elements of the array not specified by the array section.

**4 7.4.1.4 Defined assignment statement**

5 A **defined assignment statement** is an assignment statement that is defined by a subroutine and a  
 6 generic interface (4.5.1, 12.3.2.1.2) that specifies ASSIGNMENT (=). A **defined elemental assign-  
 7 ment statement** is a defined assignment statement for which the subroutine is elemental (12.7).

8 A subroutine defines the defined assignment  $x_1 = x_2$  if

- 9 (1) The subroutine is specified with a SUBROUTINE (12.5.2.2) or ENTRY (12.5.2.4) statement  
 10 that specifies two dummy arguments,  $d_1$  and  $d_2$ ,  
 11 (2) Either  
 12 (a) A generic interface (12.3.2.1) provides the subroutine with a *generic-spec* of ASSIGN-  
 13 MENT (=), or  
 14 (b) There is a generic binding (4.5.1) in the declared type of  $x_1$  or  $x_2$  with a *generic-spec*  
 15 of ASSIGNMENT (=) and there is a corresponding binding to the subroutine in the  
 16 dynamic type of  $x_1$  or  $x_2$ , respectively,  
 17 (3) The types of  $d_1$  and  $d_2$  are compatible with the dynamic types of  $x_1$  and  $x_2$ , respectively,  
 18 (4) The type parameters, if any, of  $d_1$  and  $d_2$  match the corresponding type parameters of  $x_1$   
 19 and  $x_2$ , respectively, and  
 20 (5) Either  
 21 (a) The ranks of  $x_1$  and  $x_2$  match those of  $d_1$  and  $d_2$  or  
 22 (b) The subroutine is elemental,  $x_1$  and  $x_2$  are conformable, and there is no other sub-  
 23 routine that defines the operation.

24 If  $d_1$  or  $d_2$  is an array, the shapes of  $x_1$  and  $x_2$  shall match the shapes of  $d_1$  and  $d_2$ , respectively.

**25 7.4.1.5 Interpretation of defined assignment statements**

26 The interpretation of a defined assignment is provided by the subroutine that defines it.

27 If the defined assignment is an elemental assignment and the *variable* in the assignment is an array, the  
 28 defined assignment is performed element-by-element, in any order, on corresponding elements of *variable*  
 29 and *expr*. If *expr* is a scalar, it is treated as if it were an array of the same shape as *variable* with every  
 30 element of the array equal to the scalar value of *expr*.

**NOTE 7.42**

The rules of defined assignment (12.3.2.1.2), procedure references (12.4), subroutine references (12.4.3), and elemental subroutine arguments (12.7.3) ensure that the defined assignment has the

**NOTE 7.42 (cont.)**

same effect as if the evaluation of all operations in  $x_2$  and  $x_1$  occurs before any portion of  $x_1$  is defined.

**1 7.4.2 Pointer assignment**

2 Pointer assignment causes a pointer to become associated with a target or causes its pointer association  
3 status to become disassociated or undefined. Any previous association between the pointer and a target  
4 is broken.

5 Pointer assignment for a pointer component of a structure may also take place by execution of a derived-  
6 type intrinsic assignment statement (7.4.1.3).

7 A pointer may also become associated with a target by allocation of the pointer.

8 R735 *pointer-assignment-stmt*      **is** *data-pointer-object* [ (*bounds-spec-list*) ] => *data-target*  
9    **or** *data-pointer-object* (*bounds-remapping-list*) => *data-target*  
10    **or** *proc-pointer-object* => *proc-target*  
11 R736 *data-pointer-object*             **is** *variable-name*  
12    **or** *variable* % *data-pointer-component-name*

13 C716 (R735) If *data-target* is not unlimited polymorphic, *data-pointer-object* shall be type compatible  
14 (5.1.1.2) with it, and the corresponding kind type parameters shall be equal.

15 C717 (R735) If *data-target* is unlimited polymorphic, *data-pointer-object* shall be unlimited polymor-  
16 phic, of a sequence derived type, or of a type with the BIND attribute.

17 C718 (R735) If *bounds-spec-list* is specified, the number of *bounds-specs* shall equal the rank of *data-*  
18 *pointer-object*.

19 C719 (R735) If *bounds-remapping-list* is specified, the number of *bounds-remappings* shall equal the  
20 rank of *data-pointer-object*.

21 C720 (R735) If *bounds-remapping-list* is specified, *data-target* shall have rank one; otherwise, the  
22 ranks of *data-pointer-object* and *data-target* shall be the same.

23 C721 (R736) A *variable-name* shall have the POINTER attribute.

24 C722 (R736) A *data-pointer-component-name* shall be the name of a component of *variable* that is a  
25 data pointer.

26 R737 *bounds-spec*                        **is** *lower-bound-expr* :  
27 R738 *bounds-remapping*                 **is** *lower-bound-expr* : *upper-bound-expr*  
28 R739 *data-target*                        **is** *variable*  
29    **or** *expr*

30 C723 (R739) A *variable* shall have either the TARGET or POINTER attribute, and shall not be an  
31 array section with a vector subscript.

32 C724 (R739) An *expr* shall be a reference to a function whose result is a data pointer.

33 R740 *proc-pointer-object*             **is** *proc-pointer-name*  
34    **or** *proc-component-ref*  
35 R741 *proc-component-ref*               **is** *variable* % *procedure-component-name*

36 C725 (R741) the *procedure-component-name* shall be the name of a procedure pointer component of  
37 the declared type of *variable*.

1 R742 *proc-target* is *expr*  
 2 or *procedure-name*  
 3 or *proc-component-ref*

4 C726 (R742) An *expr* shall be a reference to a function whose result is a procedure pointer.

5 C727 (R742) A *procedure-name* shall be the name of an external, module, or dummy procedure, a  
 6 specific intrinsic function listed in 13.6 and not marked with a bullet (●), or a procedure pointer.

7 C728 (R742) The *proc-target* shall not be a nonintrinsic elemental procedure.

#### 8 7.4.2.1 Data pointer assignment

9 If *data-pointer-object* is not polymorphic and *data-target* is polymorphic with dynamic type that differs  
 10 from its declared type, the assignment target is the ancestor component of *data-target* that has the type  
 11 of *data-pointer-object*. Otherwise, the assignment target is *data-target*.

12 If *data-target* is not a pointer, *data-pointer-object* becomes pointer associated with the assignment target.  
 13 Otherwise, the pointer association status of *data-pointer-object* becomes that of *data-target*; if *data-target*  
 14 is associated with an object, *data-pointer-object* becomes associated with the assignment target. If *data-*  
 15 *target* is allocatable, it shall be allocated.

16 If *data-pointer-object* is polymorphic (5.1.1.2), it assumes the dynamic type of *data-target*. If *data-*  
 17 *pointer-object* is of sequence derived type or a type with the BIND attribute, the dynamic type of  
 18 *data-target* shall be that derived type.

19 If *data-target* is a disassociated pointer, all nondeferred type parameters of the declared type of *data-*  
 20 *pointer-object* that correspond to nondeferred type parameters of *data-target* shall have the same values  
 21 as the corresponding type parameters of *data-target*. Otherwise, all nondeferred type parameters of the  
 22 declared type of *data-pointer-object* shall have the same values as the corresponding type parameters of  
 23 *data-target*.

24 If *data-pointer-object* has nondeferred type parameters that correspond to deferred type parameters of  
 25 *data-target*, *data-target* shall not be a pointer with undefined association status.

26 If *bounds-remapping-list* is specified, *data-target* shall not be a disassociated or undefined pointer, and  
 27 the size of *data-target* shall not be less than the size of *data-pointer-object*. The elements of the target  
 28 of *data-pointer-object*, in array element order (6.2.2.2), are the first SIZE(*data-pointer-object*) elements  
 29 of *data-target*.

30 If no *bounds-remapping-list* is specified, the extent of a dimension of *data-pointer-object* is the extent of  
 31 the corresponding dimension of *data-target*. If *bounds-spec-list* appears, it specifies the lower bounds;  
 32 otherwise, the lower bound of each dimension is the result of the intrinsic function LBOUND (13.7.60)  
 33 applied to the corresponding dimension of *data-target*. The upper bound of each dimension is one less  
 34 than the sum of the lower bound and the extent.

#### 35 7.4.2.2 Procedure pointer assignment

36 If the *proc-target* is not a pointer, *proc-pointer-object* becomes pointer associated with *proc-target*. Other-  
 37 wise, the pointer association status of *proc-pointer-object* becomes that of *proc-target*; if *proc-target* is  
 38 associated with a procedure, *proc-pointer-object* becomes associated with the same procedure.

39 If *proc-pointer-object* has an explicit interface, its characteristics shall be the same as *proc-target* except  
 40 that *proc-target* may be pure even if *proc-pointer-object* is not pure and *proc-target* may be an elemental  
 41 intrinsic procedure even if *proc-pointer-object* is not elemental.

42 If the characteristics of *proc-pointer-object* or *proc-target* are such that an explicit interface is required,

- 1 both *proc-pointer-object* and *proc-target* shall have an explicit interface.
- 2 If *proc-pointer-object* has an implicit interface and is explicitly typed or referenced as a function, *proc-*
- 3 *target* shall be a function. If *proc-pointer-object* has an implicit interface and is referenced as a subroutine,
- 4 *proc-target* shall be a subroutine.
- 5 If *proc-target* and *proc-pointer-object* are functions, they shall have the same type; corresponding type
- 6 parameters shall either both be deferred or both have the same value.
- 7 If *procedure-name* is a specific procedure name that is also a generic name, only the specific procedure
- 8 is associated with pointer-object.

### 9 7.4.2.3 Examples

#### NOTE 7.43

The following are examples of pointer assignment statements. (See Note 12.14 for declarations of P and BESSEL.)

```

NEW_NODE % LEFT => CURRENT_NODE
SIMPLE_NAME => TARGET_STRUCTURE % SUBSTRUCT % COMPONENT
PTR => NULL ( )
ROW => MAT2D (N, :)
WINDOW => MAT2D (I-1:I+1, J-1:J+1)
POINTER_OBJECT => POINTER_FUNCTION (ARG_1, ARG_2)
EVERY_OTHER => VECTOR (1:N:2)
WINDOW2 (0:, 0:) => MAT2D (ML:MU, NL:NU)
! P is a procedure pointer and BESSEL is a procedure with a
! compatible interface.
P => BESSEL

! Likewise for a structure component.
STRUCT % COMPONENT => BESSEL

```

#### NOTE 7.44

It is possible to obtain high-rank views of (parts of) rank-one objects by specifying upper bounds in pointer assignment statements. Consider the following example, in which a matrix is under consideration. The matrix is stored as a rank-one object in MYDATA because its diagonal is needed for some reason – the diagonal cannot be gotten as a single object from a rank-two representation. The matrix is represented as a rank-two view of MYDATA.

```

real, target :: MYDATA ( NR*NC )      ! An automatic array
real, pointer :: MATRIX ( :, : )      ! A rank-two view of MYDATA
real, pointer :: VIEW_DIAG ( : )
MATRIX( 1:NR, 1:NC ) => MYDATA        ! The MATRIX view of the data
VIEW_DIAG => MYDATA( 1::NR+1 )        ! The diagonal of MATRIX

```

Rows, columns, or blocks of the matrix can be accessed as sections of MATRIX.

### 10 7.4.3 Masked array assignment – WHERE

- 11 The masked array assignment is used to mask the evaluation of expressions and assignment of values in
- 12 array assignment statements, according to the value of a logical array expression.

### 1 7.4.3.1 General form of the masked array assignment

2 A **masked array assignment** is either a WHERE statement or a WHERE construct.

3	R743	<i>where-stmt</i>	<b>is</b>	WHERE ( <i>mask-expr</i> ) <i>where-assignment-stmt</i>
4	R744	<i>where-construct</i>	<b>is</b>	<i>where-construct-stmt</i>
5				[ <i>where-body-construct</i> ] ...
6				[ <i>masked-elsewhere-stmt</i>
7				[ <i>where-body-construct</i> ] ... ] ...
8				[ <i>elsewhere-stmt</i>
9				[ <i>where-body-construct</i> ] ... ]
10				<i>end-where-stmt</i>
11	R745	<i>where-construct-stmt</i>	<b>is</b>	[ <i>where-construct-name</i> ;] WHERE ( <i>mask-expr</i> )
12	R746	<i>where-body-construct</i>	<b>is</b>	<i>where-assignment-stmt</i>
13			<b>or</b>	<i>where-stmt</i>
14			<b>or</b>	<i>where-construct</i>
15	R747	<i>where-assignment-stmt</i>	<b>is</b>	<i>assignment-stmt</i>
16	R748	<i>mask-expr</i>	<b>is</b>	<i>logical-expr</i>
17	R749	<i>masked-elsewhere-stmt</i>	<b>is</b>	ELSEWHERE ( <i>mask-expr</i> ) [ <i>where-construct-name</i> ]
18	R750	<i>elsewhere-stmt</i>	<b>is</b>	ELSEWHERE [ <i>where-construct-name</i> ]
19	R751	<i>end-where-stmt</i>	<b>is</b>	END WHERE [ <i>where-construct-name</i> ]

20 C729 (R747) A *where-assignment-stmt* that is a defined assignment shall be elemental.

21 C730 (R744) If the *where-construct-stmt* is identified by a *where-construct-name*, the corresponding  
 22 *end-where-stmt* shall specify the same *where-construct-name*. If the *where-construct-stmt* is  
 23 not identified by a *where-construct-name*, the corresponding *end-where-stmt* shall not specify  
 24 a *where-construct-name*. If an *elsewhere-stmt* or a *masked-elsewhere-stmt* is identified by a  
 25 *where-construct-name*, the corresponding *where-construct-stmt* shall specify the same *where-*  
 26 *construct-name*.

27 C731 (R746) A statement that is part of a *where-body-construct* shall not be a branch target statement.

28 If a *where-construct* contains a *where-stmt*, a *masked-elsewhere-stmt*, or another *where-construct* then  
 29 each *mask-expr* within the *where-construct* shall have the same shape. In each *where-assignment-stmt*,  
 30 the *mask-expr* and the *variable* being defined shall be arrays of the same shape.

#### NOTE 7.45

Examples of a masked array assignment are:

```

WHERE (TEMP > 100.0) TEMP = TEMP - REDUCE_TEMP
WHERE (PRESSURE <= 1.0)
  PRESSURE = PRESSURE + INC_PRESSURE
  TEMP = TEMP - 5.0
ELSEWHERE
  RAINING = .TRUE.
END WHERE

```

### 31 7.4.3.2 Interpretation of masked array assignments

32 When a WHERE statement or a *where-construct-stmt* is executed, a control mask is established. In  
 33 addition, when a WHERE construct statement is executed, a pending control mask is established. If  
 34 the statement does not appear as part of a *where-body-construct*, the *mask-expr* of the statement is  
 35 evaluated, and the control mask is established to be the value of *mask-expr*. The pending control mask  
 36 is established to have the value *.NOT. mask-expr* upon execution of a WHERE construct statement that

- 1 does not appear as part of a *where-body-construct*. The *mask-expr* is evaluated only once.
- 2 Each statement in a WHERE construct is executed in sequence.
- 3 Upon execution of a *masked-elsewhere-stmt*, the following actions take place in sequence:
- 4 (1) The control mask  $m_c$  is established to have the value of the pending control mask.
  - 5 (2) The pending control mask is established to have the value  $m_c$  .AND. (.NOT. *mask-expr*).
  - 6 (3) The control mask  $m_c$  is established to have the value  $m_c$  .AND. *mask-expr*.
- 7 The *mask-expr* is evaluated at most once.
- 8 Upon execution of an ELSEWHERE statement, the control mask is established to have the value of the  
9 pending control mask. No new pending control mask value is established.
- 10 Upon execution of an ENDWHERE statement, the control mask and pending control mask are es-  
11 tablished to have the values they had prior to the execution of the corresponding WHERE construct  
12 statement. Following the execution of a WHERE statement that appears as a *where-body-construct*, the  
13 control mask is established to have the value it had prior to the execution of the WHERE statement.

**NOTE 7.46**

The establishment of control masks and the pending control mask is illustrated with the following example:

```

WHERE(cond1)      ! Statement 1
. . .
ELSEWHERE(cond2) ! Statement 2
. . .
ELSEWHERE        ! Statement 3
. . .
END WHERE

```

Following execution of statement 1, the control mask has the value cond1 and the pending control mask has the value .NOT. cond1. Following execution of statement 2, the control mask has the value (.NOT. cond1) .AND. cond2 and the pending control mask has the value (.NOT. cond1) .AND. (.NOT. cond2). Following execution of statement 3, the control mask has the value (.NOT. cond1) .AND. (.NOT. cond2). The false condition values are propagated through the execution of the masked ELSEWHERE statement.

- 14 Upon execution of a WHERE construct statement that is part of a *where-body-construct*, the pending  
15 control mask is established to have the value  $m_c$  .AND. (.NOT. *mask-expr*). The control mask is then  
16 established to have the value  $m_c$  .AND. *mask-expr*. The *mask-expr* is evaluated at most once.
- 17 Upon execution of a WHERE statement that is part of a *where-body-construct*, the control mask is  
18 established to have the value  $m_c$  .AND. *mask-expr*. The pending mask is not altered.
- 19 If a nonelemental function reference occurs in the *expr* or *variable* of a *where-assignment-stmt* or in a  
20 *mask-expr*, the function is evaluated without any masked control; that is, all of its argument expressions  
21 are fully evaluated and the function is fully evaluated. If the result is an array and the reference is not  
22 within the argument list of a nonelemental function, elements corresponding to true values in the control  
23 mask are selected for use in evaluating the *expr*, *variable* or *mask-expr*.
- 24 If an elemental operation or function reference occurs in the *expr* or *variable* of a *where-assignment-stmt*  
25 or in a *mask-expr*, and is not within the argument list of a nonelemental function reference, the operation  
26 is performed or the function is evaluated only for the elements corresponding to true values of the control  
27 mask.

- 1 If an array constructor appears in a *where-assignment-stmt* or in a *mask-expr*, the array constructor is  
 2 evaluated without any masked control and then the *where-assignment-stmt* is executed or the *mask-expr*  
 3 is evaluated.
- 4 When a *where-assignment-stmt* is executed, the values of *expr* that correspond to true values of the  
 5 control mask are assigned to the corresponding elements of *variable*.
- 6 The value of the control mask is established by the execution of a WHERE statement, a WHERE con-  
 7 struct statement, an ELSEWHERE statement, a masked ELSEWHERE statement, or an ENDWHERE  
 8 statement. Subsequent changes to the value of entities in a *mask-expr* have no effect on the value of the  
 9 control mask. The execution of a function reference in the mask expression of a WHERE statement is  
 10 permitted to affect entities in the assignment statement.

**NOTE 7.47**

Examples of function references in masked array assignments are:

```

WHERE (A > 0.0)
A = LOG (A)           ! LOG is invoked only for positive elements.
A = A / SUM (LOG (A)) ! LOG is invoked for all elements
                    ! because SUM is transformational.
END WHERE

```

#### 11 7.4.4 FORALL

- 12 FORALL constructs and statements are used to control the execution of assignment and pointer assign-  
 13 ment statements with selection by sets of index values and an optional mask expression.

##### 14 7.4.4.1 The FORALL Construct

- 15 The FORALL construct allows multiple assignments, masked array (WHERE) assignments, and nested  
 16 FORALL constructs and statements to be controlled by a single *forall-triplet-spec-list* and *scalar-mask-*  
 17 *expr*.

- |    |      |                               |  |   |
|----|------|-------------------------------|--|---|
| 18 | R752 | <i>forall-construct</i>       | <b>is</b>  | <i>forall-construct-stmt</i><br>[ <i>forall-body-construct</i> ] ...<br><i>end-forall-stmt</i>  |
| 21 | R753 | <i>forall-construct-stmt</i>  | <b>is</b>  | [ <i>forall-construct-name</i> :] FORALL <i>forall-header</i>   |
| 22 | R754 | <i>forall-header</i>          | <b>is</b>  | ( <i>forall-triplet-spec-list</i> [, <i>scalar-mask-expr</i> ] )  |
| 23 | R755 | <i>forall-triplet-spec</i>    | <b>is</b>  | <i>index-name</i> = <i>subscript</i> : <i>subscript</i> [ : <i>stride</i> ]   |
| 24 | R618 | <i>subscript</i>              | <b>is</b>  | <i>scalar-int-expr</i>  |
| 25 | R621 | <i>stride</i>                 | <b>is</b>  | <i>scalar-int-expr</i>  |
| 26 | R756 | <i>forall-body-construct</i>  | <b>is</b>  | <i>forall-assignment-stmt</i><br><b>or</b> <i>where-stmt</i><br><b>or</b> <i>where-construct</i><br><b>or</b> <i>forall-construct</i><br><b>or</b> <i>forall-stmt</i> |
| 31 | R757 | <i>forall-assignment-stmt</i> | <b>is</b>  | <i>assignment-stmt</i><br><b>or</b> <i>pointer-assignment-stmt</i>  |
| 33 | R758 | <i>end-forall-stmt</i>        | <b>is</b>  | END FORALL [ <i>forall-construct-name</i> ]   |
| 34 | C732 | (R758)                        | If the <i>forall-construct-stmt</i> has a <i>forall-construct-name</i> , the <i>end-forall-stmt</i> shall have<br>35 the same <i>forall-construct-name</i> . If the <i>end-forall-stmt</i> has a <i>forall-construct-name</i> , the <i>forall-</i> |   |

- 1            *construct-stmt* shall have the same *forall-construct-name*.
- 2 C733 (R754) The *scalar-mask-expr* shall be scalar and of type logical.
- 3 C734 (R754) Any procedure referenced in the *scalar-mask-expr*, including one referenced by a defined  
4 operation, shall be a pure procedure (12.6).
- 5 C735 (R755) The *index-name* shall be a named scalar variable of type integer.
- 6 C736 (R755) A *subscript* or *stride* in a *forall-triplet-spec* shall not contain a reference to any *index-  
7 name* in the *forall-triplet-spec-list* in which it appears.
- 8 C737 (R756) A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-  
9 construct*.
- 10 C738 (R756) Any procedure referenced in a *forall-body-construct*, including one referenced by a defined  
11 operation, assignment, or finalization, shall be a pure procedure.
- 12 C739 (R756) A *forall-body-construct* shall not be a branch target.

**NOTE 7.48**

An example of a FORALL construct is:

```
REAL :: A(10, 10), B(10, 10) = 1.0
. . .
FORALL (I = 1:10, J = 1:10, B(I, J) /= 0.0)
  A(I, J) = REAL (I + J - 2)
  B(I, J) = A(I, J) + B(I, J) * REAL (I * J)
END FORALL
```

**NOTE 7.49**

An assignment statement that is a FORALL body construct may be a scalar or array assignment statement, or a defined assignment statement. The variable being defined will normally use each index name in the *forall-triplet-spec-list*. For example

```
FORALL (I = 1:N, J = 1:N)
  A(:, I, :, J) = 1.0 / REAL(I + J - 1)
END FORALL
```

broadcasts scalar values to rank-two subarrays of A.

**NOTE 7.50**

An example of a FORALL construct containing a pointer assignment statement is:

```
TYPE ELEMENT
  REAL ELEMENT_WT
  CHARACTER (32), POINTER :: NAME
END TYPE ELEMENT
TYPE(ELEMENT) CHART(200)
REAL WEIGHTS (1000)
CHARACTER (32), TARGET :: NAMES (1000)
. . .
FORALL (I = 1:200, WEIGHTS (I + N - 1) > .5)
  CHART(I) % ELEMENT_WT = WEIGHTS (I + N - 1)
```

**NOTE 7.50 (cont.)**

```

    CHART(I) % NAME => NAMES (I + N - 1)
  END FORALL

```

The results of this FORALL construct cannot be achieved with a WHERE construct because a pointer assignment statement is not permitted in a WHERE construct.

- 1 An *index-name* in a *forall-construct* has a scope of the construct (16.3). It is a scalar variable that has
- 2 the type and type parameters that it would have if it were the name of a variable in the scoping unit
- 3 that includes the FORALL, and this type shall be integer type; it has no other attributes.

**NOTE 7.51**

The use of *index-name* variables in a FORALL construct does not affect variables of the same name, for example:

```

  INTEGER :: X = -1
  REAL A(5, 4)
  J = 100
  . . .
  FORALL (X = 1:5, J = 1:4)
    A (X, J) = J
  END FORALL

```

After execution of the FORALL, the variables X and J have the values -1 and 100 and A has the value

```

    1 2 3 4
    1 2 3 4
    1 2 3 4
    1 2 3 4
    1 2 3 4

```

**4 7.4.4.2 Execution of the FORALL construct**

5 There are three stages in the execution of a FORALL construct:

- 6 (1) Determination of the values for *index-name* variables,
- 7 (2) Evaluation of the *scalar-mask-expr*, and
- 8 (3) Execution of the FORALL body constructs.

**9 7.4.4.2.1 Determination of the values for index variables**

10 The subscript and stride expressions in the *forall-triplet-spec-list* are evaluated. These expressions may  
 11 be evaluated in any order. The set of values that a particular *index-name* variable assumes is determined  
 12 as follows:

- 13 (1) The lower bound  $m_1$ , the upper bound  $m_2$ , and the stride  $m_3$  are of type integer with the  
 14 same kind type parameter as the *index-name*. Their values are established by evaluating  
 15 the first subscript, the second subscript, and the stride expressions, respectively, including,  
 16 if necessary, conversion to the kind type parameter of the *index-name* according to the rules  
 17 for numeric conversion (Table 7.9). If a stride does not appear,  $m_3$  has the value 1. The  
 18 value  $m_3$  shall not be zero.
- 19 (2) Let the value of *max* be  $(m_2 - m_1 + m_3) / m_3$ . If  $max \leq 0$  for some *index-name*, the execution

- 1 of the construct is complete. Otherwise, the set of values for the *index-name* is  
 2  $m_1 + (k - 1) \times m_3$  where  $k = 1, 2, \dots, max$ .
- 3 The set of combinations of *index-name* values is the Cartesian product of the sets defined by each triplet  
 4 specification. An *index-name* becomes defined when this set is evaluated.

**NOTE 7.52**

The *stride* may be positive or negative; the FORALL body constructs are executed as long as  $max > 0$ . For the *forall-triplet-spec*

```
I = 10:1:-1
```

*max* has the value 10

#### 5 7.4.4.2.2 Evaluation of the mask expression

- 6 The *scalar-mask-expr*, if any, is evaluated for each combination of *index-name* values. If the *scalar-*  
 7 *mask-expr* is not present, it is as if it were present with the value true. The *index-name* variables may  
 8 be primaries in the *scalar-mask-expr*.
- 9 The **active combination of *index-name* values** is defined to be the subset of all possible combinations  
 10 (7.4.4.2.1) for which the *scalar-mask-expr* has the value true.

**NOTE 7.53**

The *index-name* variables may appear in the mask, for example

```
FORALL (I=1:10, J=1:10, A(I) > 0.0 .AND. B(J) < 1.0)
. . .
```

#### 11 7.4.4.2.3 Execution of the FORALL body constructs

- 12 The *forall-body-constructs* are executed in the order in which they appear. Each construct is executed  
 13 for all active combinations of the *index-name* values with the following interpretation:
- 14 Execution of a *forall-assignment-stmt* that is an *assignment-stmt* causes the evaluation of *expr* and all  
 15 expressions within *variable* for all active combinations of *index-name* values. These evaluations may be  
 16 done in any order. After all these evaluations have been performed, each *expr* value is assigned to the  
 17 corresponding *variable*. The assignments may occur in any order.
- 18 Execution of a *forall-assignment-stmt* that is a *pointer-assignment-stmt* causes the evaluation of all  
 19 expressions within *data-target* and *data-pointer-object* or *proc-target* and *proc-pointer-object*, the de-  
 20 termination of any pointers within *data-pointer-object* or *proc-pointer-object*, and the determination of  
 21 the target for all active combinations of *index-name* values. These evaluations may be done in any  
 22 order. After all these evaluations have been performed, each *data-pointer-object* or *proc-pointer-object*  
 23 is associated with the corresponding target. These associations may occur in any order.
- 24 In a *forall-assignment-stmt*, a defined assignment subroutine shall not reference any *variable* that be-  
 25 comes defined by the statement.

**NOTE 7.54**

The following FORALL construct contains two assignment statements. The assignment to array B uses the values of array A computed in the previous statement, not the values A had prior to execution of the FORALL.

**NOTE 7.54 (cont.)**

```

FORALL (I = 2:N-1, J = 2:N-1 )
  A (I, J) = A(I, J-1) + A(I, J+1) + A(I-1, J) + A(I+1, J)
  B (I, J) = 1.0 / A(I, J)
END FORALL

```

Computations that would otherwise cause error conditions can be avoided by using an appropriate *scalar-mask-expr* that limits the active combinations of the *index-name* values. For example:

```

FORALL (I = 1:N, Y(I) /= 0.0)
  X(I) = 1.0 / Y(I)
END FORALL

```

- 1 Each statement in a *where-construct* (7.4.3) within a *forall-construct* is executed in sequence. When
- 2 a *where-stmt*, *where-construct-stmt* or *masked-elsewhere-stmt* is executed, the statement's *mask-expr* is
- 3 evaluated for all active combinations of *index-name* values as determined by the outer *forall-constructs*,
- 4 masked by any control mask corresponding to outer *where-constructs*. Any *where-assignment-stmt* is
- 5 executed for all active combinations of *index-name* values, masked by the control mask in effect for the
- 6 *where-assignment-stmt*.

**NOTE 7.55**

This FORALL construct contains a WHERE statement and an assignment statement.

```

INTEGER A(5,4), B(5,4)
FORALL ( I = 1:5 )
  WHERE ( A(I,:) == 0 ) A(I,:) = I
  B (I,:) = I / A(I,:)
END FORALL

```

When executed with the input array

```

      0  0  0  0
      1  1  1  0
A =   2  2  0  2
      1  0  2  3
      0  0  0  0

```

the results will be

```

      1  1  1  1
      1  1  1  2
A =   2  2  3  2
      1  4  2  3
      5  5  5  5
      1  1  1  1
      2  2  2  1
B =   1  1  1  1
      4  1  2  1
      1  1  1  1

```

For an example of a FORALL construct containing a WHERE construct with an ELSEWHERE statement, see C.4.5.

- 7 Execution of a *forall-stmt* or *forall-construct* causes the evaluation of the *subscript* and *stride* expressions
- 8 in the *forall-triplet-spec-list* for all active combinations of the *index-name* values of the outer FORALL
- 9 construct. The set of combinations of *index-name* values for the inner FORALL is the union of the sets
- 10 defined by these bounds and strides for each active combination of the outer *index-name* values; it also

- 1 includes the outer *index-name* values. The *scalar-mask-expr* is then evaluated for all combinations of the  
 2 *index-name* values of the inner construct to produce a set of active combinations for the inner construct.  
 3 If there is no *scalar-mask-expr*, it is as if it were present with the value `.TRUE.`. Each statement in the  
 4 inner FORALL is then executed for each active combination of the *index-name* values.

**NOTE 7.56**

This FORALL construct contains a nested FORALL construct. It assigns the transpose of the strict lower triangle of array A (the section below the main diagonal) to the strict upper triangle of A.

```

INTEGER A (3, 3)
FORALL (I = 1:N-1 )
  FORALL ( J=I+1:N )
    A(I,J) = A(J,I)
  END FORALL
END FORALL

```

If prior to execution  $N = 3$  and

```

      0  3  6
A =   1  4  7
      2  5  8

```

then after execution

```

      0  1  2
A =   1  4  5
      2  5  8

```

### 5 7.4.4.3 The FORALL statement

- 6 The FORALL statement allows a single assignment statement or pointer assignment to be controlled by  
 7 a set of index values and an optional mask expression.
- 8 R759 *forall-stmt* is FORALL *forall-header forall-assignment-stmt*
- 9 A FORALL statement is equivalent to a FORALL construct containing a single *forall-body-construct*  
 10 that is a *forall-assignment-stmt*.
- 11 The scope of an *index-name* in a *forall-stmt* is the statement itself (16.3).

**NOTE 7.57**

Examples of FORALL statements are:

```
FORALL (I=1:N) A(I,I) = X(I)
```

This statement assigns the elements of vector X to the elements of the main diagonal of matrix A.

```
FORALL (I = 1:N, J = 1:N) X(I,J) = 1.0 / REAL (I+J-1)
```

Array element X(I,J) is assigned the value  $(1.0 / \text{REAL}(I+J-1))$  for values of I and J between 1 and N, inclusive.

**NOTE 7.57 (cont.)**

```
FORALL (I=1:N, J=1:N, Y(I,J) /= 0 .AND. I /= J) X(I,J) = 1.0 / Y(I,J)
```

This statement takes the reciprocal of each nonzero off-diagonal element of array Y(1:N, 1:N) and assigns it to the corresponding element of array X. Elements of Y that are zero or on the diagonal do not participate, and no assignments are made to the corresponding elements of X. The results from the execution of the example in Note 7.56 could be obtained with a single FORALL statement:

```
FORALL ( I = 1:N-1, J=1:N, J > I ) A(I,J) = A(J,I)
```

For more examples of FORALL statements, see C.4.6.

1 **7.4.4.4 Restrictions on FORALL constructs and statements**

- 2 A many-to-one assignment is more than one assignment to the same object, or association of more  
 3 than one target with the same pointer, whether the object is referenced directly or indirectly through a  
 4 pointer. A many-to-one assignment shall not occur within a single statement in a FORALL construct or  
 5 statement. It is possible to assign or pointer assign to the same object in different assignment statements  
 6 in a FORALL construct.

**NOTE 7.58**

The appearance of each *index-name* in the identification of the left-hand side of an assignment statement is helpful in eliminating many-to-one assignments, but it is not sufficient to guarantee there will be none. For example, the following is allowed

```
FORALL (I = 1:10)
  A (INDEX (I)) = B(I)
END FORALL
```

if and only if INDEX(1:10) contains no repeated values.

- 7 Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct shall  
 8 not have the same *index-name*. The *forall-header* expressions within a nested FORALL may depend on  
 9 the values of outer *index-name* variables.

## 1 Section 8: Execution control

2 The execution sequence may be controlled by constructs containing blocks and by certain executable  
3 statements that are used to alter the execution sequence.

### 4 8.1 Executable constructs containing blocks

5 The following are executable constructs that contain blocks:

- 6 (1) ASSOCIATE construct
- 7 (2) CASE construct
- 8 (3) DO construct
- 9 (4) IF construct
- 10 (5) SELECT TYPE construct

11 There is also a nonblock form of the DO construct.

12 A **block** is a sequence of executable constructs that is treated as a unit.

13 R801 *block* **is** [ *execution-part-construct* ] ...

14 Executable constructs may be used to control which blocks of a program are executed or how many times  
15 a block is executed. Blocks are always bounded by statements that are particular to the construct in  
16 which they are embedded; however, in some forms of the DO construct, a sequence of executable constructs without  
17 a terminating boundary statement shall obey all other rules governing blocks (8.1.1).

#### NOTE 8.1

A block need not contain any executable constructs. Execution of such a block has no effect.

18 Any of these constructs may be named. If a construct is named, the name shall be the first lexical token  
19 of the first statement of the construct and the last lexical token of the construct. In fixed source form, the  
20 name preceding the construct shall be placed after character position 6.

21 A statement **belongs** to the innermost construct in which it appears unless it contains a construct name,  
22 in which case it belongs to the named construct.

#### NOTE 8.2

An example of a construct containing a block is:

```
IF (A > 0.0) THEN
  B = SQRT (A)  ! These two statements
  C = LOG (A)   ! form a block.
END IF
```

### 23 8.1.1 Rules governing blocks

#### 24 8.1.1.1 Executable constructs in blocks

25 If a block contains an executable construct, the executable construct shall be entirely within the block.

### 1 8.1.1.2 Control flow in blocks

2 Transfer of control to the interior of a block from outside the block is prohibited. Transfers within a  
3 block and transfers from the interior of a block to outside the block may occur.

4 Subroutine and function references (12.4.2, 12.4.3) may appear in a block.

### 5 8.1.1.3 Execution of a block

6 Execution of a block begins with the execution of the first executable construct in the block. Execution  
7 of the block is completed when the last executable construct in the sequence is executed or when a  
8 branch out of the block takes place.

#### NOTE 8.3

The action that takes place at the terminal boundary depends on the particular construct and on the block within that construct. It is usually a transfer of control.

## 9 8.1.2 IF construct

10 The **IF construct** selects for execution at most one of its constituent blocks. The selection is based  
11 on a sequence of logical expressions. The **IF statement** controls the execution of a single statement  
12 (8.1.2.4) based on a single logical expression.

### 13 8.1.2.1 Form of the IF construct

14	R802 <i>if-construct</i>	<b>is</b>	<i>if-then-stmt</i> <i>block</i> [ <i>else-if-stmt</i> <i>block</i> ] ... [ <i>else-stmt</i> <i>block</i> ] <i>end-if-stmt</i>
21	R803 <i>if-then-stmt</i>	<b>is</b>	[ <i>if-construct-name</i> : ] IF ( <i>scalar-logical-expr</i> ) THEN
22	R804 <i>else-if-stmt</i>	<b>is</b>	ELSE IF ( <i>scalar-logical-expr</i> ) THEN [ <i>if-construct-name</i> ]
23	R805 <i>else-stmt</i>	<b>is</b>	ELSE [ <i>if-construct-name</i> ]
24	R806 <i>end-if-stmt</i>	<b>is</b>	END IF [ <i>if-construct-name</i> ]

25 C801 (R802) If the *if-then-stmt* of an *if-construct* specifies an *if-construct-name*, the corresponding  
26 *end-if-stmt* shall specify the same *if-construct-name*. If the *if-then-stmt* of an *if-construct* does  
27 not specify an *if-construct-name*, the corresponding *end-if-stmt* shall not specify an *if-construct-*  
28 *name*. If an *else-if-stmt* or *else-stmt* specifies an *if-construct-name*, the corresponding *if-then-*  
29 *stmt* shall specify the same *if-construct-name*.

### 30 8.1.2.2 Execution of an IF construct

31 At most one of the blocks in the IF construct is executed. If there is an ELSE statement in the construct,  
32 exactly one of the blocks in the construct is executed. The scalar logical expressions are evaluated in  
33 the order of their appearance in the construct until a true value is found or an ELSE statement or END  
34 IF statement is encountered. If a true value or an ELSE statement is found, the block immediately  
35 following is executed and this completes the execution of the construct. The scalar logical expressions  
36 in any remaining ELSE IF statements of the IF construct are not evaluated. If none of the evaluated  
37 expressions is true and there is no ELSE statement, the execution of the construct is completed without  
38 the execution of any block within the construct.

39 An ELSE IF statement or an ELSE statement shall not be a branch target statement. It is permissible

- 1 to branch to an END IF statement only from within its IF construct. Execution of an END IF statement  
 2 has no effect.

### 3 8.1.2.3 Examples of IF constructs

#### NOTE 8.4

```

IF (CVAR == 'RESET') THEN
  I = 0; J = 0; K = 0
END IF
PROOF_DONE: IF (PROP) THEN
  WRITE (3, '(''QED''')
  STOP
ELSE
  PROP = NEXTPROP
END IF PROOF_DONE
IF (A > 0) THEN
  B = C/A
  IF (B > 0) THEN
    D = 1.0
  END IF
ELSE IF (C > 0) THEN
  B = A/C
  D = -1.0
ELSE
  B = ABS (MAX (A, C))
  D = 0
END IF

```

### 4 8.1.2.4 IF statement

- 5 The IF statement controls a single action statement (R214).

6 R807 *if-stmt* **is** IF ( *scalar-logical-expr* ) *action-stmt*

- 7 C802 (R807) The *action-stmt* in the *if-stmt* shall not be an *if-stmt*, *end-program-stmt*, *end-function-*  
 8 *stmt*, or *end-subroutine-stmt*.

- 9 Execution of an IF statement causes evaluation of the scalar logical expression. If the value of the  
 10 expression is true, the action statement is executed. If the value is false, the action statement is not  
 11 executed and execution continues.

- 12 The execution of a function reference in the scalar logical expression may affect entities in the action  
 13 statement.

#### NOTE 8.5

An example of an IF statement is:

```

IF (A > 0.0) A = LOG (A)

```

### 1 8.1.3 CASE construct

2 The **CASE construct** selects for execution at most one of its constituent blocks. The selection is  
3 based on the value of an expression.

#### 4 8.1.3.1 Form of the CASE construct

5 R808 *case-construct*            **is** *select-case-stmt*  
6    [ *case-stmt*  
7    *block* ] ...  
8    *end-select-stmt*  
9 R809 *select-case-stmt*           **is** [ *case-construct-name* : ] SELECT CASE ( *case-expr* )  
10 R810 *case-stmt*                   **is** CASE *case-selector* [*case-construct-name*]  
11 R811 *end-select-stmt*           **is** END SELECT [ *case-construct-name* ]

12 C803 (R808) If the *select-case-stmt* of a *case-construct* specifies a *case-construct-name*, the corre-  
13 sponding *end-select-stmt* shall specify the same *case-construct-name*. If the *select-case-stmt*  
14 of a *case-construct* does not specify a *case-construct-name*, the corresponding *end-select-stmt*  
15 shall not specify a *case-construct-name*. If a *case-stmt* specifies a *case-construct-name*, the  
16 corresponding *select-case-stmt* shall specify the same *case-construct-name*.

17 R812 *case-expr*                   **is** *scalar-int-expr*  
18   **or** *scalar-char-expr*  
19   **or** *scalar-logical-expr*  
20 R813 *case-selector*             **is** ( *case-value-range-list* )  
21   **or** DEFAULT

22 C804 (R808) No more than one of the selectors of one of the CASE statements shall be DEFAULT.

23 R814 *case-value-range*           **is** *case-value*  
24   **or** *case-value* :  
25   **or** : *case-value*  
26   **or** *case-value* : *case-value*  
27 R815 *case-value*                 **is** *scalar-int-initialization-expr*  
28   **or** *scalar-char-initialization-expr*  
29   **or** *scalar-logical-initialization-expr*

30 C805 (R808) For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*. For  
31 character type, the kind type parameters shall be the same; character length differences are  
32 allowed.

33 C806 (R808) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.

34 C807 (R808) For a given *case-construct*, the *case-value-ranges* shall not overlap; that is, there shall  
35 be no possible value of the *case-expr* that matches more than one *case-value-range*.

#### 36 8.1.3.2 Execution of a CASE construct

37 The execution of the SELECT CASE statement causes the case expression to be evaluated. The resulting  
38 value is called the **case index**. For a case value range list, a match occurs if the case index matches any  
39 of the case value ranges in the list. For a case index with a value of *c*, a match is determined as follows:

- 40 (1) If the case value range contains a single value *v* without a colon, a match occurs for type  
41 logical if the expression *c* .EQV. *v* is true, and a match occurs for type integer or character  
42 if the expression *c* == *v* is true.  
43 (2) If the case value range is of the form *low* : *high*, a match occurs if the expression *low* <= *c*  
44 .AND. *c* <= *high* is true.

- 1           (3) If the case value range is of the form *low* :, a match occurs if the expression *low* <= *c* is true.  
 2           (4) If the case value range is of the form : *high*, a match occurs if the expression *c* <= *high* is  
 3           true.  
 4           (5) If no other selector matches and a DEFAULT selector appears, it matches the case index.  
 5           (6) If no other selector matches and the DEFAULT selector does not appear, there is no match.
- 6 The block following the CASE statement containing the matching selector, if any, is executed. This  
 7 completes execution of the construct.
- 8 At most one of the blocks of a CASE construct is executed.
- 9 A CASE statement shall not be a branch target statement. It is permissible to branch to an *end-select-*  
 10 *stmt* only from within its CASE construct.

### 11 8.1.3.3 Examples of CASE constructs

#### NOTE 8.6

An integer signum function:

```

INTEGER FUNCTION SIGNUM (N)
SELECT CASE (N)
CASE (:-1)
    SIGNUM = -1
CASE (0)
    SIGNUM = 0
CASE (1:)
    SIGNUM = 1
END SELECT
END
  
```

#### NOTE 8.7

A code fragment to check for balanced parentheses:

```

CHARACTER (80) :: LINE
...
LEVEL = 0
SCAN_LINE: DO I = 1, 80
    CHECK_PARENS: SELECT CASE (LINE (I:I))
        CASE ('(')
            LEVEL = LEVEL + 1
        CASE (')')
            LEVEL = LEVEL - 1
            IF (LEVEL < 0) THEN
                PRINT *, 'UNEXPECTED RIGHT PARENTHESIS'
                EXIT SCAN_LINE
            END IF
        CASE DEFAULT
            ! Ignore all other characters
        END SELECT CHECK_PARENS
    END DO SCAN_LINE
    IF (LEVEL > 0) THEN
        PRINT *, 'MISSING RIGHT PARENTHESIS'
    END IF
  
```

**NOTE 8.8**

The following three fragments are equivalent:

```

IF (SILLY == 1) THEN
  CALL THIS
ELSE
  CALL THAT
END IF
SELECT CASE (SILLY == 1)
CASE (.TRUE.)
  CALL THIS
CASE (.FALSE.)
  CALL THAT
END SELECT
SELECT CASE (SILLY)
CASE DEFAULT
  CALL THAT
CASE (1)
  CALL THIS
END SELECT

```

**NOTE 8.9**

A code fragment showing several selections of one block:

```

SELECT CASE (N)
CASE (1, 3:5, 8) ! Selects 1, 3, 4, 5, 8
  CALL SUB
CASE DEFAULT
  CALL OTHER
END SELECT

```

**1 8.1.4 ASSOCIATE construct**

2 The ASSOCIATE construct associates named entities with expressions or variables during the execution  
3 of its block. These named construct entities (16.3) are associating entities (16.4.1.5). The names are  
4 **associate names**.

**5 8.1.4.1 Form of the ASSOCIATE construct**

- |    |      |   |           |  |
|----|------|---|-----------|--|
| 6  | R816 | <i>associate-construct</i>  | <b>is</b> | <i>associate-stmt</i><br><i>block</i><br><i>end-associate-stmt</i>                 |
| 7  |      |   |           |  |
| 8  |      |   |           |  |
| 9  | R817 | <i>associate-stmt</i>   | <b>is</b> | [ <i>associate-construct-name</i> : ] ASSOCIATE ■<br>■ ( <i>association-list</i> ) |
| 10 |      |   |           |  |
| 11 | R818 | <i>association</i>  | <b>is</b> | <i>associate-name</i> => <i>selector</i>   |
| 12 | R819 | <i>selector</i>   | <b>is</b> | <i>expr</i><br><b>or</b> <i>variable</i>   |
| 13 |      |   |           |  |
| 14 | C808 | (R818) If <i>selector</i> is not a <i>variable</i> or is a <i>variable</i> that has a vector subscript, <i>associate-name</i> |           |  |
| 15 |      | shall not appear in a variable definition context (16.5.7).   |           |  |
| 16 | C809 | (R818) An <i>associate-name</i> shall not be the same as another <i>associate-name</i> in the same <i>associate-</i>          |           |  |
| 17 |      | <i>stmt</i> .   |           |  |

1 R820 *end-associate-stmt* is END ASSOCIATE [ *associate-construct-name* ]

2 C810 (R820) If the *associate-stmt* of an *associate-construct* specifies an *associate-construct-name*,  
 3 the corresponding *end-associate-stmt* shall specify the same *associate-construct-name*. If the  
 4 *associate-stmt* of an *associate-construct* does not specify an *associate-construct-name*, the cor-  
 5 responding *end-associate-stmt* shall not specify an *associate-construct-name*.

#### 6 8.1.4.2 Execution of the ASSOCIATE construct

7 Execution of an ASSOCIATE construct causes execution of its *associate-stmt* followed by execution  
 8 of its block. During execution of that block each associate name identifies an entity, which is associ-  
 9 ated (16.4.1.5) with the corresponding selector. The associating entity assumes the declared type and  
 10 type parameters of the selector. If and only if the selector is polymorphic, the associating entity is  
 11 polymorphic.

12 The other attributes of the associating entity are described in 8.1.4.3.

13 It is permissible to branch to an *end-associate-stmt* only from within its ASSOCIATE construct.

#### 14 8.1.4.3 Attributes of associate names

15 Within a SELECT TYPE or ASSOCIATE construct, each associating entity has the same rank as its  
 16 associated selector. The lower bound of each dimension is the result of the intrinsic function LBOUND  
 17 (13.7.60) applied to the corresponding dimension of *selector*. The upper bound of each dimension is one  
 18 less than the sum of the lower bound and the extent. The associating entity has the ASYNCHRONOUS,  
 19 INTENT, TARGET, or VOLATILE attribute if and only if the selector is a variable and has the attribute.  
 20 If the associating entity is polymorphic, it assumes the dynamic type and type parameter values of the  
 21 selector. If the selector has the OPTIONAL attribute, it shall be present.

22 If the selector (8.1.4.1) is not permitted to appear in a variable definition context (16.5.7) or is an array  
 23 with a vector subscript, the associate name shall not appear in a variable definition context.

#### 24 8.1.4.4 Examples of the ASSOCIATE construct

##### NOTE 8.10

The following example illustrates an association with an expression.

```
ASSOCIATE ( Z => EXP(-(X**2+Y**2)) * COS(THETA) )
  PRINT *, A+Z, A-Z
END ASSOCIATE
```

The following example illustrates an association with a derived-type variable.

```
ASSOCIATE ( XC => AX%B(I,J)%C )
  XC%DV = XC%DV + PRODUCT(XC%EV(1:N))
END ASSOCIATE
```

The following example illustrates association with an array section.

```
ASSOCIATE ( ARRAY => AX%B(I,:)%C )
  ARRAY(N)%EV = ARRAY(N-1)%EV
END ASSOCIATE
```

The following example illustrates multiple associations.

## NOTE 8.10 (cont.)

```
ASSOCIATE ( W => RESULT(I,J)%W, ZX => AX%B(I,J)%D, ZY => AY%B(I,J)%D )
  W = ZX*X + ZY*Y
END ASSOCIATE
```

1 **8.1.5 SELECT TYPE construct**

2 The SELECT TYPE construct selects for execution at most one of its constituent blocks. The selection  
3 is based on the dynamic type of an expression. A name is associated with the expression (16.3, 16.4.1.5),  
4 in the same way as for the ASSOCIATE construct.

5 **8.1.5.1 Form of the SELECT TYPE construct**

6 R821 *select-type-construct* is *select-type-stmt*  
7 [ *type-guard-stmt*  
8 *block* ] ...  
9 *end-select-type-stmt*  
10 R822 *select-type-stmt* is [ *select-construct-name* : ] SELECT TYPE ■  
11 ■ ( [ *associate-name* => ] *selector* )

12 C811 (R822) If *selector* is not a named *variable*, *associate-name* => shall appear.

13 C812 (R822) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name*  
14 shall not appear in a variable definition context (16.5.7).

15 C813 (R822) The *selector* in a *select-type-stmt* shall be polymorphic.

16 R823 *type-guard-stmt* is TYPE IS ( *type-spec* ) [ *select-construct-name* ]  
17 or CLASS IS ( *type-spec* ) [ *select-construct-name* ]  
18 or CLASS DEFAULT [ *select-construct-name* ]

19 C814 (R823) The *type-spec* shall specify that each length type parameter is assumed.

20 C815 (R823) The *type-spec* shall not specify a sequence derived type or a type with the BIND attribute.

21 C816 (R823) If *selector* is not unlimited polymorphic, the *type-spec* shall specify an extension of the  
22 declared type of *selector*.

23 C817 (R823) For a given *select-type-construct*, the same type and kind type parameter values shall  
24 not be specified in more than one TYPE IS *type-guard-stmt* and shall not be specified in more  
25 than one CLASS IS *type-guard-stmt*.

26 C818 (R823) For a given *select-type-construct*, there shall be at most one CLASS DEFAULT *type-*  
27 *guard-stmt*.

28 R824 *end-select-type-stmt* is END SELECT [ *select-construct-name* ]

29 C819 (R821) If the *select-type-stmt* of a *select-type-construct* specifies a *select-construct-name*, the  
30 corresponding *end-select-type-stmt* shall specify the same *select-construct-name*. If the *select-*  
31 *type-stmt* of a *select-type-construct* does not specify a *select-construct-name*, the corresponding  
32 *end-select-type-stmt* shall not specify a *select-construct-name*. If a *type-guard-stmt* specifies a  
33 *select-construct-name*, the corresponding *select-type-stmt* shall specify the same *select-construct-*  
34 *name*.

35 The associate name of a SELECT TYPE construct is the *associate-name* if specified; otherwise it is the  
36 *name* that constitutes the *selector*.

### 1 8.1.5.2 Execution of the SELECT TYPE construct

2 Execution of a SELECT TYPE construct whose selector is not a *variable* causes the selector expression  
3 to be evaluated.

4 A SELECT TYPE construct selects at most one block to be executed. During execution of that block,  
5 the associate name identifies an entity, which is associated (16.4.1.5) with the selector.

6 A TYPE IS type guard statement matches the selector if the dynamic type and type parameter values of  
7 the selector are the same as those specified by the statement. A CLASS IS type guard statement matches  
8 the selector if the dynamic type of the selector is an extension of the type specified by the statement  
9 and the kind type parameter values specified by the statement are the same as the corresponding type  
10 parameter values of the dynamic type of the selector.

11 The block to be executed is selected as follows:

- 12 (1) If a TYPE IS type guard statement matches the selector, the block following that statement  
13 is executed.
- 14 (2) Otherwise, if exactly one CLASS IS type guard statement matches the selector, the block  
15 following that statement is executed.
- 16 (3) Otherwise, if several CLASS IS type guard statements match the selector, one of these  
17 statements must specify a type that is an extension of all the types specified in the others;  
18 the block following that statement is executed.
- 19 (4) Otherwise, if there is a CLASS DEFAULT type guard statement, the block following that  
20 statement is executed.

#### NOTE 8.11

This algorithm does not examine the type guard statements in source text order when it looks for a match; it selects the most particular type guard when there are several potential matches.

21 Within the block following a TYPE IS type guard statement, the associating entity (16.4.5) is not  
22 polymorphic (5.1.1.2), has the type named in the type guard statement, and has the type parameter  
23 values of the selector.

24 Within the block following a CLASS IS type guard statement, the associating entity is polymorphic and  
25 has the declared type named in the type guard statement. The type parameter values of the associating  
26 entity are the corresponding type parameter values of the selector.

27 Within the block following a CLASS DEFAULT type guard statement, the associating entity is poly-  
28 morphic and has the same declared type as the selector. The type parameter values of the associating  
29 entity are those of the declared type of the selector.

#### NOTE 8.12

If the declared type of the *selector* is T, specifying CLASS DEFAULT has the same effect as specifying CLASS IS (T).

30 The other attributes of the associating entity are described in 8.1.4.3.

31 A type guard statement shall not be a branch target statement. It is permissible to branch to an  
32 *end-select-type-stmt* only from within its SELECT TYPE construct.

### 33 8.1.5.3 Examples of the SELECT TYPE construct

**NOTE 8.13**

```

TYPE POINT
  REAL :: X, Y
END TYPE POINT
TYPE, EXTENDS(POINT) :: POINT_3D
  REAL :: Z
END TYPE POINT_3D
TYPE, EXTENDS(POINT) :: COLOR_POINT
  INTEGER :: COLOR
END TYPE COLOR_POINT

TYPE(POINT), TARGET :: P
TYPE(POINT_3D), TARGET :: P3
TYPE(COLOR_POINT), TARGET :: C
CLASS(POINT), POINTER :: P_OR_C
P_OR_C => C
SELECT TYPE ( A => P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: A" implied here
  PRINT *, A%X, A%Y ! This block gets executed
TYPE IS ( POINT_3D )
  ! "TYPE ( POINT_3D ) :: A" implied here
  PRINT *, A%X, A%Y, A%Z
END SELECT

```

**NOTE 8.14**

The following example illustrates the omission of *associate-name*. It uses the declarations from Note 8.13.

```

P_OR_C => P3
SELECT TYPE ( P_OR_C )
CLASS IS ( POINT )
  ! "CLASS ( POINT ) :: P_OR_C" implied here
  PRINT *, P_OR_C%X, P_OR_C%Y
TYPE IS ( POINT_3D )
  ! "TYPE ( POINT_3D ) :: P_OR_C" implied here
  PRINT *, P_OR_C%X, P_OR_C%Y, P_OR_C%Z ! This block gets executed
END SELECT

```

**1 8.1.6 DO construct**

2 The DO construct specifies the repeated execution of a sequence of executable constructs. Such a  
3 repeated sequence is called a **loop**. The EXIT and CYCLE statements may be used to modify the  
4 execution of a loop.

5 The number of iterations of a loop may be determined at the beginning of execution of the DO construct,  
6 or may be left indefinite ("DO forever" or DO WHILE). In either case, an EXIT statement (8.1.6.4.4)  
7 anywhere in the DO construct may be executed to terminate the loop immediately. The current iteration  
8 of the loop may be curtailed by executing a CYCLE statement (8.1.6.4.3).

### 1 8.1.6.1 Forms of the DO construct

2 The DO construct may be written in either a block form or a nonblock form.

3 R825 *do-construct* is *block-do-construct*  
 4 or *nonblock-do-construct*

#### 5 8.1.6.1.1 Form of the block DO construct

6 R826 *block-do-construct* is *do-stmt*  
 7 *do-block*  
 8 *end-do*  
 9 R827 *do-stmt* is *label-do-stmt*  
 10 or *nonlabel-do-stmt*  
 11 R828 *label-do-stmt* is [ *do-construct-name* : ] DO *label* [ *loop-control* ]  
 12 R829 *nonlabel-do-stmt* is [ *do-construct-name* : ] DO [ *loop-control* ]  
 13 R830 *loop-control* is [ , ] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■  
 14 ■ [ , *scalar-int-expr* ]  
 15 or [ , ] WHILE ( *scalar-logical-expr* )  
 16 R831 *do-variable* is *scalar-int-variable*

17 C820 (R831) The *do-variable* shall be a named scalar variable of type integer.

18 R832 *do-block* is *block*  
 19 R833 *end-do* is *end-do-stmt*  
 20 or *continue-stmt*  
 21 R834 *end-do-stmt* is END DO [ *do-construct-name* ]

22 C821 (R826) If the *do-stmt* of a *block-do-construct* specifies a *do-construct-name*, the corresponding  
 23 *end-do* shall be an *end-do-stmt* specifying the same *do-construct-name*. If the *do-stmt* of a  
 24 *block-do-construct* does not specify a *do-construct-name*, the corresponding *end-do* shall not  
 25 specify a *do-construct-name*.

26 C822 (R826) If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.

27 C823 (R826) If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the  
 28 same *label*.

#### 29 8.1.6.1.2 Form of the nonblock DO construct

30 R835 *nonblock-do-construct* is *action-term-do-construct*  
 31 or *outer-shared-do-construct*  
 32 R836 *action-term-do-construct* is *label-do-stmt*  
 33 *do-body*  
 34 *do-term-action-stmt*  
 35 R837 *do-body* is [ *execution-part-construct* ] ...  
 36 R838 *do-term-action-stmt* is *action-stmt*

37 C824 (R838) A *do-term-action-stmt* shall not be a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*,  
 38 a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.

39 C825 (R835) The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer  
 40 to the same label.

41 R839 *outer-shared-do-construct* is *label-do-stmt*  
 42 *do-body*  
 43 *shared-term-do-construct*  
 44 R840 *shared-term-do-construct* is *outer-shared-do-construct*  
 45 or *inner-shared-do-construct*  
 46 R841 *inner-shared-do-construct* is *label-do-stmt*

1 *do-body*  
 2 *do-term-shared-stmt*  
 3 R842 *do-term-shared-stmt* is action-stmt  
 4 C826 (R842) A *do-term-shared-stmt* shall not be a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*,  
 5 an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.  
 6 C827 (R840) The *do-term-shared-stmt* shall be identified with a label and all of the *label-do-stmts* of the *inner-shared-*  
 7 *do-construct* and *outer-shared-do-construct* shall refer to the same label.  
 8 The *do-term-action-stmt*, *do-term-shared-stmt*, or *shared-term-do-construct* following the *do-body* of a nonblock DO con-  
 9 struct is called the **DO termination** of that construct.

10 Within a scoping unit, all DO constructs whose DO statements refer to the same label are nonblock DO constructs, and  
 11 are said to share the statement identified by that label.

## 12 8.1.6.2 Range of the DO construct

13 The **range** of a block DO construct is the *do-block*, which shall satisfy the rules for blocks (8.1.1). In  
 14 particular, transfer of control to the interior of such a block from outside the block is prohibited. It  
 15 is permitted to branch to the *end-do* of a block DO construct only from within the range of that DO  
 16 construct.

17 The range of a nonblock DO construct consists of the *do-body* and the following DO termination. The end of such a  
 18 range is not bounded by a particular statement as for the other executable constructs (e.g., END IF); nevertheless, the  
 19 range satisfies the rules for blocks (8.1.1). Transfer of control into the *do-body* or to the DO termination from outside the  
 20 range is prohibited; in particular, it is permitted to branch to a *do-term-shared-stmt* only from within the range of the  
 21 corresponding *inner-shared-do-construct*.

## 22 8.1.6.3 Active and inactive DO constructs

23 A DO construct is either **active** or **inactive**. Initially inactive, a DO construct becomes active only  
 24 when its DO statement is executed.

25 Once active, the DO construct becomes inactive only when it terminates (8.1.6.4.4).

## 26 8.1.6.4 Execution of a DO construct

27 A DO construct specifies a loop, that is, a sequence of executable constructs that is executed repeatedly.  
 28 There are three phases in the execution of a DO construct: initiation of the loop, execution of the loop  
 29 range, and termination of the loop.

### 30 8.1.6.4.1 Loop initiation

31 When the DO statement is executed, the DO construct becomes active. If *loop-control* is

32  $[ , ] \textit{do-variable} = \textit{scalar-int-expr}_1 , \textit{scalar-int-expr}_2 [ , \textit{scalar-int-expr}_3 ]$

33 the following steps are performed in sequence:

- 34 (1) The initial parameter  $m_1$ , the terminal parameter  $m_2$ , and the incrementation parameter  $m_3$   
 35 are of type integer with the same kind type parameter as the *do-variable*. Their values are es-  
 36 tablished by evaluating *scalar-int-expr*<sub>1</sub>, *scalar-int-expr*<sub>2</sub>, and *scalar-int-expr*<sub>3</sub>, respectively,  
 37 including, if necessary, conversion to the kind type parameter of the *do-variable* according  
 38 to the rules for numeric conversion (Table 7.9). If *scalar-int-expr*<sub>3</sub> does not appear,  $m_3$  has  
 39 the value 1. The value of  $m_3$  shall not be zero.
- 40 (2) The DO variable becomes defined with the value of the initial parameter  $m_1$ .

- 1 (3) The **iteration count** is established and is the value of the expression  $(m_2 - m_1 + m_3)/m_3$ ,  
 2 unless that value is negative, in which case the iteration count is 0.

**NOTE 8.15**

The iteration count is zero whenever:

$$m_1 > m_2 \text{ and } m_3 > 0, \text{ or}$$

$$m_1 < m_2 \text{ and } m_3 < 0.$$

3 If *loop-control* is omitted, no iteration count is calculated. The effect is as if a large positive iteration  
 4 count, impossible to decrement to zero, were established. If *loop-control* is [ , ] WHILE (*scalar-logical-*  
 5 *expr*), the effect is as if *loop-control* were omitted and the following statement inserted as the first  
 6 statement of the *do-block*:

7 IF (.NOT. (*scalar-logical-expr*)) EXIT

8 At the completion of the execution of the DO statement, the execution cycle begins.

**9 8.1.6.4.2 The execution cycle**

10 The **execution cycle** of a DO construct consists of the following steps performed in sequence repeatedly  
 11 until termination:

- 12 (1) The iteration count, if any, is tested. If it is zero, the loop terminates and the DO construct  
 13 becomes inactive. If *loop-control* is [ , ] WHILE (*scalar-logical-expr*), the *scalar-logical-expr*  
 14 is evaluated; if the value of this expression is false, the loop terminates and the DO construct  
 15 becomes inactive. If, as a result, all of the DO constructs sharing the *do-term-shared-stmt* are inactive,  
 16 the execution of all of these constructs is complete. However, if some of the DO constructs sharing the  
 17 *do-term-shared-stmt* are active, execution continues with step (3) of the execution cycle of the active DO  
 18 construct whose DO statement was most recently executed.
- 19 (2) If the iteration count is nonzero, the range of the loop is executed.
- 20 (3) The iteration count, if any, is decremented by one. The DO variable, if any, is incremented  
 21 by the value of the incrementation parameter  $m_3$ .

22 Except for the incrementation of the DO variable that occurs in step (3), the DO variable shall neither  
 23 be redefined nor become undefined while the DO construct is active.

**24 8.1.6.4.3 CYCLE statement**

25 Step (2) in the above execution cycle may be curtailed by executing a CYCLE statement from within  
 26 the range of the loop.

27 R843 *cycle-stmt* is CYCLE [ *do-construct-name* ]

28 C828 (R843) If a *cycle-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-*  
 29 *construct*; otherwise, it shall be within the range of at least one *do-construct*.

30 A CYCLE statement belongs to a particular DO construct. If the CYCLE statement refers to a DO  
 31 construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct  
 32 in which it appears.

33 Execution of a CYCLE statement causes immediate progression to step (3) of the current execution cycle  
 34 of the DO construct to which it belongs. If this construct is a nonblock DO construct, the *do-term-action-stmt* or  
 35 *do-term-shared-stmt* is not executed.

36 In a block DO construct, a transfer of control to the *end-do* has the same effect as execution of a CYCLE  
 37 statement belonging to that construct. In a nonblock DO construct, transfer of control to the *do-term-action-stmt*

1 or *do-term-shared-stmt* causes that statement or construct itself to be executed. Unless a further transfer of control results,  
 2 step (3) of the current execution cycle of the DO construct is then executed.

#### 3 8.1.6.4.4 Loop termination

4 The EXIT statement provides one way of terminating a loop.

5 R844 *exit-stmt* is EXIT [ *do-construct-name* ]

6 C829 (R844) If an *exit-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-*  
 7 *construct*; otherwise, it shall be within the range of at least one *do-construct*.

8 An EXIT statement belongs to a particular DO construct. If the EXIT statement refers to a DO  
 9 construct name, it belongs to that DO construct; otherwise, it belongs to the innermost DO construct  
 10 in which it appears.

11 The loop terminates, and the DO construct becomes inactive, when any of the following occurs:

- 12 (1) Determination that the iteration count is zero or the *scalar-logical-expr* is false, when tested  
 13 during step (1) of the above execution cycle
- 14 (2) Execution of an EXIT statement belonging to the DO construct
- 15 (3) Execution of an EXIT statement or a CYCLE statement that is within the range of the DO  
 16 construct, but that belongs to an outer DO construct
- 17 (4) Transfer of control from a statement within the range of a DO construct to a statement that  
 18 is neither the *end-do* nor within the range of the same DO construct
- 19 (5) Execution of a RETURN statement within the range of the DO construct
- 20 (6) Execution of a STOP statement anywhere in the program; or termination of the program  
 21 for any other reason.

22 When a DO construct becomes inactive, the DO variable, if any, of the DO construct retains its last  
 23 defined value.

#### 24 8.1.6.5 Examples of DO constructs

##### NOTE 8.16

The following program fragment computes a tensor product of two arrays:

```
DO I = 1, M
  DO J = 1, N
    C (I, J) = SUM (A (I, J, :) * B (:, I, J))
  END DO
END DO
```

##### NOTE 8.17

The following program fragment contains a DO construct that uses the WHILE form of *loop-control*. The loop will continue to execute until an end-of-file or input/output error is encountered, at which point the DO statement terminates the loop. When a negative value of X is read, the program skips immediately to the next READ statement, bypassing most of the range of the loop.

```
READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
DO WHILE (IOS == 0)
  IF (X >= 0.) THEN
    CALL SUBA (X)
```

**NOTE 8.17 (cont.)**

```

CALL SUBB (X)
...
CALL SUBZ (X)
ENDIF
READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
END DO

```

**NOTE 8.18**

The following example behaves exactly the same as the one in Note 8.17. However, the READ statement has been moved to the interior of the range, so that only one READ statement is needed. Also, a CYCLE statement has been used to avoid an extra level of IF nesting.

```

DO      ! A "DO WHILE + 1/2" loop
READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
IF (IOS /= 0) EXIT
IF (X < 0.) CYCLE
CALL SUBA (X)
CALL SUBB (X)
...
CALL SUBZ (X)
END DO

```

**NOTE 8.19**

Additional examples of DO constructs are in C.5.3.

## 1 8.2 Branching

2 **Branching** is used to alter the normal execution sequence. A branch causes a transfer of control from  
3 one statement in a scoping unit to a labeled branch target statement in the same scoping unit. Branching  
4 may be caused by a GOTO statement, a computed GOTO statement, an arithmetic IF statement, a  
5 CALL statement that has an *alt-return-spec*, or an input/output statement that has an END= or ERR=  
6 specifier. Although procedure references and control constructs can cause transfer of control, they are  
7 not branches. A **branch target statement** is an *action-stmt*, an *associate-stmt*, an *end-associate-stmt*,  
8 an *if-then-stmt*, an *end-if-stmt*, a *select-case-stmt*, an *end-select-stmt*, a *select-type-stmt*, an *end-select-*  
9 *type-stmt*, a *do-stmt*, an *end-do-stmt*, a *forall-construct-stmt*, a *do-term-action-stmt*, a *do-term-shared-stmt*, or  
10 a *where-construct-stmt*.

### 11 8.2.1 GO TO statement

12 R845 *goto-stmt* is GO TO *label*

13 C830 (R845) The *label* shall be the statement label of a branch target statement that appears in the  
14 same scoping unit as the *goto-stmt*.

15 Execution of a GO TO statement causes a transfer of control so that the branch target statement  
16 identified by the label is executed next.

## 1 8.2.2 Computed GO TO statement

2 R846 *computed-goto-stmt* is GO TO ( *label-list* ) [ , ] *scalar-int-expr*

3 C831 (R846) Each *label* in *label-list* shall be the statement label of a branch target statement that appears in the same  
4 scoping unit as the *computed-goto-stmt*.

### NOTE 8.20

The same statement label may appear more than once in a label list.

5 Execution of a computed GO TO statement causes evaluation of the scalar integer expression. If this value is  $i$  such that  
6  $1 \leq i \leq n$  where  $n$  is the number of labels in *label-list*, a transfer of control occurs so that the next statement executed is  
7 the one identified by the  $i$ th label in the list of labels. If  $i$  is less than 1 or greater than  $n$ , the execution sequence continues  
8 as though a CONTINUE statement were executed.

## 9 8.2.3 Arithmetic IF statement

10 R847 *arithmetic-if-stmt* is IF ( *scalar-numeric-expr* ) *label* , *label* , *label*

11 C832 (R847) Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the  
12 *arithmetic-if-stmt*.

13 C833 (R847) The *scalar-numeric-expr* shall not be of type complex.

### NOTE 8.21

The same label may appear more than once in one arithmetic IF statement.

14 Execution of an arithmetic IF statement causes evaluation of the numeric expression followed by a transfer of control. The  
15 branch target statement identified by the first label, the second label, or the third label is executed next depending on  
16 whether the value of the numeric expression is less than zero, equal to zero, or greater than zero, respectively.

## 17 8.3 CONTINUE statement

18 Execution of a CONTINUE statement has no effect.

19 R848 *continue-stmt* is CONTINUE

## 20 8.4 STOP statement

21 R849 *stop-stmt* is STOP [ *stop-code* ]

22 R850 *stop-code* is *scalar-char-constant*

23 or *digit* [ *digit* [ *digit* [ *digit* ] ] ] ]

24 C834 (R850) *scalar-char-constant* shall be of type default character.

25 Execution of a STOP statement causes normal termination (2.3.4) of execution of the program. At the  
26 time of termination, the stop code, if any, is available in a processor-dependent manner. Leading zero  
27 digits in the stop code are not significant. If any exception (14) is signaling, the processor shall issue  
28 a warning indicating which exceptions are signaling; this warning shall be on the unit identified by the  
29 named constant ERROR\_UNIT from the ISO\_FORTRAN\_ENV intrinsic module (13.8.2.2).

## 1 Section 9: Input/output statements

2 **Input statements** provide the means of transferring data from external media to internal storage or  
3 from an internal file to internal storage. This process is called **reading**. **Output statements** provide  
4 the means of transferring data from internal storage to external media or from internal storage to an  
5 internal file. This process is called **writing**. Some input/output statements specify that editing of the  
6 data is to be performed.

7 In addition to the statements that transfer data, there are auxiliary input/output statements to ma-  
8 nipulate the external medium, or to describe or inquire about the properties of the connection to the  
9 external medium.

10 The input/output statements are the OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, END-  
11 FILE, REWIND, FLUSH, WAIT, and INQUIRE statements.

12 The READ statement is a **data transfer input statement**. The WRITE statement and the PRINT  
13 statement are **data transfer output statements**. The OPEN statement and the CLOSE state-  
14 ment are **file connection statements**. The INQUIRE statement is a **file inquiry statement**. The  
15 BACKSPACE, ENDFILE, and REWIND statements are **file positioning statements**.

16 A file is composed of either a sequence of file storage units (9.2.4) or a sequence of records, which  
17 provide an extra level of organization to the file. A file composed of records is called a **record file**. A  
18 file composed of file storage units is called a **stream file**. A processor may allow a file to be viewed  
19 both as a record file and as a stream file; in this case the relationship between the file storage units when  
20 viewed as a stream file and the records when viewed as a record file is processor dependent.

21 A file is either an external file (9.2) or an internal file (9.3).

### 22 9.1 Records

23 A **record** is a sequence of values or a sequence of characters. For example, a line on a terminal is usually  
24 considered to be a record. However, a record does not necessarily correspond to a physical entity. There  
25 are three kinds of records:

- 26 (1) Formatted
- 27 (2) Unformatted
- 28 (3) Endfile

#### NOTE 9.1

What is called a “record” in Fortran is commonly called a “logical record”. There is no concept  
in Fortran of a “physical record.”

#### 29 9.1.1 Formatted record

30 A **formatted record** consists of a sequence of characters that are capable of representation in the  
31 processor; however, a processor may prohibit some control characters (3.1) from appearing in a formatted  
32 record. The length of a formatted record is measured in characters and depends primarily on the number  
33 of characters put into the record when it is written. However, it may depend on the processor and the  
34 external medium. The length may be zero. Formatted records may be read or written only by formatted  
35 input/output statements.

1 Formatted records may be prepared by means other than Fortran.

## 2 9.1.2 Unformatted record

3 An **unformatted record** consists of a sequence of values in a processor-dependent form and may contain  
4 data of any type or may contain no data. The length of an unformatted record is measured in file storage  
5 units (9.2.4) and depends on the output list (9.5.2) used when it is written, as well as on the processor  
6 and the external medium. The length may be zero. Unformatted records may be read or written only  
7 by unformatted input/output statements.

## 8 9.1.3 Endfile record

9 An **endfile record** is written explicitly by the ENDFILE statement; the file shall be connected for  
10 sequential access. An endfile record is written implicitly to a file connected for sequential access when  
11 the most recent data transfer statement referring to the file is a data transfer output statement, no  
12 intervening file positioning statement referring to the file has been executed, and

- 13 (1) A REWIND or BACKSPACE statement references the unit to which the file is connected  
14 or
- 15 (2) The unit is closed, either explicitly by a CLOSE statement, implicitly by a program termi-  
16 nation not caused by an error condition, or implicitly by another OPEN statement for the  
17 same unit.

18 An endfile record may occur only as the last record of a file. An endfile record does not have a length  
19 property.

### NOTE 9.2

An endfile record does not necessarily have any physical embodiment. The processor may use a record count or other means to register the position of the file at the time an ENDFILE statement is executed, so that it can take appropriate action when that position is reached again during a read operation. The endfile record, however it is implemented, is considered to exist for the BACKSPACE statement (9.7.1).

## 20 9.2 External files

21 An **external file** is any file that exists in a medium external to the program.

22 At any given time, there is a processor-dependent set of allowed **access methods**, a processor-dependent  
23 set of allowed **forms**, a processor-dependent set of allowed **actions**, and a processor-dependent set of  
24 allowed **record lengths** for a file.

### NOTE 9.3

For example, the processor-dependent set of allowed actions for a printer would likely include the write action, but not the read action.

25 A file may have a name; a file that has a name is called a **named file**. The name of a named file is  
26 represented by a character string value. The set of allowable names for a file is processor dependent.

27 An external file that is connected to a unit has a **position** property (9.2.3).

### NOTE 9.4

For more explanatory information on external files, see C.6.1.

### 1 9.2.1 File existence

- 2 At any given time, there is a processor-dependent set of external files that are said to **exist** for a program.  
 3 A file may be known to the processor, yet not exist for a program at a particular time.

#### NOTE 9.5

Security reasons may prevent a file from existing for a program. A newly created file may exist but contain no records.

- 4 To create a file means to cause a file to exist that did not exist previously. To delete a file means to  
 5 terminate the existence of the file.

- 6 All input/output statements may refer to files that exist. An INQUIRE, OPEN, CLOSE, WRITE,  
 7 PRINT, REWIND, FLUSH, or ENDFILE statement also may refer to a file that does not exist. Execu-  
 8 tion of a WRITE, PRINT, or ENDFILE statement referring to a preconnected file that does not exist  
 9 creates the file.

### 10 9.2.2 File access

- 11 There are three methods of accessing the data of an external file: sequential, direct, and stream. Some  
 12 files may have more than one allowed access method; other files may be restricted to one access method.

#### NOTE 9.6

For example, a processor may allow only sequential access to a file on magnetic tape. Thus, the set of allowed access methods depends on the file and the processor.

- 13 The method of accessing a file is determined when the file is connected to a unit (9.4.3) or when the file  
 14 is created if the file is preconnected (9.4.4).

#### 15 9.2.2.1 Sequential access

- 16 **Sequential access** is a method of accessing the records of an external record file in order.

- 17 When connected for sequential access, an external file has the following properties:

- 18 (1) The order of the records is the order in which they were written if the direct access method  
 19 is not a member of the set of allowed access methods for the file. If the direct access method  
 20 is also a member of the set of allowed access methods for the file, the order of the records  
 21 is the same as that specified for direct access. In this case, the first record accessible by  
 22 sequential access is the record whose record number is 1 for direct access. The second record  
 23 accessible by sequential access is the record whose record number is 2 for direct access, etc.  
 24 A record that has not been written since the file was created shall not be read.
- 25 (2) The records of the file are either all formatted or all unformatted, except that the last record  
 26 of the file may be an endfile record. Unless the previous reference to the file was a data  
 27 transfer output statement, the last record, if any, of the file shall be an endfile record.
- 28 (3) The records of the file shall be read or written only by sequential access input/output  
 29 statements.

#### 30 9.2.2.2 Direct access

- 31 **Direct access** is a method of accessing the records of an external record file in arbitrary order.

- 32 When connected for direct access, an external file has the following properties:

- 1 (1) Each record of the file is uniquely identified by a positive integer called the **record number**.  
2 The record number of a record is specified when the record is written. Once established,  
3 the record number of a record can never be changed. The order of the records is the order  
4 of their record numbers.

**NOTE 9.7**

A record cannot be deleted; however, a record may be rewritten.

- 5 (2) The records of the file are either all formatted or all unformatted. If the sequential access  
6 method is also a member of the set of allowed access methods for the file, its endfile record,  
7 if any, is not considered to be part of the file while it is connected for direct access. If the  
8 sequential access method is not a member of the set of allowed access methods for the file,  
9 the file shall not contain an endfile record.
- 10 (3) The records of the file shall be read or written only by direct access input/output statements.
- 11 (4) All records of the file have the same length.
- 12 (5) Records need not be read or written in the order of their record numbers. Any record may  
13 be written into the file while it is connected to a unit. For example, it is permissible to write  
14 record 3, even though records 1 and 2 have not been written. Any record may be read from  
15 the file while it is connected to a unit, provided that the record has been written since the  
16 file was created, and if a READ statement for this connection is permitted.
- 17 (6) The records of the file shall not be read or written using list-directed formatting (10.9),  
18 namelist formatting (10.10), or a nonadvancing input/output statement (9.2.3.1).

**19 9.2.2.3 Stream access**

20 **Stream access** is a method of accessing the file storage units (9.2.4) of an external stream file.

21 The properties of an external file connected for stream access depend on whether the connection is for  
22 unformatted or formatted access.

23 When connected for unformatted stream access, an external file has the following properties:

- 24 (1) The file storage units of the file shall be read or written only by stream access input/output  
25 statements.
- 26 (2) Each file storage unit in the file is uniquely identified by a positive integer called the position.  
27 The first file storage unit in the file is at position 1. The position of each subsequent file  
28 storage unit is one greater than that of its preceding file storage unit.
- 29 (3) If it is possible to position the file, the file storage units need not be read or written in  
30 order of their position. For example, it might be permissible to write the file storage unit  
31 at position 3, even though the file storage units at positions 1 and 2 have not been written.  
32 Any file storage unit may be read from the file while it is connected to a unit, provided that  
33 the file storage unit has been written since the file was created, and if a READ statement  
34 for this connection is permitted.

35 When connected for formatted stream access, an external file has the following properties:

- 36 (1) Some file storage units of the file may contain record markers; this imposes a record structure  
37 on the file in addition to its stream structure. There might or might not be a record marker  
38 at the end of the file. If there is no record marker at the end of the file, the final record is  
39 incomplete.
- 40 (2) No maximum length (9.4.5.12) is applicable to these records.
- 41 (3) Writing an empty record with no record marker has no effect.

**NOTE 9.8**

Because the record structure is determined from the record markers that are stored in the file itself, an incomplete record at the end of the file is necessarily not empty.

- 1           (4) The file storage units of the file shall be read or written only by formatted stream access  
2           input/output statements.
- 3           (5) Each file storage unit in the file is uniquely identified by a positive integer called the position.  
4           The first file storage unit in the file is at position 1. The relationship between positions of  
5           successive file storage units is processor dependent; not all positive integers need correspond  
6           to valid positions.
- 7           (6) If it is possible to position the file, the file position can be set to a position that was  
8           previously identified by the POS= specifier in an INQUIRE statement.

**NOTE 9.9**

There may be some character positions in the file that do not correspond to characters written; this is because on some processors a record marker may be written to the file as a carriage-return/line-feed or other sequence. The means of determining the position in a file connected for stream access is via the POS= specifier in an INQUIRE statement (9.9.1.21).

- 9           (7) A processor may prohibit some control characters (3.1) from appearing in a formatted stream  
10          file.

**11 9.2.3 File position**

12 Execution of certain input/output statements affects the position of an external file. Certain circum-  
13 stances can cause the position of a file to become indeterminate.

14 The **initial point** of a file is the position just before the first record or file storage unit. The **terminal**  
15 **point** is the position just after the last record or file storage unit. If there are no records or file storage  
16 units in the file, the initial point and the terminal point are the same position.

17 If a record file is positioned within a record, that record is the **current record**; otherwise, there is no  
18 current record.

19 Let  $n$  be the number of records in the file. If  $1 < i \leq n$  and a file is positioned within the  $i$ th record or  
20 between the  $(i - 1)$ th record and the  $i$ th record, the  $(i - 1)$ th record is the **preceding record**. If  $n \geq 1$   
21 and the file is positioned at its terminal point, the preceding record is the  $n$ th and last record. If  $n = 0$   
22 or if a file is positioned at its initial point or within the first record, there is no preceding record.

23 If  $1 \leq i < n$  and a file is positioned within the  $i$ th record or between the  $i$ th and  $(i + 1)$ th record, the  
24  $(i + 1)$ th record is the **next record**. If  $n \geq 1$  and the file is positioned at its initial point, the first record  
25 is the next record. If  $n = 0$  or if a file is positioned at its terminal point or within the  $n$ th (last) record,  
26 there is no next record.

27 For a file connected for stream access, the file position is either between two file storage units, at the  
28 initial point of the file, at the terminal point of the file, or undefined.

**29 9.2.3.1 Advancing and nonadvancing input/output**

30 An **advancing input/output statement** always positions a record file after the last record read or  
31 written, unless there is an error condition.

32 A **nonadvancing input/output statement** may position a record file at a character position within  
33 the current record, or a subsequent record (10.7.2). Using nonadvancing input/output, it is possible to  
34 read or write a record of the file by a sequence of input/output statements, each accessing a portion  
35 of the record. It is also possible to read variable-length records and be notified of their lengths. If a

1 nonadvancing output statement leaves a file positioned within a current record and no further output  
2 statement is executed for the file before it is closed or a BACKSPACE, ENDFILE, or REWIND statement  
3 is executed for it, the effect is as if the output statement were the corresponding advancing output  
4 statement.

### 5 **9.2.3.2 File position prior to data transfer**

6 The positioning of the file prior to data transfer depends on the method of access: sequential, direct, or  
7 stream.

8 For sequential access on input, if there is a current record, the file position is not changed. Otherwise,  
9 the file is positioned at the beginning of the next record and this record becomes the current record.  
10 Input shall not occur if there is no next record or if there is a current record and the last data transfer  
11 statement accessing the file performed output.

12 If the file contains an endfile record, the file shall not be positioned after the endfile record prior to data  
13 transfer. However, a REWIND or BACKSPACE statement may be used to reposition the file.

14 For sequential access on output, if there is a current record, the file position is not changed and the  
15 current record becomes the last record of the file. Otherwise, a new record is created as the next record  
16 of the file; this new record becomes the last and current record of the file and the file is positioned at  
17 the beginning of this record.

18 For direct access, the file is positioned at the beginning of the record specified by the REC= specifier.  
19 This record becomes the current record.

20 For stream access, the file is positioned immediately before the file storage unit specified by the POS=  
21 specifier; if there is no POS= specifier, the file position is not changed.

22 File positioning for child data transfer statements is described in [9.5.3.7](#).

### 23 **9.2.3.3 File position after data transfer**

24 If an error condition ([9.10](#)) occurred, the position of the file is indeterminate. If no error condition  
25 occurred, but an end-of-file condition ([9.10](#)) occurred as a result of reading an endfile record, the file is  
26 positioned after the endfile record.

27 For unformatted stream access, if no error condition occurred, the file position is not changed. For  
28 unformatted stream output, if the file position exceeds the previous terminal point of the file, the  
29 terminal point is set to the file position.

#### **NOTE 9.10**

An unformatted stream output statement with a POS= specifier and an empty output list can have the effect of extending the terminal point of a file without actually writing any data.

30 For a formatted stream output statement, if no error condition occurred, the terminal point of the file  
31 is set to the highest-numbered position to which data was transferred by the statement.

#### **NOTE 9.11**

The highest-numbered position might not be the current one if the output involved T or TL edit descriptors ([10.7.1.1](#)).

32 For formatted stream input, if an end-of-file condition occurred, the file position is not changed.

33 For nonadvancing input, if no error condition or end-of-file condition occurred, but an end-of-record

1 condition (9.10) occurred, the file is positioned after the record just read. If no error condition, end-of-  
2 file condition, or end-of-record condition occurred in a nonadvancing input statement, the file position  
3 is not changed. If no error condition occurred in a nonadvancing output statement, the file position is  
4 not changed.

5 In all other cases, the file is positioned after the record just read or written and that record becomes the  
6 preceding record.

#### 7 **9.2.4 File storage units**

8 A **file storage unit** is the basic unit of storage in a stream file or an unformatted record file. It is the  
9 unit of file position for stream access, the unit of record length for unformatted files, and the unit of file  
10 size for all external files.

11 Every value in a stream file or an unformatted record file shall occupy an integer number of file storage  
12 units; if the stream or record file is unformatted, this number shall be the same for all scalar values of  
13 the same type and type parameters. The number of file storage units required for an item of a given type  
14 and type parameters may be determined using the IOLENGTH= specifier of the INQUIRE statement  
15 (9.9.3).

16 For a file connected for unformatted stream access, the processor shall not have alignment restrictions  
17 that prevent a value of any type from being stored at any positive integer file position.

18 The number of bits in a file storage unit is given by the constant FILE\_STORAGE\_SIZE (13.8.2.3)  
19 defined in the intrinsic module ISO\_FORTRAN\_ENV. It is recommended that the file storage unit be  
20 an 8-bit octet where this choice is practical.

#### **NOTE 9.12**

The requirement that every data value occupy an integer number of file storage units implies that data items inherently smaller than a file storage unit will require padding. This suggests that the file storage unit be small to avoid wasted space. Ideally, the file storage unit would be chosen such that padding is never required. A file storage unit of one bit would always meet this goal, but would likely be impractical because of the alignment requirements.

The prohibition on alignment restrictions prohibits the processor from requiring data alignments larger than the file storage unit.

The 8-bit octet is recommended as a good compromise that is small enough to accommodate the requirements of many applications, yet not so small that the data alignment requirements are likely to cause significant performance problems.

### 21 **9.3 Internal files**

22 Internal files provide a means of transferring and converting data from internal storage to internal  
23 storage.

24 An internal file is a record file with the following properties:

- 25 (1) The file is a variable of default, ASCII, or ISO 10646 character type that is not an array  
26 section with a vector subscript.
- 27 (2) A record of an internal file is a scalar character variable.
- 28 (3) If the file is a scalar character variable, it consists of a single record whose length is the same  
29 as the length of the scalar character variable. If the file is a character array, it is treated  
30 as a sequence of character array elements. Each array element, if any, is a record of the  
31 file. The ordering of the records of the file is the same as the ordering of the array elements

- 1           in the array (6.2.2.2) or the array section (6.2.2.3). Every record of the file has the same  
2           length, which is the length of an array element in the array.
- 3       (4) A record of the internal file becomes defined by writing the record. If the number of  
4       characters written in a record is less than the length of the record, the remaining portion  
5       of the record is filled with blanks. The number of characters to be written shall not exceed  
6       the length of the record.
- 7       (5) A record may be read only if the record is defined.
- 8       (6) A record of an internal file may become defined (or undefined) by means other than an  
9       output statement. For example, the character variable may become defined by a character  
10      assignment statement.
- 11      (7) An internal file is always positioned at the beginning of the first record prior to data transfer,  
12      except for child data transfer statements (9.5.3.7). This record becomes the current record.
- 13      (8) The initial value of a connection mode (9.4.1) is the value that would be implied by an  
14      initial OPEN statement without the corresponding keyword.
- 15      (9) Reading and writing records shall be accomplished only by sequential access formatted  
16      input/output statements.
- 17      (10) An internal file shall not be specified as the unit in a file connection statement, a file  
18      positioning statement, or a file inquiry statement.

## 19 9.4 File connection

20 A **unit**, specified by an *io-unit*, provides a means for referring to a file.

21 R901	<i>io-unit</i>	<b>is</b>	<i>file-unit-number</i>
22		<b>or</b>	*
23		<b>or</b>	<i>internal-file-variable</i>
24 R902	<i>file-unit-number</i>	<b>is</b>	<i>scalar-int-expr</i>
25 R903	<i>internal-file-variable</i>	<b>is</b>	<i>char-variable</i>

26 C901 (R903) The *char-variable* shall not be an array section with a vector subscript.

27 C902 (R903) The *char-variable* shall be of type default character, ASCII character, or ISO 10646  
28 character.

29 A unit is either an external unit or an internal unit. An **external unit** is used to refer to an external  
30 file and is specified by an asterisk or a *file-unit-number* whose value is nonnegative or equal to one of  
31 the named constants INPUT\_UNIT, OUTPUT\_UNIT, or ERROR\_UNIT of the ISO\_FORTRAN\_ENV  
32 module (13.8.2). An **internal unit** is used to refer to an internal file and is specified by an *internal-*  
33 *file-variable* or a *file-unit-number* whose value is equal to the **unit** argument of an active derived-type  
34 input/output procedure (9.5.3.7). The value of a *file-unit-number* shall identify a valid unit.

35 The external unit identified by a particular value of a *scalar-int-expr* is the same external unit in all  
36 program units of the program.

### NOTE 9.13

In the example:

```

SUBROUTINE A
  READ (6) X
  ...
SUBROUTINE B
  N = 6
  REWIND N

```

**NOTE 9.13 (cont.)**

the value 6 used in both program units identifies the same external unit.

1 An asterisk identifies particular processor-dependent external units that are preconnected for format-  
2 ted sequential access (9.5.3.2). These units are also identified by unit numbers defined by the named  
3 constants INPUT\_UNIT and OUTPUT\_UNIT of the ISO\_FORTRAN\_ENV module (13.8.2).

4 This standard identifies a processor-dependent external unit for the purpose of error reporting. This  
5 unit shall be preconnected for sequential formatted output. The processor may define this to be the  
6 same as the output unit identified by an asterisk. This unit is also identified by a unit number defined  
7 by the named constant ERROR\_UNIT of the ISO\_FORTRAN\_ENV intrinsic module.

**8 9.4.1 Connection modes**

9 A connection for formatted input/output has several changeable modes: the blank interpretation mode  
10 (10.7.6), delimiter mode (10.9.2, 10.10.2.1), sign mode (10.7.4), decimal edit mode (10.7.8), I/O round-  
11 ing mode (10.6.1.2.6), pad mode (9.5.3.4.2), and scale factor (10.7.5). A connection for unformatted  
12 input/output has no changeable modes.

13 Values for the modes of a connection are established when the connection is initiated. If the connection  
14 is initiated by an OPEN statement, the values are as specified, either explicitly or implicitly, by the  
15 OPEN statement. If the connection is initiated other than by an OPEN statement (that is, if the file is  
16 an internal file or preconnected file) the values established are those that would be implied by an initial  
17 OPEN statement without the corresponding keywords.

18 The scale factor cannot be explicitly specified in an OPEN statement; it is implicitly 0.

19 The modes of a connection to an external file may be changed by a subsequent OPEN statement that  
20 modifies the connection.

21 The modes of a connection may be temporarily changed by a corresponding keyword specifier in a  
22 data transfer statement or by an edit descriptor. Keyword specifiers take effect at the beginning of  
23 execution of the data transfer statement. Edit descriptors take effect when they are encountered in  
24 format processing. When a data transfer statement terminates, the values for the modes are reset to the  
25 values in effect immediately before the data transfer statement was executed.

**26 9.4.2 Unit existence**

27 At any given time, there is a processor-dependent set of external units that are said to exist for a  
28 program.

29 All input/output statements may refer to units that exist. The CLOSE, INQUIRE, and WAIT state-  
30 ments also may refer to units that do not exist.

**31 9.4.3 Connection of a file to a unit**

32 An external unit has a property of being **connected** or not connected. If connected, it refers to an  
33 external file. An external unit may become connected by preconnection or by the execution of an OPEN  
34 statement. The property of connection is symmetric; the unit is connected to a file if and only if the file  
35 is connected to the unit.

36 Every input/output statement except an OPEN, CLOSE, INQUIRE, or WAIT statement shall refer to  
37 a unit that is connected to a file and thereby make use of or affect that file.

1 A file may be connected and not exist (9.2.1).

**NOTE 9.14**

An example is a preconnected external file that has not yet been written.

2 A unit shall not be connected to more than one file at the same time, and a file shall not be connected to  
3 more than one unit at the same time. However, means are provided to change the status of an external  
4 unit and to connect a unit to a different file.

5 This standard defines means of portable interoperability with C. C streams are described in 7.19.2 of  
6 the C standard. Whether a unit may be connected to a file that is also connected to a C stream is  
7 processor dependent. If the processor allows a unit to be connected to a file that is also connected to  
8 a C stream, the results of performing input/output operations on such a file are processor dependent.  
9 It is processor dependent whether the files connected to the units INPUT\_UNIT, OUTPUT\_UNIT,  
10 and ERROR\_UNIT correspond to the predefined C text streams standard input, standard output, and  
11 standard error. If a procedure defined by means of Fortran and a procedure defined by means other than  
12 Fortran perform input/output operations on the same external file, the results are processor dependent.  
13 A procedure defined by means of Fortran and a procedure defined by means other than Fortran can  
14 perform input/output operations on different external files without interference.

15 After an external unit has been disconnected by the execution of a CLOSE statement, it may be con-  
16 nected again within the same program to the same file or to a different file. After an external file has  
17 been disconnected by the execution of a CLOSE statement, it may be connected again within the same  
18 program to the same unit or to a different unit.

**NOTE 9.15**

The only means of referencing a file that has been disconnected is by the appearance of its name  
in an OPEN or INQUIRE statement. There might be no means of reconnecting an unnamed file  
once it is disconnected.

19 An internal unit is always connected to the internal file designated by the variable that identifies the  
20 unit.

**NOTE 9.16**

For more explanatory information on file connection properties, see C.6.5.

21 **9.4.4 Preconnection**

22 **Preconnection** means that the unit is connected to a file at the beginning of execution of the program  
23 and therefore it may be specified in input/output statements without the prior execution of an OPEN  
24 statement.

25 **9.4.5 The OPEN statement**

26 An **OPEN statement** initiates or modifies the connection between an external file and a specified unit.  
27 The OPEN statement may be used to connect an existing file to a unit, create a file that is preconnected,  
28 create a file and connect it to a unit, or change certain modes of a connection between a file and a unit.

29 An external unit may be connected by an OPEN statement in any program unit of a program and, once  
30 connected, a reference to it may appear in any program unit of the program.

31 If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted.  
32 If the FILE= specifier is not included in such an OPEN statement, the file to be connected to the unit  
33 is the same as the file to which the unit is already connected.

1 If the file to be connected to the unit does not exist but is the same as the file to which the unit is  
2 preconnected, the modes specified by an OPEN statement become a part of the connection.

3 If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect  
4 is as if a CLOSE statement without a STATUS= specifier had been executed for the unit immediately  
5 prior to the execution of an OPEN statement.

6 If the file to be connected to the unit is the same as the file to which the unit is connected, only the  
7 specifiers for changeable modes (9.4.1) may have values different from those currently in effect. If the  
8 POSITION= specifier appears in such an OPEN statement, the value specified shall not disagree with  
9 the current position of the file. If the STATUS= specifier is included in such an OPEN statement, it shall  
10 be specified with the value OLD. Execution of such an OPEN statement causes any new values of the  
11 specifiers for changeable modes to be in effect, but does not cause any change in any of the unspecified  
12 specifiers and the position of the file is unaffected. The ERR=, IOSTAT=, and IOMSG= specifiers from  
13 any previously executed OPEN statement have no effect on any currently executed OPEN statement.

14 A STATUS= specifier with a value of OLD is always allowed when the file to be connected to the unit is  
15 the same as the file to which the unit is connected. In this case, if the status of the file was SCRATCH  
16 before execution of the OPEN statement, the file will still be deleted when the unit is closed, and the  
17 file is still considered to have a status of SCRATCH.

18 If a file is already connected to a unit, execution of an OPEN statement on that file and a different unit  
19 is not permitted.

20	R904	<i>open-stmt</i>	<b>is</b>	OPEN ( <i>connect-spec-list</i> )
21	R905	<i>connect-spec</i>	<b>is</b>	[ UNIT = ] <i>file-unit-number</i>
22			<b>or</b>	ACCESS = <i>scalar-default-char-expr</i>
23			<b>or</b>	ACTION = <i>scalar-default-char-expr</i>
24			<b>or</b>	ASYNCHRONOUS = <i>scalar-default-char-expr</i>
25			<b>or</b>	BLANK = <i>scalar-default-char-expr</i>
26			<b>or</b>	DECIMAL = <i>scalar-default-char-expr</i>
27			<b>or</b>	DELIM = <i>scalar-default-char-expr</i>
28			<b>or</b>	ENCODING = <i>scalar-default-char-expr</i>
29			<b>or</b>	ERR = <i>label</i>
30			<b>or</b>	FILE = <i>file-name-expr</i>
31			<b>or</b>	FORM = <i>scalar-default-char-expr</i>
32			<b>or</b>	IOMSG = <i>iomsg-variable</i>
33			<b>or</b>	IOSTAT = <i>scalar-int-variable</i>
34			<b>or</b>	PAD = <i>scalar-default-char-expr</i>
35			<b>or</b>	POSITION = <i>scalar-default-char-expr</i>
36			<b>or</b>	RECL = <i>scalar-int-expr</i>
37			<b>or</b>	ROUND = <i>scalar-default-char-expr</i>
38			<b>or</b>	SIGN = <i>scalar-default-char-expr</i>
39			<b>or</b>	STATUS = <i>scalar-default-char-expr</i>
40	R906	<i>file-name-expr</i>	<b>is</b>	<i>scalar-default-char-expr</i>
41	R907	<i>iomsg-variable</i>	<b>is</b>	<i>scalar-default-char-variable</i>

42 C903 (R905) No specifier shall appear more than once in a given *connect-spec-list*.

43 C904 (R905) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the  
44 *file-unit-number* shall be the first item in the *connect-spec-list*.

45 C905 (R905) The *label* used in the ERR= specifier shall be the statement label of a branch target  
46 statement that appears in the same scoping unit as the OPEN statement.

47 If the STATUS= specifier has the value NEW or REPLACE, the FILE= specifier shall appear. If the

- 1 STATUS= specifier has the value SCRATCH, the FILE= specifier shall not appear. If the STATUS=  
 2 specifier has the value OLD, the FILE= specifier shall appear unless the unit is connected and the file  
 3 connected to the unit exists.
- 4 A specifier that requires a *scalar-default-char-expr* may have a limited list of character values. These  
 5 values are listed for each such specifier. Any trailing blanks are ignored. The value specified is without  
 6 regard to case. Some specifiers have a default value if the specifier is omitted.
- 7 The IOSTAT=, ERR=, and IOMSG= specifiers are described in [9.10](#).

**NOTE 9.17**

An example of an OPEN statement is:

```
OPEN (10, FILE = 'employee.names', ACTION = 'READ', PAD = 'YES')
```

**NOTE 9.18**

For more explanatory information on the OPEN statement, see [C.6.4](#).

**8 9.4.5.1 ACCESS= specifier in the OPEN statement**

- 9 The *scalar-default-char-expr* shall evaluate to SEQUENTIAL, DIRECT, or STREAM. The ACCESS=  
 10 specifier specifies the access method for the connection of the file as being sequential, direct, or stream.  
 11 If this specifier is omitted, the default value is SEQUENTIAL. For an existing file, the specified access  
 12 method shall be included in the set of allowed access methods for the file. For a new file, the processor  
 13 creates the file with a set of allowed access methods that includes the specified method.

**14 9.4.5.2 ACTION= specifier in the OPEN statement**

- 15 The *scalar-default-char-expr* shall evaluate to READ, WRITE, or READWRITE. READ specifies that  
 16 the WRITE, PRINT, and ENDFILE statements shall not refer to this connection. WRITE specifies  
 17 that READ statements shall not refer to this connection. READWRITE permits any input/output  
 18 statements to refer to this connection. If this specifier is omitted, the default value is processor dependent.  
 19 If READWRITE is included in the set of allowable actions for a file, both READ and WRITE also shall  
 20 be included in the set of allowed actions for that file. For an existing file, the specified action shall be  
 21 included in the set of allowed actions for the file. For a new file, the processor creates the file with a set  
 22 of allowed actions that includes the specified action.

**23 9.4.5.3 ASYNCHRONOUS= specifier in the OPEN statement**

- 24 The *scalar-default-char-expr* shall evaluate to YES or NO. If YES is specified, asynchronous input/output  
 25 on the unit is allowed. If NO is specified, asynchronous input/output on the unit is not allowed. If this  
 26 specifier is omitted, the default value is NO.

**27 9.4.5.4 BLANK= specifier in the OPEN statement**

- 28 The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier is permitted only  
 29 for a connection for formatted input/output. It specifies the current value of the blank interpretation  
 30 mode ([10.7.6](#), [9.5.1.5](#)) for input for this connection. This mode has no effect on output. It is a changeable  
 31 mode ([9.4.1](#)). If this specifier is omitted in an OPEN statement that initiates a connection, the default  
 32 value is NULL.

**1 9.4.5.5 DECIMAL= specifier in the OPEN statement**

2 The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier is per-  
3 mitted only for a connection for formatted input/output. It specifies the current value of the decimal  
4 edit mode (10.7.8, 9.5.1.6) for this connection. This is a changeable mode (9.4.1). If this specifier is  
5 omitted in an OPEN statement that initiates a connection, the default value is POINT.

**6 9.4.5.6 DELIM= specifier in the OPEN statement**

7 The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= spec-  
8 ifier is permitted only for a connection for formatted input/output. It specifies the current value of the  
9 delimiter mode (9.5.1.7) for list-directed (10.9.2) and namelist (10.10.2.1) output for the connection.  
10 This mode has no effect on input. It is a changeable mode (9.4.1). If this specifier is omitted in an  
11 OPEN statement that initiates a connection, the default value is NONE.

**12 9.4.5.7 ENCODING= specifier in the OPEN statement**

13 The *scalar-default-char-expr* shall evaluate to UTF-8 or DEFAULT. The ENCODING= specifier is  
14 permitted only for a connection for formatted input/output. The value UTF-8 specifies that the encoding  
15 form of the file is UTF-8 as specified by ISO/IEC 10646-1:2000. Such a file is called a **Unicode** file,  
16 and all characters therein are of ISO 10646 character type. The value UTF-8 shall not be specified if  
17 the processor does not support the ISO 10646 character type. The value DEFAULT specifies that the  
18 encoding form of the file is processor-dependent. If this specifier is omitted in an OPEN statement that  
19 initiates a connection, the default value is DEFAULT.

**20 9.4.5.8 FILE= specifier in the OPEN statement**

21 The value of the FILE= specifier is the name of the file to be connected to the specified unit. Any trailing  
22 blanks are ignored. The *file-name-expr* shall be a name that is allowed by the processor. If this specifier  
23 is omitted and the unit is not connected to a file, the STATUS= specifier shall be specified with a value  
24 of SCRATCH; in this case, the connection is made to a processor-dependent file. The interpretation of  
25 case is processor dependent.

**26 9.4.5.9 FORM= specifier in the OPEN statement**

27 The *scalar-default-char-expr* shall evaluate to FORMATTED or UNFORMATTED. The FORM= spec-  
28 ifier determines whether the file is being connected for formatted or unformatted input/output. If this  
29 specifier is omitted, the default value is UNFORMATTED if the file is being connected for direct access  
30 or stream access, and the default value is FORMATTED if the file is being connected for sequential  
31 access. For an existing file, the specified form shall be included in the set of allowed forms for the file.  
32 For a new file, the processor creates the file with a set of allowed forms that includes the specified form.

**33 9.4.5.10 PAD= specifier in the OPEN statement**

34 The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier is permitted only for a  
35 connection for formatted input/output. It specifies the current value of the pad mode (9.5.3.4.2, 9.5.1.9)  
36 for input for this connection. This mode has no effect on output. It is a changeable mode (9.4.1). If this  
37 specifier is omitted in an OPEN statement that initiates a connection, the default value is YES.

**NOTE 9.19**

For nondefault character types, the blank padding character is processor dependent.

1 **9.4.5.11 POSITION= specifier in the OPEN statement**

2 The *scalar-default-char-expr* shall evaluate to ASIS, REWIND, or APPEND. The connection shall be  
3 for sequential or stream access. A new file is positioned at its initial point. REWIND positions an  
4 existing file at its initial point. APPEND positions an existing file such that the endfile record is the  
5 next record, if it has one. If an existing file does not have an endfile record, APPEND positions the  
6 file at its terminal point. ASIS leaves the position unchanged if the file exists and already is connected.  
7 ASIS leaves the position unspecified if the file exists but is not connected. If this specifier is omitted,  
8 the default value is ASIS.

9 **9.4.5.12 RECL= specifier in the OPEN statement**

10 The value of the RECL= specifier shall be positive. It specifies the length of each record in a file being  
11 connected for direct access, or specifies the maximum length of a record in a file being connected for  
12 sequential access. This specifier shall not appear when a file is being connected for stream access. This  
13 specifier shall appear when a file is being connected for direct access. If this specifier is omitted when  
14 a file is being connected for sequential access, the default value is processor dependent. If the file is  
15 being connected for formatted input/output, the length is the number of characters for all records that  
16 contain only characters of type default character. When a record contains any nondefault characters,  
17 the appropriate value for the RECL= specifier is processor dependent. If the file is being connected for  
18 unformatted input/output, the length is measured in file storage units. For an existing file, the value of  
19 the RECL= specifier shall be included in the set of allowed record lengths for the file. For a new file,  
20 the processor creates the file with a set of allowed record lengths that includes the specified value.

21 **9.4.5.13 ROUND= specifier in the OPEN statement**

22 The *scalar-default-char-expr* shall evaluate to one of UP, DOWN, ZERO, NEAREST, COMPATIBLE,  
23 or PROCESSOR\_DEFINED. The ROUND= specifier is permitted only for a connection for formatted  
24 input/output. It specifies the current value of the I/O rounding mode (10.6.1.2.6, 9.5.1.12) for this  
25 connection. This is a changeable mode (9.4.1). If this specifier is omitted in an OPEN statement that  
26 initiates a connection, the I/O rounding mode is processor dependent; it shall be one of the above modes.

**NOTE 9.20**

A processor is free to select any I/O rounding mode for the default mode. The mode might correspond to UP, DOWN, ZERO, NEAREST, or COMPATIBLE; or it might be a completely different I/O rounding mode.

27 **9.4.5.14 SIGN= specifier in the OPEN statement**

28 The *scalar-default-char-expr* shall evaluate to one of PLUS, SUPPRESS, or PROCESSOR\_DEFINED.  
29 The SIGN= specifier is permitted only for a connection for formatted input/output. It specifies the  
30 current value of the sign mode (10.7.4, 9.5.1.13) for this connection. This is a changeable mode (9.4.1).  
31 If this specifier is omitted in an OPEN statement that initiates a connection, the default value is PRO-  
32 CESSOR\_DEFINED.

33 **9.4.5.15 STATUS= specifier in the OPEN statement**

34 The *scalar-default-char-expr* shall evaluate to OLD, NEW, SCRATCH, REPLACE, or UNKNOWN. If  
35 OLD is specified, the file shall exist. If NEW is specified, the file shall not exist.

36 Successful execution of an OPEN statement with NEW specified creates the file and changes the status  
37 to OLD. If REPLACE is specified and the file does not already exist, the file is created and the status is  
38 changed to OLD. If REPLACE is specified and the file does exist, the file is deleted, a new file is created  
39 with the same name, and the status is changed to OLD. If SCRATCH is specified, the file is created

1 and connected to the specified unit for use by the program but is deleted at the execution of a CLOSE  
2 statement referring to the same unit or at the normal termination of the program.

**NOTE 9.21**

SCRATCH shall not be specified with a named file.

3 If UNKNOWN is specified, the status is processor dependent. If this specifier is omitted, the default  
4 value is UNKNOWN.

5 **9.4.6 The CLOSE statement**

6 The **CLOSE statement** is used to terminate the connection of a specified unit to an external file.

7 Execution of a CLOSE statement for a unit may occur in any program unit of a program and need not  
8 occur in the same program unit as the execution of an OPEN statement referring to that unit.

9 Execution of a CLOSE statement performs a wait operation for any pending asynchronous data transfer  
10 operations for the specified unit.

11 Execution of a CLOSE statement specifying a unit that does not exist or has no file connected to it is  
12 permitted and affects no file.

13 After a unit has been disconnected by execution of a CLOSE statement, it may be connected again  
14 within the same program, either to the same file or to a different file. After a named file has been  
15 disconnected by execution of a CLOSE statement, it may be connected again within the same program,  
16 either to the same unit or to a different unit, provided that the file still exists.

17 At normal termination of execution of a program, all units that are connected are closed. Each unit  
18 is closed with status KEEP unless the file status prior to termination of execution was SCRATCH, in  
19 which case the unit is closed with status DELETE.

**NOTE 9.22**

The effect is as though a CLOSE statement without a STATUS= specifier were executed on each  
connected unit.

20 R908 *close-stmt* is CLOSE ( *close-spec-list* )  
21 R909 *close-spec* is [ UNIT = ] *file-unit-number*  
22 or IOSTAT = *scalar-int-variable*  
23 or IOMSG = *iomsg-variable*  
24 or ERR = *label*  
25 or STATUS = *scalar-default-char-expr*

26 C906 (R909) No specifier shall appear more than once in a given *close-spec-list*.

27 C907 (R909) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the  
28 *file-unit-number* shall be the first item in the *close-spec-list*.

29 C908 (R909) The *label* used in the ERR= specifier shall be the statement label of a branch target  
30 statement that appears in the same scoping unit as the CLOSE statement.

31 The *scalar-default-char-expr* has a limited list of character values. Any trailing blanks are ignored. The  
32 value specified is without regard to case.

33 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

**NOTE 9.23**

An example of a CLOSE statement is:

```
CLOSE (10, STATUS = 'KEEP')
```

**NOTE 9.24**

For more explanatory information on the CLOSE statement, see [C.6.6](#).

**1 9.4.6.1 STATUS= specifier in the CLOSE statement**

2 The *scalar-default-char-expr* shall evaluate to KEEP or DELETE. The STATUS= specifier determines  
 3 the disposition of the file that is connected to the specified unit. KEEP shall not be specified for a file  
 4 whose status prior to execution of a CLOSE statement is SCRATCH. If KEEP is specified for a file that  
 5 exists, the file continues to exist after the execution of a CLOSE statement. If KEEP is specified for a  
 6 file that does not exist, the file will not exist after the execution of a CLOSE statement. If DELETE is  
 7 specified, the file will not exist after the execution of a CLOSE statement. If this specifier is omitted, the  
 8 default value is KEEP, unless the file status prior to execution of the CLOSE statement is SCRATCH,  
 9 in which case the default value is DELETE.

**10 9.5 Data transfer statements**

11 The **READ statement** is the data transfer input statement. The **WRITE statement** and the  
 12 **PRINT statement** are the data transfer output statements.

```
13 R910  read-stmt           is  READ ( io-control-spec-list ) [ input-item-list ]
14                                     or  READ format [ , input-item-list ]
15 R911  write-stmt          is  WRITE ( io-control-spec-list ) [ output-item-list ]
16 R912  print-stmt         is  PRINT format [ , output-item-list ]
```

**NOTE 9.25**

Examples of data transfer statements are:

```
  READ (6, *) SIZE
  READ 10, A, B
  WRITE (6, 10) A, S, J
  PRINT 10, A, S, J
10 FORMAT (2E16.3, I5)
```

**17 9.5.1 Control information list**

18 A **control information list** is an *io-control-spec-list*. It governs data transfer.

```
19 R913  io-control-spec    is  [ UNIT = ] io-unit
20                                     or  [ FMT = ] format
21                                     or  [ NML = ] namelist-group-name
22                                     or  ADVANCE = scalar-default-char-expr
23                                     or  ASYNCHRONOUS = scalar-char-initialization-expr
24                                     or  BLANK = scalar-default-char-expr
25                                     or  DECIMAL = scalar-default-char-expr
26                                     or  DELIM = scalar-default-char-expr
27                                     or  END = label
28                                     or  EOR = label
```

- 1                   or ERR = *label*  
 2                   or ID = *scalar-int-variable*  
 3                   or IOMSG = *iomsg-variable*  
 4                   or IOSTAT = *scalar-int-variable*  
 5                   or PAD = *scalar-default-char-expr*  
 6                   or POS = *scalar-int-expr*  
 7                   or REC = *scalar-int-expr*  
 8                   or ROUND = *scalar-default-char-expr*  
 9                   or SIGN = *scalar-default-char-expr*  
 10                  or SIZE = *scalar-int-variable*
- 11 C909 (R913) No specifier shall appear more than once in a given *io-control-spec-list*.
- 12 C910 (R913) An *io-unit* shall be specified; if the optional characters UNIT= are omitted, the *io-unit*  
 13 shall be the first item in the *io-control-spec-list*.
- 14 C911 (R913) A DELIM= or SIGN= specifier shall not appear in a *read-stmt*.
- 15 C912 (R913) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.
- 16 C913 (R913) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a  
 17 branch target statement that appears in the same scoping unit as the data transfer statement.
- 18 C914 (R913) A *namelist-group-name* shall be the name of a namelist group.
- 19 C915 (R913) A *namelist-group-name* shall not appear if an *input-item-list* or an *output-item-list*  
 20 appears in the data transfer statement.
- 21 C916 (R913) An *io-control-spec-list* shall not contain both a *format* and a *namelist-group-name*.
- 22 C917 (R913) If *format* appears without a preceding FMT=, it shall be the second item in the *io-*  
 23 *control-spec-list* and the first item shall be *io-unit*.
- 24 C918 (R913) If *namelist-group-name* appears without a preceding NML=, it shall be the second item  
 25 in the *io-control-spec-list* and the first item shall be *io-unit*.
- 26 C919 (R913) If *io-unit* is not a *file-unit-number*, the *io-control-spec-list* shall not contain a REC=  
 27 specifier or a POS= specifier.
- 28 C920 (R913) If the REC= specifier appears, an END= specifier shall not appear, a *namelist-group-*  
 29 *name* shall not appear, and the *format*, if any, shall not be an asterisk.
- 30 C921 (R913) An ADVANCE= specifier may appear only in a formatted sequential or stream in-  
 31 put/output statement with explicit format specification (10.1) whose control information list  
 32 does not contain an *internal-file-variable* as the *io-unit*.
- 33 C922 (R913) If an EOR= specifier appears, an ADVANCE= specifier also shall appear.
- 34 C923 (R913) If a SIZE= specifier appears, an ADVANCE= specifier also shall appear.
- 35 C924 (R913) The *scalar-char-initialization-expr* in an ASYNCHRONOUS= specifier shall be of type  
 36 default character and shall have the value YES or NO.
- 37 C925 (R913) An ASYNCHRONOUS= specifier with a value YES shall not appear unless *io-unit* is a  
 38 *file-unit-number*.
- 39 C926 (R913) If an ID= specifier appears, an ASYNCHRONOUS= specifier with the value YES shall

- 1 also appear.
- 2 C927 (R913) If a POS= specifier appears, the *io-control-spec-list* shall not contain a REC= specifier.
- 3 C928 (R913) If a DECIMAL=, BLANK=, PAD=, SIGN=, or ROUND= specifier appears, a *format*  
4 or *namelist-group-name* shall also appear.
- 5 C929 (R913) If a DELIM= specifier appears, either *format* shall be an asterisk or *namelist-group-name*  
6 shall appear.
- 7 A SIZE= specifier may appear only in an input statement that contains an ADVANCE= specifier with  
8 the value NO.
- 9 An EOR= specifier may appear only in an input statement that contains an ADVANCE= specifier with  
10 the value NO.
- 11 If the data transfer statement contains a *format* or *namelist-group-name*, the statement is a **formatted**  
12 **input/output statement**; otherwise, it is an **unformatted input/output statement**.
- 13 The ADVANCE=, ASYNCHRONOUS=, DECIMAL=, BLANK=, DELIM=, PAD=, SIGN=, and  
14 ROUND= specifiers have a limited list of character values. Any trailing blanks are ignored. The  
15 values specified are without regard to case.
- 16 The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.10.

**NOTE 9.26**

An example of a READ statement is:

```
READ (IOSTAT = IOS, UNIT = 6, FMT = '(10F8.2)') A, B
```

17 **9.5.1.1 FMT= specifier in a data transfer statement**

18 The FMT= specifier supplies a format specification or specifies list-directed formatting for a formatted  
19 input/output statement.

20 R914 *format*                                    **is** *default-char-expr*  
21    **or** *label*  
22    **or** \*

23 C930 (R914) The *label* shall be the label of a FORMAT statement that appears in the same scoping  
24 unit as the statement containing the FMT= specifier.

25 The *default-char-expr* shall evaluate to a valid format specification (10.1.1 and 10.1.2).

**NOTE 9.27**

A *default-char-expr* includes a character constant.

26 If *default-char-expr* is an array, it is treated as if all of the elements of the array were specified in array  
27 element order and were concatenated.

28 If *format* is \*, the statement is a **list-directed input/output statement**.

**NOTE 9.28**

An example in which the format is a character expression is:

```
READ (6, FMT = "(" // CHAR_FMT // ")") X, Y, Z
```

**NOTE 9.28 (cont.)**

where CHAR\_FMT is a default character variable.

**1 9.5.1.2 NML= specifier in a data transfer statement**

2 The NML= specifier supplies the *namelist-group-name* (5.4). This name identifies a particular collection  
3 of data objects on which transfer is to be performed.

4 If a *namelist-group-name* appears, the statement is a **namelist input/output statement**.

**5 9.5.1.3 ADVANCE= specifier in a data transfer statement**

6 The *scalar-default-char-expr* shall evaluate to YES or NO. The ADVANCE= specifier determines wheth-  
7 er advancing input/output occurs for this input/output statement. If YES is specified, advancing in-  
8 put/output occurs. If NO is specified, nonadvancing input/output occurs (9.2.3.1). If this specifier is  
9 omitted from an input/output statement that allows the specifier, the default value is YES.

**10 9.5.1.4 ASYNCHRONOUS= specifier in a data transfer statement**

11 The ASYNCHRONOUS= specifier determines whether this input/output statement is synchronous or  
12 asynchronous. If YES is specified, the statement and the input/output operation are said to be asyn-  
13 chronous. If NO is specified or if the specifier is omitted, the statement and the input/output operation  
14 are said to be synchronous.

15 Asynchronous input/output is permitted only for external files opened with an ASYNCHRONOUS=  
16 specifier with the value YES in the OPEN statement.

**NOTE 9.29**

Both synchronous and asynchronous input/output are allowed for files opened with an ASYN-  
CHRONOUS= specifier of YES. For other files, only synchronous input/output is allowed; this  
includes files opened with an ASYNCHRONOUS= specifier of NO, files opened without an ASYN-  
CHRONOUS= specifier, preconnected files accessed without an OPEN statement, and internal  
files.

The ASYNCHRONOUS= specifier value in a data transfer statement is an initialization expression  
because it effects compiler optimizations and, therefore, needs to be known at compile time.

17 The processor may perform an asynchronous data transfer operation asynchronously, but it is not re-  
18 quired to do so. For each external file, records and file storage units read or written by asynchronous  
19 data transfer statements are read, written, and processed in the same order as they would have been if  
20 the data transfer statements were synchronous.

21 If a variable is used in an asynchronous data transfer statement as

- 22 (1) an item in an input/output list,
- 23 (2) a group object in a namelist, or
- 24 (3) a SIZE= specifier

25 the base object of the *data-ref* is implicitly given the ASYNCHRONOUS attribute in the scoping unit  
26 of the data transfer statement. This attribute may be confirmed by explicit declaration.

27 When an asynchronous input/output statement is executed, the set of storage units specified by the  
28 item list or NML= specifier, plus the storage units specified by the SIZE= specifier, is defined to be the  
29 pending input/output storage sequence for the data transfer operation.

**NOTE 9.30**

A pending input/output storage sequence is not necessarily a contiguous set of storage units.

1 A pending input/output storage sequence **affector** is a variable of which any part is associated with a  
2 storage unit in a pending input/output storage sequence.

**3 9.5.1.5 BLANK= specifier in a data transfer statement**

4 The *scalar-default-char-expr* shall evaluate to NULL or ZERO. The BLANK= specifier temporarily  
5 changes (9.4.1) the blank interpretation mode (10.7.6, 9.4.5.4) for the connection. If the specifier is  
6 omitted, the mode is not changed.

**7 9.5.1.6 DECIMAL= specifier in a data transfer statement**

8 The *scalar-default-char-expr* shall evaluate to COMMA or POINT. The DECIMAL= specifier temporar-  
9 ily changes (9.4.1) the decimal edit mode (10.7.8, 9.4.5.5) for the connection. If the specifier is omitted,  
10 the mode is not changed.

**11 9.5.1.7 DELIM= specifier in a data transfer statement**

12 The *scalar-default-char-expr* shall evaluate to APOSTROPHE, QUOTE, or NONE. The DELIM= spec-  
13 ifier temporarily changes (9.4.1) the delimiter mode (10.9.2, 10.10.2.1, 9.4.5.6) for the connection. If the  
14 specifier is omitted, the mode is not changed.

**15 9.5.1.8 ID= specifier in a data transfer statement**

16 Successful execution of an asynchronous data transfer statement containing an ID= specifier causes the  
17 variable specified in the ID= specifier to become defined with a processor-dependent value. This value  
18 is referred to as the identifier of the data transfer operation. It can be used in a subsequent WAIT or  
19 INQUIRE statement to identify the particular data transfer operation.

20 If an error occurs during the execution of a data transfer statement containing an ID= specifier, the  
21 variable specified in the ID= specifier becomes undefined.

22 A child data transfer statement shall not specify the ID= specifier.

**23 9.5.1.9 PAD= specifier in a data transfer statement**

24 The *scalar-default-char-expr* shall evaluate to YES or NO. The PAD= specifier temporarily changes  
25 (9.4.1) the pad mode (9.5.3.4.2, 9.4.5.10) for the connection. If the specifier is omitted, the mode is not  
26 changed.

**27 9.5.1.10 POS= specifier in a data transfer statement**

28 The POS= specifier specifies the file position in file storage units. This specifier may appear in a data  
29 transfer statement only if the statement specifies a unit connected for stream access. A child data  
30 transfer statement shall not specify this specifier.

31 A processor may prohibit the use of POS= with particular files that do not have the properties necessary  
32 to support random positioning. A processor may also prohibit positioning a particular file to any  
33 position prior to its current file position if the file does not have the properties necessary to support such  
34 positioning.

**NOTE 9.31**

A file that represents connection to a device or data stream might not be positionable.

1 If the file is connected for formatted stream access, the file position specified by POS= shall be equal to  
 2 either 1 (the beginning of the file) or a value previously returned by a POS= specifier in an INQUIRE  
 3 statement for the file.

4 **9.5.1.11 REC= specifier in a data transfer statement**

5 The REC= specifier specifies the number of the record that is to be read or written. This specifier  
 6 may appear only in an input/output statement that specifies a unit connected for direct access; it  
 7 shall not appear in a child data transfer statement. If the control information list contains a REC=  
 8 specifier, the statement is a **direct access input/output statement**. A child data transfer statement  
 9 is a direct access data transfer statement if the parent is a direct access data transfer statement. Any  
 10 other data transfer statement is a **sequential access input/output statement** or a **stream access**  
 11 **input/output statement**, depending on whether the file connection is for sequential access or stream  
 12 access.

13 **9.5.1.12 ROUND= specifier in a data transfer statement**

14 The *scalar-default-char-expr* shall evaluate to one of the values specified in 9.4.5.13. The ROUND=  
 15 specifier temporarily changes (9.4.1) the I/O rounding mode (10.6.1.2.6, 9.4.5.13) for the connection. If  
 16 the specifier is omitted, the mode is not changed.

17 **9.5.1.13 SIGN= specifier in a data transfer statement**

18 The *scalar-default-char-expr* shall evaluate to PLUS, SUPPRESS, or PROCESSOR\_DEFINED. The  
 19 SIGN= specifier temporarily changes (9.4.1) the sign mode (10.7.4, 9.4.5.14) for the connection. If the  
 20 specifier is omitted, the mode is not changed.

21 **9.5.1.14 SIZE= specifier in a data transfer statement**

22 When a synchronous nonadvancing input statement terminates, the variable specified in the SIZE=  
 23 specifier becomes defined with the count of the characters transferred by data edit descriptors during  
 24 execution of the current input statement. Blanks inserted as padding (9.5.3.4.2) are not counted.

25 For asynchronous nonadvancing input, the storage units specified in the SIZE= specifier become defined  
 26 with the count of the characters transferred when the corresponding wait operation is executed.

27 **9.5.2 Data transfer input/output list**

28 An input/output list specifies the entities whose values are transferred by a data transfer input/output  
 29 statement.

30	R915 <i>input-item</i>	is	<i>variable</i>
31		or	<i>io-implied-do</i>
32	R916 <i>output-item</i>	is	<i>expr</i>
33		or	<i>io-implied-do</i>
34	R917 <i>io-implied-do</i>	is	( <i>io-implied-do-object-list</i> , <i>io-implied-do-control</i> )
35	R918 <i>io-implied-do-object</i>	is	<i>input-item</i>
36		or	<i>output-item</i>
37	R919 <i>io-implied-do-control</i>	is	<i>do-variable</i> = <i>scalar-int-expr</i> , ■

- 1 ■ *scalar-int-expr* [ , *scalar-int-expr* ]
- 2 C931 (R915) A variable that is an *input-item* shall not be a whole assumed-size array.
- 3 C932 (R915) A variable that is an *input-item* shall not be a procedure pointer.
- 4 C933 (R919) The *do-variable* shall be a named scalar variable of type integer.
- 5 C934 (R918) In an *input-item-list*, an *io-implied-do-object* shall be an *input-item*. In an *output-item-*  
6 *list*, an *io-implied-do-object* shall be an *output-item*.
- 7 C935 (R916) An expression that is an *output-item* shall not have a value that is a procedure pointer.
- 8 An *input-item* shall not appear as, nor be associated with, the *do-variable* of any *io-implied-do* that  
9 contains the *input-item*.

**NOTE 9.32**

A constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear as an output list item but shall not appear as an input list item.

- 10 If an input item is a pointer, it shall be associated with a definable target and data are transferred from  
11 the file to the associated target. If an output item is a pointer, it shall be associated with a target and  
12 data are transferred from the target to the file.

**NOTE 9.33**

Data transfers always involve the movement of values between a file and internal storage. A pointer as such cannot be read or written. Therefore, a pointer shall not appear as an item in an input/output list unless it is associated with a target that can receive a value (input) or can deliver a value (output).

- 13 If an input item or an output item is allocatable, it shall be allocated.
- 14 A list item shall not be polymorphic unless it is processed by a user-defined derived-type input/output  
15 procedure (9.5.3.7).
- 16 The *do-variable* of an *io-implied-do* that is in another *io-implied-do* shall not appear as, nor be associated  
17 with, the *do-variable* of the containing *io-implied-do*.
- 18 The following rules describing whether to expand an input/output list item are re-applied to each  
19 expanded list item until none of the rules apply.
- 20 If an array appears as an input/output list item, it is treated as if the elements, if any, were specified in  
21 array element order (6.2.2.2). However, no element of that array may affect the value of any expression  
22 in the *input-item*, nor may any element appear more than once in an *input-item*.

**NOTE 9.34**

For example:

```

INTEGER A (100), J (100)
...
READ *, A (A)                ! Not allowed
READ *, A (LBOUND (A, 1) : UBOUND (A, 1)) ! Allowed
READ *, A (J)                ! Allowed if no two elements
                                !   of J have the same value
A(1) = 1; A(10) = 10

```

**NOTE 9.34 (cont.)**

READ *, A (A (1) : A (10))	! Not allowed
----------------------------	---------------

- 1 If a list item of derived type in an unformatted input/output statement is not processed by a user-defined  
 2 derived-type input/output procedure (9.5.3.7), and if any subobject of that list item would be processed  
 3 by a user-defined derived-type input/output procedure, the list item is treated as if all of the components  
 4 of the object were specified in the list in component order (4.5.3.5); those components shall be accessible  
 5 in the scoping unit containing the input/output statement and shall not be pointers or allocatable.
- 6 An effective input/output list item of derived type in an unformatted input/output statement is treated  
 7 as a single value in a processor-dependent form unless the list item or a subobject thereof is processed  
 8 by a user-defined derived-type input/output procedure (9.5.3.7).

**NOTE 9.35**

The appearance of a derived-type object as an input/output list item in an unformatted input/output statement is not equivalent to the list of its components.

Unformatted input/output involving derived-type list items forms the single exception to the rule that the appearance of an aggregate list item (such as an array) is equivalent to the appearance of its expanded list of component parts. This exception permits the processor greater latitude in improving efficiency or in matching the processor-dependent sequence of values for a derived-type object to similar sequences for aggregate objects used by means other than Fortran. However, formatted input/output of all list items and unformatted input/output of list items other than those of derived types adhere to the above rule.

- 9 If a list item of derived type in a formatted input/output statement is not processed by a user-defined  
 10 derived-type input/output procedure, that list item is treated as if all of the components of the list item  
 11 were specified in the list in component order; those components shall be accessible in the scoping unit  
 12 containing the input/output statement and shall not be pointers or allocatable.
- 13 If a derived-type list item is not treated as a list of its individual components, that list item's ultimate  
 14 components shall not have the POINTER or ALLOCATABLE attribute unless that list item is processed  
 15 by a user-defined derived-type input/output procedure.
- 16 The scalar objects resulting when a data transfer statement's list items are expanded according to the  
 17 rules in this section for handling array and derived-type list items are called **effective items**. Zero-sized  
 18 arrays and *io-implied-dos* with an iteration count of zero do not contribute to the effective list items. A  
 19 scalar character item of zero length is an effective list item.

**NOTE 9.36**

In a formatted input/output statement, edit descriptors are associated with effective list items, which are always scalar. The rules in 9.5.2 determine the set of effective list items corresponding to each actual list item in the statement. These rules may have to be applied repetitively until all of the effective list items are scalar items.

- 20 For an *io-implied-do*, the loop initialization and execution is the same as for a DO construct (8.1.6.4).
- 21 An input/output list shall not contain an item of nondefault character type if the input/output statement  
 22 specifies an internal file of default character type. An input/output list shall not contain an item of  
 23 nondefault character type other than ISO 10646 or ASCII character type if the input/output statement  
 24 specifies an internal file of ISO 10646 character type. An input/output list shall not contain a character  
 25 item of any character type other than ASCII character type if the input/output statement specifies an

1 internal file of ASCII character type.

**NOTE 9.37**

An example of an output list with an implied DO is:

```
WRITE (LP, FMT = '(10F8.2)') (LOG (A (I)), I = 1, N + 9, K), G
```

2 **9.5.3 Execution of a data transfer input/output statement**

3 Execution of a WRITE or PRINT statement for a file that does not exist creates the file unless an error  
4 condition occurs.

5 The effect of executing a synchronous data transfer input/output statement shall be as if the following  
6 operations were performed in the order specified:

- 7 (1) Determine the direction of data transfer.
- 8 (2) Identify the unit.
- 9 (3) Perform a wait operation for all pending input/output operations for the unit. If an error,  
10 end-of-file, or end-of-record condition occurs during any of the wait operations, steps 4  
11 through 8 are skipped for the current data transfer statement.
- 12 (4) Establish the format if one is specified.
- 13 (5) Position the file prior to data transfer (9.2.3.2) unless the statement is a child data transfer  
14 statement (9.5.3.7).
- 15 (6) Transfer data between the file and the entities specified by the input/output list (if any) or  
16 namelist.
- 17 (7) Determine whether an error, end-of-file, or end-of-record condition has occurred.
- 18 (8) Position the file after data transfer (9.2.3.3) unless the statement is a child data transfer  
19 statement (9.5.3.7).
- 20 (9) Cause any variable specified in a SIZE= specifier to become defined.
- 21 (10) If an error, end-of-file, or end-of-record condition occurred, processing continues as specified  
22 in 9.10; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.

23 The effect of executing an asynchronous data transfer input/output statement shall be as if the following  
24 operations were performed in the order specified:

- 25 (1) Determine the direction of data transfer.
  - 26 (2) Identify the unit.
  - 27 (3) Establish the format if one is specified.
  - 28 (4) Position the file prior to data transfer (9.2.3.2).
  - 29 (5) Establish the set of storage units identified by the input/output list. For a READ statement,  
30 this might require some or all of the data in the file to be read if an input variable is used  
31 as a *scalar-int-expr* in an *io-implied-do-control* in the input/output list, as a *subscript*,  
32 *substring-range*, *stride*, or is otherwise referenced.
  - 33 (6) Initiate an asynchronous data transfer between the file and the entities specified by the  
34 input/output list (if any) or namelist. The asynchronous data transfer may complete (and  
35 an error, end-of-file, or end-of-record condition may occur) during the execution of this data  
36 transfer statement or during a later wait operation.
  - 37 (7) Determine whether an error, end-of-file, or end-of-record condition has occurred. The con-  
38 ditions may occur during the execution of this data transfer statement or during the corre-  
39 sponding wait operation, but not both.
- 40 Also, any of these conditions that would have occurred during the corresponding wait oper-  
41 ation for a previously pending data transfer operation that does not have an ID= specifier

- 1           may occur during the execution of this data transfer statement.
- 2       (8)   Position the file as if the data transfer had finished (9.2.3.3).
- 3       (9)   Cause any variable specified in a SIZE= specifier to become undefined.
- 4       (10)  If an error, end-of-file, or end-of-record condition occurred, processing continues as specified
- 5           in 9.10; otherwise any variable specified in an IOSTAT= specifier is assigned the value zero.
- 6   For an asynchronous data transfer statement, the data transfers may occur during execution of the
- 7   statement, during execution of the corresponding wait operation, or anywhere between. The data transfer
- 8   operation is considered to be pending until a corresponding wait operation is performed.
- 9   For asynchronous output, a pending input/output storage sequence affector (9.5.1.4) shall not be rede-
- 10   fined, become undefined, or have its pointer association status changed.
- 11   For asynchronous input, a pending input/output storage sequence affector shall not be referenced, be-
- 12   come defined, become undefined, become associated with a dummy argument that has the VALUE
- 13   attribute, or have its pointer association status changed.
- 14   Error, end-of-file, and end-of-record conditions in an asynchronous data transfer operation may occur
- 15   during execution of either the data transfer statement or the corresponding wait operation. If an ID=
- 16   specifier does not appear in the initiating data transfer statement, the conditions may occur during the
- 17   execution of any subsequent data transfer or wait operation for the same unit. When a condition occurs
- 18   for a previously executed asynchronous data transfer statement, a wait operation is performed for all
- 19   pending data transfer operations on that unit. When a condition occurs during a subsequent statement,
- 20   any actions specified by IOSTAT=, IOMSG=, ERR=, END=, and EOR= specifiers for that statement
- 21   are taken.

**NOTE 9.38**

Because end-of-file and error conditions for asynchronous data transfer statements without an ID= specifier may be reported by the processor during the execution of a subsequent data transfer statement, it may be impossible for the user to determine which input/output statement caused the condition. Reliably detecting which READ statement caused an end-of-file condition requires that all asynchronous READ statements for the unit include an ID= specifier.

**22 9.5.3.1 Direction of data transfer**

23   Execution of a READ statement causes values to be transferred from a file to the entities specified by

24   the input list, if any, or specified within the file itself for namelist input. Execution of a WRITE or

25   PRINT statement causes values to be transferred to a file from the entities specified by the output list

26   and format specification, if any, or by the *namelist-group-name* for namelist output.

**27 9.5.3.2 Identifying a unit**

28   A data transfer input/output statement that contains an input/output control list includes a UNIT=

29   specifier that identifies an external or internal unit. A READ statement that does not contain an

30   input/output control list specifies a particular processor-dependent unit, which is the same as the unit

31   identified by \* in a READ statement that contains an input/output control list. The PRINT statement

32   specifies some other processor-dependent unit, which is the same as the unit identified by \* in a WRITE

33   statement. Thus, each data transfer input/output statement identifies an external or internal unit.

34   The unit identified by a data transfer input/output statement shall be connected to a file when execution

35   of the statement begins.

**NOTE 9.39**

The file may be preconnected.

**1 9.5.3.3 Establishing a format**

2 If the input/output control list contains \* as a format, list-directed formatting is established. If *namelist-*  
3 *group-name* appears, namelist formatting is established. If no *format* or *namelist-group-name* is spec-  
4 ified, unformatted data transfer is established. Otherwise, the format specification identified by the  
5 FMT= specifier is established.

6 On output, if an internal file has been specified, a format specification that is in the file or is associated  
7 with the file shall not be specified.

**8 9.5.3.4 Data transfer**

9 Data are transferred between the file and the entities specified by the input/output list or namelist.  
10 The list items are processed in the order of the input/output list for all data transfer input/output  
11 statements except namelist formatted data transfer statements. The list items for a namelist input  
12 statement are processed in the order of the entities specified within the input records. The list items  
13 for a namelist output statement are processed in the order in which the variables are specified in the  
14 *namelist-group-object-list*. Effective items are derived from the input/output list items as described in  
15 9.5.2.

16 All values needed to determine which entities are specified by an input/output list item are determined  
17 at the beginning of the processing of that item.

18 All values are transmitted to or from the entities specified by a list item prior to the processing of any  
19 succeeding list item for all data transfer input/output statements.

**NOTE 9.40**

In the example,

```
READ (N) N, X (N)
```

the old value of N identifies the unit, but the new value of N is the subscript of X.

20 All values following the *name=* part of the namelist entity (10.10) within the input records are transmit-  
21 ted to the matching entity specified in the *namelist-group-object-list* prior to processing any succeeding  
22 entity within the input record for namelist input statements. If an entity is specified more than once  
23 within the input record during a namelist formatted data transfer input statement, the last occurrence  
24 of the entity specifies the value or values to be used for that entity.

25 An input list item, or an entity associated with it, shall not contain any portion of an established format  
26 specification.

27 If the input/output item is a pointer, data are transferred between the file and the associated target.

28 If an internal file has been specified, an input/output list item shall not be in the file or associated with  
29 the file.

**NOTE 9.41**

The file is a data object.

30 A DO variable becomes defined and its iteration count established at the beginning of processing of the

1 items that constitute the range of an *io-implied-do*.

2 On output, every entity whose value is to be transferred shall be defined.

### 3 **9.5.3.4.1 Unformatted data transfer**

4 During unformatted data transfer, data are transferred without editing between the file and the entities  
5 specified by the input/output list. If the file is connected for sequential or direct access, exactly one  
6 record is read or written.

7 Objects of intrinsic or derived types may be transferred by means of an unformatted data transfer  
8 statement.

9 A value in the file is stored in a contiguous sequence of file storage units, beginning with the file storage  
10 unit immediately following the current file position.

11 After each value is transferred, the current file position is moved to a point immediately after the last  
12 file storage unit of the value.

13 On input from a file connected for sequential or direct access, the number of file storage units required  
14 by the input list shall be less than or equal to the number of file storage units in the record.

15 On input, if the file storage units transferred do not contain a value with the same type and type  
16 parameters as the input list entity, then the resulting value of the entity is processor-dependent except  
17 in the following cases:

18 (1) A complex list entity may correspond to two real values of the same kind stored in the file,  
19 or vice-versa.

20 (2) A default character list entity of length  $n$  may correspond to  $n$  default characters stored in  
21 the file, regardless of the length parameters of the entities that were written to these storage  
22 units of the file. If the file is connected for stream input, the characters may have been  
23 written by formatted stream output.

24 On output to a file connected for unformatted direct access, the output list shall not specify more values  
25 than can fit into the record. If the file is connected for direct access and the values specified by the  
26 output list do not fill the record, the remainder of the record is undefined.

27 If the file is connected for unformatted sequential access, the record is created with a length sufficient  
28 to hold the values from the output list. This length shall be one of the set of allowed record lengths for  
29 the file and shall not exceed the value specified in the RECL= specifier, if any, of the OPEN statement  
30 that established the connection.

31 If the file is not connected for unformatted input/output, unformatted data transfer is prohibited.

32 The unit specified shall be an external unit.

### 33 **9.5.3.4.2 Formatted data transfer**

34 During formatted data transfer, data are transferred with editing between the file and the entities  
35 specified by the input/output list or by the *namelist-group-name*. Format control is initiated and editing  
36 is performed as described in Section 10.

37 The current record and possibly additional records are read or written.

38 Values may be transferred by means of a formatted data transfer statement to or from objects of intrinsic  
39 or derived types. In the latter case, the transfer is in the form of values of intrinsic types to or from the  
40 components of intrinsic types that ultimately comprise these structured objects unless the derived-type  
41 list item is processed by a user-defined derived-type input/output procedure (9.5.3.7).

- 1 If the file is not connected for formatted input/output, formatted data transfer is prohibited.
- 2 During advancing input when the pad mode has the value NO, the input list and format specification
- 3 shall not require more characters from the record than the record contains.
- 4 During advancing input when the pad mode has the value YES, blank characters are supplied by the
- 5 processor if the input list and format specification require more characters from the record than the
- 6 record contains.
- 7 During nonadvancing input when the pad mode has the value NO, an end-of-record condition (9.10)
- 8 occurs if the input list and format specification require more characters from the record than the record
- 9 contains, and the record is complete (9.2.2.3). If the record is incomplete, an end-of-file condition occurs
- 10 instead of an end-of-record condition.
- 11 During nonadvancing input when the pad mode has the value YES, blank characters are supplied by the
- 12 processor if an input item and its corresponding data edit descriptor require more characters from the
- 13 record than the record contains. If the record is incomplete, an end-of-file condition occurs; otherwise
- 14 an end-of-record condition occurs.
- 15 If the file is connected for direct access, the record number is increased by one as each succeeding record
- 16 is read or written.
- 17 On output, if the file is connected for direct access or is an internal file and the characters specified by
- 18 the output list and format do not fill a record, blank characters are added to fill the record.
- 19 On output, the output list and format specification shall not specify more characters for a record than
- 20 have been specified by a RECL= specifier in the OPEN statement or the record length of an internal
- 21 file.

#### 22 **9.5.3.5 List-directed formatting**

- 23 If list-directed formatting has been established, editing is performed as described in 10.9.

#### 24 **9.5.3.6 Namelist formatting**

- 25 If namelist formatting has been established, editing is performed as described in 10.10.

26 Every allocatable *namelist-group-object* in the namelist group shall be allocated and every *namelist-*  
27 *group-object* that is a pointer shall be associated with a target. If a namelist-group-object is polymorphic  
28 or has an ultimate component that is allocatable or a pointer, that object shall be processed by a user-  
29 defined derived-type input/output procedure (9.5.3.7).

#### 30 **9.5.3.7 User-defined derived-type input/output**

- 31 User-defined derived-type input/output procedures allow a program to override the default handling of
- 32 derived-type objects and values in data transfer input/output statements described in 9.5.2.

33 A user-defined derived-type input/output procedure is a procedure accessible by a *dtio-generic-spec*  
34 (12.3.2.1). A particular user-defined derived-type input/output procedure is selected as described in  
35 9.5.3.7.3.

#### 36 **9.5.3.7.1 Executing user-defined derived-type input/output data transfers**

- 37 If a derived-type input/output procedure is selected as specified in 9.5.3.7.3, the processor shall call the se-
- 38 lected user-defined derived-type input/output procedure for any appropriate data transfer input/output
- 39 statements executed in that scoping unit. The user-defined derived-type input/output procedure controls
- 40 the actual data transfer operations for the derived-type list item.

1 A data transfer statement that includes a derived-type list item and that causes a user-defined derived-  
 2 type input/output procedure to be invoked is called a **parent data transfer statement**. A data  
 3 transfer statement that is executed while a parent data transfer statement is being processed and that  
 4 specifies the unit passed into a user-defined derived-type input/output procedure is called a **child data**  
 5 **transfer statement**.

**NOTE 9.42**

A user-defined derived-type input/output procedure will usually contain child data transfer statements that read values from or write values to the current record or at the current file position. The effect of executing the user-defined derived-type input/output procedure is similar to that of substituting the list items from any child data transfer statements into the parent data transfer statement's list items, along with similar substitutions in the format specification.

**NOTE 9.43**

A particular execution of a READ, WRITE or PRINT statement can be both a parent and a child data transfer statement. A user-defined derived-type input/output procedure can indirectly call itself or another user-defined derived-type input/output procedure by executing a child data transfer statement containing a list item of derived type, where a matching interface is accessible for that derived type. If a user-defined derived-type input/output procedure calls itself indirectly in this manner, it shall be declared RECURSIVE.

6 A child data transfer statement is processed differently from a nonchild data transfer statement in the  
 7 following ways:

- 8 • Executing a child data transfer statement does not position the file prior to data transfer.
- 9 • An unformatted child data transfer statement does not position the file after data transfer is  
 10 complete.

**9.5.3.7.2 User-defined derived-type input/output procedures**

12 For a particular derived type and a particular set of kind type parameter values, there are four possible  
 13 sets of characteristics for user-defined derived-type input/output procedures; one each for formatted  
 14 input, formatted output, unformatted input, and unformatted output. The user need not supply all four  
 15 procedures. The procedures are specified to be used for derived-type input/output by interface blocks  
 16 (12.3.2.1) or by generic bindings (4.5.4), with a *dtio-generic-spec* (R1208).

17 In the four interfaces, which specify the characteristics of user-defined procedures for derived-type in-  
 18 put/output, the following syntax term is used:

```
19 R920 dtv-type-spec           is TYPE( derived-type-spec )
20                                or CLASS( derived-type-spec )
```

21 C936 (R920) If *derived-type-spec* specifies an extensible type, the CLASS keyword shall be used;  
 22 otherwise, the TYPE keyword shall be used.

23 C937 (R920) All length type parameters of *derived-type-spec* shall be assumed.

24 If the *dtio-generic-spec* is READ (FORMATTED), the characteristics shall be the same as those specified  
 25 by the following interface:

```
26           SUBROUTINE my_read_routine_formatted           &
27           (dtv,                                         &
```

```

1          unit,          &
2          iotype, v_list, &
3          iostat, iomsg)
4      ! the derived-type value/variable
5      dtv-type-spec, INTENT(INOUT) :: dtv
6      INTEGER, INTENT(IN) :: unit ! unit number
7      ! the edit descriptor string
8      CHARACTER (LEN=*), INTENT(IN) :: iotype
9      INTEGER, INTENT(IN) :: v_list(:)
10     INTEGER, INTENT(OUT) :: iostat
11     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
12     END

```

13 If the *dtio-generic-spec* is READ (UNFORMATTED), the characteristics shall be the same as those  
14 specified by the following interface:

```

15     SUBROUTINE my_read_routine_unformatted      &
16         (dtv,                                  &
17         unit,                                  &
18         iostat, iomsg)
19     ! the derived-type value/variable
20     dtv-type-spec, INTENT(INOUT) :: dtv
21     INTEGER, INTENT(IN) :: unit
22     INTEGER, INTENT(OUT) :: iostat
23     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
24     END

```

25 If the *dtio-generic-spec* is WRITE (FORMATTED), the characteristics shall be the same as those spec-  
26 ified by the following interface:

```

27     SUBROUTINE my_write_routine_formatted      &
28         (dtv,                                  &
29         unit,                                  &
30         iotype, v_list,                        &
31         iostat, iomsg)
32     ! the derived-type value/variable
33     dtv-type-spec, INTENT(IN) :: dtv
34     INTEGER, INTENT(IN) :: unit
35     ! the edit descriptor string
36     CHARACTER (LEN=*), INTENT(IN) :: iotype
37     INTEGER, INTENT(IN) :: v_list(:)
38     INTEGER, INTENT(OUT) :: iostat
39     CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
40     END

```

41 If the *dtio-generic-spec* is WRITE (UNFORMATTED), the characteristics shall be the same as those  
42 specified by the following interface:

```

43     SUBROUTINE my_write_routine_unformatted    &
44         (dtv,                                  &
45         unit,                                  &

```

```

1           iostat, iomsg)
2       ! the derived-type value/variable
3       dtv-type-spec, INTENT(IN) :: dtv
4       INTEGER, INTENT(IN) :: unit
5       INTEGER, INTENT(OUT) :: iostat
6       CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
7       END

```

8 The actual specific procedure names (the `my...routine...` procedure names above) are not significant. In the discussion here and elsewhere, the dummy arguments in these interfaces are referred by the names given above; the names are, however, arbitrary.

11 In addition to the characteristics specified by the above interfaces, the `dtv` dummy argument may optionally have the VOLATILE attribute.

13 When a user-defined derived-type input/output procedure is invoked, the processor shall pass a `unit` argument that has a value as follows:

- 15 • If the parent data transfer statement uses a *file-unit-number*, the value of the `unit` argument shall be that of the *file-unit-number*.
- 17 • If the parent data transfer statement is a WRITE statement with an asterisk unit or a PRINT statement, the `unit` argument shall have the same value as the OUTPUT\_UNIT named constant of the ISO\_FORTRAN\_ENV intrinsic module (13.8.2).
- 19 • If the parent data transfer statement is a READ statement with an asterisk unit or a READ statement without an *io-control-spec-list*, the `unit` argument shall have the same value as the INPUT\_UNIT named constant of the ISO\_FORTRAN\_ENV intrinsic module (13.8.2).
- 21 • Otherwise the parent data transfer statement must access an internal file, in which case the `unit` argument shall have a processor-dependent negative value.

#### NOTE 9.44

Because the `unit` argument value will be negative when the parent data transfer statement specifies an internal file, a user-defined derived-type input/output procedure should not execute an INQUIRE statement without checking that the `unit` argument is nonnegative or is equal to one of the named constants INPUT\_UNIT, OUTPUT\_UNIT, or ERROR\_UNIT of the ISO\_FORTRAN\_ENV intrinsic module (13.8.2).

25 For formatted data transfer, the processor shall pass an `iotype` argument that has a value as follows:

- 26 • “LISTDIRECTED” if the parent data transfer statement specified list directed formatting,
- 27 • “NAMELIST” if the parent data transfer statement specified namelist formatting, or
- 28 • “DT” concatenated with the *char-literal-constant*, if any, of the edit descriptor, if the parent data transfer statement contained a format specification and the list item’s corresponding edit descriptor was a DT edit descriptor.

31 If the parent data transfer statement is a READ statement, the `dtv` dummy argument is argument associated with the effective list item that caused the user-defined derived-type input procedure to be invoked, as if the effective list item were an actual argument in this procedure reference (2.5.6).

- 1 If the parent data transfer statement is a WRITE or PRINT statement, the processor shall provide the  
2 value of the effective list item in the `dtv` dummy argument.
- 3 If the *v-list* of the edit descriptor appears in the parent data transfer statement, the processor shall  
4 provide the values from it in the `v_list` dummy argument, with the same number of elements in the  
5 same order as *v-list*. If there is no *v-list* in the edit descriptor or if the data transfer statement specifies  
6 list-directed or namelist formatting, the processor shall provide `v_list` as a zero-sized array.

**NOTE 9.45**

The user's procedure may choose to interpret an element of the `v_list` argument as a field width, but this is not required. If it does, it would be appropriate to fill an output field with "\*"s if the width is too small.

- 7 The `iostat` argument is used to report whether an error, end-of-record, or end-of-file condition (9.10)  
8 occurs. If an error condition occurs, the user-defined derived-type input/output procedure shall assign  
9 a positive value to the `iostat` argument. Otherwise, if an end-of-file condition occurs, the user-defined  
10 derived-type input procedure shall assign the value of the named constant `IOSTAT_END` (13.8.2.5) to the  
11 `iostat` argument. Otherwise, if an end-of-record condition occurs, the user-defined derived-type input  
12 procedure shall assign the value of the named constant `IOSTAT_EOR` (13.8.2.6) to `iostat`. Otherwise,  
13 the user-defined derived-type input/output procedure shall assign the value zero to the `iostat` argument.
- 14 If the user-defined derived-type input/output procedure returns a nonzero value for the `iostat` argument,  
15 the procedure shall also return an explanatory message in the `iomsg` argument. Otherwise, the procedure  
16 shall not change the value of the `iomsg` argument.

**NOTE 9.46**

The values of the `iostat` and `iomsg` arguments set in a user-defined derived-type input/output procedure need not be passed to all of the parent data transfer statements.

- 17 If the `iostat` argument of the user-defined derived-type input/output procedure has a nonzero value  
18 when that procedure returns, and the processor therefore terminates execution of the program as de-  
19 scribed in 9.10, the processor shall make the value of the `iomsg` argument available in a processor-  
20 dependent manner.
- 21 When a parent READ statement is active, an input/output statement shall not read from any external  
22 unit other than the one specified by the dummy argument `unit` and shall not perform output to any  
23 external unit.
- 24 When a parent WRITE or PRINT statement is active, an input/output statement shall not perform  
25 output to any external unit other than the one specified by the dummy argument `unit` and shall not  
26 read from any external unit.
- 27 When a parent data transfer statement is active, a data transfer statement that specifies an internal file  
28 is permitted.
- 29 OPEN, CLOSE, BACKSPACE, ENDFILE, and REWIND statements shall not be executed while a  
30 parent data transfer statement is active.
- 31 A user-defined derived-type input/output procedure may use a FORMAT with a DT edit descriptor for  
32 handling a component of the derived type that is itself of a derived type. A child data transfer statement  
33 that is a list directed or namelist input/output statement may contain a list item of derived type.
- 34 Because a child data transfer statement does not position the file prior to data transfer, the child data  
35 transfer statement starts transferring data from where the file was positioned by the parent data transfer  
36 statement's most recently processed effective list item or record positioning edit descriptor. This is not

- 1 necessarily at the beginning of a record.
- 2 A record positioning edit descriptor, such as TL and TR, used on `unit` by a child data transfer statement
- 3 shall not cause the record position to be positioned before the record position at the time the user-defined
- 4 derived-type input/output procedure was invoked.

**NOTE 9.47**

A robust user-defined derived-type input/output procedure may wish to use `INQUIRE` to determine the settings of `BLANK=`, `PAD=`, `ROUND=`, `DECIMAL=`, and `DELIM=` for an external unit. The `INQUIRE` provides values as specified in 9.9.

- 5 Neither a parent nor child data transfer statement shall be asynchronous.
- 6 A user-defined derived-type input/output procedure, and any procedures invoked therefrom, shall not
- 7 define, nor cause to become undefined, any storage location referenced by any input/output list item,
- 8 the corresponding format, or any specifier in any active parent data transfer statement, except through
- 9 the `dtv` argument.

**NOTE 9.48**

A child data transfer statement shall not specify the `ID=`, `POS=`, or `REC=` specifiers in an input/output control list.

**NOTE 9.49**

A simple example of derived type formatted output follows. The derived type variable `chairman` has two components. The type and an associated write formatted procedure are defined in a module so as to be accessible from wherever they might be needed. It would also be possible to check that `iotype` indeed has the value 'DT' and to set `iostat` and `iomsg` accordingly.

```

MODULE p

  TYPE :: person
    CHARACTER (LEN=20) :: name
    INTEGER :: age
  CONTAINS
    GENERIC :: WRITE(FORMATTED) => pwf
  END TYPE person

CONTAINS

  SUBROUTINE pwf (dtv,unit,iotype,vlist,iostat,iomsg)
! argument definitions
    CLASS(person), INTENT(IN) :: dtv
    INTEGER, INTENT(IN) :: unit
    CHARACTER (LEN=*), INTENT(IN) :: iotype
    INTEGER, INTENT(IN) :: vlist(:)
    INTEGER, INTENT(OUT) :: iostat
    CHARACTER (LEN=*), INTENT(INOUT) :: iomsg
! local variable
    CHARACTER (LEN=9) :: pfmt

!   vlist(1) and (2) are to be used as the field widths of the two
!   components of the derived type variable. First set up the format to
!   be used for output.

```

**NOTE 9.49 (cont.)**

```

WRITE(pfmt,'(A,I2,A,I2,A)' ) '(A', vlist(1), ',I', vlist(2), ' )'

! now the basic output statement
WRITE(unit, FMT=pfmt, IOSTAT=iostat) dtv%name, dtv%age

END SUBROUTINE pwf

END MODULE p

PROGRAM
  USE p
  INTEGER id, members
  TYPE (person) :: chairman
  ...
  WRITE(6, FMT="(I2, DT (15,6), I5)" ) id, chairman, members
! this writes a record with four fields, with lengths 2, 15, 6, 5
! respectively

END PROGRAM

```

**NOTE 9.50**

In the following example, the variables of the derived type `node` form a linked list, with a single value at each node. The subroutine `pwf` is used to write the values in the list, one per line.

```

MODULE p

  TYPE node
    INTEGER :: value = 0
    TYPE (NODE), POINTER :: next_node => NULL ( )
  CONTAINS
    GENERIC :: WRITE(FORMATTED) => pwf
  END TYPE node

CONTAINS

  RECURSIVE SUBROUTINE pwf (dtv,unit,iotype,vlist,iostat,iomsg)
! Write the chain of values, each on a separate line in I9 format.
    CLASS(node), INTENT(IN) :: dtv
    INTEGER, INTENT(IN) :: unit
    CHARACTER (LEN=*), INTENT(IN) :: iotype
    INTEGER, INTENT(IN) :: vlist(:)
    INTEGER, INTENT(OUT) :: iostat
    CHARACTER (LEN=*), INTENT(INOUT) :: iomsg

    WRITE(unit,'(i9 /)', IOSTAT = iostat) dtv%value
    IF(iostat/=0) RETURN
    IF(ASSOCIATED(dtv%next_node)) WRITE(unit,'(dt)', IOSTAT=iostat) dtv%next_node
  END SUBROUTINE pwf

END MODULE p

```

### 1 9.5.3.7.3 Resolving derived-type input/output procedure references

2 A suitable generic interface for user-defined derived-type input/output of an effective item is one that  
3 has a *dtio-generic-spec* that is appropriate to the direction (read or write) and form (formatted or  
4 unformatted) of the data transfer as specified in 9.5.3.7, and has a specific interface whose *dtv* argument  
5 is compatible with the effective item according to the rules for argument association in 12.4.1.2.

6 When an effective item (9.5.2) that is of derived-type is encountered during a data transfer, user-defined  
7 derived-type input/output occurs if both of the following conditions are true:

- 8 (1) The circumstances of the input/output are such that user-defined derived-type input/output  
9 is permitted; that is, either
  - 10 (a) the transfer was initiated by a list-directed, namelist, or unformatted input/output  
11 statement, or
  - 12 (b) a format specification is supplied for the input/output statement, and the edit de-  
13 scriptor corresponding to the effective item is a DT edit descriptor.
- 14 (2) A suitable user-defined derived-type input/output procedure is available; that is, either
  - 15 (a) the declared type of the effective item has a suitable generic type-bound procedure,  
16 or
  - 17 (b) a suitable generic interface is accessible.

18 If (2a) is true, the procedure referenced is determined as for explicit type-bound procedure references  
19 (12.4); that is, the binding with the appropriate specific interface is located in the declared type of the  
20 effective item, and the corresponding binding in the dynamic type of the effective item is selected.

21 If (2a) is false and (2b) is true, the reference is to the procedure identified by the appropriate specific  
22 interface in the interface block. This reference shall not be to a dummy procedure that is not present,  
23 or to a disassociated procedure pointer.

### 24 9.5.4 Termination of data transfer statements

25 Termination of an input/output data transfer statement occurs when any of the following conditions are  
26 met:

- 27 (1) Format processing encounters a data edit descriptor and there are no remaining elements  
28 in the *input-item-list* or *output-item-list*.
- 29 (2) Unformatted or list-directed data transfer exhausts the *input-item-list* or *output-item-list*.
- 30 (3) Namelist output exhausts the *namelist-group-object-list*.
- 31 (4) An error condition occurs.
- 32 (5) An end-of-file condition occurs.
- 33 (6) A slash (/) is encountered as a value separator (10.9, 10.10) in the record being read during  
34 list-directed or namelist input.
- 35 (7) An end-of-record condition occurs during execution of a nonadvancing input statement  
36 (9.10).

### 37 9.6 Waiting on pending data transfer

38 Execution of an asynchronous data transfer statement in which neither an error, end-of-record, nor end-  
39 of-file condition occurs initiates a pending data transfer operation. There may be multiple pending data  
40 transfer operations for the same or multiple units simultaneously. A pending data transfer operation  
41 remains pending until a corresponding wait operation is performed. A wait operation may be performed  
42 by a WAIT, INQUIRE, CLOSE, or file positioning statement.

### 1 9.6.1 WAIT statement

2 A WAIT statement performs a wait operation for specified pending asynchronous data transfer opera-  
3 tions.

#### NOTE 9.51

The CLOSE, INQUIRE, and file positioning statements may also perform wait operations.

- |    |                       |    |                                     |
|----|-----------------------|----|-------------------------------------|
| 4  | R921 <i>wait-stmt</i> | is | WAIT ( <i>wait-spec-list</i> )      |
| 5  | R922 <i>wait-spec</i> | is | [ UNIT = ] <i>file-unit-number</i>  |
| 6  |                       | or | END = <i>label</i>                  |
| 7  |                       | or | EOR = <i>label</i>                  |
| 8  |                       | or | ERR = <i>label</i>                  |
| 9  |                       | or | ID = <i>scalar-int-variable</i>     |
| 10 |                       | or | IOMSG = <i>iomsg-variable</i>       |
| 11 |                       | or | IOSTAT = <i>scalar-int-variable</i> |
- 12 C938 (R922) No specifier shall appear more than once in a given *wait-spec-list*.
- 13 C939 (R922) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the  
14 *file-unit-number* shall be the first item in the *wait-spec-list*.
- 15 C940 (R922) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a  
16 branch target statement that appears in the same scoping unit as the WAIT statement.
- 17 The IOSTAT=, ERR=, EOR=, END=, and IOMSG= specifiers are described in 9.10.
- 18 The value of the variable specified in the ID= specifier shall be the identifier of a pending data transfer  
19 operation for the specified unit. If the ID= specifier appears, a wait operation for the specified data  
20 transfer operation is performed. If the ID= specifier is omitted, wait operations for all pending data  
21 transfers for the specified unit are performed.
- 22 Execution of a WAIT statement specifying a unit that does not exist, has no file connected to it, or was  
23 not opened for asynchronous input/output is permitted, provided that the WAIT statement has no ID=  
24 specifier; such a WAIT statement does not cause an error or end-of-file condition to occur.

#### NOTE 9.52

An EOR= specifier has no effect if the pending data transfer operation is not a nonadvancing read.  
And END= specifier has no effect if the pending data transfer operation is not a READ.

### 25 9.6.2 Wait operation

26 A wait operation completes the processing of a pending data transfer operation. Each wait operation  
27 completes only a single data transfer operation, although a single statement may perform multiple wait  
28 operations.

29 If the actual data transfer is not yet complete, the wait operation first waits for its completion. If the  
30 data transfer operation is an input operation that completed without error, the storage units of the  
31 input/output storage sequence then become defined with the values as described in 9.5.1.14 and 9.5.3.4.

32 If any error, end-of-file, or end-of-record conditions occur, the applicable actions specified by the IO-  
33 STAT=, IOMSG=, ERR=, END=, and EOR= specifiers of the statement that performs the wait oper-  
34 ation are taken.

35 If an error or end-of-file condition occurs during a wait operation for a unit, the processor performs a

- 1 wait operation for all pending data transfer operations for that unit.

**NOTE 9.53**

Error, end-of-file, and end-of-record conditions may be raised either during the data transfer statement that initiates asynchronous input/output, a subsequent asynchronous data transfer statement for the same unit, or during the wait operation. If such conditions are raised during a data transfer statement, they trigger actions according to the IOSTAT=, ERR=, END=, and EOR= specifiers of that statement; if they are raised during the wait operation, the actions are in accordance with the specifiers of the statement that performs the wait operation.

- 2 After completion of the wait operation, the data transfer operation and its input/output storage sequence  
3 are no longer considered to be pending.

## 4 **9.7 File positioning statements**

- |    |      |                       |   |
|----|------|-----------------------|---|
| 5  | R923 | <i>backspace-stmt</i> | <b>is</b> BACKSPACE <i>file-unit-number</i>       |
| 6  |      |                       | <b>or</b> BACKSPACE ( <i>position-spec-list</i> ) |
| 7  | R924 | <i>endfile-stmt</i>   | <b>is</b> ENDFILE <i>file-unit-number</i>         |
| 8  |      |                       | <b>or</b> ENDFILE ( <i>position-spec-list</i> )   |
| 9  | R925 | <i>rewind-stmt</i>    | <b>is</b> REWIND <i>file-unit-number</i>          |
| 10 |      |                       | <b>or</b> REWIND ( <i>position-spec-list</i> )    |

- 11 A file that is connected for direct access shall not be referred to by a BACKSPACE, ENDFILE, or  
12 REWIND statement. A file that is connected for unformatted stream access shall not be referred to by a  
13 BACKSPACE statement. A file that is connected with an ACTION= specifier having the value READ  
14 shall not be referred to by an ENDFILE statement.

- |    |      |                      |   |
|----|------|----------------------|---|
| 15 | R926 | <i>position-spec</i> | <b>is</b> [ UNIT = ] <i>file-unit-number</i>  |
| 16 |      |                      | <b>or</b> IOMSG = <i>iormsg-variable</i>      |
| 17 |      |                      | <b>or</b> IOSTAT = <i>scalar-int-variable</i> |
| 18 |      |                      | <b>or</b> ERR = <i>label</i>                  |

- 19 C941 (R926) No specifier shall appear more than once in a given *position-spec-list*.

- 20 C942 (R926) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the  
21 *file-unit-number* shall be the first item in the *position-spec-list*.

- 22 C943 (R926) The *label* in the ERR= specifier shall be the statement label of a branch target statement  
23 that appears in the same scoping unit as the file positioning statement.

- 24 The IOSTAT=, ERR=, and IOMSG= specifiers are described in 9.10.

- 25 Execution of a file positioning statement performs a wait operation for all pending asynchronous data  
26 transfer operations for the specified unit.

### 27 **9.7.1 BACKSPACE statement**

- 28 Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned  
29 before the current record if there is a current record, or before the preceding record if there is no current  
30 record. If the file is at its initial point, the position of the file is not changed.

**NOTE 9.54**

If the preceding record is an endfile record, the file is positioned before the endfile record.

- 1 If a BACKSPACE statement causes the implicit writing of an endfile record, the file is positioned before
- 2 the record that precedes the endfile record.
- 3 Backspacing a file that is connected but does not exist is prohibited.
- 4 Backspacing over records written using list-directed or namelist formatting is prohibited.

**NOTE 9.55**

An example of a BACKSPACE statement is:

```
BACKSPACE (10, IOSTAT = N)
```

**5 9.7.2 ENDFILE statement**

- 6 Execution of an ENDFILE statement for a file connected for sequential access writes an endfile record
- 7 as the next record of the file. The file is then positioned after the endfile record, which becomes the last
- 8 record of the file. If the file also may be connected for direct access, only those records before the endfile
- 9 record are considered to have been written. Thus, only those records may be read during subsequent
- 10 direct access connections to the file.
- 11 After execution of an ENDFILE statement for a file connected for sequential access, a BACKSPACE
- 12 or REWIND statement shall be used to reposition the file prior to execution of any data transfer
- 13 input/output statement or ENDFILE statement.
- 14 Execution of an ENDFILE statement for a file connected for stream access causes the terminal point of
- 15 the file to become equal to the current file position. Only file storage units before the current position are
- 16 considered to have been written; thus only those file storage units may be subsequently read. Subsequent
- 17 stream output statements may be used to write further data to the file.
- 18 Execution of an ENDFILE statement for a file that is connected but does not exist creates the file; if
- 19 the file is connected for sequential access, it is created prior to writing the endfile record.

**NOTE 9.56**

An example of an ENDFILE statement is:

```
ENDFILE K
```

**20 9.7.3 REWIND statement**

- 21 Execution of a REWIND statement causes the specified file to be positioned at its initial point.

**NOTE 9.57**

If the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

- 22 Execution of a REWIND statement for a file that is connected but does not exist is permitted and has
- 23 no effect on any file.

**NOTE 9.58**

An example of a REWIND statement is:

```
REWIND 10
```

## 1 9.8 FLUSH statement

2 The form of the FLUSH statement is:

3	R927	<i>flush-stmt</i>	is	FLUSH <i>file-unit-number</i>
4			or	FLUSH ( <i>flush-spec-list</i> )
5	R928	<i>flush-spec</i>	is	[UNIT =] <i>file-unit-number</i>
6			or	IOSTAT = <i>scalar-int-variable</i>
7			or	IOMSG = <i>iomsg-variable</i>
8			or	ERR = <i>label</i>

9 C944 (R928) No specifier shall appear more than once in a given *flush-spec-list*.

10 C945 (R928) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted from  
11 the unit specifier, the *file-unit-number* shall be the first item in the *flush-spec-list*.

12 C946 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement  
13 that appears in the same scoping unit as the flush statement.

14 The IOSTAT=, IOMSG= and ERR= specifiers are described in 9.10. The IOSTAT= variable shall  
15 be set to a processor-dependent positive value if an error occurs, to zero if the processor-dependent  
16 flush operation was successful, or to a processor-dependent negative value if the flush operation is not  
17 supported for the unit specified.

18 Execution of a FLUSH statement causes data written to an external file to be available to other processes,  
19 or causes data placed in an external file by means other than Fortran to be available to a READ  
20 statement. The action is processor dependent.

21 Execution of a FLUSH statement for a file that is connected but does not exist is permitted and has no  
22 effect on any file. A FLUSH statement has no effect on file position.

### NOTE 9.59

Because this standard does not specify the mechanism of file storage, the exact meaning of the flush operation is not precisely defined. The intention is that the flush operation should make all data written to a file available to other processes or devices, or make data recently added to a file by other processes or devices available to the program via a subsequent read operation. This is commonly called "flushing I/O buffers".

### NOTE 9.60

An example of a FLUSH statement is:

```
FLUSH( 10, IOSTAT=N)
```

## 23 9.9 File inquiry

24 The INQUIRE statement may be used to inquire about properties of a particular named file or of the  
25 connection to a particular unit. There are three forms of the INQUIRE statement: **inquire by file**,  
26 which uses the FILE= specifier, **inquire by unit**, which uses the UNIT= specifier, and **inquire by**  
27 **output list**, which uses only the IOLENGTH= specifier. All specifier value assignments are performed  
28 according to the rules for assignment statements.

29 An INQUIRE statement may be executed before, while, or after a file is connected to a unit. All values  
30 assigned by an INQUIRE statement are those that are current at the time the statement is executed.

1 R929 *inquire-stmt*                   is INQUIRE ( *inquire-spec-list* )  
 2                                       or INQUIRE ( IOLENGTH = *scalar-int-variable* ) ■  
 3                                       ■ *output-item-list*

**NOTE 9.61**

Examples of INQUIRE statements are:

```
INQUIRE (IOLENGTH = IOL) A (1:N)
INQUIRE (UNIT = JOAN, OPENED = LOG_01, NAMED = LOG_02, &
        FORM = CHAR_VAR, IOSTAT = IOS)
```

**4 9.9.1 Inquiry specifiers**

5 Unless constrained, the following inquiry specifiers may be used in either of the inquire by file or inquire  
 6 by unit forms of the INQUIRE statement:

7 R930 *inquire-spec*                   is [ UNIT = ] *file-unit-number*  
 8                                       or FILE = *file-name-expr*  
 9                                       or ACCESS = *scalar-default-char-variable*  
 10                                      or ACTION = *scalar-default-char-variable*  
 11                                      or ASYNCHRONOUS = *scalar-default-char-variable*  
 12                                      or BLANK = *scalar-default-char-variable*  
 13                                      or DECIMAL = *scalar-default-char-variable*  
 14                                      or DELIM = *scalar-default-char-variable*  
 15                                      or DIRECT = *scalar-default-char-variable*  
 16                                      or ENCODING = *scalar-default-char-variable*  
 17                                      or ERR = *label*  
 18                                      or EXIST = *scalar-default-logical-variable*  
 19                                      or FORM = *scalar-default-char-variable*  
 20                                      or FORMATTED = *scalar-default-char-variable*  
 21                                      or ID = *scalar-int-variable*  
 22                                      or IOMSG = *iomsg-variable*  
 23                                      or IOSTAT = *scalar-int-variable*  
 24                                      or NAME = *scalar-default-char-variable*  
 25                                      or NAMED = *scalar-default-logical-variable*  
 26                                      or NEXTREC = *scalar-int-variable*  
 27                                      or NUMBER = *scalar-int-variable*  
 28                                      or OPENED = *scalar-default-logical-variable*  
 29                                      or PAD = *scalar-default-char-variable*  
 30                                      or PENDING = *scalar-default-logical-variable*  
 31                                      or POS = *scalar-int-variable*  
 32                                      or POSITION = *scalar-default-char-variable*  
 33                                      or READ = *scalar-default-char-variable*  
 34                                      or READWRITE = *scalar-default-char-variable*  
 35                                      or RECL = *scalar-int-variable*  
 36                                      or ROUND = *scalar-default-char-variable*  
 37                                      or SEQUENTIAL = *scalar-default-char-variable*  
 38                                      or SIGN = *scalar-default-char-variable*  
 39                                      or SIZE = *scalar-int-variable*  
 40                                      or STREAM = *scalar-default-char-variable*  
 41                                      or UNFORMATTED = *scalar-default-char-variable*



1 **9.9.1.6 DECIMAL= specifier in the INQUIRE statement**

2 The *scalar-default-char-variable* in the DECIMAL= specifier is assigned the value COMMA or POINT,  
3 corresponding to the decimal edit mode in effect for a connection for formatted input/output. If there  
4 is no connection, or if the connection is not for formatted input/output, the *scalar-default-char-variable*  
5 is assigned the value UNDEFINED.

6 **9.9.1.7 DELIM= specifier in the INQUIRE statement**

7 The *scalar-default-char-variable* in the DELIM= specifier is assigned the value APOSTROPHE, QUOTE,  
8 or NONE, corresponding to the delimiter mode in effect for a connection for formatted input/output.  
9 If there is no connection or if the connection is not for formatted input/output, the *scalar-default-char-*  
10 *variable* is assigned the value UNDEFINED.

11 **9.9.1.8 DIRECT= specifier in the INQUIRE statement**

12 The *scalar-default-char-variable* in the DIRECT= specifier is assigned the value YES if DIRECT is  
13 included in the set of allowed access methods for the file, NO if DIRECT is not included in the set of  
14 allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or  
15 not DIRECT is included in the set of allowed access methods for the file.

16 **9.9.1.9 ENCODING= specifier in the INQUIRE statement**

17 The *scalar-default-char-variable* in the ENCODING= specifier is assigned the value UTF-8 if the file  
18 is connected for formatted input/output with an encoding form of UTF-8, and is assigned the value  
19 UNDEFINED if the file is connected for unformatted input/output. If there is no connection, it is  
20 assigned the value UTF-8 if the processor is able to determine that the encoding form of the file is  
21 UTF-8. If the processor is unable to determine the encoding form of the file, the variable is assigned the  
22 value UNKNOWN.

**NOTE 9.62**

The value assigned may be something other than UTF-8, UNDEFINED, or UNKNOWN if the processor supports other specific encoding forms (e.g. UTF-16BE).

23 **9.9.1.10 EXIST= specifier in the INQUIRE statement**

24 Execution of an INQUIRE by file statement causes the *scalar-default-logical-variable* in the EXIST=  
25 specifier to be assigned the value true if there exists a file with the specified name; otherwise, false is  
26 assigned. Execution of an INQUIRE by unit statement causes true to be assigned if the specified unit  
27 exists; otherwise, false is assigned.

28 **9.9.1.11 FORM= specifier in the INQUIRE statement**

29 The *scalar-default-char-variable* in the FORM= specifier is assigned the value FORMATTED if the  
30 file is connected for formatted input/output, and is assigned the value UNFORMATTED if the file is  
31 connected for unformatted input/output. If there is no connection, it is assigned the value UNDEFINED.

32 **9.9.1.12 FORMATTED= specifier in the INQUIRE statement**

33 The *scalar-default-char-variable* in the FORMATTED= specifier is assigned the value YES if FORMAT-  
34 TTED is included in the set of allowed forms for the file, NO if FORMATTED is not included in the set  
35 of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether or not  
36 FORMATTED is included in the set of allowed forms for the file.

1 **9.9.1.13 ID= specifier in the INQUIRE statement**

2 The value of the variable specified in the ID= specifier shall be the identifier of a pending data transfer  
3 operation for the specified unit. This specifier interacts with the PENDING= specifier (9.9.1.20).

4 **9.9.1.14 NAME= specifier in the INQUIRE statement**

5 The *scalar-default-char-variable* in the NAME= specifier is assigned the value of the name of the file if  
6 the file has a name; otherwise, it becomes undefined.

**NOTE 9.63**

If this specifier appears in an INQUIRE by file statement, its value is not necessarily the same as the name given in the FILE= specifier. However, the value returned shall be suitable for use as the value of the *file-name-expr* in the FILE= specifier in an OPEN statement.

The processor may return a file name qualified by a user identification, device, directory, or other relevant information.

7 The case of the characters assigned to *scalar-default-char-variable* is processor dependent.

8 **9.9.1.15 NAMED= specifier in the INQUIRE statement**

9 The *scalar-default-logical-variable* in the NAMED= specifier is assigned the value true if the file has a  
10 name; otherwise, it is assigned the value false.

11 **9.9.1.16 NEXTREC= specifier in the INQUIRE statement**

12 The *scalar-int-variable* in the NEXTREC= specifier is assigned the value  $n + 1$ , where  $n$  is the record  
13 number of the last record read from or written to the file connected for direct access. If the file is con-  
14 nected but no records have been read or written since the connection, the *scalar-int-variable* is assigned  
15 the value 1. If the file is not connected for direct access or if the position of the file is indeterminate  
16 because of a previous error condition, the *scalar-int-variable* becomes undefined. If there are pending  
17 data transfer operations for the specified unit, the value assigned is computed as if all the pending data  
18 transfers had already completed.

19 **9.9.1.17 NUMBER= specifier in the INQUIRE statement**

20 The *scalar-int-variable* in the NUMBER= specifier is assigned the value of the external unit number of  
21 the unit that is connected to the file. If there is no unit connected to the file, the value -1 is assigned.

22 **9.9.1.18 OPENED= specifier in the INQUIRE statement**

23 Execution of an INQUIRE by file statement causes the *scalar-default-logical-variable* in the OPENED=  
24 specifier to be assigned the value true if the file specified is connected to a unit; otherwise, false is  
25 assigned. Execution of an INQUIRE by unit statement causes the *scalar-default-logical-variable* to be  
26 assigned the value true if the specified unit is connected to a file; otherwise, false is assigned.

27 **9.9.1.19 PAD= specifier in the INQUIRE statement**

28 The *scalar-default-char-variable* in the PAD= specifier is assigned the value YES or NO, corresponding  
29 to the pad mode in effect for a connection for formatted input/output. If there is no connection or if  
30 the connection is not for formatted input/output, the *scalar-default-char-variable* is assigned the value  
31 UNDEFINED.

1 **9.9.1.20 PENDING= specifier in the INQUIRE statement**

2 The PENDING= specifier is used to determine whether or not previously pending asynchronous data  
3 transfers are complete. A data transfer operation is previously pending if it is pending at the beginning  
4 of execution of the INQUIRE statement.

5 If an ID= specifier appears and the specified data transfer operation is complete, then the variable  
6 specified in the PENDING= specifier is assigned the value false and the INQUIRE statement performs  
7 the wait operation for the specified data transfer.

8 If the ID= specifier is omitted and all previously pending data transfer operations for the specified unit  
9 are complete, then the variable specified in the PENDING= specifier is assigned the value false and the  
10 INQUIRE statement performs wait operations for all previously pending data transfers for the specified  
11 unit.

12 In all other cases, the variable specified in the PENDING= specifier is assigned the value true and no  
13 wait operations are performed; in this case the previously pending data transfers remain pending after  
14 the execution of the INQUIRE statement.

**NOTE 9.64**

The processor has considerable flexibility in defining when it considers a transfer to be complete.  
Any of the following approaches could be used:

- (1) The INQUIRE statement could consider an asynchronous data transfer to be incomplete until after the corresponding wait operation. In this case PENDING= would always return true unless there were no previously pending data transfers for the unit.
- (2) The INQUIRE statement could wait for all specified data transfers to complete and then always return false for PENDING=.
- (3) The INQUIRE statement could actually test the state of the specified data transfer operations.

15 **9.9.1.21 POS= specifier in the INQUIRE statement**

16 The *scalar-int-variable* in the POS= specifier is assigned the number of the file storage unit immediately  
17 following the current position of a file connected for stream access. If the file is positioned at its terminal  
18 position, the variable is assigned a value one greater than the number of the highest-numbered file storage  
19 unit in the file. If the file is not connected for stream access or if the position of the file is indeterminate  
20 because of previous error conditions, the variable becomes undefined.

21 **9.9.1.22 POSITION= specifier in the INQUIRE statement**

22 The *scalar-default-char-variable* in the POSITION= specifier is assigned the value REWIND if the file  
23 is connected by an OPEN statement for positioning at its initial point, APPEND if the file is connected  
24 for positioning before its endfile record or at its terminal point, and ASIS if the file is connected without  
25 changing its position. If there is no connection or if the file is connected for direct access, the *scalar-*  
26 *default-char-variable* is assigned the value UNDEFINED. If the file has been repositioned since the  
27 connection, the *scalar-default-char-variable* is assigned a processor-dependent value, which shall not be  
28 REWIND unless the file is positioned at its initial point and shall not be APPEND unless the file is  
29 positioned so that its endfile record is the next record or at its terminal point if it has no endfile record.

30 **9.9.1.23 READ= specifier in the INQUIRE statement**

31 The *scalar-default-char-variable* in the READ= specifier is assigned the value YES if READ is included  
32 in the set of allowed actions for the file, NO if READ is not included in the set of allowed actions for

1 the file, and UNKNOWN if the processor is unable to determine whether or not READ is included in  
2 the set of allowed actions for the file.

### 3 **9.9.1.24 READWRITE= specifier in the INQUIRE statement**

4 The *scalar-default-char-variable* in the READWRITE= specifier is assigned the value YES if READ-  
5 WRITE is included in the set of allowed actions for the file, NO if READWRITE is not included in the  
6 set of allowed actions for the file, and UNKNOWN if the processor is unable to determine whether or  
7 not READWRITE is included in the set of allowed actions for the file.

### 8 **9.9.1.25 RECL= specifier in the INQUIRE statement**

9 The *scalar-int-variable* in the RECL= specifier is assigned the value of the record length of a file con-  
10 nected for direct access, or the value of the maximum record length for a file connected for sequential  
11 access. If the file is connected for formatted input/output, the length is the number of characters for all  
12 records that contain only characters of type default character. If the file is connected for unformatted  
13 input/output, the length is measured in file storage units. If there is no connection, or if the connection  
14 is for stream access, the *scalar-int-variable* becomes undefined.

### 15 **9.9.1.26 ROUND= specifier in the INQUIRE statement**

16 The *scalar-default-char-variable* in the ROUND= specifier is assigned the value UP, DOWN, ZERO,  
17 NEAREST, COMPATIBLE, or PROCESSOR\_DEFINED, corresponding to the I/O rounding mode in  
18 effect for a connection for formatted input/output. If there is no connection or if the connection is not  
19 for formatted input/output, the *scalar-default-char-variable* is assigned the value UNDEFINED. The  
20 processor shall return the value PROCESSOR\_DEFINED only if the I/O rounding mode currently in  
21 effect behaves differently than the UP, DOWN, ZERO, NEAREST, and COMPATIBLE modes.

### 22 **9.9.1.27 SEQUENTIAL= specifier in the INQUIRE statement**

23 The *scalar-default-char-variable* in the SEQUENTIAL= specifier is assigned the value YES if SEQUEN-  
24 TIAL is included in the set of allowed access methods for the file, NO if SEQUENTIAL is not included  
25 in the set of allowed access methods for the file, and UNKNOWN if the processor is unable to determine  
26 whether or not SEQUENTIAL is included in the set of allowed access methods for the file.

### 27 **9.9.1.28 SIGN= specifier in the INQUIRE statement**

28 The *scalar-default-char-variable* in the SIGN= specifier is assigned the value PLUS, SUPPRESS, or  
29 PROCESSOR\_DEFINED, corresponding to the sign mode in effect for a connection for formatted in-  
30 put/output. If there is no connection, or if the connection is not for formatted input/output, the  
31 *scalar-default-char-variable* is assigned the value UNDEFINED.

### 32 **9.9.1.29 SIZE= specifier in the INQUIRE statement**

33 The *scalar-int-variable* in the SIZE= specifier is assigned the size of the file in file storage units. If the  
34 file size cannot be determined, the variable is assigned the value -1.

35 For a file that may be connected for stream access, the file size is the number of the highest-numbered  
36 file storage unit in the file.

37 For a file that may be connected for sequential or direct access, the file size may be different from the  
38 number of storage units implied by the data in the records; the exact relationship is processor-dependent.

### 1 9.9.1.30 STREAM= specifier in the INQUIRE statement

2 The *scalar-default-char-variable* in the STREAM= specifier is assigned the value YES if STREAM is  
3 included in the set of allowed access methods for the file, NO if STREAM is not included in the set of  
4 allowed access methods for the file, and UNKNOWN if the processor is unable to determine whether or  
5 not STREAM is included in the set of allowed access methods for the file.

### 6 9.9.1.31 UNFORMATTED= specifier in the INQUIRE statement

7 The *scalar-default-char-variable* in the UNFORMATTED= specifier is assigned the value YES if UN-  
8 FORMATTED is included in the set of allowed forms for the file, NO if UNFORMATTED is not included  
9 in the set of allowed forms for the file, and UNKNOWN if the processor is unable to determine whether  
10 or not UNFORMATTED is included in the set of allowed forms for the file.

### 11 9.9.1.32 WRITE= specifier in the INQUIRE statement

12 The *scalar-default-char-variable* in the WRITE= specifier is assigned the value YES if WRITE is included  
13 in the set of allowed actions for the file, NO if WRITE is not included in the set of allowed actions for  
14 the file, and UNKNOWN if the processor is unable to determine whether or not WRITE is included in  
15 the set of allowed actions for the file.

## 16 9.9.2 Restrictions on inquiry specifiers

17 The *inquire-spec-list* in an INQUIRE by file statement shall contain exactly one FILE= specifier and  
18 shall not contain a UNIT= specifier. The *inquire-spec-list* in an INQUIRE by unit statement shall  
19 contain exactly one UNIT= specifier and shall not contain a FILE= specifier. The unit specified need  
20 not exist or be connected to a file. If it is connected to a file, the inquiry is being made about the  
21 connection and about the file connected.

## 22 9.9.3 Inquire by output list

23 The inquire by output list form of the INQUIRE statement has only an IOLENGTH= specifier and an  
24 output list.

25 The *scalar-int-variable* in the IOLENGTH= specifier is assigned the processor-dependent number of file  
26 storage units that would be required to store the data of the output list in an unformatted file. The  
27 value shall be suitable as a RECL= specifier in an OPEN statement that connects a file for unformatted  
28 direct access when there are input/output statements with the same input/output list.

29 The output list in an INQUIRE statement shall not contain any derived-type list items that require a  
30 user-defined derived-type input/output procedure as described in section 9.5.2. If a derived-type list  
31 item appears in the output list, the value returned for the IOLENGTH= specifier assumes that no  
32 user-defined derived-type input/output procedure will be invoked.

## 33 9.10 Error, end-of-record, and end-of-file conditions

34 The set of input/output error conditions is processor dependent.

35 An **end-of-record condition** occurs when a nonadvancing input statement attempts to transfer data  
36 from a position beyond the end of the current record, unless the file is a stream file and the current  
37 record is at the end of the file (an end-of-file condition occurs instead).

38 An **end-of-file condition** occurs in the following cases:

- 39 (1) When an endfile record is encountered during the reading of a file connected for sequential  
40 access.

- 1           (2) When an attempt is made to read a record beyond the end of an internal file.  
2           (3) When an attempt is made to read beyond the end of a stream file.

3 An end-of-file condition may occur at the beginning of execution of an input statement. An end-of-file  
4 condition also may occur during execution of a formatted input statement when more than one record  
5 is required by the interaction of the input list and the format. An end-of-file condition also may occur  
6 during execution of a stream input statement.

### 7 **9.10.1 Error conditions and the ERR= specifier**

8 If an error condition occurs during execution of an input/output statement, the position of the file  
9 becomes indeterminate.

10 If an error condition occurs during execution of an input/output statement that contains neither an  
11 ERR= nor IOSTAT= specifier, execution of the program is terminated. If an error condition occurs  
12 during execution of an input/output statement that contains either an ERR= specifier or an IOSTAT=  
13 specifier then

- 14           (1) Processing of the input/output list, if any, terminates,  
15           (2) If the statement is a data transfer statement or the error occurs during a wait operation,  
16               all *do-variables* in the statement that initiated the transfer become undefined,  
17           (3) If an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes  
18               defined as specified in 9.10.4,  
19           (4) If an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,  
20           (5) If the statement is a READ statement and it contains a SIZE= specifier, the *scalar-int-*  
21               *variable* in the SIZE= specifier becomes defined as specified in 9.5.1.14,  
22           (6) If the statement is a READ statement or the error condition occurs in a wait operation for  
23               a transfer initiated by a READ statement, all input items or namelist group objects in the  
24               statement that initiated the transfer become undefined, and  
25           (7) If an ERR= specifier appears, execution continues with the statement labeled by the *label*  
26               in the ERR= specifier.

### 27 **9.10.2 End-of-file condition and the END= specifier**

28 If an end-of-file condition occurs during execution of an input/output statement that contains neither  
29 an END= specifier nor an IOSTAT= specifier, execution of the program is terminated. If an end-of-file  
30 condition occurs during execution of an input/output statement that contains either an END= specifier  
31 or an IOSTAT= specifier, and an error condition does not occur then

- 32           (1) Processing of the input list, if any, terminates,  
33           (2) If the statement is a data transfer statement or the error occurs during a wait operation,  
34               all *do-variables* in the statement that initiated the transfer become undefined,  
35           (3) If the statement is a READ statement or the end-of-file condition occurs in a wait operation  
36               for a transfer initiated by a READ statement, all input list items or namelist group objects  
37               in the statement that initiated the transfer become undefined,  
38           (4) If the file specified in the input statement is an external record file, it is positioned after the  
39               endfile record,  
40           (5) If an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes  
41               defined as specified in 9.10.4,  
42           (6) If an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,  
43               and  
44           (7) If an END= specifier appears, execution continues with the statement labeled by the *label*  
45               in the END= specifier.

### 1 9.10.3 End-of-record condition and the EOR= specifier

2 If an end-of-record condition occurs during execution of an input/output statement that contains neither  
 3 an EOR= specifier nor an IOSTAT= specifier, execution of the program is terminated. If an end-of-  
 4 record condition occurs during execution of an input/output statement that contains either an EOR=  
 5 specifier or an IOSTAT= specifier, and an error condition does not occur then

- 6 (1) If the pad mode has the value YES, the record is padded with blanks to satisfy the input  
 7 list item (9.5.3.4.2) and corresponding data edit descriptor that requires more characters  
 8 than the record contains. If the pad mode has the value NO, the input list item becomes  
 9 undefined.
- 10 (2) Processing of the input list, if any, terminates,
- 11 (3) If the statement is a data transfer statement or the error occurs during a wait operation,  
 12 all *do-variables* in the statement that initiated the transfer become undefined,
- 13 (4) The file specified in the input statement is positioned after the current record,
- 14 (5) If an IOSTAT= specifier appears, the *scalar-int-variable* in the IOSTAT= specifier becomes  
 15 defined as specified in 9.10.4,
- 16 (6) If an IOMSG= specifier appears, the *iomsg-variable* becomes defined as specified in 9.10.5,
- 17 (7) If a SIZE= specifier appears, the *scalar-int-variable* in the SIZE= specifier becomes defined  
 18 as specified in (9.5.1.14), and
- 19 (8) If an EOR= specifier appears, execution continues with the statement labeled by the *label*  
 20 in the EOR= specifier.

### 21 9.10.4 IOSTAT= specifier

22 Execution of an input/output statement containing the IOSTAT= specifier causes the *scalar-int-variable*  
 23 in the IOSTAT= specifier to become defined with

- 24 (1) A zero value if neither an error condition, an end-of-file condition, nor an end-of-record  
 25 condition occurs,
- 26 (2) A processor-dependent positive integer value if an error condition occurs,
- 27 (3) The processor-dependent negative integer value of the constant IOSTAT\_END (13.8.2.5) if  
 28 an end-of-file condition occurs and no error condition occurs, or
- 29 (4) The processor-dependent negative integer value of the constant IOSTAT\_EOR (13.8.2.6) if  
 30 an end-of-record condition occurs and no error condition or end-of-file condition occurs.

#### NOTE 9.65

An end-of-file condition may occur only for sequential or stream input and an end-of-record condition may occur only for nonadvancing input.

Consider the example:

```

READ (FMT = "(E8.3)", UNIT = 3, IOSTAT = IOSS) X
IF (IOSS < 0) THEN
    ! Perform end-of-file processing on the file connected to unit 3.
    CALL END_PROCESSING
ELSE IF (IOSS > 0) THEN
    ! Perform error processing
    CALL ERROR_PROCESSING
END IF

```

### 1 9.10.5 IOMSG= specifier

2 If an error, end-of-file, or end-of-record condition occurs during execution of an input/output statement,  
3 the processor shall assign an explanatory message to *iomsg-variable*. If no such condition occurs, the  
4 processor shall not change the value of *iomsg-variable*.

## 5 9.11 Restrictions on input/output statements

6 If a unit, or a file connected to a unit, does not have all of the properties required for the execution of  
7 certain input/output statements, those statements shall not refer to the unit.

8 An input/output statement that is executed while another input/output statement is being executed is  
9 called a **recursive input/output statement**.

10 A recursive input/output statement shall not identify an external unit except that a child data transfer  
11 statement may identify its parent data transfer statement external unit.

12 An input/output statement shall not cause the value of any established format specification to be  
13 modified.

14 A recursive input/output statement shall not modify the value of any internal unit except that a recursive  
15 WRITE statement may modify the internal unit identified by that recursive WRITE statement.

16 The value of a specifier in an input/output statement shall not depend on any *input-item*, *io-implied-*  
17 *do do-variable*, or on the definition or evaluation of any other specifier in the *io-control-spec-list* or  
18 *inquire-spec-list* in that statement.

19 The value of any subscript or substring bound of a variable that appears in a specifier in an input/output  
20 statement shall not depend on any *input-item*, *io-implied-do do-variable*, or on the definition or evalua-  
21 tion of any other specifier in the *io-control-spec-list* or *inquire-spec-list* in that statement.

22 In a data transfer statement, the variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier, if  
23 any, shall not be associated with any entity in the data transfer input/output list (9.5.2) or *namelist-*  
24 *group-object-list*, nor with a *do-variable* of an *io-implied-do* in the data transfer input/output list.

25 In a data transfer statement, if a variable specified in an IOSTAT=, IOMSG=, or SIZE= specifier is an  
26 array element reference, its subscript values shall not be affected by the data transfer, the *io-implied-do*  
27 processing, or the definition or evaluation of any other specifier in the *io-control-spec-list*.

28 A variable that may become defined or undefined as a result of its use in a specifier in an INQUIRE  
29 statement, or any associated entity, shall not appear in another specifier in the same INQUIRE statement.

30 A STOP statement shall not be executed during execution of an input/output statement.

#### NOTE 9.66

Restrictions on the evaluation of expressions (7.1.8) prohibit certain side effects.



## 1 Section 10: Input/output editing

2 A format used in conjunction with an input/output statement provides information that directs the  
3 editing between the internal representation of data and the characters of a sequence of formatted records.

4 A FMT= specifier (9.5.1.1) in an input/output statement may refer to a FORMAT statement or to a  
5 character expression that contains a format specification. A format specification provides explicit editing  
6 information. The FMT= specifier alternatively may be an asterisk (\*), which indicates list-directed  
7 formatting (10.9). Namelist formatting (10.10) may be indicated by specifying a *namelist-group-name*  
8 instead of a *format*.

### 9 10.1 Explicit format specification methods

10 Explicit format specification may be given

- 11 (1) In a FORMAT statement or
- 12 (2) In a character expression.

#### 13 10.1.1 FORMAT statement

14 R1001 *format-stmt* is FORMAT *format-specification*  
15 R1002 *format-specification* is ( [ *format-item-list* ] )

16 C1001 (R1001) The *format-stmt* shall be labeled.

17 C1002 (R1002) The comma used to separate *format-items* in a *format-item-list* may be omitted

- 18 (1) Between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit  
19 descriptor (10.7.5), possibly preceded by a repeat specifier,
- 20 (2) Before a slash edit descriptor when the optional repeat specification is not present (10.7.2),
- 21 (3) After a slash edit descriptor, or
- 22 (4) Before or after a colon edit descriptor (10.7.3)

23 Blank characters may precede the initial left parenthesis of the format specification. Additional blank  
24 characters may appear at any point within the format specification, with no effect on the interpretation  
25 of the format specification, except within a character string edit descriptor (10.8).

#### NOTE 10.1

Examples of FORMAT statements are:

```
5   FORMAT (1PE12.4, I10)
9   FORMAT (I12, /, ' Dates: ', 2 (2I3, I5))
```

#### 26 10.1.2 Character format specification

27 A character expression used as a *format* in a formatted input/output statement shall evaluate to a  
28 character string whose leading part is a valid format specification.

**NOTE 10.2**

The format specification begins with a left parenthesis and ends with a right parenthesis.

- 1 All character positions up to and including the final right parenthesis of the format specification shall be
- 2 defined at the time the input/output statement is executed, and shall not become redefined or undefined
- 3 during the execution of the statement. Character positions, if any, following the right parenthesis that
- 4 ends the format specification need not be defined and may contain any character data with no effect on
- 5 the interpretation of the format specification.
- 6 If the *format* is a character array, it is treated as if all of the elements of the array were specified in array
- 7 element order and were concatenated. However, if a *format* is a character array element, the format
- 8 specification shall be entirely within that array element.

**NOTE 10.3**

If a character constant is used as a *format* in an input/output statement, care shall be taken that the value of the character constant is a valid format specification. In particular, if a format specification delimited by apostrophes contains a character constant edit descriptor delimited with apostrophes, two apostrophes shall be written to delimit the edit descriptor and four apostrophes shall be written for each apostrophe that occurs within the edit descriptor. For example, the text:

```
2 ISN'T 3
```

may be written by various combinations of output statements and format specifications:

```
WRITE (6, 100) 2, 3
100 FORMAT (1X, I1, 1X, 'ISN''T', 1X, I1)
WRITE (6, '(1X, I1, 1X, ''ISN''''T'', 1X, I1)') 2, 3
WRITE (6, '(A)' ) ' 2 ISN''T 3'
```

Doubling of internal apostrophes usually can be avoided by using quotation marks to delimit the format specification and doubling of internal quotation marks usually can be avoided by using apostrophes as delimiters.

## 9 **10.2 Form of a format item list**

- |    |       |                    |           |  |
|----|-------|--------------------|-----------|--|
| 10 | R1003 | <i>format-item</i> | <b>is</b> | [ <i>r</i> ] <i>data-edit-desc</i>       |
| 11 |       |                    | <b>or</b> | <i>control-edit-desc</i>                 |
| 12 |       |                    | <b>or</b> | <i>char-string-edit-desc</i>             |
| 13 |       |                    | <b>or</b> | [ <i>r</i> ] ( <i>format-item-list</i> ) |
| 14 | R1004 | <i>r</i>           | <b>is</b> | <i>int-literal-constant</i>              |

15 C1003 (R1004) *r* shall be positive.

16 C1004 (R1004) *r* shall not have a kind parameter specified for it.

17 The integer literal constant *r* is called a **repeat specification**.

### 18 **10.2.1 Edit descriptors**

19 An **edit descriptor** is a **data edit descriptor**, a **control edit descriptor**, or a **character string**  
 20 **edit descriptor**.

- 1 R1005 *data-edit-desc*            **is** I *w* [ . *m* ]  
2    **or** B *w* [ . *m* ]  
3    **or** O *w* [ . *m* ]  
4    **or** Z *w* [ . *m* ]  
5    **or** F *w* . *d*  
6    **or** E *w* . *d* [ E *e* ]  
7    **or** EN *w* . *d* [ E *e* ]  
8    **or** ES *w* . *d* [ E *e* ]  
9    **or** G *w* . *d* [ E *e* ]  
10     **or** L *w*  
11     **or** A [ *w* ]  
12     **or** D *w* . *d*  
13     **or** DT [ *char-literal-constant* ] [ ( *v-list* ) ]  
14 R1006 *w*                                **is** *int-literal-constant*  
15 R1007 *m*                                **is** *int-literal-constant*  
16 R1008 *d*                                **is** *int-literal-constant*  
17 R1009 *e*                                **is** *int-literal-constant*  
18 R1010 *v*                                **is** *signed-int-literal-constant*
- 19 C1005 (R1009) *e* shall be positive.
- 20 C1006 (R1006) *w* shall be zero or positive for the I, B, O, Z, and F edit descriptors. *w* shall be positive  
21 for all other edit descriptors.
- 22 C1007 (R1005) *w*, *m*, *d*, *e*, and *v* shall not have kind parameters specified for them.
- 23 C1008 (R1005) The *char-literal-constant* in the DT edit descriptor shall not have a kind parameter  
24 specified for it.
- 25 I, B, O, Z, F, E, EN, ES, G, L, A, D, and DT indicate the manner of editing.
- 26 R1011 *control-edit-desc*            **is** *position-edit-desc*  
27    **or** [ *r* ] /  
28    **or** :  
29    **or** *sign-edit-desc*  
30    **or** *k* P  
31    **or** *blank-interp-edit-desc*  
32    **or** *round-edit-desc*  
33    **or** *decimal-edit-desc*  
34 R1012 *k*                                **is** *signed-int-literal-constant*
- 35 C1009 (R1012) *k* shall not have a kind parameter specified for it.
- 36 In *k* P, *k* is called the **scale factor**.
- 37 R1013 *position-edit-desc*            **is** T *n*  
38    **or** TL *n*  
39    **or** TR *n*  
40    **or** *n* X  
41 R1014 *n*                                **is** *int-literal-constant*
- 42 C1010 (R1014) *n* shall be positive.
- 43 C1011 (R1014) *n* shall not have a kind parameter specified for it.
- 44 R1015 *sign-edit-desc*                **is** SS  
45    **or** SP

1			or S
2	R1016	<i>blank-interp-edit-desc</i>	is BN
3			or BZ
4	R1017	<i>round-edit-desc</i>	is RU
5			or RD
6			or RZ
7			or RN
8			or RC
9			or RP
10	R1018	<i>decimal-edit-desc</i>	is DC
11			or DP

12 T, TL, TR, X, slash, colon, SS, SP, S, P, BN, BZ, RU, RD, RZ, RN, RC, RP, DC, and DP indicate the  
13 manner of editing.

14 R1019 *char-string-edit-desc* is *char-literal-constant*

15 C1012 (R1019) The *char-literal-constant* shall not have a kind parameter specified for it.

16 Each *rep-char* in a character string edit descriptor shall be one of the characters capable of representation  
17 by the processor.

18 The character string edit descriptors provide constant data to be output, and are not valid for input.

19 The edit descriptors are without regard to case except for the characters in the character constants.

## 20 10.2.2 Fields

21 A **field** is a part of a record that is read on input or written on output when format control encounters  
22 a data edit descriptor or a character string edit descriptor. The **field width** is the size in characters of  
23 the field.

## 24 10.3 Interaction between input/output list and format

25 The start of formatted data transfer using a format specification initiates **format control** (9.5.3.4.2).

26 Each action of format control depends on information jointly provided by

- 27 (1) The next edit descriptor in the format specification and
- 28 (2) The next effective item in the input/output list, if one exists.

29 If an input/output list specifies at least one effective list item, at least one data edit descriptor shall  
30 exist in the format specification.

### NOTE 10.4

An empty format specification of the form ( ) may be used only if the input/output list has no effective list items (9.5.3.4). Zero length character items are effective list items, but zero sized arrays and implied DO lists with an iteration count of zero are not.

31 A format specification is interpreted from left to right. The exceptions are format items preceded by a  
32 repeat specification *r*, and format reversion (described below).

33 A format item preceded by a repeat specification is processed as a list of *r* items, each identical to the  
34 format item but without the repeat specification and separated by commas.

**NOTE 10.5**

An omitted repeat specification is treated in the same way as a repeat specification whose value is one.

- 1 To each data edit descriptor interpreted in a format specification, there corresponds one effective item  
 2 specified by the input/output list (9.5.2), except that an input/output list item of type complex requires  
 3 the interpretation of two F, E, EN, ES, D, or G edit descriptors. For each control edit descriptor or  
 4 character edit descriptor, there is no corresponding item specified by the input/output list, and format  
 5 control communicates information directly with the record.
- 6 Whenever format control encounters a data edit descriptor in a format specification, it determines  
 7 whether there is a corresponding effective item specified by the input/output list. If there is such an  
 8 item, it transmits appropriately edited information between the item and the record, and then format  
 9 control proceeds. If there is no such item, format control terminates.
- 10 If format control encounters a colon edit descriptor in a format specification and another effective in-  
 11 put/output list item is not specified, format control terminates.
- 12 If format control encounters the rightmost parenthesis of a complete format specification and another  
 13 effective input/output list item is not specified, format control terminates. However, if another effective  
 14 input/output list item is specified, the file is positioned in a manner identical to the way it is positioned  
 15 when a slash edit descriptor is processed (10.7.2). Format control then reverts to the beginning of the  
 16 format item terminated by the last preceding right parenthesis that is not part of a DT edit descriptor.  
 17 If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the  
 18 format specification. If any reversion occurs, the reused portion of the format specification shall contain  
 19 at least one data edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat  
 20 specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on  
 21 the changeable modes (9.4.1).

**NOTE 10.6**

Example: The format specification:

```
10 FORMAT (1X, 2(F10.3, I5))
```

with an output list of

```
WRITE (10,10) 10.1, 3, 4.7, 1, 12.4, 5, 5.2, 6
```

produces the same output as the format specification:

```
10 FORMAT (1X, F10.3, I5, F10.3, I5/F10.3, I5, F10.3, I5)
```

## 22 10.4 Positioning by format control

- 23 After each data edit descriptor or character string edit descriptor is processed, the file is positioned after  
 24 the last character read or written in the current record.
- 25 After each T, TL, TR, or X edit descriptor is processed, the file is positioned as described in 10.7.1.  
 26 After each slash edit descriptor is processed, the file is positioned as described in 10.7.2.
- 27 During formatted stream output, processing of an A edit descriptor may cause file positioning to occur  
 28 (10.6.3).

- 1 If format control reverts as described in 10.3, the file is positioned in a manner identical to the way it is  
2 positioned when a slash edit descriptor is processed (10.7.2).
- 3 During a read operation, any unprocessed characters of the current record are skipped whenever the  
4 next record is read.

## 5 10.5 Decimal symbol

6 The **decimal symbol** is the character that separates the whole and fractional parts in the decimal  
7 representation of a real number in an internal or external file. When the decimal edit mode is POINT,  
8 the decimal symbol is a decimal point. When the decimal edit mode is COMMA, the decimal symbol is  
9 a comma.

## 10 10.6 Data edit descriptors

11 Data edit descriptors cause the conversion of data to or from its internal representation; during formatted  
12 stream output, the A data edit descriptor may also cause file positioning. On input, the specified variable  
13 becomes defined unless an error condition, an end-of-file condition, or an end-of-record condition occurs.  
14 On output, the specified expression is evaluated.

15 During input from a Unicode file,

- 16 (1) characters in the record that correspond to an ASCII character variable shall have a position  
17 in the ISO 10646 character type collating sequence of 127 or less, and
- 18 (2) characters in the record that correspond to a default character variable shall be representable  
19 in the default character type.

20 During input from a non-Unicode file,

- 21 (1) characters in the record that correspond to a character variable shall have the kind of the  
22 character variable, and
- 23 (2) characters in the record that correspond to a numeric or logical variable shall be of default  
24 character type.

25 During output to a Unicode file, all characters transmitted to the record are of ISO 10646 character  
26 type. If a character input/output list item or character string edit descriptor contains a character that  
27 is not representable in the ISO 10646 character type, the result is processor-dependent.

28 During output to a non-Unicode file, characters transmitted to the record as a result of processing a  
29 character string edit descriptor or as a result of evaluating a numeric, logical, or default character data  
30 entity, are of type default character.

### 31 10.6.1 Numeric editing

32 The I, B, O, Z, F, E, EN, ES, D, and G edit descriptors may be used to specify the input/output of  
33 integer, real, and complex data. The following general rules apply:

- 34 (1) On input, leading blanks are not significant. When the input field is not an IEEE excep-  
35 tional specification (10.6.1.2.1), the interpretation of blanks, other than leading blanks, is  
36 determined by the blank interpretation mode (10.7.6). Plus signs may be omitted. A field  
37 containing only blanks is considered to be zero.
- 38 (2) On input, with F, E, EN, ES, D, and G editing, a decimal symbol appearing in the input  
39 field overrides the portion of an edit descriptor that specifies the decimal symbol location.  
40 The input field may have more digits than the processor uses to approximate the value of  
41 the datum.

- 1 (3) On output with I, F, E, EN, ES, D, and G editing, the representation of a positive or zero  
2 internal value in the field may be prefixed with a plus sign, as controlled by the S, SP, and  
3 SS edit descriptors or the processor. The representation of a negative internal value in the  
4 field shall be prefixed with a minus sign.
- 5 (4) On output, the representation is right justified in the field. If the number of characters  
6 produced by the editing is smaller than the field width, leading blanks are inserted in the  
7 field.
- 8 (5) On output, if the number of characters produced exceeds the field width or if an exponent  
9 exceeds its specified length using the *Ew.d Ee*, *ENw.d Ee*, *ESw.d Ee*, or *Gw.d Ee* edit  
10 descriptor, the processor shall fill the entire field of width *w* with asterisks. However,  
11 the processor shall not produce asterisks if the field width is not exceeded when optional  
12 characters are omitted.

**NOTE 10.7**

When an SP edit descriptor is in effect, a plus sign is not optional.

- 13 (6) On output, with I, B, O, Z, and F editing, the specified value of the field width *w* may be  
14 zero. In such cases, the processor selects the smallest positive actual field width that does  
15 not result in a field filled with asterisks. The specified value of *w* shall not be zero on input.

**10.6.1.1 Integer editing**

17 The *Iw*, *Iw.m*, *Bw*, *Bw.m*, *Ow*, *Ow.m*, *Zw*, and *Zw.m* edit descriptors indicate that the field to be edited  
18 occupies *w* positions, except when *w* is zero. When *w* is zero, the processor selects the field width. On  
19 input, *w* shall not be zero. The specified input/output list item shall be of type integer. The G edit  
20 descriptor also may be used to edit integer data (10.6.4.1.1).

21 On input, *m* has no effect.

22 In the input field for the I edit descriptor, the character string shall be a *signed-digit-string* (R408),  
23 except for the interpretation of blanks. For the B, O, and Z edit descriptors, the character string shall  
24 consist of binary, octal, or hexadecimal digits (as in R412, R413, R414) in the respective input field. The  
25 lower-case hexadecimal digits a through f in a hexadecimal input field are equivalent to the corresponding  
26 upper-case hexadecimal digits.

27 The output field for the *Iw* edit descriptor consists of zero or more leading blanks followed by a minus  
28 sign if the internal value is negative, or an optional plus sign otherwise, followed by the magnitude of  
29 the internal value as a *digit-string* without leading zeros.

**NOTE 10.8**

A *digit-string* always consists of at least one digit.

30 The output field for the *Bw*, *Ow*, and *Zw* descriptors consists of zero or more leading blanks followed by  
31 the internal value in a form identical to the digits of a binary, octal, or hexadecimal constant, respectively,  
32 with the same value and without leading zeros.

**NOTE 10.9**

A binary, octal, or hexadecimal constant always consists of at least one digit.

33 The output field for the *Iw.m*, *Bw.m*, *Ow.m*, and *Zw.m* edit descriptor is the same as for the *Iw*, *Bw*,  
34 *Ow*, and *Zw* edit descriptor, respectively, except that the *digit-string* consists of at least *m* digits. If  
35 necessary, sufficient leading zeros are included to achieve the minimum of *m* digits. The value of *m* shall  
36 not exceed the value of *w*, except when *w* is zero. If *m* is zero and the internal value is zero, the output  
37 field consists of only blank characters, regardless of the sign control in effect. When *m* and *w* are both

1 zero, and the internal value is zero, one blank character is produced.

## 2 10.6.1.2 Real and complex editing

3 The F, E, EN, ES, and D edit descriptors specify the editing of real and complex data. An input/output  
4 list item corresponding to an F, E, EN, ES, or D edit descriptor shall be real or complex. The G edit  
5 descriptor also may be used to edit real and complex data (10.6.4.1.2).

6 A lower-case letter is equivalent to the corresponding upper-case letter in an IEEE exceptional speci-  
7 fication or the exponent in a numeric input field.

### 8 10.6.1.2.1 F editing

9 The *Fw.d* edit descriptor indicates that the field occupies  $w$  positions, the fractional part of which  
10 consists of  $d$  digits. When  $w$  is zero, the processor selects the field width. On input,  $w$  shall not be zero.

11 The input field is either an IEEE exceptional specification or consists of an optional sign, followed by a  
12 string of one or more digits optionally containing a decimal symbol, including any blanks interpreted as  
13 zeros. The  $d$  has no effect on input if the input field contains a decimal symbol. If the decimal symbol  
14 is omitted, the rightmost  $d$  digits of the string, with leading zeros assumed if necessary, are interpreted  
15 as the fractional part of the value represented. The string of digits may contain more digits than a  
16 processor uses to approximate the value. The basic form may be followed by an exponent of one of the  
17 following forms:

- 18 (1) A *sign* followed by a *digit-string*
- 19 (2) E followed by zero or more blanks, followed by a *signed-digit-string*
- 20 (3) D followed by zero or more blanks, followed by a *signed-digit-string*

21 An exponent containing a D is processed identically to an exponent containing an E.

#### NOTE 10.10

If the input field does not contain an exponent, the effect is as if the basic form were followed by  
an exponent with a value of  $-k$ , where  $k$  is the established scale factor (10.7.5).

22 An input field that is an IEEE exceptional specification consists of optional blanks, followed by either  
23 of

- 24 (1) an optional sign, followed by the string 'INF' or the string 'INFINITY' or
- 25 (2) an optional sign, followed by the string 'NaN', optionally followed by zero or more alphanu-  
26 meric characters enclosed in parentheses,

27 optionally followed by blanks.

28 The value specified by form (1) is an IEEE infinity; this form shall not be used if the processor does  
29 not support IEEE infinities for the input variable. The value specified by form (2) is an IEEE NaN;  
30 this form shall not be used if the processor does not support IEEE NaNs for the input variable. The  
31 NaN value is a quiet NaN if the only nonblank characters in the field are 'NaN' or 'NaN()'; otherwise,  
32 the NaN value is processor-dependent. The interpretation of a sign in a NaN input field is processor  
33 dependent.

34 For an internal value that is an IEEE infinity, the output field consists of blanks, if necessary, followed  
35 by a minus sign for negative infinity or an optional plus sign otherwise, followed by the letters 'Inf' or  
36 'Infinity', right justified within the field. If  $w$  is less than 3, the field is filled with asterisks; otherwise,  
37 if  $w$  is less than 8, 'Inf' is produced.

38 For an internal value that is an IEEE NaN, the output field consists of blanks, if necessary, followed by

- 1 the letters 'NaN' and optionally followed by one to  $w - 5$  alphanumeric processor-dependent characters  
 2 enclosed in parentheses, right justified within the field. If  $w$  is less than 3, the field is filled with asterisks.

**NOTE 10.11**

The processor-dependent characters following 'NaN' may convey additional information about that particular NaN.

- 3 For an internal value that is neither an IEEE infinity nor a NaN, the output field consists of blanks, if  
 4 necessary, followed by a minus sign if the internal value is negative, or an optional plus sign otherwise,  
 5 followed by a string of digits that contains a decimal symbol and represents the magnitude of the internal  
 6 value, as modified by the established scale factor and rounded to  $d$  fractional digits. Leading zeros are  
 7 not permitted except for an optional zero immediately to the left of the decimal symbol if the magnitude  
 8 of the value in the output field is less than one. The optional zero shall appear if there would otherwise  
 9 be no digits in the output field.

**10 10.6.1.2.2 E and D editing**

- 11 The *Ew.d*, *Dw.d*, and *Ew.d Ee* edit descriptors indicate that the external field occupies  $w$  positions, the  
 12 fractional part of which consists of  $d$  digits, unless a scale factor greater than one is in effect, and the  
 13 exponent part consists of  $e$  digits. The  $e$  has no effect on input.

- 14 The form and interpretation of the input field is the same as for *Fw.d* editing (10.6.1.2.1).

- 15 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for  
 16 *Fw.d*.

- 17 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field for a scale  
 18 factor of zero is:

19  $[ \pm ] [0].x_1x_2 \dots x_d exp$

- 20 where:

21  $\pm$  signifies a plus sign or a minus sign.

22  $.$  signifies a decimal symbol (10.5).

23  $x_1x_2 \dots x_d$  are the  $d$  most significant digits of the internal value after rounding.

24  $exp$  is a decimal exponent having one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
<i>Ew.d</i>	$ exp  \leq 99$	$E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 <  exp  \leq 999$	$\pm z_1z_2z_3$
<i>Ew.d Ee</i>	$ exp  \leq 10^e - 1$	$E\pm z_1z_2 \dots z_e$
<i>Dw.d</i>	$ exp  \leq 99$	$D\pm z_1z_2$ or $E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 <  exp  \leq 999$	$\pm z_1z_2z_3$

- 25 where each  $z$  is a digit.

- 26 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit  
 27 descriptor forms *Ew.d* and *Dw.d* shall not be used if  $|exp| > 999$ .

- 28 The scale factor  $k$  controls the decimal normalization (10.2.1, 10.7.5). If  $-d < k \leq 0$ , the output field  
 29 contains exactly  $|k|$  leading zeros and  $d - |k|$  significant digits after the decimal symbol. If  $0 < k < d + 2$ ,  
 30 the output field contains exactly  $k$  significant digits to the left of the decimal symbol and  $d - k + 1$

1 significant digits to the right of the decimal symbol. Other values of  $k$  are not permitted.

2 **10.6.1.2.3 EN editing**

3 The EN edit descriptor produces an output field in the form of a real number in engineering notation  
 4 such that the decimal exponent is divisible by three and the absolute value of the significand (R418) is  
 5 greater than or equal to 1 and less than 1000, except when the output value is zero. The scale factor  
 6 has no effect on output.

7 The forms of the edit descriptor are  $ENw.d$  and  $ENw.d Ee$  indicating that the external field occupies  $w$   
 8 positions, the fractional part of which consists of  $d$  digits and the exponent part consists of  $e$  digits.

9 The form and interpretation of the input field is the same as for  $Fw.d$  editing (10.6.1.2.1).

10 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for  
 11  $Fw.d$ .

12 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is:

13  $[ \pm ] yyy . x_1x_2 \dots x_d exp$

14 where:

15  $\pm$  signifies a plus sign or a minus sign.

16  $yyy$  are the 1 to 3 decimal digits representative of the most significant digits of the internal  
 17 value after rounding ( $yyy$  is an integer such that  $1 \leq yyy < 1000$  or, if the output value is  
 18 zero,  $yyy = 0$ ).

19  $.$  signifies a decimal symbol (10.5).

20  $x_1x_2 \dots x_d$  are the  $d$  next most significant digits of the internal value after rounding.

21  $exp$  is a decimal exponent, divisible by three, of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
$ENw.d$	$ exp  \leq 99$	$E\pm z_1z_2$ or $\pm 0z_1z_2$
	$99 <  exp  \leq 999$	$\pm z_1z_2z_3$
$ENw.d Ee$	$ exp  \leq 10^e - 1$	$E\pm z_1z_2 \dots z_e$

22 where each  $z$  is a digit.

23 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit  
 24 descriptor form  $ENw.d$  shall not be used if  $|exp| > 999$ .

**NOTE 10.12**

Examples:	
Internal Value	Output field Using SS, EN12.3
6.421	6.421E+00
-.5	-500.000E-03
.00217	2.170E-03
4721.3	4.721E+03

25 **10.6.1.2.4 ES editing**

26 The ES edit descriptor produces an output field in the form of a real number in scientific notation such  
 27 that the absolute value of the significand (R418) is greater than or equal to 1 and less than 10, except

- 1 when the output value is zero. The scale factor has no effect on output.
- 2 The forms of the edit descriptor are  $ESw.d$  and  $ESw.d Ee$  indicating that the external field occupies  $w$   
3 positions, the fractional part of which consists of  $d$  digits and the exponent part consists of  $e$  digits.
- 4 The form and interpretation of the input field is the same as for  $Fw.d$  editing (10.6.1.2.1).
- 5 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for  
6  $Fw.d$ .
- 7 For an internal value that is neither an IEEE infinity nor a NaN, the form of the output field is:

$$8 \quad [ \pm ] y . x_1 x_2 \dots x_d exp$$

9 where:

- 10  $\pm$  signifies a plus sign or a minus sign.
- 11  $y$  is a decimal digit representative of the most significant digit of the internal value after  
12 rounding.
- 13  $.$  signifies a decimal symbol (10.5).
- 14  $x_1 x_2 \dots x_d$  are the  $d$  next most significant digits of the internal value after rounding.
- 15  $exp$  is a decimal exponent having one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
$ESw.d$	$ exp  \leq 99$	$E\pm z_1 z_2$ or $\pm 0 z_1 z_2$
	$99 <  exp  \leq 999$	$\pm z_1 z_2 z_3$
$ESw.d Ee$	$ exp  \leq 10^e - 1$	$E\pm z_1 z_2 \dots z_e$

- 16 where each  $z$  is a digit.
- 17 The sign in the exponent is produced. A plus sign is produced if the exponent value is zero. The edit  
18 descriptor form  $ESw.d$  shall not be used if  $|exp| > 999$ .

#### NOTE 10.13

Examples:

Internal Value	Output field Using SS, ES12.3
6.421	6.421E+00
-.5	-5.000E-01
.00217	2.170E-03
4721.3	4.721E+03

#### 19 10.6.1.2.5 Complex editing

- 20 A complex datum consists of a pair of separate real data. The editing of a scalar datum of complex type  
21 is specified by two edit descriptors each of which specifies the editing of real data. The first of the edit  
22 descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may  
23 be different. Control and character string edit descriptors may be processed between the edit descriptor  
24 for the real part and the edit descriptor for the imaginary part.

#### 25 10.6.1.2.6 Rounding mode

- 26 The rounding mode can be specified by an OPEN statement (9.4.1), a data transfer input/output  
27 statement (9.5.1.12), or an edit descriptor (10.7.7).

1 In what follows, the term “decimal value” means the exact decimal number as given by the character  
2 string, while the term “internal value” means the number actually stored (typically in binary form) in  
3 the processor. For example, in dealing with the decimal constant 0.1, the decimal value is the exact  
4 mathematical quantity  $1/10$ , which has no exact representation on most processors. Formatted output  
5 of real data involves conversion from an internal value to a decimal value; formatted input involves  
6 conversion from a decimal value to an internal value.

7 When the I/O rounding mode is UP, the value resulting from conversion shall be the smallest repre-  
8 sentable value that is greater than or equal to the original value. When the I/O rounding mode is  
9 DOWN, the value resulting from conversion shall be the largest representable value that is less than or  
10 equal to the original value. When the I/O rounding mode is ZERO, the value resulting from conversion  
11 shall be the value closest to the original value and no greater in magnitude than the original value. When  
12 the I/O rounding mode is NEAREST, the value resulting from conversion shall be the closer of the two  
13 nearest representable values if one is closer than the other. If the two nearest representable values are  
14 equidistant from the original value, it is processor dependent which one of them is chosen. When the  
15 I/O rounding mode is COMPATIBLE, the value resulting from conversion shall be the closer of the  
16 two nearest representable values or the value away from zero if halfway between them. When the I/O  
17 rounding mode is PROCESSOR\_DEFINED, rounding during conversion shall be a processor dependent  
18 default mode, which may correspond to one of the other modes.

19 On processors that support IEEE rounding on conversions, NEAREST shall correspond to round to  
20 nearest, as specified in the IEEE standard.

#### NOTE 10.14

On processors that support IEEE rounding on conversions, the I/O rounding modes COMPATIBLE and NEAREST will produce the same results except when the datum is halfway between the two representable values. In that case, NEAREST will pick the even value, but COMPATIBLE will pick the value away from zero. The I/O rounding modes UP, DOWN, and ZERO have the same effect as those specified in the IEEE standard for round toward  $+\infty$ , round toward  $-\infty$ , and round toward 0, respectively.

### 21 10.6.2 Logical editing

22 The  $Lw$  edit descriptor indicates that the field occupies  $w$  positions. The specified input/output list  
23 item shall be of type logical. The G edit descriptor also may be used to edit logical data (10.6.4.2).

24 The input field consists of optional blanks, optionally followed by a period, followed by a T for true or  
25 F for false. The T or F may be followed by additional characters in the field, which are ignored.

26 A lower-case letter is equivalent to the corresponding upper-case letter in a logical input field.

#### NOTE 10.15

The logical constants .TRUE. and .FALSE. are acceptable input forms.

27 The output field consists of  $w - 1$  blanks followed by a T or F, depending on whether the internal value  
28 is true or false, respectively.

### 29 10.6.3 Character editing

30 The  $A[w]$  edit descriptor is used with an input/output list item of type character. The G edit descriptor  
31 also may be used to edit character data (10.6.4.3). The kind type parameter of all characters transferred  
32 and converted under control of one A or G edit descriptor is implied by the kind of the corresponding  
33 list item.

- 1 If a field width  $w$  is specified with the A edit descriptor, the field consists of  $w$  characters. If a field  
 2 width  $w$  is not specified with the A edit descriptor, the number of characters in the field is the length of  
 3 the corresponding list item, regardless of the value of the kind type parameter.
- 4 Let  $len$  be the length of the input/output list item. If the specified field width  $w$  for an A edit descriptor  
 5 corresponding to an input item is greater than or equal to  $len$ , the rightmost  $len$  characters will be taken  
 6 from the input field. If the specified field width  $w$  is less than  $len$ , the  $w$  characters will appear left  
 7 justified with  $len-w$  trailing blanks in the internal value.
- 8 If the specified field width  $w$  for an A edit descriptor corresponding to an output item is greater than  
 9  $len$ , the output field will consist of  $w - len$  blanks followed by the  $len$  characters from the internal value.  
 10 If the specified field width  $w$  is less than or equal to  $len$ , the output field will consist of the leftmost  $w$   
 11 characters from the internal value.

**NOTE 10.16**

For nondefault character types, the blank padding character is processor dependent.

- 12 If the file is connected for stream access, the output may be split across more than one record if it  
 13 contains newline characters. A newline character is a nonblank character returned by the intrinsic  
 14 function NEW\_LINE. Beginning with the first character of the output field, each character that is not  
 15 a newline is written to the current record in successive positions; each newline character causes file  
 16 positioning at that point as if by slash editing (the current record is terminated at that point, a new  
 17 empty record is created following the current record, this new record becomes the last and current record  
 18 of the file, and the file is positioned at the beginning of this new record).

**NOTE 10.17**

If the intrinsic function NEW\_LINE returns a blank character for a particular character kind, then  
 the processor does not support using a character of that kind to cause record termination in a  
 formatted stream file.

## 19 **10.6.4 Generalized editing**

- 20 The  $Gw.d$  and  $Gw.d Ee$  edit descriptors are used with an input/output list item of any intrinsic type.  
 21 These edit descriptors indicate that the external field occupies  $w$  positions, the fractional part of which  
 22 consists of a maximum of  $d$  digits and the exponent part consists of  $e$  digits. When these edit descriptors  
 23 are used to specify the input/output of integer, logical, or character data,  $d$  and  $e$  have no effect.

### 24 **10.6.4.1 Generalized numeric editing**

- 25 When used to specify the input/output of integer, real, and complex data, the  $Gw.d$  and  $Gw.d Ee$  edit  
 26 descriptors follow the general rules for numeric editing (10.6.1).

**NOTE 10.18**

The  $Gw.d Ee$  edit descriptor follows any additional rules for the  $Ew.d Ee$  edit descriptor.

#### 27 **10.6.4.1.1 Generalized integer editing**

- 28 When used to specify the input/output of integer data, the  $Gw.d$  and  $Gw.d Ee$  edit descriptors follow  
 29 the rules for the  $Iw$  edit descriptor (10.6.1.1), except that  $w$  shall not be zero.

#### 30 **10.6.4.1.2 Generalized real and complex editing**

- 31 The form and interpretation of the input field is the same as for  $Fw.d$  editing (10.6.1.2.1).

- 1 For an internal value that is an IEEE infinity or NaN, the form of the output field is the same as for  
 2 *Fw.d*.
- 3 Otherwise, the method of representation in the output field depends on the magnitude of the internal  
 4 value being edited. Let  $N$  be the magnitude of the internal value and  $r$  be the rounding mode value  
 5 defined in the table below. If  $0 < N < 0.1 - r \times 10^{-d-1}$  or  $N \geq 10^d - r$ , or  $N$  is identically 0 and  
 6  $d$  is 0, *Gw.d* output editing is the same as  $k$  *PEw.d* output editing and *Gw.d Ee* output editing is  
 7 the same as  $k$  *PEw.d Ee* output editing, where  $k$  is the scale factor (10.7.5) currently in effect. If  
 8  $0.1 - r \times 10^{-d-1} \leq N < 10^d - r$  or  $N$  is identically 0 and  $d$  is not zero, the scale factor has no effect,  
 9 and the value of  $N$  determines the editing as follows:

Magnitude of Internal Value	Equivalent Conversion
$N = 0$	$F(w - n).(d - 1), n('b')$
$0.1 - r \times 10^{-d-1} \leq N < 1 - r \times 10^{-d}$	$F(w - n).d, n('b')$
$1 - r \times 10^{-d} \leq N < 10 - r \times 10^{-d+1}$	$F(w - n).(d - 1), n('b')$
$10 - r \times 10^{-d+1} \leq N < 100 - r \times 10^{-d+2}$	$F(w - n).(d - 2), n('b')$
.	.
.	.
.	.
$10^{d-2} - r \times 10^{-2} \leq N < 10^{d-1} - r \times 10^{-1}$	$F(w - n).1, n('b')$
$10^{d-1} - r \times 10^{-1} \leq N < 10^d - r$	$F(w - n).0, n('b')$

- 10 where  $b$  is a blank,  $n$  is 4 for *Gw.d* and  $e + 2$  for *Gw.d Ee*, and  $r$  is defined for each rounding mode as  
 11 follows:

I/O Rounding Mode	$r$
COMPATIBLE	0.5
NEAREST	0.5 if the higher value is even -0.5 if the lower value is even
UP	1
DOWN	0
ZERO	1 if internal value is negative 0 if internal value is positive

- 12 The value of  $w - n$  shall be positive

**NOTE 10.19**

The scale factor has no effect on output unless the magnitude of the datum to be edited is outside the range that permits effective use of F editing.

13 **10.6.4.2 Generalized logical editing**

- 14 When used to specify the input/output of logical data, the *Gw.d* and *Gw.d Ee* edit descriptors follow  
 15 the rules for the *Lw* edit descriptor (10.6.2).

16 **10.6.4.3 Generalized character editing**

- 17 When used to specify the input/output of character data, the *Gw.d* and *Gw.d Ee* edit descriptors follow  
 18 the rules for the *Aw* edit descriptor (10.6.3).

## 1 10.6.5 User-defined derived-type editing

2 The DT edit descriptor allows a user-provided procedure to be used instead of the processor's default  
3 input/output formatting for processing a list item of derived type.

4 The DT edit descriptor may include a character literal constant. The character value "DT" concatenated  
5 with the character literal constant is passed to the user-defined derived-type input/output procedure as  
6 the `iotype` argument (9.5.3.7). The  $v$  values of the edit descriptor are passed to the user-defined  
7 derived-type input/output procedure as the `v_list` array argument.

### NOTE 10.20

For the edit descriptor <code>DT'Link List'(10, 4, 2)</code> , <code>iotype</code> is "DTLink List" and <code>v_list</code> is (/10, 4, 2/).
--

8 If a derived-type variable or value corresponds to a DT edit descriptor, there shall be an accessible  
9 interface to a corresponding derived-type input/output procedure for that derived type (9.5.3.7). A DT  
10 edit descriptor shall not correspond with a list item that is not of a derived type.

## 11 10.7 Control edit descriptors

12 A control edit descriptor does not cause the transfer of data or the conversion of data to or from internal  
13 representation, but may affect the conversions performed by subsequent data edit descriptors.

### 14 10.7.1 Position editing

15 The T, TL, TR, and X edit descriptors specify the position at which the next character will be transmitted  
16 to or from the record. If any character skipped by a T, TL, TR, or X edit descriptor is of type nondefault  
17 character, and the unit is an internal file of type default character or an external non-Unicode file, the  
18 result of that position editing is processor dependent.

19 The position specified by a T edit descriptor may be in either direction from the current position. On  
20 input, this allows portions of a record to be processed more than once, possibly with different editing.

21 The position specified by an X edit descriptor is forward from the current position. On input, a position  
22 beyond the last character of the record may be specified if no characters are transmitted from such  
23 positions.

### NOTE 10.21

An $nX$ edit descriptor has the same effect as a $TR_n$ edit descriptor.
--

24 On output, a T, TL, TR, or X edit descriptor does not by itself cause characters to be transmitted and  
25 therefore does not by itself affect the length of the record. If characters are transmitted to positions at  
26 or after the position specified by a T, TL, TR, or X edit descriptor, positions skipped and not previously  
27 filled are filled with blanks. The result is as if the entire record were initially filled with blanks.

28 On output, a character in the record may be replaced. However, a T, TL, TR, or X edit descriptor never  
29 directly causes a character already placed in the record to be replaced. Such edit descriptors may result  
30 in positioning such that subsequent editing causes a replacement.

#### 31 10.7.1.1 T, TL, and TR editing

32 The **left tab limit** affects file positioning by the T and TL edit descriptors. Immediately prior to data  
33 transfer, the left tab limit becomes defined as the character position of the current record or the current

1 position of the stream file. If, during data transfer, the file is positioned to another record, the left tab  
2 limit becomes defined as character position one of that record.

3 The  $T_n$  edit descriptor indicates that the transmission of the next character to or from a record is to  
4 occur at the  $n$ th character position of the record, relative to the left tab limit.

5 The  $TL_n$  edit descriptor indicates that the transmission of the next character to or from the record is  
6 to occur at the character position  $n$  characters backward from the current position. However, if  $n$  is  
7 greater than the difference between the current position and the left tab limit, the  $TL_n$  edit descriptor  
8 indicates that the transmission of the next character to or from the record is to occur at the left tab  
9 limit.

10 The  $TR_n$  edit descriptor indicates that the transmission of the next character to or from the record is  
11 to occur at the character position  $n$  characters forward from the current position.

#### NOTE 10.22

The  $n$  in a  $T_n$ ,  $TL_n$ , or  $TR_n$  edit descriptor shall be specified and shall be greater than zero.

### 12 10.7.1.2 X editing

13 The  $nX$  edit descriptor indicates that the transmission of the next character to or from a record is to  
14 occur at the position  $n$  characters forward from the current position.

#### NOTE 10.23

The  $n$  in an  $nX$  edit descriptor shall be specified and shall be greater than zero.

### 15 10.7.2 Slash editing

16 The slash edit descriptor indicates the end of data transfer to or from the current record.

17 On input from a file connected for sequential or stream access, the remaining portion of the current  
18 record is skipped and the file is positioned at the beginning of the next record. This record becomes  
19 the current record. On output to a file connected for sequential or stream access, a new empty record  
20 is created following the current record; this new record then becomes the last and current record of the  
21 file and the file is positioned at the beginning of this new record.

22 For a file connected for direct access, the record number is increased by one and the file is positioned  
23 at the beginning of the record that has that record number, if there is such a record, and this record  
24 becomes the current record.

#### NOTE 10.24

A record that contains no characters may be written on output. If the file is an internal file or a  
file connected for direct access, the record is filled with blank characters.

An entire record may be skipped on input.

25 The repeat specification is optional in the slash edit descriptor. If it is not specified, the default value is  
26 one.

### 27 10.7.3 Colon editing

28 The colon edit descriptor terminates format control if there are no more effective items in the in-  
29 put/output list (9.5.2). The colon edit descriptor has no effect if there are more effective items in the  
30 input/output list.

#### 1 **10.7.4 SS, SP, and S editing**

2 The SS, SP, and S edit descriptors temporarily change (9.4.1) the sign mode (9.4.5.14, 9.5.1.13) for the  
3 connection. The edit descriptors SS, SP, and S set the sign mode corresponding to the SIGN= specifier  
4 values SUPPRESS, PLUS, and PROCESSOR\_DEFINED, respectively.

5 The sign mode controls optional plus characters in numeric output fields. When the sign mode is PLUS,  
6 the processor shall produce a plus sign in any position that normally contains an optional plus sign.  
7 When the sign mode is SUPPRESS, the processor shall not produce a plus sign in such positions. When  
8 the sign mode is PROCESSOR\_DEFINED, the processor has the option of producing a plus sign or not  
9 in such positions, subject to 10.6.1(5).

10 The SS, SP, and S edit descriptors affect only I, F, E, EN, ES, D, and G editing during the execution of  
11 an output statement. The SS, SP, and S edit descriptors have no effect during the execution of an input  
12 statement.

#### 13 **10.7.5 P editing**

14 The  $k$ P edit descriptor temporarily changes (9.4.1) the scale factor for the connection to  $k$ . The scale  
15 factor affects the editing of F, E, EN, ES, D, and G edit descriptors for numeric quantities.

16 The scale factor  $k$  affects the appropriate editing in the following manner:

- 17 (1) On input, with F, E, EN, ES, D, and G editing (provided that no exponent exists in the  
18 field) and F output editing, the scale factor effect is that the externally represented number  
19 equals the internally represented number multiplied by  $10^k$ .
- 20 (2) On input, with F, E, EN, ES, D, and G editing, the scale factor has no effect if there is an  
21 exponent in the field.
- 22 (3) On output, with E and D editing, the significand (R418) part of the quantity to be produced  
23 is multiplied by  $10^k$  and the exponent is reduced by  $k$ .
- 24 (4) On output, with G editing, the effect of the scale factor is suspended unless the magnitude  
25 of the datum to be edited is outside the range that permits the use of F editing. If the use  
26 of E editing is required, the scale factor has the same effect as with E output editing.
- 27 (5) On output, with EN and ES editing, the scale factor has no effect.

28 If UP, DOWN, ZERO, or NEAREST I/O rounding mode is in effect then

- 29 (1) On input, the scale factor is applied to the external decimal value and then this is converted  
30 using the current I/O rounding mode, and
- 31 (2) On output, the internal value is converted using the current I/O rounding mode and then  
32 the scale factor is applied to the converted decimal value.

#### 33 **10.7.6 BN and BZ editing**

34 The BN and BZ edit descriptors temporarily change (9.4.1) the blank interpretation mode (9.4.5.4,  
35 9.5.1.5) for the connection. The edit descriptors BN and BZ set the blank interpretation mode corre-  
36 sponding to the BLANK= specifier values NULL and ZERO, respectively.

37 The blank interpretation mode controls the interpretation of nonleading blanks in numeric input fields.  
38 Such blank characters are interpreted as zeros when the blank interpretation mode has the value ZERO;  
39 they are ignored when the blank interpretation mode has the value NULL. The effect of ignoring blanks  
40 is to treat the input field as if blanks had been removed, the remaining portion of the field right justified,  
41 and the blanks replaced as leading blanks. However, a field containing only blanks has the value zero.

42 The blank interpretation mode affects only numeric editing (10.6.1) and generalized numeric editing  
43 (10.6.4.1) on input. It has no effect on output.

### 1 10.7.7 RU, RD, RZ, RN, RC, and RP editing

2 The round edit descriptors temporarily change (9.4.1) the connection's I/O rounding mode (9.4.5.13,  
3 9.5.1.12, 10.6.1.2.6). The round edit descriptors RU, RD, RZ, RN, RC, and RP set the I/O rounding  
4 mode corresponding to the ROUND= specifier values UP, DOWN, ZERO, NEAREST, COMPATIBLE,  
5 and PROCESSOR\_DEFINED, respectively. The I/O rounding mode affects the conversion of real and  
6 complex values in formatted input/output. It affects only D, E, EN, ES, F, and G editing.

### 7 10.7.8 DC and DP editing

8 The decimal edit descriptors temporarily change (9.4.1) the decimal edit mode (9.4.5.5, 9.5.1.6) for the  
9 connection. The edit descriptors DC and DP set the decimal edit mode corresponding to the DECIMAL=  
10 specifier values COMMA and POINT, respectively.

11 The decimal edit mode controls the representation of the decimal symbol (10.5) during conversion of  
12 real and complex values in formatted input/output. The decimal edit mode affects only D, E, EN, ES,  
13 F, and G editing. If the mode is COMMA during list-directed input/output, the character used as a  
14 value separator is a semicolon in place of a comma.

## 15 10.8 Character string edit descriptors

16 A character string edit descriptor shall not be used on input.

17 The character string edit descriptor causes characters to be written from the enclosed characters of the  
18 edit descriptor itself, including blanks. For a character string edit descriptor, the width of the field is  
19 the number of characters between the delimiting characters. Within the field, two consecutive delimiting  
20 characters are counted as a single character.

#### NOTE 10.25

A delimiter for a character string edit descriptor is either an apostrophe or quote.

## 21 10.9 List-directed formatting

22 List-directed input/output allows data editing according to the type of the list item instead of by a  
23 format specification. It also allows data to be free-field, that is, separated by commas (or semicolons)  
24 or blanks.

25 The characters in one or more list-directed records constitute a sequence of values and value separators.  
26 The end of a record has the same effect as a blank character, unless it is within a character constant. Any  
27 sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character  
28 constant.

29 Each value is either a null value or one of the forms:

30 
$$\begin{array}{l} c \\ r*c \\ r* \end{array}$$

31 where  $c$  is a literal constant, optionally signed if integer or real, or an undelimited character constant  
32 and  $r$  is an unsigned, nonzero, integer literal constant. Neither  $c$  nor  $r$  shall have kind type parameters  
33 specified. The constant  $c$  is interpreted as though it had the same kind type parameter as the corre-  
34 sponding list item. The  $r*c$  form is equivalent to  $r$  successive appearances of the constant  $c$ , and the  
35  $r*$  form is equivalent to  $r$  successive appearances of the null value. Neither of these forms may contain  
36 embedded blanks, except where permitted within the constant  $c$ .

1 A value separator is

- 2 (1) A comma optionally preceded by one or more contiguous blanks and optionally followed by  
 3 one or more contiguous blanks, unless the decimal edit mode is COMMA, in which case a  
 4 semicolon is used in place of the comma,
- 5 (2) A slash optionally preceded by one or more contiguous blanks and optionally followed by  
 6 one or more contiguous blanks, or
- 7 (3) One or more contiguous blanks between two nonblank values or following the last nonblank  
 8 value, where a nonblank value is a constant, an  $r^*c$  form, or an  $r^*$  form.

**NOTE 10.26**

Although a slash encountered in an input record is referred to as a separator, it actually causes termination of list-directed and namelist input statements; it does not actually separate two values.

**NOTE 10.27**

If no list items are specified in a list-directed input/output statement, one input record is skipped or one empty output record is written.

9 **10.9.1 List-directed input**

10 Input forms acceptable to edit descriptors for a given type are acceptable for list-directed formatting,  
 11 except as noted below. The form of the input value shall be acceptable for the type of the next effective  
 12 item in the list. Blanks are never used as zeros, and embedded blanks are not permitted in constants,  
 13 except within character constants and complex constants as specified below.

14 For the  $r^*c$  form of an input value, the constant  $c$  is interpreted as an undelimited character constant  
 15 if the first list item corresponding to this value is of type default, ASCII, or ISO 10646 character, there  
 16 is a nonblank character immediately after  $r^*$ , and that character is not an apostrophe or a quotation  
 17 mark; otherwise,  $c$  is interpreted as a literal constant.

**NOTE 10.28**

The end of a record has the effect of a blank, except when it appears within a character constant.

18 When the next effective item is of type integer, the value in the input record is interpreted as if an  $Iw$   
 19 edit descriptor with a suitable value of  $w$  were used.

20 When the next effective item is of type real, the input form is that of a numeric input field. A numeric  
 21 input field is a field suitable for F editing (10.6.1.2.1) that is assumed to have no fractional digits unless  
 22 a decimal symbol appears within the field.

23 When the next effective item is of type complex, the input form consists of a left parenthesis followed  
 24 by an ordered pair of numeric input fields separated by a separator, and followed by a right parenthesis.  
 25 The first numeric input field is the real part of the complex constant and the second is the imaginary  
 26 part. Each of the numeric input fields may be preceded or followed by any number of blanks and ends of  
 27 records. The separator is a comma if the decimal edit mode is POINT; it is a semicolon if the decimal  
 28 edit mode is COMMA. The end of a record may occur between the real part and the separator or between  
 29 the separator and the imaginary part.

30 When the next effective item is of type logical, the input form shall not include value separators among  
 31 the optional characters permitted for L editing.

32 When the next effective item is of type character, the input form consists of a possibly delimited sequence  
 33 of zero or more *rep-chars* whose kind type parameter is implied by the kind of the effective list item.  
 34 Character sequences may be continued from the end of one record to the beginning of the next record,

1 but the end of record shall not occur between a doubled apostrophe in an apostrophe-delimited character  
2 sequence, nor between a doubled quote in a quote-delimited character sequence. The end of the record  
3 does not cause a blank or any other character to become part of the character sequence. The character  
4 sequence may be continued on as many records as needed. The characters blank, comma, semicolon,  
5 and slash may appear in default, ASCII, or ISO 10646 character sequences.

6 If the next effective item is of type default, ASCII, or ISO 10646 character and

- 7 (1) The character sequence does not contain value separators,
- 8 (2) The character sequence does not cross a record boundary,
- 9 (3) The first nonblank character is not a quotation mark or an apostrophe,
- 10 (4) The leading characters are not *digits* followed by an asterisk, and
- 11 (5) The character sequence contains at least one character,

12 the delimiting apostrophes or quotation marks are not required. If the delimiters are omitted, the char-  
13 acter sequence is terminated by the first blank, comma, slash, or end of record; in this case apostrophes  
14 and quotation marks within the datum are not to be doubled.

15 Let  $len$  be the length of the next effective item, and let  $w$  be the length of the character sequence. If  
16  $len$  is less than or equal to  $w$ , the leftmost  $len$  characters of the sequence are transmitted to the next  
17 effective item. If  $len$  is greater than  $w$ , the sequence is transmitted to the leftmost  $w$  characters of the  
18 next effective item and the remaining  $len-w$  characters of the next effective item are filled with blanks.  
19 The effect is as though the sequence were assigned to the next effective item in a character assignment  
20 statement (7.4.1.3).

#### 21 10.9.1.1 Null values

22 A null value is specified by

- 23 (1) The  $r^*$  form,
- 24 (2) No characters between consecutive value separators, or
- 25 (3) No characters before the first value separator in the first record read by each execution of a  
26 list-directed input statement.

#### NOTE 10.29

The end of a record following any other value separator, with or without separating blanks, does not specify a null value in list-directed input.

27 A null value has no effect on the definition status of the next effective item. A null value shall not be  
28 used for either the real or imaginary part of a complex constant, but a single null value may represent  
29 an entire complex constant.

30 A slash encountered as a value separator during execution of a list-directed input statement causes  
31 termination of execution of that input statement after the assignment of the previous value. Any  
32 characters remaining in the current record are ignored. If there are additional items in the input list,  
33 the effect is as if null values had been supplied for them. Any *do-variable* in the input list is defined as  
34 though enough null values had been supplied for any remaining input list items.

#### NOTE 10.30

All blanks in a list-directed input record are considered to be part of some value separator except for the following:

- (1) Blanks embedded in a character sequence
- (2) Embedded blanks surrounding the real or imaginary part of a complex constant

**NOTE 10.30 (cont.)**

- (3) Leading blanks in the first record read by each execution of a list-directed input statement, unless immediately followed by a slash or comma

**NOTE 10.31**

List-directed input example:

```
INTEGER I; REAL X (8); CHARACTER (11) P;
COMPLEX Z; LOGICAL G
...
READ *, I, X, P, Z, G
...
```

The input data records are:

```
12345,12345,,2*1.5,4*
ISN'T_BOB'S,(123,0),.TEXAS$
```

The results are:

Variable	Value
I	12345
X (1)	12345.0
X (2)	unchanged
X (3)	1.5
X (4)	1.5
X (5) – X (8)	unchanged
P	ISN'T_BOB'S
Z	(123.0,0.0)
G	true

## 1 10.9.2 List-directed output

- 2 The form of the values produced is the same as that required for input, except as noted otherwise. With  
 3 the exception of adjacent undelimited character sequences, the values are separated by one or more  
 4 blanks or by a comma, or a semicolon if the decimal edit mode is comma, optionally preceded by one or  
 5 more blanks and optionally followed by one or more blanks.
- 6 The processor may begin new records as necessary, but the end of record shall not occur within a  
 7 constant except for complex constants and character sequences. The processor shall not insert blanks  
 8 within character sequences or within constants, except for complex constants.
- 9 Logical output values are T for the value true and F for the value false.
- 10 Integer output constants are produced with the effect of an *Iw* edit descriptor.
- 11 Real constants are produced with the effect of either an *F* edit descriptor or an *E* edit descriptor,  
 12 depending on the magnitude  $x$  of the value and a range  $10^{d_1} \leq x < 10^{d_2}$ , where  $d_1$  and  $d_2$  are processor-  
 13 dependent integers. If the magnitude  $x$  is within this range or is zero, the constant is produced using  
 14 *0PFw.d*; otherwise, *1PEw.dEe* is used.
- 15 For numeric output, reasonable processor-dependent values of  $w$ ,  $d$ , and  $e$  are used for each of the  
 16 numeric constants output.

1 Complex constants are enclosed in parentheses with a separator between the real and imaginary parts,  
 2 each produced as defined above for real constants. The separator is a comma if the decimal edit mode is  
 3 POINT; it is a semicolon if the decimal edit mode is COMMA. The end of a record may occur between  
 4 the separator and the imaginary part only if the entire constant is as long as, or longer than, an entire  
 5 record. The only embedded blanks permitted within a complex constant are between the separator and  
 6 the end of a record and one blank at the beginning of the next record.

7 Character sequences produced when the delimiter mode has a value of NONE

- 8 (1) Are not delimited by apostrophes or quotation marks,
- 9 (2) Are not separated from each other by value separators,
- 10 (3) Have each internal apostrophe or quotation mark represented externally by one apostrophe  
 11 or quotation mark, and
- 12 (4) Have a blank character inserted by the processor at the beginning of any record that begins  
 13 with the continuation of a character sequence from the preceding record.

14 Character sequences produced when the delimiter mode has a value of QUOTE are delimited by quotes,  
 15 are preceded and followed by a value separator, and have each internal quote represented on the external  
 16 medium by two contiguous quotes.

17 Character sequences produced when the delimiter mode has a value of APOSTROPHE are delimited  
 18 by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe  
 19 represented on the external medium by two contiguous apostrophes.

20 If two or more successive values in an output record have identical values, the processor has the option  
 21 of producing a repeated constant of the form  $r*c$  instead of the sequence of identical values.

22 Slashes, as value separators, and null values are not produced as output by list-directed formatting.

23 Except for continuation of delimited character sequences, each output record begins with a blank char-  
 24 acter.

#### NOTE 10.32

The length of the output records is not specified exactly and may be processor dependent.

## 25 10.10 Namelist formatting

26 Namelist input/output allows data editing with NAME=value subsequences. This facilitates documen-  
 27 tation of input and output files and more flexibility on input.

28 The characters in one or more namelist records constitute a sequence of **name-value subsequences**,  
 29 each of which consists of an object designator followed by an equals and followed by one or more values  
 30 and value separators. The equals may optionally be preceded or followed by one or more contiguous  
 31 blanks. The end of a record has the same effect as a blank character, unless it is within a character  
 32 constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within  
 33 a character constant.

34 The name may be any name in the *namelist-group-object-list* (5.4).

35 Each value is either a null value (10.10.1.4) or one of the forms

36 
$$\begin{array}{l} c \\ r*c \\ r* \end{array}$$

37 where  $c$  is a literal constant, optionally signed if integer or real, and  $r$  is an unsigned, nonzero, integer

1 literal constant. Neither  $c$  nor  $r$  may have kind type parameters specified. The constant  $c$  is interpreted  
2 as though it had the same kind type parameter as the corresponding list item. The  $r^*c$  form is equivalent  
3 to  $r$  successive appearances of the constant  $c$ , and the  $r^*$  form is equivalent to  $r$  successive null values.  
4 Neither of these forms may contain embedded blanks, except where permitted within the constant  $c$ .

5 A value separator for namelist formatting is the same as for list-directed formatting (10.9).

### 6 10.10.1 Namelist input

7 Input for a namelist input statement consists of

- 8 (1) Optional blanks and namelist comments,
- 9 (2) The character & followed immediately by the *namelist-group-name* as specified in the  
10 NAMELIST statement,
- 11 (3) One or more blanks,
- 12 (4) A sequence of zero or more name-value subsequences separated by value separators, and
- 13 (5) A slash to terminate the namelist input.

#### NOTE 10.33

A slash encountered in a namelist input record causes the input statement to terminate. A slash cannot be used to separate two values in a namelist input statement.
--

14 In each name-value subsequence, the name shall be the name of a namelist group object list item with  
15 an optional qualification and the name with the optional qualification shall not be a zero-sized array, a  
16 zero-sized array section, or a zero-length character string. The optional qualification, if any, shall not  
17 contain a vector subscript.

18 A group name or object name is without regard to case.

#### 19 10.10.1.1 Namelist group object names

20 Within the input data, each name shall correspond to a particular namelist group object name. Sub-  
21 scripts, strides, and substring range expressions used to qualify group object names shall be optionally  
22 signed integer literal constants with no kind type parameters specified. If a namelist group object is  
23 an array, the input record corresponding to it may contain either the array name or the designator of  
24 a subobject of that array, using the syntax of object designators (R603). If the namelist group object  
25 name is the name of a variable of derived type, the name in the input record may be either the name of  
26 the variable or the designator of one of its components, indicated by qualifying the variable name with  
27 the appropriate component name. Successive qualifications may be applied as appropriate to the shape  
28 and type of the variable represented.

29 The order of names in the input records need not match the order of the namelist group object items.  
30 The input records need not contain all the names of the namelist group object items. The definition  
31 status of any names from the *namelist-group-object-list* that do not occur in the input record remains  
32 unchanged. In the input record, each object name or subobject designator may be preceded and followed  
33 by one or more optional blanks but shall not contain embedded blanks.

#### 34 10.10.1.2 Namelist input values

35 The datum  $c$  (10.10) is any input value acceptable to format specifications for a given type, except for a  
36 restriction on the form of input values corresponding to list items of types logical, integer, and character  
37 as specified in 10.10.1.3. The form of a real or complex value is dependent on the decimal edit mode  
38 in effect (10.7.8). The form of an input value shall be acceptable for the type of the namelist group  
39 object list item. The number and forms of the input values that may follow the equals in a name-value

- 1 subsequence depend on the shape and type of the object represented by the name in the input record.  
2 When the name in the input record is that of a scalar variable of an intrinsic type, the equals shall  
3 not be followed by more than one value. Blanks are never used as zeros, and embedded blanks are not  
4 permitted in constants except within character constants and complex constants as specified in 10.10.1.3.
- 5 The name-value subsequences are evaluated serially, in left-to-right order. A namelist group object  
6 designator may appear in more than one name-value sequence.
- 7 When the name in the input record represents an array variable or a variable of derived type, the effect  
8 is as if the variable represented were expanded into a sequence of scalar list items of intrinsic data types,  
9 in the same way that formatted input/output list items are expanded (9.5.2). Each input value following  
10 the equals shall then be acceptable to format specifications for the intrinsic type of the list item in the  
11 corresponding position in the expanded sequence, except as noted in 10.10.1.3. The number of values  
12 following the equals shall not exceed the number of list items in the expanded sequence, but may be less;  
13 in the latter case, the effect is as if sufficient null values had been appended to match any remaining list  
14 items in the expanded sequence.

**NOTE 10.34**

For example, if the name in the input record is the name of an integer array of size 100, at most 100 values, each of which is either a digit string or a null value, may follow the equals; these values would then be assigned to the elements of the array in array element order.

- 15 A slash encountered as a value separator during the execution of a namelist input statement causes  
16 termination of execution of that input statement after assignment of the previous value. If there are  
17 additional items in the namelist group object being transferred, the effect is as if null values had been  
18 supplied for them.
- 19 A namelist comment may appear after any value separator except a slash. A namelist comment is also  
20 permitted to start in the first nonblank position of an input record except within a character literal  
21 constant.
- 22 Successive namelist records are read by namelist input until a slash is encountered; the remainder of the  
23 record is ignored and need not follow the rules for namelist input values.

**24 10.10.1.3 Namelist group object list items**

- 25 When the next effective namelist group object list item is of type real, the input form of the input value  
26 is that of a numeric input field. A numeric input field is a field suitable for F editing (10.6.1.2.1) that is  
27 assumed to have no fractional digits unless a decimal symbol appears within the field.
- 28 When the next effective item is of type complex, the input form of the input value consists of a left  
29 parenthesis followed by an ordered pair of numeric input fields separated by a comma and followed by a  
30 right parenthesis. The first numeric input field is the real part of the complex constant and the second  
31 part is the imaginary part. Each of the numeric input fields may be preceded or followed by any number  
32 of blanks and ends of records. The end of a record may occur between the real part and the comma or  
33 between the comma and the imaginary part.
- 34 When the next effective item is of type logical, the input form of the input value shall not include equals  
35 or value separators among the optional characters permitted for L editing (10.6.2).
- 36 When the next effective item is of type integer, the value in the input record is interpreted as if an *Iw*  
37 edit descriptor with a suitable value of *w* were used.
- 38 When the next effective item is of type character, the input form consists of a delimited sequence of zero  
39 or more *rep-chars* whose kind type parameter is implied by the kind of the corresponding list item. Such  
40 a sequence may be continued from the end of one record to the beginning of the next record, but the

1 end of record shall not occur between a doubled apostrophe in an apostrophe-delimited sequence, nor  
 2 between a doubled quote in a quote-delimited sequence. The end of the record does not cause a blank  
 3 or any other character to become part of the sequence. The sequence may be continued on as many  
 4 records as needed. The characters blank, comma, and slash may appear in such character sequences.

**NOTE 10.35**

A character sequence corresponding to a namelist input item of character type shall be delimited either with apostrophes or with quotes. The delimiter is required to avoid ambiguity between undelimited character sequences and object names. The value of the DELIM= specifier, if any, in the OPEN statement for an external file is ignored during namelist input (9.4.5.6).

5 Let  $len$  be the length of the next effective item, and let  $w$  be the length of the character sequence. If  
 6  $len$  is less than or equal to  $w$ , the leftmost  $len$  characters of the sequence are transmitted to the next  
 7 effective item. If  $len$  is greater than  $w$ , the constant is transmitted to the leftmost  $w$  characters of the  
 8 next effective item and the remaining  $len-w$  characters of the next effective item are filled with blanks.  
 9 The effect is as though the sequence were assigned to the next effective item in a character assignment  
 10 statement (7.4.1.3).

11 **10.10.1.4 Null values**

12 A null value is specified by

- 13 (1) The  $r^*$  form,
- 14 (2) Blanks between two consecutive value separators following an equals,
- 15 (3) Zero or more blanks preceding the first value separator and following an equals, or
- 16 (4) Two consecutive nonblank value separators.

17 A null value has no effect on the definition status of the corresponding input list item. If the namelist  
 18 group object list item is defined, it retains its previous value; if it is undefined, it remains undefined. A  
 19 null value shall not be used as either the real or imaginary part of a complex constant, but a single null  
 20 value may represent an entire complex constant.

**NOTE 10.36**

The end of a record following a value separator, with or without intervening blanks, does not specify a null value in namelist input.

21 **10.10.1.5 Blanks**

22 All blanks in a namelist input record are considered to be part of some value separator except for

- 23 (1) Blanks embedded in a character constant,
- 24 (2) Embedded blanks surrounding the real or imaginary part of a complex constant,
- 25 (3) Leading blanks following the equals unless followed immediately by a slash or comma, or a  
 26 semicolon if the decimal edit mode is comma, and
- 27 (4) Blanks between a name and the following equals.

28 **10.10.1.6 Namelist Comments**

29 Except within a character literal constant, a “!” character after a value separator or in the first nonblank  
 30 position of a namelist input record initiates a comment. The comment extends to the end of the current  
 31 input record and may contain any graphic character in the processor-dependent character set. The  
 32 comment is ignored. A slash within the namelist comment does not terminate execution of the namelist  
 33 input statement. Namelist comments are not allowed in stream input because comments depend on

1 record structure.

**NOTE 10.37**

Namelist input example:

```
INTEGER I; REAL X (8); CHARACTER (11) P; COMPLEX Z;
LOGICAL G
NAMELIST / TODAY / G, I, P, Z, X
READ (*, NML = TODAY)
```

The input data records are:

```
&TODAY I = 12345, X(1) = 12345, X(3:4) = 2*1.5, I=6, ! This is a comment.
P = 'ISN'T_BOB'S', Z = (123,0)/
```

The results stored are:

Variable	Value
I	6
X (1)	12345.0
X (2)	unchanged
X (3)	1.5
X (4)	1.5
X (5) – X (8)	unchanged
P	ISN'T_BOB'S
Z	(123.0,0.0)
G	unchanged

2 **10.10.2 Namelist output**

3 The form of the output produced is the same as that required for input, except for the forms of real,  
4 character, and logical values. The name in the output is in upper case. With the exception of adjacent  
5 undelimited character values, the values are separated by one or more blanks or by a comma, or a  
6 semicolon if the decimal edit mode is COMMA, optionally preceded by one or more blanks and optionally  
7 followed by one or more blanks.

8 Namelist output shall not include namelist comments.

9 The processor may begin new records as necessary. However, except for complex constants and character  
10 values, the end of a record shall not occur within a constant, character value, or name, and blanks shall  
11 not appear within a constant, character value, or name.

**NOTE 10.38**

The length of the output records is not specified exactly and may be processor dependent.

12 **10.10.2.1 Namelist output editing**

13 Logical output values are T for the value true and F for the value false.

14 Integer output constants are produced with the effect of an *Iw* edit descriptor.

15 Real constants are produced with the effect of either an F edit descriptor or an E edit descriptor,  
16 depending on the magnitude  $x$  of the value and a range  $10^{d_1} \leq x < 10^{d_2}$ , where  $d_1$  and  $d_2$  are processor-  
17 dependent integers. If the magnitude  $x$  is within this range or is zero, the constant is produced using

- 1 0PF*w.d*; otherwise, 1PE*w.d Ee* is used.
- 2 For numeric output, reasonable processor-dependent integer values of *w*, *d*, and *e* are used for each of  
3 the numeric constants output.
- 4 Complex constants are enclosed in parentheses with a separator between the real and imaginary parts,  
5 each produced as defined above for real constants. The separator is a comma if the decimal edit mode is  
6 POINT; it is a semicolon if the decimal edit mode is COMMA. The end of a record may occur between  
7 the separator and the imaginary part only if the entire constant is as long as, or longer than, an entire  
8 record. The only embedded blanks permitted within a complex constant are between the separator and  
9 the end of a record and one blank at the beginning of the next record.
- 10 Character sequences produced when the delimiter mode has a value of NONE
- 11 (1) Are not delimited by apostrophes or quotation marks,
  - 12 (2) Are not separated from each other by value separators,
  - 13 (3) Have each internal apostrophe or quotation mark represented externally by one apostrophe  
14 or quotation mark, and
  - 15 (4) Have a blank character inserted by the processor at the beginning of any record that begins  
16 with the continuation of a character sequence from the preceding record.

**NOTE 10.39**

Namelist output records produced with a DELIM= specifier with a value of NONE and which contain a character sequence might not be acceptable as namelist input records.

- 17 Character sequences produced when the delimiter mode has a value of QUOTE are delimited by quotes,  
18 are preceded and followed by a value separator, and have each internal quote represented on the external  
19 medium by two contiguous quotes.
- 20 Character sequences produced when the delimiter mode has a value of APOSTROPHE are delimited  
21 by apostrophes, are preceded and followed by a value separator, and have each internal apostrophe  
22 represented on the external medium by two contiguous apostrophes.
- 23 **10.10.2.2 Namelist output records**
- 24 If two or more successive values in an array in an output record produced have identical values, the  
25 processor has the option of producing a repeated constant of the form  $r*c$  instead of the sequence of  
26 identical values.
- 27 The name of each namelist group object list item is placed in the output record followed by an equals  
28 and a list of values of the namelist group object list item.
- 29 An ampersand character followed immediately by a *namelist-group-name* will be produced by namelist  
30 formatting at the start of the first output record to indicate which particular group of data objects is  
31 being output. A slash is produced by namelist formatting to indicate the end of the namelist formatting.
- 32 A null value is not produced by namelist formatting.
- 33 Except for continuation of delimited character sequences, each output record begins with a blank char-  
34 acter.



## 1 Section 11: Program units

2 The terms and basic concepts of program units were introduced in 2.2. A program unit is a main  
3 program, an external subprogram, a module, or a block data program unit.

4 This section describes main programs, modules, and block data program units. Section 12 describes  
5 external subprograms.

### 6 11.1 Main program

7 A Fortran **main program** is a program unit that does not contain a SUBROUTINE, FUNCTION,  
8 MODULE, or BLOCK DATA statement as its first statement.

```

9 R1101 main-program           is [ program-stmt ]
10                                [ specification-part ]
11                                [ execution-part ]
12                                [ internal-subprogram-part ]
13                                end-program-stmt
14 R1102 program-stmt         is PROGRAM program-name
15 R1103 end-program-stmt     is END [ PROGRAM [ program-name ] ]
```

16 C1101 (R1101) In a *main-program*, the *execution-part* shall not contain a RETURN statement or an  
17 ENTRY statement.

18 C1102 (R1101) The *program-name* may be included in the *end-program-stmt* only if the optional  
19 *program-stmt* is used and, if included, shall be identical to the *program-name* specified in the  
20 *program-stmt*.

21 C1103 (R1101) An automatic object shall not appear in the *specification-part* (R204) of a main program.

#### NOTE 11.1

The program name is global to the program (16.1). For explanatory information about uses for the program name, see section C.8.1.

#### NOTE 11.2

An example of a main program is:

```

PROGRAM ANALYZE
  REAL A, B, C (10,10)      ! Specification part
  CALL FIND                 ! Execution part
CONTAINS
  SUBROUTINE FIND          ! Internal subprogram
  ...
  END SUBROUTINE FIND
END PROGRAM ANALYZE
```

22 The main program may be defined by means other than Fortran; in that case, the program shall not  
23 contain a *main-program* program unit.

24 A reference to a Fortran *main-program* shall not appear in any program unit in the program, including

1 itself.

## 2 11.2 Modules

3 A **module** contains specifications and definitions that are to be accessible to other program units. A  
 4 module that is provided as an inherent part of the processor is an **intrinsic module**. A **nonintrinsic**  
 5 **module** is defined by a module program unit or a means other than Fortran.

6 Procedures and types defined in an intrinsic module are not themselves intrinsic.

7	R1104	<i>module</i>	<b>is</b>	<i>module-stmt</i>
8				[ <i>specification-part</i> ]
9				[ <i>module-subprogram-part</i> ]
10				<i>end-module-stmt</i>
11	R1105	<i>module-stmt</i>	<b>is</b>	MODULE <i>module-name</i>
12	R1106	<i>end-module-stmt</i>	<b>is</b>	END [ MODULE [ <i>module-name</i> ] ]
13	R1107	<i>module-subprogram-part</i>	<b>is</b>	<i>contains-stmt</i>
14				<i>module-subprogram</i>
15				[ <i>module-subprogram</i> ] ...
16	R1108	<i>module-subprogram</i>	<b>is</b>	<i>function-subprogram</i>
17			<b>or</b>	<i>subroutine-subprogram</i>

18 C1104 (R1104) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the  
 19 *module-name* specified in the *module-stmt*.

20 C1105 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a  
 21 *format-stmt*.

22 C1106 (R1104) An automatic object shall not appear in the *specification-part* of a module.

23 C1107 (R1104) If an object of a type for which *component-initialization* is specified (R444) appears  
 24 in the *specification-part* of a module and does not have the ALLOCATABLE or POINTER  
 25 attribute, the object shall have the SAVE attribute.

### NOTE 11.3

The module name is global to the program (16.1).

### NOTE 11.4

Although statement function definitions, ENTRY statements, and FORMAT statements shall not appear in the specification part of a module, they may appear in the specification part of a module subprogram in the module.

A module is host to any module subprograms (12.1.2.2) it contains, and the entities in the module are therefore accessible in the module subprograms through host association.

### NOTE 11.5

For a discussion of the impact of modules on dependent compilation, see section C.8.2.

### NOTE 11.6

For examples of the use of modules, see section C.8.3.

26 If a procedure declared in the scoping unit of a module has an implicit interface, it shall be given the

1 EXTERNAL attribute in that scoping unit; if it is a function, its type and type parameters shall be  
2 explicitly declared in a type declaration statement in that scoping unit.

3 If an intrinsic procedure is declared in the scoping unit of a module, it shall explicitly be given the  
4 INTRINSIC attribute in that scoping unit or be used as an intrinsic procedure in that scoping unit.

### 5 11.2.1 The USE statement and use association

6 The **USE statement** specifies use association. A USE statement is a **module reference** to the module  
7 it specifies. At the time a USE statement is processed, the public portions of the specified module shall  
8 be available. A module shall not reference itself, either directly or indirectly.

9 The **USE statement** provides the means by which a scoping unit accesses named data objects, derived  
10 types, interface blocks, procedures, abstract interfaces, generic identifiers (12.3.2.1), and namelist groups  
11 in a module. The entities in the scoping unit are said to be **use associated** with the entities in the  
12 module. The accessed entities have the attributes specified in the module. The entities made accessible  
13 are identified by the names or generic identifiers used to identify them in the module. By default, they  
14 are identified by the same identifiers in the scoping unit containing the USE statement, but it is possible  
15 to specify that different local identifiers are used.

#### NOTE 11.7

The accessibility of module entities may be controlled by accessibility attributes (4.5.1.1, 5.1.2.1), and the ONLY option of the USE statement. Definability of module entities can be controlled by the PROTECTED attribute (5.1.2.12).

16	R1109	<i>use-stmt</i>	<b>is</b> USE [ [ , <i>module-nature</i> ] :: ] <i>module-name</i> [ , <i>rename-list</i> ]
17			<b>or</b> USE [ [ , <i>module-nature</i> ] :: ] <i>module-name</i> , ■
18			■ ONLY : [ <i>only-list</i> ]
19	R1110	<i>module-nature</i>	<b>is</b> INTRINSIC
20			<b>or</b> NON_INTRINSIC
21	R1111	<i>rename</i>	<b>is</b> <i>local-name</i> => <i>use-name</i>
22			<b>or</b> OPERATOR ( <i>local-defined-operator</i> ) => ■
23			■ OPERATOR ( <i>use-defined-operator</i> )
24	R1112	<i>only</i>	<b>is</b> <i>generic-spec</i>
25			<b>or</b> <i>only-use-name</i>
26			<b>or</b> <i>rename</i>
27	R1113	<i>only-use-name</i>	<b>is</b> <i>use-name</i>

28 C1108 (R1109) If *module-nature* is INTRINSIC, *module-name* shall be the name of an intrinsic module.

29 C1109 (R1109) If *module-nature* is NON\_INTRINSIC, *module-name* shall be the name of a nonintrinsic  
30 module.

31 C1110 (R1109) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the  
32 same name.

33 C1111 (R1111) OPERATOR(*use-defined-operator*) shall not identify a *generic-binding*.

34 C1112 (R1112) The *generic-spec* shall not identify a *generic-binding*.



1 The appearance of such a local identifier in a PUBLIC statement in a module causes the entity accessible  
 2 by the USE statement to be a public entity of that module. If the identifier appears in a PRIVATE  
 3 statement in a module, the entity is not a public entity of that module. If the local identifier does not  
 4 appear in either a PUBLIC or PRIVATE statement, it assumes the default accessibility attribute (5.2.1)  
 5 of that scoping unit.

**NOTE 11.10**

The constraints in sections 5.5.1, 5.5.2, and 5.4 prohibit the *local-name* from appearing as a *common-block-object* in a COMMON statement, an *equivalence-object* in an EQUIVALENCE statement, or a *namelist-group-name* in a NAMELIST statement, respectively. There is no prohibition against the *local-name* appearing as a *common-block-name* or a *namelist-group-object*.

**NOTE 11.11**

For a discussion of the impact of the ONLY clause and renaming on dependent compilation, see section C.8.2.1.

**NOTE 11.12**

Examples:

```
USE STATS_LIB
```

provides access to all public entities in the module STATS\_LIB.

```
USE MATH_LIB; USE STATS_LIB, SPROD => PROD
```

makes all public entities in both MATH\_LIB and STATS\_LIB accessible. If MATH\_LIB contains an entity called PROD, it is accessible by its own name while the entity PROD of STATS\_LIB is accessible by the name SPROD.

```
USE STATS_LIB, ONLY: YPROD; USE STATS_LIB, ONLY : PROD
```

makes public entities YPROD and PROD in STATS\_LIB accessible.

```
USE STATS_LIB, ONLY : YPROD; USE STATS_LIB
```

makes all public entities in STATS\_LIB accessible.

## 6 11.3 Block data program units

7 A **block data program unit** is used to provide initial values for data objects in named common blocks.

8 R1116 *block-data* is *block-data-stmt*

9 [ *specification-part* ]

10 *end-block-data-stmt*

11 R1117 *block-data-stmt* is BLOCK DATA [ *block-data-name* ]

12 R1118 *end-block-data-stmt* is END [ BLOCK DATA [ *block-data-name* ] ]

13 C1116 (R1116) The *block-data-name* shall be included in the *end-block-data-stmt* only if it was provided  
 14 in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-*

1 *data-stmt.*

2 C1117 (R1116) A *block-data specification-part* shall contain only derived-type definitions and ASYN-  
3 CHRONOUS, BIND, COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, INTRIN-  
4 SIC, PARAMETER, POINTER, SAVE, TARGET, USE, VOLATILE, and type declaration  
5 statements.

6 C1118 (R1116) A type declaration statement in a *block-data specification-part* shall not contain AL-  
7 LOCATABLE, EXTERNAL, or BIND attribute specifiers.

**NOTE 11.13**

For explanatory information about the uses for the *block-data-name*, see section C.8.1.

8 If an object in a named common block is initially defined, all storage units in the common block storage  
9 sequence shall be specified even if they are not all initially defined. More than one named common block  
10 may have objects initially defined in a single block data program unit.

**NOTE 11.14**

In the example

```
BLOCK DATA INIT
  REAL A, B, C, D, E, F
  COMMON /BLOCK1/ A, B, C, D
  DATA A /1.2/, C /2.3/
  COMMON /BLOCK2/ E, F
  DATA F /6.5/
END BLOCK DATA INIT
```

common blocks BLOCK1 and BLOCK2 both have objects that are being initialized in a single block data program unit. B, D, and E are not initialized but they need to be specified as part of the common blocks.

11 Only an object in a named common block may be initially defined in a block data program unit.

**NOTE 11.15**

Objects associated with an object in a common block are considered to be in that common block.

12 The same named common block shall not be specified in more than one block data program unit in a  
13 program.

14 There shall not be more than one unnamed block data program unit in a program.

**NOTE 11.16**

An example of a block data program unit is:

```
BLOCK DATA WORK
  COMMON /WRKCOM/ A, B, C (10, 10)
  REAL :: A, B, C
  DATA A /1.0/, B /2.0/, C /100 * 0.0/
END BLOCK DATA WORK
```

## 1 Section 12: Procedures

2 The concept of a procedure was introduced in 2.2.3. This section contains a complete description of  
3 procedures. The actions specified by a procedure are performed when the procedure is invoked by  
4 execution of a reference to it.

5 The sequence of actions encapsulated by a procedure has access to entities in the invoking scoping  
6 unit by way of argument association (12.4.1). A **dummy argument** is a name that appears in the  
7 SUBROUTINE, FUNCTION, or ENTRY statement in the declaration of a procedure (R1233). Dummy  
8 arguments are also specified for intrinsic procedures and procedures in intrinsic modules in Sections 13,  
9 14, and 15.

10 The entities in the invoking scoping unit are specified by actual arguments. An **actual argument** is an  
11 entity that appears in a procedure reference (R1221).

12 A procedure may also have access to entities in other scoping units, not necessarily the invoking scoping  
13 unit, by use association (16.4.1.2), host association (16.4.1.3), linkage association (16.4.1.4), storage  
14 association (16.4.3), or by reference to external procedures (5.1.2.6).

### 15 12.1 Procedure classifications

16 A procedure is classified according to the form of its reference and the way it is defined.

#### 17 12.1.1 Procedure classification by reference

18 The definition of a procedure specifies it to be a function or a subroutine. A reference to a function either  
19 appears explicitly as a primary within an expression, or is implied by a defined operation (7.1.3) within  
20 an expression. A reference to a subroutine is a CALL statement or a defined assignment statement  
21 (7.4.1.4).

22 A procedure is classified as **elemental** if it is a procedure that may be referenced elementally (12.7).

#### 23 12.1.2 Procedure classification by means of definition

24 A procedure is either an intrinsic procedure, an external procedure, a module procedure, an internal  
25 procedure, a dummy procedure (which may be a dummy procedure pointer), a nondummy procedure  
26 pointer, or a statement function.

##### 27 12.1.2.1 Intrinsic procedures

28 A procedure that is provided as an inherent part of the processor is an **intrinsic procedure**.

##### 29 12.1.2.2 External, internal, and module procedures

30 An **external procedure** is a procedure that is defined by an external subprogram or by a means other  
31 than Fortran.

32 An **internal procedure** is a procedure that is defined by an internal subprogram. Internal subprograms  
33 may appear in the main program, in an external subprogram, or in a module subprogram. Internal  
34 subprograms shall not appear in other internal subprograms. Internal subprograms are the same as  
35 external subprograms except that the name of the internal procedure is not a global identifier, an

1 internal subprogram shall not contain an ENTRY statement, the internal procedure name shall not be  
2 argument associated with a dummy procedure (12.4.1.3), and the internal subprogram has access to host  
3 entities by host association.

4 A **module procedure** is a procedure that is defined by a module subprogram.

5 A subprogram defines a procedure for the SUBROUTINE or FUNCTION statement. If the subprogram  
6 has one or more ENTRY statements, it also defines a procedure for each of them.

### 7 12.1.2.3 Dummy procedures

8 A dummy argument that is specified to be a procedure or appears in a procedure reference is a **dummy**  
9 **procedure**. A dummy procedure with the POINTER attribute is a dummy procedure pointer.

### 10 12.1.2.4 Procedure pointers

11 A procedure pointer is a procedure that has the EXTERNAL and POINTER attributes; it may be  
12 pointer associated with an external procedure, a module procedure, an intrinsic procedure, or a dummy  
13 procedure that is not a procedure pointer.

### 14 12.1.2.5 Statement functions

15 A function that is defined by a single statement is a **statement function** (12.5.4).

## 16 12.2 Characteristics of procedures

17 The **characteristics of a procedure** are the classification of the procedure as a function or subroutine,  
18 whether it is pure, whether it is elemental, whether it has the BIND attribute, the characteristics of its  
19 dummy arguments, and the characteristics of its result value if it is a function.

### 20 12.2.1 Characteristics of dummy arguments

21 Each dummy argument has the characteristic that it is a dummy data object, a dummy procedure, a  
22 dummy procedure pointer, or an asterisk (alternate return indicator). A dummy argument other than an asterisk  
23 may be specified to have the OPTIONAL attribute. This attribute means that the dummy argument  
24 need not be associated with an actual argument for any particular reference to the procedure.

#### 25 12.2.1.1 Characteristics of dummy data objects

26 The characteristics of a dummy data object are its type, its type parameters (if any), its shape, its  
27 intent (5.1.2.7, 5.2.7), whether it is optional (5.1.2.9, 5.2.8), whether it is allocatable (5.1.2.5.3), whether  
28 it has the VALUE (5.1.2.15), ASYNCHRONOUS (5.1.2.3), or VOLATILE (5.1.2.16) attributes, whether  
29 it is polymorphic, and whether it is a pointer (5.1.2.11, 5.2.10) or a target (5.1.2.14, 5.2.13). If a type  
30 parameter of an object or a bound of an array is not an initialization expression, the exact dependence  
31 on the entities in the expression is a characteristic. If a shape, size, or type parameter is assumed or  
32 deferred, it is a characteristic.

#### 33 12.2.1.2 Characteristics of dummy procedures and dummy procedure pointers

34 The characteristics of a dummy procedure are the explicitness of its interface (12.3.1), its characteristics  
35 as a procedure if the interface is explicit, whether it is a pointer, and whether it is optional (5.1.2.9, 5.2.8).

#### 36 12.2.1.3 Characteristics of asterisk dummy arguments

37 An asterisk as a dummy argument has no characteristics.

## 1 12.2.2 Characteristics of function results

2 The characteristics of a function result are its type, type parameters (if any), rank, whether it is poly-  
 3 morphic, whether it is allocatable, whether it is a pointer, and whether it is a procedure pointer. If a  
 4 function result is an array that is not allocatable or a pointer, its shape is a characteristic. If a type  
 5 parameter of a function result or a bound of a function result array is not an initialization expression, the  
 6 exact dependence on the entities in the expression is a characteristic. If type parameters of a function  
 7 result are deferred, which parameters are deferred is a characteristic. If the length of a character function  
 8 result is assumed, this is a characteristic.

## 9 12.3 Procedure interface

10 The **interface** of a procedure determines the forms of reference through which it may be invoked.  
 11 The procedure's interface consists of its abstract interface, its name, its binding label if any, and the  
 12 procedure's generic identifiers, if any. The characteristics of a procedure are fixed, but the remainder of  
 13 the interface may differ in different scoping units.

14 An **abstract interface** consists of procedure characteristics and the names of dummy arguments.

### NOTE 12.1

For more explanatory information on procedure interfaces, see [C.9.3](#).

## 15 12.3.1 Implicit and explicit interfaces

16 If a procedure is accessible in a scoping unit, its interface is either **explicit** or **implicit** in that scoping  
 17 unit. The interface of an internal procedure, module procedure, or intrinsic procedure is always explicit  
 18 in such a scoping unit. The interface of a subroutine or a function with a separate result name is explicit  
 19 within the subprogram that defines it. The interface of a statement function is always implicit. The interface of  
 20 an external procedure or dummy procedure is explicit in a scoping unit other than its own if an interface  
 21 body ([12.3.2.1](#)) for the procedure is supplied or accessible, and implicit otherwise.

### NOTE 12.2

For example, the subroutine LLS of [C.8.3.5](#) has an explicit interface.

### 22 12.3.1.1 Explicit interface

23 A procedure other than a statement function shall have an explicit interface if it is referenced and

- 24 (1) A reference to the procedure appears
- 25 (a) With an argument keyword ([12.4.1](#)),
  - 26 (b) As a reference by its generic name ([12.3.2.1](#)),
  - 27 (c) As a defined assignment (subroutines only),
  - 28 (d) In an expression as a defined operator (functions only), or
  - 29 (e) In a context that requires it to be pure,
- 30 (2) The procedure has a dummy argument that
- 31 (a) has the ALLOCATABLE, ASYNCHRONOUS, OPTIONAL, POINTER, TARGET,  
32 VALUE, or VOLATILE attribute,
  - 33 (b) is an assumed-shape array,
  - 34 (c) is of a parameterized derived type, or
  - 35 (d) is polymorphic,

- 1           (3) The procedure has a result that  
 2               (a) is an array,  
 3               (b) is a pointer or is allocatable, or  
 4               (c) has a nonassumed type parameter value that is not an initialization expression,  
 5           (4) The procedure is elemental, or  
 6           (5) The procedure has the BIND attribute.

### 7 12.3.2 Specification of the procedure interface

8 The interface for an internal, external, module, or dummy procedure is specified by a FUNCTION,  
 9 SUBROUTINE, or ENTRY statement and by specification statements for the dummy arguments and  
 10 the result of a function. These statements may appear in the procedure definition, in an interface body,  
 11 or both, except that the ENTRY statement shall not appear in an interface body.

#### NOTE 12.3

An interface body cannot be used to describe the interface of an internal procedure, a module procedure, or an intrinsic procedure because the interfaces of such procedures are already explicit. However, the name of a procedure may appear in a PROCEDURE statement in an interface block (12.3.2.1).

#### 12 12.3.2.1 Interface block

- |    |                                      |  |  |
|----|--------------------------------------|--|--|
| 13 | R1201 <i>interface-block</i>         | <b>is</b>  | <i>interface-stmt</i><br>[ <i>interface-specification</i> ] ...<br><i>end-interface-stmt</i> |
| 14 |                                      |  |  |
| 15 |                                      |  |  |
| 16 | R1202 <i>interface-specification</i> | <b>is</b>  | <i>interface-body</i>  |
| 17 |                                      | <b>or</b>  | <i>procedure-stmt</i>  |
| 18 | R1203 <i>interface-stmt</i>          | <b>is</b>  | INTERFACE [ <i>generic-spec</i> ]  |
| 19 |                                      | <b>or</b>  | ABSTRACT INTERFACE   |
| 20 | R1204 <i>end-interface-stmt</i>      | <b>is</b>  | END INTERFACE [ <i>generic-spec</i> ]  |
| 21 | R1205 <i>interface-body</i>          | <b>is</b>  | <i>function-stmt</i><br>[ <i>specification-part</i> ]<br><i>end-function-stmt</i>            |
| 22 |                                      |  |  |
| 23 |                                      |  |  |
| 24 |                                      | <b>or</b>  | <i>subroutine-stmt</i><br>[ <i>specification-part</i> ]<br><i>end-subroutine-stmt</i>        |
| 25 |                                      |  |  |
| 26 |                                      |  |  |
| 27 | R1206 <i>procedure-stmt</i>          | <b>is</b>  | [ MODULE ] PROCEDURE <i>procedure-name-list</i>  |
| 28 | R1207 <i>generic-spec</i>            | <b>is</b>  | <i>generic-name</i>  |
| 29 |                                      | <b>or</b>  | OPERATOR ( <i>defined-operator</i> )   |
| 30 |                                      | <b>or</b>  | ASSIGNMENT ( = )   |
| 31 |                                      | <b>or</b>  | <i>dtio-generic-spec</i>   |
| 32 | R1208 <i>dtio-generic-spec</i>       | <b>is</b>  | READ (FORMATTED)   |
| 33 |                                      | <b>or</b>  | READ (UNFORMATTED)   |
| 34 |                                      | <b>or</b>  | WRITE (FORMATTED)  |
| 35 |                                      | <b>or</b>  | WRITE (UNFORMATTED)  |
| 36 | R1209 <i>import-stmt</i>             | <b>is</b>  | IMPORT [ [ :: ] <i>import-name-list</i> ]  |
| 37 | C1201 (R1201)                        | An <i>interface-block</i> in a subprogram shall not contain an <i>interface-body</i> for a procedure defined by that subprogram.   |  |
| 38 |                                      |  |  |
| 39 | C1202 (R1201)                        | The <i>generic-spec</i> shall be included in the <i>end-interface-stmt</i> only if it is provided in the <i>interface-stmt</i> . If the <i>end-interface-stmt</i> includes <i>generic-name</i> , the <i>interface-stmt</i> shall specify the same <i>generic-name</i> . If the <i>end-interface-stmt</i> includes ASSIGNMENT(=), the <i>interface-</i> |  |
| 40 |                                      |  |  |
| 41 |                                      |  |  |

- 1        *stmt* shall specify ASSIGNMENT(=). If the *end-interface-stmt* includes *dtio-generic-spec*,  
 2        the *interface-stmt* shall specify the same *dtio-generic-spec*. If the *end-interface-stmt* includes  
 3        OPERATOR(*defined-operator*), the *interface-stmt* shall specify the same *defined-operator*. If  
 4        one *defined-operator* is .LT., .LE., .GT., .GE., .EQ., or .NE., the other is permitted to be the  
 5        corresponding operator <, <=, >, >=, ==, or /=.
- 6 C1203 (R1203) If the *interface-stmt* is ABSTRACT INTERFACE, then the *function-name* in the  
 7        *function-stmt* or the *subroutine-name* in the *subroutine-stmt* shall not be the same as a keyword  
 8        that specifies an intrinsic type.
- 9 C1204 (R1202) A *procedure-stmt* is allowed only in an interface block that has a *generic-spec*.
- 10 C1205 (R1205) An *interface-body* of a pure procedure shall specify the intents of all dummy arguments  
 11        except pointer, alternate return, and procedure arguments.
- 12 C1206 (R1205) An *interface-body* shall not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt*-  
 13        *function-stmt*.
- 14 C1207 (R1206) A *procedure-name* shall have an explicit interface and shall refer to an accessible pro-  
 15        cedure pointer, external procedure, dummy procedure, or module procedure.
- 16 C1208 (R1206) If MODULE appears in a *procedure-stmt*, each *procedure-name* in that statement shall  
 17        be accessible in the current scope as a module procedure.
- 18 C1209 (R1206) A *procedure-name* shall not specify a procedure that is specified previously in any  
 19        *procedure-stmt* in any accessible interface with the same generic identifier.
- 20 C1210 (R1209) The IMPORT statement is allowed only in an *interface-body*.
- 21 C1211 (R1209) Each *import-name* shall be the name of an entity in the host scoping unit.
- 22 An external or module subprogram specifies a **specific interface** for the procedures defined in that  
 23 subprogram. Such a specific interface is explicit for module procedures and implicit for external pro-  
 24 cedures.
- 25 An interface block introduced by ABSTRACT INTERFACE is an **abstract interface block**. An  
 26 interface body in an abstract interface block specifies an abstract interface. An interface block with a  
 27 generic specification is a **generic interface block**. An interface block with neither ABSTRACT nor a  
 28 generic specification is a **specific interface block**.
- 29 The name of the entity declared by an interface body is the *function-name* in the *function-stmt* or the  
 30 *subroutine-name* in the *subroutine-stmt* that begins the interface body.
- 31 An interface body in a generic or specific interface block specifies the EXTERNAL attribute and an  
 32 explicit specific interface for an external procedure or a dummy procedure. If the name of the declared  
 33 procedure is that of a dummy argument in the subprogram containing the interface body, the procedure  
 34 is a dummy procedure; otherwise, it is an external procedure.
- 35 An interface body specifies all of the characteristics of the explicit specific interface or abstract interface.  
 36 The specification part of an interface body may specify attributes or define values for data entities that  
 37 do not determine characteristics of the procedure. Such specifications have no effect.
- 38 If an explicit specific interface is specified by an interface body or a procedure declaration statement  
 39 (12.3.2.3) for an external procedure, the characteristics shall be consistent with those specified in the  
 40 procedure definition, except that the interface may specify a procedure that is not pure if the procedure  
 41 is defined to be pure. An interface for a procedure named by an ENTRY statement may be specified by  
 42 using the entry name as the procedure name in the interface body. An explicit specific interface may  
 43 be specified by an interface body for an external procedure that does not exist in the program if the

- 1 procedure is never used in any way. A procedure shall not have more than one explicit specific interface  
 2 in a given scoping unit.

**NOTE 12.4**

The dummy argument names may be different because the name of a dummy argument is not a characteristic.

- 3 The IMPORT statement specifies that the named entities from the host scoping unit are accessible in  
 4 the interface body by host association. An entity that is imported in this manner and is defined in the  
 5 host scoping unit shall be explicitly declared prior to the interface body. The name of an entity made  
 6 accessible by an IMPORT statement shall not appear in any of the contexts described in 16.4.1.3 that  
 7 cause the host entity of that name to be inaccessible.
- 8 Within an interface body, if an IMPORT statement with no *import-name-list* appears, each host entity  
 9 not named in an IMPORT statement also is made accessible by host association if its name does not  
 10 appear in any of the contexts described in 16.4.1.3 that cause the host entity of that name to be  
 11 inaccessible.

**NOTE 12.5**

An example of an interface block without a generic specification is:

```
INTERFACE
  SUBROUTINE EXT1 (X, Y, Z)
    REAL, DIMENSION (100, 100) :: X, Y, Z
  END SUBROUTINE EXT1
  SUBROUTINE EXT2 (X, Z)
    REAL X
    COMPLEX (KIND = 4) Z (2000)
  END SUBROUTINE EXT2
  FUNCTION EXT3 (P, Q)
    LOGICAL EXT3
    INTEGER P (1000)
    LOGICAL Q (1000)
  END FUNCTION EXT3
END INTERFACE
```

This interface block specifies explicit interfaces for the three external procedures EXT1, EXT2, and EXT3. Invocations of these procedures may use argument keywords (12.4.1); for example:

```
EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

**NOTE 12.6**

The IMPORT statement can be used to allow module procedures to have dummy arguments that are procedures with assumed-shape arguments of an opaque type. For example:

```
MODULE M
  TYPE T
    PRIVATE      ! T is an opaque type
    ...
  END TYPE
CONTAINS
  SUBROUTINE PROCESS(X,Y,RESULT,MONITOR)
    TYPE(T),INTENT(IN) :: X(:,,:),Y(:,,)
```

**NOTE 12.6 (cont.)**

```

TYPE(T), INTENT(OUT) :: RESULT(:, :)
INTERFACE
  SUBROUTINE MONITOR(ITERATION_NUMBER, CURRENT_ESTIMATE)
    IMPORT T
    INTEGER, INTENT(IN) :: ITERATION_NUMBER
    TYPE(T), INTENT(IN) :: CURRENT_ESTIMATE(:, :)
  END SUBROUTINE
END INTERFACE
...
END SUBROUTINE
END MODULE

```

The MONITOR dummy procedure requires an explicit interface because it has an assumed-shape array argument, but TYPE(T) would not be available inside the interface body without the IMPORT statement.

- 1 A generic interface block specifies a **generic interface** for each of the procedures in the interface
- 2 block. The PROCEDURE statement lists procedure pointers, external procedures, dummy procedures,
- 3 or module procedures that have this generic interface. A generic interface is always explicit.
  
- 4 Any procedure may be referenced via its specific interface if the specific interface is accessible. It also
- 5 may be referenced via a generic interface. The *generic-spec* in an *interface-stmt* is a **generic identifier**
- 6 for all the procedures in the interface block. The rules specifying how any two procedures with the same
- 7 generic identifier shall differ are given in [16.2.3](#). They ensure that any generic invocation applies to at
- 8 most one specific procedure.
  
- 9 A **generic name** specifies a single name to reference all of the procedure names in the interface block.
- 10 A generic name may be the same as any one of the procedure names in the interface block, or the same
- 11 as any accessible generic name.
  
- 12 A generic name may be the same as a derived-type name, in which case all of the procedures in the
- 13 interface block shall be functions.

**NOTE 12.7**

An example of a generic procedure interface is:

```

INTERFACE SWITCH
  SUBROUTINE INT_SWITCH (X, Y)
    INTEGER, INTENT (INOUT) :: X, Y
  END SUBROUTINE INT_SWITCH
  SUBROUTINE REAL_SWITCH (X, Y)
    REAL, INTENT (INOUT) :: X, Y
  END SUBROUTINE REAL_SWITCH
  SUBROUTINE COMPLEX_SWITCH (X, Y)
    COMPLEX, INTENT (INOUT) :: X, Y
  END SUBROUTINE COMPLEX_SWITCH
END INTERFACE SWITCH

```

Any of these three subroutines (INT\_SWITCH, REAL\_SWITCH, COMPLEX\_SWITCH) may be referenced with the generic name SWITCH, as well as by its specific name. For example, a reference to INT\_SWITCH could take the form:

**NOTE 12.7 (cont.)**

```
CALL SWITCH (MAX_VAL, LOC_VAL) ! MAX_VAL and LOC_VAL are of type INTEGER
```

- 1 An *interface-stmt* having a *dtio-generic-spec* is an interface for a user-defined derived-type input/output  
 2 procedure (9.5.3.7)

**3 12.3.2.1.1 Defined operations**

- 4 If OPERATOR is specified in a generic specification, all of the procedures specified in the generic interface  
 5 shall be functions that may be referenced as defined operations (7.1.3, 7.1.8.7, 7.2, 12.4). In the case of  
 6 functions of two arguments, infix binary operator notation is implied. In the case of functions of one  
 7 argument, prefix operator notation is implied. OPERATOR shall not be specified for functions with no  
 8 arguments or for functions with more than two arguments. The dummy arguments shall be nonoptional  
 9 dummy data objects and shall be specified with INTENT (IN). The function result shall not have assumed  
 10 character length. If the operator is an *intrinsic-operator* (R310), the number of function arguments shall  
 11 be consistent with the intrinsic uses of that operator, and the types, kind type parameters, or ranks of  
 12 the dummy arguments shall differ from those required for the intrinsic operation (7.1.2).

- 13 A defined operation is treated as a reference to the function. For a unary defined operation, the operand  
 14 corresponds to the function's dummy argument; for a binary operation, the left-hand operand corre-  
 15 sponds to the first dummy argument of the function and the right-hand operand corresponds to the  
 16 second dummy argument.

**NOTE 12.8**

An example of the use of the OPERATOR generic specification is:

```
INTERFACE OPERATOR ( * )
  FUNCTION BOOLEAN_AND ( B1, B2 )
    LOGICAL, INTENT ( IN ) :: B1 (:), B2 ( SIZE ( B1 ) )
    LOGICAL :: BOOLEAN_AND ( SIZE ( B1 ) )
  END FUNCTION BOOLEAN_AND
END INTERFACE OPERATOR ( * )
```

This allows, for example

```
SENSOR ( 1:N ) * ACTION ( 1:N )
```

as an alternative to the function call

```
BOOLEAN_AND ( SENSOR ( 1:N ), ACTION ( 1:N ) ) ! SENSOR and ACTION are
! of type LOGICAL
```

- 17 A given defined operator may, as with generic names, apply to more than one function, in which case  
 18 it is generic in exact analogy to generic procedure names. For intrinsic operator symbols, the generic  
 19 properties include the intrinsic operations they represent. Because both forms of each relational operator  
 20 have the same interpretation (7.2), extending one form (such as <=) has the effect of defining both forms  
 21 (<= and .LE.).

**NOTE 12.9**

In Fortran 90 and Fortran 95, it was not possible to define operators on pointers because pointer  
 dummy arguments were disallowed from having an INTENT attribute. The restriction against  
 INTENT for pointer dummy arguments is now lifted, so defined operations on pointers are now

**NOTE 12.9 (cont.)**

possible.

However, the POINTER attribute cannot be used to resolve generic procedures (16.2.3), so it is not possible to define a generic operator that has one procedure for pointers and another procedure for nonpointers.

**1 12.3.2.1.2 Defined assignments**

2 If ASSIGNMENT ( = ) is specified in a generic specification, all the procedures in the generic interface  
3 shall be subroutines that may be referenced as defined assignments (7.4.1.4). Defined assignment may,  
4 as with generic names, apply to more than one subroutine, in which case it is generic in exact analogy  
5 to generic procedure names.

6 Each of these subroutines shall have exactly two dummy arguments. Each argument shall be nonoptional.  
7 The first argument shall have INTENT (OUT) or INTENT (INOUT) and the second argument shall  
8 have INTENT (IN). Either the second argument shall be an array whose rank differs from that of the  
9 first argument, the declared types and kind type parameters of the arguments shall not conform as  
10 specified in Table 7.8, or the first argument shall be of derived type. A defined assignment is treated  
11 as a reference to the subroutine, with the left-hand side as the first argument and the right-hand side  
12 enclosed in parentheses as the second argument. The ASSIGNMENT generic specification specifies that  
13 assignment is extended or redefined.

**NOTE 12.10**

An example of the use of the ASSIGNMENT generic specification is:

```
INTERFACE ASSIGNMENT ( = )

  SUBROUTINE LOGICAL_TO_NUMERIC ( N, B )
    INTEGER, INTENT (OUT) :: N
    LOGICAL, INTENT (IN)  :: B
  END SUBROUTINE LOGICAL_TO_NUMERIC
  SUBROUTINE CHAR_TO_STRING ( S, C )
    USE STRING_MODULE      ! Contains definition of type STRING
    TYPE (STRING), INTENT (OUT) :: S ! A variable-length string
    CHARACTER (*), INTENT (IN) :: C
  END SUBROUTINE CHAR_TO_STRING
END INTERFACE ASSIGNMENT ( = )
```

Example assignments are:

```
KOUNT = SENSOR ( J )    ! CALL LOGICAL_TO_NUMERIC ( KOUNT, ( SENSOR ( J ) ) )
NOTE  = '89AB'          ! CALL CHAR_TO_STRING ( NOTE, ( '89AB' ) )
```

**14 12.3.2.1.3 User-defined derived-type input/output procedure interfaces**

15 All of the procedures specified in an interface block for a user-defined derived-type input/output proce-  
16 dure shall be subroutines that have interfaces as described in 9.5.3.7.2.

**17 12.3.2.2 EXTERNAL statement**

18 An **EXTERNAL statement** specifies the EXTERNAL attribute (5.1.2.6) for a list of names.

19 R1210 *external-stmt*                    **is** EXTERNAL [ :: ] *external-name-list*

1 Each *external-name* shall be the name of an external procedure, a dummy argument, a procedure pointer,  
 2 or a block data program unit. In an external subprogram, an EXTERNAL statement shall not specify  
 3 the name of a procedure defined by the subprogram.

4 The appearance of the name of a block data program unit in an EXTERNAL statement confirms that  
 5 the block data program unit is a part of the program.

**NOTE 12.11**

For explanatory information on potential portability problems with external procedures, see section [C.9.1](#).

**NOTE 12.12**

An example of an EXTERNAL statement is:

```
EXTERNAL FOCUS
```

**6 12.3.2.3 Procedure declaration statement**

7 A procedure declaration statement declares procedure pointers, dummy procedures, and external pro-  
 8 cedures. It specifies the EXTERNAL attribute ([5.1.2.6](#)) for all procedure entities in the *proc-decl-list*.

9	R1211	<i>procedure-declaration-stmt</i>	is	PROCEDURE ( [ <i>proc-interface</i> ] ) ■
10				■ [ [ , <i>proc-attr-spec</i> ] ... :: ] <i>proc-decl-list</i>
11	R1212	<i>proc-interface</i>	is	<i>interface-name</i>
12			or	<i>declaration-type-spec</i>
13	R1213	<i>proc-attr-spec</i>	is	<i>access-spec</i>
14			or	<i>proc-language-binding-spec</i>
15			or	INTENT ( <i>intent-spec</i> )
16			or	OPTIONAL
17			or	POINTER
18			or	SAVE
19	R1214	<i>proc-decl</i>	is	<i>procedure-entity-name</i> [ => <i>null-init</i> ]
20	R1215	<i>interface-name</i>	is	<i>name</i>

21 C1212 (R1215) The *name* shall be the name of an abstract interface or of a procedure that has an  
 22 explicit interface. If *name* is declared by a *procedure-declaration-stmt* it shall be previously  
 23 declared. If *name* denotes an intrinsic procedure it shall be one that is listed in [13.6](#) and not  
 24 marked with a bullet (●).

25 C1213 (R1215) The *name* shall not be the same as a keyword that specifies an intrinsic type.

26 C1214 If a procedure entity has the INTENT attribute or SAVE attribute, it shall also have the  
 27 POINTER attribute.

28 C1215 (R1211) If a *proc-interface* describes an elemental procedure, each *procedure-entity-name* shall  
 29 specify an external procedure.

30 C1216 (R1214) If => appears in *proc-decl*, the procedure entity shall have the POINTER attribute.

31 C1217 (R1211) If *proc-language-binding-spec* with a NAME= is specified, then *proc-decl-list* shall con-  
 32 tain exactly one *proc-decl*, which shall neither have the POINTER attribute nor be a dummy  
 33 procedure.

34 C1218 (R1211) If *proc-language-binding-spec* is specified, the *proc-interface* shall appear, it shall be an  
 35 *interface-name*, and *interface-name* shall be declared with a *proc-language-binding-spec*.

- 1 If *proc-interface* appears and consists of *interface-name*, it specifies an explicit specific interface (12.3.2.1)  
 2 for the declared procedures or procedure pointers. The abstract interface (12.3) is that specified by the  
 3 interface named by *interface-name*.
- 4 If *proc-interface* appears and consists of *declaration-type-spec*, it specifies that the declared procedures  
 5 or procedure pointers are functions having implicit interfaces and the specified result type. If a type is  
 6 specified for an external function, its function definition (12.5.2.1) shall specify the same result type and  
 7 type parameters.
- 8 If *proc-interface* does not appear, the procedure declaration statement does not specify whether the  
 9 declared procedures or procedure pointers are subroutines or functions.
- 10 If a *proc-attr-spec* other than a *proc-language-binding-spec* appears, it specifies that the declared pro-  
 11 cedures or procedure pointers have that attribute. These attributes are described in 5.1.2. If a *proc-*  
 12 *language-binding-spec* with NAME= appears, it specifies a binding label as described in 15.4.1. A *proc-*  
 13 *language-binding-spec* without a NAME= is allowed, but is redundant with the *proc-interface* required  
 14 by C1218.
- 15 If => *null-init* appears in a *proc-decl* in a *procedure-declaration-stmt*, it specifies that the initial associ-  
 16 ation status of the corresponding procedure entity is disassociated. It also implies the SAVE attribute,  
 17 which may be reaffirmed by explicit use of the SAVE attribute in the *procedure-declaration-stmt* or by  
 18 a SAVE statement.”

**NOTE 12.13**

In contrast to the EXTERNAL statement, it is not possible to use the PROCEDURE statement to identify a BLOCK DATA subprogram.

**NOTE 12.14**

The following code illustrates PROCEDURE statements. Note 7.43 illustrates the use of the P and BESSEL defined by this code.

```

ABSTRACT INTERFACE
  FUNCTION REAL_FUNC (X)
    REAL, INTENT (IN) :: X
    REAL :: REAL_FUNC
  END FUNCTION REAL_FUNC
END INTERFACE

INTERFACE
  SUBROUTINE SUB (X)
    REAL, INTENT (IN) :: X
  END SUBROUTINE SUB
END INTERFACE

!-- Some external or dummy procedures with explicit interface.
PROCEDURE (REAL_FUNC) :: BESSEL, GAMMA
PROCEDURE (SUB) :: PRINT_REAL
!-- Some procedure pointers with explicit interface,
!-- one initialized to NULL().
PROCEDURE (REAL_FUNC), POINTER :: P, R => NULL()
PROCEDURE (REAL_FUNC), POINTER :: PTR_TO_GAMMA
!-- A derived type with a procedure pointer component ...
TYPE STRUCT_TYPE
  PROCEDURE (REAL_FUNC), POINTER :: COMPONENT

```

**NOTE 12.14 (cont.)**

```

END TYPE STRUCT_TYPE
!-- ... and a variable of that type.
TYPE(STRUCT_TYPE) :: STRUCT
!-- An external or dummy function with implicit interface
PROCEDURE (REAL) :: PSI

```

**1 12.3.2.4 INTRINSIC statement**

2 An **INTRINSIC statement** specifies a list of names that have the INTRINSIC attribute (5.1.2.8).

3 R1216 *intrinsic-stmt*                    **is** INTRINSIC [ :: ] *intrinsic-procedure-name-list*

4 C1219 (R1216) Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.

**NOTE 12.15**

A name shall not be explicitly specified to have both the EXTERNAL and INTRINSIC attributes in the same scoping unit.

**5 12.3.2.5 Implicit interface specification**

6 In a scoping unit where the interface of a function is implicit, the type and type parameters of the  
7 function result are specified by an implicit or explicit type specification of the function name. The type,  
8 type parameters, and shape of dummy arguments of a procedure referenced from a scoping unit where  
9 the interface of the procedure is implicit shall be such that the actual arguments are consistent with the  
10 characteristics of the dummy arguments.

**11 12.4 Procedure reference**

12 The form of a procedure reference is dependent on the interface of the procedure or procedure pointer,  
13 but is independent of the means by which the procedure is defined. The forms of procedure references  
14 are:

15 R1217 *function-reference*                    **is** *procedure-designator* ( [ *actual-arg-spec-list* ] )

16 C1220 (R1217) The *procedure-designator* shall designate a function.

17 C1221 (R1217) The *actual-arg-spec-list* shall not contain an *alt-return-spec*.

18 R1218 *call-stmt*                            **is** CALL *procedure-designator* [ ( [ *actual-arg-spec-list* ] ) ]

19 C1222 (R1218) The *procedure-designator* shall designate a subroutine.

20 R1219 *procedure-designator*                **is** *procedure-name*  
21    **or** *proc-component-ref*  
22    **or** *data-ref* % *binding-name*

23 C1223 (R1219) A *procedure-name* shall be the name of a procedure or procedure pointer.

24 C1224 (R1219) A *binding-name* shall be a binding name (4.5.4) of the declared type of *data-ref*.

25 Resolving references to type-bound procedures is described in 12.4.5.

26 A function may also be referenced as a defined operation (12.3.2.1.1). A subroutine may also be referenced  
27 as a defined assignment (12.3.2.1.2).

- 1 R1220 *actual-arg-spec* is [ *keyword* = ] *actual-arg*  
 2 R1221 *actual-arg* is *expr*  
 3 or *variable*  
 4 or *procedure-name*  
 5 or *proc-component-ref*  
 6 or *alt-return-spec*  
 7 R1222 *alt-return-spec* is \* *label*
- 8 C1225 (R1220) The *keyword* = shall not appear if the interface of the procedure is implicit in the  
 9 scoping unit.
- 10 C1226 (R1220) The *keyword* = shall not be omitted from an *actual-arg-spec* unless it has been omitted  
 11 from each preceding *actual-arg-spec* in the argument list.
- 12 C1227 (R1220) Each *keyword* shall be the name of a dummy argument in the explicit interface of the  
 13 procedure.
- 14 C1228 (R1221) A nonintrinsic elemental procedure shall not be used as an actual argument.
- 15 C1229 (R1221) A *procedure-name* shall be the name of an external procedure, a dummy procedure, a  
 16 module procedure, a procedure pointer, or a specific intrinsic function that is listed in 13.6 and  
 17 not marked with a bullet(●).

**NOTE 12.16**

This standard does not allow internal procedures to be used as actual arguments, in part to simplify the problem of ensuring that internal procedures with recursive hosts access entities from the correct instance (12.5.2.3) of the host. If, as an extension, a processor allows internal procedures to be used as actual arguments, the correct instance in this case is the instance in which the procedure is supplied as an actual argument, even if the corresponding dummy argument is eventually invoked from a different instance.

- 18 C1230 (R1221) In a reference to a pure procedure, a *procedure-name actual-arg* shall be the name of a  
 19 pure procedure (12.6).

**NOTE 12.17**

This constraint ensures that the purity of a procedure cannot be undermined by allowing it to call a nonpure procedure.

- 20 C1231 (R1222) The *label* used in the *alt-return-spec* shall be the statement label of a branch target statement that  
 21 appears in the same scoping unit as the *call-stmt*.

**NOTE 12.18**

Successive commas shall not be used to omit optional arguments.

**NOTE 12.19**

Examples of procedure reference using procedure pointers:

```
P => BESSEL
WRITE (*, *) P(2.5)      !-- BESSEL(2.5)

S => PRINT_REAL
CALL S(3.14)
```

### 1 12.4.1 Actual arguments, dummy arguments, and argument association

2 In either a subroutine reference or a function reference, the actual argument list identifies the corre-  
 3 spondence between the actual arguments supplied and the dummy arguments of the procedure. This  
 4 correspondence may be established either by keyword or by position. If an argument keyword appears,  
 5 the actual argument is associated with the dummy argument whose name is the same as the argument  
 6 keyword (using the dummy argument names from the interface accessible in the scoping unit containing  
 7 the procedure reference). In the absence of an argument keyword, an actual argument is associated  
 8 with the dummy argument occupying the corresponding position in the reduced dummy argument list;  
 9 that is, the first actual argument is associated with the first dummy argument in the reduced list, the  
 10 second actual argument is associated with the second dummy argument in the reduced list, etc. The  
 11 reduced dummy argument list is either the full dummy argument list or, if there is a passed-object  
 12 dummy argument (4.5.3.3), the dummy argument list with the passed object dummy argument omitted.  
 13 Exactly one actual argument shall be associated with each nonoptional dummy argument. At most one  
 14 actual argument may be associated with each optional dummy argument. Each actual argument shall  
 15 be associated with a dummy argument.

#### NOTE 12.20

For example, the procedure defined by

```
SUBROUTINE SOLVE (FUNCT, SOLUTION, METHOD, STRATEGY, PRINT)
  INTERFACE
    FUNCTION FUNCT (X)
      REAL FUNCT, X
    END FUNCTION FUNCT
  END INTERFACE
  REAL SOLUTION
  INTEGER, OPTIONAL :: METHOD, STRATEGY, PRINT
  ...
```

may be invoked with

```
CALL SOLVE (FUN, SOL, PRINT = 6)
```

provided its interface is explicit; if the interface is specified by an interface block, the name of the last argument shall be PRINT.

#### 16 12.4.1.1 The passed-object dummy argument and argument association

17 In a reference to a type-bound procedure that has a passed-object dummy argument (4.5.3.3), the *data-*  
 18 *ref* of the *function-reference* or *call-stmt* is associated, as an actual argument, with the passed-object  
 19 dummy argument.

#### 20 12.4.1.2 Actual arguments associated with dummy data objects

21 If a dummy argument is neither allocatable nor a pointer, it shall be type compatible (5.1.1.2) with  
 22 the associated actual argument. If a dummy argument is allocatable or a pointer, the associated actual  
 23 argument shall be polymorphic if and only if the dummy argument is polymorphic, and the declared  
 24 type of the actual argument shall be the same as the declared type of the dummy argument.

#### NOTE 12.21

The dynamic type of a polymorphic allocatable or pointer dummy argument may change as a result of execution of an allocate statement or pointer assignment in the subprogram. Because of this the corresponding actual argument needs to be polymorphic and have a declared type that

**NOTE 12.21 (cont.)**

is the same as the declared type of the dummy argument or an extension of that type. However, type compatibility requires that the declared type of the dummy argument be the same as, or an extension of, the type of the actual argument. Therefore, the dummy and actual arguments need to have the same declared type.

Dynamic type information is not maintained for a non-polymorphic allocatable or pointer dummy argument. However, allocating or pointer assigning such a dummy argument would require maintenance of this information if the corresponding actual argument is polymorphic. Therefore, the corresponding actual argument needs to be non-polymorphic.

- 1 The type parameter values of the actual argument shall agree with the corresponding ones of the dummy
- 2 argument that are not assumed or deferred, except for the case of the character length parameter of
- 3 an actual argument of type default character associated with a dummy argument that is not assumed
- 4 shape.
  
- 5 If a scalar dummy argument is of type default character, the length *len* of the dummy argument shall
- 6 be less than or equal to the length of the actual argument. The dummy argument becomes associated
- 7 with the leftmost *len* characters of the actual argument. If an array dummy argument is of type default
- 8 character and is not assumed shape, it becomes associated with the leftmost characters of the actual
- 9 argument element sequence (12.4.1.5) and it shall not extend beyond the end of that sequence.
  
- 10 The values of assumed type parameters of a dummy argument are assumed from the corresponding type
- 11 parameters of the associated actual argument.
  
- 12 An actual argument associated with a dummy argument that is allocatable or a pointer shall have
- 13 deferred the same type parameters as the dummy argument.
  
- 14 If the dummy argument is a pointer, the actual argument shall be a pointer and the nondeferred type
- 15 parameters and ranks shall agree. If a dummy argument is allocatable, the actual argument shall be
- 16 allocatable and the nondeferred type parameters and ranks shall agree. It is permissible for the actual
- 17 argument to have an allocation status of unallocated.
  
- 18 At the invocation of the procedure, the pointer association status of an actual argument associated with
- 19 a pointer dummy argument becomes undefined if the dummy argument has INTENT(OUT).
  
- 20 Except in references to intrinsic inquiry functions, if the dummy argument is not a pointer and the
- 21 corresponding actual argument is a pointer, the actual argument shall be associated with a target and
- 22 the dummy argument becomes argument associated with that target.
  
- 23 Except in references to intrinsic inquiry functions, if the dummy argument is not allocatable and the
- 24 actual argument is allocatable, the actual argument shall be allocated.
  
- 25 If the dummy argument has the VALUE attribute it becomes associated with a definable anonymous
- 26 data object whose initial value is that of the actual argument. Subsequent changes to the value or
- 27 definition status of the dummy argument do not affect the actual argument.

**NOTE 12.22**

Fortran argument association is usually similar to call by reference and call by value-result. If the VALUE attribute is specified, the effect is as if the actual argument is assigned to a temporary, and the temporary is then argument associated with the dummy argument. The actual mechanism by which this happens is determined by the processor.

- 28 If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated
- 29 with the actual argument do not become associated with the corresponding dummy argument on in-

1 vocation of the procedure. If such a dummy argument is associated with an actual argument that is  
2 a dummy argument with the TARGET attribute, whether any pointers associated with the original  
3 actual argument become associated with the dummy argument with the TARGET attribute is processor  
4 dependent.

5 If the dummy argument has the TARGET attribute, does not have the VALUE attribute, and is either  
6 a scalar or an assumed-shape array, and the corresponding actual argument has the TARGET attribute  
7 but is not an array section with a vector subscript then

- 8 (1) Any pointers associated with the actual argument become associated with the corresponding  
9 dummy argument on invocation of the procedure and
- 10 (2) When execution of the procedure completes, any pointers that do not become undefined  
11 (16.4.2.1.3) and are associated with the dummy argument remain associated with the actual  
12 argument.

13 If the dummy argument has the TARGET attribute and is an explicit-shape array or is an assumed-size  
14 array, and the corresponding actual argument has the TARGET attribute but is not an array section  
15 with a vector subscript then

- 16 (1) On invocation of the procedure, whether any pointers associated with the actual argument  
17 become associated with the corresponding dummy argument is processor dependent and
- 18 (2) When execution of the procedure completes, the pointer association status of any pointer  
19 that is pointer associated with the dummy argument is processor dependent.

20 If the dummy argument has the TARGET attribute and the corresponding actual argument does not  
21 have the TARGET attribute or is an array section with a vector subscript, any pointers associated with  
22 the dummy argument become undefined when execution of the procedure completes.

23 If the dummy argument has the TARGET attribute and the VALUE attribute, any pointers associated  
24 with the dummy argument become undefined when execution of the procedure completes.

25 If the actual argument is scalar, the corresponding dummy argument shall be scalar unless the actual  
26 argument is of type default character, of type character with the C character kind (15.1), or is an element  
27 or substring of an element of an array that is not an assumed-shape or pointer array. If the procedure  
28 is nonelemental and is referenced by a generic name or as a defined operator or defined assignment, the  
29 ranks of the actual arguments and corresponding dummy arguments shall agree.

30 If a dummy argument is an assumed-shape array, the rank of the actual argument shall be the same as  
31 the rank of the dummy argument; the actual argument shall not be an assumed-size array (including an  
32 array element designator or an array element substring designator).

33 Except when a procedure reference is elemental (12.7), each element of an array actual argument or of  
34 a sequence in a sequence association (12.4.1.5) is associated with the element of the dummy array that  
35 has the same position in array element order (6.2.2.2).

#### NOTE 12.23

For type default character sequence associations, the interpretation of element is provided in 12.4.1.5.
---

36 A scalar dummy argument of a nonelemental procedure may be associated only with a scalar actual  
37 argument.

38 If a nonpointer dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument shall  
39 be definable. If a dummy argument has INTENT (OUT), the corresponding actual argument becomes  
40 undefined at the time the association is established, except for components of an object of derived type  
41 for which default initialization has been specified. If the dummy argument is not polymorphic and the

- 1 type of the actual argument is an extension type of the type of the dummy argument, only the part of  
 2 the actual argument that is of the same type as the dummy argument becomes undefined.
- 3 If the actual argument is an array section having a vector subscript, the dummy argument is not defin-  
 4 able and shall not have the INTENT (OUT), INTENT (INOUT), VOLATILE, or ASYNCHRONOUS  
 5 attributes.

**NOTE 12.24**

Argument intent specifications serve several purposes. See Note 5.14.

**NOTE 12.25**

For more explanatory information on argument association and evaluation, see section C.9.5. For more explanatory information on pointers and targets as dummy arguments, see section C.9.6.

- 6 C1232 (R1221) If an actual argument is an array section or an assumed-shape array, and the corre-  
 7 sponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that  
 8 dummy argument shall be an assumed-shape array.
- 9 C1233 (R1221) If an actual argument is a pointer array, and the corresponding dummy argument  
 10 has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be an  
 11 assumed-shape array or a pointer array.

**NOTE 12.26**

The constraints on actual arguments that correspond to a dummy argument with either the ASYN-  
 CHRONOUS or VOLATILE attribute are designed to avoid forcing a processor to use the so-called  
 copy-in/copy-out argument passing mechanism. Making a copy of actual arguments whose values  
 are likely to change due to an asynchronous I/O operation completing or in some unpredictable  
 manner will cause those new values to be lost when a called procedure returns and the copy-out  
 overwrites the actual argument.

**12 12.4.1.3 Actual arguments associated with dummy procedure entities**

- 13 If a dummy argument is a procedure pointer, the associated actual argument shall be a procedure pointer,  
 14 a reference to a function that returns a procedure pointer, or a reference to the NULL intrinsic function.
- 15 If a dummy argument is a dummy procedure without the POINTER attribute, the associated actual  
 16 argument shall be the specific name of an external, module, dummy, or intrinsic procedure, an associated  
 17 procedure pointer, or a reference to a function that returns an associated procedure pointer. The only  
 18 intrinsic procedures permitted are those listed in 13.6 and not marked with a bullet (●). If the specific  
 19 name is also a generic name, only the specific procedure is associated with the dummy argument.
- 20 If an external procedure name or a dummy procedure name is used as an actual argument, its interface  
 21 shall be explicit or it shall be explicitly declared to have the EXTERNAL attribute.
- 22 If the interface of the dummy argument is explicit, the characteristics listed in 12.2 shall be the same  
 23 for the associated actual argument and the corresponding dummy argument, except that a pure actual  
 24 argument may be associated with a dummy argument that is not pure and an elemental intrinsic actual  
 25 procedure may be associated with a dummy procedure (which is prohibited from being elemental).
- 26 If the interface of the dummy argument is implicit and either the name of the dummy argument is  
 27 explicitly typed or it is referenced as a function, the dummy argument shall not be referenced as a  
 28 subroutine and the actual argument shall be a function, function procedure pointer, or dummy procedure.
- 29 If the interface of the dummy argument is implicit and a reference to it appears as a subroutine reference,

1 the actual argument shall be a subroutine, subroutine procedure pointer, or dummy procedure.

#### 2 **12.4.1.4 Actual arguments associated with alternate return indicators**

3 If a dummy argument is an asterisk (12.5.2.2), the associated actual argument shall be an alternate return specifier (12.4).

#### 4 **12.4.1.5 Sequence association**

5 An actual argument represents an **element sequence** if it is an array expression, an array element  
6 designator, a scalar of type default character, or a scalar of type character with the C character kind  
7 (15.1.1). If the actual argument is an array expression, the element sequence consists of the elements  
8 in array element order. If the actual argument is an array element designator, the element sequence  
9 consists of that array element and each element that follows it in array element order.

10 If the actual argument is of type default character or of type character with the C character kind, and is  
11 an array expression, array element, or array element substring designator, the element sequence consists  
12 of the storage units beginning with the first storage unit of the actual argument and continuing to the  
13 end of the array. The storage units of an array element substring designator are viewed as array elements  
14 consisting of consecutive groups of storage units having the character length of the dummy array.

15 If the actual argument is of type default character or of type character with the C character kind, and  
16 is a scalar that is not an array element or array element substring designator, the element sequence  
17 consists of the storage units of the actual argument.

#### **NOTE 12.27**

Some of the elements in the element sequence may consist of storage units from different elements of the original array.
--

18 An actual argument that represents an element sequence and corresponds to a dummy argument that is  
19 an array is sequence associated with the dummy argument if the dummy argument is an explicit-shape  
20 or assumed-size array. The rank and shape of the actual argument need not agree with the rank and  
21 shape of the dummy argument, but the number of elements in the dummy argument shall not exceed  
22 the number of elements in the element sequence of the actual argument. If the dummy argument is  
23 assumed-size, the number of elements in the dummy argument is exactly the number of elements in the  
24 element sequence.

#### 25 **12.4.1.6 Restrictions on dummy arguments not present**

26 A dummy argument or an entity that is host associated with a dummy argument is not **present** if the  
27 dummy argument

- 28 (1) is not associated with an actual argument, or
- 29 (2) is associated with an actual argument that is not present.

30 Otherwise, it is present. A dummy argument that is not optional shall be present. An optional dummy  
31 argument that is not present is subject to the following restrictions:

- 32 (1) If it is a data object, it shall not be referenced or be defined. If it is of a type for which  
33 default initialization is specified for some component, the initialization has no effect.
- 34 (2) It shall not be used as the *data-target* or *proc-target* of a pointer assignment.
- 35 (3) If it is a procedure or procedure pointer, it shall not be invoked.
- 36 (4) It shall not be supplied as an actual argument corresponding to a nonoptional dummy  
37 argument other than as the argument of the PRESENT intrinsic function or as an argument  
38 of a function reference that meets the requirements of (6) or (8) in 7.1.7.

- 1 (5) A designator with it as the base object and with at least one component selector, array  
 2 section selector, array element selector, or substring selector shall not be supplied as an  
 3 actual argument.
- 4 (6) If it is an array, it shall not be supplied as an actual argument to an elemental procedure  
 5 unless an array of the same rank is supplied as an actual argument corresponding to a  
 6 nonoptional dummy argument of that elemental procedure.
- 7 (7) If it is a pointer, it shall not be allocated, deallocated, nullified, pointer-assigned, or supplied  
 8 as an actual argument corresponding to an optional nonpointer dummy argument.
- 9 (8) If it is allocatable, it shall not be allocated, deallocated, or supplied as an actual argument  
 10 corresponding to an optional nonallocatable dummy argument.
- 11 (9) If it has length type parameters, they shall not be the subject of an inquiry.
- 12 (10) It shall not be used as the *selector* in a SELECT TYPE or ASSOCIATE construct.

13 Except as noted in the list above, it may be supplied as an actual argument corresponding to an optional  
 14 dummy argument, which is then also considered not to be associated with an actual argument.

#### 15 12.4.1.7 Restrictions on entities associated with dummy arguments

16 While an entity is associated with a dummy argument, the following restrictions hold:

- 17 (1) Action that affects the allocation status of the entity or a subobject thereof shall be taken  
 18 through the dummy argument. Action that affects the value of the entity or any subobject  
 19 of it shall be taken only through the dummy argument unless
- 20 (a) the dummy argument has the POINTER attribute or
- 21 (b) the dummy argument has the TARGET attribute, the dummy argument does not  
 22 have INTENT (IN), the dummy argument is a scalar object or an assumed-shape  
 23 array, and the actual argument is a target other than an array section with a vector  
 24 subscript.

#### NOTE 12.28

```
In
SUBROUTINE OUTER
  REAL, POINTER :: A (:)
  ...
  ALLOCATE (A (1:N))
  ...
  CALL INNER (A)
  ...
CONTAINS
  SUBROUTINE INNER (B)
    REAL :: B (:)
    ...
  END SUBROUTINE INNER
  SUBROUTINE SET (C, D)
    REAL, INTENT (OUT) :: C
    REAL, INTENT (IN) :: D
    C = D
  END SUBROUTINE SET
END SUBROUTINE OUTER
```

an assignment statement such as

**NOTE 12.28 (cont.)**

```
A (1) = 1.0
```

would not be permitted during the execution of INNER because this would be changing A without using B, but statements such as

```
B (1) = 1.0
```

or

```
CALL SET (B (1), 1.0)
```

would be allowed. Similarly,

```
DEALLOCATE (A)
```

would not be allowed because this affects the allocation of B without using B. In this case,

```
DEALLOCATE (B)
```

also would not be permitted. If B were declared with the POINTER attribute, either of the statements

```
DEALLOCATE (A)
```

and

```
DEALLOCATE (B)
```

would be permitted, but not both.

**NOTE 12.29**

If there is a partial or complete overlap between the actual arguments associated with two different dummy arguments of the same procedure and the dummy arguments have neither the POINTER nor TARGET attribute, the overlapped portions shall not be defined, redefined, or become undefined during the execution of the procedure. For example, in

```
CALL SUB (A (1:5), A (3:9))
```

A (3:5) shall not be defined, redefined, or become undefined through the first dummy argument because it is part of the argument associated with the second dummy argument and shall not be defined, redefined, or become undefined through the second dummy argument because it is part of the argument associated with the first dummy argument. A (1:2) remains definable through the first dummy argument and A (6:9) remains definable through the second dummy argument.

**NOTE 12.30**

This restriction applies equally to pointer targets. In

```
REAL, DIMENSION (10), TARGET :: A
REAL, DIMENSION (:), POINTER :: B, C
B => A (1:5)
C => A (3:9)
```

**NOTE 12.30 (cont.)**

CALL SUB (B, C) ! The dummy arguments of SUB are neither pointers nor targets.

B (3:5) cannot be defined because it is part of the argument associated with the second dummy argument. C (1:3) cannot be defined because it is part of the argument associated with the first dummy argument. A (1:2) [which is B (1:2)] remains definable through the first dummy argument and A (6:9) [which is C (4:7)] remains definable through the second dummy argument.

**NOTE 12.31**

Because a nonpointer dummy argument declared with INTENT(IN) shall not be used to change the associated actual argument, the associated actual argument remains constant throughout the execution of the procedure.

- 1           (2) If the allocation status of the entity or a subobject thereof is affected through the dummy  
2           argument, then at any time during the execution of the procedure, either before or after  
3           the allocation or deallocation, it may be referenced only through the dummy argument. If  
4           the value the entity or any subobject of it is affected through the dummy argument, then  
5           at any time during the execution of the procedure, either before or after the definition, it  
6           may be referenced only through that dummy argument unless
- 7           (a) the dummy argument has the POINTER attribute or  
8           (b) the dummy argument has the TARGET attribute, the dummy argument does not  
9           have INTENT (IN), the dummy argument is a scalar object or an assumed-shape  
10           array, and the actual argument is a target other than an array section with a vector  
11           subscript.

**NOTE 12.32**

In

```

MODULE DATA
  REAL :: W, X, Y, Z
END MODULE DATA

PROGRAM MAIN
  USE DATA
  ...
  CALL INIT (X)
  ...
END PROGRAM MAIN
SUBROUTINE INIT (V)
  USE DATA
  ...
  READ (*, *) V
  ...
END SUBROUTINE INIT

```

variable X shall not be directly referenced at any time during the execution of INIT because it is being defined through the dummy argument V. X may be (indirectly) referenced through V. W, Y, and Z may be directly referenced. X may, of course, be directly referenced once execution of INIT is complete.

**NOTE 12.33**

The restrictions on entities associated with dummy arguments are intended to facilitate a variety of optimizations in the translation of the subprogram, including implementations of argument association in which the value of an actual argument that is neither a pointer nor a target is maintained in a register or in local storage.

**1 12.4.2 Function reference**

2 A function is invoked during expression evaluation by a *function-reference* or by a defined operation  
3 (7.1.3). When it is invoked, all actual argument expressions are evaluated, then the arguments are  
4 associated, and then the function is executed. When execution of the function is complete, the value  
5 of the function result is available for use in the expression that caused the function to be invoked. The  
6 characteristics of the function result (12.2.2) are determined by the interface of the function. A reference  
7 to an elemental function (12.7) is an elemental reference if one or more actual arguments are arrays and  
8 all array arguments have the same shape.

**9 12.4.3 Subroutine reference**

10 A subroutine is invoked by execution of a CALL statement, execution of a defined assignment statement  
11 (7.4.1.4), user-defined derived-type input/output(9.5.3.7.1), or finalization(4.5.5). When a subroutine is  
12 invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the  
13 subroutine is executed. When the actions specified by the subroutine are completed, the execution of  
14 the CALL statement, the execution of the defined assignment statement, or the processing of an input  
15 or output list item is also completed. If a CALL statement includes one or more alternate return specifiers among  
16 its arguments, control may be transferred to one of the statements indicated, depending on the action specified by the  
17 subroutine. A reference to an elemental subroutine (12.7) is an elemental reference if there is at least one  
18 actual argument corresponding to an INTENT(OUT) or INTENT(INOUT) dummy argument, all such  
19 actual arguments are arrays, and all actual arguments are conformable.

**20 12.4.4 Resolving named procedure references**

21 The rules for interpreting a procedure reference depend on whether the procedure name in the reference  
22 is established by the available declarations and specifications to be generic in the scoping unit containing  
23 the reference, is established to be only specific in the scoping unit containing the reference, or is not  
24 established.

- 25 (1) A procedure name is established to be generic in a scoping unit
- 26 (a) if that scoping unit contains an interface block with that name;
  - 27 (b) if that scoping unit contains an INTRINSIC attribute specification for that name and  
28 it is the name of a generic intrinsic procedure;
  - 29 (c) if that scoping unit contains a USE statement that makes that procedure name ac-  
30 cessible and the corresponding name in the module is established to be generic; or
  - 31 (d) if that scoping unit contains no declarations of that name, that scoping unit has a  
32 host scoping unit, and that name is established to be generic in the host scoping unit.
- 33 (2) A procedure name is established to be only specific in a scoping unit if it is established to  
34 be specific and not established to be generic. It is established to be specific
- 35 (a) if that scoping unit contains a module subprogram, internal subprogram, or statement  
36 function that defines a procedure with that name;
  - 37 (b) if that scoping unit contains an INTRINSIC attribute specification for that name and  
38 if it is the name of a specific intrinsic procedure;
  - 39 (c) if that scoping unit contains an explicit EXTERNAL attribute specification (5.1.2.6)  
40 for that name;

- 1 (d) if that scoping unit contains a USE statement that makes that procedure name ac-  
 2 cessible and the corresponding name in the module is established to be specific; or  
 3 (e) if that scoping unit contains no declarations of that name, that scoping unit has a  
 4 host scoping unit, and that name is established to be specific in the host scoping unit.
- 5 (3) A procedure name is not established in a scoping unit if it is neither established to be generic  
 6 nor established to be specific.

#### 7 12.4.4.1 Resolving procedure references to names established to be generic

- 8 (1) If the reference is consistent with a nonelemental reference to one of the specific interfaces of  
 9 a generic interface that has that name and either is in the scoping unit in which the reference  
 10 appears or is made accessible by a USE statement in the scoping unit, the reference is to  
 11 the specific procedure in the interface block that provides that interface. The rules in 16.2.3  
 12 ensure that there can be at most one such specific procedure.
- 13 (2) If (1) does not apply, if the reference is consistent with an elemental reference to one of the  
 14 specific interfaces of a generic interface that has that name and either is in the scoping unit  
 15 in which the reference appears or is made accessible by a USE statement in the scoping unit,  
 16 the reference is to the specific elemental procedure in the interface block that provides that  
 17 interface. The rules in 16.2.3 ensure that there can be at most one such specific elemental  
 18 procedure.

#### NOTE 12.34

These rules allow particular instances of a generic function to be used for particular array ranks and a general elemental version to be used for other ranks. Given an interface block such as:

```
INTERFACE RANF
  ELEMENTAL FUNCTION SCALAR_RANF(X)
  REAL X
  END FUNCTION SCALAR_RANF
  FUNCTION VECTOR_RANDOM(X)
  REAL X(:)
  REAL VECTOR_RANDOM(SIZE(X))
  END FUNCTION VECTOR_RANDOM
END INTERFACE RANF
```

and a declaration such as:

```
REAL A(10,10), AA(10,10)
```

then the statement

```
A = RANF(AA)
```

is an elemental reference to SCALAR\_RANF. The statement

```
A = RANF(AA(6:10,2))
```

is a nonelemental reference to VECTOR\_RANDOM.

- 19 (3) If (1) and (2) do not apply, if the scoping unit contains either an INTRINSIC attribute  
 20 specification for that name or a USE statement that makes that name accessible from a  
 21 module in which the corresponding name is specified to have the INTRINSIC attribute, and  
 22 if the reference is consistent with the interface of that intrinsic procedure, the reference is  
 23 to that intrinsic procedure.

**NOTE 12.35**

In the USE statement case, it is possible, because of the renaming facility, for the name in the reference to be different from the name of the intrinsic procedure.

- 1       (4) If (1), (2), and (3) do not apply, if the scoping unit has a host scoping unit, if the name  
 2       is established to be generic in that host scoping unit, and if there is agreement between  
 3       the scoping unit and the host scoping unit as to whether the name is a function name or  
 4       a subroutine name, the name is resolved by applying the rules in this section to the host  
 5       scoping unit.

**12.4.4.2 Resolving procedure references to names established to be only specific**

- 6       (1) If the scoping unit contains an interface body or EXTERNAL attribute specification for the  
 7       name, if the scoping unit is a subprogram, and if the name is the name of a dummy argument  
 8       of that subprogram, the dummy argument is a dummy procedure and the reference is to  
 9       that dummy procedure. That is, the procedure invoked by executing that reference is the  
 10      procedure supplied as the actual argument corresponding to that dummy procedure.  
 11  
 12      (2) If the scoping unit contains an interface body or EXTERNAL attribute specification for the  
 13      name and if (1) does not apply, the reference is to an external procedure with that name.  
 14      (3) If the scoping unit contains a module subprogram, internal subprogram, or statement function  
 15      that defines a procedure with the name, the reference is to the procedure so defined.  
 16      (4) If the scoping unit contains an INTRINSIC attribute specification for the name, the reference  
 17      is to the intrinsic with that name.  
 18      (5) If the scoping unit contains a USE statement that makes a procedure accessible by the  
 19      name, the reference is to that procedure.

**NOTE 12.36**

Because of the renaming facility of the USE statement, the name in the reference may be different from the original name of the procedure.

- 20      (6) If none of the above apply, the scoping unit shall have a host scoping unit, and the reference  
 21      is resolved by applying the rules in this section to the host scoping unit.

**12.4.4.3 Resolving procedure references to names not established**

- 22      (1) If the scoping unit is a subprogram and if the name is the name of a dummy argument  
 23      of that subprogram, the dummy argument is a dummy procedure and the reference is to  
 24      that dummy procedure. That is, the procedure invoked by executing that reference is the  
 25      procedure supplied as the actual argument corresponding to that dummy procedure.  
 26      (2) If (1) does not apply, if the name is the name of an intrinsic procedure, and if there is  
 27      agreement between the reference and the status of the intrinsic procedure as being a function  
 28      or subroutine, the reference is to that intrinsic procedure.  
 29      (3) If (1) and (2) do not apply, the reference is to an external procedure with that name.  
 30

**12.4.5 Resolving type-bound procedure references**

32 If the *binding-name* in a *procedure-designator* (R1219) is that of a specific type-bound procedure, the  
 33 procedure referenced is the one bound to that name in the dynamic type of the *data-ref*.

34 If the *binding-name* in a *procedure-designator* is that of a generic type bound procedure, the generic  
 35 binding with that name in the declared type of the *data-ref* is used to select a specific binding:

- 36      (1) If the reference is consistent with one of the specific bindings of that generic binding, that  
 37      specific binding is selected.  
 38      (2) Otherwise, the reference shall be consistent with an elemental reference to one of the specific  
 39      bindings of that generic binding; that specific binding is selected.

1 The reference is to the procedure bound to the same name as the selected specific binding in the dynamic  
2 type of the *data-ref*.

## 3 12.5 Procedure definition

### 4 12.5.1 Intrinsic procedure definition

5 Intrinsic procedures are defined as an inherent part of the processor. A standard-conforming processor  
6 shall include the intrinsic procedures described in Section 13, but may include others. However, a  
7 standard-conforming program shall not make use of intrinsic procedures other than those described in  
8 Section 13.

### 9 12.5.2 Procedures defined by subprograms

10 When a procedure defined by a subprogram is invoked, an instance (12.5.2.3) of the subprogram is  
11 created and executed. Execution begins with the first executable construct following the FUNCTION,  
12 SUBROUTINE, or ENTRY statement specifying the name of the procedure invoked.

#### 13 12.5.2.1 Function subprogram

14 A **function subprogram** is a subprogram that has a FUNCTION statement as its first statement.

15 R1223 *function-subprogram*        **is** *function-stmt*  
16    [ *specification-part* ]  
17    [ *execution-part* ]  
18    [ *internal-subprogram-part* ]  
19    *end-function-stmt*

20 R1224 *function-stmt*                **is** [ *prefix* ] FUNCTION *function-name* ■  
21    ■ ( [ *dummy-arg-name-list* ] ) [ *suffix* ]

22 C1234 (R1224) If RESULT is specified, *result-name* shall not be the same as *function-name* and shall  
23 not be the same as the *entry-name* in any ENTRY statement in the subprogram.

24 C1235 (R1224) If RESULT is specified, the *function-name* shall not appear in any specification state-  
25 ment in the scoping unit of the function subprogram.

26 R1225 *proc-language-binding-spec* **is** *language-binding-spec*

27 C1236 (R1225) A *proc-language-binding-spec* with a NAME= specifier shall not be specified in the  
28 *function-stmt* or *subroutine-stmt* of an interface body for an abstract interface or a dummy  
29 procedure.

30 C1237 (R1225) A *proc-language-binding-spec* shall not be specified for an internal procedure.

31 C1238 (R1225) If *proc-language-binding-spec* is specified for a procedure, each of the procedure's dummy  
32 arguments shall be a nonoptional interoperable variable (15.2.4, 15.2.5) or an interoperable  
33 procedure (15.2.6). If *proc-language-binding-spec* is specified for a function, the function result  
34 shall be an interoperable variable.

35 R1226 *dummy-arg-name*                **is** *name*

36 C1239 (R1226) A *dummy-arg-name* shall be the name of a dummy argument.

37 R1227 *prefix*                                **is** *prefix-spec* [ *prefix-spec* ] ...

38 R1228 *prefix-spec*                        **is** *declaration-type-spec*

- 1                                 **or** RECURSIVE  
2                                 **or** PURE  
3                                 **or** ELEMENTAL
- 4 C1240 (R1227) A *prefix* shall contain at most one of each *prefix-spec*.
- 5 C1241 (R1227) A *prefix* shall not specify both ELEMENTAL and RECURSIVE.
- 6 C1242 (R1227) A *prefix* shall not specify ELEMENTAL if *proc-language-binding-spec* appears in the  
7     *function-stmt* or *subroutine-stmt*.
- 8 R1229 *suffix*                         **is** *proc-language-binding-spec* [ RESULT ( *result-name* ) ]  
9   **or** RESULT ( *result-name* ) [ *proc-language-binding-spec* ]
- 10 R1230 *end-function-stmt*                 **is** END [ FUNCTION [ *function-name* ] ]
- 11 C1243 (R1230) FUNCTION shall appear in the *end-function-stmt* of an internal or module function.
- 12 C1244 (R1223) An internal function subprogram shall not contain an ENTRY statement.
- 13 C1245 (R1223) An internal function subprogram shall not contain an *internal-subprogram-part*.
- 14 C1246 (R1230) If a *function-name* appears in the *end-function-stmt*, it shall be identical to the *function-*  
15     *name* specified in the *function-stmt*.
- 16 The name of the function is *function-name*.
- 17 The type and type parameters (if any) of the result of the function defined by a function subprogram  
18 may be specified by a type specification in the FUNCTION statement or by the name of the result  
19 variable appearing in a type declaration statement in the declaration part of the function subprogram.  
20 They shall not be specified both ways. If they are not specified either way, they are determined by  
21 the implicit typing rules in force within the function subprogram. If the function result is an array,  
22 allocatable, or a pointer, this shall be specified by specifications of the name of the result variable within  
23 the function body. The specifications of the function result attributes, the specification of dummy  
24 argument attributes, and the information in the procedure heading collectively define the characteristics  
25 of the function (12.2).
- 26 The *prefix-spec* RECURSIVE shall appear if the function directly or indirectly invokes itself or a function  
27 defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall appear if a  
28 function defined by an ENTRY statement in the subprogram directly or indirectly invokes itself, another  
29 function defined by an ENTRY statement in that subprogram, or the function defined by the FUNCTION  
30 statement.
- 31 If RESULT is specified, the name of the result variable of the function is *result-name* and all occurrences  
32 of the function name in *execution-part* statements in the scoping unit refer to the function itself. If  
33 RESULT is not specified, the result variable is *function-name* and all occurrences of the function name  
34 in *execution-part* statements in the scoping unit are references to the result variable. The characteristics  
35 (12.2.2) of the function result are those of the result variable. On completion of execution of the function,  
36 the value returned is that of its result variable. If the function result is a pointer, the shape of the value  
37 returned by the function is determined by the shape of the result variable when the execution of the  
38 function is completed. If the result variable is not a pointer, its value shall be defined by the function.  
39 If the function result is a pointer, the function shall either associate a target with the result variable  
40 pointer or cause the association status of this pointer to become defined as disassociated.

**NOTE 12.37**

The result variable is similar to any other variable local to a function subprogram. Its existence begins when execution of the function is initiated and ends when execution of the function is
--

**NOTE 12.37 (cont.)**

terminated. However, because the final value of this variable is used subsequently in the evaluation of the expression that invoked the function, an implementation may wish to defer releasing the storage occupied by that variable until after its value has been used in expression evaluation.

- 1 If the *prefix-spec* PURE or ELEMENTAL appears, the subprogram is a pure subprogram and shall meet the additional constraints of 12.6.
- 2
- 3 If the *prefix-spec* ELEMENTAL appears, the subprogram is an elemental subprogram and shall meet the additional constraints of 12.7.1.
- 4

**NOTE 12.38**

An example of a recursive function is:

```

RECURSIVE FUNCTION CUMM_SUM (ARRAY) RESULT (C_SUM)
  REAL, INTENT (IN), DIMENSION (: ) :: ARRAY
  REAL, DIMENSION (SIZE (ARRAY)) :: C_SUM
  INTEGER N
  N = SIZE (ARRAY)
  IF (N <= 1) THEN
    C_SUM = ARRAY
  ELSE
    N = N / 2
    C_SUM (:N) = CUMM_SUM (ARRAY (:N))
    C_SUM (N+1:) = C_SUM (N) + CUMM_SUM (ARRAY (N+1:))
  END IF
END FUNCTION CUMM_SUM

```

**NOTE 12.39**

The following is an example of the declaration of an interface body with the BIND attribute, and a reference to the procedure declared.

```

USE, INTRINSIC :: ISO_C_BINDING

INTERFACE
  FUNCTION JOE (I, J, R), BIND(C,NAME="FrEd")
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER(C_INT) :: JOE
    INTEGER(C_INT), VALUE :: I, J
    REAL(C_FLOAT), VALUE :: R
  END FUNCTION JOE
END INTERFACE

INT = JOE(1_C_INT, 3_C_INT, 4.0_C_FLOAT)
END PROGRAM

```

The invocation of the function JOE results in a reference to a function with a binding label "FrEd". FrEd may be a C function described by the C prototype

```
int FrEd(int n, int m, float x);
```

1 **12.5.2.2 Subroutine subprogram**

2 A **subroutine subprogram** is a subprogram that has a SUBROUTINE statement as its first statement.

```

3 R1231 subroutine-subprogram      is subroutine-stmt
4                                     [ specification-part ]
5                                     [ execution-part ]
6                                     [ internal-subprogram-part ]
7                                     end-subroutine-stmt
8 R1232 subroutine-stmt           is [ prefix ] SUBROUTINE subroutine-name ■
9                                     ■ [ ( [ dummy-arg-list ] ) [ proc-language-binding-spec ] ]
```

10 C1247 (R1232) The *prefix* of a *subroutine-stmt* shall not contain a *declaration-type-spec*.

```

11 R1233 dummy-arg                 is dummy-arg-name
12                                     or *
```

```

13 R1234 end-subroutine-stmt       is END [ SUBROUTINE [ subroutine-name ] ]
```

14 C1248 (R1234) SUBROUTINE shall appear in the *end-subroutine-stmt* of an internal or module sub-  
15 routine.

16 C1249 (R1231) An internal subroutine subprogram shall not contain an ENTRY statement.

17 C1250 (R1231) An internal subroutine subprogram shall not contain an *internal-subprogram-part*.

18 C1251 (R1234) If a *subroutine-name* appears in the *end-subroutine-stmt*, it shall be identical to the  
19 *subroutine-name* specified in the *subroutine-stmt*.

20 The name of the subroutine is *subroutine-name*.

21 The *prefix-spec* RECURSIVE shall appear if the subroutine directly or indirectly invokes itself or a  
22 subroutine defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall  
23 appear if a subroutine defined by an ENTRY statement in the subprogram directly or indirectly invokes  
24 itself, another subroutine defined by an ENTRY statement in that subprogram, or the subroutine defined  
25 by the SUBROUTINE statement.

26 If the *prefix-spec* PURE or ELEMENTAL appears, the subprogram is a pure subprogram and shall meet  
27 the additional constraints of 12.6.

28 If the *prefix-spec* ELEMENTAL appears, the subprogram is an elemental subprogram and shall meet  
29 the additional constraints of 12.7.1.

30 **12.5.2.3 Instances of a subprogram**

31 When a function or subroutine defined by a subprogram is invoked, an **instance** of that subprogram is  
32 created. When a statement function is invoked, an instance of that statement function is created.

33 Each instance has an independent sequence of execution and an independent set of dummy arguments  
34 and local unsaved data objects. If an internal procedure or statement function in the subprogram is invoked  
35 directly from an instance of the subprogram or from an internal subprogram or statement function that  
36 has access to the entities of that instance, the created instance of the internal subprogram or statement  
37 function also has access to the entities of that instance of the host subprogram.

38 All other entities are shared by all instances of the subprogram.

**NOTE 12.40**

The value of a saved data object appearing in one instance may have been defined in a previous instance or by initialization in a DATA statement or type declaration statement.

1 **12.5.2.4 ENTRY statement**

2 An **ENTRY statement** permits a procedure reference to begin with a particular executable statement  
3 within the function or subroutine subprogram in which the ENTRY statement appears.

4 R1235 *entry-stmt*                                    **is** ENTRY *entry-name* [ ( [ *dummy-arg-list* ] ) [ *suffix* ] ]

5 C1252 (R1235) If RESULT is specified, the *entry-name* shall not appear in any specification or type-  
6 declaration statement in the scoping unit of the function program.

7 C1253 (R1235) An *entry-stmt* shall appear only in an *external-subprogram* or *module-subprogram*. An  
8 *entry-stmt* shall not appear within an *executable-construct*.

9 C1254 (R1235) RESULT shall appear only if the *entry-stmt* is in a function subprogram.

10 C1255 (R1235) Within the subprogram containing the *entry-stmt*, the *entry-name* shall not appear  
11 as a dummy argument in the FUNCTION or SUBROUTINE statement or in another ENTRY  
12 statement nor shall it appear in an EXTERNAL, INTRINSIC, or PROCEDURE statement.

13 C1256 (R1235) A *dummy-arg* shall not be an alternate return indicator if the ENTRY statement is in a function  
14 subprogram.

15 C1257 (R1235) If RESULT is specified, *result-name* shall not be the same as the *function-name* in the  
16 FUNCTION statement and shall not be the same as the *entry-name* in any ENTRY statement  
17 in the subprogram.

18 Optionally, a subprogram may have one or more ENTRY statements.

19 If the ENTRY statement is in a function subprogram, an additional function is defined by that subpro-  
20 gram. The name of the function is *entry-name* and the name of its result variable is *result-name* or  
21 is *entry-name* if no *result-name* is provided. The characteristics of the function result are specified by  
22 specifications of the result variable. The dummy arguments of the function are those specified in the  
23 ENTRY statement. If the characteristics of the result of the function named in the ENTRY statement  
24 are the same as the characteristics of the result of the function named in the FUNCTION statement,  
25 their result variables identify the same variable, although their names need not be the same. Otherwise,  
26 they are storage associated and shall all be nonpointer, nonallocatable scalars of type default integer,  
27 default real, double precision real, default complex, or default logical.

28 If the ENTRY statement is in a subroutine subprogram, an additional subroutine is defined by that  
29 subprogram. The name of the subroutine is *entry-name*. The dummy arguments of the subroutine are  
30 those specified in the ENTRY statement.

31 The order, number, types, kind type parameters, and names of the dummy arguments in an ENTRY  
32 statement may differ from the order, number, types, kind type parameters, and names of the dummy  
33 arguments in the FUNCTION or SUBROUTINE statement in the containing subprogram.

34 Because an ENTRY statement defines an additional function or an additional subroutine, it is referenced  
35 in the same manner as any other function or subroutine (12.4).

36 In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear  
37 in an executable statement preceding that ENTRY statement, unless it also appears in a FUNCTION,  
38 SUBROUTINE, or ENTRY statement that precedes the executable statement.

1 In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear in the expression  
 2 of a statement function unless the name is also a dummy argument of the statement function, appears in a FUNCTION  
 3 or SUBROUTINE statement, or appears in an ENTRY statement that precedes the statement function statement.

4 If a dummy argument appears in an executable statement, the execution of the executable statement is  
 5 permitted during the execution of a reference to the function or subroutine only if the dummy argument  
 6 appears in the dummy argument list of the procedure name referenced.

7 If a dummy argument is used in a specification expression to specify an array bound or character length  
 8 of an object, the appearance of the object in a statement that is executed during a procedure reference  
 9 is permitted only if the dummy argument appears in the dummy argument list of the procedure name  
 10 referenced and it is present (12.4.1.6).

11 A scoping unit containing a reference to a procedure defined by an ENTRY statement may have access to  
 12 an interface body for the procedure. The procedure header for the interface body shall be a FUNCTION  
 13 statement for an entry in a function subprogram and shall be a SUBROUTINE statement for an entry  
 14 in a subroutine subprogram.

15 The keyword RECURSIVE is not used in an ENTRY statement. Instead, the presence or absence of  
 16 RECURSIVE in the initial SUBROUTINE or FUNCTION statement controls whether the procedure  
 17 defined by an ENTRY statement is permitted to reference itself.

18 The keyword PURE is not used in an ENTRY statement. Instead, the procedure defined by an ENTRY  
 19 statement is pure if and only if PURE or ELEMENTAL is specified in the SUBROUTINE or FUNCTION  
 20 statement.

21 The keyword ELEMENTAL is not used in an ENTRY statement. Instead, the procedure defined by  
 22 an ENTRY statement is elemental if and only if ELEMENTAL is specified in the SUBROUTINE or  
 23 FUNCTION statement.

#### 24 12.5.2.5 RETURN statement

25 R1236 *return-stmt* is RETURN [ *scalar-int-expr* ]

26 C1258 (R1236) The *return-stmt* shall be in the scoping unit of a function or subroutine subprogram.

27 C1259 (R1236) The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.

28 Execution of the RETURN statement completes execution of the instance of the subprogram in which  
 29 it appears. If the expression appears and has a value  $n$  between 1 and the number of asterisks in the dummy argument  
 30 list, the CALL statement that invoked the subroutine transfers control to the statement identified by the  $n$ th alternate  
 31 return specifier in the actual argument list. If the expression is omitted or has a value outside the required range, there is  
 32 no transfer of control to an alternate return.

33 Execution of an *end-function-stmt* or *end-subroutine-stmt* is equivalent to executing a RETURN state-  
 34 ment with no expression.

#### 35 12.5.2.6 CONTAINS statement

36 R1237 *contains-stmt* is CONTAINS

37 The CONTAINS statement separates the body of a main program, module, or subprogram from any  
 38 internal or module subprograms it may contain, or it introduces the type-bound procedure part of a  
 39 derived-type definition (4.5.1). The CONTAINS statement is not executable.

### 1 12.5.3 Definition and invocation of procedures by means other than Fortran

2 A procedure may be defined by means other than Fortran. The interface of a procedure defined by means  
 3 other than Fortran may be specified in an interface block. If the interface of such a procedure does not  
 4 have a *proc-language-binding-spec*, the means by which the procedure is defined are processor dependent.  
 5 A reference to such a procedure is made as though it were defined by an external subprogram.

6 If the interface of a procedure has a *proc-language-binding-spec*, the procedure is interoperable (15.4).

7 Interoperation with C functions is described in 15.4.

#### NOTE 12.41

For explanatory information on definition of procedures by means other than Fortran, see section C.9.2.

### 8 12.5.4 Statement function

9 A statement function is a function defined by a single statement.

10 R1238 *stmt-function-stmt* is *function-name* ( [ *dummy-arg-name-list* ] ) = *scalar-expr*

11 C1260 (R1238) The *primaries* of the *scalar-expr* shall be constants (literal and named), references to variables, references  
 12 to functions and function dummy procedures, and intrinsic operations. If *scalar-expr* contains a reference to a  
 13 function or a function dummy procedure, the reference shall not require an explicit interface, the function shall  
 14 not require an explicit interface unless it is an intrinsic, the function shall not be a transformational intrinsic,  
 15 and the result shall be scalar. If an argument to a function or a function dummy procedure is an array, it shall  
 16 be an array name. If a reference to a statement function appears in *scalar-expr*, its definition shall have been  
 17 provided earlier in the scoping unit and shall not be the name of the statement function being defined.

18 C1261 (R1238) Named constants in *scalar-expr* shall have been declared earlier in the scoping unit or made accessible  
 19 by use or host association. If array elements appear in *scalar-expr*, the array shall have been declared as an array  
 20 earlier in the scoping unit or made accessible by use or host association.

21 C1262 (R1238) If a *dummy-arg-name*, variable, function reference, or dummy function reference is typed by the implicit  
 22 typing rules, its appearance in any subsequent type declaration statement shall confirm this implied type and  
 23 the values of any implied type parameters.

24 C1263 (R1238) The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.

25 C1264 (R1238) A given *dummy-arg-name* shall not appear more than once in any *dummy-arg-name-list*.

26 C1265 (R1238) Each variable reference in *scalar-expr* may be either a reference to a dummy argument of the statement  
 27 function or a reference to a variable accessible in the same scoping unit as the statement function statement.

28 The definition of a statement function with the same name as an accessible entity from the host shall be preceded by the  
 29 declaration of its type in a type declaration statement.

30 The dummy arguments have a scope of the statement function statement. Each dummy argument has the same type and  
 31 type parameters as the entity of the same name in the scoping unit containing the statement function.

32 A statement function shall not be supplied as a procedure argument.

33 The value of a statement function reference is obtained by evaluating the expression using the values of the actual arguments  
 34 for the values of the corresponding dummy arguments and, if necessary, converting the result to the declared type and  
 35 type attributes of the function.

36 A function reference in the scalar expression shall not cause a dummy argument of the statement function to become

1 redefined or undefined.

## 2 12.6 Pure procedures

3 A pure procedure is

- 4 (1) A pure intrinsic function (13.1),
- 5 (2) A pure intrinsic subroutine (13.1),
- 6 (3) Defined by a pure subprogram, or
- 7 (4) A statement function that references only pure functions.

8 A pure subprogram is a subprogram that has the *prefix-spec* PURE or ELEMENTAL. The following  
9 additional constraints apply to pure subprograms.

- 10 C1266 The *specification-part* of a pure function subprogram shall specify that all its nonpointer dummy  
11 data objects have INTENT(IN).
- 12 C1267 The *specification-part* of a pure subroutine subprogram shall specify the intents of all its non-  
13 pointer dummy data objects.
- 14 C1268 A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure subpro-  
15 gram shall not have the SAVE attribute.

### NOTE 12.42

Variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the SAVE attribute; there-  
fore, such initialization is also disallowed.

- 16 C1269 The *specification-part* of a pure subprogram shall specify that all its dummy procedures are  
17 pure.
- 18 C1270 If a procedure that is neither an intrinsic procedure nor a statement function is used in a context  
19 that requires it to be pure, then its interface shall be explicit in the scope of that use. The  
20 interface shall specify that the procedure is pure.
- 21 C1271 All internal subprograms in a pure subprogram shall be pure.
- 22 C1272 In a pure subprogram any designator with a base object that is in common or accessed by  
23 host or use association, is a dummy argument of a pure function, is a dummy argument with  
24 INTENT (IN) of a pure subroutine, or an object that is storage associated with any such variable,  
25 shall not be used in the following contexts:
- 26 (1) In a variable definition context(16.5.7);
  - 27 (2) As the *data-target* in a *pointer-assignment-stmt*;
  - 28 (3) As the *expr* corresponding to a component with the POINTER attribute in a *structure-*  
29 *constructor*.
  - 30 (4) As the *expr* of an intrinsic assignment statement in which the *variable* is of a derived type  
31 if the derived type has a pointer component at any level of component selection; or

### NOTE 12.43

This requires that processors be able to determine if entities with the PRIVATE attribute or with  
private components have a pointer component.

- 32 (5) As an actual argument associated with a dummy argument with INTENT (OUT) or IN-  
33 TENT (INOUT) or with the POINTER attribute.

- 1 C1273 Any procedure referenced in a pure subprogram, including one referenced via a defined operation,  
2 assignment, or finalization, shall be pure.
- 3 C1274 A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-*  
4 *stmt*, *rewind-stmt*, *flush-stmt*, *wait-stmt*, or *inquire-stmt*.
- 5 C1275 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-*  
6 *number* or *\**.
- 7 C1276 A pure subprogram shall not contain a *stop-stmt*.

**NOTE 12.44**

The above constraints are designed to guarantee that a pure procedure is free from side effects (modifications of data visible outside the procedure), which means that it is safe to reference it in constructs such as a FORALL *assignment-stmt* where there is no explicit order of evaluation.

The constraints on pure subprograms may appear complicated, but it is not necessary for a programmer to be intimately familiar with them. From the programmer's point of view, these constraints can be summarized as follows: a pure subprogram shall not contain any operation that could conceivably result in an assignment or pointer assignment to a common variable, a variable accessed by use or host association, or an INTENT (IN) dummy argument; nor shall a pure subprogram contain any operation that could conceivably perform any external file input/output or STOP operation. Note the use of the word conceivably; it is not sufficient for a pure subprogram merely to be side-effect free in practice. For example, a function that contains an assignment to a global variable but in a block that is not executed in any invocation of the function is nevertheless not a pure function. The exclusion of functions of this nature is required if strict compile-time checking is to be used.

It is expected that most library procedures will conform to the constraints required of pure procedures, and so can be declared pure and referenced in FORALL statements and constructs and within user-defined pure procedures.

**NOTE 12.45**

Pure subroutines are included to allow subroutine calls from pure procedures in a safe way, and to allow *forall-assignment-stmts* to be defined assignments. The constraints for pure subroutines are based on the same principles as for pure functions, except that side effects to INTENT (OUT), INTENT (INOUT), and pointer dummy arguments are permitted.

## 8 12.7 Elemental procedures

### 9 12.7.1 Elemental procedure declaration and interface

- 10 An **elemental procedure** is an elemental intrinsic procedure or a procedure that is defined by an  
11 elemental subprogram.
- 12 An elemental subprogram has the *prefix-spec* ELEMENTAL. An elemental subprogram is a pure sub-  
13 program. The PURE *prefix-spec* need not appear; it is implied by the ELEMENTAL *prefix-spec*. The  
14 following additional constraints apply to elemental subprograms.
- 15 C1277 All dummy arguments of an elemental procedure shall be scalar dummy data objects and shall  
16 not have the POINTER or ALLOCATABLE attribute.
- 17 C1278 The result variable of an elemental function shall be scalar and shall not have the POINTER or

- 1           ALLOCATABLE attribute.
- 2 C1279 In the scoping unit of an elemental subprogram, an object designator with a dummy argument  
3           as the base object shall not appear in a *specification-expr* except as the argument to one of the  
4           intrinsic functions BIT\_SIZE, KIND, LEN, or the numeric inquiry functions (13.5.6).

**NOTE 12.46**

An elemental subprogram is a pure subprogram and all of the constraints for pure subprograms also apply.

**NOTE 12.47**

The restriction on dummy arguments in specification expressions is imposed primarily to facilitate optimization. An example of usage that is not permitted is

```
ELEMENTAL REAL FUNCTION F (A, N)
  REAL, INTENT (IN) :: A
  INTEGER, INTENT (IN) :: N
  REAL :: WORK_ARRAY(N)   ! Invalid
  ...
END FUNCTION F
```

An example of usage that is permitted is

```
ELEMENTAL REAL FUNCTION F (A)
  REAL, INTENT (IN) :: A
  REAL (SELECTED_REAL_KIND (PRECISION (A)*2)) :: WORK
  ...
END FUNCTION F
```

## 5 12.7.2 Elemental function actual arguments and results

- 6 If a generic name or a specific name is used to reference an elemental function, the shape of the result is  
7 the same as the shape of the actual argument with the greatest rank. If there are no actual arguments  
8 or the actual arguments are all scalar, the result is scalar. For those elemental functions that have more  
9 than one argument, all actual arguments shall be conformable. In the array case, the values of the  
10 elements, if any, of the result are the same as would have been obtained if the scalar function had been  
11 applied separately, in any order, to corresponding elements of each array actual argument.

**NOTE 12.48**

An example of an elemental reference to the intrinsic function MAX:

if X and Y are arrays of shape (M, N),

```
MAX (X, 0.0, Y)
```

is an array expression of shape (M, N) whose elements have values

```
MAX (X(I, J), 0.0, Y(I, J)), I = 1, 2, ..., M, J = 1, 2, ..., N
```

### 1 **12.7.3 Elemental subroutine actual arguments**

2 An elemental subroutine is one that has only scalar dummy arguments, but may have array actual  
3 arguments. In a reference to an elemental subroutine, either all actual arguments shall be scalar, or  
4 all actual arguments associated with INTENT (OUT) and INTENT (INOUT) dummy arguments shall  
5 be arrays of the same shape and the remaining actual arguments shall be conformable with them. In  
6 the case that the actual arguments associated with INTENT (OUT) and INTENT (INOUT) dummy  
7 arguments are arrays, the values of the elements, if any, of the results are the same as would be obtained  
8 if the subroutine had been applied separately, in any order, to corresponding elements of each array  
9 actual argument.

10 In a reference to the intrinsic subroutine MVBITS, the actual arguments corresponding to the TO and  
11 FROM dummy arguments may be the same variable and may be associated scalar variables or associated  
12 array variables all of whose corresponding elements are associated. Apart from this, the actual arguments  
13 in a reference to an elemental subroutine must satisfy the restrictions of [12.4.1.7](#).



## 1 Section 13: Intrinsic procedures and modules

2 There are four classes of intrinsic procedures: inquiry functions, elemental functions, transformational  
3 functions, and subroutines. Some intrinsic subroutines are elemental.

4 There are three sets of standard intrinsic modules: a Fortran environment module (13.8.2), modules to  
5 support exception handling and IEEE arithmetic, and a module to support interoperability with the C  
6 programming language. The later two sets of modules are described in sections 14 and 15, respectively.

### 7 13.1 Classes of intrinsic procedures

8 An **inquiry function** is one whose result depends on the properties of one or more of its arguments  
9 instead of their values; in fact, these argument values may be undefined. Unless the description of  
10 an inquiry function states otherwise, these arguments are permitted to be unallocated allocatables or  
11 pointers that are not associated. An **elemental intrinsic function** is one that is specified for scalar  
12 arguments, but may be applied to array arguments as described in 12.7. All other intrinsic functions  
13 are **transformational functions**; they almost all have one or more array arguments or an array result.  
14 All standard intrinsic functions are pure.

15 The subroutine MOVE\_ALLOC and the elemental subroutine MVBITS are pure. No other standard  
16 intrinsic subroutine is pure.

#### NOTE 13.1

As with user-written elemental subroutines, an elemental intrinsic subroutine is pure. The effects  
of MOVE\_ALLOC are limited to its arguments. The remaining nonelemental intrinsic subroutines  
all have side effects (or reflect system side effects) and thus are not pure.

17 **Generic names** of standard intrinsic procedures are listed in 13.5. In most cases, generic functions  
18 accept arguments of more than one type and the type of the result is the same as the type of the  
19 arguments. **Specific names** of standard intrinsic functions with corresponding generic names are listed  
20 in 13.6.

21 If an intrinsic function is used as an actual argument to a procedure, its specific name shall be used and  
22 it may be referenced in the called procedure only with scalar arguments. If an intrinsic function does  
23 not have a specific name, it shall not be used as an actual argument (12.4.1.3).

24 Elemental intrinsic procedures behave as described in 12.7.

### 25 13.2 Arguments to intrinsic procedures

26 All intrinsic procedures may be invoked with either positional arguments or argument keywords (12.4).  
27 The descriptions in 13.5 through 13.7 give the argument keyword names and positional sequence for  
28 standard intrinsic procedures.

29 Many of the intrinsic procedures have optional arguments. These arguments are identified by the notation  
30 “optional” in the argument descriptions. In addition, the names of the optional arguments are enclosed  
31 in square brackets in description headings and in lists of procedures. The valid forms of reference for  
32 procedures with optional arguments are described in 12.4.1.

**NOTE 13.2**

The text CMPLX (X [, Y, KIND]) indicates that Y and KIND are both optional arguments. Valid reference forms include CMPLX(*x*), CMPLX(*x*, *y*), CMPLX(*x*, KIND=*kind*), CMPLX(*x*, *y*, *kind*), and CMPLX(KIND=*kind*, X=*x*, Y=*y*).

**NOTE 13.3**

Some intrinsic procedures impose additional requirements on their optional arguments. For example, SELECTED\_REAL\_KIND requires that at least one of its optional arguments be present, and RANDOM\_SEED requires that at most one of its optional arguments be present.

1 The dummy arguments of the specific intrinsic procedures in 13.6 have INTENT(IN). The dummy  
 2 arguments of the generic intrinsic procedures in 13.7 have INTENT(IN) if the intent is not stated  
 3 explicitly.

4 The actual argument associated with an intrinsic function dummy argument named KIND shall be a  
 5 scalar integer initialization expression and its value shall specify a representation method for the function  
 6 result that exists on the processor.

7 Intrinsic subroutines that assign values to arguments of type character do so in accordance with the  
 8 rules of intrinsic assignment (7.4.1.3).

### 9 13.2.1 The shape of array arguments

10 Unless otherwise specified, the inquiry intrinsic functions accept array arguments for which the shape  
 11 need not be defined. The shape of array arguments to transformational and elemental intrinsic functions  
 12 shall be defined.

### 13 13.2.2 Mask arguments

14 Some array intrinsic functions have an optional MASK argument of type logical that is used by the  
 15 function to select the elements of one or more arguments to be operated on by the function. Any  
 16 element not selected by the mask need not be defined at the time the function is invoked.

17 The MASK affects only the value of the function, and does not affect the evaluation, prior to invoking  
 18 the function, of arguments that are array expressions.

## 19 13.3 Bit model

20 The bit manipulation procedures are ten elemental functions and one elemental subroutine. Logical  
 21 operations on bits are provided by the elemental functions IOR, IAND, NOT, and IEOR; shift operations  
 22 are provided by the elemental functions ISHFT and ISHFTC; bit subfields may be referenced by the  
 23 elemental function IBITS and by the elemental subroutine MVBITS; single-bit processing is provided  
 24 by the elemental functions BTEST, IBSET, and IBCLR.

25 For the purposes of these procedures, a bit is defined to be a binary digit  $w$  located at position  $k$  of a  
 26 nonnegative integer scalar object based on a model nonnegative integer defined by

$$j = \sum_{k=0}^{z-1} w_k \times 2^k$$

27 and for which  $w_k$  may have the value 0 or 1. An example of a model number compatible with the  
 28 examples used in 13.4 would have  $z = 32$ , thereby defining a 32-bit integer.

- 1 An inquiry function `BIT_SIZE` is available to determine the parameter  $z$  of the model.  
 2 Effectively, this model defines an integer object to consist of  $z$  bits in sequence numbered from right  
 3 to left from 0 to  $z - 1$ . This model is valid only in the context of the use of such an object as the  
 4 argument or result of one of the bit manipulation procedures. In all other contexts, the model defined  
 5 for an integer in 13.4 applies. In particular, whereas the models are identical for  $w_{z-1} = 0$ , they do not  
 6 correspond for  $w_{z-1} = 1$  and the interpretation of bits in such objects is processor dependent.

## 7 13.4 Numeric models

- 8 The numeric manipulation and inquiry functions are described in terms of a model for the representation  
 9 and behavior of numbers on a processor. The model has parameters that are determined so as to make  
 10 the model best fit the machine on which the program is executed.  
 11 The model set for integer  $i$  is defined by

$$i = s \times \sum_{k=0}^{q-1} w_k \times r^k$$

- 12 where  $r$  is an integer exceeding one,  $q$  is a positive integer, each  $w_k$  is a nonnegative integer less than  $r$ ,  
 13 and  $s$  is +1 or -1.  
 14 The model set for real  $x$  is defined by

$$x = \begin{cases} 0 \text{ or} \\ s \times b^e \times \sum_{k=1}^p f_k \times b^{-k} \end{cases},$$

- 15 where  $b$  and  $p$  are integers exceeding one; each  $f_k$  is a nonnegative integer less than  $b$ , with  $f_1$  nonzero;  $s$   
 16 is +1 or -1; and  $e$  is an integer that lies between some integer maximum  $e_{\max}$  and some integer minimum  
 17  $e_{\min}$  inclusively. For  $x = 0$ , its exponent  $e$  and digits  $f_k$  are defined to be zero. The integer parameters  
 18  $r$  and  $q$  determine the set of model integers and the integer parameters  $b$ ,  $p$ ,  $e_{\min}$ , and  $e_{\max}$  determine  
 19 the set of model floating point numbers. The parameters of the integer and real models are available  
 20 for each integer and real type implemented by the processor. The parameters characterize the set of  
 21 available numbers in the definition of the model. The floating-point manipulation functions (13.5.10)  
 22 and numeric inquiry functions (13.5.6) provide values of some parameters and other values related to  
 23 the models.

### NOTE 13.4

Examples of these functions in 13.7 use the models

$$i = s \times \sum_{k=0}^{30} w_k \times 2^k$$

and

$$x = 0 \text{ or } s \times 2^e \times \left( \frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right), \quad -126 \leq e \leq 127$$

## 1 13.5 Standard generic intrinsic procedures

- 2 For all of the standard intrinsic procedures, the arguments shown are the names that shall be used for  
3 argument keywords if the keyword form is used for actual arguments.

### NOTE 13.5

For example, a reference to CMPLX may be written in the form CMPLX (A, B, M) or in the form CMPLX (Y = B, KIND = M, X = A).

### NOTE 13.6

Many of the argument keywords have names that are indicative of their usage. For example:

KIND	Describes the kind type parameter of the result
STRING, STRING_A	An arbitrary character string
BACK	Controls the direction of string scan (forward or backward)
MASK	A mask that may be applied to the arguments
DIM	A selected dimension of an array argument

### 4 13.5.1 Numeric functions

5	ABS (A)	Absolute value
6	AIMAG (Z)	Imaginary part of a complex number
7	AINT (A [, KIND])	Truncation to whole number
8	ANINT (A [, KIND])	Nearest whole number
9	CEILING (A [, KIND])	Least integer greater than or equal to number
10	CMPLX (X [, Y, KIND])	Conversion to complex type
11	CONJG (Z)	Conjugate of a complex number
12	DBLE (A)	Conversion to double precision real type
13	DIM (X, Y)	Positive difference
14	DPROD (X, Y)	Double precision real product
15	FLOOR (A [, KIND])	Greatest integer less than or equal to number
16	INT (A [, KIND])	Conversion to integer type
17	MAX (A1, A2 [, A3,...])	Maximum value
18	MIN (A1, A2 [, A3,...])	Minimum value
19	MOD (A, P)	Remainder function
20	MODULO (A, P)	Modulo function
21	NINT (A [, KIND])	Nearest integer
22	REAL (A [, KIND])	Conversion to real type
23	SIGN (A, B)	Transfer of sign

### 24 13.5.2 Mathematical functions

25	ACOS (X)	Arccosine
26	ASIN (X)	Arcsine
27	ATAN (X)	Arctangent
28	ATAN2 (Y, X)	Arctangent
29	COS (X)	Cosine
30	COSH (X)	Hyperbolic cosine
31	EXP (X)	Exponential
32	LOG (X)	Natural logarithm
33	LOG10 (X)	Common logarithm (base 10)
34	SIN (X)	Sine

1	SINH (X)	Hyperbolic sine
2	SQRT (X)	Square root
3	TAN (X)	Tangent
4	TANH (X)	Hyperbolic tangent
5	<b>13.5.3 Character functions</b>	
6	ACHAR (I [, KIND])	Character in given position in ASCII collating sequence
7		
8	ADJUSTL (STRING)	Adjust left
9	ADJUSTR (STRING)	Adjust right
10	CHAR (I [, KIND])	Character in given position in processor collating sequence
11		
12	IACHAR (C [, KIND])	Position of a character in ASCII collating sequence
13		
14	ICHAR (C [, KIND])	Position of a character in processor collating sequence
15		
16	INDEX (STRING, SUBSTRING [, BACK, KIND])	Starting position of a substring
17	LEN_TRIM (STRING [, KIND])	Length without trailing blank characters
18	LGE (STRING_A, STRING_B)	Lexically greater than or equal
19	LGT (STRING_A, STRING_B)	Lexically greater than
20	LLE (STRING_A, STRING_B)	Lexically less than or equal
21	LLT (STRING_A, STRING_B)	Lexically less than
22	MAX (A1, A2 [, A3,...])	Maximum value
23	MIN (A1, A2 [, A3,...])	Minimum value
24	REPEAT (STRING, NCOPIES)	Repeated concatenation
25	SCAN (STRING, SET [, BACK, KIND])	Scan a string for a character in a set
26	TRIM (STRING)	Remove trailing blank characters
27	VERIFY (STRING, SET [, BACK, KIND])	Verify the set of characters in a string
28	<b>13.5.4 Kind functions</b>	
29	KIND (X)	Kind type parameter value
30	SELECTED_CHAR_KIND (NAME)	Character kind type parameter value, given character set name
31		
32	SELECTED_INT_KIND (R)	Integer kind type parameter value, given range
33	SELECTED_REAL_KIND ([P, R])	Real kind type parameter value, given precision and range
34		
35	<b>13.5.5 Miscellaneous type conversion functions</b>	
36	LOGICAL (L [, KIND])	Convert between objects of type logical with different kind type parameters
37		
38	TRANSFER (SOURCE, MOLD [, SIZE])	Treat first argument as if of type of second argument
39		
40	<b>13.5.6 Numeric inquiry functions</b>	
41	DIGITS (X)	Number of significant digits of the model
42	EPSILON (X)	Number that is almost negligible compared to one
43		
44	HUGE (X)	Largest number of the model
45	MAXEXPONENT (X)	Maximum exponent of the model

1	MINEXPONENT (X)	Minimum exponent of the model
2	PRECISION (X)	Decimal precision
3	RADIX (X)	Base of the model
4	RANGE (X)	Decimal exponent range
5	TINY (X)	Smallest positive number of the model
6	<b>13.5.7 Array inquiry functions</b>	
7	LBOUND (ARRAY [, DIM, KIND])	Lower dimension bounds of an array
8	SHAPE (SOURCE [, KIND])	Shape of an array or scalar
9	SIZE (ARRAY [, DIM, KIND])	Total number of elements in an array
10	UBOUND (ARRAY [, DIM, KIND])	Upper dimension bounds of an array
11	<b>13.5.8 Other inquiry functions</b>	
12	ALLOCATED (ARRAY) or ALLOCATED (SCALAR)	Allocation status
13	ASSOCIATED (POINTER [, TARGET])	Association status inquiry or comparison
14	BIT_SIZE (I)	Number of bits of the model
15	EXTENDS_TYPE_OF (A, MOLD)	Same dynamic type or an extension
16	LEN (STRING [, KIND])	Length of a character entity
17	NEW_LINE (A)	Newline character
18	PRESENT (A)	Argument presence
19	SAME_TYPE_AS (A, B)	Same dynamic type
20	<b>13.5.9 Bit manipulation procedures</b>	
21	BTEST (I, POS)	Bit testing
22	IAND (I, J)	Bitwise AND
23	IBCLR (I, POS)	Clear bit
24	IBITS (I, POS, LEN)	Bit extraction
25	IBSET (I, POS)	Set bit
26	IEOR (I, J)	Exclusive OR
27	IOR (I, J)	Inclusive OR
28	ISHFT (I, SHIFT)	Logical shift
29	ISHFTC (I, SHIFT [, SIZE])	Circular shift
30	MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)	Copies bits from one integer to another
31	NOT (I)	Bitwise complement
32	<b>13.5.10 Floating-point manipulation functions</b>	
33	EXPONENT (X)	Exponent part of a model number
34	FRACTION (X)	Fractional part of a number
35	NEAREST (X, S)	Nearest different processor number in given direction
36		
37	RRSPACING (X)	Reciprocal of the relative spacing of model numbers near given number
38		
39	SCALE (X, I)	Multiply a real by its base to an integer power
40	SET_EXPONENT (X, I)	Set exponent part of a number
41	SPACING (X)	Absolute spacing of model numbers near given number
42		

### 1 13.5.11 Vector and matrix multiply functions

2	DOT_PRODUCT (VECTOR_A, VECTOR_B)	Dot product of two rank-one arrays
3	MATMUL (MATRIX_A, MATRIX_B)	Matrix multiplication

### 4 13.5.12 Array reduction functions

5	ALL (MASK [, DIM])	True if all values are true
6	ANY (MASK [, DIM])	True if any value is true
7	COUNT (MASK [, DIM, KIND])	Number of true elements in an array
8	MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])	Maximum value in an array
9	MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])	Minimum value in an array
10	PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])	Product of array elements
11	SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])	Sum of array elements

### 12 13.5.13 Array construction functions

13	CSHIFT (ARRAY, SHIFT [, DIM])	Circular shift
14	EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])	End-off shift
15	MERGE (TSOURCE, FSOURCE, MASK)	Merge under mask
16	PACK (ARRAY, MASK [, VECTOR])	Pack an array into an array of rank one under a mask
17	RESHAPE (SOURCE, SHAPE[, PAD, ORDER])	Reshape an array
18	SPREAD (SOURCE, DIM, NCOPIES)	Replicates array by adding a dimension
19	TRANSPOSE (MATRIX)	Transpose of an array of rank two
20	UNPACK (VECTOR, MASK, FIELD)	Unpack an array of rank one into an array under a mask
21		
22		

### 23 13.5.14 Array location functions

24	MAXLOC (ARRAY, DIM [, MASK, KIND]) or MAXLOC (ARRAY [, MASK, KIND])	Location of a maximum value in an array
25	MINLOC (ARRAY, DIM [, MASK, KIND]) or MINLOC (ARRAY [, MASK, KIND])	Location of a minimum value in an array

### 26 13.5.15 Null function

27	NULL ([MOLD])	Returns disassociated or unallocated result
----	---------------	---

### 28 13.5.16 Allocation transfer procedure

29	MOVE_ALLOC (FROM, TO)	Moves an allocation from one allocatable object to another
30		

### 31 13.5.17 Random number subroutines

32	RANDOM_NUMBER (HARVEST)	Returns pseudorandom number
----	-------------------------	-----------------------------

1     RANDOM\_SEED ([SIZE, PUT, GET])            Initializes or restarts the pseudorandom number  
2    generator

### 3   **13.5.18 System environment procedures**

4     COMMAND\_ARGUMENT\_COUNT ()                Number of command arguments  
5     CPU\_TIME (TIME)                          Obtain processor time  
6     DATE\_AND\_TIME ([DATE, TIME, ZONE,  
7         VALUES])                              Obtain date and time  
8     GET\_COMMAND ([COMMAND,  
9         LENGTH, STATUS])                    Returns entire command  
10    GET\_COMMAND\_ARGUMENT (NUMBER  
11        [, VALUE, LENGTH, STATUS])         Returns a command argument  
12    GET\_ENVIRONMENT\_VARIABLE (NAME  
      [, VALUE, LENGTH, STATUS,  
      TRIM\_NAME])                            Obtain the value of an environment variable  
13    IS\_IOSTAT\_END (I)                        Test for end-of-file value  
14    IS\_IOSTAT\_EOR (I)                        Test for end-of-record value  
15    SYSTEM\_CLOCK ([COUNT,  
16        COUNT\_RATE, COUNT\_MAX])            Obtain data from the system clock

### 13   **13.6 Specific names for standard intrinsic functions**

14    Except for AMAX0, AMIN0, MAX1, and MIN1, the result type of the specific function is the same as the  
15    result type of the corresponding generic function would be if it were invoked with the same arguments  
16    as the specific function.

Specific Name	Generic Name	Argument Type
ABS	ABS	default real
ACOS	ACOS	default real
AIMAG	AIMAG	default complex
AINT	AINT	default real
ALOG	LOG	default real
ALOG10	LOG10	default real
• AMAX0 (...)	REAL (MAX (...))	default integer
• AMAX1	MAX	default real
• AMIN0 (...)	REAL (MIN (...))	default integer
• AMIN1	MIN	default real
AMOD	MOD	default real
ANINT	ANINT	default real
ASIN	ASIN	default real
ATAN	ATAN	default real
ATAN2	ATAN2	default real
CABS	ABS	default complex
CCOS	COS	default complex
CEXP	EXP	default complex
CHAR	CHAR	default integer
CLOG	LOG	default complex
CONJG	CONJG	default complex
COS	COS	default real
COSH	COSH	default real
CSIN	SIN	default complex
CSQRT	SQRT	default complex

Specific Name	Generic Name	Argument Type
DABS	ABS	double precision real
DACOS	ACOS	double precision real
DASIN	ASIN	double precision real
DATAN	ATAN	double precision real
DATAN2	ATAN2	double precision real
DCOS	COS	double precision real
DCOSH	COSH	double precision real
DDIM	DIM	double precision real
DEXP	EXP	double precision real
DIM	DIM	default real
DINT	AINT	double precision real
DLOG	LOG	double precision real
DLOG10	LOG10	double precision real
• DMAX1	MAX	double precision real
• DMIN1	MIN	double precision real
DMOD	MOD	double precision real
DNINT	ANINT	double precision real
DPROD	DPROD	default real
DSIGN	SIGN	double precision real
DSIN	SIN	double precision real
DSINH	SINH	double precision real
DSQRT	SQRT	double precision real
DTAN	TAN	double precision real
DTANH	TANH	double precision real
EXP	EXP	default real
• FLOAT	REAL	default integer
IABS	ABS	default integer
• ICHAR	ICHAR	default character
IDIM	DIM	default integer
• IDINT	INT	double precision real
IDNINT	NINT	double precision real
• IFIX	INT	default real
INDEX	INDEX	default character
• INT	INT	default real
ISIGN	SIGN	default integer
LEN	LEN	default character
• LGE	LGE	default character
• LGT	LGT	default character
• LLE	LLE	default character
• LLT	LLT	default character
• MAX0	MAX	default integer
• MAX1 (...)	INT (MAX (...))	default real
• MIN0	MIN	default integer
• MIN1 (...)	INT (MIN (...))	default real
MOD	MOD	default integer
NINT	NINT	default real
• REAL	REAL	default integer
SIGN	SIGN	default real
SIN	SIN	default real
SINH	SINH	default real
• SNGL	REAL	double precision real
SQRT	SQRT	default real

Specific Name	Generic Name	Argument Type
TAN	TAN	default real
TANH	TANH	default real

1 A specific intrinsic function marked with a bullet (●) shall not be used as an actual argument or as a  
2 target in a procedure pointer assignment statement.

### 3 13.7 Specifications of the standard intrinsic procedures

4 Detailed specifications of the standard generic intrinsic procedures are provided here in alphabetical  
5 order.

6 The types and type parameters of standard intrinsic procedure arguments and function results are de-  
7 termined by these specifications. The “Argument(s)” paragraphs specify requirements on the actual  
8 arguments of the procedures. The result characteristics are sometimes specified in terms of the charac-  
9 teristics of dummy arguments. A program is prohibited from invoking an intrinsic procedure under cir-  
10 cumstances where a value to be returned in a subroutine argument or function result is outside the range  
11 of values representable by objects of the specified type and type parameters, unless the intrinsic module  
12 IEEE\_ARITHMETIC (section 14) is accessible and there is support for an infinite or a NaN result, as  
13 appropriate. If an infinite result is returned, the flag IEEE\_OVERFLOW or IEEE\_DIVIDE\_BY\_ZERO  
14 shall signal; if a NaN result is returned, the flag IEEE\_INVALID shall signal. If all results are normal,  
15 these flags shall have the same status as when the intrinsic procedure was invoked.

#### 16 13.7.1 ABS (A)

17 **Description.** Absolute value.

18 **Class.** Elemental function.

19 **Argument.** A shall be of type integer, real, or complex.

20 **Result Characteristics.** The same as A except that if A is complex, the result is real.

21 **Result Value.** If A is of type integer or real, the value of the result is  $|A|$ ; if A is complex with  
22 value  $(x, y)$ , the result is equal to a processor-dependent approximation to  $\sqrt{x^2 + y^2}$ .

23 **Example.** ABS ((3.0, 4.0)) has the value 5.0 (approximately).

#### 24 13.7.2 ACHAR (I [, KIND])

25 **Description.** Returns the character in a specified position of the ASCII collating sequence. It  
26 is the inverse of the IACHAR function.

27 **Class.** Elemental function.

28 **Arguments.**

29 I shall be of type integer.

30 KIND (optional) shall be a scalar integer initialization expression.

31 **Result Characteristics.** Character of length one. If KIND is present, the kind type parameter  
32 is that specified by the value of KIND; otherwise, the kind type parameter is that of default  
33 character type.

1       **Result Value.** If I has a value in the range  $0 \leq I \leq 127$ , the result is the character in  
2       position I of the ASCII collating sequence, provided the processor is capable of representing  
3       that character in the character type of the result; otherwise, the result is processor dependent.  
4       ACHAR (IACHAR (C)) shall have the value C for any character C capable of representation in  
5       the default character type.

6       **Example.** ACHAR (88) has the value 'X'.

### 7   **13.7.3    ACOS (X)**

8       **Description.** Arccosine (inverse cosine) function.

9       **Class.** Elemental function.

10      **Argument.** X shall be of type real with a value that satisfies the inequality  $|X| \leq 1$ .

11      **Result Characteristics.** Same as X.

12      **Result Value.** The result has a value equal to a processor-dependent approximation to arc-  
13      cos(X), expressed in radians. It lies in the range  $0 \leq \text{ACOS}(X) \leq \pi$ .

14      **Example.** ACOS (0.54030231) has the value 1.0 (approximately).

### 15   **13.7.4    ADJUSTL (STRING)**

16      **Description.** Adjust to the left, removing leading blanks and inserting trailing blanks.

17      **Class.** Elemental function.

18      **Argument.** STRING shall be of type character.

19      **Result Characteristics.** Character of the same length and kind type parameter as STRING.

20      **Result Value.** The value of the result is the same as STRING except that any leading blanks  
21      have been deleted and the same number of trailing blanks have been inserted.

22      **Example.** ADJUSTL (' WORD') has the value 'WORD '.

### 23   **13.7.5    ADJUSTR (STRING)**

24      **Description.** Adjust to the right, removing trailing blanks and inserting leading blanks.

25      **Class.** Elemental function.

26      **Argument.** STRING shall be of type character.

27      **Result Characteristics.** Character of the same length and kind type parameter as STRING.

28      **Result Value.** The value of the result is the same as STRING except that any trailing blanks  
29      have been deleted and the same number of leading blanks have been inserted.

30      **Example.** ADJUSTR ('WORD ') has the value ' WORD'.

### 31   **13.7.6    AIMAG (Z)**

32      **Description.** Imaginary part of a complex number.

33      **Class.** Elemental function.

1       **Argument.**  $Z$  shall be of type complex.

2       **Result Characteristics.** Real with the same kind type parameter as  $Z$ .

3       **Result Value.** If  $Z$  has the value  $(x, y)$ , the result has the value  $y$ .

4       **Example.** AIMAG ((2.0, 3.0)) has the value 3.0.

### 5   13.7.7    **AINT (A [, KIND])**

6       **Description.** Truncation to a whole number.

7       **Class.** Elemental function.

8       **Arguments.**

9       A                   shall be of type real.

10      KIND (optional) shall be a scalar integer initialization expression.

11      **Result Characteristics.** The result is of type real. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of A.

12      **Result Value.** If  $|A| < 1$ , AINT (A) has the value 0; if  $|A| \geq 1$ , AINT (A) has a value equal to the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

13      **Examples.** AINT (2.783) has the value 2.0. AINT (-2.783) has the value -2.0.

### 14   13.7.8    **ALL (MASK [, DIM])**

15      **Description.** Determine whether all values are true in MASK along dimension DIM.

16      **Class.** Transformational function.

17      **Arguments.**

18      MASK               shall be of type logical. It shall not be scalar.

19      DIM (optional) shall be scalar and of type integer with value in the range  $1 \leq \text{DIM} \leq n$ , where  $n$  is the rank of MASK. The corresponding actual argument shall not be an optional dummy argument.

20      **Result Characteristics.** The result is of type logical with the same kind type parameter as MASK. It is scalar if DIM is absent; otherwise, the result has rank  $n - 1$  and shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of MASK.

21      **Result Value.**

22      *Case (i):*       The result of ALL (MASK) has the value true if all elements of MASK are true or if MASK has size zero, and the result has value false if any element of MASK is false.

23      *Case (ii):*     If MASK has rank one, ALL(MASK,DIM) is equal to ALL(MASK). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of ALL (MASK, DIM) is equal to ALL (MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ ).

24      **Examples.**

25      *Case (i):*       The value of ALL ((/ .TRUE., .FALSE., .TRUE. /)) is false.

1           Case (ii):    If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$  and C is the array  $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$  then ALL (B /= C,  
2                           DIM = 1) is [true, false, false] and ALL (B /= C, DIM = 2) is [false, false].

### 3   **13.7.9    ALLOCATED (ARRAY) or ALLOCATED (SCALAR)**

4           **Description.** Indicate whether an allocatable variable is allocated.

5           **Class.** Inquiry function.

6           **Arguments.**

7           ARRAY           shall be an allocatable array.

8           SCALAR          shall be an allocatable scalar.

9           **Result Characteristics.** Default logical scalar.

10          **Result Value.** The result has the value true if the argument (ARRAY or SCALAR) is allocated  
11          and has the value false if the argument is unallocated.

### 12   **13.7.10   ANINT (A [, KIND])**

13          **Description.** Nearest whole number.

14          **Class.** Elemental function.

15          **Arguments.**

16          A                shall be of type real.

17          KIND (optional) shall be a scalar integer initialization expression.

18          **Result Characteristics.** The result is of type real. If KIND is present, the kind type parameter  
19          is that specified by the value of KIND; otherwise, the kind type parameter is that of A.

20          **Result Value.** The result is the integer nearest A, or if there are two integers equally near A,  
21          the result is whichever such integer has the greater magnitude.

22          **Examples.** ANINT (2.783) has the value 3.0. ANINT (-2.783) has the value -3.0.

### 23   **13.7.11   ANY (MASK [, DIM])**

24          **Description.** Determine whether any value is true in MASK along dimension DIM.

25          **Class.** Transformational function.

26          **Arguments.**

27          MASK            shall be of type logical. It shall not be scalar.

28          DIM (optional) shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
                          where  $n$  is the rank of MASK. The corresponding actual argument shall not  
                          be an optional dummy argument.

29          **Result Characteristics.** The result is of type logical with the same kind type parameter as  
30          MASK. It is scalar if DIM is absent; otherwise, the result has rank  $n - 1$  and shape  $(d_1, d_2,$   
31           $\dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of MASK.

1 **Result Value.**

2 *Case (i):* The result of ANY (MASK) has the value true if any element of MASK is true  
3 and has the value false if no elements are true or if MASK has size zero.

4 *Case (ii):* If MASK has rank one, ANY(MASK,DIM) is equal to ANY(MASK). Otherwise,  
5 the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of ANY(MASK, DIM)  
6 is equal to ANY(MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$ ).

7 **Examples.**

8 *Case (i):* The value of ANY ((/ .TRUE., .FALSE., .TRUE. /)) is true.

9 *Case (ii):* If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$  and C is the array  $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$  then ANY(B /= C,  
10 DIM = 1) is [true, false, true] and ANY(B /= C, DIM = 2) is [true, true].

11 **13.7.12 ASIN (X)**

12 **Description.** Arcsine (inverse sine) function.

13 **Class.** Elemental function.

14 **Argument.** X shall be of type real. Its value shall satisfy the inequality  $|X| \leq 1$ .

15 **Result Characteristics.** Same as X.

16 **Result Value.** The result has a value equal to a processor-dependent approximation to arc-  
17 sin(X), expressed in radians. It lies in the range  $-\pi/2 \leq \text{ASIN}(X) \leq \pi/2$ .

18 **Example.** ASIN (0.84147098) has the value 1.0 (approximately).

19 **13.7.13 ASSOCIATED (POINTER [, TARGET])**

20 **Description.** Returns the association status of its pointer argument or indicates whether the  
21 pointer is associated with the target.

22 **Class.** Inquiry function.

23 **Arguments.**

24 POINTER shall be a pointer. It may be of any type or may be a procedure pointer. Its  
pointer association status shall not be undefined.

25 TARGET shall be allowable as the *data-target* or *proc-target* in a pointer assignment  
(optional) statement (7.4.2) in which POINTER is *pointer-object*. If TARGET is a  
pointer then its pointer association status shall not be undefined.

26 **Result Characteristics.** Default logical scalar.

27 **Result Value.**

28 *Case (i):* If TARGET is absent, the result is true if POINTER is associated with a target  
29 and false if it is not.

30 *Case (ii):* If TARGET is present and is a procedure, the result is true if POINTER is  
31 associated with TARGET.

32 *Case (iii):* If TARGET is present and is a procedure pointer, the result is true if POINTER  
33 and TARGET are associated with the same procedure. If either POINTER or  
34 TARGET is disassociated, the result is false.

- 1        *Case (iv):*     If TARGET is present and is a scalar target, the result is true if TARGET is not a  
 2                                zero-sized storage sequence and the target associated with POINTER occupies the  
 3                                same storage units as TARGET. Otherwise, the result is false. If the POINTER  
 4                                is disassociated, the result is false.
- 5        *Case (v):*        If TARGET is present and is an array target, the result is true if the target  
 6                                associated with POINTER and TARGET have the same shape, are neither of  
 7                                size zero nor arrays whose elements are zero-sized storage sequences, and occupy  
 8                                the same storage units in array element order. Otherwise, the result is false. If  
 9                                POINTER is disassociated, the result is false.
- 10       *Case (vi):*        If TARGET is present and is a scalar pointer, the result is true if the target  
 11                                associated with POINTER and the target associated with TARGET are not zero-  
 12                                sized storage sequences and they occupy the same storage units. Otherwise, the  
 13                                result is false. If either POINTER or TARGET is disassociated, the result is  
 14                                false.
- 15       *Case (vii):*       If TARGET is present and is an array pointer, the result is true if the target  
 16                                associated with POINTER and the target associated with TARGET have the  
 17                                same shape, are neither of size zero nor arrays whose elements are zero-sized  
 18                                storage sequences, and occupy the same storage units in array element order.  
 19                                Otherwise, the result is false. If either POINTER or TARGET is disassociated,  
 20                                the result is false.

21       **Examples.** ASSOCIATED (CURRENT, HEAD) is true if CURRENT is associated with the  
 22       target HEAD. After the execution of

23                A\_PART => A (:N)

24       ASSOCIATED (A\_PART, A) is true if N is equal to UBOUND (A, DIM = 1). After the  
 25       execution of

26                NULLIFY (CUR); NULLIFY (TOP)

27       ASSOCIATED (CUR, TOP) is false.

### 28    13.7.14    ATAN (X)

29       **Description.** Arctangent (inverse tangent) function.

30       **Class.** Elemental function.

31       **Argument.** X shall be of type real.

32       **Result Characteristics.** Same as X.

33       **Result Value.** The result has a value equal to a processor-dependent approximation to arc-  
 34       tan(X), expressed in radians, that lies in the range  $-\pi/2 \leq \text{ATAN}(X) \leq \pi/2$ .

35       **Example.** ATAN (1.5574077) has the value 1.0 (approximately).

### 36    13.7.15    ATAN2 (Y, X)

37       **Description.** Arctangent (inverse tangent) function. The result is the principal value of the  
 38       argument of the nonzero complex number (X, Y).

39       **Class.** Elemental function.

40       **Arguments.**

1 Y shall be of type real.

2 X shall be of the same type and kind type parameter as Y. If Y has the value zero, X shall not have the value zero.

3 **Result Characteristics.** Same as X.

4 **Result Value.** The result has a value equal to a processor-dependent approximation to the  
5 principal value of the argument of the complex number (X, Y), expressed in radians. It lies in  
6 the range  $-\pi \leq \text{ATAN2}(Y,X) \leq \pi$  and is equal to a processor-dependent approximation to a  
7 value of  $\arctan(Y/X)$  if  $X \neq 0$ . If  $Y > 0$ , the result is positive. If  $Y = 0$  and  $X > 0$ , the result  
8 is Y. If  $Y = 0$  and  $X < 0$ , then the result is  $\pi$  if Y is positive real zero or the processor cannot  
9 distinguish between positive and negative real zero, and  $-\pi$  if Y is negative real zero. If  $Y < 0$ ,  
10 the result is negative. If  $X = 0$ , the absolute value of the result is  $\pi/2$ .

11 **Examples.**  $\text{ATAN2}(1.5574077, 1.0)$  has the value 1.0 (approximately). If Y has the value  
12  $\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}$  and X has the value  $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$ , the value of  $\text{ATAN2}(Y, X)$  is approximately  
13  $\begin{bmatrix} \frac{3\pi}{4} & \frac{\pi}{4} \\ -\frac{3\pi}{4} & -\frac{\pi}{4} \end{bmatrix}$ .

#### 14 13.7.16 BIT\_SIZE (I)

15 **Description.** Returns the number of bits  $z$  defined by the model of 13.3.

16 **Class.** Inquiry function.

17 **Argument.** I shall be of type integer. It may be a scalar or an array.

18 **Result Characteristics.** Scalar integer with the same kind type parameter as I.

19 **Result Value.** The result has the value of the number of bits  $z$  of the model integer defined  
20 for bit manipulation contexts in 13.3.

21 **Example.**  $\text{BIT\_SIZE}(1)$  has the value 32 if  $z$  of the model is 32.

#### 22 13.7.17 BTEST (I, POS)

23 **Description.** Tests a bit of an integer value.

24 **Class.** Elemental function.

25 **Arguments.**

26 I shall be of type integer.

27 POS shall be of type integer. It shall be nonnegative and be less than  $\text{BIT\_SIZE}(I)$ .

28 **Result Characteristics.** Default logical.

29 **Result Value.** The result has the value true if bit POS of I has the value 1 and has the value  
30 false if bit POS of I has the value 0. The model for the interpretation of an integer value as a  
31 sequence of bits is in 13.3.

32 **Examples.**  $\text{BTEST}(8, 3)$  has the value true. If A has the value  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ , the value of

1           BTEST (A, 2) is  $\begin{bmatrix} \text{false} & \text{false} \\ \text{false} & \text{true} \end{bmatrix}$  and the value of BTEST (2, A) is  $\begin{bmatrix} \text{true} & \text{false} \\ \text{false} & \text{false} \end{bmatrix}$ .

### 2   **13.7.18   CEILING (A [, KIND])**

3           **Description.** Returns the least integer greater than or equal to its argument.

4           **Class.** Elemental function.

5           **Arguments.**

6           A                   shall be of type real.

7           KIND (optional) shall be a scalar integer initialization expression.

8           **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
9 by the value of KIND; otherwise, the kind type parameter is that of default integer type.

10          **Result Value.** The result has a value equal to the least integer greater than or equal to A.

11          **Examples.** CEILING (3.7) has the value 4. CEILING (-3.7) has the value -3.

### 12   **13.7.19   CHAR (I [, KIND])**

13          **Description.** Returns the character in a given position of the processor collating sequence  
14 associated with the specified kind type parameter. It is the inverse of the ICHAR function.

15          **Class.** Elemental function.

16          **Arguments.**

17          I                   shall be of type integer with a value in the range  $0 \leq I \leq n - 1$ , where  $n$  is the  
18 number of characters in the collating sequence associated with the specified  
19 kind type parameter.

20          KIND (optional) shall be a scalar integer initialization expression.

21          **Result Characteristics.** Character of length one. If KIND is present, the kind type parameter  
22 is that specified by the value of KIND; otherwise, the kind type parameter is that of default  
23 character type.

24          **Result Value.** The result is the character in position I of the collating sequence associated  
25 with the specified kind type parameter. ICHAR (CHAR (I, KIND (C))) shall have the value I  
26 for  $0 \leq I \leq n - 1$  and CHAR (ICCHAR (C), KIND (C)) shall have the value C for any character  
27 C capable of representation in the processor.

28          **Example.** CHAR (88) has the value 'X' on a processor using the ASCII collating sequence.

### 29   **13.7.20   CMPLX (X [, Y, KIND])**

30          **Description.** Convert to complex type.

31          **Class.** Elemental function.

32          **Arguments.**

33          X                   shall be of type integer, real, or complex, or a *boz-literal-constant*.

Y (optional) shall be of type integer or real, or a *boz-literal-constant*. If X is of type complex, Y shall not be present, nor shall Y be associated with an optional dummy argument.

KIND (optional) shall be a scalar integer initialization expression.

**Result Characteristics.** The result is of type complex. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise, the kind type parameter is that of default real type.

**Result Value.** If Y is absent and X is not complex, it is as if Y were present with the value zero. If X is complex, it is as if X were real with the value REAL (X, KIND) and Y were present with the value AIMAG (X, KIND). CMPLX (X, Y, KIND) has the complex value whose real part is REAL (X, KIND) and whose imaginary part is REAL (Y, KIND).

**Example.** CMPLX (-3) has the value (-3.0, 0.0).

### 13.7.21 COMMAND\_ARGUMENT\_COUNT ()

**Description.** Returns the number of command arguments.

**Class.** Inquiry function.

**Arguments.** None.

**Result Characteristics.** Scalar default integer.

**Result Value.** The result value is equal to the number of command arguments available. If there are no command arguments available or if the processor does not support command arguments, then the result value is 0. If the processor has a concept of a command name, the command name does not count as one of the command arguments.

**Example.** See 13.7.42.

### 13.7.22 CONJG (Z)

**Description.** Conjugate of a complex number.

**Class.** Elemental function.

**Argument.** Z shall be of type complex.

**Result Characteristics.** Same as Z.

**Result Value.** If Z has the value  $(x, y)$ , the result has the value  $(x, -y)$ .

**Example.** CONJG ((2.0, 3.0)) has the value (2.0, -3.0).

### 13.7.23 COS (X)

**Description.** Cosine function.

**Class.** Elemental function.

**Argument.** X shall be of type real or complex.

**Result Characteristics.** Same as X.

**Result Value.** The result has a value equal to a processor-dependent approximation to  $\cos(X)$ .

1 If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is  
2 regarded as a value in radians.

3 **Example.** COS (1.0) has the value 0.54030231 (approximately).

#### 4 **13.7.24 COSH (X)**

5 **Description.** Hyperbolic cosine function.

6 **Class.** Elemental function.

7 **Argument.** X shall be of type real.

8 **Result Characteristics.** Same as X.

9 **Result Value.** The result has a value equal to a processor-dependent approximation to cosh(X).

10 **Example.** COSH (1.0) has the value 1.5430806 (approximately).

#### 11 **13.7.25 COUNT (MASK [, DIM, KIND])**

12 **Description.** Count the number of true elements of MASK along dimension DIM.

13 **Class.** Transformational function.

14 **Arguments.**

15 MASK shall be of type logical. It shall not be scalar.

16 DIM (optional) shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
where  $n$  is the rank of MASK. The corresponding actual argument shall not  
be an optional dummy argument.

17 KIND (optional) shall be a scalar integer initialization expression.

18 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that spec-  
ified by the value of KIND; otherwise the kind type parameter is that of default integer type.  
19 The result is scalar if DIM is absent; otherwise, the result has rank  $n - 1$  and shape  $(d_1, d_2,$   
20  $\dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of MASK.  
21

22 **Result Value.**

23 *Case (i):* The result of COUNT (MASK) has a value equal to the number of true elements  
24 of MASK or has the value zero if MASK has size zero.

25 *Case (ii):* If MASK has rank one, COUNT (MASK, DIM) has a value equal to that of  
26 COUNT (MASK). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1},$   
27  $\dots, s_n)$  of COUNT (MASK, DIM) is equal to COUNT (MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1},$   
28  $:, s_{\text{DIM}+1}, \dots, s_n)$ ).

29 **Examples.**

30 *Case (i):* The value of COUNT ((/ .TRUE., .FALSE., .TRUE. /)) is 2.

31 *Case (ii):* If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$  and C is the array  $\begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}$ , COUNT (B /= C,  
32 DIM = 1) is [2, 0, 1] and COUNT (B /= C, DIM = 2) is [1, 2].

#### 33 **13.7.26 CPU\_TIME (TIME)**

34 **Description.** Returns the processor time.

1        **Class.** Subroutine.

2        **Argument.** TIME shall be scalar and of type real. It is an INTENT(OUT) argument that is  
3 assigned a processor-dependent approximation to the processor time in seconds. If the processor  
4 cannot return a meaningful time, a processor-dependent negative value is returned.

5        **Example.**

```
6        REAL T1, T2
7        ...
8        CALL CPU_TIME(T1)
9        ... ! Code to be timed.
10       CALL CPU_TIME(T2)
11       WRITE (*,*) 'Time taken by code was ', T2-T1, ' seconds'
```

12       writes the processor time taken by a piece of code.

#### NOTE 13.7

A processor for which a single result is inadequate (for example, a parallel processor) might choose to provide an additional version for which time is an array.

The exact definition of time is left imprecise because of the variability in what different processors are able to provide. The primary purpose is to compare different algorithms on the same processor or discover which parts of a calculation are the most expensive.

The start time is left imprecise because the purpose is to time sections of code, as in the example.

Most computer systems have multiple concepts of time. One common concept is that of time expended by the processor for a given program. This might or might not include system overhead, and has no obvious connection to elapsed “wall clock” time.

### 13 13.7.27 CSHIFT (ARRAY, SHIFT [, DIM])

14        **Description.** Perform a circular shift on an array expression of rank one or perform circular  
15 shifts on all the complete rank one sections along a given dimension of an array expression of  
16 rank two or greater. Elements shifted out at one end of a section are shifted in at the other end.  
17 Different sections may be shifted by different amounts and in different directions.

18        **Class.** Transformational function.

19        **Arguments.**

20        ARRAY        may be of any type. It shall not be scalar.

21        SHIFT        shall be of type integer and shall be scalar if ARRAY has rank one; otherwise,  
it shall be scalar or of rank  $n - 1$  and of shape  $(d_1, d_2, \dots, d_{DIM-1}, d_{DIM+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

22        DIM (optional) shall be a scalar and of type integer with a value in the range  $1 \leq DIM \leq n$ ,  
where  $n$  is the rank of ARRAY. If DIM is omitted, it is as if it were present  
with the value 1.

23        **Result Characteristics.** The result is of the type and type parameters of ARRAY, and has  
24 the shape of ARRAY.

1 **Result Value.**

2 *Case (i):* If ARRAY has rank one, element  $i$  of the result is ARRAY (1 + MODULO ( $i$  +  
3 SHIFT - 1, SIZE (ARRAY))).

4 *Case (ii):* If ARRAY has rank greater than one, section ( $s_1, s_2, \dots, s_{DIM-1}, :, s_{DIM+1}, \dots,$   
5  $s_n$ ) of the result has a value equal to CSHIFT (ARRAY ( $s_1, s_2, \dots, s_{DIM-1}, :,$   
6  $s_{DIM+1}, \dots, s_n$ ),  $sh, 1$ ), where  $sh$  is SHIFT or SHIFT ( $s_1, s_2, \dots, s_{DIM-1}, s_{DIM+1},$   
7  $\dots, s_n$ ).

8 **Examples.**

9 *Case (i):* If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V circularly to the left by  
10 two positions is achieved by CSHIFT (V, SHIFT = 2) which has the value [3, 4,  
11 5, 6, 1, 2]; CSHIFT (V, SHIFT = -2) achieves a circular shift to the right by two  
12 positions and has the value [5, 6, 1, 2, 3, 4].

13 *Case (ii):* The rows of an array of rank two may all be shifted by the same amount or by

14 different amounts. If M is the array  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ , the value of

15 CSHIFT (M, SHIFT = -1, DIM = 2) is  $\begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}$ , and the value of

16 CSHIFT (M, SHIFT = (/ -1, 1, 0 /), DIM = 2) is  $\begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$ .

17 **13.7.28 DATE\_AND\_TIME ([DATE, TIME, ZONE, VALUES])**

18 **Description.** Returns data about the real-time clock and date in a form compatible with the  
19 representations defined in ISO 8601:1988.

20 **Class.** Subroutine.

21 **Arguments.**

22 DATE (optional) shall be scalar and of type default character. It is an INTENT (OUT) argu-  
ment. It is assigned a value of the form *CCYYMMDD*, where *CC* is the  
century, *YY* is the year within the century, *MM* is the month within the year,  
and *DD* is the day within the month. If there is no date available, DATE is  
assigned all blanks.

23 TIME (optional) shall be scalar and of type default character. It is an INTENT (OUT) argu-  
ment. It is assigned a value of the form *hhmmss.sss*, where *hh* is the hour of  
the day, *mm* is the minutes of the hour, and *ss.sss* is the seconds and  
milliseconds of the minute. If there is no clock available, TIME is assigned  
all blanks.

24 ZONE (optional) shall be scalar and of type default character. It is an INTENT (OUT) argu-  
ment. It is assigned a value of the form *+hhmm* or *-hhmm*, where *hh* and *mm*  
are the time difference with respect to Coordinated Universal Time (UTC) in  
hours and minutes, respectively. If this information is not available, ZONE  
is assigned all blanks.

25 VALUES (optional) shall be of type default integer and of rank one. It is an INTENT (OUT)  
argument. Its size shall be at least 8. The values returned in VALUES are  
as follows:

- 1           VALUES (1)    the year, including the century (for example, 1990), or –HUGE (0) if there is  
no date available;
- 2           VALUES (2)    the month of the year, or –HUGE (0) if there is no date available;
- 3           VALUES (3)    the day of the month, or –HUGE (0) if there is no date available;
- 4           VALUES (4)    the time difference with respect to Coordinated Universal Time (UTC) in  
minutes, or –HUGE (0) if this information is not available;
- 5           VALUES (5)    the hour of the day, in the range of 0 to 23, or –HUGE (0) if there is no clock;
- 6           VALUES (6)    the minutes of the hour, in the range 0 to 59, or –HUGE (0) if there is no  
clock;
- 7           VALUES (7)    the seconds of the minute, in the range 0 to 60, or –HUGE (0) if there is no  
clock;
- 8           VALUES (8)    the milliseconds of the second, in the range 0 to 999, or –HUGE (0) if there  
is no clock.

9           **Example.**

10           INTEGER DATE\_TIME (8)  
11           CHARACTER (LEN = 10) BIG\_BEN (3)  
12           CALL DATE\_AND\_TIME (BIG\_BEN (1), BIG\_BEN (2), &  
13                                   BIG\_BEN (3), DATE\_TIME)

14           If run in Geneva, Switzerland on April 12, 1985 at 15:27:35.5 with a system configured for the  
15           local time zone, this sample would have assigned the value 19850412 to BIG\_BEN (1), the value  
16           152735.500 to BIG\_BEN (2), the value +0100 to BIG\_BEN (3), and the value (/ 1985, 4, 12,  
17           60, 15, 27, 35, 500 /) to DATE\_TIME.

**NOTE 13.8**

UTC is defined by ISO 8601:1988.

18   **13.7.29   DBLE (A)**

19           **Description.** Convert to double precision real type.

20           **Class.** Elemental function.

21           **Argument.** A shall be of type integer, real, or complex, or a *boz-literal-constant*.

22           **Result Characteristics.** Double precision real.

23           **Result Value.** The result has the value REAL (A, KIND (0.0D0)).

24           **Example.** DBLE (–3) has the value –3.0D0.

25   **13.7.30   DIGITS (X)**

26           **Description.** Returns the number of significant digits of the model representing numbers of  
27           the same type and kind type parameter as the argument.

28           **Class.** Inquiry function.

1       **Argument.** X shall be of type integer or real. It may be a scalar or an array.

2       **Result Characteristics.** Default integer scalar.

3       **Result Value.** The result has the value  $q$  if X is of type integer and  $p$  if X is of type real,  
4 where  $q$  and  $p$  are as defined in 13.4 for the model representing numbers of the same type and  
5 kind type parameter as X.

6       **Example.** DIGITS (X) has the value 24 for real X whose model is as in Note 13.4.

### 7   **13.7.31    DIM (X, Y)**

8       **Description.** The difference  $X-Y$  if it is positive; otherwise zero.

9       **Class.** Elemental function.

10      **Arguments.**

11      X                   shall be of type integer or real.

12      Y                   shall be of the same type and kind type parameter as X.

13      **Result Characteristics.** Same as X.

14      **Result Value.** The value of the result is  $X-Y$  if  $X>Y$  and zero otherwise.

15      **Example.** DIM (-3.0, 2.0) has the value 0.0.

### 16   **13.7.32    DOT\_PRODUCT (VECTOR\_A, VECTOR\_B)**

17      **Description.** Performs dot-product multiplication of numeric or logical vectors.

18      **Class.** Transformational function.

19      **Arguments.**

20      VECTOR\_A       shall be of numeric type (integer, real, or complex) or of logical type. It shall  
be a rank-one array.

21      VECTOR\_B       shall be of numeric type if VECTOR\_A is of numeric type or of type logical  
if VECTOR\_A is of type logical. It shall be a rank-one array. It shall be of  
the same size as VECTOR\_A.

22      **Result Characteristics.** If the arguments are of numeric type, the type and kind type pa-  
23 rameter of the result are those of the expression VECTOR\_A \* VECTOR\_B determined by the  
24 types of the arguments according to 7.1.4.2. If the arguments are of type logical, the result is  
25 of type logical with the kind type parameter of the expression VECTOR\_A .AND. VECTOR\_B  
26 according to 7.1.4.2. The result is scalar.

27      **Result Value.**

28      *Case (i):*       If VECTOR\_A is of type integer or real, the result has the value SUM (VEC-  
29 TOR\_A\*VECTOR\_B). If the vectors have size zero, the result has the value zero.

30      *Case (ii):*     If VECTOR\_A is of type complex, the result has the value SUM (CONJG (VEC-  
31 TOR\_A)\*VECTOR\_B). If the vectors have size zero, the result has the value  
32 zero.

33      *Case (iii):*    If VECTOR\_A is of type logical, the result has the value ANY (VECTOR\_A  
34 .AND. VECTOR\_B). If the vectors have size zero, the result has the value false.

1           **Example.** DOT\_PRODUCT ((/ 1, 2, 3 /), (/ 2, 3, 4 /)) has the value 20.

### 2   **13.7.33   DPROD (X, Y)**

3           **Description.** Double precision real product.

4           **Class.** Elemental function.

5           **Arguments.**

6           X                   shall be of type default real.

7           Y                   shall be of type default real.

8           **Result Characteristics.** Double precision real.

9           **Result Value.** The result has a value equal to a processor-dependent approximation to the  
10           product of X and Y.

11          **Example.** DPROD (-3.0, 2.0) has the value -6.0D0.

### 12   **13.7.34   EOSHIFT (ARRAY, SHIFT [, BOUNDARY, DIM])**

13          **Description.** Perform an end-off shift on an array expression of rank one or perform end-off  
14          shifts on all the complete rank-one sections along a given dimension of an array expression of  
15          rank two or greater. Elements are shifted off at one end of a section and copies of a boundary  
16          value are shifted in at the other end. Different sections may have different boundary values and  
17          may be shifted by different amounts and in different directions.

18          **Class.** Transformational function.

19          **Arguments.**

20          ARRAY               may be of any type. It shall not be scalar.

            SHIFT               shall be of type integer and shall be scalar if ARRAY has rank one; otherwise,  
                                  it shall be scalar or of rank  $n - 1$  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$   
21                                    $\dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

            BOUNDARY           shall be of the same type and type parameters as ARRAY and shall be scalar  
                                  if ARRAY has rank one; otherwise, it shall be either scalar or of rank  $n - 1$   
                                  and of shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$ . BOUNDARY may be  
                                  omitted for the types in the following table and, in this case, it is as if it were  
                                  present with the scalar value shown.

Type of ARRAY	Value of BOUNDARY
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	false
Character ( <i>len</i> )	<i>len</i> blanks

            DIM (optional)      shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
23                                   where  $n$  is the rank of ARRAY. If DIM is omitted, it is as if it were present  
                                  with the value 1.

24          **Result Characteristics.** The result has the type, type parameters, and shape of ARRAY.

1 **Result Value.** Element  $(s_1, s_2, \dots, s_n)$  of the result has the value  $\text{ARRAY}(s_1, s_2, \dots, s_{\text{DIM}-1},$   
 2  $s_{\text{DIM}} + sh, s_{\text{DIM}+1}, \dots, s_n)$  where  $sh$  is SHIFT or SHIFT  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$   
 3 provided the inequality  $\text{LBOUND}(\text{ARRAY}, \text{DIM}) \leq s_{\text{DIM}} + sh \leq \text{UBOUND}(\text{ARRAY}, \text{DIM})$   
 4 holds and is otherwise BOUNDARY or BOUNDARY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$ .

5 **Examples.**

6 *Case (i):* If V is the array [1, 2, 3, 4, 5, 6], the effect of shifting V end-off to the left by 3  
 7 positions is achieved by EOSHIFT (V, SHIFT = 3), which has the value [4, 5,  
 8 6, 0, 0, 0]; EOSHIFT (V, SHIFT = -2, BOUNDARY = 99) achieves an end-off  
 9 shift to the right by 2 positions with the boundary value of 99 and has the value  
 10 [99, 99, 1, 2, 3, 4].

11 *Case (ii):* The rows of an array of rank two may all be shifted by the same amount or by  
 12 different amounts and the boundary elements can be the same or different. If M is

13 the array  $\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$ , then the value of EOSHIFT (M, SHIFT = -1, BOUND-

14 ARY = '\*', DIM = 2) is  $\begin{bmatrix} * & A & B \\ * & D & E \\ * & G & H \end{bmatrix}$ , and the value of EOSHIFT (M, SHIFT =

15 (/ -1, 1, 0 /), BOUNDARY = (/ '\*', '/' , '?' /), DIM = 2) is  $\begin{bmatrix} * & A & B \\ E & F & / \\ G & H & I \end{bmatrix}$ .

### 16 13.7.35 EPSILON (X)

17 **Description.** Returns a positive model number that is almost negligible compared to unity of  
 18 the model representing numbers of the same type and kind type parameter as the argument.

19 **Class.** Inquiry function.

20 **Argument.** X shall be of type real. It may be a scalar or an array.

21 **Result Characteristics.** Scalar of the same type and kind type parameter as X.

22 **Result Value.** The result has the value  $b^{1-p}$  where  $b$  and  $p$  are as defined in 13.4 for the model  
 23 representing numbers of the same type and kind type parameter as X.

24 **Example.** EPSILON (X) has the value  $2^{-23}$  for real X whose model is as in Note 13.4.

### 25 13.7.36 EXP (X)

26 **Description.** Exponential.

27 **Class.** Elemental function.

28 **Argument.** X shall be of type real or complex.

29 **Result Characteristics.** Same as X.

30 **Result Value.** The result has a value equal to a processor-dependent approximation to  $e^X$ . If  
 31 X is of type complex, its imaginary part is regarded as a value in radians.

32 **Example.** EXP (1.0) has the value 2.7182818 (approximately).

### 33 13.7.37 EXPONENT (X)

1       **Description.** Returns the exponent part of the argument when represented as a model number.

2       **Class.** Elemental function.

3       **Argument.** X shall be of type real.

4       **Result Characteristics.** Default integer.

5       **Result Value.** The result has a value equal to the exponent  $e$  of the model representation  
6 (13.4) for the value of X, provided X is nonzero and  $e$  is within the range for default integers. If  
7 X has the value zero, the result has the value zero. If X is an IEEE infinity or NaN, the result  
8 has the value HUGE(0).

9       **Examples.** EXPONENT (1.0) has the value 1 and EXPONENT (4.1) has the value 3 for  
10 reals whose model is as in Note 13.4.

### 11 13.7.38 EXTENDS\_TYPE\_OF (A, MOLD)

12       **Description.** Inquires whether the dynamic type of A is an extension type (4.5.6) of the  
13 dynamic type of MOLD.

14       **Class.** Inquiry function.

15       **Arguments.**

16       A                   shall be an object of extensible type. If it is a pointer, it shall not have an  
                                  undefined association status.

17       MOLD               shall be an object of extensible type. If it is a pointer, it shall not have an  
                                  undefined association status.

18       **Result Characteristics.** Default logical scalar.

19       **Result Value.** If MOLD is unlimited polymorphic and is either a disassociated pointer or  
20 unallocated allocatable, the result is true; otherwise if A is unlimited polymorphic and is either  
21 a disassociated pointer or unallocated allocatable, the result is false; otherwise the result is true  
22 if and only if the dynamic type of A is an extension type of the dynamic type of MOLD.

#### NOTE 13.9

The dynamic type of a disassociated pointer or unallocated allocatable is its declared type.

### 23 13.7.39 FLOOR (A [, KIND])

24       **Description.** Returns the greatest integer less than or equal to its argument.

25       **Class.** Elemental function.

26       **Arguments.**

27       A                   shall be of type real.

28       KIND (optional) shall be a scalar integer initialization expression.

29       **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
30 by the value of KIND; otherwise, the kind type parameter is that of default integer type.

31       **Result Value.** The result has a value equal to the greatest integer less than or equal to A.

1       **Examples.** FLOOR (3.7) has the value 3. FLOOR (-3.7) has the value -4.

## 2   **13.7.40    FRACTION (X)**

3       **Description.** Returns the fractional part of the model representation of the argument value.

4       **Class.** Elemental function.

5       **Argument.** X shall be of type real.

6       **Result Characteristics.** Same as X.

7       **Result Value.** The result has the value  $X \times b^{-e}$ , where  $b$  and  $e$  are as defined in 13.4 for the  
8 model representation of X. If X has the value zero, the result has the value zero. If X is an IEEE  
9 infinity, the result is that infinity. If X is an IEEE NaN, the result is that NaN.

10       **Example.** FRACTION (3.0) has the value 0.75 for reals whose model is as in Note 13.4.

## 11   **13.7.41    GET\_COMMAND ([COMMAND, LENGTH, STATUS])**

12       **Description.** Returns the entire command by which the program was invoked.

13       **Class.** Subroutine.

14       **Arguments.**

15       COMMAND       shall be scalar and of type default character. It is an INTENT(OUT) argu-  
(optional)           ment. It is assigned the entire command by which the program was invoked.  
If the command cannot be determined, COMMAND is assigned all blanks.

16       LENGTH        shall be scalar and of type default integer. It is an INTENT(OUT) argument.  
(optional)           It is assigned the significant length of the command by which the program  
was invoked. The significant length may include trailing blanks if the pro-  
cessor allows commands with significant trailing blanks. This length does  
not consider any possible truncation or padding in assigning the command  
to the COMMAND argument; in fact the COMMAND argument need not  
even be present. If the command length cannot be determined, a length of 0  
is assigned.

17       STATUS         shall be scalar and of type default integer. It is an INTENT(OUT) argument.  
(optional)           It is assigned the value -1 if the COMMAND argument is present and has  
a length less than the significant length of the command. It is assigned a  
processor-dependent positive value if the command retrieval fails. Otherwise  
it is assigned the value 0.

## 18   **13.7.42    GET\_COMMAND\_ARGUMENT (NUMBER [, VALUE, LENGTH, STA- TUS])**

19       **Description.** Returns a command argument.

20       **Class.** Subroutine.

21       **Arguments.**

22       NUMBER        shall be scalar and of type default integer. It is an INTENT(IN) argument.

It specifies the number of the command argument that the other arguments give information about. Useful values of NUMBER are those between 0 and the argument count returned by the COMMAND\_ARGUMENT\_COUNT intrinsic. Other values are allowed, but will result in error status return (see below).

Command argument 0 is defined to be the command name by which the program was invoked if the processor has such a concept. It is allowed to call the GET\_COMMAND\_ARGUMENT procedure for command argument number 0, even if the processor does not define command names or other command arguments.

The remaining command arguments are numbered consecutively from 1 to the argument count in an order determined by the processor.

VALUE  
(optional)

shall be scalar and of type default character. It is an INTENT(OUT) argument. It is assigned the value of the command argument specified by NUMBER. If the command argument value cannot be determined, VALUE is assigned all blanks.

LENGTH  
(optional)

shall be scalar and of type default integer. It is an INTENT(OUT) argument. It is assigned the significant length of the command argument specified by NUMBER. The significant length may include trailing blanks if the processor allows command arguments with significant trailing blanks. This length does not consider any possible truncation or padding in assigning the command argument value to the VALUE argument; in fact the VALUE argument need not even be present. If the command argument length cannot be determined, a length of 0 is assigned.

STATUS  
(optional)

shall be scalar and of type default integer. It is an INTENT(OUT) argument. It is assigned the value -1 if the VALUE argument is present and has a length less than the significant length of the command argument specified by NUMBER. It is assigned a processor-dependent positive value if the argument retrieval fails. Otherwise it is assigned the value 0.

#### NOTE 13.10

One possible reason for failure is that NUMBER is negative or greater than COMMAND\_ARGUMENT\_COUNT().

#### Example.

```

Program echo
  integer :: i
  character :: command*32, arg*128
  call get_command_argument(0, command)
  write (*,*) "Command name is: ", command
  do i = 1 , command_argument_count()
    call get_command_argument(i, arg)
    write (*,*) "Argument ", i, " is ", arg
  end do
end program echo

```

### 13.7.43 GET\_ENVIRONMENT\_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM\_NAME])

**Description.** Gets the value of an environment variable.

**Class.** Subroutine.

**Arguments.**

NAME shall be a scalar and of type default character. It is an INTENT(IN) argument. The interpretation of case is processor dependent

VALUE shall be a scalar of type default character. It is an INTENT(OUT) argument. It is assigned the value of the environment variable specified by NAME. VALUE is assigned all blanks if the environment variable does not exist or does not have a value or if the processor does not support environment variables.

LENGTH shall be a scalar of type default integer. It is an INTENT(OUT) argument. If the specified environment variable exists and has a value, LENGTH is set to the length of that value. Otherwise LENGTH is set to 0.

STATUS shall be scalar and of type default integer. It is an INTENT(OUT) argument. If the environment variable exists and either has no value or its value is successfully assigned to VALUE, STATUS is set to zero. STATUS is set to -1 if the VALUE argument is present and has a length less than the significant length of the environment variable. It is assigned the value 1 if the specified environment variable does not exist, or 2 if the processor does not support environment variables. Processor-dependent values greater than 2 may be returned for other error conditions.

TRIM\_NAME shall be a scalar of type logical. It is an INTENT(IN) argument. If TRIM\_NAME is present with the value false then trailing blanks in NAME are considered significant if the processor supports trailing blanks in environment variable names. Otherwise trailing blanks in NAME are not considered part of the environment variable's name.

### 13.7.44 HUGE (X)

**Description.** Returns the largest number of the model representing numbers of the same type and kind type parameter as the argument.

**Class.** Inquiry function.

**Argument.** X shall be of type integer or real. It may be a scalar or an array.

**Result Characteristics.** Scalar of the same type and kind type parameter as X.

**Result Value.** The result has the value  $r^q - 1$  if X is of type integer and  $(1 - b^{-p})b^{e_{\max}}$  if X is of type real, where  $r$ ,  $q$ ,  $b$ ,  $p$ , and  $e_{\max}$  are as defined in 13.4 for the model representing numbers of the same type and kind type parameter as X.

**Example.** HUGE (X) has the value  $(1 - 2^{-24}) \times 2^{127}$  for real X whose model is as in Note 13.4.

### 13.7.45 IACHAR (C [, KIND])

**Description.** Returns the position of a character in the ASCII collating sequence. This is the

1 inverse of the ACHAR function.

2 **Class.** Elemental function.

3 **Arguments.**

4 C shall be of type character and of length one.

5 KIND (optional) shall be a scalar integer initialization expression.

6 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
7 by the value of KIND; otherwise, the kind type parameter is that of default integer type.

8 **Result Value.** If C is in the collating sequence defined by the codes specified in ISO/IEC  
9 646:1991 (International Reference Version), the result is the position of C in that sequence and  
10 satisfies the inequality  $(0 \leq \text{IACHAR}(C) \leq 127)$ . A processor-dependent value is returned if C  
11 is not in the ASCII collating sequence. The results are consistent with the LGE, LGT, LLE,  
12 and LLT lexical comparison functions. For example, if LLE (C, D) is true, IACHAR (C) <=  
13 IACHAR (D) is true where C and D are any two characters representable by the processor.

14 **Example.** IACHAR ('X') has the value 88.

### 15 13.7.46 IAND (I, J)

16 **Description.** Performs a bitwise AND.

17 **Class.** Elemental function.

18 **Arguments.**

19 I shall be of type integer.

20 J shall be of type integer with the same kind type parameter as I.

21 **Result Characteristics.** Same as I.

22 **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to  
23 the following truth table:

I	J	IAND (I, J)
1	1	1
1	0	0
0	1	0
0	0	0

24 The model for the interpretation of an integer value as a sequence of bits is in [13.3](#).

25 **Example.** IAND (1, 3) has the value 1.

### 26 13.7.47 IBCLR (I, POS)

27 **Description.** Clears one bit to zero.

28 **Class.** Elemental function.

29 **Arguments.**

30 I shall be of type integer.

1           POS                   shall be of type integer. It shall be nonnegative and less than BIT\_SIZE (I).

2           **Result Characteristics.** Same as I.

3           **Result Value.** The result has the value of the sequence of bits of I, except that bit POS is  
4 zero. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

5           **Examples.** IBCLR (14, 1) has the result 12. If V has the value [1, 2, 3, 4], the value of  
6 IBCLR (POS = V, I = 31) is [29, 27, 23, 15].

### 7 **13.7.48 IBITS (I, POS, LEN)**

8           **Description.** Extracts a sequence of bits.

9           **Class.** Elemental function.

10          **Arguments.**

11          I                    shall be of type integer.

12          POS                 shall be of type integer. It shall be nonnegative and POS + LEN shall be  
less than or equal to BIT\_SIZE (I).

13          LEN                 shall be of type integer and nonnegative.

14          **Result Characteristics.** Same as I.

15          **Result Value.** The result has the value of the sequence of LEN bits in I beginning at bit POS,  
16 right-adjusted and with all other bits zero. The model for the interpretation of an integer value  
17 as a sequence of bits is in 13.3.

18          **Example.** IBITS (14, 1, 3) has the value 7.

### 19 **13.7.49 IBSET (I, POS)**

20          **Description.** Sets one bit to one.

21          **Class.** Elemental function.

22          **Arguments.**

23          I                    shall be of type integer.

24          POS                 shall be of type integer. It shall be nonnegative and less than BIT\_SIZE (I).

25          **Result Characteristics.** Same as I.

26          **Result Value.** The result has the value of the sequence of bits of I, except that bit POS is one.  
27 The model for the interpretation of an integer value as a sequence of bits is in 13.3.

28          **Examples.** IBSET (12, 1) has the value 14. If V has the value [1, 2, 3, 4], the value of  
29 IBSET (POS = V, I = 0) is [2, 4, 8, 16].

### 30 **13.7.50 ICHAR (C [, KIND])**

31          **Description.** Returns the position of a character in the processor collating sequence associated  
32 with the kind type parameter of the character. This is the inverse of the CHAR function.

33          **Class.** Elemental function.

1       **Arguments.**

2           C               shall be of type character and of length one. Its value shall be that of a  
3           character capable of representation in the processor.

4           KIND (optional) shall be a scalar integer initialization expression.

5       **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
6       by the value of KIND; otherwise, the kind type parameter is that of default integer type.

7       **Result Value.** The result is the position of C in the processor collating sequence associated  
8       with the kind type parameter of C and is in the range  $0 \leq \text{ICHAR}(C) \leq n - 1$ , where  $n$  is  
9       the number of characters in the collating sequence. For any characters C and D capable of  
10       representation in the processor,  $C \leq D$  is true if and only if  $\text{ICHAR}(C) \leq \text{ICHAR}(D)$  is true  
11       and  $C == D$  is true if and only if  $\text{ICHAR}(C) == \text{ICHAR}(D)$  is true.

12       **Example.** ICHAR ('X') has the value 88 on a processor using the ASCII collating sequence  
13       for the default character type.

13   **13.7.51    IEOR (I, J)**

14       **Description.** Performs a bitwise exclusive OR.

15       **Class.** Elemental function.

16       **Arguments.**

17       I                shall be of type integer.

18       J                shall be of type integer with the same kind type parameter as I.

19       **Result Characteristics.** Same as I.

20       **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to  
21       the following truth table:

I	J	IEOR (I, J)
1	1	0
1	0	1
0	1	1
0	0	0

22       The model for the interpretation of an integer value as a sequence of bits is in [13.3](#).

23       **Example.** IEOB (1, 3) has the value 2.

24   **13.7.52    INDEX (STRING, SUBSTRING [, BACK, KIND])**

25       **Description.** Returns the starting position of a substring within a string.

26       **Class.** Elemental function.

27       **Arguments.**

28       STRING           shall be of type character.

29       SUBSTRING       shall be of type character with the same kind type parameter as STRING.

1 BACK (optional) shall be of type logical.

2 KIND (optional) shall be a scalar integer initialization expression.

3 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
4 by the value of KIND; otherwise the kind type parameter is that of default integer type.

5 **Result Value.**

6 *Case (i):* If BACK is absent or has the value false, the result is the minimum positive value  
7 of I such that  $\text{STRING}(I : I + \text{LEN}(\text{SUBSTRING}) - 1) = \text{SUBSTRING}$  or zero if  
8 there is no such value. Zero is returned if  $\text{LEN}(\text{STRING}) < \text{LEN}(\text{SUBSTRING})$   
9 and one is returned if  $\text{LEN}(\text{SUBSTRING}) = 0$ .

10 *Case (ii):* If BACK is present with the value true, the result is the maximum value of  
11 I less than or equal to  $\text{LEN}(\text{STRING}) - \text{LEN}(\text{SUBSTRING}) + 1$  such that  
12  $\text{STRING}(I : I + \text{LEN}(\text{SUBSTRING}) - 1) = \text{SUBSTRING}$  or zero if there is  
13 no such value. Zero is returned if  $\text{LEN}(\text{STRING}) < \text{LEN}(\text{SUBSTRING})$  and  
14  $\text{LEN}(\text{STRING}) + 1$  is returned if  $\text{LEN}(\text{SUBSTRING}) = 0$ .

15 **Examples.** INDEX ('FORTRAN', 'R') has the value 3.

16 INDEX ('FORTRAN', 'R', BACK = .TRUE.) has the value 5.

### 17 13.7.53 INT (A [, KIND])

18 **Description.** Convert to integer type.

19 **Class.** Elemental function.

20 **Arguments.**

21 A shall be of type integer, real, or complex, or a *boz-literal-constant*.

22 KIND (optional) shall be a scalar integer initialization expression.

23 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
24 by the value of KIND; otherwise, the kind type parameter is that of default integer type.

25 **Result Value.**

26 *Case (i):* If A is of type integer,  $\text{INT}(A) = A$ .

27 *Case (ii):* If A is of type real, there are two cases: if  $|A| < 1$ , INT (A) has the value 0; if  
28  $|A| \geq 1$ , INT (A) is the integer whose magnitude is the largest integer that does  
29 not exceed the magnitude of A and whose sign is the same as the sign of A.

30 *Case (iii):* If A is of type complex,  $\text{INT}(A) = \text{INT}(\text{REAL}(A, \text{KIND}(A)))$ .

31 *Case (iv):* If A is a *boz-literal-constant*, it is treated as if it were an *int-literal-constant* with  
32 a *kind-param* that specifies the representation method with the largest decimal  
33 exponent range supported by the processor.

34 **Example.** INT (-3.7) has the value -3.

### 35 13.7.54 IOR (I, J)

36 **Description.** Performs a bitwise inclusive OR.

37 **Class.** Elemental function.

38 **Arguments.**

1 I shall be of type integer.

2 J shall be of type integer with the same kind type parameter as I.

3 **Result Characteristics.** Same as I.

4 **Result Value.** The result has the value obtained by combining I and J bit-by-bit according to  
5 the following truth table:

I	J	IOR (I, J)
1	1	1
1	0	1
0	1	1
0	0	0

6 The model for the interpretation of an integer value as a sequence of bits is in [13.3](#).

7 **Example.** IOR (5, 3) has the value 7.

### 8 **13.7.55 ISHFT (I, SHIFT)**

9 **Description.** Performs a logical shift.

10 **Class.** Elemental function.

11 **Arguments.**

12 I shall be of type integer.

13 SHIFT shall be of type integer. The absolute value of SHIFT shall be less than or  
equal to BIT\_SIZE (I).

14 **Result Characteristics.** Same as I.

15 **Result Value.** The result has the value obtained by shifting the bits of I by SHIFT positions.  
16 If SHIFT is positive, the shift is to the left; if SHIFT is negative, the shift is to the right;  
17 and if SHIFT is zero, no shift is performed. Bits shifted out from the left or from the right, as  
18 appropriate, are lost. Zeros are shifted in from the opposite end. The model for the interpretation  
19 of an integer value as a sequence of bits is in [13.3](#).

20 **Example.** ISHFT (3, 1) has the result 6.

### 21 **13.7.56 ISHFTC (I, SHIFT [, SIZE])**

22 **Description.** Performs a circular shift of the rightmost bits.

23 **Class.** Elemental function.

24 **Arguments.**

25 I shall be of type integer.

26 SHIFT shall be of type integer. The absolute value of SHIFT shall be less than or  
equal to SIZE.

27 SIZE (optional) shall be of type integer. The value of SIZE shall be positive and shall not  
exceed BIT\_SIZE (I). If SIZE is absent, it is as if it were present with the  
value of BIT\_SIZE (I).

1       **Result Characteristics.** Same as I.

2       **Result Value.** The result has the value obtained by shifting the SIZE rightmost bits of I  
3 circularly by SHIFT positions. If SHIFT is positive, the shift is to the left; if SHIFT is negative,  
4 the shift is to the right; and if SHIFT is zero, no shift is performed. No bits are lost. The  
5 unshifted bits are unaltered. The model for the interpretation of an integer value as a sequence  
6 of bits is in 13.3.

7       **Example.** ISHFTC (3, 2, 3) has the value 5.

### 8   **13.7.57   IS\_IOSTAT\_END (I)**

9       **Description.** Determine whether a value indicates an end-of-file condition.

10      **Class.** Elemental function.

11      **Argument.** I shall be of type integer.

12      **Result Characteristics.** Default logical.

13      **Result Value.** The result has the value true if and only if I is a value for the *scalar-int-variable*  
14 in an IOSTAT= specifier (9.10.4) that would indicate an end-of-file condition.

### 15   **13.7.58   IS\_IOSTAT\_EOR (I)**

16      **Description.** Determine whether a value indicates an end-of-record condition.

17      **Class.** Elemental function.

18      **Argument.** I shall be of type integer.

19      **Result Characteristics.** Default logical.

20      **Result Value.** The result has the value true if and only if I is a value for the *scalar-int-variable*  
21 in an IOSTAT= specifier (9.10.4) that would indicate an end-of-record condition.

### 22   **13.7.59   KIND (X)**

23      **Description.** Returns the value of the kind type parameter of X.

24      **Class.** Inquiry function.

25      **Argument.** X may be of any intrinsic type. It may be a scalar or an array.

26      **Result Characteristics.** Default integer scalar.

27      **Result Value.** The result has a value equal to the kind type parameter value of X.

28      **Example.** KIND (0.0) has the kind type parameter value of default real.

### 29   **13.7.60   LBOUND (ARRAY [, DIM, KIND])**

30      **Description.** Returns all the lower bounds or a specified lower bound of an array.

31      **Class.** Inquiry function.

32      **Arguments.**

33      ARRAY            may be of any type. It shall not be scalar. It shall not be an unallocated  
allocatable or a pointer that is not associated.

DIM (optional) shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ , where  $n$  is the rank of ARRAY. The corresponding actual argument shall not be an optional dummy argument.

KIND (optional) shall be a scalar integer initialization expression.

**Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type. The result is scalar if DIM is present; otherwise, the result is an array of rank one and size  $n$ , where  $n$  is the rank of ARRAY.

**Result Value.**

*Case (i):* If ARRAY is a whole array or array structure component and either ARRAY is an assumed-size array of rank DIM or dimension DIM of ARRAY has nonzero extent, LBOUND (ARRAY, DIM) has a value equal to the lower bound for subscript DIM of ARRAY. Otherwise the result value is 1.

*Case (ii):* LBOUND (ARRAY) has a value whose  $i$ th component is equal to LBOUND (ARRAY,  $i$ ), for  $i = 1, 2, \dots, n$ , where  $n$  is the rank of ARRAY.

**Examples.** If A is declared by the statement

```
REAL A (2:3, 7:10)
```

then LBOUND (A) is [2, 7] and LBOUND (A, DIM=2) is 7.

### 13.7.61 LEN (STRING [, KIND])

**Description.** Returns the length of a character entity.

**Class.** Inquiry function.

**Arguments.**

STRING shall be of type character. It may be a scalar or an array. If it is an unallocated allocatable or a pointer that is not associated, its length type parameter shall not be deferred.

KIND (optional) shall be a scalar integer initialization expression.

**Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that specified by the value of KIND; otherwise the kind type parameter is that of default integer type.

**Result Value.** The result has a value equal to the number of characters in STRING if it is scalar or in an element of STRING if it is an array.

**Example.** If C is declared by the statement

```
CHARACTER (11) C (100)
```

LEN (C) has the value 11.

### 13.7.62 LEN\_TRIM (STRING [, KIND])

**Description.** Returns the length of the character argument without counting trailing blank characters.

**Class.** Elemental function.

1       **Arguments.**

2       STRING           shall be of type character.

3       KIND (optional) shall be a scalar integer initialization expression.

4       **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
5       by the value of KIND; otherwise the kind type parameter is that of default integer type.6       **Result Value.** The result has a value equal to the number of characters remaining after any  
7       trailing blanks in STRING are removed. If the argument contains no nonblank characters, the  
8       result is zero.9       **Examples.** LEN\_TRIM (' A B ') has the value 4 and LEN\_TRIM (' ') has the value 0.10   **13.7.63    LGE (STRING\_A, STRING\_B)**11       **Description.** Test whether a string is lexically greater than or equal to another string, based  
12       on the ASCII collating sequence.13       **Class.** Elemental function.14       **Arguments.**

15       STRING\_A        shall be of type default character.

16       STRING\_B        shall be of type default character.

17       **Result Characteristics.** Default logical.18       **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
19       string were extended on the right with blanks to the length of the longer string. If either string  
20       contains a character not in the ASCII character set, the result is processor dependent. The  
21       result is true if the strings are equal or if STRING\_A follows STRING\_B in the ASCII collating  
22       sequence; otherwise, the result is false.**NOTE 13.11**

The result is true if both STRING\_A and STRING\_B are of zero length.

23       **Example.** LGE ('ONE', 'TWO') has the value false.24   **13.7.64    LGT (STRING\_A, STRING\_B)**25       **Description.** Test whether a string is lexically greater than another string, based on the ASCII  
26       collating sequence.27       **Class.** Elemental function.28       **Arguments.**

29       STRING\_A        shall be of type default character.

30       STRING\_B        shall be of type default character.

31       **Result Characteristics.** Default logical.32       **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
33       string were extended on the right with blanks to the length of the longer string. If either string

1 contains a character not in the ASCII character set, the result is processor dependent. The  
2 result is true if STRING\_A follows STRING\_B in the ASCII collating sequence; otherwise, the  
3 result is false.

**NOTE 13.12**

The result is false if both STRING\_A and STRING\_B are of zero length.

4 **Example.** LGT ('ONE', 'TWO') has the value false.

**13.7.65 LLE (STRING\_A, STRING\_B)**

6 **Description.** Test whether a string is lexically less than or equal to another string, based on  
7 the ASCII collating sequence.

8 **Class.** Elemental function.

9 **Arguments.**

10 STRING\_A shall be of type default character.

11 STRING\_B shall be of type default character.

12 **Result Characteristics.** Default logical.

13 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
14 string were extended on the right with blanks to the length of the longer string. If either  
15 string contains a character not in the ASCII character set, the result is processor dependent.  
16 The result is true if the strings are equal or if STRING\_A precedes STRING\_B in the ASCII  
17 collating sequence; otherwise, the result is false.

**NOTE 13.13**

The result is true if both STRING\_A and STRING\_B are of zero length.

18 **Example.** LLE ('ONE', 'TWO') has the value true.

**13.7.66 LLT (STRING\_A, STRING\_B)**

20 **Description.** Test whether a string is lexically less than another string, based on the ASCII  
21 collating sequence.

22 **Class.** Elemental function.

23 **Arguments.**

24 STRING\_A shall be of type default character.

25 STRING\_B shall be of type default character.

26 **Result Characteristics.** Default logical.

27 **Result Value.** If the strings are of unequal length, the comparison is made as if the shorter  
28 string were extended on the right with blanks to the length of the longer string. If either string  
29 contains a character not in the ASCII character set, the result is processor dependent. The  
30 result is true if STRING\_A precedes STRING\_B in the ASCII collating sequence; otherwise, the  
31 result is false.

**NOTE 13.14**

The result is false if both `STRING_A` and `STRING_B` are of zero length.

1       **Example.** `LLT ('ONE', 'TWO')` has the value true.

2   **13.7.67   LOG (X)**

3       **Description.** Natural logarithm.

4       **Class.** Elemental function.

5       **Argument.** `X` shall be of type real or complex. If `X` is real, its value shall be greater than  
6 zero. If `X` is complex, its value shall not be zero.

7       **Result Characteristics.** Same as `X`.

8       **Result Value.** The result has a value equal to a processor-dependent approximation to  $\log_e X$ .  
9 A result of type complex is the principal value with imaginary part  $\omega$  in the range  $-\pi \leq \omega \leq \pi$ .  
10 If the real part of `X` is less than zero and the imaginary part of `X` is zero, then the imaginary  
11 part of the result is  $\pi$  if the imaginary part of `X` is positive real zero or the processor cannot  
12 distinguish between positive and negative real zero, and  $-\pi$  if the imaginary part of `X` is negative  
13 real zero.

14       **Example.** `LOG (10.0)` has the value 2.3025851 (approximately).

15   **13.7.68   LOG10 (X)**

16       **Description.** Common logarithm.

17       **Class.** Elemental function.

18       **Argument.** `X` shall be of type real. The value of `X` shall be greater than zero.

19       **Result Characteristics.** Same as `X`.

20       **Result Value.** The result has a value equal to a processor-dependent approximation to  $\log_{10} X$ .

21       **Example.** `LOG10 (10.0)` has the value 1.0 (approximately).

22   **13.7.69   LOGICAL (L [, KIND])**

23       **Description.** Converts between kinds of logical.

24       **Class.** Elemental function.

25       **Arguments.**

26       `L` shall be of type logical.

27       `KIND` (optional) shall be a scalar integer initialization expression.

28       **Result Characteristics.** Logical. If `KIND` is present, the kind type parameter is that specified  
29 by the value of `KIND`; otherwise, the kind type parameter is that of default logical.

30       **Result Value.** The value is that of `L`.

31       **Example.** `LOGICAL (L .OR. .NOT. L)` has the value true and is of type default logical,

1 regardless of the kind type parameter of the logical variable L.

## 2 13.7.70 MATMUL (MATRIX\_A, MATRIX\_B)

3 **Description.** Performs matrix multiplication of numeric or logical matrices.

4 **Class.** Transformational function.

5 **Arguments.**

6 MATRIX\_A shall be of numeric type (integer, real, or complex) or of logical type. It shall be an array of rank one or two.

7 MATRIX\_B shall be of numeric type if MATRIX\_A is of numeric type and of logical type if MATRIX\_A is of logical type. It shall be an array of rank one or two. If MATRIX\_A has rank one, MATRIX\_B shall have rank two. If MATRIX\_B has rank one, MATRIX\_A shall have rank two. The size of the first (or only) dimension of MATRIX\_B shall equal the size of the last (or only) dimension of MATRIX\_A.

8 **Result Characteristics.** If the arguments are of numeric type, the type and kind type parameter of the result are determined by the types of the arguments as specified in 7.1.4.2 for the \* operator. If the arguments are of type logical, the result is of type logical with the kind type parameter of the arguments as specified in 7.1.4.2 for the .AND. operator. The shape of the result depends on the shapes of the arguments as follows:

9 *Case (i):* If MATRIX\_A has shape  $(n, m)$  and MATRIX\_B has shape  $(m, k)$ , the result has shape  $(n, k)$ .

10 *Case (ii):* If MATRIX\_A has shape  $(m)$  and MATRIX\_B has shape  $(m, k)$ , the result has shape  $(k)$ .

11 *Case (iii):* If MATRIX\_A has shape  $(n, m)$  and MATRIX\_B has shape  $(m)$ , the result has shape  $(n)$ .

12 **Result Value.**

13 *Case (i):* Element  $(i, j)$  of the result has the value SUM (MATRIX\_A  $(i, :)$  \* MATRIX\_B  $(:, j)$ ) if the arguments are of numeric type and has the value ANY (MATRIX\_A  $(i, :)$  .AND. MATRIX\_B  $(:, j)$ ) if the arguments are of logical type.

14 *Case (ii):* Element  $(j)$  of the result has the value SUM (MATRIX\_A  $(:, j)$  \* MATRIX\_B  $(:, j)$ ) if the arguments are of numeric type and has the value ANY (MATRIX\_A  $(:, j)$  .AND. MATRIX\_B  $(:, j)$ ) if the arguments are of logical type.

15 *Case (iii):* Element  $(i)$  of the result has the value SUM (MATRIX\_A  $(i, :)$  \* MATRIX\_B  $(:, :)$ ) if the arguments are of numeric type and has the value ANY (MATRIX\_A  $(i, :)$  .AND. MATRIX\_B  $(:, :)$ ) if the arguments are of logical type.

16 **Examples.** Let A and B be the matrices  $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}$  and  $\begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}$ ; let X and Y be the

17 vectors  $[1, 2]$  and  $[1, 2, 3]$ .

18 *Case (i):* The result of MATMUL (A, B) is the matrix-matrix product AB with the value  $\begin{bmatrix} 14 & 20 \\ 20 & 29 \end{bmatrix}$ .

19 *Case (ii):* The result of MATMUL (X, A) is the vector-matrix product XA with the value  $[5, 8, 11]$ .

1           *Case (iii):*   The result of MATMUL (A, Y) is the matrix-vector product AY with the value  
2                           [14, 20].

### 3   **13.7.71   MAX (A1, A2 [, A3, ...])**

4           **Description.** Maximum value.

5           **Class.** Elemental function.

6           **Arguments.**   The arguments shall all have the same type which shall be integer, real, or  
7                           character and they shall all have the same kind type parameter.

8           **Result Characteristics.** The type and kind type parameter of the result are the same as those  
9                           of the arguments. For arguments of character type, the length of the result is the length of the  
10                           longest argument.

11           **Result Value.** The value of the result is that of the largest argument. For arguments of  
12                           character type, the result is the value that would be selected by application of intrinsic relational  
13                           operators; that is, the collating sequence for characters with the kind type parameter of the  
14                           arguments is applied. If the selected argument is shorter than the longest argument, the result  
15                           is extended with blanks on the right to the length of the longest argument.

16           **Examples.** MAX (-9.0, 7.0, 2.0) has the value 7.0, MAX ("Z", "BB") has the value "Z ", and  
17                           MAX ((/"A", "Z"/), (/ "BB", "Y "/)) has the value (/ "BB", "Z "/).

### 18   **13.7.72   MAXEXPONENT (X)**

19           **Description.** Returns the maximum exponent of the model representing numbers of the same  
20                           type and kind type parameter as the argument.

21           **Class.** Inquiry function.

22           **Argument.** X shall be of type real. It may be a scalar or an array.

23           **Result Characteristics.** Default integer scalar.

24           **Result Value.** The result has the value  $e_{\max}$ , as defined in 13.4 for the model representing  
25                           numbers of the same type and kind type parameter as X.

26           **Example.** MAXEXPONENT (X) has the value 127 for real X whose model is as in Note 13.4.

### 27   **13.7.73   MAXLOC (ARRAY, DIM [, MASK, KIND]) or                   MAXLOC (ARRAY [, MASK, KIND])**

28           **Description.** Determine the location of the first element of ARRAY along dimension DIM  
29                           having the maximum value of the elements identified by MASK.

30           **Class.** Transformational function.

31           **Arguments.**

32           ARRAY           shall be of type integer, real, or character. It shall not be scalar.

33           DIM               shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
                              where  $n$  is the rank of ARRAY. The corresponding actual argument shall not  
                              be an optional dummy argument.

34           MASK (optional) shall be of type logical and shall be conformable with ARRAY.

1 KIND (optional) shall be a scalar integer initialization expression.

2 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
 3 by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM  
 4 is absent, the result is an array of rank one and of size equal to the rank of ARRAY; otherwise,  
 5 the result is of rank  $n - 1$  and shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$ , where  $(d_1, d_2, \dots, d_n)$   
 6 is the shape of ARRAY.

7 **Result Value.**

8 *Case (i):* The result of MAXLOC (ARRAY) is a rank-one array whose element values are  
 9 the values of the subscripts of an element of ARRAY whose value equals the  
 10 maximum value of all of the elements of ARRAY. The  $i$ th subscript returned lies  
 11 in the range 1 to  $e_i$ , where  $e_i$  is the extent of the  $i$ th dimension of ARRAY. If  
 12 more than one element has the maximum value, the element whose subscripts are  
 13 returned is the first such element, taken in array element order. If ARRAY has  
 14 size zero, all elements of the result are zero.

15 *Case (ii):* The result of MAXLOC (ARRAY, MASK = MASK) is a rank-one array whose  
 16 element values are the values of the subscripts of an element of ARRAY, corre-  
 17 sponding to a true element of MASK, whose value equals the maximum value of  
 18 all such elements of ARRAY. The  $i$ th subscript returned lies in the range 1 to  
 19  $e_i$ , where  $e_i$  is the extent of the  $i$ th dimension of ARRAY. If more than one such  
 20 element has the maximum value, the element whose subscripts are returned is  
 21 the first such element taken in array element order. If ARRAY has size zero or  
 22 every element of MASK has the value false, all elements of the result are zero.

23 *Case (iii):* If ARRAY has rank one, MAXLOC (ARRAY, DIM = DIM [, MASK = MASK])  
 24 is a scalar whose value is equal to that of the first element of MAXLOC (ARRAY [,  
 25 MASK = MASK]). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1},$   
 26  $\dots, s_n)$  of the result is equal to

27  $\text{MAXLOC (ARRAY (} s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n), \text{DIM}=1$   
 28  $[\text{, MASK = MASK (} s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) ])$ .

29 If ARRAY has type character, the result is the value that would be selected by application of  
 30 intrinsic relational operators; that is, the collating sequence for characters with the kind type  
 31 parameter of the arguments is applied.

32 **Examples.**

33 *Case (i):* The value of MAXLOC ((/ 2, 6, 4, 6 /)) is [2].

34 *Case (ii):* If A has the value  $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$ , MAXLOC (A, MASK = A ; 6) has the  
 35 value [3, 2]. Note that this is independent of the declared lower bounds for A.

36 *Case (iii):* The value of MAXLOC ((/ 5, -9, 3 /), DIM = 1) is 1. If B has the value  
 37  $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$ , MAXLOC (B, DIM = 1) is [2, 1, 2] and MAXLOC (B, DIM = 2)  
 38 is [2, 3]. Note that this is independent of the declared lower bounds for B.

### 39 13.7.74 MAXVAL (ARRAY, DIM [, MASK]) or MAXVAL (ARRAY [, MASK])

40 **Description.** Maximum value of the elements of ARRAY along dimension DIM corresponding  
 41 to the true elements of MASK.

42 **Class.** Transformational function.

1        **Arguments.**

2        ARRAY            shall be of type integer, real, or character. It shall not be scalar.

3        DIM                shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
where  $n$  is the rank of ARRAY. The corresponding actual argument shall not  
be an optional dummy argument.

4        MASK (optional) shall be of type logical and shall be conformable with ARRAY.

5        **Result Characteristics.** The result is of the same type and type parameters as ARRAY. It is  
6        scalar if DIM is absent; otherwise, the result has rank  $n-1$  and shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$   
7         $\dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.8        **Result Value.**9        *Case (i):*        The result of MAXVAL (ARRAY) has a value equal to the maximum value of  
10        all the elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size  
11        zero and type integer or real, the result has the value of the negative number of  
12        the largest magnitude supported by the processor for numbers of the type and  
13        kind type parameter of ARRAY. If ARRAY has size zero and type character, the  
14        result has the value of a string of characters of length LEN (ARRAY), with each  
15        character equal to CHAR (0, KIND = KIND (ARRAY)).16        *Case (ii):*        The result of MAXVAL (ARRAY, MASK = MASK) has a value equal to that of  
17        MAXVAL (PACK (ARRAY, MASK)).18        *Case (iii):*        The result of MAXVAL (ARRAY, DIM = DIM [,MASK = MASK]) has a value  
19        equal to that of MAXVAL (ARRAY [,MASK = MASK]) if ARRAY has rank one.  
20        Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of the result  
21        is equal to22                                MAXVAL (ARRAY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$   
23                                [, MASK = MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  ] ).24        If ARRAY has type character, the result is the value that would be selected by application of  
25        intrinsic relational operators; that is, the collating sequence for characters with the kind type  
26        parameter of the arguments is applied.27        **Examples.**28        *Case (i):*        The value of MAXVAL ((/ 1, 2, 3 /)) is 3.29        *Case (ii):*        MAXVAL (C, MASK = C < 0.0) finds the maximum of the negative elements of  
30        C.31        *Case (iii):*        If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ , MAXVAL (B, DIM = 1) is [2, 4, 6] and MAX-  
32        VAL (B, DIM = 2) is [5, 6].33        **13.7.75    MERGE (TSOURCE, FSOURCE, MASK)**34        **Description.** Choose alternative value according to the value of a mask.35        **Class.** Elemental function.36        **Arguments.**

37        TSOURCE            may be of any type.

38        FSOURCE            shall be of the same type and type parameters as TSOURCE.

1 MASK shall be of type logical.

2 **Result Characteristics.** Same as TSOURCE.

3 **Result Value.** The result is TSOURCE if MASK is true and FSOURCE otherwise.

4 **Examples.** If TSOURCE is the array  $\begin{bmatrix} 1 & 6 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ , FSOURCE is the array  $\begin{bmatrix} 0 & 3 & 2 \\ 7 & 4 & 8 \end{bmatrix}$   
 5 and MASK is the array  $\begin{bmatrix} T & \cdot & T \\ \cdot & \cdot & T \end{bmatrix}$ , where “T” represents true and “.” represents false, then  
 6 MERGE (TSOURCE, FSOURCE, MASK) is  $\begin{bmatrix} 1 & 3 & 5 \\ 7 & 4 & 6 \end{bmatrix}$ . The value of MERGE (1.0, 0.0,  
 7  $K > 0$ ) is 1.0 for  $K = 5$  and 0.0 for  $K = -2$ .

### 8 13.7.76 MIN (A1, A2 [, A3, ...])

9 **Description.** Minimum value.

10 **Class.** Elemental function.

11 **Arguments.** The arguments shall all be of the same type which shall be integer, real, or  
 12 character and they shall all have the same kind type parameter.

13 **Result Characteristics.** The type and kind type parameter of the result are the same as those  
 14 of the arguments. For arguments of character type, the length of the result is the length of the  
 15 longest argument.

16 **Result Value.** The value of the result is that of the smallest argument. For arguments  
 17 of character type, the result is the value that would be selected by application of intrinsic  
 18 relational operators; that is, the collating sequence for characters with the kind type parameter  
 19 of the arguments is applied. If the selected argument is shorter than the longest argument, the  
 20 result is extended with blanks on the right to the length of the longest argument.

21 **Example.** MIN (-9.0, 7.0, 2.0) has the value -9.0, MIN (“A”, ”YY”) has the value “A “, and  
 22 MIN ((/”Z”, ”A”/), (/”YY”, ”B “/)) has the value (/”YY”, ”A “/).

### 23 13.7.77 MINEXPONENT (X)

24 **Description.** Returns the minimum (most negative) exponent of the model representing num-  
 25 bers of the same type and kind type parameter as the argument.

26 **Class.** Inquiry function.

27 **Argument.** X shall be of type real. It may be a scalar or an array.

28 **Result Characteristics.** Default integer scalar.

29 **Result Value.** The result has the value  $e_{\min}$ , as defined in 13.4 for the model representing  
 30 numbers of the same type and kind type parameter as X.

31 **Example.** MINEXPONENT (X) has the value -126 for real X whose model is as in Note 13.4.

### 32 13.7.78 MINLOC (ARRAY, DIM [, MASK, KIND]) or 33 MINLOC (ARRAY [, MASK, KIND])

34 **Description.** Determine the location of the first element of ARRAY along dimension DIM  
 having the minimum value of the elements identified by MASK.

1       **Class.** Transformational function.

2       **Arguments.**

3       ARRAY           shall be of type integer, real, or character. It shall not be scalar.

4       DIM             shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
       where  $n$  is the rank of ARRAY. The corresponding actual argument shall not  
       be an optional dummy argument.

5       MASK (optional) shall be of type logical and shall be conformable with ARRAY.

6       KIND (optional) shall be a scalar integer initialization expression.

7       **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
 8       by the value of KIND; otherwise the kind type parameter is that of default integer type. If DIM  
 9       is absent, the result is an array of rank one and of size equal to the rank of ARRAY; otherwise,  
 10      the result is of rank  $n - 1$  and shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$ , where  $(d_1, d_2, \dots, d_n)$   
 11      is the shape of ARRAY.

12      **Result Value.**

13      *Case (i):*       The result of MINLOC (ARRAY) is a rank-one array whose element values are  
 14      the values of the subscripts of an element of ARRAY whose value equals the  
 15      minimum value of all the elements of ARRAY. The  $i$ th subscript returned lies  
 16      in the range 1 to  $e_i$ , where  $e_i$  is the extent of the  $i$ th dimension of ARRAY. If  
 17      more than one element has the minimum value, the element whose subscripts are  
 18      returned is the first such element, taken in array element order. If ARRAY has  
 19      size zero, all elements of the result are zero.

20      *Case (ii):*     The result of MINLOC (ARRAY, MASK = MASK) is a rank-one array whose  
 21      element values are the values of the subscripts of an element of ARRAY, corre-  
 22      sponding to a true element of MASK, whose value equals the minimum value of  
 23      all such elements of ARRAY. The  $i$ th subscript returned lies in the range 1 to  
 24       $e_i$ , where  $e_i$  is the extent of the  $i$ th dimension of ARRAY. If more than one such  
 25      element has the minimum value, the element whose subscripts are returned is the  
 26      first such element taken in array element order. If ARRAY has size zero or every  
 27      element of MASK has the value false, all elements of the result are zero.

28      *Case (iii):*    If ARRAY has rank one, MINLOC (ARRAY, DIM = DIM [, MASK = MASK]) is  
 29      a scalar whose value is equal to that of the first element of MINLOC (ARRAY [,  
 30      MASK = MASK]). Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1},$   
 31       $\dots, s_n)$  of the result is equal to

32                           MINLOC (ARRAY ( $s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$ ), DIM=1  
 33                           [, MASK = MASK ( $s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$ ) ] ).

34      If ARRAY has type character, the result is the value that would be selected by application of  
 35      intrinsic relational operators; that is, the collating sequence for characters with the kind type  
 36      parameter of the arguments is applied.

37      **Examples.**

38      *Case (i):*       The value of MINLOC ((/ 4, 3, 6, 3 /)) is [2].

39      *Case (ii):*     If A has the value  $\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{bmatrix}$ , MINLOC (A, MASK = A > -4) has  
 40      the value [1, 4]. Note that this is true even if A has a declared lower bound other  
 41      than 1.

1        *Case (iii):*    The value of MINLOC ((/ 5, -9, 3 /), DIM = 1) is 2. If B has the value  
 2                     $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$ , MINLOC (B, DIM = 1) is [1, 2, 1] and MINLOC (B, DIM = 2)  
 3                    is [3, 1]. Note that this is true even if B has a declared lower bound other than 1.

#### 4    **13.7.79    MINVAL (ARRAY, DIM [, MASK]) or MINVAL (ARRAY [, MASK])**

5        **Description.** Minimum value of all the elements of ARRAY along dimension DIM correspond-  
 6        ing to true elements of MASK.

7        **Class.** Transformational function.

8        **Arguments.**

9        ARRAY            shall be of type integer, real, or character. It shall not be scalar.

10        DIM                shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
 where  $n$  is the rank of ARRAY. The corresponding actual argument shall not  
 11        be an optional dummy argument.

12        MASK (optional) shall be of type logical and shall be conformable with ARRAY.

13        **Result Characteristics.** The result is of the same type and type parameters as ARRAY. It is  
 14        scalar if DIM is absent; otherwise, the result has rank  $n-1$  and shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1},$   
 $\dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

15        **Result Value.**

16        *Case (i):*        The result of MINVAL (ARRAY) has a value equal to the minimum value of all  
 17        the elements of ARRAY if the size of ARRAY is not zero. If ARRAY has size  
 18        zero and type integer or real, the result has the value of the positive number of  
 19        the largest magnitude supported by the processor for numbers of the type and  
 20        kind type parameter of ARRAY. If ARRAY has size zero and type character,  
 21        the result has the value of a string of characters of length LEN (ARRAY), with  
 22        each character equal to CHAR ( $n-1$ , KIND = KIND (ARRAY)), where  $n$  is the  
 23        number of characters in the collating sequence for characters with the kind type  
 24        parameter of ARRAY.

25        *Case (ii):*        The result of MINVAL (ARRAY, MASK = MASK) has a value equal to that of  
 26        MINVAL (PACK (ARRAY, MASK)).

27        *Case (iii):*        The result of MINVAL (ARRAY, DIM = DIM [, MASK = MASK]) has a value  
 28        equal to that of MINVAL (ARRAY [, MASK = MASK]) if ARRAY has rank one.  
 29        Otherwise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of the result  
 30        is equal to

31                    MINVAL (ARRAY ( $s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$ )  
 32                    [, MASK= MASK ( $s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n$ ) ] ).

33        If ARRAY has type character, the result is the value that would be selected by application of  
 34        intrinsic relational operators; that is, the collating sequence for characters with the kind type  
 35        parameter of the arguments is applied.

36        **Examples.**

37        *Case (i):*        The value of MINVAL ((/ 1, 2, 3 /)) is 1.

38        *Case (ii):*        MINVAL (C, MASK = C > 0.0) forms the minimum of the positive elements of  
 39        C.

1           *Case (iii):*    If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ , MINVAL (B, DIM = 1) is [1, 3, 5] and MINVAL (B,  
2                            DIM = 2) is [1, 2].

### 3   **13.7.80    MOD (A, P)**

4           **Description.** Remainder function.

5           **Class.** Elemental function.

6           **Arguments.**

7           A                   shall be of type integer or real.

8           P                   shall be of the same type and kind type parameter as A. P shall not be zero.

9           **Result Characteristics.** Same as A.

10          **Result Value.** The value of the result is  $A - \text{INT} (A/P) * P$ .

11          **Examples.** MOD (3.0, 2.0) has the value 1.0 (approximately). MOD (8, 5) has the value 3.  
12          MOD (-8, 5) has the value -3. MOD (8, -5) has the value 3. MOD (-8, -5) has the value -3.

### 13   **13.7.81    MODULO (A, P)**

14          **Description.** Modulo function.

15          **Class.** Elemental function.

16          **Arguments.**

17          A                   shall be of type integer or real.

18          P                   shall be of the same type and kind type parameter as A. P shall not be zero.

19          **Result Characteristics.** Same as A.

20          **Result Value.**

21          *Case (i):*        A is of type integer. MODULO (A, P) has the value R such that  $A = Q \times P + R$ ,  
22                            where Q is an integer, the inequalities  $0 \leq R < P$  hold if  $P > 0$ , and  $P < R \leq 0$   
23                            hold if  $P < 0$ .

24          *Case (ii):*     A is of type real. The value of the result is  $A - \text{FLOOR} (A / P) * P$ .

25          **Examples.** MODULO (8, 5) has the value 3. MODULO (-8, 5) has the value 2. MOD-  
26          ULO (8, -5) has the value -2. MODULO (-8, -5) has the value -3.

### 27   **13.7.82    MOVE\_ALLOC (FROM, TO)**

28          **Description.** Moves an allocation from one allocatable object to another.

29          **Class.** Subroutine.

30          **Arguments.**

31          FROM                may be of any type and rank. It shall be allocatable. It is an IN-  
                                  TENT(INOUT) argument.

TO shall be type compatible (5.1.1.2) with FROM and have the same rank. It shall be allocatable. It shall be polymorphic if FROM is polymorphic. It is an INTENT(OUT) argument. Each nondeferred parameter of the declared type of TO shall have the same value as the corresponding parameter of the declared type of FROM.

The allocation status of TO becomes unallocated if FROM is unallocated on entry to MOVE\_ALLOC. Otherwise, TO becomes allocated with dynamic type, type parameters, array bounds, and value identical to those that FROM had on entry to MOVE\_ALLOC.

If TO has the TARGET attribute, any pointer associated with FROM on entry to MOVE\_ALLOC becomes correspondingly associated with TO. If TO does not have the TARGET attribute, the pointer association status of any pointer associated with FROM on entry becomes undefined.

The allocation status of FROM becomes unallocated.

#### Example.

```

REAL, ALLOCATABLE :: GRID(:), TEMPGRID(:)
...
ALLOCATE(GRID(-N:N))          ! initial allocation of GRID
...
! "reallocation" of GRID to allow intermediate points
ALLOCATE(TEMPGRID(-2*N:2*N)) ! allocate bigger grid
TEMPGRID(::2)=GRID ! distribute values to new locations
CALL MOVE_ALLOC(TO=GRID, FROM=TEMPGRID)
                          ! old grid is deallocated because TO is
                          ! INTENT(OUT), and GRID then "takes over"
                          ! new grid allocation

```

#### NOTE 13.15

It is expected that the implementation of allocatable objects will typically involve descriptors to locate the allocated storage; MOVE\_ALLOC could then be implemented by transferring the contents of the descriptor for FROM to the descriptor for TO and clearing the descriptor for FROM.

### 13.7.83 MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

**Description.** Copies a sequence of bits from one data object to another.

**Class.** Elemental subroutine.

#### Arguments.

FROM shall be of type integer. It is an INTENT (IN) argument.

FROMPOS shall be of type integer and nonnegative. It is an INTENT (IN) argument. FROMPOS + LEN shall be less than or equal to BIT\_SIZE (FROM). The model for the interpretation of an integer value as a sequence of bits is in 13.3.

LEN shall be of type integer and nonnegative. It is an INTENT (IN) argument.

TO shall be a variable of type integer with the same kind type parameter value as FROM and may be associated with FROM (12.7.3). It is an INTENT (IN-OUT) argument. TO is defined by copying the sequence of bits of length LEN, starting at position FROMPOS of FROM to position TOPOS of TO. No other bits of TO are altered. On return, the LEN bits of TO starting at TOPOS are equal to the value that the LEN bits of FROM starting at FROMPOS had on entry. The model for the interpretation of an integer value as a sequence of bits is in 13.3.

TOPOS shall be of type integer and nonnegative. It is an INTENT (IN) argument. TOPOS + LEN shall be less than or equal to BIT\_SIZE (TO).

**Example.** If TO has the initial value 6, the value of TO after the statement CALL MVBITS (7, 2, 2, TO, 0) is 5.

### 13.7.84 NEAREST (X, S)

**Description.** Returns the nearest different machine-representable number in a given direction.

**Class.** Elemental function.

**Arguments.**

X shall be of type real.

S shall be of type real and not equal to zero.

**Result Characteristics.** Same as X.

**Result Value.** The result has a value equal to the machine-representable number distinct from X and nearest to it in the direction of the infinity with the same sign as S.

**Example.** NEAREST (3.0, 2.0) has the value  $3 + 2^{-22}$  on a machine whose representation is that of the model in Note 13.4.

#### NOTE 13.16

Unlike other floating-point manipulation functions, NEAREST operates on machine-representable numbers rather than model numbers. On many systems there are machine-representable numbers that lie between adjacent model numbers.

### 13.7.85 NEW\_LINE (A)

**Description.** Returns a newline character.

**Class.** Inquiry function.

**Argument.** A shall be of type character. It may be a scalar or an array.

**Result Characteristics.** Character scalar of length one with the same kind type parameter as A.

**Result Value.**

*Case (i):* If A is of the default character type and the character in position 10 of the ASCII collating sequence is representable in the default character set, then the result is ACHAR(10).

*Case (ii):* If A is of the ASCII character type or the ISO 10646 character type, then the

- 1 result is CHAR(10,KIND(A)).  
 2 *Case (iii):* Otherwise, the result is a processor-dependent character that represents a new-  
 3 line in output to files connected for formatted stream output if there is such a  
 4 character.  
 5 *Case (iv):* Otherwise, the result is the blank character.

### 6 13.7.86 NINT (A [, KIND])

7 **Description.** Nearest integer.

8 **Class.** Elemental function.

9 **Arguments.**

10 A shall be of type real.

11 KIND (optional) shall be a scalar integer initialization expression.

12 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
 13 by the value of KIND; otherwise, the kind type parameter is that of default integer type.

14 **Result Value.** The result is the integer nearest A, or if there are two integers equally near A,  
 15 the result is whichever such integer has the greater magnitude.

16 **Example.** NINT (2.783) has the value 3.

### 17 13.7.87 NOT (I)

18 **Description.** Performs a bitwise complement.

19 **Class.** Elemental function.

20 **Argument.** I shall be of type integer.

21 **Result Characteristics.** Same as I.

22 **Result Value.** The result has the value obtained by complementing I bit-by-bit according to  
 23 the following truth table:

I	NOT (I)
1	0
0	1

24 The model for the interpretation of an integer value as a sequence of bits is in [13.3](#).

25 **Example.** If I is represented by the string of bits 01010101, NOT (I) has the binary value  
 26 10101010.

### 27 13.7.88 NULL ([MOLD])

28 **Description.** Returns a disassociated pointer or designates an unallocated allocatable com-  
 29 ponent of a structure constructor.

30 **Class.** Transformational function.

31 **Argument.** MOLD shall be a pointer or allocatable. It may be of any type or may be a

1 procedure pointer. If MOLD is a pointer its pointer association status may be undefined,  
 2 disassociated, or associated. If MOLD is allocatable its allocation status may be allocated or  
 3 unallocated. It need not be defined with a value.

4 **Result Characteristics.** If MOLD is present, the characteristics are the same as MOLD. If  
 5 MOLD has deferred type parameters, those type parameters of the result are deferred.

6 If MOLD is absent, the characteristics of the result are determined by the entity with which the  
 7 reference is associated. See Table 13.1. MOLD shall not be absent in any other context. If any  
 8 type parameters of the contextual entity are deferred, those type parameters of the result are  
 9 deferred. If any type parameters of the contextual entity are assumed, MOLD shall be present.

10 If the context of the reference to NULL is an actual argument to a generic procedure, MOLD  
 11 shall be present if the type, type parameters, or rank is required to resolve the generic reference.  
 12 MOLD shall also be present if the reference appears as an actual argument corresponding to a  
 13 dummy argument with assumed character length.

Table 13.1: Characteristics of the result of NULL ( )

Appearance of NULL ( )	Type, type parameters, and rank of result:
right side of a pointer assignment	pointer on the left side
initialization for an object in a declaration	the object
default initialization for a component	the component
in a structure constructor	the corresponding component
as an actual argument	the corresponding dummy argument
in a DATA statement	the corresponding pointer object

14 **Result.** The result is a disassociated pointer or an unallocated allocatable entity.

15 **Examples.**

16 *Case (i):* REAL, POINTER, DIMENSION(:) :: VEC => NULL ( ) defines the initial  
 17 association status of VEC to be disassociated.

18 *Case (ii):* The MOLD argument is required in the following:

```

19     INTERFACE GEN
20         SUBROUTINE S1 (J, PI)
21             INTEGER J
22             INTEGER, POINTER :: PI
23         END SUBROUTINE S1
24         SUBROUTINE S2 (K, PR)
25             INTEGER K
26             REAL, POINTER :: PR
27         END SUBROUTINE S2
28     END INTERFACE
29     REAL, POINTER :: REAL_PTR
30     CALL GEN (7, NULL (REAL_PTR) )      ! Invokes S2

```

### 31 13.7.89 PACK (ARRAY, MASK [, VECTOR])

32 **Description.** Pack an array into an array of rank one under the control of a mask.

1        **Class.** Transformational function.

2        **Arguments.**

3        ARRAY                may be of any type. It shall not be scalar.

4        MASK                 shall be of type logical and shall be conformable with ARRAY.

5        VECTOR                shall be of the same type and type parameters as ARRAY and shall have  
(optional)                rank one. VECTOR shall have at least as many elements as there are true  
                                  elements in MASK. If MASK is scalar with the value true, VECTOR shall  
                                  have at least as many elements as there are in ARRAY.

6        **Result Characteristics.** The result is an array of rank one with the same type and type  
7        parameters as ARRAY. If VECTOR is present, the result size is that of VECTOR; otherwise,  
8        the result size is the number  $t$  of true elements in MASK unless MASK is scalar with the value  
9        true, in which case the result size is the size of ARRAY.

10       **Result Value.** Element  $i$  of the result is the element of ARRAY that corresponds to the  $i$ th  
11       true element of MASK, taking elements in array element order, for  $i = 1, 2, \dots, t$ . If VECTOR  
12       is present and has size  $n > t$ , element  $i$  of the result has the value VECTOR ( $i$ ), for  $i = t + 1,$   
13        $\dots, n$ .

14       **Examples.** The nonzero elements of an array M with the value  $\begin{bmatrix} 0 & 0 & 0 \\ 9 & 0 & 0 \\ 0 & 0 & 7 \end{bmatrix}$  may be “gath-  
15       ered” by the function PACK. The result of PACK (M, MASK = M /= 0) is [9, 7] and the result  
16       of PACK (M, M /= 0, VECTOR = (/ 2, 4, 6, 8, 10, 12 /)) is [9, 7, 6, 8, 10, 12].

### 17    13.7.90    PRECISION (X)

18       **Description.** Returns the decimal precision of the model representing real numbers with the  
19       same kind type parameter as the argument.

20       **Class.** Inquiry function.

21       **Argument.** X shall be of type real or complex. It may be a scalar or an array.

22       **Result Characteristics.** Default integer scalar.

23       **Result Value.** The result has the value  $\text{INT}((p - 1) * \text{LOG10}(b)) + k$ , where  $b$  and  $p$  are as  
24       defined in 13.4 for the model representing real numbers with the same value for the kind type  
25       parameter as X, and where  $k$  is 1 if  $b$  is an integral power of 10 and 0 otherwise.

26       **Example.** PRECISION (X) has the value  $\text{INT}(23 * \text{LOG10}(2.)) = \text{INT}(6.92\dots) = 6$  for real  
27       X whose model is as in Note 13.4.

### 28    13.7.91    PRESENT (A)

29       **Description.** Determine whether an optional argument is present.

30       **Class.** Inquiry function.

31       **Argument.** A shall be the name of an optional dummy argument that is accessible in the  
32       subprogram in which the PRESENT function reference appears. It may be of any type and it  
33       may be a pointer. It may be a scalar or an array. It may be a dummy procedure. The dummy  
34       argument A has no INTENT attribute.

1       **Result Characteristics.** Default logical scalar.

2       **Result Value.** The result has the value true if A is present (12.4.1.6) and otherwise has the  
3 value false.

#### 4   **13.7.92   PRODUCT (ARRAY, DIM [, MASK]) or PRODUCT (ARRAY [, MASK])**

5       **Description.** Product of all the elements of ARRAY along dimension DIM corresponding to  
6 the true elements of MASK.

7       **Class.** Transformational function.

8       **Arguments.**

9       ARRAY               shall be of type integer, real, or complex. It shall not be scalar.

10       DIM                 shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
where  $n$  is the rank of ARRAY. The corresponding actual argument shall not  
11 be an optional dummy argument.

12       MASK (optional) shall be of type logical and shall be conformable with ARRAY.

13       **Result Characteristics.** The result is of the same type and kind type parameter as AR-  
14 RAY. It is scalar if DIM is absent; otherwise, the result has rank  $n - 1$  and shape  $(d_1, d_2, \dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

15       **Result Value.**

16       Case (i):       The result of PRODUCT (ARRAY) has a value equal to a processor-dependent  
17 approximation to the product of all the elements of ARRAY or has the value one  
18 if ARRAY has size zero.

19       Case (ii):       The result of PRODUCT (ARRAY, MASK = MASK) has a value equal to a  
20 processor-dependent approximation to the product of the elements of ARRAY  
21 corresponding to the true elements of MASK or has the value one if there are no  
22 true elements.

23       Case (iii):      If ARRAY has rank one, PRODUCT (ARRAY, DIM = DIM [, MASK = MASK])  
24 has a value equal to that of PRODUCT (ARRAY [, MASK = MASK ]). Other-  
25 wise, the value of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  of PRODUCT (AR-  
26 RAY, DIM = DIM [, MASK = MASK]) is equal to

27                        PRODUCT (ARRAY  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  [, MASK =  
28 MASK  $(s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n)$  ] ).

29       **Examples.**

30       Case (i):       The value of PRODUCT ((/ 1, 2, 3 /)) is 6.

31       Case (ii):       PRODUCT (C, MASK = C > 0.0) forms the product of the positive elements of  
32 C.

33       Case (iii):      If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ , PRODUCT (B, DIM = 1) is [2, 12, 30] and  
34 PRODUCT (B, DIM = 2) is [15, 48].

#### 35   **13.7.93   RADIX (X)**

36       **Description.** Returns the base of the model representing numbers of the same type and kind  
37 type parameter as the argument.

38       **Class.** Inquiry function.

1       **Argument.** X shall be of type integer or real. It may be a scalar or an array.

2       **Result Characteristics.** Default integer scalar.

3       **Result Value.** The result has the value  $r$  if X is of type integer and the value  $b$  if X is of type  
4       real, where  $r$  and  $b$  are as defined in 13.4 for the model representing numbers of the same type  
5       and kind type parameter as X.

6       **Example.** RADIX (X) has the value 2 for real X whose model is as in Note 13.4.

### 7   13.7.94   RANDOM\_NUMBER (HARVEST)

8       **Description.** Returns one pseudorandom number or an array of pseudorandom numbers from  
9       the uniform distribution over the range  $0 \leq x < 1$ .

10      **Class.** Subroutine.

11      **Argument.** HARVEST shall be of type real. It is an INTENT (OUT) argument. It may be a  
12      scalar or an array variable. It is assigned pseudorandom numbers from the uniform distribution  
13      in the interval  $0 \leq x < 1$ .

14      **Examples.**

```
15       REAL X, Y (10, 10)
16       ! Initialize X with a pseudorandom number
17       CALL RANDOM_NUMBER (HARVEST = X)
18       CALL RANDOM_NUMBER (Y)
19       ! X and Y contain uniformly distributed random numbers
```

### 20 13.7.95   RANDOM\_SEED ([SIZE, PUT, GET])

21      **Description.** Restarts or queries the pseudorandom number generator used by RANDOM\_  
22      NUMBER.

23      **Class.** Subroutine.

24      **Arguments.** There shall either be exactly one or no arguments present.

25      SIZE (optional) shall be scalar and of type default integer. It is an INTENT (OUT) argument.  
It is assigned the number  $N$  of integers that the processor uses to hold the  
value of the seed.

26      PUT (optional) shall be a default integer array of rank one and size  $\geq N$ . It is an IN-  
TENT (IN) argument. It is used in a processor-dependent manner to compute  
the seed value accessed by the pseudorandom number generator.

27      GET (optional) shall be a default integer array of rank one and size  $\geq N$ . It is an IN-  
TENT (OUT) argument. It is assigned the current value of the seed.

28      If no argument is present, the processor assigns a processor-dependent value to the seed.

29      The pseudorandom number generator used by RANDOM\_NUMBER maintains a seed that is  
30      updated during the execution of RANDOM\_NUMBER and that may be specified or returned by  
31      RANDOM\_SEED. Computation of the seed from the argument PUT is performed in a processor-  
32      dependent manner. The value returned by GET need not be the same as the value specified

1 by PUT in an immediately preceding reference to RANDOM\_SEED. For example, following  
2 execution of the statements

```
3 CALL RANDOM_SEED (PUT=SEED1)
4 CALL RANDOM_SEED (GET=SEED2)
```

5 SEED2 need not equal SEED1. When the values differ, the use of either value as the PUT  
6 argument in a subsequent call to RANDOM\_SEED shall result in the same sequence of pseudo-  
7 random numbers being generated. For example, after execution of the statements

```
8 CALL RANDOM_SEED (PUT=SEED1)
9 CALL RANDOM_SEED (GET=SEED2)
10 CALL RANDOM_NUMBER (X1)
11 CALL RANDOM_SEED (PUT=SEED2)
12 CALL RANDOM_NUMBER (X2)
```

13 X2 equals X1.

14 **Examples.**

```
15 CALL RANDOM_SEED ! Processor initialization
16 CALL RANDOM_SEED (SIZE = K) ! Puts size of seed in K
17 CALL RANDOM_SEED (PUT = SEED (1 : K)) ! Define seed
18 CALL RANDOM_SEED (GET = OLD (1 : K)) ! Read current seed
```

### 19 13.7.96 RANGE (X)

20 **Description.** Returns the decimal exponent range of the model representing integer or real  
21 numbers with the same kind type parameter as the argument.

22 **Class.** Inquiry function.

23 **Argument.** X shall be of type integer, real, or complex. It may be a scalar or an array.

24 **Result Characteristics.** Default integer scalar.

25 **Result Value.**

26 *Case (i):* For an integer argument, the result has the value INT (LOG10 (HUGE(X))).

27 *Case (ii):* For a real argument, the result has the value INT (MIN (LOG10 (HUGE(X)),  
28 -LOG10 (TINY(X)))).

29 *Case (iii):* For a complex argument, the result has the value RANGE(REAL(X)).

30 **Examples.** RANGE (X) has the value 38 for real X whose model is as in Note 13.4, because  
31 in this case  $HUGE(X) = (1 - 2^{-24}) \times 2^{127}$  and  $TINY(X) = 2^{-127}$ .

### 32 13.7.97 REAL (A [, KIND])

33 **Description.** Convert to real type.

34 **Class.** Elemental function.

1       **Arguments.**2       A                   shall be of type integer, real, or complex, or a *boz-literal-constant*.

3       KIND (optional) shall be a scalar integer initialization expression.

4       **Result Characteristics.** Real.5       *Case (i):*       If A is of type integer or real and KIND is present, the kind type parameter is  
6                   that specified by the value of KIND. If A is of type integer or real and KIND is  
7                   not present, the kind type parameter is that of default real type.8       *Case (ii):*      If A is of type complex and KIND is present, the kind type parameter is that  
9                   specified by the value of KIND. If A is of type complex and KIND is not present,  
10                  the kind type parameter is the kind type parameter of A.11      *Case (iii):*     If A is a *boz-literal-constant* and KIND is present, the kind type parameter is  
12                  that specified by the value of KIND. If A is a *boz-literal-constant* and KIND is  
13                  not present, the kind type parameter is that of default real type.14      **Result Value.**15      *Case (i):*       If A is of type integer or real, the result is equal to a processor-dependent ap-  
16                  proximation to A.17      *Case (ii):*      If A is of type complex, the result is equal to a processor-dependent approximation  
18                  to the real part of A.19      *Case (iii):*     If A is a *boz-literal-constant*, the value of the result is equal to the value that a  
20                  variable of the same type and kind type parameters as the result would have if its  
21                  value were the bit pattern specified by the *boz-literal-constant*. The interpretation  
22                  of the value of the bit pattern is processor dependent.23      **Examples.** REAL (-3) has the value -3.0. REAL (Z) has the same kind type parameter and  
24      the same value as the real part of the complex variable Z.25   **13.7.98 REPEAT (STRING, NCOPIES)**26      **Description.** Concatenate several copies of a string.27      **Class.** Transformational function.28      **Arguments.**

29      STRING            shall be scalar and of type character.

30      NCOPIES           shall be scalar and of type integer. Its value shall not be negative.

31      **Result Characteristics.** Character scalar of length NCOPIES times that of STRING, with  
32      the same kind type parameter as STRING.33      **Result Value.** The value of the result is the concatenation of NCOPIES copies of STRING.34      **Examples.** REPEAT ('H', 2) has the value HH. REPEAT ('XYZ', 0) has the value of a  
35      zero-length string.36   **13.7.99 RESHAPE (SOURCE, SHAPE [, PAD, ORDER])**37      **Description.** Constructs an array of a specified shape from the elements of a given array.38      **Class.** Transformational function.

1     **Arguments.**

2     SOURCE            may be of any type. It shall not be scalar. If PAD is absent or of size zero, the size of SOURCE shall be greater than or equal to PRODUCT (SHAPE). The size of the result is the product of the values of the elements of SHAPE.

3     SHAPE             shall be of type integer, rank one, and constant size. Its size shall be positive and less than 8. It shall not have an element whose value is negative.

4     PAD (optional)    shall be of the same type and type parameters as SOURCE. PAD shall not be scalar.

5     ORDER (optional) shall be of type integer, shall have the same shape as SHAPE, and its value shall be a permutation of (1, 2, ...,  $n$ ), where  $n$  is the size of SHAPE. If absent, it is as if it were present with value (1, 2, ...,  $n$ ).

6     **Result Characteristics.** The result is an array of shape SHAPE (that is, SHAPE (RE-SHAPE (SOURCE, SHAPE, PAD, ORDER)) is equal to SHAPE) with the same type and type parameters as SOURCE.

9     **Result Value.** The elements of the result, taken in permuted subscript order ORDER (1), ..., ORDER ( $n$ ), are those of SOURCE in normal array element order followed if necessary by those of PAD in array element order, followed if necessary by additional copies of PAD in array element order.

13    **Examples.**    RESHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /)) has the value  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ . RE-SHAPE ((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 4 /), (/ 0, 0 /), (/ 2, 1 /)) has the value  $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$ .

15    **13.7.100 RRSPACING (X)**

16    **Description.** Returns the reciprocal of the relative spacing of model numbers near the argument value.

18    **Class.** Elemental function.

19    **Argument.** X shall be of type real.

20    **Result Characteristics.** Same as X.

21    **Result Value.** The result has the value  $|X \times b^{-e}| \times b^p$ , where  $b$ ,  $e$ , and  $p$  are as defined in 13.4 for the model representation of X. If X is an IEEE infinity, the result is zero. If X is an IEEE NaN, the result is that NaN.

24    **Example.**    RRSPACING (-3.0) has the value  $0.75 \times 2^{24}$  for reals whose model is as in Note 13.4.

26    **13.7.101 SAME\_TYPE\_AS (A, B)**

27    **Description.** Inquires whether the dynamic type of A is the same as the dynamic type of B.

28    **Class.** Inquiry function.

29    **Arguments.**

30    A                 shall be an object of extensible type. If it is a pointer, it shall not have an undefined association status.

1           B                   shall be an object of extensible type. If it is a pointer, it shall not have an  
 2                                undefined association status.

3           **Result Characteristics.** Default logical scalar.

4           **Result Value.** The result is true if and only if the dynamic type of A is the same as the  
 5                                dynamic type of B.

**NOTE 13.17**

The dynamic type of a disassociated pointer or unallocated allocatable is its declared type.

6   **13.7.102 SCALE (X, I)**

7           **Description.** Returns  $X \times b^I$  where  $b$  is the base of the model representation of X.

8           **Class.** Elemental function.

9           **Arguments.**

10          X                   shall be of type real.

11          I                   shall be of type integer.

12          **Result Characteristics.** Same as X.

13          **Result Value.** The result has the value  $X \times b^I$ , where  $b$  is defined in 13.4 for model numbers  
 14                                representing values of X, provided this result is within range; if not, the result is processor  
 15                                dependent.

16          **Example.** SCALE (3.0, 2) has the value 12.0 for reals whose model is as in Note 13.4.

17   **13.7.103 SCAN (STRING, SET [, BACK, KIND])**

18          **Description.** Scan a string for any one of the characters in a set of characters.

19          **Class.** Elemental function.

20          **Arguments.**

21          STRING            shall be of type character.

22          SET                shall be of type character with the same kind type parameter as STRING.

23          BACK (optional) shall be of type logical.

24          KIND (optional) shall be a scalar integer initialization expression.

25          **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
 26                                by the value of KIND; otherwise the kind type parameter is that of default integer type.

27          **Result Value.**

28          *Case (i):*        If BACK is absent or is present with the value false and if STRING contains at  
 29                                least one character that is in SET, the value of the result is the position of the  
 30                                leftmost character of STRING that is in SET.

31          *Case (ii):*       If BACK is present with the value true and if STRING contains at least one  
 32                                character that is in SET, the value of the result is the position of the rightmost  
 33                                character of STRING that is in SET.

1        Case (iii):    The value of the result is zero if no character of STRING is in SET or if the  
2                           length of STRING or SET is zero.

### 3        **Examples.**

4        Case (i):     SCAN ('FORTRAN', 'TR') has the value 3.

5        Case (ii):    SCAN ('FORTRAN', 'TR', BACK = .TRUE.) has the value 5.

6        Case (iii):   SCAN ('FORTRAN', 'BCD') has the value 0.

## 7    **13.7.104   SELECTED\_CHAR\_KIND (NAME)**

8        **Description.** Returns the value of the kind type parameter of the character set named by the  
9                           argument.

10       **Class.** Transformational function.

11       **Argument.** NAME shall be scalar and of type default character.

12       **Result Characteristics.** Default integer scalar.

13       **Result Value.** If NAME has the value DEFAULT, then the result has a value equal to that of  
14                           the kind type parameter of the default character type. If NAME has the value ASCII, then the  
15                           result has a value equal to that of the kind type parameter of the ASCII character type if the  
16                           processor supports such a type; otherwise the result has the value -1. If NAME has the value  
17                           ISO\_10646, then the result has a value equal to that of the kind type parameter of the ISO/IEC  
18                           10646-1:2000 UCS-4 character type if the processor supports such a type; otherwise the result  
19                           has the value -1. If NAME is a processor-defined name of some other character type supported  
20                           by the processor, then the result has a value equal to that of the kind type parameter of that  
21                           character type. If NAME is not the name of a supported character type, then the result has the  
22                           value -1. The NAME is interpreted without respect to case or trailing blanks.

23       **Examples.** SELECTED\_CHAR\_KIND ('ASCII') has the value 1 on a processor that uses 1  
24                           as the kind type parameter for the ASCII character set. The following subroutine produces a  
25                           Japanese date stamp.

```
26        SUBROUTINE create_date_string(string)
27            INTRINSIC date_and_time,selected_char_kind
28            INTEGER,PARAMETER :: ucs4 = selected_char_kind("ISO_10646")
29            CHARACTER(1,UCS4),PARAMETER :: nen=CHAR(INT(Z'5e74'),UCS4), & !year
30                gatsu=CHAR(INT(Z'6708'),UCS4), & !month
31                nichi=CHAR(INT(Z'65e5'),UCS4) !day
32            CHARACTER(len= *, kind= ucs4) string
33            INTEGER values(8)
34            CALL date_and_time(values=values)
35            WRITE(string,1) values(1),nen,values(2),gatsu,values(3),nichi
36            1 FORMAT(IO,A,IO,A,IO,A)
37            END SUBROUTINE"
```

## 38    **13.7.105   SELECTED\_INT\_KIND (R)**

39        **Description.** Returns a value of the kind type parameter of an integer type that represents all  
40                           integer values  $n$  with  $-10^R < n < 10^R$ .

1       **Class.** Transformational function.

2       **Argument.** R shall be scalar and of type integer.

3       **Result Characteristics.** Default integer scalar.

4       **Result Value.** The result has a value equal to the value of the kind type parameter of an  
 5 integer type that represents all values  $n$  in the range  $-10^R < n < 10^R$ , or if no such kind type  
 6 parameter is available on the processor, the result is  $-1$ . If more than one kind type parameter  
 7 meets the criterion, the value returned is the one with the smallest decimal exponent range,  
 8 unless there are several such values, in which case the smallest of these kind values is returned.

9       **Example.** Assume a processor supports two integer kinds, 32 with representation method  
 10  $r = 2$  and  $q = 31$ , and 64 with representation method  $r = 2$  and  $q = 63$ . On this processor  
 11 `SELECTED_INT_KIND(9)` has the value 32 and `SELECTED_INT_KIND(10)` has the value 64.

### 12 13.7.106 `SELECTED_REAL_KIND ([P, R])`

13       **Description.** Returns a value of the kind type parameter of a real type with decimal precision  
 14 of at least P digits and a decimal exponent range of at least R.

15       **Class.** Transformational function.

16       **Arguments.** At least one argument shall be present.

17       P (optional)       shall be scalar and of type integer.

18       R (optional)       shall be scalar and of type integer.

19       **Result Characteristics.** Default integer scalar.

20       **Result Value.** If P or R is absent, the result value is the same as if it were present with the  
 21 value zero. The result has a value equal to a value of the kind type parameter of a real type with  
 22 decimal precision, as returned by the function `PRECISION`, of at least P digits and a decimal  
 23 exponent range, as returned by the function `RANGE`, of at least R, or if no such kind type  
 24 parameter is available on the processor, the result is  $-1$  if the processor does not support a real  
 25 type with a precision greater than or equal to P but does support a real type with an exponent  
 26 range greater than or equal to R,  $-2$  if the processor does not support a real type with an  
 27 exponent range greater than or equal to R but does support a real type with a precision greater  
 28 than or equal to P,  $-3$  if the processor supports no real type with either of these properties, and  
 29  $-4$  if the processor supports real types for each separately but not together. If more than one  
 30 kind type parameter value meets the criteria, the value returned is the one with the smallest  
 31 decimal precision, unless there are several such values, in which case the smallest of these kind  
 32 values is returned.

33       **Example.** `SELECTED_REAL_KIND (6, 70)` has the value `KIND (0.0)` on a machine that  
 34 supports a default real approximation method with  $b = 16$ ,  $p = 6$ ,  $e_{\min} = -64$ , and  $e_{\max} = 63$   
 35 and does not have a less precise approximation method.

### 36 13.7.107 `SET_EXPONENT (X, I)`

37       **Description.** Returns the model number whose fractional part is the fractional part of the  
 38 model representation of X and whose exponent part is I.

39       **Class.** Elemental function.

40       **Arguments.**

1 X shall be of type real.

2 I shall be of type integer.

3 **Result Characteristics.** Same as X.

4 **Result Value.** The result has the value  $X \times b^{I-e}$ , where  $b$  and  $e$  are as defined in 13.4 for the  
5 model representation of X. If X has value zero, the result has value zero.

6 **Example.** SET\_EXPONENT (3.0, 1) has the value 1.5 for reals whose model is as in Note  
7 13.4.

### 8 13.7.108 SHAPE (SOURCE [, KIND])

9 **Description.** Returns the shape of an array or a scalar.

10 **Class.** Inquiry function.

11 **Arguments.**

SOURCE may be of any type. It may be a scalar or an array. It shall not be an  
12 unallocated allocatable or a pointer that is not associated. It shall not be an  
assumed-size array.

13 KIND (optional) shall be a scalar integer initialization expression.

14 **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
15 by the value of KIND; otherwise the kind type parameter is that of default integer type. The  
16 result is an array of rank one whose size is equal to the rank of SOURCE.

17 **Result Value.** The value of the result is the shape of SOURCE.

18 **Examples.** The value of SHAPE (A (2:5, -1:1) ) is [4, 3]. The value of SHAPE (3) is the  
19 rank-one array of size zero.

### 20 13.7.109 SIGN (A, B)

21 **Description.** Magnitude of A with the sign of B.

22 **Class.** Elemental function.

23 **Arguments.**

24 A shall be of type integer or real.

25 B shall be of the same type and kind type parameter as A.

26 **Result Characteristics.** Same as A.

27 **Result Value.**

28 *Case (i):* If  $B > 0$ , the value of the result is  $|A|$ .

29 *Case (ii):* If  $B < 0$ , the value of the result is  $-|A|$ .

30 *Case (iii):* If B is of type integer and  $B=0$ , the value of the result is  $|A|$ .

31 *Case (iv):* If B is of type real and is zero, then

32 (1) If the processor cannot distinguish between positive and negative real zero,  
33 the value of the result is  $|A|$ .

34 (2) If B is positive real zero, the value of the result is  $|A|$ .

1 (3) If B is negative real zero, the value of the result is  $-|A|$ .

2 **Example.** SIGN (-3.0, 2.0) has the value 3.0.

### 3 13.7.110 SIN (X)

4 **Description.** Sine function.

5 **Class.** Elemental function.

6 **Argument.** X shall be of type real or complex.

7 **Result Characteristics.** Same as X.

8 **Result Value.** The result has a value equal to a processor-dependent approximation to  $\sin(X)$ .  
 9 If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is  
 10 regarded as a value in radians.

11 **Example.** SIN (1.0) has the value 0.84147098 (approximately).

### 12 13.7.111 SINH (X)

13 **Description.** Hyperbolic sine function.

14 **Class.** Elemental function.

15 **Argument.** X shall be of type real.

16 **Result Characteristics.** Same as X.

17 **Result Value.** The result has a value equal to a processor-dependent approximation to  $\sinh(X)$ .

18 **Example.** SINH (1.0) has the value 1.1752012 (approximately).

### 19 13.7.112 SIZE (ARRAY [, DIM, KIND])

20 **Description.** Returns the extent of an array along a specified dimension or the total number  
 21 of elements in the array.

22 **Class.** Inquiry function.

23 **Arguments.**

24 ARRAY may be of any type. It shall not be scalar. It shall not be an unallocated  
 allocatable or a pointer that is not associated. If ARRAY is an assumed-size  
 array, DIM shall be present with a value less than the rank of ARRAY.

25 DIM (optional) shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
 where  $n$  is the rank of ARRAY.

26 KIND (optional) shall be a scalar integer initialization expression.

27 **Result Characteristics.** Integer scalar. If KIND is present, the kind type parameter is that  
 28 specified by the value of KIND; otherwise the kind type parameter is that of default integer  
 29 type.

30 **Result Value.** The result has a value equal to the extent of dimension DIM of ARRAY or, if  
 31 DIM is absent, the total number of elements of ARRAY.

1       **Examples.** The value of SIZE (A (2:5, -1:1), DIM=2) is 3. The value of SIZE (A (2:5, -1:1)  
2       ) is 12.

### 3   13.7.113   SPACING (X)

4       **Description.** Returns the absolute spacing of model numbers near the argument value.

5       **Class.** Elemental function.

6       **Argument.** X shall be of type real.

7       **Result Characteristics.** Same as X.

8       **Result Value.** If X does not have the value zero, the result has the value  $b^{\max(e-p, e_{\text{MIN}}-1)}$ ,  
9       where  $b$ ,  $e$ , and  $p$  are as defined in 13.4 for the model representation of X. If X has the value  
10       zero, the result is the same as that of TINY (X). If X is an IEEE infinity, the result is positive  
11       infinity. If X is an IEEE NaN, the result is that NaN.

12       **Example.** SPACING (3.0) has the value  $2^{-22}$  for reals whose model is as in Note 13.4.

### 13   13.7.114   SPREAD (SOURCE, DIM, NCOPIES)

14       **Description.** Replicates an array by adding a dimension. Broadcasts several copies of SOURCE  
15       along a specified dimension (as in forming a book from copies of a single page) and thus forms  
16       an array of rank one greater.

17       **Class.** Transformational function.

18       **Arguments.**

19       SOURCE        may be of any type. It may be a scalar or an array. The rank of SOURCE  
                      shall be less than 7.

20       DIM           shall be scalar and of type integer with value in the range  $1 \leq \text{DIM} \leq n + 1$ ,  
                      where  $n$  is the rank of SOURCE.

21       NCOPIES       shall be scalar and of type integer.

22       **Result Characteristics.** The result is an array of the same type and type parameters as  
23       SOURCE and of rank  $n + 1$ , where  $n$  is the rank of SOURCE.

24       *Case (i):*       If SOURCE is scalar, the shape of the result is (MAX (NCOPIES, 0)).

25       *Case (ii):*    If SOURCE is an array with shape  $(d_1, d_2, \dots, d_n)$ , the shape of the result is  
26        $(d_1, d_2, \dots, d_{\text{DIM}-1}, \text{MAX}(\text{NCOPIES}, 0), d_{\text{DIM}}, \dots, d_n)$ .

27       **Result Value.**

28       *Case (i):*       If SOURCE is scalar, each element of the result has a value equal to SOURCE.

29       *Case (ii):*    If SOURCE is an array, the element of the result with subscripts  $(r_1, r_2, \dots, r_{n+1})$   
30       has the value SOURCE  $(r_1, r_2, \dots, r_{\text{DIM}-1}, r_{\text{DIM}+1}, \dots, r_{n+1})$ .

31       **Examples.** If A is the array [2, 3, 4], SPREAD (A, DIM=1, NCOPIES=NC) is the array  
32       
$$\begin{bmatrix} 2 & 3 & 4 \\ 2 & 3 & 4 \\ 2 & 3 & 4 \end{bmatrix}$$
 if NC has the value 3 and is a zero-sized array if NC has the value 0.

### 33   13.7.115   SQRT (X)

34       **Description.** Square root.

1 **Class.** Elemental function.

2 **Argument.** X shall be of type real or complex. Unless X is complex, its value shall be greater  
3 than or equal to zero.

4 **Result Characteristics.** Same as X.

5 **Result Value.** The result has a value equal to a processor-dependent approximation to the  
6 square root of X. A result of type complex is the principal value with the real part greater than  
7 or equal to zero. When the real part of the result is zero, the imaginary part has the same sign  
8 as the imaginary part of X.

9 **Example.** SQRT (4.0) has the value 2.0 (approximately).

### 10 13.7.116 SUM (ARRAY, DIM [, MASK]) or SUM (ARRAY [, MASK])

11 **Description.** Sum all the elements of ARRAY along dimension DIM corresponding to the true  
12 elements of MASK.

13 **Class.** Transformational function.

14 **Arguments.**

15 ARRAY shall be of type integer, real, or complex. It shall not be scalar.

DIM shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
16 where  $n$  is the rank of ARRAY. The corresponding actual argument shall not  
be an optional dummy argument.

17 MASK (optional) shall be of type logical and shall be conformable with ARRAY.

18 **Result Characteristics.** The result is of the same type and kind type parameter as AR-  
19 RAY. It is scalar if DIM is absent; otherwise, the result has rank  $n - 1$  and shape  $(d_1, d_2,$   
20  $\dots, d_{\text{DIM}-1}, d_{\text{DIM}+1}, \dots, d_n)$  where  $(d_1, d_2, \dots, d_n)$  is the shape of ARRAY.

21 **Result Value.**

22 *Case (i):* The result of SUM (ARRAY) has a value equal to a processor-dependent approx-  
23 imation to the sum of all the elements of ARRAY or has the value zero if ARRAY  
24 has size zero.

25 *Case (ii):* The result of SUM (ARRAY, MASK = MASK) has a value equal to a processor-  
26 dependent approximation to the sum of the elements of ARRAY corresponding  
27 to the true elements of MASK or has the value zero if there are no true elements.

28 *Case (iii):* If ARRAY has rank one, SUM (ARRAY, DIM = DIM [, MASK = MASK]) has a  
29 value equal to that of SUM (ARRAY [, MASK = MASK]). Otherwise, the value  
30 of element  $(s_1, s_2, \dots, s_{\text{DIM}-1}, s_{\text{DIM}+1}, \dots, s_n)$  SUM (ARRAY, DIM = DIM [  
31 , MASK = MASK]) is equal to

32 
$$\text{SUM} (\text{ARRAY} (s_1, s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) [, \text{MASK} = \text{MASK} (s_1,$$
  
33  $s_2, \dots, s_{\text{DIM}-1}, :, s_{\text{DIM}+1}, \dots, s_n) ] ).$

34 **Examples.**

35 *Case (i):* The value of SUM ((/ 1, 2, 3 /)) is 6.

36 *Case (ii):* SUM (C, MASK = C > 0.0) forms the sum of the positive elements of C.

37 *Case (iii):* If B is the array  $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$ , SUM (B, DIM = 1) is [3, 7, 11] and SUM (B,  
38 DIM = 2) is [9, 12].

### 1 13.7.117 SYSTEM\_CLOCK ([COUNT, COUNT\_RATE, COUNT\_MAX])

2 **Description.** Returns numeric data from a real-time clock.

3 **Class.** Subroutine.

4 **Arguments.**

COUNT shall be scalar and of type integer. It is an INTENT (OUT) argument. It  
(optional) is assigned a processor-dependent value based on the current value of the  
processor clock, or -HUGE (COUNT) if there is no clock. The processor-  
dependent value is incremented by one for each clock count until the value  
COUNT\_MAX is reached and is reset to zero at the next count. It lies in the  
5 range 0 to COUNT\_MAX if there is a clock.

COUNT\_RATE shall be scalar and of type integer or real. It is an INTENT (OUT) argument.  
(optional) It is assigned a processor-dependent approximation to the number of processor  
6 clock counts per second, or zero if there is no clock.

COUNT\_MAX shall be scalar and of type integer. It is an INTENT(OUT) argument. It is  
(optional) assigned the maximum value that COUNT can have, or zero if there is no  
7 clock.

8 **Example.** If the processor clock is a 24-hour clock that registers time at approximately  
9 18.20648193 ticks per second, at 11:30 A.M. the reference

10 CALL SYSTEM\_CLOCK (COUNT = C, COUNT\_RATE = R, COUNT\_MAX = M)

11 defines  $C = (11 \times 3600 + 30 \times 60) \times 18.20648193 = 753748$ ,  $R = 18.20648193$ , and  $M =$   
12  $24 \times 3600 \times 18.20648193 - 1 = 1573039$ .

### 13 13.7.118 TAN (X)

14 **Description.** Tangent function.

15 **Class.** Elemental function.

16 **Argument.** X shall be of type real.

17 **Result Characteristics.** Same as X.

18 **Result Value.** The result has a value equal to a processor-dependent approximation to  $\tan(X)$ ,  
19 with X regarded as a value in radians.

20 **Example.** TAN (1.0) has the value 1.5574077 (approximately).

### 21 13.7.119 TANH (X)

22 **Description.** Hyperbolic tangent function.

23 **Class.** Elemental function.

24 **Argument.** X shall be of type real.

25 **Result Characteristics.** Same as X.

26 **Result Value.** The result has a value equal to a processor-dependent approximation to  $\tanh(X)$ .

1       **Example.** TANH (1.0) has the value 0.76159416 (approximately).

## 2   **13.7.120 TINY (X)**

3       **Description.** Returns the smallest positive number of the model representing numbers of the  
4 same type and kind type parameter as the argument.

5       **Class.** Inquiry function.

6       **Argument.** X shall be of type real. It may be a scalar or an array.

7       **Result Characteristics.** Scalar with the same type and kind type parameter as X.

8       **Result Value.** The result has the value  $b^{e_{\min}-1}$  where  $b$  and  $e_{\min}$  are as defined in 13.4 for the  
9 model representing numbers of the same type and kind type parameter as X.

10       **Example.** TINY (X) has the value  $2^{-127}$  for real X whose model is as in Note 13.4.

## 11   **13.7.121 TRANSFER (SOURCE, MOLD [, SIZE])**

12       **Description.** Returns a result with a physical representation identical to that of SOURCE but  
13 interpreted with the type and type parameters of MOLD.

14       **Class.** Transformational function.

15       **Arguments.**

16       SOURCE           may be of any type. It may be a scalar or an array.

17       MOLD             may be of any type. It may be a scalar or an array. If it is a variable, it need  
not be defined.

18       SIZE (optional) shall be scalar and of type integer. The corresponding actual argument shall  
not be an optional dummy argument.

19       **Result Characteristics.** The result is of the same type and type parameters as MOLD.

20       *Case (i):*       If MOLD is a scalar and SIZE is absent, the result is a scalar.

21       *Case (ii):*     If MOLD is an array and SIZE is absent, the result is an array and of rank one.  
22 Its size is as small as possible such that its physical representation is not shorter  
23 than that of SOURCE.

24       *Case (iii):*    If SIZE is present, the result is an array of rank one and size SIZE.

25       **Result Value.** If the physical representation of the result has the same length as that of  
26 SOURCE, the physical representation of the result is that of SOURCE. If the physical rep-  
27 resentation of the result is longer than that of SOURCE, the physical representation of the  
28 leading part is that of SOURCE and the remainder is processor dependent. If the physical  
29 representation of the result is shorter than that of SOURCE, the physical representation of the  
30 result is the leading part of SOURCE. If D and E are scalar variables such that the physical  
31 representation of D is as long as or longer than that of E, the value of TRANSFER (TRANS-  
32 FER (E, D), E) shall be the value of E. IF D is an array and E is an array of rank one, the  
33 value of TRANSFER (TRANSFER (E, D), E, SIZE (E)) shall be the value of E.

34       **Examples.**

35       *Case (i):*       TRANSFER (1082130432, 0.0) has the value 4.0 on a processor that represents  
36 the values 4.0 and 1082130432 as the string of binary digits 0100 0000 1000 0000  
37 0000 0000 0000 0000.

1        *Case (ii):*     TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /)) is a complex rank-one array of  
 2        length two whose first element has the value (1.1, 2.2) and whose second element  
 3        has a real part with the value 3.3. The imaginary part of the second element is  
 4        processor dependent.

5        *Case (iii):*    TRANSFER ((/ 1.1, 2.2, 3.3 /), (/ (0.0, 0.0) /), 1) is a complex rank-one array  
 6        of length one whose only element has the value (1.1, 2.2).

### 7    13.7.122    TRANSPOSE (MATRIX)

8        **Description.** Transpose an array of rank two.

9        **Class.** Transformational function.

10       **Argument.** MATRIX may be of any type and shall have rank two.

11       **Result Characteristics.** The result is an array of the same type and type parameters as  
 12       MATRIX and with rank two and shape  $(n, m)$  where  $(m, n)$  is the shape of MATRIX.

13       **Result Value.** Element  $(i, j)$  of the result has the value  $\text{MATRIX}(j, i)$ ,  $i = 1, 2, \dots, n$ ;  
 14        $j = 1, 2, \dots, m$ .

15       **Example.** If A is the array  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ , then TRANSPOSE (A) has the value  $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$ .

### 16   13.7.123    TRIM (STRING)

17       **Description.** Returns the argument with trailing blank characters removed.

18       **Class.** Transformational function.

19       **Argument.** STRING shall be of type character and shall be a scalar.

20       **Result Characteristics.** Character with the same kind type parameter value as STRING and  
 21       with a length that is the length of STRING less the number of trailing blanks in STRING.

22       **Result Value.** The value of the result is the same as STRING except any trailing blanks are  
 23       removed. If STRING contains no nonblank characters, the result has zero length.

24       **Example.** TRIM (' A B ') has the value ' A B '.

### 25   13.7.124    UBOUND (ARRAY [, DIM, KIND])

26       **Description.** Returns all the upper bounds of an array or a specified upper bound.

27       **Class.** Inquiry function.

28       **Arguments.**

29       ARRAY            may be of any type. It shall not be scalar. It shall not be an unallocated  
                              allocatable or a pointer that is not associated. If ARRAY is an assumed-size  
                              array, DIM shall be present with a value less than the rank of ARRAY.

30       DIM (optional)    shall be scalar and of type integer with a value in the range  $1 \leq \text{DIM} \leq n$ ,  
                              where  $n$  is the rank of ARRAY. The corresponding actual argument shall not  
                              be an optional dummy argument.

31       KIND (optional)   shall be a scalar integer initialization expression.

1       **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
 2 by the value of KIND; otherwise the kind type parameter is that of default integer type. It is  
 3 scalar if DIM is present; otherwise, the result is an array of rank one and size  $n$ , where  $n$  is the  
 4 rank of ARRAY.

5       **Result Value.**

6       *Case (i):*       For an array section or for an array expression, other than a whole array or array  
 7 structure component, UBOUND (ARRAY, DIM) has a value equal to the number  
 8 of elements in the given dimension; otherwise, it has a value equal to the upper  
 9 bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have  
 10 size zero and has the value zero if dimension DIM has size zero.

11       *Case (ii):*       UBOUND (ARRAY) has a value whose  $i$ th element is equal to UBOUND (AR-  
 12 RAY,  $i$ ), for  $i = 1, 2, \dots, n$ , where  $n$  is the rank of ARRAY.

13       **Examples.** If A is declared by the statement

14       REAL A (2:3, 7:10)

15       then UBOUND (A) is [3, 10] and UBOUND (A, DIM = 2) is 10.

### 16 13.7.125 UNPACK (VECTOR, MASK, FIELD)

17       **Description.** Unpack an array of rank one into an array under the control of a mask.

18       **Class.** Transformational function.

19       **Arguments.**

20       VECTOR           may be of any type. It shall have rank one. Its size shall be at least  $t$  where  
 21  $t$  is the number of true elements in MASK.

22       MASK             shall be an array of type logical.

23       FIELD            shall be of the same type and type parameters as VECTOR and shall be  
 24 conformable with MASK.

25       **Result Characteristics.** The result is an array of the same type and type parameters as  
 26 VECTOR and the same shape as MASK.

27       **Result Value.** The element of the result that corresponds to the  $i$ th true element of MASK,  
 28 in array element order, has the value VECTOR ( $i$ ) for  $i = 1, 2, \dots, t$ , where  $t$  is the number of  
 true values in MASK. Each other element has a value equal to FIELD if FIELD is scalar or to  
 the corresponding element of FIELD if it is an array.

29       **Examples.** Particular values may be “scattered” to particular positions in an array by us-

30       ing UNPACK. If M is the array  $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ , V is the array [1, 2, 3], and Q is the logical

31       mask  $\begin{bmatrix} \cdot & \text{T} & \cdot \\ \text{T} & \cdot & \cdot \\ \cdot & \cdot & \text{T} \end{bmatrix}$ , where “T” represents true and “.” represents false, then the result of

32       UNPACK (V, MASK = Q, FIELD = M) has the value  $\begin{bmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$  and the result of UN-

1           PACK (V, MASK = Q, FIELD = 0) has the value  $\begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ .

## 2   13.7.126   **VERIFY (STRING, SET [, BACK, KIND])**

3           **Description.** Verify that a set of characters contains all the characters in a string by identifying  
4           the position of the first character in a string of characters that does not appear in a given set of  
5           characters.

6           **Class.** Elemental function.

7           **Arguments.**

8           STRING           shall be of type character.

9           SET               shall be of type character with the same kind type parameter as STRING.

10          BACK (optional) shall be of type logical.

11          KIND (optional) shall be a scalar integer initialization expression.

12          **Result Characteristics.** Integer. If KIND is present, the kind type parameter is that specified  
13          by the value of KIND; otherwise the kind type parameter is that of default integer type.

14          **Result Value.**

15          *Case (i):*       If BACK is absent or has the value false and if STRING contains at least one  
16                            character that is not in SET, the value of the result is the position of the leftmost  
17                            character of STRING that is not in SET.

18          *Case (ii):*     If BACK is present with the value true and if STRING contains at least one  
19                            character that is not in SET, the value of the result is the position of the rightmost  
20                            character of STRING that is not in SET.

21          *Case (iii):*    The value of the result is zero if each character in STRING is in SET or if STRING  
22                            has zero length.

23          **Examples.**

24          *Case (i):*       VERIFY ('ABBA', 'A') has the value 2.

25          *Case (ii):*     VERIFY ('ABBA', 'A', BACK = .TRUE.) has the value 3.

26          *Case (iii):*    VERIFY ('ABBA', 'AB') has the value 0.

## 27   13.8   **Standard intrinsic modules**

28   This standard defines several intrinsic modules. A processor may extend the standard intrinsic modules  
29   to provide public entities in them in addition to those specified in this standard.

### NOTE 13.18

To avoid potential name conflicts with program entities, it is recommended that a program use the ONLY option in any USE statement that accesses a standard intrinsic module.

### 30   13.8.1   **The IEEE modules**

31   The IEEE\_EXCEPTIONS, IEEE\_ARITHMETIC, and IEEE\_FEATURES intrinsic modules are describ-  
32   ed in Section 14.

## 1 **13.8.2 The ISO\_FORTRAN\_ENV intrinsic module**

2 The intrinsic module ISO\_FORTRAN\_ENV provides public entities relating to the Fortran environment.

3 The processor shall provide the named constants described in the following subclauses.

### 4 **13.8.2.1 CHARACTER\_STORAGE\_SIZE**

5 The value of the default integer scalar constant CHARACTER\_STORAGE\_SIZE is the size expressed  
6 in bits of the character storage unit (16.4.3.1).

### 7 **13.8.2.2 ERROR\_UNIT**

8 The value of the default integer scalar constant ERROR\_UNIT identifies the processor-dependent pre-  
9 connected external unit used for the purpose of error reporting (9.4). This unit may be the same as  
10 OUTPUT\_UNIT. The value shall not be -1.

### 11 **13.8.2.3 FILE\_STORAGE\_SIZE**

12 The value of the default integer scalar constant FILE\_STORAGE\_SIZE is the size expressed in bits of  
13 the file storage unit (9.2.4).

### 14 **13.8.2.4 INPUT\_UNIT**

15 The value of the default integer scalar constant INPUT\_UNIT identifies the same processor-dependent  
16 preconnected external unit as the one identified by an asterisk in a READ statement (9.4). The value  
17 shall not be -1.

### 18 **13.8.2.5 IOSTAT\_END**

19 The value of the default integer scalar constant IOSTAT\_END is assigned to the variable specified in  
20 an IOSTAT= specifier (9.10.4) if an end-of-file condition occurs during execution of an input/output  
21 statement and no error condition occurs. This value shall be negative.

### 22 **13.8.2.6 IOSTAT\_EOR**

23 The value of the default integer scalar constant IOSTAT\_EOR is assigned to the variable specified in  
24 an IOSTAT= specifier (9.10.4) if an end-of-record condition occurs during execution of an input/output  
25 statement and no end-of-file or error condition occurs. This value shall be negative and different from  
26 the value of IOSTAT\_END.

### 27 **13.8.2.7 NUMERIC\_STORAGE\_SIZE**

28 The value of the default integer scalar constant NUMERIC\_STORAGE\_SIZE is the size expressed in  
29 bits of the numeric storage unit (16.4.3.1).

### 30 **13.8.2.8 OUTPUT\_UNIT**

31 The value of the default integer scalar constant OUTPUT\_UNIT identifies the same processor-dependent  
32 preconnected external unit as the one identified by an asterisk in a WRITE statement (9.4). The value  
33 shall not be -1.

1 **13.8.3 The ISO\_C\_BINDING module**

2 The ISO\_C\_BINDING intrinsic module is described in Section 15.



## 1 Section 14: Exceptions and IEEE arithmetic

2 The intrinsic modules IEEE\_EXCEPTIONS, IEEE\_ARITHMETIC, and IEEE\_FEATURES provide sup-  
3 port for exceptions and IEEE arithmetic. Whether the modules are provided is processor dependent.  
4 If the module IEEE\_FEATURES is provided, which of the named constants defined in this standard  
5 are included is processor dependent. The module IEEE\_ARITHMETIC behaves as if it contained a  
6 USE statement for IEEE\_EXCEPTIONS; everything that is public in IEEE\_EXCEPTIONS is public in  
7 IEEE\_ARITHMETIC.

### NOTE 14.1

The types and procedures defined in these modules are not themselves intrinsic.

8 If IEEE\_EXCEPTIONS or IEEE\_ARITHMETIC is accessible in a scoping unit, IEEE\_OVERFLOW  
9 and IEEE\_DIVIDE\_BY\_ZERO are supported in the scoping unit for all kinds of real and complex  
10 data. Which other exceptions are supported can be determined by the function IEEE\_SUPPORT\_  
11 FLAG (14.10.26); whether control of halting is supported can be determined by the function IEEE\_SUP-  
12 PORT\_HALTING. The extent of support of the other exceptions may be influenced by the accessibility  
13 of the named constants IEEE\_INEXACT\_FLAG, IEEE\_INVALID\_FLAG, and IEEE\_UNDERFLOW\_  
14 FLAG of the module IEEE\_FEATURES. If a scoping unit has access to IEEE\_UNDERFLOW\_FLAG  
15 of IEEE\_FEATURES, within the scoping unit the processor shall support underflow and return true  
16 from IEEE\_SUPPORT\_FLAG( IEEE\_UNDERFLOW, X) for at least one kind of real. Similarly, if  
17 IEEE\_INEXACT\_FLAG or IEEE\_INVALID\_FLAG is accessible, within the scoping unit the processor  
18 shall support the exception and return true from the corresponding inquiry for at least one kind of real.  
19 Also, if IEEE\_HALTING is accessible, within the scoping unit the processor shall support control of  
20 halting and return true from IEEE\_SUPPORT\_HALTING(FLAGS) for the flag.

### NOTE 14.2

IEEE\_INVALID is not required to be supported whenever IEEE\_EXCEPTIONS is accessed. This is to allow a non-IEEE processor to provide support for overflow and divide\_by\_zero. On an IEEE machine, invalid is an equally serious condition.

### NOTE 14.3

The IEEE\_FEATURES module is provided to allow a reasonable amount of cooperation between the programmer and the processor in controlling the extent of IEEE arithmetic support. On some processors some IEEE features are natural for the processor to support, others may be inefficient at run time, and others are essentially impossible to support. If IEEE\_FEATURES is not used, the processor will support only the natural operations. Within IEEE\_FEATURES the processor will define the named constants (14.1) corresponding to the time-consuming features (as well as the natural ones for completeness) but will not define named constants corresponding to the impossible features. If the programmer accesses IEEE\_FEATURES, the processor shall provide support for all of the IEEE\_FEATURES that are reasonably possible. If the programmer uses an ONLY clause on a USE statement to access a particular feature name, the processor shall provide support for the corresponding feature, or issue an error message saying the name is not defined in the module.

When used this way, the named constants in the IEEE\_FEATURES are similar to what are frequently called command line switches for the compiler. They can specify compilation options in a reasonably portable manner.

1 If a scoping unit does not access IEEE\_FEATURES, IEEE\_EXCEPTIONS, or IEEE\_ARITHMETIC,  
2 the level of support is processor dependent, and need not include support for any exceptions. If a flag is  
3 signaling on entry to such a scoping unit, the processor ensures that it is signaling on exit. If a flag is  
4 quiet on entry to such a scoping unit, whether it is signaling on exit is processor dependent.

5 Further IEEE support is available through the module IEEE\_ARITHMETIC. The extent of support  
6 may be influenced by the accessibility of the named constants of the module IEEE\_FEATURES. If a  
7 scoping unit has access to IEEE\_DATATYPE of IEEE\_FEATURES, within the scoping unit the pro-  
8 cessor shall support IEEE arithmetic and return true from IEEE\_SUPPORT\_DATATYPE(X) (14.10.23)  
9 for at least one kind of real. Similarly, if IEEE\_DENORMAL, IEEE\_DIVIDE, IEEE\_INF, IEEE\_NAN,  
10 IEEE\_ROUNDING, or IEEE\_SQRT is accessible, within the scoping unit the processor shall support the  
11 feature and return true from the corresponding inquiry function for at least one kind of real. In the case of  
12 IEEE\_ROUNDING, it shall return true for all the rounding modes IEEE\_NEAREST, IEEE\_TO\_ZERO,  
13 IEEE\_UP, and IEEE\_DOWN.

14 Execution might be slowed on some processors by the support of some features. If IEEE\_EXCEPTIONS  
15 or IEEE\_ARITHMETIC is accessed but IEEE\_FEATURES is not accessed, the supported subset of fea-  
16 tures is processor dependent. The processor's fullest support is provided when all of IEEE\_FEATURES  
17 is accessed as in

```
18 USE, INTRINSIC :: IEEE_ARITHMETIC; USE, INTRINSIC :: IEEE_FEATURES
```

19 but execution might then be slowed by the presence of a feature that is not needed. In all cases, the  
20 extent of support can be determined by the inquiry functions.

## 21 14.1 Derived types and constants defined in the modules

22 The modules IEEE\_EXCEPTIONS, IEEE\_ARITHMETIC, and IEEE\_FEATURES define five derived  
23 types, whose components are all private.

24 The module IEEE\_EXCEPTIONS defines

- 25 • IEEE\_FLAG\_TYPE, for identifying a particular exception flag. Its only possible values are those  
26 of named constants defined in the module: IEEE\_INVALID, IEEE\_OVERFLOW, IEEE\_DIVIDE\_-  
27 BY\_ZERO, IEEE\_UNDERFLOW, and IEEE\_INEXACT. The module also defines the array named  
28 constants IEEE\_USUAL = (/ IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO, IEEE\_INVALID /)  
29 and IEEE\_ALL = (/ IEEE\_USUAL, IEEE\_UNDERFLOW, IEEE\_INEXACT /).
- 30 • IEEE\_STATUS\_TYPE, for saving the current floating point status.

31 The module IEEE\_ARITHMETIC defines

- 32 • IEEE\_CLASS\_TYPE, for identifying a class of floating-point values. Its only possible values  
33 are those of named constants defined in the module: IEEE\_SIGNALING\_NAN, IEEE\_QUI-  
34 ET\_NAN, IEEE\_NEGATIVE\_INF, IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_DENORM-  
35 AL, IEEE\_NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO, IEEE\_POSITIVE\_DENORMAL, IEEE\_-  
36 POSITIVE\_NORMAL, IEEE\_POSITIVE\_INF, and IEEE\_OTHER\_VALUE.
- 37 • IEEE\_ROUND\_TYPE, for identifying a particular rounding mode. Its only possible values are  
38 those of named constants defined in the module: IEEE\_NEAREST, IEEE\_TO\_ZERO, IEEE\_UP,  
39 and IEEE\_DOWN for the IEEE modes; and IEEE\_OTHER for any other mode.

- 1     • The elemental operator `==` for two values of one of these types to return true if the values are the  
2     same and false otherwise.
- 3     • The elemental operator `/=` for two values of one of these types to return true if the values differ  
4     and false otherwise.

5 The module `IEEE_FEATURES` defines

- 6     • `IEEE_FEATURES_TYPE`, for expressing the need for particular IEEE features. Its only possible  
7     values are those of named constants defined in the module: `IEEE_DATATYPE`, `IEEE_DENOR-`  
8     `MAL`, `IEEE_DIVIDE`, `IEEE_HALTING`, `IEEE_INEXACT_FLAG`, `IEEE_INF`, `IEEE_INVALID_-`  
9     `FLAG`, `IEEE_NAN`, `IEEE_ROUNDING`, `IEEE_SQRT`, and `IEEE_UNDERFLOW_FLAG`.

## 10 **14.2 The exceptions**

11 The exceptions are

- 12     • `IEEE_OVERFLOW`

13         This exception occurs when the result for an intrinsic real operation or assignment has an absolute  
14         value greater than a processor-dependent limit, or the real or imaginary part of the result for an  
15         intrinsic complex operation or assignment has an absolute value greater than a processor-dependent  
16         limit.

- 17     • `IEEE_DIVIDE_BY_ZERO`

18         This exception occurs when a real or complex division has a nonzero numerator and a zero denom-  
19         inator.

- 20     • `IEEE_INVALID`

21         This exception occurs when a real or complex operation or assignment is invalid; possible examples  
22         are `SQRT(X)` when `X` is real and has a nonzero negative value, and conversion to an integer (by  
23         assignment, an intrinsic procedure, or a procedure defined in an intrinsic module) when the result  
24         is too large to be representable.

- 25     • `IEEE_UNDERFLOW`

26         This exception occurs when the result for an intrinsic real operation or assignment has an absolute  
27         value less than a processor-dependent limit and loss of accuracy is detected, or the real or imaginary  
28         part of the result for an intrinsic complex operation or assignment has an absolute value less than  
29         a processor-dependent limit and loss of accuracy is detected.

- 30     • `IEEE_INEXACT`

31         This exception occurs when the result of a real or complex operation or assignment is not exact.

32 Each exception has a flag whose value is either quiet or signaling. The value can be determined by  
33 the function `IEEE_GET_FLAG`. Its initial value is quiet and it signals when the associated excep-  
34 tion occurs. Its status can also be changed by the subroutine `IEEE_SET_FLAG` or the subroutine  
35 `IEEE_SET_STATUS`. Once signaling within a procedure, it remains signaling unless set quiet by an  
36 invocation of the subroutine `IEEE_SET_FLAG` or the subroutine `IEEE_SET_STATUS`.

37 If a flag is signaling on entry to a procedure, the processor will set it to quiet on entry and restore it to  
38 signaling on return.

**NOTE 14.4**

If a flag signals during execution of a procedure, the processor shall not set it to quiet on return.

- 1 Evaluation of a specification expression may cause an exception to signal.
- 2 In a scoping unit that has access to IEEE\_EXCEPTIONS or IEEE\_ARITHMETIC, if an intrinsic  
 3 procedure or a procedure defined in an intrinsic module executes normally, the values of the flags  
 4 IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO, and IEEE\_INVALID shall be as on entry to the proce-  
 5 dure, even if one or more signals during the calculation. If a real or complex result is too large for the  
 6 procedure to handle, IEEE\_OVERFLOW may signal. If a real or complex result is a NaN because of an  
 7 invalid operation (for example, LOG(-1.0)), IEEE\_INVALID may signal. Similar rules apply to format  
 8 processing and to intrinsic operations: no signaling flag shall be set quiet and no quiet flag shall be set  
 9 signaling because of an intermediate calculation that does not affect the result.

**NOTE 14.5**

An implementation may provide alternative versions of an intrinsic procedure; a practical example of such alternatives might be one version suitable for a call from a scoping unit with access to IEEE\_EXCEPTIONS or IEEE\_ARITHMETIC and one for other cases.

- 10 In a sequence of statements that has no invocations of IEEE\_GET\_FLAG, IEEE\_SET\_FLAG, IEEE\_-  
 11 GET\_STATUS, IEEE\_SET\_HALTING, or IEEE\_SET\_STATUS, if the execution of an operation would  
 12 cause an exception to signal but after execution of the sequence no value of a variable depends on the  
 13 operation, whether the exception is signaling is processor dependent. For example, when Y has the value  
 14 zero, whether the code

15       X = 1.0/Y  
 16       X = 3.0

- 17 signals IEEE\_DIVIDE\_BY\_ZERO is processor dependent. Another example is the following:

18       REAL, PARAMETER :: X=0.0, Y=6.0  
 19       IF (1.0/X == Y) PRINT \*, 'Hello world'

- 20 where the processor is permitted to discard the IF statement because the logical expression can never  
 21 be true and no value of a variable depends on it.

- 22 An exception shall not signal if this could arise only during execution of an operation beyond those  
 23 required or permitted by the standard. For example, the statement

24       IF (F(X)>0.0) Y = 1.0/Z

- 25 shall not signal IEEE\_DIVIDE\_BY\_ZERO when both F(X) and Z are zero and the statement

26       WHERE(A>0.0) A = 1.0/A

- 27 shall not signal IEEE\_DIVIDE\_BY\_ZERO. On the other hand, when X has the value 1.0 and Y has the  
 28 value 0.0, the expression

1 X>0.00001 .OR. X/Y>0.00001

2 is permitted to cause the signaling of IEEE\_DIVIDE\_BY\_ZERO.

3 The processor need not support IEEE\_INVALID, IEEE\_UNDERFLOW, and IEEE\_INEXACT. If an  
4 exception is not supported, its flag is always quiet. The function IEEE\_SUPPORT\_FLAG can be used  
5 to inquire whether a particular flag is supported.

### 6 **14.3 The rounding modes**

7 The IEEE standard specifies four rounding modes:

- 8 • IEEE\_NEAREST rounds the exact result to the nearest representable value.
- 9 • IEEE\_TO\_ZERO rounds the exact result towards zero to the next representable value.
- 10 • IEEE\_UP rounds the exact result towards +infinity to the next representable value.
- 11 • IEEE\_DOWN rounds the exact result towards -infinity to the next representable value.

12 The function IEEE\_GET\_ROUNDING\_MODE can be used to inquire which rounding mode is in opera-  
13 tion. Its value is one of the above four or IEEE\_OTHER if the rounding mode does not conform to the  
14 IEEE standard.

15 If the processor supports the alteration of the rounding mode during execution, the subroutine IEEE\_  
16 SET\_ROUNDING\_MODE can be used to alter it. The function IEEE\_SUPPORT\_ROUNDING can be  
17 used to inquire whether this facility is available for a particular mode. The function IEEE\_SUPPORT\_IO  
18 can be used to inquire whether rounding for base conversion in formatted input/output (9.4.5.13, 9.5.1.12,  
19 10.6.1.2.6) is as specified in the IEEE standard.

20 In a procedure other than IEEE\_SET\_ROUNDING\_MODE or IEEE\_SET\_STATUS, the processor shall  
21 not change the rounding mode on entry, and on return shall ensure that the rounding mode is the same  
22 as it was on entry.

#### **NOTE 14.6**

Within a program, all literal constants that have the same form have the same value (4.1.2). Therefore, the value of a literal constant is not affected by the rounding mode.
--

### 23 **14.4 Underflow mode**

24 Some processors allow control during program execution of whether underflow produces a denormalized  
25 number in conformance with the IEEE standard (gradual underflow) or produces zero instead (abrupt  
26 underflow). On some processors, floating-point performance is typically better in abrupt underflow mode  
27 than in gradual underflow mode.

28 Control over the underflow mode is exercised by invocation of IEEE\_SET\_UNDERFLOW\_MODE. The  
29 function IEEE\_GET\_UNDERFLOW\_MODE can be used to inquire which underflow mode is in op-  
30 eration. The function IEEE\_SUPPORT\_UNDERFLOW\_MODE can be used to inquire whether this  
31 facility is available. The initial underflow mode is processor dependent. In a procedure other than  
32 IEEE\_SET\_UNDERFLOW\_MODE or IEEE\_SET\_STATUS, the processor shall not change the under-  
33 flow mode on entry, and on return shall ensure that the underflow mode is the same as it was on entry.

34 The underflow mode affects only floating-point calculations whose type is that of an X for which  
35 IEEE\_SUPPORT\_UNDERFLOW\_CONTROL returns true.”

## 1 14.5 Halting

2 Some processors allow control during program execution of whether to abort or continue execution after  
3 an exception. Such control is exercised by invocation of the subroutine IEEE.SET\_HALTING\_MODE.  
4 Halting is not precise and may occur any time after the exception has occurred. The function IEEE.SUP-  
5 PORT\_HALTING can be used to inquire whether this facility is available. The initial halting mode is  
6 processor dependent. In a procedure other than IEEE.SET\_HALTING\_MODE or IEEE.SET\_STATUS,  
7 the processor shall not change the halting mode on entry, and on return shall ensure that the halting  
8 mode is the same as it was on entry.

## 9 14.6 The floating point status

10 The values of all the supported flags for exceptions, rounding mode, underflow mode, and halting are  
11 called the floating point status. The floating point status can be saved in a scalar variable of type  
12 TYPE(IEEE.STATUS\_TYPE) with the subroutine IEEE.GET\_STATUS and restored with the subrou-  
13 tine IEEE.SET\_STATUS. There are no facilities for finding the values of particular flags held within such  
14 a variable. Portions of the floating point status can be saved with the subroutines IEEE.GET\_FLAG,  
15 IEEE.GET\_HALTING\_MODE, and IEEE.GET\_ROUNDING\_MODE, and set with the subroutines  
16 IEEE.SET\_FLAG, IEEE.SET\_HALTING\_MODE, and IEEE.SET\_ROUNDING\_MODE.

### NOTE 14.7

Some processors hold all these flags in a floating point status register that can be saved and restored as a whole much faster than all individual flags can be saved and restored. These procedures are provided to exploit this feature.

### NOTE 14.8

The processor is required to ensure that a call to a Fortran procedure does not change the floating point status other than by setting exception flags to signaling.

## 17 14.7 Exceptional values

18 The IEEE standard specifies the following exceptional floating point values:

- 19 • Denormalized values have very small absolute values and lowered precision.
- 20 • Infinite values (+infinity and -infinity) are created by overflow or division by zero.
- 21 • Not-a-Number ( NaN) values are undefined values or values created by an invalid operation.

22 In this standard, the term **normal** is used for values that are not in one of these exceptional classes.

23 The functions IEEE.IS\_FINITE, IEEE.IS\_NAN, IEEE.IS\_NEGATIVE, and IEEE.IS\_NORMAL are pro-  
24 vided to test whether a value is finite, NaN, negative, or normal. The function IEEE.VALUE is pro-  
25 vided to generate an IEEE number of any class, including an infinity or a NaN. The functions IEEE.-  
26 SUPPORT\_DENORMAL, IEEE.SUPPORT\_INF, and IEEE.SUPPORT\_NAN are provided to determine  
27 whether these facilities are available for a particular kind of real.

## 28 14.8 IEEE arithmetic

29 The function IEEE.SUPPORT\_DATATYPE can be used to inquire whether IEEE arithmetic is sup-  
30 ported for a particular kind of real. Complete conformance with the IEEE standard is not required,

1 but the normalized numbers shall be exactly those of an IEEE floating-point format; the operations of  
 2 addition, subtraction, and multiplication shall be implemented with at least one of the IEEE rounding  
 3 modes; the IEEE operation `rem` shall be provided by the function `IEEE_REM`; and the IEEE functions  
 4 `copysign`, `scalb`, `logb`, `nextafter`, and `unordered` shall be provided by the functions `IEEE_COPY_SIGN`,  
 5 `IEEE_SCALB`, `IEEE_LOGB`, `IEEE_NEXT_AFTER`, and `IEEE_UNORDERED`, respectively. The in-  
 6 quiry function `IEEE_SUPPORT_DIVIDE` is provided to inquire whether the processor supports divide  
 7 with the accuracy specified by the IEEE standard. For each of the operations of addition, subtraction,  
 8 and multiplication, the result shall be as specified in the IEEE standard whenever the IEEE result is  
 9 normalized and the operands are normalized (if floating point) or are valid and within range (if another  
 10 type).

11 The inquiry function `IEEE_SUPPORT_NAN` is provided to inquire whether the processor supports  
 12 IEEE NaNs. Where these are supported, their behavior for unary and binary operations, including  
 13 those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the  
 14 specifications in the IEEE standard.

15 The inquiry function `IEEE_SUPPORT_INF` is provided to inquire whether the processor supports IEEE  
 16 infinities. Where these are supported, their behavior for unary and binary operations, including those  
 17 defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifi-  
 18 cations in the IEEE standard.

19 The inquiry function `IEEE_SUPPORT_INF` is provided to inquire whether the processor supports IEEE  
 20 infinities. Where these are supported, their behavior for unary and binary operations, including those  
 21 defined by intrinsic functions and by functions in intrinsic modules, shall be consistent with the specifi-  
 22 cations in the IEEE standard.

23 The inquiry function `IEEE_SUPPORT_DENORMAL` is provided to inquire whether the processor sup-  
 24 ports IEEE denormals. Where these are supported, their behavior for unary and binary operations,  
 25 including those defined by intrinsic functions and by functions in intrinsic modules, shall be consistent  
 26 with the specifications in the IEEE standard.

27 The IEEE standard specifies a square root function that returns `-0.0` for the square root of `-0.0` and has  
 28 certain accuracy requirements. The function `IEEE_SUPPORT_SQRT` can be used to inquire whether  
 29 `SQRT` is implemented in accord with the IEEE standard for a particular kind of real.

30 The inquiry function `IEEE_SUPPORT_STANDARD` is provided to inquire whether the processor sup-  
 31 ports all the IEEE facilities defined in this standard for a particular kind of real.

## 32 14.9 Tables of the procedures

33 For all of the procedures defined in the modules, the arguments shown are the names that shall be used  
 34 for argument keywords if the keyword form is used for the actual arguments.

35 The procedure classification terms “inquiry function” and “transformational function” are used here  
 36 with the same meanings as in [13.1](#).

### 37 14.9.1 Inquiry functions

38 The module `IEEE_EXCEPTIONS` contains the following inquiry functions:

39	<code>IEEE_SUPPORT_FLAG</code> ( <code>FLAG</code> [, <code>X</code> ])	Are IEEE exceptions supported?
40	<code>IEEE_SUPPORT_HALTING</code> ( <code>FLAG</code> )	Is IEEE halting control supported?

41 The module `IEEE_ARITHMETIC` contains the following inquiry functions:

1	IEEE_SUPPORT_DATATYPE ([X])	Is IEEE arithmetic supported?
2	IEEE_SUPPORT_DENORMAL ([X])	Are IEEE denormalized numbers supported?
3	IEEE_SUPPORT_DIVIDE ([X])	Is IEEE divide supported?
4	IEEE_SUPPORT_INF ([X])	Is IEEE infinity supported?
5	IEEE_SUPPORT_IO ([X])	Is IEEE formatting supported?
6	IEEE_SUPPORT_NAN ([X])	Are IEEE NaNs supported?
	IEEE_SUPPORT_ROUNDING	Is IEEE rounding supported?
7	(ROUND_VALUE [, X])	
8	IEEE_SUPPORT_SQRT ([X])	Is IEEE square root supported?
9	IEEE_SUPPORT_STANDARD ([X])	Are all IEEE facilities supported?
10	IEEE_SUPPORT_UNDERFLOW_CONTROL ([X])	Is IEEE underflow control supported?

### 11 14.9.2 Elemental functions

12 The module IEEE\_ARITHMETIC contains the following elemental functions for reals X and Y for which  
 13 IEEE\_SUPPORT\_DATATYPE(X) and IEEE\_SUPPORT\_DATATYPE(Y) are true:

14	IEEE_CLASS (X)	IEEE class.
15	IEEE_COPY_SIGN (X,Y)	IEEE copysign function.
16	IEEE_IS_FINITE (X)	Determine if value is finite.
17	IEEE_IS_NAN (X)	Determine if value is IEEE Not-a-Number.
18	IEEE_IS_NORMAL (X)	Determine if a value is normal, that is, neither an infinity, a NaN, nor denormalized.
19		
20	IEEE_IS_NEGATIVE (X)	Determine if value is negative.
21	IEEE_LOGB (X)	Unbiased exponent in the IEEE floating point format.
22		
23	IEEE_NEXT_AFTER (X,Y)	Returns the next representable neighbor of X in the direction toward Y.
24		
25	IEEE_REM (X,Y)	The IEEE REM function, that is $X - Y*N$ , where N is the integer nearest to the exact value $X/Y$ .
26		
27		
28	IEEE_RINT (X)	Round to an integer value according to the current rounding mode.
29		
30	IEEE_SCALB (X,I)	Returns $X \times 2^I$ .
31	IEEE_UNORDERED (X,Y)	IEEE unordered function. True if X or Y is a NaN and false otherwise.
32		
33	IEEE_VALUE (X,CLASS)	Generate an IEEE value.

### 34 14.9.3 Kind function

35 The module IEEE\_ARITHMETIC contains the following transformational function:

36	IEEE_SELECTED_REAL_KIND ([P, R])	Kind type parameter value for an IEEE real with given precision and range.
37		

### 38 14.9.4 Elemental subroutines

39 The module IEEE\_EXCEPTIONS contains the following elemental subroutines:

40	IEEE_GET_FLAG (FLAG,FLAG_VALUE)	Get an exception flag.
41	IEEE_GET_HALTING_MODE (FLAG, HALTING)	Get halting mode for an exception.

## 1 14.9.5 Nonelemental subroutines

2 The module IEEE\_EXCEPTIONS contains the following nonelemental subroutines:

3	IEEE_GET_STATUS (STATUS_VALUE)	Get the current state of the floating point environment.
4		
5	IEEE_SET_FLAG (FLAG,FLAG_VALUE)	Set an exception flag.
6	IEEE_SET_HALTING_MODE (FLAG, HALTING)	Controls continuation or halting on exceptions.
7	IEEE_SET_STATUS (STATUS_VALUE)	Restore the state of the floating point environment.
8		

9 The module IEEE\_ARITHMETIC contains the following nonelemental subroutines:

10	IEEE_GET_ROUNDING_MODE (ROUND_VALUE)	Get the current IEEE rounding mode.
11	IEEE_GET_UNDERFLOW_MODE (GRADUAL)	Get the current underflow mode.
12	IEEE_SET_ROUNDING_MODE (ROUND_VALUE)	Set the current IEEE rounding mode.
13	IEEE_SET_UNDERFLOW_MODE (GRADUAL)	Set the current underflow mode.

## 14 14.10 Specifications of the procedures

15 In the detailed descriptions below, procedure names are generic and are not specific. All the functions  
16 are pure. The dummy arguments of the intrinsic module procedures in 14.9.1, 14.9.2, and 14.9.3 have  
17 INTENT(IN). The dummy arguments of the intrinsic module procedures in 14.9.4 and 14.9.5 have  
18 INTENT(IN) if the intent is not stated explicitly. In the examples, it is assumed that the processor  
19 supports IEEE arithmetic for default real.

### NOTE 14.9

It is intended that a processor should not check a condition given in a paragraph labeled “**Restriction**” at compile time, but rather should rely on the programmer writing code such as

```

IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  C = IEEE_CLASS(X)
ELSE
  .
  .
ENDIF

```

to avoid a call being made on a processor for which the condition is violated.

20 For the elemental functions of IEEE\_ARITHMETIC, as tabulated in 14.9.2, if X or Y has a value that  
21 is an infinity or a NaN, the result shall be consistent with the general rules in 6.1 and 6.2 of the IEEE  
22 standard. For example, the result for an infinity shall be constructed as the limiting case of the result  
23 with a value of arbitrarily large magnitude, if such a limit exists.

### 24 14.10.1 IEEE\_CLASS (X)

25 **Description.** IEEE class function.

1       **Class.** Elemental function.

2       **Argument.** X shall be of type real.

3       **Restriction.** IEEE\_CLASS(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) has  
4 the value false.

5       **Result Characteristics.** TYPE(IEEE\_CLASS\_TYPE).

6       **Result Value.** The result value shall be IEEE\_SIGNALING\_NAN or IEEE\_QUIET\_NAN  
7 if IEEE\_SUPPORT\_NAN(X) has the value true and the value of X is a signaling or quiet  
8 NaN, respectively. The result value shall be IEEE\_NEGATIVE\_INF or IEEE\_POSITIVE\_INF  
9 if IEEE\_SUPPORT\_INF(X) has the value true and the value of X is negative or positive  
10 infinity, respectively. The result value shall be IEEE\_NEGATIVE\_DENORMAL or IEEE-  
11 \_POSITIVE\_DENORMAL if IEEE\_SUPPORT\_DENORMAL(X) has the value true and the  
12 value of X is a negative or positive denormalized value, respectively. The result value shall be  
13 IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO, or IEEE-  
14 \_POSITIVE\_NORMAL if value of X is negative normal, negative zero, positive zero, or positive  
15 normal, respectively. Otherwise, the result value shall be IEEE\_OTHER\_VALUE.

16       **Example.** IEEE\_CLASS(-1.0) has the value IEEE\_NEGATIVE\_NORMAL.

#### NOTE 14.10

The result value IEEE\_OTHER\_VALUE is needed for implementing the module on systems which are basically IEEE, but do not implement all of it. It might be needed, for example, if an unformatted file were written in a program executing with gradual underflow enabled and read with it disabled.

### 17 14.10.2 IEEE\_COPY\_SIGN (X, Y)

18       **Description.** IEEE copysign function.

19       **Class.** Elemental function.

20       **Arguments.** The arguments shall be of type real.

21       **Restriction.** IEEE\_COPY\_SIGN(X,Y) shall not be invoked if IEEE\_SUPPORT\_DATA-  
22 TYPE(X) or IEEE\_SUPPORT\_DATATYPE(Y) has the value false.

23       **Result Characteristics.** Same as X.

24       **Result Value.** The result has the value of X with the sign of Y. This is true even for IEEE  
25 special values, such as a NaN or an infinity (on processors supporting such values).

26       **Example.** The value of IEEE\_COPY\_SIGN(X,1.0) is ABS(X) even when X is NaN.

### 27 14.10.3 IEEE\_GET\_FLAG (FLAG, FLAG\_VALUE)

28       **Description.** Get an exception flag.

29       **Class.** Elemental subroutine.

30       **Arguments.**

31       FLAG           shall be of type TYPE(IEEE\_FLAG\_TYPE). It specifies the IEEE flag to be  
obtained.

FLAG\_VALUE shall be of type default logical. It is an INTENT(OUT) argument. If the value of FLAG is IEEE\_INVALID, IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO, IEEE\_UNDERFLOW, or IEEE\_INEXACT, the result value is true if the corresponding exception flag is signaling and is false otherwise.

**Example.** Following CALL IEEE\_GET\_FLAG(IEEE\_OVERFLOW,FLAG\_VALUE), FLAG\_VALUE is true if the IEEE\_OVERFLOW flag is signaling and is false if it is quiet.

#### 14.10.4 IEEE\_GET\_HALTING\_MODE (FLAG, HALTING)

**Description.** Get halting mode for an exception.

**Class.** Elemental subroutine.

**Arguments.**

FLAG shall be of type TYPE(IEEE\_FLAG\_TYPE). It specifies the IEEE flag. It shall have one of the values IEEE\_INVALID, IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO, IEEE\_UNDERFLOW, or IEEE\_INEXACT.

HALTING shall be of type default logical. It is of INTENT(OUT). The value is true if the exception specified by FLAG will cause halting. Otherwise, the value is false.

**Example.** To store the halting mode for IEEE\_OVERFLOW, do a calculation without halting, and restore the halting mode later:

```

USE, INTRINSIC :: IEEE_ARITHMETIC
LOGICAL HALTING
...
CALL IEEE_GET_HALTING_MODE(IEEE_OVERFLOW,HALTING) ! Store halting mode
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,.FALSE.) ! No halting
...! calculation without halting
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,HALTING) ! Restore halting mode

```

#### 14.10.5 IEEE\_GET\_ROUNDING\_MODE (ROUND\_VALUE)

**Description.** Get the current IEEE rounding mode.

**Class.** Subroutine.

**Argument.** ROUND\_VALUE shall be scalar of type TYPE(IEEE\_ROUND\_TYPE). It is an INTENT(OUT) argument and returns the floating point rounding mode, with value IEEE\_NEAREST, IEEE\_TO\_ZERO, IEEE\_UP, or IEEE\_DOWN if one of the IEEE modes is in operation and IEEE\_OTHER otherwise.

**Example.** To store the rounding mode, do a calculation with round to nearest, and restore the rounding mode later:

```

USE, INTRINSIC :: IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
...
CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode

```

```

1      CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
2      ... ! calculation with round to nearest
3      CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE) ! Restore the rounding mode

```

#### 4 14.10.6 IEEE\_GET\_STATUS (STATUS\_VALUE)

5 **Description.** Get the current value of the floating point status (14.6).

6 **Class.** Subroutine.

7 **Argument.** STATUS\_VALUE shall be scalar of type TYPE(IEEE\_STATUS\_TYPE). It is an  
8 INTENT(OUT) argument and returns the floating point status.

9 **Example.** To store all the exception flags, do a calculation involving exception handling, and  
10 restore them later:

```

11      USE, INTRINSIC :: IEEE_ARITHMETIC
12      TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
13      ...
14      CALL IEEE_GET_STATUS(STATUS_VALUE) ! Get the flags
15      CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set the flags quiet.
16      ... ! calculation involving exception handling
17      CALL IEEE_SET_STATUS(STATUS_VALUE) ! Restore the flags

```

#### 18 14.10.7 IEEE\_GET\_UNDERFLOW\_MODE (GRADUAL)

19 **Description.** Get the current underflow mode (14.4).

20 **Class.** Subroutine.

21 **Argument.** GRADUAL shall be of type default logical. It is an INTENT(OUT) argument.  
22 The value is true if the current underflow mode is gradual underflow, and false if the current  
23 underflow mode is abrupt underflow.

24 **Restriction.** IEEE\_GET\_UNDERFLOW\_MODE shall not be invoked unless IEEE\_SUPPORT-  
25 \_UNDERFLOW\_CONTROL(X) is true for some X.

26 **Example.** After CALL IEEE\_SET\_UNDERFLOW\_MODE(.FALSE.), a subsequent CALL  
27 IEEE\_GET\_UNDERFLOW\_MODE(GRADUAL) will set GRADUAL to false.

#### 28 14.10.8 IEEE\_IS\_FINITE (X)

29 **Description.** Determine if a value is finite.

30 **Class.** Elemental function.

31 **Argument.** X shall be of type real.

32 **Restriction.** IEEE\_IS\_FINITE(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X)  
33 has the value false.

34 **Result Characteristics.** Default logical.

1       **Result Value.** The result has the value true if the value of X is finite, that is, IEEE\_CLASS(X)  
2       has one of the values IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_DENORMAL, IEEE\_  
3       NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO, IEEE\_POSITIVE\_DENORMAL, or IEEE\_POS-  
4       ITIVE\_NORMAL; otherwise, the result has the value false.

5       **Example.** IEEE\_IS\_FINITE(1.0) has the value true.

#### 6   **14.10.9   IEEE\_IS\_NAN (X)**

7       **Description.** Determine if a value is IEEE Not-a-Number.

8       **Class.** Elemental function.

9       **Argument.** X shall be of type real.

10       **Restriction.** IEEE\_IS\_NAN(X) shall not be invoked if IEEE\_SUPPORT\_NAN(X) has the value  
11       false.

12       **Result Characteristics.** Default logical.

13       **Result Value.** The result has the value true if the value of X is an IEEE NaN; otherwise, it  
14       has the value false.

15       **Example.** IEEE\_IS\_NAN(SQRT(-1.0)) has the value true if IEEE\_SUPPORT\_SQRT(1.0) has  
16       the value true.

#### 17   **14.10.10 IEEE\_IS\_NEGATIVE (X)**

18       **Description.** Determine if a value is negative.

19       **Class.** Elemental function.

20       **Argument.** X shall be of type real.

21       **Restriction.** IEEE\_IS\_NEGATIVE(X) shall not be invoked if IEEE\_SUPPORT\_DATA-  
22       TYPE(X) has the value false.

23       **Result Characteristics.** Default logical.

24       **Result Value.** The result has the value true if IEEE\_CLASS(X) has one of the values  
25       IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_DENORMAL, IEEE\_NEGATIVE\_ZERO or  
26       IEEE\_NEGATIVE\_INF; otherwise, the result has the value false.

27       **Example.** IEEE\_IS\_NEGATIVE(0.0) has the value false.

#### 28   **14.10.11 IEEE\_IS\_NORMAL (X)**

29       **Description.** Determine if a value is normal, that is, neither an infinity, a NaN, nor denormal-  
30       ized.

31       **Class.** Elemental function.

32       **Argument.** X shall be of type real.

33       **Restriction.** IEEE\_IS\_NORMAL(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X)  
34       has the value false.

35       **Result Characteristics.** Default logical.

1       **Result Value.** The result has the value true if IEEE\_CLASS(X) has one of the values  
2       IEEE\_NEGATIVE\_NORMAL, IEEE\_NEGATIVE\_ZERO, IEEE\_POSITIVE\_ZERO or IEEE\_  
3       POSITIVE\_NORMAL; otherwise, the result has the value false.

4       **Example.** IEEE\_IS\_NORMAL(SQRT(-1.0)) has the value false if IEEE\_SUPPORT\_SQRT(1.0)  
5       has the value true.

#### 6   **14.10.12 IEEE\_LOGB (X)**

7       **Description.** Unbiased exponent in the IEEE floating point format.

8       **Class.** Elemental function.

9       **Argument.** X shall be of type real.

10       **Restriction.** IEEE\_LOGB(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) has  
11       the value false.

12       **Result Characteristics.** Same as X.

13       **Result Value.**

14       *Case (i):* If the value of X is neither zero, infinity, nor NaN, the result has the value of the  
15       unbiased exponent of X. Note: this value is equal to EXPONENT(X)-1.

16       *Case (ii):* If X==0, the result is -infinity if IEEE\_SUPPORT\_INF(X) is true and -HUGE(X)  
17       otherwise; IEEE\_DIVIDE\_BY\_ZERO signals.

18       **Example.** IEEE\_LOGB(-1.1) has the value 0.0.

#### 19   **14.10.13 IEEE\_NEXT\_AFTER (X, Y)**

20       **Description.** Returns the next representable neighbor of X in the direction toward Y.

21       **Class.** Elemental function.

22       **Arguments.** The arguments shall be of type real.

23       **Restriction.** IEEE\_NEXT\_AFTER(X,Y) shall not be invoked if IEEE\_SUPPORT\_DATA-  
24       TYPE(X) or IEEE\_SUPPORT\_DATATYPE(Y) has the value false.

25       **Result Characteristics.** Same as X.

26       **Result Value.**

27       *Case (i):* If X == Y, the result is X and no exception is signaled.

28       *Case (ii):* If X /= Y, the result has the value of the next representable neighbor of X  
29       in the direction of Y. The neighbors of zero (of either sign) are both nonzero.  
30       IEEE\_OVERFLOW is signaled when X is finite but IEEE\_NEXT\_AFTER(X,Y)  
31       is infinite; IEEE\_UNDERFLOW is signaled when IEEE\_NEXT\_AFTER(X,Y) is  
32       denormalized; in both cases, IEEE\_INEXACT signals.

33       **Example.** The value of IEEE\_NEXT\_AFTER(1.0,2.0) is 1.0+EPSILON(X).

#### 34   **14.10.14 IEEE\_REM (X, Y)**

35       **Description.** IEEE REM function.

36       **Class.** Elemental function.

1       **Arguments.** The arguments shall be of type real.

2       **Restriction.** IEEE\_REM(X,Y) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) or  
3       IEEE\_SUPPORT\_DATATYPE(Y) has the value false.

4       **Result Characteristics.** Real with the kind type parameter of whichever argument has the  
5       greater precision.

6       **Result Value.** The result value, regardless of the rounding mode, shall be exactly  $X - Y*N$ ,  
7       where N is the integer nearest to the exact value  $X/Y$ ; whenever  $|N - X/Y| = 1/2$ , N shall be  
8       even. If the result value is zero, the sign shall be that of X.

9       **Examples.** The value of IEEE\_REM(4.0,3.0) is 1.0, the value of IEEE\_REM(3.0,2.0) is -1.0,  
10       and the value of IEEE\_REM(5.0,2.0) is 1.0.

### 11   **14.10.15 IEEE\_RINT (X)**

12       **Description.** Round to an integer value according to the current rounding mode.

13       **Class.** Elemental function.

14       **Argument.** X shall be of type real.

15       **Restriction.** IEEE\_RINT(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) has the  
16       value false.

17       **Result Characteristics.** Same as X.

18       **Result Value.** The value of the result is the value of X rounded to an integer according to the  
19       current rounding mode. If the result has the value zero, the sign is that of X.

20       **Examples.** If the current rounding mode is round to nearest, the value of IEEE\_RINT(1.1) is  
21       1.0. If the current rounding mode is round up, the value of IEEE\_RINT(1.1) is 2.0.

### 22   **14.10.16 IEEE\_SCALB (X, I)**

23       **Description.** Returns  $X \times 2^I$ .

24       **Class.** Elemental function.

25       **Arguments.**

26       X                   shall be of type real.

27       I                   shall be of type integer.

28       **Restriction.** IEEE\_SCALB(X) shall not be invoked if IEEE\_SUPPORT\_DATATYPE(X) has  
29       the value false.

30       **Result Characteristics.** Same as X.

31       **Result Value.**

32       *Case (i):*       If  $X \times 2^I$  is representable as a normal number, the result has this value.

33       *Case (ii):*      If X is finite and  $X \times 2^I$  is too large, the IEEE\_OVERFLOW exception shall  
34       occur. If IEEE\_SUPPORT\_INF(X) is true, the result value is infinity with the  
35       sign of X; otherwise, the result value is SIGN(HUGE(X),X).

36       *Case (iii):*     If  $X \times 2^I$  is too small and there is loss of accuracy, the IEEE\_UNDERFLOW  
37       exception shall occur. The result is the representable number having a magnitude

1                               nearest to  $|2^I|$  and the same sign as X.

2       *Case (iv):*     If X is infinite, the result is the same as X; no exception signals.

3       **Example.** The value of IEEE\_SCALB(1.0,2) is 4.0.

#### 4   **14.10.17 IEEE\_SELECTED\_REAL\_KIND ([P, R])**

5       **Description.** Returns a value of the kind type parameter of an IEEE real data type with  
6       decimal precision of at least P digits and a decimal exponent range of at least R. For data  
7       objects of such a type, IEEE\_SUPPORT\_DATATYPE(X) has the value true.

8       **Class.** Transformational function.

9       **Arguments.** At least one argument shall be present.

10      P (optional)       shall be scalar and of type integer.

11      R (optional)       shall be scalar and of type integer.

12      **Result Characteristics.** Default integer scalar.

13      **Result Value.** The result has a value equal to a value of the kind type parameter of an IEEE  
14      real type with decimal precision, as returned by the function PRECISION, of at least P digits  
15      and a decimal exponent range, as returned by the function RANGE, of at least R, or if no such  
16      kind type parameter is available on the processor, the result is -1 if the precision is not available,  
17      -2 if the exponent range is not available, and -3 if neither is available. If more than one kind  
18      type parameter value meets the criteria, the value returned is the one with the smallest decimal  
19      precision, unless there are several such values, in which case the smallest of these kind values is  
20      returned.

21      **Example.** IEEE\_SELECTED\_REAL\_KIND(6,30) has the value KIND(0.0) on a machine that  
22      supports IEEE single precision arithmetic for its default real approximation method.

#### 23   **14.10.18 IEEE\_SET\_FLAG (FLAG, FLAG\_VALUE)**

24      **Description.** Assign a value to an exception flag.

25      **Class.** Subroutine.

26      **Arguments.**

27      FLAG               shall be a scalar or array of type TYPE(IEEE\_FLAG\_TYPE). If a value of  
                              FLAG is IEEE\_INVALID, IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO,  
                              IEEE\_UNDERFLOW, or IEEE\_INEXACT, the corresponding exception flag  
                              is assigned a value. No two elements of FLAG shall have the same value.

28      FLAG\_VALUE        shall be a scalar or array of type default logical. It shall be conformable with  
                              FLAG. If an element has the value true, the corresponding flag is set to be  
                              signaling; otherwise, the flag is set to be quiet.

29      **Example.** CALL IEEE\_SET\_FLAG(IEEE\_OVERFLOW, .TRUE.) sets the IEEE\_OVERFLOW  
30      flag to be signaling.

#### 31   **14.10.19 IEEE\_SET\_HALTING\_MODE (FLAG, HALTING)**

32      **Description.** Controls continuation or halting after an exception.

33      **Class.** Subroutine.

1     **Arguments.**

2     **FLAG**           shall be a scalar or array of type TYPE(IEEE\_FLAG\_TYPE). It shall have only the values IEEE\_INVALID, IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO, IEEE\_UNDERFLOW, or IEEE\_INEXACT. No two elements of FLAG shall have the same value.

3     **HALTING**       shall be a scalar or array of type default logical. It shall be conformable with FLAG. If an element has the value is true, the corresponding exception specified by FLAG will cause halting. Otherwise, execution will continue after this exception.

4     **Restriction.** IEEE\_SET\_HALTING\_MODE(FLAG,HALTING) shall not be invoked if IEEE\_SUPPORT\_HALTING(FLAG) has the value false.

6     **Example.**    CALL IEEE\_SET\_HALTING\_MODE(IEEE\_DIVIDE\_BY\_ZERO,.TRUE.) causes halting after a divide\_by\_zero exception.

## NOTE 14.11

The initial halting mode is processor dependent. Halting is not precise and may occur some time after the exception has occurred.

8     **14.10.20 IEEE\_SET\_ROUNDING\_MODE (ROUND\_VALUE)**

9     **Description.** Set the current IEEE rounding mode.

10    **Class.** Subroutine.

11    **Argument.** ROUND\_VALUE shall be scalar and of type TYPE(IEEE\_ROUND\_TYPE). It specifies the mode to be set.

13    **Restriction.** IEEE\_SET\_ROUNDING\_MODE(ROUND\_VALUE) shall not be invoked unless IEEE\_SUPPORT\_ROUNDING(ROUND\_VALUE,X) is true for some X such that IEEE\_SUPPORT\_DATATYPE(X) is true.

16    **Example.** To store the rounding mode, do a calculation with round to nearest, and restore the rounding mode later:

```
18        USE, INTRINSIC :: IEEE_ARITHMETIC
19        TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
20        ...
21        CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
22        CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
23        : ! calculation with round to nearest
24        CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE) ! Restore the rounding mode
```

25    **14.10.21 IEEE\_SET\_STATUS (STATUS\_VALUE)**

26    **Description.** Restore the value of the floating point status (14.6).

27    **Class.** Subroutine.

28    **Argument.** STATUS\_VALUE shall be scalar and of type TYPE(IEEE\_STATUS\_TYPE). Its value shall have been set in a previous invocation of IEEE\_GET\_STATUS.

1       **Example.** To store all the exceptions flags, do a calculation involving exception handling, and  
2       restore them later:

```
3           USE, INTRINSIC :: IEEE_EXCEPTIONS
4           TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
5           ...
6           CALL IEEE_GET_STATUS(STATUS_VALUE) ! Store the flags
7           CALL IEEE_SET_FLAGS(IEEE_ALL,.FALSE.) ! Set them quiet
8           ... ! calculation involving exception handling
9           CALL IEEE_SET_STATUS(STATUS_VALUE) ! Restore the flags
```

#### NOTE 14.12

On some processors this may be a very time consuming process.
---

### 10 14.10.22 IEEE\_SET\_UNDERFLOW\_MODE (GRADUAL)

11       **Description.** Set the current underflow mode.

12       **Class.** Subroutine.

13       **Argument.** GRADUAL shall be of type default logical. If it is true, the current underflow  
14       mode is set to gradual underflow. If it is false, the current underflow mode is set to abrupt  
15       underflow.

16       **Restriction.** IEEE\_SET\_UNDERFLOW\_MODE shall not be invoked unless IEEE\_SUPPORT-  
17       \_UNDERFLOW\_CONTROL(X) is true for some X.

18       **Example.** To perform some calculations with abrupt underflow and then restore the previous  
19       mode:

```
20           USE,INTRINSIC :: IEEE_ARITHMETIC
21           LOGICAL SAVE_UNDERFLOW_MODE
22           ...
23           CALL IEEE_GET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)
24           CALL IEEE_SET_UNDERFLOW_MODE(GRADUAL=.FALSE.)
25           ... ! Perform some calculations with abrupt underflow
26           CALL IEEE_SET_UNDERFLOW_MODE(SAVE_UNDERFLOW_MODE)"
```

### 27 14.10.23 IEEE\_SUPPORT\_DATATYPE () or IEEE\_SUPPORT\_DATATYPE (X)

28       **Description.** Inquire whether the processor supports IEEE arithmetic.

29       **Class.** Inquiry function.

30       **Argument.** X shall be of type real. It may be a scalar or an array.

31       **Result Characteristics.** Default logical scalar.

32       **Result Value.** The result has the value true if the processor supports IEEE arithmetic for all  
33       reals (X absent) or for real variables of the same kind type parameter as X; otherwise, it has  
34       the value false. Here, support is as defined in the first paragraph of [14.8](#).

1       **Example.** If default reals are implemented as in the IEEE standard except that underflow values  
2 flush to zero instead of being denormal, IEEE\_SUPPORT\_DATATYPE(1.0) has the value true.

### 3 **14.10.24 IEEE\_SUPPORT\_DENORMAL () or IEEE\_SUPPORT\_DENORMAL (X)**

4       **Description.** Inquire whether the processor supports IEEE denormalized numbers.

5       **Class.** Inquiry function.

6       **Argument.** X shall be of type real. It may be a scalar or an array.

7       **Result Characteristics.** Default logical scalar.

8       **Result Value.**

9       *Case (i):*       IEEE\_SUPPORT\_DENORMAL(X) has the value true if IEEE\_SUPPORT\_  
10                    DATATYPE(X) has the value true and the processor supports arithmetic op-  
11                    erations and assignments with denormalized numbers (biased exponent  $e = 0$   
12                    and fraction  $f \neq 0$ , see section 3.2 of the IEEE standard) for real variables of the  
13                    same kind type parameter as X; otherwise, it has the value false.

14       *Case (ii):*     IEEE\_SUPPORT\_DENORMAL() has the value true if and only if IEEE\_SUP-  
15                    PORT\_DENORMAL(X) has the value true for all real X.

16       **Example.** IEEE\_SUPPORT\_DENORMAL(X) has the value true if the processor supports  
17 denormalized numbers for X.

#### **NOTE 14.13**

The denormalized numbers are not included in the 13.4 model for real numbers; they satisfy the inequality  $ABS(X) < TINY(X)$ . They usually occur as a result of an arithmetic operation whose exact result is less than  $TINY(X)$ . Such an operation causes IEEE\_UNDERFLOW to signal unless the result is exact. IEEE\_SUPPORT\_DENORMAL(X) is false if the processor never returns a denormalized number as the result of an arithmetic operation.

### 18 **14.10.25 IEEE\_SUPPORT\_DIVIDE () or IEEE\_SUPPORT\_DIVIDE (X)**

19       **Description.** Inquire whether the processor supports divide with the accuracy specified by the  
20 IEEE standard.

21       **Class.** Inquiry function.

22       **Argument.** X shall be of type real. It may be a scalar or an array.

23       **Result Characteristics.** Default logical scalar.

24       **Result Value.**

25       *Case (i):*       IEEE\_SUPPORT\_DIVIDE(X) has the value true if the processor supports divide  
26                    with the accuracy specified by the IEEE standard for real variables of the same  
27                    kind type parameter as X; otherwise, it has the value false.

28       *Case (ii):*     IEEE\_SUPPORT\_DIVIDE() has the value true if and only if IEEE\_SUPPORT\_  
29                    DIVIDE(X) has the value true for all real X.

30       **Example.** IEEE\_SUPPORT\_DIVIDE(X) has the value true if the processor supports IEEE  
31 divide for X.

### 32 **14.10.26 IEEE\_SUPPORT\_FLAG (FLAG) or IEEE\_SUPPORT\_FLAG (FLAG, X)**

1       **Description.** Inquire whether the processor supports an exception.

2       **Class.** Inquiry function.

3       **Arguments.**

4       FLAG           shall be scalar and of type TYPE(IEEE\_FLAG\_TYPE). Its value shall  
                      be one of IEEE\_INVALID, IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO,  
                      IEEE\_UNDERFLOW, or IEEE\_INEXACT.

5       X              shall be of type real. It may be a scalar or an array.

6       **Result Characteristics.** Default logical scalar.

7       **Result Value.**

8       *Case (i):*       IEEE\_SUPPORT\_FLAG(FLAG, X) has the value true if the processor supports  
9                       detection of the specified exception for real variables of the same kind type pa-  
10                      rameter as X; otherwise, it has the value false.

11       *Case (ii):*    IEEE\_SUPPORT\_FLAG(FLAG) has the value true if and only if IEEE\_SUP-  
12                      PORT\_FLAG(FLAG, X) has the value true for all real X.

13       **Example.** IEEE\_SUPPORT\_FLAG(IEEE\_INEXACT) has the value true if the processor sup-  
14       ports the inexact exception.

#### 15   **14.10.27 IEEE\_SUPPORT\_HALTING (FLAG)**

16       **Description.** Inquire whether the processor supports the ability to control during program  
17       execution whether to abort or continue execution after an exception.

18       **Class.** Inquiry function.

19       **Argument.** FLAG shall be scalar and of type TYPE(IEEE\_FLAG\_TYPE). Its value shall be  
20       one of IEEE\_INVALID, IEEE\_OVERFLOW, IEEE\_DIVIDE\_BY\_ZERO, IEEE\_UNDERFLOW,  
21       or IEEE\_INEXACT.

22       **Result Characteristics.** Default logical scalar.

23       **Result Value.** The result has the value true if the processor supports the ability to control  
24       during program execution whether to abort or continue execution after the exception specified  
25       by FLAG; otherwise, it has the value false. Support includes the ability to change the mode by  
26       CALL IEEE\_SET\_HALTING(FLAG).

27       **Example.** IEEE\_SUPPORT\_HALTING(IEEE\_OVERFLOW) has the value true if the proces-  
28       sor supports control of halting after an overflow.

#### 29   **14.10.28 IEEE\_SUPPORT\_INF () or IEEE\_SUPPORT\_INF (X)**

30       **Description.** Inquire whether the processor supports the IEEE infinity facility.

31       **Class.** Inquiry function.

32       **Argument.** X shall be of type real. It may be a scalar or an array.

33       **Result Characteristics.** Default logical scalar.

34       **Result Value.**

35       *Case (i):*       IEEE\_SUPPORT\_INF(X) has the value true if the processor supports IEEE in-

1 finities (positive and negative) for real variables of the same kind type parameter  
2 as X; otherwise, it has the value false.

3 *Case (ii):* IEEE\_SUPPORT\_INF() has the value true if and only if IEEE\_SUPPORT\_-  
4 INF(X) has the value true for all real X.

5 **Example.** IEEE\_SUPPORT\_INF(X) has the value true if the processor supports IEEE infinities  
6 for X.

### 7 **14.10.29 IEEE\_SUPPORT\_IO () or IEEE\_SUPPORT\_IO (X)**

8 **Description.** Inquire whether the processor supports IEEE base conversion rounding during  
9 formatted input/output (9.4.5.13, 9.5.1.12, 10.6.1.2.6).

10 **Class.** Inquiry function.

11 **Argument.** X shall be of type real. It may be a scalar or an array.

12 **Result Characteristics.** Default logical scalar.

13 **Result Value.**

14 *Case (i):* IEEE\_SUPPORT\_IO(X) has the value true if the processor supports IEEE base  
15 conversion during formatted input/output (9.4.5.13, 9.5.1.12, 10.6.1.2.6) as de-  
16 scribed in the IEEE standard for the modes UP, DOWN, ZERO, and NEAREST  
17 for real variables of the same kind type parameter as X; otherwise, it has the  
18 value false.

19 *Case (ii):* IEEE\_SUPPORT\_IO() has the value true if and only if IEEE\_SUPPORT\_IO(X)  
20 has the value true for all real X.

21 **Example.** IEEE\_SUPPORT\_IO(X) has the value true if the processor supports IEEE base  
22 conversion for X.

### 23 **14.10.30 IEEE\_SUPPORT\_NAN () or IEEE\_SUPPORT\_NAN (X)**

24 **Description.** Inquire whether the processor supports the IEEE Not-a-Number facility.

25 **Class.** Inquiry function.

26 **Argument.** X shall be of type real. It may be a scalar or an array.

27 **Result Characteristics.** Default logical scalar.

28 **Result Value.**

29 *Case (i):* IEEE\_SUPPORT\_NAN(X) has the value true if the processor supports IEEE  
30 NaNs for real variables of the same kind type parameter as X; otherwise, it has  
31 the value false.

32 *Case (ii):* IEEE\_SUPPORT\_NAN() has the value true if and only if IEEE\_SUPPORT\_-  
33 NAN(X) has the value true for all real X.

34 **Example.** IEEE\_SUPPORT\_NAN(X) has the value true if the processor supports IEEE NaNs  
35 for X.

### 36 **14.10.31 IEEE\_SUPPORT\_ROUNDING (ROUND\_VALUE) or IEEE\_SUPPORT\_ROUNDING (ROUND\_VALUE, X)**

37 **Description.** Inquire whether the processor supports a particular IEEE rounding mode.

1       **Class.** Inquiry function.

2       **Arguments.**

3       ROUND\_VALUE shall be of type TYPE(IEEE\_ROUND\_TYPE).

4       X                   shall be of type real. It may be a scalar or an array.

5       **Result Characteristics.** Default logical scalar.

6       **Result Value.**

7       *Case (i):*       IEEE\_SUPPORT\_ROUNDING(ROUND\_VALUE, X) has the value true if the  
8                   processor supports the rounding mode defined by ROUND\_VALUE for real vari-  
9                   ables of the same kind type parameter as X; otherwise, it has the value false. Sup-  
10                  port includes the ability to change the mode by CALL IEEE\_SET\_ROUNDING-  
11                  MODE(ROUND\_VALUE).

12       *Case (ii):*     IEEE\_SUPPORT\_ROUNDING(ROUND\_VALUE) has the value true if and only  
13                   if IEEE\_SUPPORT\_ROUNDING(ROUND\_VALUE, X) has the value true for all  
14                   real X.

15       **Example.** IEEE\_SUPPORT\_ROUNDING(IEEE\_TO\_ZERO) has the value true if the processor  
16       supports rounding to zero for all reals.

### 17 **14.10.32 IEEE\_SUPPORT\_SQRT () or IEEE\_SUPPORT\_SQRT (X)**

18       **Description.** Inquire whether the processor implements SQRT in accord with the IEEE stan-  
19       dard.

20       **Class.** Inquiry function.

21       **Argument.** X shall be of type real. It may be a scalar or an array.

22       **Result Characteristics.** Default logical scalar.

23       **Result Value.**

24       *Case (i):*       IEEE\_SUPPORT\_SQRT(X) has the value true if the processor implements SQRT  
25                   in accord with the IEEE standard for real variables of the same kind type param-  
26                   eter as X; otherwise, it has the value false.

27       *Case (ii):*     IEEE\_SUPPORT\_SQRT() has the value true if and only if IEEE\_SUPPORT\_-  
28                   SQRT(X) has the value true for all real X.

29       **Example.** IEEE\_SUPPORT\_SQRT(X) has the value true if the processor implements SQRT(X)  
30       in accord with the IEEE standard. In this case, SQRT(-0.0) has the value -0.0.

### 31 **14.10.33 IEEE\_SUPPORT\_STANDARD () or IEEE\_SUPPORT\_STANDARD (X)**

32       **Description.** Inquire whether the processor supports all the IEEE facilities defined in this  
33       standard.

34       **Class.** Inquiry function.

35       **Argument.** X shall be of type real. It may be a scalar or an array.

36       **Result Characteristics.** Default logical scalar.

37       **Result Value.**

- 1        *Case (i):*        IEEE\_SUPPORT\_STANDARD(X) has the value true if the results of all the func-  
 2                            tions IEEE\_SUPPORT\_DATATYPE(X), IEEE\_SUPPORT\_DENORMAL(X),  
 3                            IEEE\_SUPPORT\_DIVIDE(X), IEEE\_SUPPORT\_FLAG(FLAG,X) for valid  
 4                            FLAG, IEEE\_SUPPORT\_HALTING(FLAG) for valid FLAG, IEEE\_SUP-  
 5                            PORT\_INF(X), IEEE\_SUPPORT\_NAN(X), IEEE\_SUPPORT\_ROUNDING  
 6                            (ROUND\_VALUE,X) for valid ROUND\_VALUE, and IEEE\_SUPPORT\_SQRT  
 7                            (X) are all true; otherwise, the result has the value false.
- 8        *Case (ii):*        IEEE\_SUPPORT\_STANDARD() has the value true if and only if IEEE\_SUP-  
 9                            PORT\_STANDARD(X) has the value true for all real X.

10        **Example.** IEEE\_SUPPORT\_STANDARD() has the value false if the processor supports both  
 11        IEEE and non-IEEE kinds of reals.

### 12 **14.10.34 IEEE\_SUPPORT\_UNDERFLOW\_CONTROL() or** 13 **IEEE\_SUPPORT\_UNDERFLOW\_CONTROL(X)**

14        **Description.** Inquire whether the procedure supports the ability to control the underflow mode  
 15        during program execution.

16        **Class.** Inquiry function.

17        **Argument.** X shall be of type real. It may be a scalar or an array.

18        **Result Characteristics.** Default logical scalar.

19        **Result Value.**

20        *Case (i):*        IEEE\_SUPPORT\_UNDERFLOW\_CONTROL(X) has the value true if the proces-  
 21                            sor supports control of the underflow mode for floating-point calculations with  
 22                            the same type as X, and false otherwise.

23        *Case (ii):*        IEEE\_SUPPORT\_UNDERFLOW\_CONTROL() has the value true if the proces-  
 24                            sor supports control of the underflow mode for all floating-point calculations, and  
 25                            false otherwise.

26        **Example.** IEEE\_SUPPORT\_UNDERFLOW\_CONTROL(2.5) has the value true if the proces-  
 27        sor supports underflow mode control for calculations of type default real.

### 28 **14.10.35 IEEE\_UNORDERED (X, Y)**

29        **Description.** IEEE unordered function. True if X or Y is a NaN, and false otherwise.

30        **Class.** Elemental function.

31        **Arguments.** The arguments shall be of type real.

32        **Restriction.** IEEE\_UNORDERED(X,Y) shall not be invoked if IEEE\_SUPPORT\_DATA-  
 33        TYPE(X) or IEEE\_SUPPORT\_DATATYPE(Y) has the value false.

34        **Result Characteristics.** Default logical.

35        **Result Value.** The result has the value true if X or Y is a NaN or both are NaNs; otherwise,  
 36        it has the value false.

37        **Example.** IEEE\_UNORDERED(0.0,SQRT(-1.0)) has the value true if IEEE\_SUPPORT\_-  
 38        SQRT(1.0) has the value true.

### 39 **14.10.36 IEEE\_VALUE (X, CLASS)**

1       **Description.** Generate an IEEE value.

2       **Class.** Elemental function.

3       **Arguments.**

4       X               shall be of type real.

5       CLASS           shall be of type TYPE(IEEE\_CLASS\_TYPE). The value is permitted to  
       be: IEEE\_SIGNALING\_NAN or IEEE\_QUIET\_NAN if IEEE\_SUPPORT\_-  
       NAN(X) has the value true, IEEE\_NEGATIVE\_INF or IEEE\_POSITIVE\_-  
       INF if IEEE\_SUPPORT\_INF(X) has the value true, IEEE\_NEGATIVE\_-  
       DENORMAL or IEEE\_POSITIVE\_DENORMAL if IEEE\_SUPPORT\_DE-  
       NORMAL(X) has the value true, IEEE\_NEGATIVE\_NORMAL, IEEE\_NEG-  
       ATIVE\_ZERO, IEEE\_POSITIVE\_ZERO or IEEE\_POSITIVE\_NORMAL.

6       **Restriction.** IEEE\_VALUE(X,CLASS) shall not be invoked if IEEE\_SUPPORT\_DATA-  
       TYPE(X) has the value false.

8       **Result Characteristics.** Same as X.

9       **Result Value.** The result value is an IEEE value as specified by CLASS. Although in most  
 10      cases the value is processor dependent, the value shall not vary between invocations for any  
 11      particular X kind type parameter and CLASS value.

12      **Example.** IEEE\_VALUE(1.0,IEEE\_NEGATIVE\_INF) has the value -infinity.

## 13 14.11 Examples

### NOTE 14.14

```

MODULE DOT
! Module for dot product of two real arrays of rank 1.
! The caller needs to ensure that exceptions do not cause halting.
  USE, INTRINSIC :: IEEE_EXCEPTIONS
  LOGICAL :: MATRIX_ERROR = .FALSE.
  INTERFACE OPERATOR(.dot.)
    MODULE PROCEDURE MULT
  END INTERFACE
CONTAINS
  REAL FUNCTION MULT(A,B)
    REAL, INTENT(IN) :: A(:),B(:)
    INTEGER I
    LOGICAL OVERFLOW
    IF (SIZE(A)/=SIZE(B)) THEN
      MATRIX_ERROR = .TRUE.
      RETURN
    END IF
! The processor ensures that IEEE_OVERFLOW is quiet
    MULT = 0.0
    DO I = 1, SIZE(A)

```

**NOTE 14.14 (cont.)**

```

        MULT = MULT + A(I)*B(I)
    END DO
    CALL IEEE_GET_FLAG(IEEE_OVERFLOW,OVERFLOW)
    IF (OVERFLOW) MATRIX_ERROR = .TRUE.
END FUNCTION MULT
END MODULE DOT

```

This module provides the dot product of two real arrays of rank 1. If the sizes of the arrays are different, an immediate return occurs with MATRIX\_ERROR true. If overflow occurs during the actual calculation, the IEEE\_OVERFLOW flag will signal and MATRIX\_ERROR will be true.

**NOTE 14.15**

```

USE, INTRINSIC :: IEEE_EXCEPTIONS
USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_INVALID_FLAG
! The other exceptions of IEEE_USUAL (IEEE_OVERFLOW and
! IEEE_DIVIDE_BY_ZERO) are always available with IEEE_EXCEPTIONS
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
LOGICAL, DIMENSION(3) :: FLAG_VALUE
...
CALL IEEE_GET_STATUS(STATUS_VALUE)
CALL IEEE_SET_HALTING_MODE(IEEE_USUAL,.FALSE.) ! Needed in case the
!           default on the processor is to halt on exceptions
CALL IEEE_SET_FLAG(IEEE_USUAL,.FALSE.)
! First try the "fast" algorithm for inverting a matrix:
MATRIX1 = FAST_INV(MATRIX) ! This shall not alter MATRIX.
CALL IEEE_GET_FLAG(IEEE_USUAL,FLAG_VALUE)
IF (ANY(FLAG_VALUE)) THEN
! "Fast" algorithm failed; try "slow" one:
    CALL IEEE_SET_FLAG(IEEE_USUAL,.FALSE.)
    MATRIX1 = SLOW_INV(MATRIX)
    CALL IEEE_GET_FLAG(IEEE_USUAL,FLAG_VALUE)
    IF (ANY(FLAG_VALUE)) THEN
        WRITE (*, *) 'Cannot invert matrix'
        STOP
    END IF
END IF
CALL IEEE_SET_STATUS(STATUS_VALUE)

```

In this example, the function FAST\_INV may cause a condition to signal. If it does, another try is made with SLOW\_INV. If this still fails, a message is printed and the program stops. Note, also, that it is important to set the flags quiet before the second try. The state of all the flags is stored and restored.

## NOTE 14.16

```

USE, INTRINSIC :: IEEE_EXCEPTIONS
LOGICAL FLAG_VALUE
...
CALL IEEE_SET_HALTING_MODE(IEEE_OVERFLOW,.FALSE.)
! First try a fast algorithm for inverting a matrix.
CALL IEEE_SET_FLAG(IEEE_OVERFLOW,.FALSE.)
DO K = 1, N
...
    CALL IEEE_GET_FLAG(IEEE_OVERFLOW,FLAG_VALUE)
    IF (FLAG_VALUE) EXIT
END DO
IF (FLAG_VALUE) THEN
! Alternative code which knows that K-1 steps have executed normally.
...
END IF

```

Here the code for matrix inversion is in line and the transfer is made more precise by adding extra tests of the flag.

## NOTE 14.17

```

REAL FUNCTION HYPOT(X, Y)
! In rare circumstances this may lead to the signaling of IEEE_OVERFLOW
! The caller needs to ensure that exceptions do not cause halting.
USE, INTRINSIC :: IEEE_ARITHMETIC
USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_UNDERFLOW_FLAG
! IEEE_OVERFLOW is always available with IEEE_ARITHMETIC
REAL X, Y
REAL SCALED_X, SCALED_Y, SCALED_RESULT
LOGICAL, DIMENSION(2) :: FLAGS
TYPE(IEEE_FLAG_TYPE), PARAMETER, DIMENSION(2) :: &
    OUT_OF_RANGE = (/ IEEE_OVERFLOW, IEEE_UNDERFLOW /)
INTRINSIC SQRT, ABS, EXPONENT, MAX, DIGITS, SCALE
! The processor clears the flags on entry
! Try a fast algorithm first
HYPOT = SQRT( X**2 + Y**2 )
CALL IEEE_GET_FLAG(OUT_OF_RANGE,FLAGS)
IF ( ANY(FLAGS) ) THEN
    CALL IEEE_SET_FLAG(OUT_OF_RANGE,.FALSE.)
    IF ( X==0.0 .OR. Y==0.0 ) THEN
        HYPOT = ABS(X) + ABS(Y)
    ELSE IF ( 2*ABS(EXPONENT(X)-EXPONENT(Y)) > DIGITS(X)+1 ) THEN
        HYPOT = MAX( ABS(X), ABS(Y) )! one of X and Y can be ignored
    
```

## NOTE 14.17 (cont.)

```
ELSE ! scale so that ABS(X) is near 1
  SCALED_X = SCALE( X, -EXPONENT(X) )
  SCALED_Y = SCALE( Y, -EXPONENT(X) )
  SCALED_RESULT = SQRT( SCALED_X**2 + SCALED_Y**2 )
  HYPOT = SCALE( SCALED_RESULT, EXPONENT(X) ) ! may cause overflow
END IF
END IF
! The processor resets any flag that was signaling on entry
END FUNCTION HYPOT
```

An attempt is made to evaluate this function directly in the fastest possible way. This will work almost every time, but if an exception occurs during this fast computation, a safe but slower way evaluates the function. This slower evaluation might involve scaling and unscaling, and in (very rare) extreme cases this unscaling can cause overflow (after all, the true result might overflow if X and Y are both near the overflow limit). If the IEEE\_OVERFLOW or IEEE\_UNDERFLOW flag is signaling on entry, it is reset on return by the processor, so that earlier exceptions are not lost.



## 1 Section 15: Interoperability with C

2 Fortran provides a means of referencing procedures that are defined by means of the C programming  
3 language or procedures that can be described by C prototypes as defined in 6.7.5.3 of the C standard,  
4 even if they are not actually defined by means of C. Conversely, there is a means of specifying that a  
5 procedure defined by a Fortran subprogram can be referenced from a function defined by means of C.  
6 In addition, there is a means of declaring global variables that are linked with C variables that have  
7 external linkage as defined in 6.2.2 of the C standard.

8 The ISO\_C\_BINDING module provides access to named constants that represent kind type parameters  
9 of data representations compatible with C types. Fortran also provides facilities for defining derived  
10 types (4.5) and enumerations (4.6) that correspond to C types.

### 11 15.1 The ISO\_C\_BINDING intrinsic module

12 The processor shall provide the intrinsic module ISO\_C\_BINDING. This module shall make accessible  
13 the following entities: named constants with the names listed in the second column of Table 15.2 and  
14 the first column of Table 15.1, the procedures specified in 15.1.2, C\_PTR, C\_FUNPTR, C\_NULL\_PTR,  
15 and C\_NULL\_FUNPTR. A processor may provide other public entities in the ISO\_C\_BINDING intrinsic  
16 module in addition to those listed here.

#### NOTE 15.1

To avoid potential name conflicts with program entities, it is recommended that a program use the ONLY option in any USE statement that accesses the ISO_C_BINDING intrinsic module.
--

#### 17 15.1.1 Named constants and derived types in the module

18 The entities listed in the second column of Table 15.2, shall be named constants of type default integer.

19 The value of C\_INT shall be a valid value for an integer kind parameter on the processor. The values  
20 of C\_SHORT, C\_LONG, C\_LONG\_LONG, C\_SIGNED\_CHAR, C\_SIZE\_T, C\_INT8\_T, C\_INT16\_T, C\_  
21 INT32\_T, C\_INT64\_T, C\_INT\_LEAST8\_T, C\_INT\_LEAST16\_T, C\_INT\_LEAST32\_T, C\_INT\_LEAST64\_  
22 \_T, C\_INT\_FAST8\_T, C\_INT\_FAST16\_T, C\_INT\_FAST32\_T, C\_INT\_FAST64\_T, C\_INTMAX\_T, and C\_  
23 INTPTR\_T shall each be a valid value for an integer kind type parameter on the processor or shall be  
24 -1 if the companion C processor defines the corresponding C type and there is no interoperating Fortran  
25 processor kind or -2 if the C processor does not define the corresponding C type.

26 The values of C\_FLOAT, C\_DOUBLE, and C\_LONG\_DOUBLE shall each be a valid value for a real  
27 kind type parameter on the processor or shall be -1 if the C processor's type does not have a precision  
28 equal to the precision of any of the Fortran processor's real kinds, -2 if the C processor's type does not  
29 have a range equal to the range of any of the Fortran processor's real kinds, -3 if the C processor's  
30 type has neither the precision nor range of any of the Fortran processor's real kinds, and equal to -4  
31 if there is no interoperating Fortran processor kind for other reasons. The values of C\_COMPLEX,  
32 C\_DOUBLE\_COMPLEX, and C\_LONG\_DOUBLE\_COMPLEX shall be the same as those of C\_FLOAT,  
33 C\_DOUBLE, and C\_LONG\_DOUBLE, respectively.

**NOTE 15.2**

If the C processor supports more than one variety of float, double or long double, the Fortran processor may find it helpful to select from among more than one ISO\_C\_BINDING module by a processor dependent means.

- 1 The value of C\_BOOL shall be a valid value for a logical kind parameter on the processor or shall be -1.
- 2 The value of C\_CHAR shall be a valid value for a character kind type parameter on the processor or shall be -1. The value of C\_CHAR is known as the **C character kind**.
- 4 The following entities shall be named constants of type character with a length parameter of one. The kind parameter value shall be equal to the value of C\_CHAR unless C\_CHAR = -1, in which case the kind parameter value shall be the same as for default kind. The values of these constants are specified in Table 15.1. In the case that C\_CHAR ≠ -1 the value is specified using C syntax. The semantics of these values are explained in 5.2.1 and 5.2.2 of the C standard.

Table 15.1: Names of C characters with special semantics

Name	C definition	Value	
		C.CHAR = -1	C.CHAR ≠ -1
C_NULL_CHAR	null character	CHAR(0)	'\0'
C_ALERT	alert	ACHAR(7)	'\a'
C_BACKSPACE	backspace	ACHAR(8)	'\b'
C_FORM_FEED	form feed	ACHAR(12)	'\f'
C_NEW_LINE	new line	ACHAR(10)	'\n'
C_CARRIAGE_RETURN	carriage return	ACHAR(13)	'\r'
C_HORIZONTAL_TAB	horizontal tab	ACHAR(9)	'\t'
C_VERTICAL_TAB	vertical tab	ACHAR(11)	'\v'

**NOTE 15.3**

The value of NEW\_LINE(C\_NEW\_LINE) is C\_NEW\_LINE (13.7.85).

- 9 The entities C\_PTR and C\_FUNPTR are described in 15.2.2.
- 10 The entity C\_NULL\_PTR shall be a named constant of type C\_PTR. The value of C\_NULL\_PTR shall be the same as the value NULL in C. The entity C\_NULL\_FUNPTR shall be a named constant of type C\_FUNPTR. The value of C\_NULL\_FUNPTR shall be that of a null pointer to a function in C.

**15.1.2 Procedures in the module**

14 In the detailed descriptions below, procedure names are generic and not specific.

15 A C procedure argument is often defined in terms of a **C address**. The C\_LOC and C\_FUNLOC functions are provided so that Fortran applications can determine the appropriate value to use with C facilities. The C\_ASSOCIATED function is provided so that Fortran programs can compare C addresses. The C\_F\_POINTER and C\_F\_PROCPOINTER subroutines provide a means of associating a Fortran pointer with the target of a C pointer.

**20 15.1.2.1 C\_ASSOCIATED (C\_PTR\_1 [, C\_PTR\_2])**

21 **Description.** Indicates the association status of C\_PTR\_1 or indicates whether C\_PTR\_1 and  
22 C\_PTR\_2 are associated with the same entity.

23 **Class.** Inquiry function.

1        **Arguments.**

2        C\_PTR\_1            shall be a scalar of type C\_PTR or C\_FUNPTR.

3        C\_PTR\_2            shall be a scalar of the same type as C\_PTR\_1.  
(optional)4        **Result Characteristics.** Default logical scalar.5        **Result Value.**6        *Case (i):*        If C\_PTR\_2 is absent, the result is false if C\_PTR\_1 is a C null pointer and true  
7                            otherwise.8        *Case (ii):*       If C\_PTR\_2 is present, the result is false if C\_PTR\_1 is a C null pointer. Otherwise,  
9                            the result is true if C\_PTR\_1 compares equal to C\_PTR\_2 in the sense of 6.3.2.3  
10                           and 6.5.9 of the C standard, and false otherwise.**NOTE 15.4**

The following example illustrates the use of C\_LOC and C\_ASSOCIATED.

```

USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_PTR, C_FLOAT, C_ASSOCIATED, C_LOC
INTERFACE
  SUBROUTINE FOO(GAMMA), BIND(C)
    IMPORT C_PTR
    TYPE(C_PTR), VALUE :: GAMMA
  END SUBROUTINE FOO
END INTERFACE
REAL(C_FLOAT), TARGET, DIMENSION(100) :: ALPHA
TYPE(C_PTR) :: BETA
...
IF (.NOT. C_ASSOCIATED(BETA)) THEN
  BETA = C_LOC(ALPHA)
ENDIF
CALL FOO(BETA)

```

11 **15.1.2.2 C\_F\_POINTER (CPTR, FPTR [, SHAPE])**12        **Description.** Associates a data pointer with the target of a C pointer and specifies its shape.13        **Class.** Subroutine.14        **Arguments.**CPTR            shall be a scalar of type C\_PTR. It is an INTENT(IN) argument. Its value  
shall be:

- (1) The C address of an interoperable data entity, or
- (2) The result of a reference to C\_LOC with a noninteroperable argument.

15                            The value of CPTR shall not be the C address of a Fortran variable that does  
16                            not have the TARGET attribute.

FPTR shall be a pointer. It is an INTENT(OUT) argument.

- (1) If the value of CPTR is the C address of an interoperable data entity, FPTR shall be a data pointer with type and type parameters interoperable with the type of the entity. In this case, FPTR becomes pointer associated with the target of CPTR. If FPTR is an array, its shape is specified by SHAPE and each lower bound is 1.
- (2) If the value of CPTR is the result of a reference to C\_LOC with a noninteroperable argument X, FPTR shall be a nonpolymorphic scalar pointer with the same type and type parameters as X. In this case, X or its target if it is a pointer shall not have been deallocated or have become undefined due to execution of a RETURN or END statement since the reference. FPTR becomes pointer associated with X or its target.

SHAPE shall be of type integer and rank one. It is an INTENT(IN) argument. (optional) SHAPE shall be present if and only if FPTR is an array; its size shall be equal to the rank of FPTR.

### 15.1.2.3 C\_F\_PROCPOINTER (CPTR, FPTR)

**Description.** Associates a procedure pointer with the target of a C function pointer.

**Class.** Subroutine.

**Arguments.**

CPTR shall be a scalar of type C\_FUNPTR. It is an INTENT(IN) argument. Its value shall be the C address of a procedure that is interoperable.

FPTR shall be a procedure pointer. It is an INTENT(OUT) argument. The interface for FPTR shall be interoperable with the prototype that describes the target of CPTR. FPTR becomes pointer associated with the target of CPTR.

#### NOTE 15.5

The term “target” in the descriptions of C\_F\_POINTER and C\_F\_PROCPOINTER denotes the entity referenced by a C pointer, as described in 6.2.5 of the C standard.

### 15.1.2.4 C\_FUNLOC (X)

**Description.** Returns the C address of the argument.

**Class.** Inquiry function.

**Argument.** X shall either be a procedure that is interoperable, or a procedure pointer associated with an interoperable procedure.

**Result Characteristics.** Scalar of type C\_FUNPTR.

**Result Value.**

The result value will be described using the result name CPTR. The result is determined as if C\_FUNPTR were a derived type containing an implicit-interface procedure pointer component PX and the pointer assignment CPTR%PX => X were executed.

The result is a value that can be used as an actual CPTR argument in a call to C\_F\_PROC-

1 POINTER where FPTR has attributes that would allow the pointer assignment  $FPTR \Rightarrow X$ .  
 2 Such a call to C\_F\_PROCPOINTER shall have the effect of the pointer assignment  $FPTR \Rightarrow$   
 3 X.

#### 4 15.1.2.5 C\_LOC (X)

5 **Description.** Returns the C address of the argument.

6 **Class.** Inquiry function.

7 **Argument.** X shall either

- 8 (1) have interoperable type and type parameters and be  
 9 (a) a variable that has the TARGET attribute and is interoperable,  
 10 (b) an allocated allocatable variable that has the TARGET attribute and is not an array  
 11 of zero size, or  
 12 (c) an associated scalar pointer, or  
 13 (2) be a nonpolymorphic scalar, have no length type parameters, and be  
 14 (a) a nonallocatable, nonpointer variable that has the TARGET attribute,  
 15 (b) an allocated allocatable variable that has the TARGET attribute, or  
 16 (c) an associated pointer.

17 **Result Characteristics.** Scalar of type C\_PTR.

18 **Result Value.**

19 The result value will be described using the result name CPTR.

- 20 (1) If X is a scalar data entity, the result is determined as if C\_PTR were a derived type  
 21 containing a scalar pointer component PX of the type and type parameters of X and the  
 22 pointer assignment  $CPTR\%PX \Rightarrow X$  were executed.  
 23 (2) If X is an array data entity, the result is determined as if C\_PTR were a derived type  
 24 containing a scalar pointer component PX of the type and type parameters of X and the  
 25 pointer assignment of  $CPTR\%PX$  to the first element of X were executed.

26 If X is a data entity that is interoperable or has interoperable type and type parameters, the  
 27 result is the value that the C processor returns as the result of applying the unary “&” operator  
 28 (as defined in the C standard, 6.5.3.2) to the target of CPTR

29 The result is a value that can be used as an actual CPTR argument in a call to C\_F\_POINTER  
 30 where FPTR has attributes that would allow the pointer assignment  $FPTR \Rightarrow X$ . Such a call  
 31 to C\_F\_POINTER shall have the effect of the pointer assignment  $FPTR \Rightarrow X$ .

#### NOTE 15.6

Where the actual argument is of noninteroperable type or type parameters, the result of C\_LOC provides an opaque “handle” for it. In an actual implementation, this handle may be the C address of the argument; however, portable C functions should treat it as a void (generic) C pointer that cannot be dereferenced (6.5.3.2 in the C standard).

## 32 15.2 Interoperability between Fortran and C entities

33 The following subclauses define the conditions under which a Fortran entity is interoperable. If a Fortran  
 34 entity is interoperable, an equivalent entity may be defined by means of C and the Fortran entity is said

1 to be interoperable with the C entity. There does not have to be such an interoperating C entity.

**NOTE 15.7**

A Fortran entity can be interoperable with more than one C entity.

2 **15.2.1 Interoperability of intrinsic types**

3 Table 15.2 shows the interoperability between Fortran intrinsic types and C types. A Fortran intrinsic  
4 type with particular type parameter values is interoperable with a C type if the type and kind type  
5 parameter value are listed in the table on the same row as that C type; if the type is character, inter-  
6 operability also requires that the length type parameter be omitted or be specified by an initialization  
7 expression whose value is one. A combination of Fortran type and type parameters that is interoperable  
8 with a C type listed in the table is also interoperable with any unqualified C type that is compatible  
9 with the listed C type.

10 The second column of the table refers to the named constants made accessible by the ISO\_C\_BINDING  
11 intrinsic module. If the value of any of these named constants is negative, there is no combination of  
12 Fortran type and type parameters interoperable with the C type shown in that row.

13 A combination of intrinsic type and type parameters is **interoperable** if it is interoperable with a C  
14 type.

Table 15.2: **Interoperability between Fortran and C types**

Fortran type	Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)	C type
INTEGER	C_INT	int
	C_SHORT	short int
	C_LONG	long int
	C_LONG_LONG	long long int
	C_SIGNED_CHAR	signed char unsigned char
	C_SIZE_T	size_t
	C_INT8_T	int8_t
	C_INT16_T	int16_t
	C_INT32_T	int32_t
	C_INT64_T	int64_t
	C_INT_LEAST8_T	int_least8_t
	C_INT_LEAST16_T	int_least16_t
	C_INT_LEAST32_T	int_least32_t
	C_INT_LEAST64_T	int_least64_t
	C_INT_FAST8_T	int_fast8_t
	C_INT_FAST16_T	int_fast16_t
	C_INT_FAST32_T	int_fast32_t
	C_INT_FAST64_T	int_fast64_t
C_INTMAX_T	intmax_t	
C_INTPTR_T	intptr_t	
REAL	C_FLOAT	float
	C_DOUBLE	double

**Interoperability between Fortran and C types**

(cont.)

Fortran type	Named constant from the ISO_C_BINDING module (kind type parameter if value is positive)	C type
	C_LONG_DOUBLE	long double
COMPLEX	C_FLOAT_COMPLEX	float _Complex
	C_DOUBLE_COMPLEX	double _Complex
	C_LONG_DOUBLE_COMPLEX	long double _Complex
LOGICAL	C_BOOL	_Bool
CHARACTER	C_CHAR	char
The above mentioned C types are defined in the C standard, clauses 6.2.5, 7.17, and 7.18.1.		

**NOTE 15.8**

For example, the type integer with a kind type parameter of C\_SHORT is interoperable with the C type short or any C type derived (via typedef) from short.

**NOTE 15.9**

The C standard specifies that the representations for nonnegative signed integers are the same as the corresponding values of unsigned integers. Because Fortran does not provide direct support for unsigned kinds of integers, the ISO\_C\_BINDING module does not make accessible named constants for their kind type parameter values. Instead a user can use the signed kinds of integers to interoperate with the unsigned types and all their qualified versions as well. This has the potentially surprising side effect that the C type unsigned char is interoperable with the type integer with a kind type parameter of C\_SIGNED\_CHAR.

**1 15.2.2 Interoperability with C pointer types**

- 2 C\_PTR and C\_FUNPTR shall be a derived types with private components. C\_PTR is interoperable with
- 3 any C object pointer type. C\_FUNPTR is interoperable with any C function pointer type.

**NOTE 15.10**

This implies that a C processor is required to have the same representation method for all C object pointer types and the same representation method for all C function pointer types if the C processor is to be the target of interoperability of a Fortran processor. The C standard does not impose this requirement.

**NOTE 15.11**

The function C\_LOC can be used to return a value of type C\_PTR that is the C address of an allocated allocatable variable. The function C\_FUNLOC can be used to return a value of type C\_FUNPTR that is the C address of a procedure. For C\_LOC and C\_FUNLOC the returned value is of an interoperable type and thus may be used in contexts where the procedure or allocatable variable is not directly allowed. For example, it could be passed as an actual argument to a C function.

Similarly, type C\_FUNPTR or C\_PTR can be used in a dummy argument or structure component and can have a value that is the C address of a procedure or allocatable variable, even in contexts where a procedure or allocatable variable is not directly allowed.

### 1 15.2.3 Interoperability of derived types and C struct types

- 2 A Fortran derived type is **interoperable** if it has the BIND attribute.
- 3 C1501 (R429) A derived type with the BIND attribute shall not be a SEQUENCE type.
- 4 C1502 (R429) A derived type with the BIND attribute shall not have type parameters.
- 5 C1503 (R429) A derived type with the BIND attribute shall not have the EXTENDS attribute.
- 6 C1504 (R429) A derived type with the BIND attribute shall not have a *type-bound-procedure-part*.
- 7 C1505 (R429) Each component of a derived type with the BIND attribute shall be a nonpointer,  
8 nonallocatable data component with interoperable type and type parameters.

#### NOTE 15.12

The syntax rules and their constraints require that a derived type that is interoperable have components that are all data objects that are interoperable. No component is permitted to be a procedure or allocatable, but a component of type C\_FUNPTR or C\_PTR may hold the C address of such an entity.

- 9 A Fortran derived type is interoperable with a C struct type if the derived-type definition of the Fortran  
10 type specifies BIND(C) (4.5.1), the Fortran derived type and the C struct type have the same number  
11 of components, and the components of the Fortran derived type have types and type parameters that  
12 are interoperable with the types of the corresponding components of the struct type. A component of  
13 a Fortran derived type and a component of a C struct type correspond if they are declared in the same  
14 relative position in their respective type definitions.

#### NOTE 15.13

The names of the corresponding components of the derived type and the C struct type need not be the same.

- 15 There is no Fortran type that is interoperable with a C struct type that contains a bit field or that  
16 contains a flexible array member. There is no Fortran type that is interoperable with a C union type.

#### NOTE 15.14

For example, the C type myctype, declared below, is interoperable with the Fortran type myftype, declared below.

```
typedef struct {
    int m, n;
    float r;
} myctype
```

```
USE, INTRINSIC :: ISO_C_BINDING
TYPE, BIND(C) :: MYFTYPE
    INTEGER(C_INT) :: I, J
    REAL(C_FLOAT) :: S
END TYPE MYFTYPE
```

The names of the types and the names of the components are not significant for the purposes of determining whether a Fortran derived type is interoperable with a C struct type.

**NOTE 15.15**

The C standard requires the names and component names to be the same in order for the types to be compatible (C standard, clause 6.2.7). This is similar to Fortran's rule describing when sequence derived types are considered to be the same type. This rule was not extended to determine whether a Fortran derived type is interoperable with a C struct type because the case of identifiers is significant in C but not in Fortran.

1 **15.2.4 Interoperability of scalar variables**

2 A scalar Fortran variable is **interoperable** if its type and type parameters are interoperable and it has  
3 neither the pointer nor the allocatable attribute.

4 An interoperable scalar Fortran variable is interoperable with a scalar C entity if their types and type  
5 parameters are interoperable.

6 **15.2.5 Interoperability of array variables**

7 An array Fortran variable is **interoperable** if its type and type parameters are interoperable and it is  
8 of explicit shape or assumed size.

9 An explicit-shape or assumed-size array of rank  $r$ , with a shape of  $[ e_1 \dots e_r ]$  is interoperable with  
10 a C array if its size is nonzero and

- 11 (1) either  
12 (a) the array is assumed size, and the C array does not specify a size, or  
13 (b) the array is an explicit shape array, and the extent of the last dimension ( $e_r$ ) is the  
14 same as the size of the C array, and
- 15 (2) either  
16 (a)  $r$  is equal to one, and an element of the array is interoperable with an element of the  
17 C array, or  
18 (b)  $r$  is greater than one, and an explicit-shape array with shape of  $[ e_1 \dots e_{r-1} ]$ ,  
19 with the same type and type parameters as the original array, is interoperable with a  
20 C array of a type equal to the element type of the original C array.

**NOTE 15.16**

An element of a multi-dimensional C array is an array type, so a Fortran array of rank one is not interoperable with a multidimensional C array.

**NOTE 15.17**

A polymorphic, allocatable, or pointer array is never interoperable. Such arrays are not explicit shape or assumed size.

**NOTE 15.18**

For example, a Fortran array declared as

```
INTEGER :: A(18, 3:7, *)
```

is interoperable with a C array declared as

```
int b[][5][18]
```

**NOTE 15.19**

The C programming language defines null-terminated strings, which are actually arrays of the C type `char` that have a C null character in them to indicate the last valid element. A Fortran array of type `character` with a `kind` type parameter equal to `C_CHAR` is interoperable with a C string.

Fortran's rules of sequence association (12.4.1.5) permit a character scalar actual argument to be associated with a dummy argument array. This makes it possible to argument associate a Fortran character string with a C string.

Note 15.23 has an example of interoperation between Fortran and C strings.

## 1 15.2.6 Interoperability of procedures and procedure interfaces

2 A Fortran procedure is **interoperable** if it has the `BIND` attribute, that is, if its interface is specified  
3 with a *proc-language-binding-spec*.

4 A Fortran procedure interface is interoperable with a C function prototype if

- 5 (1) the interface has the `BIND` attribute;
- 6 (2) either
  - 7 (a) the interface describes a function whose result variable is a scalar that is interoperable  
8 with the result of the prototype or
  - 9 (b) the interface describes a subroutine, and the prototype has a result type of `void`;
- 10 (3) the number of dummy arguments of the interface is equal to the number of formal parameters  
11 of the prototype;
- 12 (4) any dummy argument with the `VALUE` attribute is interoperable with the corresponding  
13 formal parameter of the prototype;
- 14 (5) any dummy argument without the `VALUE` attribute corresponds to a formal parameter of  
15 the prototype that is of a pointer type, and the dummy argument is interoperable with an  
16 entity of the referenced type (C standard, 6.2.5, 7.17, and 7.18.1) of the formal parameter;  
17 and
- 18 (6) the prototype does not have variable arguments as denoted by the ellipsis (...).

**NOTE 15.20**

The **referenced type** of a C pointer type is the C type of the object that the C pointer type points to. For example, the referenced type of the pointer type `int *` is `int`.

**NOTE 15.21**

The C language allows specification of a C function that can take a variable number of arguments (C standard, 7.15). This standard does not provide a mechanism for Fortran procedures to interoperate with such C functions.

19 A formal parameter of a C function prototype corresponds to a dummy argument of a Fortran interface if  
20 they are in the same relative positions in the C parameter list and the dummy argument list, respectively.

**NOTE 15.22**

For example, a Fortran procedure interface described by

```
INTERFACE
  FUNCTION FUNC(I, J, K, L, M), BIND(C)
    USE, INTRINSIC :: ISO_C_BINDING
```

**NOTE 15.22 (cont.)**

```

INTEGER(C_SHORT) :: FUNC
INTEGER(C_INT), VALUE :: I
REAL(C_DOUBLE) :: J
INTEGER(C_INT) :: K, L(10)
TYPE(C_PTR), VALUE :: M
END FUNCTION FUNC
END INTERFACE

```

is interoperable with the C function prototype

```
short func(int i, double *j, int *k, int l[10], void *m)
```

A C pointer may correspond to a Fortran dummy argument of type C\_PTR or to a Fortran scalar that does not have the VALUE attribute. In the above example, the C pointers `j` and `k` correspond to the Fortran scalars `J` and `K`, respectively, and the C pointer `m` corresponds to the Fortran dummy argument `M` of type C\_PTR.

**NOTE 15.23**

The interoperability of Fortran procedure interfaces with C function prototypes is only one part of invocation of a C function from Fortran. There are four pieces to consider in such an invocation: the procedure reference, the Fortran procedure interface, the C function prototype, and the C function. Conversely, the invocation of a Fortran procedure from C involves the function reference, the C function prototype, the Fortran procedure interface, and the Fortran procedure. In order to determine whether a reference is allowed, it is necessary to consider all four pieces.

For example, consider a C function that can be described by the C function prototype

```
void copy(char in[], char out[]);
```

Such a function may be invoked from Fortran as follows:

```

USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_CHAR, C_NULL_CHAR
INTERFACE
  SUBROUTINE COPY(IN, OUT), BIND(C)
    IMPORT C_CHAR
    CHARACTER(KIND=C_CHAR), DIMENSION(*) :: IN, OUT
  END SUBROUTINE COPY
END INTERFACE

CHARACTER(LEN=10, KIND=C_CHAR) :: &
&    DIGIT_STRING = C_CHAR_'123456789' // C_NULL_CHAR
CHARACTER(KIND=C_CHAR) :: DIGIT_ARR(10)

CALL COPY(DIGIT_STRING, DIGIT_ARR)
PRINT '(1X, A1)', DIGIT_ARR(1:9)
END

```

**NOTE 15.23 (cont.)**

The procedure reference has character string actual arguments. These correspond to character array dummy arguments in the procedure interface body as allowed by Fortran's rules of sequence association (12.4.1.5). Those array dummy arguments in the procedure interface are interoperable with the formal parameters of the C function prototype. The C function is not shown here, but is assumed to be compatible with the C function prototype.

**1 15.3 Interoperation with C global variables**

2 A C variable with external linkage may interoperate with a common block or with a variable declared  
3 in the scope of a module. The common block or variable shall be specified to have the BIND attribute.

4 At most one variable that is associated with a particular C variable with external linkage is permitted  
5 to be declared within a program. A variable shall not be initially defined by more than one processor.

6 If a common block is specified in a BIND statement, it shall be specified in a BIND statement with  
7 the same binding label in each scoping unit in which it is declared. A C variable with external linkage  
8 interoperates with a common block that has been specified in a BIND statement

9 (1) if the C variable is of a struct type and the variables that are members of the common block  
10 are interoperable with corresponding components of the struct type, or

11 (2) if the common block contains a single variable, and the variable is interoperable with the C  
12 variable.

13 There does not have to be an associated C entity for a Fortran entity with the BIND attribute.

**NOTE 15.24**

The following are examples of the usage of the BIND attribute for variables and for a common block. The Fortran variables, C\_EXTERN and C2, interoperate with the C variables, c\_extern and myVariable, respectively. The Fortran common blocks, COM and SINGLE, interoperate with the C variables, com and single, respectively.

```

MODULE LINK_TO_C_VARS
  USE, INTRINSIC :: ISO_C_BINDING
  INTEGER(C_INT), BIND(C) :: C_EXTERN
  INTEGER(C_LONG) :: C2
  BIND(C, NAME='myVariable'):: C2

  COMMON /COM/ R, S
  REAL(C_FLOAT) :: R, S, T
  BIND(C) :: /COM/, /SINGLE/
  COMMON /SINGLE/ T
END MODULE LINK_TO_C_VARS

int c_extern;
long myVariable;
struct {float r, s;} com;
float single;

```

### 1 **15.3.1 Binding labels for common blocks and variables**

2 The **binding label** of a variable or common block is a value of type default character that specifies the  
3 name by which the variable or common block is known to the companion processor.

4 If a variable or common block has the BIND attribute specified with a NAME= specifier, the binding  
5 label is the value of the expression specified for the NAME= specifier. The case of letters in the binding  
6 label is significant, but leading and trailing blanks are ignored. If a variable or common block has the  
7 BIND attribute specified without a NAME= specifier, the binding label is the same as the name of the  
8 entity using lower case letters.

9 The binding label of a C variable with external linkage is the same as the name of the C variable. A  
10 Fortran variable or common block with the BIND attribute that has the same binding label as a C  
11 variable with external linkage is associated with that variable.

## 12 **15.4 Interoperation with C functions**

13 A procedure that is interoperable may be defined either by means other than Fortran or by means of a  
14 Fortran subprogram, but not both.

15 If the procedure is defined by means other than Fortran, it shall

- 16 (1) be describable by a C prototype that is interoperable with the interface,
- 17 (2) have external linkage as defined by 6.2.2 of the C standard, and
- 18 (3) have the same binding label as the interface.

19 A reference to such a procedure causes the function described by the C prototype to be called as specified  
20 in the C standard.

21 A reference in C to a procedure that has the BIND attribute, has the same binding label, and is defined  
22 by means of Fortran, causes the Fortran procedure to be invoked.

23 A procedure defined by means of Fortran shall not invoke setjmp or longjmp (C standard, 7.13). If a  
24 procedure defined by means other than Fortran invokes setjmp or longjmp, that procedure shall not  
25 cause any procedure defined by means of Fortran to be invoked. A procedure defined by means of  
26 Fortran shall not be invoked as a signal handler (C standard, 7.14.1).

27 If a procedure defined by means of Fortran and a procedure defined by means other than Fortran perform  
28 input/output operations on the same external file, the results are processor dependent (9.4.3).

### 29 **15.4.1 Binding labels for procedures**

30 A **binding label** is a value of type default character that specifies the name by which a procedure with  
31 the BIND attribute is known to the companion processor.

32 If a procedure has the BIND attribute with the NAME= specifier, the procedure has a binding label  
33 whose value is that of the expression in the NAME= specifier. The case of letters in the binding label is  
34 significant, but leading and trailing blanks are ignored. If a procedure has the BIND attribute with no  
35 NAME= specifier, and the procedure is not a dummy procedure, then the binding label of the procedure  
36 is the same as the name of the procedure using lower case letters.

37 The binding label for a C function with external linkage is the same as the C function name.

**NOTE 15.25**

In the following sample, the binding label of C\_SUB is "c\_sub", and the binding label of C\_FUNC is "C\_funC".

```
SUBROUTINE C_SUB, BIND(C)
  ...
END SUBROUTINE C_SUB

INTEGER(C_INT) FUNCTION C_FUNC(), BIND(C, NAME="C_funC")
  USE, INTRINSIC :: ISO_C_BINDING
  ...
END FUNCTION C_FUNC
```

The C standard permits functions to have names that are not permitted as Fortran names; it also distinguishes between names that would be considered as the same name in Fortran. For example, a C name may begin with an underscore, and C names that differ in case are distinct names.

The specification of a binding label allows a program to use a Fortran name to refer to a procedure defined by a companion processor.

**1 15.4.2 Exceptions and IEEE arithmetic procedures**

- 2 A procedure defined by means other than Fortran shall not use signal (C standard, 7.14.1) to change
- 3 the handling of any exception that is being handled by the Fortran processor.
- 4 A procedure defined by means other than Fortran shall not alter the floating point status (14.6 other
- 5 than by setting an exception flag to signaling.
- 6 The values of the floating point exception flags on entry to a procedure defined by means other than
- 7 Fortran are processor-dependent.

## 1 Section 16: Scope, association, and definition

2 Entities are identified by identifiers within a **scope** that is a program, a scoping unit, a construct, a  
3 single statement, or part of a statement.

- 4 • A **global identifier** has a scope of a program (2.2.1);
- 5 • A **local identifier** has a scope of a scoping unit (2.2);
- 6 • An identifier of a **construct entity** has a scope of a construct (7.4.3, 7.4.4, 8.1);
- 7 • An identifier of a **statement entity** has a scope of a statement or part of a statement (3.3).

8 An entity may be identified by

- 9 (1) A name (3.2.1),
- 10 (2) A statement label (3.2.4),
- 11 (3) An external input/output unit number (9.4),
- 12 (4) An identifier of a pending data transfer operation (9.5.1.8, 9.6),
- 13 (5) A generic identifier (12.3.2.1), or
- 14 (6) A binding label (15.4.1, 15.3.1).

15 By means of association, an entity may be referred to by the same identifier or a different identifier in  
16 a different scoping unit, or by a different identifier in the same scoping unit.

### 17 16.1 Scope of global identifiers

18 Program units, common blocks, external procedures, procedure binding labels, and variables that have  
19 the BIND attribute are **global entities** of a program. The name of a program unit, common block,  
20 or external procedure is a global identifier and shall not be the same as the name of any other such  
21 global entity in the same program, except that an intrinsic module may have the same name as another  
22 program unit, common block, or external procedure in the same program. A binding label of a global  
23 entity of the program is a global identifier and shall not be the same as the binding label of any other  
24 global entity of the program; nor shall it be the same as a name of any other global entity of the program  
25 that is not an intrinsic module, ignoring differences in case. A global entity of the program shall not be  
26 identified by more than one binding label.

#### NOTE 16.1

The name of a global entity may be the same as a binding label that identifies the same global entity.

#### NOTE 16.2

Of the various types of procedures, only external procedures have global names. An implementation may wish to assign global names to other entities in the Fortran program such as internal procedures, intrinsic procedures, procedures implementing intrinsic operators, procedures implementing input/output operations, etc. If this is done, it is the responsibility of the processor to ensure that none of these names conflicts with any of the names of the external procedures, with other globally named entities in a standard-conforming program, or with each other. For example,

**NOTE 16.2 (cont.)**

this might be done by including in each such added name a character that is not allowed in a standard-conforming name or by using such a character to combine a local designation with the global name of the program unit in which it appears.

1 External input/output units and pending data transfer operations are global entities.

## 2 **16.2 Scope of local identifiers**

3 Within a scoping unit, identifiers of entities in the following classes:

- 4 (1) Named variables that are not statement or construct entities (16.3), named constants, named  
 5 constructs, statement functions, internal procedures, module procedures, dummy procedures,  
 6 intrinsic procedures, abstract interfaces, generic interfaces, derived types, namelist groups,  
 7 external procedures accessed via USE, and statement labels,  
 8 (2) Type parameters, components, and type-bound procedure bindings, in a separate class for  
 9 each type, and  
 10 (3) Argument keywords, in a separate class for each procedure with an explicit interface

11 are local identifiers in that scoping unit.

12 Within a scoping unit, a local identifier of an entity of class (1) shall not be the same as a global identifier  
 13 used in that scoping unit unless the global identifier

- 14 (1) is used only as the *use-name* of a *rename* in a USE statement,  
 15 (2) is a common block name (16.2.1),  
 16 (3) is an external procedure name that is also a generic name, or  
 17 (4) is an external function name and the scoping unit is its defining subprogram (16.2.2).

18 Within a scoping unit, a local identifier of one class shall not be the same as another local identifier of  
 19 the same class, except that a generic name may be the same as the name of a procedure as explained  
 20 in 12.3.2.1 or the same as the name of a derived type (4.5.9). A local identifier of one class may be the  
 21 same as a local identifier of another class.

**NOTE 16.3**

An intrinsic procedure is inaccessible by its own name in a scoping unit that uses the same name as a local identifier of class (1) for a different entity. For example, in the program fragment

```

SUBROUTINE SUB
  ...
  A = SIN (K)
  ...
CONTAINS
  FUNCTION SIN (X)
    ...
  END FUNCTION SIN
END SUBROUTINE SUB

```

any reference to function SIN in subroutine SUB refers to the internal function SIN, not to the intrinsic function of the same name.

1 A local identifier identifies an entity in a scoping unit and may be used to identify an entity in another  
2 scoping unit except in the following cases:

- 3 (1) The name that appears as a *subroutine-name* in a *subroutine-stmt* has limited use within  
4 the scope established by the *subroutine-stmt*. It can be used to identify recursive references  
5 of the subroutine or to identify a common block (the latter is possible only for internal and  
6 module subroutines).
- 7 (2) The name that appears as a *function-name* in a *function-stmt* has limited use within the  
8 scope established by that *function-stmt*. It can be used to identify the result variable, to  
9 identify recursive references of the function, or to identify a common block (the latter is  
10 possible only for internal and module functions).
- 11 (3) The name that appears as an *entry-name* in an *entry-stmt* has limited use within the scope  
12 of the subprogram in which the *entry-stmt* appears. It can be used to identify the result  
13 variable if the subprogram is a function, to identify recursive references, or to identify a  
14 common block (the latter is possible only if the *entry-stmt* is in a module subprogram).

### 15 16.2.1 Local identifiers that are the same as common block names

16 A name that identifies a common block in a scoping unit shall not be used to identify a constant or an  
17 intrinsic procedure in that scoping unit. If a local identifier is also the name of a common block, the  
18 appearance of that name in any context other than as a common block name in a COMMON or SAVE  
19 statement is an appearance of the local identifier.

#### NOTE 16.4

An intrinsic procedure name may be a common block name in a scoping unit that does not reference the intrinsic procedure.

### 20 16.2.2 Function results

21 For each FUNCTION statement or ENTRY statement in a function subprogram, there is a result  
22 variable. If there is no RESULT clause, the result variable has the same name as the function being  
23 defined; otherwise, the result variable has the name specified in the RESULT clause.

### 24 16.2.3 Restrictions on generic declarations

25 This subclause contains the rules that shall be satisfied by every pair of specific procedures that have  
26 the same generic identifier within a scoping unit. If a generic procedure is accessed from a module, the  
27 rules apply to all the specific versions even if some of them are inaccessible by their specific names.

#### NOTE 16.5

In most scoping units, the possible sources of procedures with a particular generic identifier are the accessible interface blocks and the generic bindings other than names for the accessible objects in that scoping unit. In a type definition, they are the generic bindings, including those from a parent type.

28 Two dummy arguments are **distinguishable** if neither is a subroutine and neither is TKR compatible  
29 (5.1.1.2) with the other.

30 Within a scoping unit, if two procedures have the same generic operator and the same number of  
31 arguments or both define assignment, one shall have a dummy argument that corresponds by position  
32 in the argument list to a dummy argument of the other that is distinguishable with it.

33 Within a scoping unit, if two procedures have the same *dtio-generic-spec* (12.3.2.1), their *dtv* arguments

- 1 shall be type incompatible or have different kind type parameters.
- 2 Within a scoping unit, two procedures that have the same generic name shall both be subroutines or  
3 both be functions, and
- 4 (1) there is a non-passed-object dummy data object in one or the other of them such that
- 5 (a) the number of dummy data objects in one that are nonoptional, are not passed-object,  
6 and with which that dummy data object is TKR compatible, possibly including that  
7 dummy data object itself,  
8 exceeds
- 9 (b) the number of non-passed-object dummy data objects, both optional and nonoptional,  
10 in the other that are not distinguishable with that dummy data object;
- 11 (2) both have passed-object dummy arguments and the passed-object dummy arguments are  
12 distinguishable; or
- 13 (3) at least one of them shall have both
- 14 (a) A nonoptional non-passed-object dummy argument at an effective position such that  
15 either the other procedure has no dummy argument at that effective position or the  
16 dummy argument at that position is distinguishable with it; and
- 17 (b) A nonoptional non-passed-object dummy argument whose name is such that either  
18 the other procedure has no dummy argument with that name or the dummy argument  
19 with that name is distinguishable with it.
- 20 Further, the dummy argument that disambiguates by position shall either be the same as  
21 or occur earlier in the argument list than the one that disambiguates by name.

22 The **effective position** of a dummy argument is its position in the argument list after any passed-object  
23 dummy argument has been removed.

24 Within a scoping unit, if a generic name is the same as the name of a generic intrinsic procedure, the  
25 generic intrinsic procedure is not accessible if the procedures in the interface and the intrinsic procedure  
26 are not all functions or not all subroutines. If a generic invocation applies to both a specific procedure  
27 from an interface and an accessible generic intrinsic procedure, it is the specific procedure from the  
28 interface that is referenced.

#### NOTE 16.6

An extensive explanation of the application of these rules is in [C.11.2](#).

### 29 16.2.4 Components, type parameters, and bindings

30 A component name has the scope of its derived-type definition. Outside the type definition, it may  
31 appear only within a designator of a component of a structure of that type or as a component keyword  
32 in a structure constructor for that type.

33 A type parameter name has the scope of its derived-type definition. Outside the derived-type definition,  
34 it may appear only as a type parameter keyword in a *derived-type-spec* for the type or as the *type-param-*  
35 *name* of a *type-param-inquiry*.

36 The binding name ([4.5.4](#)) of a type-bound procedure has the scope of its derived-type definition. Outside  
37 of the derived-type definition, it may appear only as the *binding-name* in a procedure reference.

38 A generic binding for which the *generic-spec* is not a *generic-name* has a scope that consists of all scoping  
39 units in which an entity of the type is accessible.

40 A component name or binding name may appear only in scoping units in which it is accessible.

1 The accessibility of components and bindings is specified in 4.5.3.6 and 4.5.4.

## 2 **16.2.5 Argument keywords**

3 As an argument keyword, a dummy argument name in an internal procedure, module procedure, or  
4 an interface body has a scope of the scoping unit of the host of the procedure or interface body. It  
5 may appear only in a procedure reference for the procedure of which it is a dummy argument. If the  
6 procedure or interface body is accessible in another scoping unit by use association or host association  
7 (16.4.1.2, 16.4.1.3), the argument keyword is accessible for procedure references for that procedure in  
8 that scoping unit.

9 A dummy argument name in an intrinsic procedure has a scope as an argument keyword of the scoping  
10 unit in which the reference to the procedure occurs. As an argument keyword, it may appear only in a  
11 procedure reference for the procedure of which it is a dummy argument.

## 12 **16.3 Statement and construct entities**

13 A variable that appears as a *data-i-do-variable* in a DATA statement or an *ac-do-variable* in an array  
14 constructor, as a dummy argument in a statement function statement, or as an *index-name* in a FORALL  
15 statement is a statement entity. A variable that appears as an *index-name* in a FORALL construct or  
16 an *associate-name* in a SELECT TYPE or ASSOCIATE construct is a construct entity.

17 The name of a *data-i-do-variable* in a DATA statement or an *ac-do-variable* in an array constructor has  
18 a scope of its *data-implied-do*. It is a scalar variable that has the type and type parameters that it would  
19 have if it were the name of a variable in the scoping unit that includes the DATA statement or array  
20 constructor, and this type shall be integer type; it has no other attributes. The appearance of a name as  
21 a *data-i-do-variable* of an implied-DO in a DATA statement or an *ac-do-variable* in an array constructor  
22 is not an implicit declaration of a variable whose scope is the scoping unit that contains the statement.

23 The name of a variable that appears as an *index-name* in a FORALL statement or FORALL construct  
24 has a scope of the statement or construct. It is a scalar variable that has the type and type parameters  
25 that it would have if it were the name of a variable in the scoping unit that includes the FORALL, and  
26 this type shall be integer type; it has no other attributes. The appearance of a name as an *index-name*  
27 in a FORALL statement or FORALL construct is not an implicit declaration of a variable whose scope  
28 is the scoping unit that contains the statement or construct.

29 The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement  
30 in which it appears. It is a scalar that has the type and type parameters that it would have if it were the name of a variable  
31 in the scoping unit that includes the statement function; it has no other attributes.

32 Except for a common block name or a scalar variable name, a global identifier or a local identifier of  
33 class (1) (16.2) in the scoping unit that contains a statement shall not be the name of a statement entity  
34 of that statement. Within the scope of a statement entity, another statement entity shall not have the  
35 same name.

36 If a global or local identifier accessible in the scoping unit of a statement is the same as the name of a  
37 statement entity in that statement, the name is interpreted within the scope of the statement entity as  
38 that of the statement entity. Elsewhere in the scoping unit, including parts of the statement outside the  
39 scope of the statement entity, the name is interpreted as the global or local identifier.

40 Except for a common block name or a scalar variable name, a global identifier or a local identifier of  
41 class (1) (16.2) in the scoping unit of a FORALL statement or FORALL construct shall not be the same  
42 as any of its *index-names*. An *index-name* of a contained FORALL statement or FORALL construct  
43 shall not be the same as an *index-name* of any of its containing FORALL constructs.

1 If a global or local identifier accessible in the scoping unit of a FORALL statement or FORALL construct  
2 is the same as the *index-name*, the name is interpreted within the scope of the FORALL statement or  
3 FORALL construct as that of the *index-name*. Elsewhere in the scoping unit, the name is interpreted  
4 as the global or local identifier.

5 The associate name of a SELECT TYPE construct has a separate scope for each block of the construct.  
6 Within each block, it has the declared type, dynamic type, type parameters, rank, and bounds specified  
7 in 8.1.5.2.

8 The associate names of an ASSOCIATE construct have the scope of the block. They have the declared  
9 type, dynamic type, type parameters, rank, and bounds specified in 8.1.4.2.

10 If a global or local identifier accessible in the scoping unit of a SELECT TYPE or ASSOCIATE construct  
11 is the same as an associate name, the name is interpreted within the blocks of the SELECT TYPE or  
12 ASSOCIATE construct as that of the associate name. Elsewhere in the scoping unit, the name is  
13 interpreted as the global or local identifier.

## 14 16.4 Association

15 Two entities may become associated by name association, pointer association, storage association, or  
16 inheritance association.

### 17 16.4.1 Name association

18 There are five forms of **name association**: argument association, use association, host association,  
19 linkage association, and construct association. Argument, use, and host association provide mechanisms  
20 by which entities known in one scoping unit may be accessed in another scoping unit.

#### 21 16.4.1.1 Argument association

22 The rules governing argument association are given in Section 12. As explained in 12.4, execution of a  
23 procedure reference establishes an association between an actual argument and its corresponding dummy  
24 argument. Argument association may be sequence association (12.4.1.5).

25 The name of the dummy argument may be different from the name, if any, of its associated actual  
26 argument. The dummy argument name is the name by which the associated actual argument is known,  
27 and by which it may be accessed, in the referenced procedure.

#### NOTE 16.7

An actual argument may be a nameless data entity, such as an expression that is not simply a variable or constant.
--

28 Upon termination of execution of a procedure reference, all argument associations established by that  
29 reference are terminated. A dummy argument of that procedure may be associated with an entirely  
30 different actual argument in a subsequent invocation of the procedure.

#### 31 16.4.1.2 Use association

32 **Use association** is the association of names in different scoping units specified by a USE statement. The  
33 rules governing use association are given in 11.2.1. They allow for renaming of entities being accessed.  
34 Use association allows access in one scoping unit to entities defined in another scoping unit; it remains  
35 in effect throughout the execution of the program.

1 **16.4.1.3 Host association**

2 An internal subprogram, a module subprogram, or a derived-type definition has access to the named  
 3 entities from its host via **host association**. An interface body has access via host association to the  
 4 named entities from its host that are made accessible by IMPORT statements in the interface body.  
 5 The accessed entities are known by the same name and have the same attributes as in the host; they  
 6 are named data objects, derived types, abstract interfaces, procedures, generic identifiers (12.3.2.1), and  
 7 namelist groups.

8 If an entity that is accessed by use association has the same nongeneric name as a host entity, the host  
 9 entity is inaccessible by that name. Within the scoping unit, a name that is declared to be an external  
 10 procedure name by an *external-stmt*, *procedure-declaration-stmt*, or *interface-body*, or that appears as a  
 11 *module-name* in a *use-stmt* is a global identifier; any entity of the host that has this as its nongeneric  
 12 name is inaccessible by that name. A name that appears in the scoping unit as

- 13 (1) A *function-name* in a *stmt-function-stmt* or in an *entity-decl* in a *type-declaration-stmt*;
- 14 (2) An *object-name* in an *entity-decl* in a *type-declaration-stmt*, in a *pointer-stmt*, in a *save-stmt*,  
 15 in an *allocatable-stmt*, or in a *target-stmt*;
- 16 (3) A *type-param-name* in a *derived-type-stmt*;
- 17 (4) A *named-constant* in a *named-constant-def* in a *parameter-stmt*;
- 18 (5) An *array-name* in a *dimension-stmt*;
- 19 (6) A *variable-name* in a *common-block-object* in a *common-stmt*;
- 20 (7) A *proc-pointer-name* in a *common-block-object* in a *common-stmt*;
- 21 (8) The name of a variable that is wholly or partially initialized in a *data-stmt*;
- 22 (9) The name of an object that is wholly or partially equivalenced in an *equivalence-stmt*;
- 23 (10) A *dummy-arg-name* in a *function-stmt*, in a *subroutine-stmt*, in an *entry-stmt*, or in a *stmt-*  
 24 *function-stmt*;
- 25 (11) A *result-name* in a *function-stmt* or in an *entry-stmt*;
- 26 (12) The name of an entity declared by an interface body;
- 27 (13) An *intrinsic-procedure-name* in an *intrinsic-stmt*;
- 28 (14) A *namelist-group-name* in a *namelist-stmt*;
- 29 (15) A *generic-name* in a *generic-spec* in an *interface-stmt*; or
- 30 (16) The name of a named construct

31 is a local identifier in the scoping unit and any entity of the host that has this as its nongeneric name is  
 32 inaccessible by that name by host association. If a scoping unit is the host of a derived-type definition  
 33 or a subprogram, the name of the derived type or of any procedure defined by the subprogram is a local  
 34 identifier in the scoping unit; any entity of the host that has this as its nongeneric name is inaccessible  
 35 by that name. Local identifiers of a subprogram are not accessible to its host.

**NOTE 16.8**

A name that appears in an ASYNCHRONOUS or VOLATILE statement is not necessarily the name of a local variable. If a variable that is accessible via host association other than by an IMPORT statement is specified in an ASYNCHRONOUS or VOLATILE statement, that host variable is given the ASYNCHRONOUS or VOLATILE attribute in the scope of the current internal or module procedure.

36 If a host entity is inaccessible only because a local variable with the same name is wholly or partially  
 37 initialized in a DATA statement, the local variable shall not be referenced or defined prior to the DATA  
 38 statement.

39 If a derived-type name of a host is inaccessible, data entities of that type or subobjects of such data

1 entities still can be accessible.

**NOTE 16.9**

An interface body accesses by host association only those entities made accessible by IMPORT statements.

2 If an external or dummy procedure with an implicit interface is accessed via host association, then it  
 3 shall have the EXTERNAL attribute in the host scoping unit; if it is invoked as a function in the inner  
 4 scoping unit, its type and type parameters shall be established in the host scoping unit. The type and  
 5 type parameters of a function with the EXTERNAL attribute are established in a scoping unit if that  
 6 scoping unit explicitly declares them, invokes the function, accesses the function from a module, or  
 7 accesses the function from its host where its type and type parameters are established.

8 If an intrinsic procedure is accessed via host association, then it shall be established to be intrinsic in the  
 9 host scoping unit. An intrinsic procedure is established to be intrinsic in a scoping unit if that scoping  
 10 unit explicitly gives it the INTRINSIC attribute, invokes it as an intrinsic procedure, accesses it from a  
 11 module, or accesses it from its host where it is established to be intrinsic.

**NOTE 16.10**

A host subprogram and an internal subprogram may contain the same and differing use-associated entities, as illustrated in the following example.

```

MODULE B; REAL BX, Q; INTEGER IX, JX; END MODULE B
MODULE C; REAL CX; END MODULE C
MODULE D; REAL DX, DY, DZ; END MODULE D
MODULE E; REAL EX, EY, EZ; END MODULE E
MODULE F; REAL FX; END MODULE F
MODULE G; USE F; REAL GX; END MODULE G
PROGRAM A
USE B; USE C; USE D
...
CONTAINS
  SUBROUTINE INNER_PROC (Q)
    USE C          ! Not needed
    USE B, ONLY: BX ! Entities accessible are BX, IX, and JX
                   ! if no other IX or JX
                   ! is accessible to INNER_PROC
                   ! Q is local to INNER_PROC,
                   ! because Q is a dummy argument
    USE D, X => DX ! Entities accessible are DX, DY, and DZ
                   ! X is local name for DX in INNER_PROC
                   ! X and DX denote same entity if no other
                   ! entity DX is local to INNER_PROC
    USE E, ONLY: EX ! EX is accessible in INNER_PROC, not in program A
                   ! EY and EZ are not accessible in INNER_PROC
                   ! or in program A
    USE G          ! FX and GX are accessible in INNER_PROC
    ...
  
```

**NOTE 16.10 (cont.)**

```

END SUBROUTINE INNER_PROC
END PROGRAM A

```

Because program A contains the statement

```

USE B

```

all of the entities in module B, except for Q, are accessible in INNER\_PROC, even though INNER\_PROC contains the statement

```

USE B, ONLY: BX

```

The USE statement with the ONLY keyword means that this particular statement brings in only the entity named, not that this is the only variable from the module accessible in this scoping unit.

**NOTE 16.11**

For more examples of host association, see section [C.11.1](#).

**1 16.4.1.4 Linkage association**

2 Linkage association occurs between a module variable that has the BIND attribute and the C variable  
3 with which it interoperates, or between a Fortran common block and the C variable with which it  
4 interoperates ([15.3](#)). Such association remains in effect throughout the execution of the program.

**5 16.4.1.5 Construct association**

6 Execution of a SELECT TYPE statement establishes an association between the selector and the asso-  
7 ciate name of the construct. Execution of an ASSOCIATE statement establishes an association between  
8 each selector and the corresponding associate name of the construct.

9 If the selector is allocatable, it shall be allocated; the associate name is associated with the data object  
10 and does not have the ALLOCATABLE attribute.

11 If the selector has the POINTER attribute, it shall be associated; the associate name is associated with  
12 the target of the pointer and does not have the POINTER attribute.

13 If the selector is a variable other than an array section having a vector subscript, the association is  
14 with the data object specified by the selector; otherwise, the association is with the value of the selector  
15 expression, which is evaluated prior to execution of the block.

16 Each associate name remains associated with the corresponding selector throughout the execution of the  
17 executed block. Within the block, each selector is known by and may be accessed by the corresponding  
18 associate name. Upon termination of the construct, the association is terminated.

**NOTE 16.12**

The association between the associate name and a data object is established prior to execution of the block and is not affected by subsequent changes to variables that were used in subscripts or substring ranges in the *selector*.

## 1 16.4.2 Pointer association

2 Pointer association between a pointer and a target allows the target to be referenced by a reference to the  
3 pointer. At different times during the execution of a program, a pointer may be undefined, associated  
4 with different targets, or be disassociated. If a pointer is associated with a target, the definition status  
5 of the pointer is either defined or undefined, depending on the definition status of the target. If the  
6 pointer has deferred type parameters or shape, their values are assumed from the target. If the pointer  
7 is polymorphic, its dynamic type is the dynamic type of the target.

### 8 16.4.2.1 Pointer association status

9 A pointer may have a **pointer association status** of associated, disassociated, or undefined. Its  
10 association status may change during execution of a program. Unless a pointer is initialized (explicitly  
11 or by default), it has an initial association status of undefined. A pointer may be initialized to have an  
12 association status of disassociated.

#### NOTE 16.13

A pointer from a module program unit may be accessible in a subprogram via use association. Such pointers have a lifetime that is greater than targets that are declared in the subprogram, unless such targets are saved. Therefore, if such a pointer is associated with a local target, there is the possibility that when a procedure defined by the subprogram completes execution, the target will cease to exist, leaving the pointer “dangling”. This standard considers such pointers to have an undefined association status. They are neither associated nor disassociated. They shall not be used again in the program until their status has been reestablished. There is no requirement on a processor to be able to detect when a pointer target ceases to exist.

#### 13 16.4.2.1.1 Events that cause pointers to become associated

14 A pointer becomes associated when

- 15 (1) The pointer is allocated (6.3.1) as the result of the successful execution of an ALLOCATE  
16 statement referencing the pointer, or
- 17 (2) The pointer is pointer-assigned to a target (7.4.2) that is associated or is specified with the  
18 TARGET attribute and, if allocatable, is allocated.

#### 19 16.4.2.1.2 Events that cause pointers to become disassociated

20 A pointer becomes disassociated when

- 21 (1) The pointer is nullified (6.3.2),
- 22 (2) The pointer is deallocated (6.3.3),
- 23 (3) The pointer is pointer-assigned (7.4.2) to a disassociated pointer, or
- 24 (4) The pointer is an ultimate component of an object of a type for which default initialization  
25 is specified for the component and
  - 26 (a) a procedure is invoked with this object as an actual argument corresponding to a  
27 nonpointer nonallocatable dummy argument with INTENT (OUT),
  - 28 (b) a procedure with this object as an unsaved nonpointer nonallocatable local object  
29 that is not accessed by use or host association is invoked, or
  - 30 (c) this object is allocated.

#### 31 16.4.2.1.3 Events that cause the association status of pointers to become undefined

32 The association status of a pointer becomes undefined when

- 1 (1) The pointer is pointer-assigned to a target that has an undefined association status,
- 2 (2) The target of the pointer is deallocated other than through the pointer,
- 3 (3) The allocation transfer procedure (13.7.82) is executed with the pointer associated with the  
4 argument FROM and an object without the target attribute associated with the argument  
5 TO.
- 6 (4) Execution of a RETURN or END statement causes the pointer's target to become undefined  
7 (item (3) of 16.5.6),
- 8 (5) A procedure is terminated by execution of a RETURN or END statement and the pointer  
9 is declared or accessed in the subprogram that defines the procedure unless the pointer
  - 10 (a) Has the SAVE attribute,
  - 11 (b) Is in blank common,
  - 12 (c) Is in a named common block that appears in at least one other scoping unit that is  
13 in execution,
  - 14 (d) Is in the scoping unit of a module if the module also is accessed by another scoping  
15 unit that is in execution,
  - 16 (e) Is accessed by host association, or
  - 17 (f) Is the return value of a function declared to have the POINTER attribute,
- 18 (6) The pointer is an ultimate component of an object, default initialization is not specified  
19 for the component, and a procedure is invoked with this object as an actual argument  
20 corresponding to a dummy argument with INTENT(OUT), or
- 21 (7) A procedure is invoked with the pointer as an actual argument corresponding to a pointer  
22 dummy argument with INTENT(OUT).

#### 23 16.4.2.1.4 Other events that change the association status of pointers

24 When a pointer becomes associated with another pointer by argument association, construct association,  
25 or host association, the effects on its association status are specified in 16.4.5.

26 While two pointers are name associated, storage associated, or inheritance associated, if the association  
27 status of one pointer changes, the association status of the other changes accordingly.

#### 28 16.4.2.2 Pointer definition status

29 The definition status of a pointer is that of its target. If a pointer is associated with a definable target,  
30 the definition status of the pointer may be defined or undefined according to the rules for a variable  
31 (16.5).

#### 32 16.4.2.3 Relationship between association status and definition status

33 If the association status of a pointer is disassociated or undefined, the pointer shall not be referenced  
34 or deallocated. Whatever its association status, a pointer always may be nullified, allocated, or pointer  
35 assigned. A nullified pointer is disassociated. When a pointer is allocated, it becomes associated but  
36 undefined. When a pointer is pointer assigned, its association and definition status become those of the  
37 specified *data-target* or *proc-target*.

#### 38 16.4.3 Storage association

39 Storage sequences are used to describe relationships that exist among variables, common blocks, and  
40 result variables. **Storage association** is the association of two or more data objects that occurs when  
41 two or more storage sequences share or are aligned with one or more storage units.

### 1 16.4.3.1 Storage sequence

2 A **storage sequence** is a sequence of storage units. The **size of a storage sequence** is the number  
 3 of storage units in the storage sequence. A **storage unit** is a character storage unit, a numeric storage  
 4 unit, a file storage unit (9.2.4), or an unspecified storage unit. The sizes of the numeric storage unit, the  
 5 character storage unit and the file storage unit are the value of constants in the ISO\_FORTRAN\_ENV  
 6 intrinsic module (13.8.2).

7 In a storage association context

- 8 (1) A nonpointer scalar object of type default integer, default real, or default logical occupies a  
 9 single **numeric storage unit**;
- 10 (2) A nonpointer scalar object of type double precision real or default complex occupies two  
 11 contiguous numeric storage units;
- 12 (3) A nonpointer scalar object of type default character and character length *len* occupies *len*  
 13 contiguous **character storage units**;
- 14 (4) A nonpointer scalar object of type character with the C character kind (15.1.1) and character  
 15 length *len* occupies *len* contiguous **unspecified storage units**.
- 16 (5) A nonpointer scalar object of sequence type with no type parameters occupies a sequence  
 17 of storage sequences corresponding to the sequence of its ultimate components;
- 18 (6) A nonpointer scalar object of any type not specified in items (1)-(5) occupies a single  
 19 unspecified storage unit that is different for each case and each set of type parameter values,  
 20 and that is different from the unspecified storage units of item (4);
- 21 (7) A nonpointer array occupies a sequence of contiguous storage sequences, one for each array  
 22 element, in array element order (6.2.2.2); and
- 23 (8) A pointer occupies a single unspecified storage unit that is different from that of any non-  
 24 pointer object and is different for each combination of type, type parameters, and rank.

25 A sequence of storage sequences forms a storage sequence. The order of the storage units in such a  
 26 composite storage sequence is that of the individual storage units in each of the constituent storage  
 27 sequences taken in succession, ignoring any zero-sized constituent sequences.

28 Each common block has a storage sequence (5.5.2.1).

### 29 16.4.3.2 Association of storage sequences

30 Two nonzero-sized storage sequences  $s_1$  and  $s_2$  are **storage associated** if the  $i$ th storage unit of  $s_1$  is  
 31 the same as the  $j$ th storage unit of  $s_2$ . This causes the  $(i + k)$ th storage unit of  $s_1$  to be the same as  
 32 the  $(j + k)$ th storage unit of  $s_2$ , for each integer  $k$  such that  $1 \leq i + k \leq \text{size of } s_1$  and  $1 \leq j + k \leq$   
 33  $\text{size of } s_2$ .

34 Storage association also is defined between two zero-sized storage sequences, and between a zero-sized  
 35 storage sequence and a storage unit. A zero-sized storage sequence in a sequence of storage sequences is  
 36 storage associated with its successor, if any. If the successor is another zero-sized storage sequence, the  
 37 two sequences are storage associated. If the successor is a nonzero-sized storage sequence, the zero-sized  
 38 sequence is storage associated with the first storage unit of the successor. Two storage units that are  
 39 each storage associated with the same zero-sized storage sequence are the same storage unit.

#### NOTE 16.14

Zero-sized objects may occur in a storage association context as the result of changing a parameter.  
 For example, a program might contain the following declarations:

```
INTEGER, PARAMETER :: PROBSIZE = 10
INTEGER, PARAMETER :: ARRAYSIZE = PROBSIZE * 100
```

**NOTE 16.14 (cont.)**

```

REAL, DIMENSION (ARRAYSIZE) :: X
INTEGER, DIMENSION (ARRAYSIZE) :: IX
...
COMMON / EXAMPLE / A, B, C, X, Y, Z
EQUIVALENCE (X, IX)
...

```

If the first statement is subsequently changed to assign zero to PROBSIZE, the program still will conform to the standard.

**1 16.4.3.3 Association of scalar data objects**

2 Two scalar data objects are storage associated if their storage sequences are storage associated. Two  
3 scalar entities are **totally associated** if they have the same storage sequence. Two scalar entities are  
4 **partially associated** if they are associated without being totally associated.

5 The definition status and value of a data object affects the definition status and value of any storage  
6 associated entity. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement  
7 may cause storage association of storage sequences.

8 An EQUIVALENCE statement causes storage association of data objects only within one scoping unit,  
9 unless one of the equivalenced entities is also in a common block (5.5.1.1 and 5.5.2.1).

10 COMMON statements cause data objects in one scoping unit to become storage associated with data  
11 objects in another scoping unit.

12 A common block is permitted to contain a sequence of differing storage units. All scoping units that  
13 access named common blocks with the same name shall specify an identical sequence of storage units.  
14 Blank common blocks may be declared with differing sizes in different scoping units. For any two blank  
15 common blocks, the initial sequence of storage units of the longer blank common block shall be identical  
16 to the sequence of storage units of the shorter common block. If two blank common blocks are the same  
17 length, they shall have the same sequence of storage units.

18 An ENTRY statement in a function subprogram causes storage association of the result variables.

19 Partial association may exist only between

- 20 (1) An object of default character or character sequence type and an object of default character  
21 or character sequence type or
- 22 (2) An object of default complex, double precision real, or numeric sequence type and an object  
23 of default integer, default real, default logical, double precision real, default complex, or  
24 numeric sequence type.

25 For noncharacter entities, partial association may occur only through the use of COMMON, EQUIV-  
26 ALENCE, or ENTRY statements. For character entities, partial association may occur only through  
27 argument association or the use of COMMON or EQUIVALENCE statements.

**NOTE 16.15**

In the example:

```

REAL A (4), B
COMPLEX C (2)

```

**NOTE 16.15 (cont.)**

```
DOUBLE PRECISION D
```

```
EQUIVALENCE (C (2), A (2), B), (A, D)
```

the third storage unit of C, the second storage unit of A, the storage unit of B, and the second storage unit of D are specified as the same. The storage sequences may be illustrated as:

```
Storage unit      1      2      3      4      5
                  ----C(1)----|---C(2)----
                   A(1)  A(2)  A(3)  A(4)
                   --B--
                   -----D-----
```

A (2) and B are totally associated. The following are partially associated: A (1) and C (1), A (2) and C (2), A (3) and C (2), B and C (2), A (1) and D, A (2) and D, B and D, C (1) and D, and C (2) and D. Although C (1) and C (2) are each storage associated with D, C (1) and C (2) are not storage associated with each other.

- 1 Partial association of character entities occurs when some, but not all, of the storage units of the entities
- 2 are the same.

**NOTE 16.16**

In the example:

```
CHARACTER A*4, B*4, C*3
```

```
EQUIVALENCE (A (2:3), B, C)
```

A, B, and C are partially associated.

- 3 A storage unit shall not be explicitly initialized more than once in a program. Explicit initialization
- 4 overrides default initialization, and default initialization for an object of derived type overrides default
- 5 initialization for a component of the object (4.5.1). Default initialization may be specified for a storage
- 6 unit that is storage associated provided the objects supplying the default initialization are of the same
- 7 type and type parameters, and supply the same value for the storage unit.

**8 16.4.4 Inheritance association**

- 9 Inheritance association occurs between components of the parent component and components inherited
- 10 by type extension into an extended type (4.5.6.1). This association is persistent; it is not affected by the
- 11 accessibility of the inherited components.

**12 16.4.5 Establishing associations**

- 13 When an association is established between two entities by argument association, host association,
- 14 or construct association, certain characteristics of the **associating entity** become those of the **pre-**
- 15 **existing** entity.

- 16 For argument association, the associating entity is the dummy argument and the pre-existing entity
- 17 is the actual argument. For host association, the associating entity is the entity in the host scoping
- 18 unit and the pre-existing entity is the entity in the contained scoping unit. If the host scoping unit
- 19 is a recursive procedure, the pre-existing entity that participates in the association is the one from the

1 innermost procedure instance that invoked, directly or indirectly, the contained procedure. For construct  
2 association, the associating entity is identified by the associate name and the pre-existing entity is the  
3 selector.

4 When an association is established by argument association, host association, or construct association,  
5 the following applies:

6 (1) If the associating entity has the POINTER attribute, its pointer association status becomes  
7 the same as that of the pre-existing entity. If the pre-existing entity has a pointer association  
8 status of associated, the associating entity becomes pointer associated with the same target  
9 and, if they are arrays, the bounds of the associating entity become the same as those of  
10 the pre-existing entity.

11 (2) If the associating entity has the ALLOCATABLE attribute, its allocation status becomes  
12 the same as that of the pre-existing entity. If the pre-existing entity is allocated, the bounds  
13 (if it is an array), values of deferred type parameters, definition status, and value (if it is  
14 defined) become the same as those of the pre-existing entity. If the associating entity is  
15 polymorphic and the pre-existing entity is allocated, the dynamic type of the associating  
16 entity becomes the same as that of the pre-existing entity.

17 If the associating entity is neither a pointer nor allocatable, its definition status and value (if it is defined)  
18 become the same as those of the pre-existing entity. If the entities are arrays and the association is not  
19 argument association, the bounds of the associating entity become the same as those of the pre-existing  
20 entity.

## 21 **16.5 Definition and undefinition of variables**

22 A variable may be defined or may be undefined and its definition status may change during execution of  
23 a program. An action that causes a variable to become undefined does not imply that the variable was  
24 previously defined. An action that causes a variable to become defined does not imply that the variable  
25 was previously undefined.

### 26 **16.5.1 Definition of objects and subobjects**

27 Arrays, including sections, and variables of derived, character, or complex type are objects that consist of  
28 zero or more subobjects. Associations may be established between variables and subobjects and between  
29 subobjects of different variables. These subobjects may become defined or undefined.

- 30 (1) An array is defined if and only if all of its elements are defined.  
31 (2) A derived-type scalar object is defined if and only if all of its nonpointer components are  
32 defined.  
33 (3) A complex or character scalar object is defined if and only if all of its subobjects are defined.  
34 (4) If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.

### 35 **16.5.2 Variables that are always defined**

36 Zero-sized arrays and zero-length strings are always defined.

### 37 **16.5.3 Variables that are initially defined**

38 The following variables are initially defined:

- 39 (1) Variables specified to have initial values by DATA statements,  
40 (2) Variables specified to have initial values by type declaration statements,

- 1 (3) Nonpointer default-initialized subcomponents of variables that do not have the ALLOCAT-  
 2 ABLE or POINTER attribute, and are either saved or are declared in a main program,  
 3 MODULE, or BLOCK DATA scoping unit,  
 4 (4) Variables that are always defined, and  
 5 (5) Variables with the BIND attribute that are initialized by means other than Fortran.

**NOTE 16.17**

Fortran code:

```
module mod
  integer, bind(c,name="blivet") :: foo
end module mod
```

C code:

```
int blivet = 123;
```

In the above example, the Fortran variable foo is initially defined to have the value 123 by means other than Fortran.

#### 6 **16.5.4 Variables that are initially undefined**

7 All other variables are initially undefined.

#### 8 **16.5.5 Events that cause variables to become defined**

9 Variables become defined as follows:

- 10 (1) Execution of an intrinsic assignment statement other than a masked array assignment or  
 11 FORALL assignment statement causes the variable that precedes the equals to become  
 12 defined. Execution of a defined assignment statement may cause all or part of the variable  
 13 that precedes the equals to become defined.
- 14 (2) Execution of a masked array assignment or FORALL assignment statement may cause some  
 15 or all of the array elements in the assignment statement to become defined (7.4.3).
- 16 (3) As execution of an input statement proceeds, each variable that is assigned a value from  
 17 the input file becomes defined at the time that data is transferred to it. (See (4) in 16.5.6.)  
 18 Execution of a WRITE statement whose unit specifier identifies an internal file causes each  
 19 record that is written to become defined.
- 20 (4) Execution of a DO statement causes the DO variable, if any, to become defined.
- 21 (5) Beginning of execution of the action specified by an *io-implied-do* in a synchronous in-  
 22 put/output statement causes the *do-variable* to become defined.
- 23 (6) A reference to a procedure causes the entire dummy argument data object to become defined  
 24 if the dummy argument does not have INTENT(OUT) and the entire corresponding actual  
 25 argument is defined.  
 26 A reference to a procedure causes a subobject of a dummy argument to become defined if  
 27 the dummy argument does not have INTENT(OUT) and the corresponding subobject of  
 28 the corresponding actual argument is defined.
- 29 (7) Execution of an input/output statement containing an IOSTAT= specifier causes the spec-  
 30 ified integer variable to become defined.
- 31 (8) Execution of a synchronous READ statement containing a SIZE= specifier causes the spec-  
 32 ified integer variable to become defined.

- 1 (9) Execution of a wait operation corresponding to an asynchronous input statement containing  
2 a SIZE= specifier causes the specified integer variable to become defined.
- 3 (10) Execution of an INQUIRE statement causes any variable that is assigned a value during the  
4 execution of the statement to become defined if no error condition exists.
- 5 (11) If an error, end-of-file, or end-of-record condition occurs during execution of an input/output  
6 statement that has an IOMSG= specifier, the *iomsg-variable* becomes defined.
- 7 (12) When a character storage unit becomes defined, all associated character storage units be-  
8 come defined.
- 9 When a numeric storage unit becomes defined, all associated numeric storage units of the  
10 same type become defined. When an entity of double precision real type becomes defined,  
11 all totally associated entities of double precision real type become defined.
- 12 When an unspecified storage unit becomes defined, all associated unspecified storage units  
13 become defined.
- 14 (13) When a default complex entity becomes defined, all partially associated default real entities  
15 become defined.
- 16 (14) When both parts of a default complex entity become defined as a result of partially associ-  
17 ated default real or default complex entities becoming defined, the default complex entity  
18 becomes defined.
- 19 (15) When all components of a structure of a numeric sequence type or character sequence type  
20 become defined as a result of partially associated objects becoming defined, the structure  
21 becomes defined.
- 22 (16) Execution of an ALLOCATE or DEALLOCATE statement with a STAT= specifier causes  
23 the variable specified by the STAT= specifier to become defined.
- 24 (17) If an error condition occurs during execution of an ALLOCATE or DEALLOCATE state-  
25 ment that has an ERRMSG= specifier, the *errmsg-variable* becomes defined.
- 26 (18) Allocation of a zero-sized array causes the array to become defined.
- 27 (19) Allocation of an object that has a nonpointer default-initialized subcomponent causes that  
28 subcomponent to become defined.
- 29 (20) Invocation of a procedure causes any automatic object of zero size in that procedure to  
30 become defined.
- 31 (21) Execution of a pointer assignment statement that associates a pointer with a target that is  
32 defined causes the pointer to become defined.
- 33 (22) Invocation of a procedure that contains an unsaved nonpointer nonallocatable local variable  
34 causes all nonpointer default-initialized subcomponents of the object to become defined.
- 35 (23) Invocation of a procedure that has a nonpointer nonallocatable INTENT (OUT) dummy  
36 argument causes all nonpointer default-initialized subcomponents of the dummy argument  
37 to become defined.
- 38 (24) Invocation of a nonpointer function of a derived type causes all nonpointer default-initialized  
39 subcomponents of the function result to become defined.
- 40 (25) In a FORALL construct, the *index-name* becomes defined when the *index-name* value set  
41 is evaluated.
- 42 (26) An object with the VOLATILE attribute that is changed by a means listed in 5.1.2.16  
43 becomes defined.

#### 44 16.5.6 Events that cause variables to become undefined

45 Variables become undefined as follows:

- 46 (1) When a variable of a given type becomes defined, all associated variables of different type  
47 become undefined. However, when a variable of type default real is partially associated with  
48 a variable of type default complex, the complex variable does not become undefined when

- 1           the real variable becomes defined and the real variable does not become undefined when  
 2           the complex variable becomes defined. When a variable of type default complex is partially  
 3           associated with another variable of type default complex, definition of one does not cause  
 4           the other to become undefined.
- 5           (2) If the evaluation of a function may cause a variable to become defined and if a reference to  
 6           the function appears in an expression in which the value of the function is not needed to  
 7           determine the value of the expression, the variable becomes undefined when the expression  
 8           is evaluated.
- 9           (3) When execution of an instance of a subprogram completes,  
 10           (a) its unsaved local variables become undefined,  
 11           (b) unsaved variables in a named common block that appears in the subprogram become  
 12           undefined if they have been defined or redefined, unless another active scoping unit  
 13           is referencing the common block,  
 14           (c) unsaved nonfinalizable local variables of a module become undefined unless another  
 15           active scoping unit is referencing the module, and

**NOTE 16.18**

A module subprogram inherently references the module that is its host. Therefore, for processors that keep track of when modules are in use, a module is in use whenever any procedure in the module is active, even if no other active scoping units reference the module; this situation can arise if a module procedure is invoked via a procedure pointer, a type-bound procedure, or a companion processor.

- 16           (d) unsaved finalizable local variables of a module may be finalized if no other active  
 17           scoping unit is referencing the module – following which they become undefined.
- 18           (4) When an error condition or end-of-file condition occurs during execution of an input state-  
 19           ment, all of the variables specified by the input list or namelist group of the statement  
 20           become undefined.
- 21           (5) When an error condition, end-of-file condition, or end-of-record condition occurs during  
 22           execution of an input/output statement and the statement contains any *io-implied-dos*, all  
 23           of the *do-variables* in the statement become undefined (9.10).
- 24           (6) Execution of a direct access input statement that specifies a record that has not been written  
 25           previously causes all of the variables specified by the input list of the statement to become  
 26           undefined.
- 27           (7) Execution of an INQUIRE statement may cause the NAME=, RECL=, and NEXTREC=  
 28           variables to become undefined (9.9).
- 29           (8) When a character storage unit becomes undefined, all associated character storage units  
 30           become undefined.  
 31           When a numeric storage unit becomes undefined, all associated numeric storage units be-  
 32           come undefined unless the undefinition is a result of defining an associated numeric storage  
 33           unit of different type (see (1) above).  
 34           When an entity of double precision real type becomes undefined, all totally associated  
 35           entities of double precision real type become undefined.  
 36           When an unspecified storage unit becomes undefined, all associated unspecified storage units  
 37           become undefined.
- 38           (9) When an allocatable entity is deallocated, it becomes undefined.
- 39           (10) When the allocation transfer procedure (13.5.16) causes the allocation status of an allocat-  
 40           able entity to become unallocated, the entity becomes undefined.
- 41           (11) Successful execution of an ALLOCATE statement for a nonzero-sized object that has a sub-  
 42           component for which default initialization has not been specified causes the subcomponent  
 43           to become undefined.

- 1 (12) Execution of an INQUIRE statement causes all inquiry specifier variables to become un-  
 2 defined if an error condition exists, except for any variables in an IOSTAT= or IOMSG=  
 3 specifier.
- 4 (13) When a procedure is invoked
- 5 (a) An optional dummy argument that is not associated with an actual argument becomes  
 6 undefined;
- 7 (b) A dummy argument with INTENT (OUT) becomes undefined except for any non-  
 8 pointer default-initialized subcomponents of the argument;
- 9 (c) An actual argument associated with a dummy argument with INTENT (OUT) be-  
 10 comes undefined except for any nonpointer default-initialized subcomponents of the  
 11 argument;
- 12 (d) A subobject of a dummy argument that does not have INTENT (OUT) becomes  
 13 undefined if the corresponding subobject of the actual argument is undefined; and
- 14 (e) The result variable of a function becomes undefined except for any of its nonpointer  
 15 default-initialized subcomponents.
- 16 (14) When the association status of a pointer becomes undefined or disassociated (16.4.2.1.2-  
 17 16.4.2.1.3), the pointer becomes undefined.
- 18 (15) When the execution of a FORALL construct completes, the *index-names* become undefined.
- 19 (16) Execution of an asynchronous READ statement causes all of the variables specified by the  
 20 input list or SIZE= specifier to become undefined. Execution of an asynchronous namelist  
 21 READ statement causes any variable in the namelist group to become undefined if that  
 22 variable will subsequently be defined during the execution of the READ statement or the  
 23 corresponding WAIT operation.
- 24 (17) When execution of a RETURN or END statement causes a variable to become undefined,  
 25 any variable of type C\_PTR becomes undefined if its value is the C address of any part of  
 26 the variable that becomes undefined.
- 27 (18) When a variable with the TARGET attribute is deallocated, any variable of type C\_PTR  
 28 becomes undefined if its value is the C address of any part of the variable that is deallocated.

**NOTE 16.19**

Execution of a defined assignment statement may leave all or part of the variable that precedes the equals undefined.

29 **16.5.7 Variable definition context**

30 Some variables are prohibited from appearing in a syntactic context that would imply definition or un-  
 31 definition of the variable (5.1.2.7, 5.1.2.12, 12.6). The following are the contexts in which the appearance  
 32 of a variable implies such definition or undefinition of the variable:

- 33 (1) The *variable* of an *assignment-stmt*,
- 34 (2) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,
- 35 (3) A *do-variable* in a *do-stmt* or *io-IMPLIED-DO*,
- 36 (4) An *input-item* in a *read-stmt*,
- 37 (5) A *variable-name* in a *namelist-stmt* if the *namelist-group-name* appears in a NML= specifier  
 38 in a *read-stmt*,
- 39 (6) An *internal-file-variable* in a *write-stmt*,
- 40 (7) An IOSTAT=, SIZE=, or IOMSG= specifier in an input/output statement,
- 41 (8) A definable variable in an INQUIRE statement,
- 42 (9) A *stat-variable*, *allocate-object*, or *errmsg-variable* in an *allocate-stmt* or a *deallocate-stmt*,
- 43 (10) An actual argument in a reference to a procedure with an explicit interface if the associated

- 1 dummy argument has the INTENT(OUT) or INTENT(INOUT) attribute, or  
2 (11) A *variable* that is the *selector* in a SELECT TYPE or ASSOCIATE construct if the associate  
3 name of that construct appears in a variable definition context.

# Annex A

(Informative)

## Glossary of technical terms

The following is a list of the principal technical terms used in the standard and their definitions. A reference in parentheses immediately after a term is to the section where the term is defined or explained. The wording of a definition here is not necessarily the same as in the standard.

**abstract type** (4.5.6) : A type that has the ABSTRACT attribute. A nonpolymorphic object shall not be declared to be of abstract type. A polymorphic object shall not be constructed or allocated to have a dynamic abstract type.

**action statement** (2.1) : A single statement specifying or controlling a computational action (R214).

**actual argument** (12, 12.4.1) : An expression, a variable, a procedure, or an alternate return specifier that is specified in a procedure reference.

**allocatable variable** (5.1.2.2) : A variable having the ALLOCATABLE attribute. It may be referenced or defined only when it is allocated. If it is an array, it has a shape only when it is allocated. It may be a named variable or a structure component.

**argument** (12) : An actual argument or a dummy argument.

**argument association** (16.4.1.1) : The relationship between an actual argument and a dummy argument during the execution of a procedure reference.

**array** (2.4.5) : A set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern. It may be a named array, an array section, a structure component, a function value, or an expression. Its rank is at least one. Note that in FORTRAN 77, arrays were always named and never constants.

**array element** (2.4.5, 6.2.2) : One of the scalar data that make up an array that is either named or is a structure component.

**array pointer** (5.1.2.5.3) : A pointer to an array.

**array section** (2.4.5, 6.2.2.3) : A subobject that is an array and is not a structure component.

**assignment statement** (7.4.1.1) : A statement of the form “variable = expression”.

**associate name** (8.1.4.1) : The name by which a selector of a SELECT TYPE or ASSOCIATE construct is known within the construct.

**association** (16.4) : Name association, pointer association, storage association, or inheritance association.

**assumed-shape array** (5.1.2.5.2) : A nonpointer dummy array that takes its shape from the associated actual argument.

**assumed-size array** (5.1.2.5.4) : A dummy array whose size is assumed from the associated actual argument. Its last upper bound is specified by an asterisk.

- 1 **attribute** (5) : A property of a data object that may be specified in a type declaration statement  
2 (R501).
- 3 **automatic data object** (5.1) : A data object that is a local entity of a subprogram, that is not a dummy  
4 argument, and that has a length type parameter or array bound that is specified by an expression that  
5 is not an initialization expression.
- 6 **base type** (4.5.6) : An extensible type that is not an extension of another type.
- 7 **belong** (8.1.6.4.3, 8.1.6.4.4) : If an EXIT or a CYCLE statement contains a construct name, the  
8 statement **belongs** to the DO construct using that name. Otherwise, it **belongs** to the innermost DO  
9 construct in which it appears.
- 10 **binding label** (15.4.1, 15.3.1) : A value of type default character that uniquely identifies how a variable,  
11 common block, subroutine, or function is known to a companion processor.
- 12 **block** (8.1) : A sequence of executable constructs embedded in another executable construct, bounded  
13 by statements that are particular to the construct, and treated as an integral unit.
- 14 **block data program unit** (11.3) : A program unit that provides initial values for data objects in  
15 named common blocks.
- 16 **bounds** (5.1.2.5.1) : For a named array, the limits within which the values of the subscripts of its array  
17 elements shall lie.
- 18 **character** (3.1) : A letter, digit, or other symbol.
- 19 **class** (5.1.1.2) : A class named N is the set of types extended from the type named N.
- 20 **characteristics** (12.2) :
- 21 (1) Of a procedure, its classification as a function or subroutine, whether it is pure, whether  
22 it is elemental, whether it has the BIND attribute, the value of its binding label, the char-  
23 acteristics of its dummy arguments, and the characteristics of its function result if it is a  
24 function.
- 25 (2) Of a dummy argument, whether it is a data object, is a procedure, is a procedure pointer,  
26 is an asterisk (alternate return indicator), or has the OPTIONAL attribute.
- 27 (3) Of a dummy data object, its type, type parameters, shape, the exact dependence of an  
28 array bound or type parameter on other entities, intent, whether it is optional, whether  
29 it is a pointer or a target, whether it is allocatable, whether it has the VALUE, ASYN-  
30 CHRONOUS, or VOLATILE attributes, whether it is polymorphic, and whether the shape,  
31 size, or a type parameter is assumed.
- 32 (4) Of a dummy procedure or procedure pointer, whether the interface is explicit, the charac-  
33 teristics of the procedure if the interface is explicit, and whether it is optional.
- 34 (5) Of a function result, its type, type parameters, which type parameters are deferred, whether  
35 it is polymorphic, whether it is a pointer or allocatable, whether it is a procedure pointer,  
36 rank if it is a pointer or allocatable, shape if it is not a pointer or allocatable, the exact  
37 dependence of an array bound or type parameter on other entities, and whether the character  
38 length is assumed.
- 39 **character length parameter** (2.4.1.1) : The type parameter that specifies the number of characters  
40 for an entity of type character.
- 41 **character string** (4.4.4) : A sequence of characters numbered from left to right 1, 2, 3, ...
- 42 **character storage unit** (16.4.3.1) : The unit of storage for holding a scalar that is not a pointer and  
43 is of type default character and character length one.

- 1 **collating sequence** (4.4.4.3) : An ordering of all the different characters of a particular kind type  
2 parameter.
- 3 **common block** (5.5.2) : A block of physical storage that may be accessed by any of the scoping units  
4 in a program.
- 5 **companion processor** (2.5.10): A mechanism by which global data and procedures may be referenced  
6 or defined. It may be a mechanism that references and defines such entities by means other than Fortran.  
7 The procedures can be described by a C function prototype.
- 8 **component** (4.5) : A constituent of a derived type.
- 9 **component order** (4.5.3.5) : The ordering of the components of a derived type that is used for intrinsic  
10 formatted input/output and for structure constructors.
- 11 **conformable** (2.4.5) : Two arrays are said to be **conformable** if they have the same shape. A scalar  
12 is conformable with any array.
- 13 **conformance** (1.5) : A program conforms to the standard if it uses only those forms and relationships  
14 described therein and if the program has an interpretation according to the standard. A program unit  
15 conforms to the standard if it can be included in a program in a manner that allows the program to be  
16 standard conforming. A processor conforms to the standard if it executes standard-conforming programs  
17 in a manner that fulfills the interpretations prescribed in the standard and contains the capability of  
18 detection and reporting as listed in 1.5.
- 19 **connected** (9.4.3) :
- 20       (1) For an external unit, the property of referring to an external file.  
21       (2) For an external file, the property of having an external unit that refers to it.
- 22 **constant** (2.4.3.1.2) : A data object whose value shall not change during execution of a program. It  
23 may be a named constant or a literal constant.
- 24 **construct** (7.4.3, 7.4.4, 8.1) : A sequence of statements starting with an ASSOCIATE, DO, FORALL,  
25 IF, SELECT CASE, SELECT TYPE, or WHERE statement and ending with the corresponding terminal  
26 statement.
- 27 **construct entity** (16) : An entity defined by a lexical token whose scope is a construct.
- 28 **control mask** (7.4.3) : In a WHERE statement or construct, an array of type logical whose value  
29 determines which elements of an array, in a *where-assignment-stmt*, will be defined.
- 30 **data** : Plural of datum.
- 31 **data entity** (2.4.3) : A data object, the result of the evaluation of an expression, or the result of the  
32 execution of a function reference (called the function result). A data entity has a type (either intrinsic  
33 or derived) and has, or may have, a data value (the exception is an undefined variable). Every data  
34 entity has a rank and is thus either a scalar or an array.
- 35 **data object** (2.4.3.1) : A data entity that is a constant, a variable, or a subobject of a constant.
- 36 **data type** (4) : See type.
- 37 **datum** : A single quantity that may have any of the set of values specified for its type.
- 38 **decimal symbol** (9.9.1.6, 10.5, 10.7.8) : The character that separates the whole and fractional parts in  
39 the decimal representation of a real number in a file. By default the decimal symbol is a decimal point  
40 (also known as a period). The current decimal symbol is determined by the current decimal edit mode.

- 1 **declared type** (5.1.1.2, 7.1.4) : The type that a data entity is declared to have. May differ from the  
2 type during execution (the dynamic type) for polymorphic data entities.
- 3 **default initialization** (4.5) : If initialization is specified in a type definition, an object of the type will  
4 be automatically initialized. Nonpointer components may be initialized with values by default; pointer  
5 components may be initially disassociated by default. Default initialization is not provided for objects  
6 of intrinsic type.
- 7 **default-initialized** (4.5.3.4) : A subcomponent is said to be default-initialized if it will be initialized  
8 by default initialization.
- 9 **deferred binding** (4.5.4) : A binding with the DEFERRED attribute. A deferred binding shall appear  
10 only in an abstract type definition (4.5.6).
- 11 **deferred type parameter** (4.3) : A length type parameter whose value is not specified in the decla-  
12 ration of an object, but instead is specified when the object is allocated or pointer-assigned.
- 13 **definable** (2.5.5) : A variable is **definable** if its value may be changed by the appearance of its  
14 designator on the left of an assignment statement. An allocatable variable that has not been allocated  
15 is an example of a data object that is not definable. An example of a subobject that is not definable is  
16 C (I) when C is an array that is a constant and I is an integer variable.
- 17 **defined** (2.5.5) : For a data object, the property of having or being given a valid value.
- 18 **defined assignment statement** (7.4.1.4, 12.3.2.1.2) : An assignment statement that is not an intrinsic  
19 assignment statement; it is defined by a subroutine and a generic interface that specifies ASSIGNMENT  
20 (=).
- 21 **defined operation** (7.1.3, 12.3.2.1.1) : An operation that is not an intrinsic operation and is defined  
22 by a function that is associated with a generic identifier.
- 23 **deleted feature** (1.8) : A feature in a previous Fortran standard that is considered to have been  
24 redundant and largely unused. See B.1 for a list of features that are in a previous Fortran standard, but  
25 are not in this standard. A feature designated as an obsolescent feature in the standard may become a  
26 deleted feature in the next revision.
- 27 **derived type** (2.4.1.2, 4.5) : A type whose data have components, each of which is either of intrinsic  
28 type or of another derived type.
- 29 **designator** (2.5.1) : A name, followed by zero or more component selectors, array section selectors,  
30 array element selectors, and substring selectors.
- 31 **disassociated** (2.4.6) : A disassociated pointer is not associated with a target. A pointer is disassociated  
32 following execution of a NULLIFY statement, following pointer assignment with a disassociated pointer,  
33 by default initialization, or by explicit initialization. A data pointer may also be disassociated by  
34 execution of a DEALLOCATE statement.
- 35 **dummy argument** (12, 12.5.2.1, 12.5.2.2, 12.5.2.4, 12.5.4) : An entity by which an associated actual  
36 argument is accessed during execution of a procedure.
- 37 **dummy array** : A dummy argument that is an array.
- 38 **dummy data object** (12.2.1.1, 12.4.1.2) : A dummy argument that is a data object.
- 39 **dummy pointer** : A dummy argument that is a pointer.
- 40 **dummy procedure** (12.1.2.3) : A dummy argument that is specified or referenced as a procedure.

- 1 **dynamic type** (5.1.1.2, 7.1.4) : The type of a data entity during execution of a program. The dynamic  
2 type of a data entity that is not polymorphic is the same as its declared type.
- 3 **effective item** (9.5.2) : A scalar object resulting from expanding an input/output list according to the  
4 rules in 9.5.2.
- 5 **elemental** (2.4.5, 7.4.1.4, 12.7) : An adjective applied to an operation, procedure, or assignment state-  
6 ment that is applied independently to elements of an array or corresponding elements of a set of con-  
7 formable arrays and scalars.
- 8 **entity** : The term used for any of the following: a program unit, procedure, abstract interface, operator,  
9 generic interface, common block, external unit, statement function, type, data entity, statement label,  
10 construct, or namelist group.
- 11 **executable construct** (2.1) : An action statement (R214) or an ASSOCIATE, CASE, DO, FORALL,  
12 IF, SELECT TYPE, or WHERE construct.
- 13 **executable statement** (2.3.1) : An instruction to perform or control one or more computational  
14 actions.
- 15 **explicit initialization** (5.1) : Explicit initialization may be specified for objects of intrinsic or derived  
16 type in type declaration statements or DATA statements. An object of a derived type that specifies  
17 default initialization shall not appear in a DATA statement.
- 18 **explicit interface** (12.3.1) : If a procedure has explicit interface at the point of a reference to it, the  
19 processor is able to verify that the characteristics of the reference and declaration are related as required  
20 by this standard. A procedure has explicit interface if it is an internal procedure, a module procedure,  
21 an intrinsic procedure, an external procedure that has an interface body, a procedure reference in its  
22 own scoping unit, or a dummy procedure that has an interface body.
- 23 **explicit-shape array** (5.1.2.5.1) : A named array that is declared with explicit bounds.
- 24 **expression** (2.4.3.2, 7.1) : A sequence of operands, operators, and parentheses (R722). It may be a  
25 variable, a constant, a function reference, or may represent a computation.
- 26 **extended type** (4.5.6) : An extensible type that is an extension of another type. A type that is declared  
27 with the EXTENDS attribute.
- 28 **extensible type** (4.5.6) : A type from which new types may be derived using the EXTENDS attribute.  
29 A nonsequence type that does not have the BIND attribute.
- 30 **extension type** (4.5.6) : A base type is an extension type of itself only. An extended type is an  
31 extension type of itself and of all types for which its parent type is an extension.
- 32 **extent** (2.4.5) : The size of one dimension of an array.
- 33 **external file** (9.2) : A sequence of records that exists in a medium external to the program.
- 34 **external linkage** : The characteristic describing that a C entity is global to the program; defined in  
35 clause 6.2.2 of the C standard.
- 36 **external procedure** (2.2.3.1) : A procedure that is defined by an external subprogram or by a means  
37 other than Fortran.
- 38 **external subprogram** (2.2) : A subprogram that is not in a main program, module, or another  
39 subprogram. Note that a module is not called a subprogram. Note that in FORTRAN 77, a block data  
40 program unit is called a subprogram.

- 1 **external unit** (9.4) : A mechanism that is used to refer to an external file. It is identified by a  
2 nonnegative integer.
- 3 **file** (9) : An internal file or an external file.
- 4 **file storage unit** (9.2.4) : The unit of storage for an unformatted or stream file.
- 5 **final subroutine** (4.5.5) : A subroutine that is called automatically by the processor during finalization.
- 6 **finalizable** (4.5.5) : A type that has final subroutines, or that has a finalizable component. An object  
7 of finalizable type.
- 8 **finalization** (4.5.5.1) : The process of calling user-defined final subroutines immediately before destroy-  
9 ing an object.
- 10 **function** (2.2.3) : A procedure that is invoked in an expression and computes a value which is then  
11 used in evaluating the expression.
- 12 **function result** (12.5.2.1) : The data object that returns the value of a function.
- 13 **function subprogram** (12.5.2.1) : A sequence of statements beginning with a FUNCTION statement  
14 that is not in an interface block and ending with the corresponding END statement.
- 15 **generic identifier** (12.3.2.1) : A lexical token that appears in an INTERFACE statement and is  
16 associated with all the procedures in the interface block.
- 17 **generic interface** (4.5.1, 12.3.2.1) : An interface specified by a generic procedure binding or a generic  
18 interface block.
- 19 **generic interface block** (12.3.2.1) : An interface block with a generic specification.
- 20 **global entity** (16.1) : An entity with an identifier whose scope is a program.
- 21 **host** (2.2) : Host scoping unit.
- 22 **host association** (16.4.1.3) : The process by which a contained scoping unit accesses entities of its  
23 host.
- 24 **host scoping unit** (2.2) : A scoping unit that immediately surrounds another scoping unit.
- 25 **implicit interface** (12.3.1) : For a procedure referenced in a scoping unit, the property of not having  
26 an explicit interface. A statement function always has an implicit interface
- 27 **inherit** (4.5.6) : To acquire from a parent. Type parameters, components, or procedure bindings of an  
28 extended type that are automatically acquired from its parent type without explicit declaration in the  
29 extended type are said to be inherited.
- 30 **inheritance association** (4.5.6.1, 16.4.4) : The relationship between the inherited components and the  
31 parent component in an extended type.
- 32 **inquiry function** (13.1) : A function that is either intrinsic or is defined in an intrinsic module and  
33 whose result depends on properties of one or more of its arguments instead of their values.
- 34 **instance of a subprogram** (12.5.2.3) : The copy of a subprogram that is created when a procedure  
35 defined by the subprogram is invoked.
- 36 **intent** (5.1.2.7) : An attribute of a dummy data object that indicates whether it is used to transfer data  
37 into the procedure, out of the procedure, or both.

- 1 **interface block** (12.3.2.1) : A sequence of statements from an INTERFACE statement to the corre-  
2 sponding END INTERFACE statement.
- 3 **interface body** (12.3.2.1) : A sequence of statements in an interface block from a FUNCTION or  
4 SUBROUTINE statement to the corresponding END statement.
- 5 **interface of a procedure** (12.3) : See procedure interface.
- 6 **internal file** (9.3) : A character variable that is used to transfer and convert data from internal storage  
7 to internal storage.
- 8 **internal procedure** (2.2.3.3) : A procedure that is defined by an internal subprogram.
- 9 **internal subprogram** (2.2) : A subprogram in a main program or another subprogram.
- 10 **interoperable** (15.2) : The property of a Fortran entity that ensures that an equivalent entity may be  
11 defined by means of C.
- 12 **intrinsic** (2.5.7) : An adjective that may be applied to types, operators, assignment statements, proce-  
13 dures, and modules. Intrinsic types, operators, and assignment statements are defined in this standard  
14 and may be used in any scoping unit without further definition or specification. Intrinsic procedures are  
15 defined in this standard or provided by a processor, and may be used in a scoping unit without further  
16 definition or specification. Intrinsic modules are defined in this standard or provided by a processor,  
17 and may be accessed by use association; procedures and types defined in an intrinsic module are not  
18 themselves intrinsic.
- 19 Intrinsic procedures and modules that are not defined in this standard are called nonstandard intrinsic  
20 procedures and modules.
- 21 **invoke** (2.2.3) :
- 22 (1) To call a subroutine by a CALL statement or by a defined assignment statement.  
23 (2) To call a function by a reference to it by name or operator during the evaluation of an  
24 expression.
- 25 **keyword** (2.5.2) : A word that is part of the syntax of a statement or a name that is used to identify  
26 an item in a list.
- 27 **kind type parameter** (2.4.1.1, 4.4.1, 4.4.2, 4.4.3, 4.4.4, 4.4.5) : A parameter whose values label the  
28 available kinds of an intrinsic type.
- 29 **label** : See statement label.
- 30 **length of a character string** (4.4.4) : The number of characters in the character string.
- 31 **lexical token** (3.2) : A sequence of one or more characters with a specified interpretation.
- 32 **line** (3.3) : A sequence of 0 to 132 characters, which may contain Fortran statements, a comment, or  
33 an INCLUDE line.
- 34 **linked** (12.5.3) : When a C function with external linkage has the same binding label as a Fortran  
35 procedure, they are said to be linked. It is also possible for two Fortran entities to be linked.
- 36 **literal constant** (2.4.3.1.2, 4.4) : A constant without a name. Note that in FORTRAN 77, this was  
37 called simply a constant.
- 38 **local entity** (16.2) : An entity identified by a lexical token whose scope is a scoping unit.

- 1 **local variable** (2.4.3.1.1) : A variable local to a particular scoping unit; not imported through use or  
2 host association, not a dummy argument, and not a variable in common.
- 3 **main program** (2.3.4, 11.1) : A Fortran main program or a replacement defined by means other than  
4 Fortran.
- 5 **many-one array section** (6.2.2.3.2) : An array section with a vector subscript having two or more  
6 elements with the same value.
- 7 **module** (2.2.4, 11.2) : A program unit that contains or accesses definitions to be accessed by other  
8 program units.
- 9 **module procedure** (2.2.3.2) : A procedure that is defined by a module subprogram.
- 10 **module subprogram** (2.2) : A subprogram that is in a module but is not an internal subprogram.
- 11 **name** (3.2.1) : A lexical token consisting of a letter followed by up to 62 alphanumeric characters  
12 (letters, digits, and underscores). Note that in FORTRAN 77, this was called a symbolic name.
- 13 **name association** (16.4.1) : Argument association, use association, or host association.
- 14 **named** : Having a name. That is, in a phrase such as “named variable,” the word “named” signifies that  
15 the variable name is not qualified by a subscript list, substring specification, and so on. For example,  
16 if X is an array variable, the reference “X” is a named variable while the reference “X(1)” is an object  
17 designator.
- 18 **named constant** (2.4.3.1.2) : A constant that has a name. Note that in FORTRAN 77, this was called  
19 a symbolic constant.
- 20 **NaN** (14.7) : A Not-a-Number value of IEEE arithmetic. It represents an undefined value or a value  
21 created by an invalid operation.
- 22 **nonexecutable statement** (2.3.1) : A statement used to configure the program environment in which  
23 computational actions take place.
- 24 **numeric storage unit** (16.4.3.1) : The unit of storage for holding a scalar that is not a pointer and is  
25 of type default real, default integer, or default logical.
- 26 **numeric type** (4.4) : Integer, real or complex type.
- 27 **object** (2.4.3.1) : Data object.
- 28 **object designator** (2.5.1) : A designator for a data object.
- 29 **obsolescent feature** (1.8) : A feature that is considered to have been redundant but that is still in  
30 frequent use.
- 31 **operand** (2.5.8) : An expression that precedes or succeeds an operator.
- 32 **operation** (7.1.2) : A computation involving one or two operands.
- 33 **operator** (2.5.8) : A lexical token that specifies an operation.
- 34 **override** (4.5.1, 4.5.6) : When explicit initialization or default initialization overrides default initializa-  
35 tion, it is as if only the overriding initialization were specified. If a procedure is bound to an extensible  
36 type, it overrides the one that would have been inherited from the parent type.
- 37 **parent component** (4.5.6.1) : The component of an entity of extended type that corresponds to its  
38 inherited portion.

- 1 **parent type** (4.5.6) : The extensible type from which an extended type is derived.
- 2 **passed-object dummy argument** (4.5.1) : The dummy argument of a type-bound procedure or  
3 procedure pointer component that becomes associated with the object through which the procedure was  
4 invoked.
- 5 **pointer** (2.4.6) : An entity that has the POINTER attribute.
- 6 **pointer assignment** (7.4.2) : The pointer association of a pointer with a target by the execution of a  
7 pointer assignment statement or the execution of an assignment statement for a data object of derived  
8 type having the pointer as a subobject.
- 9 **pointer assignment statement** (7.4.2) : A statement of the form “pointer-object => target”.
- 10 **pointer associated** (6.3, 7.4.2) : The relationship between a pointer and a target following a pointer  
11 assignment or a valid execution of an ALLOCATE statement.
- 12 **pointer association** (16.4.2) : The process by which a pointer becomes pointer associated with a target.
- 13 **polymorphic** (5.1.1.2) : Able to be of differing types during program execution. An object declared  
14 with the CLASS keyword is polymorphic.
- 15 **preconnected** (9.4.4) : A property describing a unit that is connected to an external file at the beginning  
16 of execution of a program. Such a unit may be specified in input/output statements without an OPEN  
17 statement being executed for that unit.
- 18 **procedure** (2.2.3, 12.1) : A computation that may be invoked during program execution. It may be a  
19 function or a subroutine. It may be an intrinsic procedure, an external procedure, a module procedure,  
20 an internal procedure, a dummy procedure, or a statement function. A subprogram may define more than  
21 one procedure if it contains ENTRY statements.
- 22 **procedure designator** (2.5.1) : A designator for a procedure.
- 23 **procedure interface** (12.3) : The characteristics of a procedure, the name of the procedure, the name  
24 of each dummy argument, and the generic identifiers (if any) by which it may be referenced.
- 25 **processor** (1.2) : The combination of a computing system and the mechanism by which programs are  
26 transformed for use on that computing system.
- 27 **processor dependent** (1.5) : The designation given to a facility that is not completely specified by  
28 this standard. Such a facility shall be provided by a processor, with methods or semantics determined  
29 by the processor.
- 30 **program** (2.2.1) : A set of program units that includes exactly one main program.
- 31 **program unit** (2.2) : The fundamental component of a program. A sequence of statements, comments,  
32 and INCLUDE lines. It may be a main program, a module, an external subprogram, or a block data  
33 program unit.
- 34 **prototype** : The C analog of a function interface body; defined in 6.7.5.3 of the C standard.
- 35 **pure procedure** (12.6) : A procedure that is a pure intrinsic procedure (13.1), is defined by a pure  
36 subprogram, or is a statement function that references only pure functions.
- 37 **rank** (2.4.4, 2.4.5) : The number of dimensions of an array. Zero for a scalar.
- 38 **record** (9.1) : A sequence of values or characters that is treated as a whole within a file.

- 1 **reference** (2.5.6) : The appearance of an object designator in a context requiring the value at that point  
2 during execution, the appearance of a procedure name, its operator symbol, or a defined assignment  
3 statement in a context requiring execution of the procedure at that point, or the appearance of a module  
4 name in a USE statement. Neither the act of defining a variable nor the appearance of the name of a  
5 procedure as an actual argument is regarded as a reference.
- 6 **result variable** (2.2.3, 12.5.2.1) : The variable that returns the value of a function.
- 7 **rounding mode** (14.3, 10.6.1.2.6) : The method used to choose the result of an operation that cannot  
8 be represented exactly. In IEEE arithmetic, there are four modes; nearest, towards zero, up (towards  
9  $\infty$ ), and down (towards  $-\infty$ ). In addition, for input/output the two additional modes COMPATIBLE  
10 and PROCESSOR\_DEFINED are provided.
- 11 **scalar** (2.4.4) :
- 12 (1) A single datum that is not an array.
  - 13 (2) Not having the property of being an array.
- 14 **scope** (16) : That part of a program within which a lexical token has a single interpretation. It may be  
15 a program, a scoping unit, a construct, a single statement, or a part of a statement.
- 16 **scoping unit** (2.2) : One of the following:
- 17 (1) A program unit or subprogram, excluding any scoping units in it,
  - 18 (2) A derived-type definition, or
  - 19 (3) An interface body, excluding any scoping units in it.
- 20 **section subscript** (6.2.2) : A subscript, vector subscript, or subscript triplet in an array section selector.
- 21 **selector** (6.1.1, 6.1.2, 6.1.3, 8.1.3, 8.1.4) : A syntactic mechanism for designating
- 22 (1) Part of a data object. It may designate a substring, an array element, an array section, or  
23 a structure component.
  - 24 (2) The set of values for which a CASE block is executed.
  - 25 (3) The object whose type determines which branch of a SELECT TYPE construct is executed.
  - 26 (4) The object that is associated with the *associate-name* in an ASSOCIATE construct.
- 27 **shape** (2.4.5) : The rank and extents of an array. The shape may be represented by the rank-one array  
28 whose elements are the extents in each dimension.
- 29 **size** (2.4.5) : The total number of elements of an array.
- 30 **specification expression** (7.1.6) : An expression with limitations that make it suitable for use in  
31 specifications such as length type parameters or array bounds.
- 32 **specification function** (7.1.6) : A nonintrinsic function that may be used in a specification expression.
- 33 **standard-conforming program** (1.5) : A program that uses only those forms and relationships de-  
34 scribed in this standard, and that has an interpretation according to this standard.
- 35 **statement** (3.3) : A sequence of lexical tokens. It usually consists of a single line, but a statement may  
36 be continued from one line to another and the semicolon symbol may be used to separate statements  
37 within a line.
- 38 **statement entity** (16) : An entity identified by a lexical token whose scope is a single statement or  
39 part of a statement.
- 40 **statement function** (12.5.4) : A procedure specified by a single statement that is similar in form to an assignment  
41 statement.

- 1 **statement label** (3.2.4) : A lexical token consisting of up to five digits that precedes a statement and  
2 may be used to refer to the statement.
- 3 **storage association** (16.4.3) : The relationship between two storage sequences if a storage unit of one  
4 is the same as a storage unit of the other.
- 5 **storage sequence** (16.4.3.1) : A sequence of contiguous storage units.
- 6 **storage unit** (16.4.3.1) : A character storage unit, a numeric storage unit, a file storage unit, or an  
7 unspecified storage unit.
- 8 **stride** (6.2.2.3.1) : The increment specified in a subscript triplet.
- 9 **struct** : The C analog of a sequence derived type; defined in 6.2.5 of the C standard.
- 10 **structure** (2.4.1.2) : A scalar data object of derived type.
- 11 **structure component** (6.1.2) : A part of an object of derived type. It may be referenced by an object  
12 designator.
- 13 **structure constructor** (4.5.9) : A syntactic mechanism for constructing a value of derived type.
- 14 **subcomponent** (6.1.2) : A subcomponent of an object of derived type is a component of that object  
15 or of a subobject of that object.
- 16 **subobject** (2.4.3.1) : A portion of a data object that may be referenced or defined independently of  
17 other portions. It may be an array element, an array section, a structure component, a substring, or the  
18 real or imaginary part of a complex object.
- 19 **subprogram** (2.2) : A function subprogram or a subroutine subprogram. Note that in FORTRAN 77, a  
20 block data program unit was called a subprogram.
- 21 **subroutine** (2.2.3) : A procedure that is invoked by a CALL statement or by a defined assignment  
22 statement.
- 23 **subroutine subprogram** (12.5.2.2) : A sequence of statements beginning with a SUBROUTINE state-  
24 ment that is not in an interface block and ending with the corresponding END statement.
- 25 **subscript** (6.2.2) : One of the list of scalar integer expressions in an array element selector. Note that  
26 in FORTRAN 77, the whole list was called the subscript.
- 27 **subscript triplet** (6.2.2) : An item in the list of an array section selector that contains a colon and  
28 specifies a regular sequence of integer values.
- 29 **substring** (6.1.1) : A contiguous portion of a scalar character string. Note that an array section can  
30 include a substring selector; the result is called an array section and not a substring.
- 31 **target** (2.4.6, 5.1.2.14, 6.3.1.2) : A data entity that has the TARGET attribute, or an entity that is  
32 associated with a pointer.
- 33 **transformational function** (13.1) : A function that is either intrinsic or is defined in an intrinsic  
34 module and that is neither an elemental function nor an inquiry function.
- 35 **type** (2.4.1) : A named category of data that is characterized by a set of values, together with a way to  
36 denote these values and a collection of operations that interpret and manipulate the values. The set of  
37 data values depends on the values of the type parameters.
- 38 **type-bound procedure** (4.5.4) : A procedure binding in a type definition. The procedure may be  
39 referenced by the *binding-name* via any object of that dynamic type, as a defined operator, by defined

- 1 assignment, or as part of the finalization process.
- 2 **type compatible** (5.1.1.2) : All entities are type compatible with other entities of the same type.  
3 Unlimited polymorphic entities are type compatible with all entities; other polymorphic entities are type  
4 compatible with entities whose dynamic type is an extension type of the polymorphic entity's declared  
5 type.
- 6 **type declaration statement** (5) : An INTEGER, REAL, DOUBLE PRECISION, COMPLEX,  
7 CHARACTER, LOGICAL, or TYPE (*type-name*) statement.
- 8 **type parameter** (2.4.1.1) : A parameter of a data type. KIND and LEN are the type parameters of  
9 intrinsic types. The type parameters of a derived type are defined in the derived-type definition.
- 10 **type parameter order** (4.5.2.1) : The ordering of the type parameters of a derived type that is used  
11 for derived-type specifiers.
- 12 **ultimate component** (4.5) : For a structure, a component that is of intrinsic type, has the ALLOCAT-  
13 ABLE attribute, or has the POINTER attribute, or an ultimate component of a derived-type component  
14 that does not have the POINTER attribute or the ALLOCATABLE attribute.
- 15 **undefined** (2.5.5) : For a data object, the property of not having a determinate value.
- 16 **unsigned** : A qualifier of a C numeric type indicating that it is comprised only of nonnegative values;  
17 defined in 6.2.5 of the C standard. There is nothing analogous in Fortran.
- 18 **unspecified storage unit** (16.4.3.1) : A unit of storage for holding a pointer or a scalar that is not a  
19 pointer and is of type other than default integer, default character, default real, double precision real,  
20 default logical, or default complex.
- 21 **use association** (16.4.1.2) : The association of names in different scoping units specified by a USE  
22 statement.
- 23 **variable** (2.4.3.1.1) : A data object whose value can be defined and redefined during the execution of  
24 a program. It may be a named data object, an array element, an array section, a structure component,  
25 or a substring. Note that in FORTRAN 77, a variable was always scalar and named.
- 26 **vector subscript** (6.2.2.3.2) : A section subscript that is an integer expression of rank one.
- 27 **void** : A C type comprising an empty set of values; defined in 6.2.5 of the C standard. There is nothing  
28 analogous in Fortran.
- 29 **whole array** (6.2.1) : A named array.

# Annex B

(Informative)

## Decremental features

### B.1 Deleted features

The deleted features are those features of Fortran 90 that were redundant and are considered largely unused. Section 1.8.1 describes the nature of the deleted features. The Fortran 90 features that are not contained in Fortran 95 or this standard are the following:

- (1) Real and double precision DO variables.

The ability present in FORTRAN 77, and for consistency also in Fortran 90, for a DO variable to be of type real or double precision in addition to type integer, has been deleted. A similar result can be achieved by using a DO construct with no loop control and the appropriate exit test.

- (2) Branching to an END IF statement from outside its block.

In FORTRAN 77, and for consistency also in Fortran 90, it was possible to branch to an END IF statement from outside the IF construct; this has been deleted. A similar result can be achieved by branching to a CONTINUE statement that is immediately after the END IF statement.

- (3) PAUSE statement.

The PAUSE statement, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, has been deleted. A similar result can be achieved by writing a message to the appropriate unit, followed by reading from the appropriate unit.

- (4) ASSIGN and assigned GO TO statements and assigned format specifiers.

The ASSIGN statement and the related assigned GO TO statement, present in FORTRAN 66, FORTRAN 77 and for consistency also in Fortran 90, have been deleted. Further, the ability to use an assigned integer as a format, present in FORTRAN 77 and Fortran 90, has been deleted. A similar result can be achieved by using other control constructs instead of the assigned GOTO statement and by using a default character variable to hold a format specification instead of using an assigned integer.

- (5) H edit descriptor.

In FORTRAN 77, and for consistency also in Fortran 90, there was an alternative form of character string edit descriptor, which had been the only such form in FORTRAN 66; this has been deleted. A similar result can be achieved by using a character string edit descriptor.

The following is a list of the previous editions of the international Fortran standard, along with their informal names:

ISO/IEC 1539:1972	FORTAN 66
ISO/IEC 1539:1978	FORTAN 77
ISO/IEC 1539:1991	Fortran 90
ISO/IEC 1539-1:1997	Fortran 95

See the Fortran 90 standard for detailed rules of how these deleted features work.

## 1 B.2 Obsolescent features

2 The obsolescent features are those features of Fortran 90 that were redundant and for which better  
3 methods were available in Fortran 90. Section 1.8.2 describes the nature of the obsolescent features.  
4 The obsolescent features in this standard are the following:

- 5 (1) Arithmetic IF — use the IF statement (8.1.2.4) or IF construct (8.1.2).
- 6 (2) Shared DO termination and termination on a statement other than END DO or CON-  
7 TINUE — use an END DO or a CONTINUE statement for each DO statement.
- 8 (3) Alternate return — see B.2.1.
- 9 (4) Computed GO TO statement — see B.2.2.
- 10 (5) Statement functions — see B.2.3.
- 11 (6) DATA statements amongst executable statements — see B.2.4.
- 12 (7) Assumed length character functions — see B.2.5.
- 13 (8) Fixed form source — see B.2.6.
- 14 (9) CHARACTER\* form of CHARACTER declaration — see B.2.7.

### 15 B.2.1 Alternate return

16 An alternate return introduces labels into an argument list to allow the called procedure to direct the  
17 execution of the caller upon return. The same effect can be achieved with a return code that is used  
18 in a CASE construct on return. This avoids an irregularity in the syntax and semantics of argument  
19 association. For example,

```
20 CALL SUBR_NAME (X, Y, Z, *100, *200, *300)
```

21 may be replaced by

```
22 CALL SUBR_NAME (X, Y, Z, RETURN_CODE)  
23 SELECT CASE (RETURN_CODE)  
24 CASE (1)  
25 ...  
26 CASE (2)  
27 ...  
28 CASE (3)  
29 ...  
30 CASE DEFAULT  
31 ...  
32 END SELECT
```

### 33 B.2.2 Computed GO TO statement

34 The computed GO TO has been superseded by the CASE construct, which is a generalized, easier to  
35 use and more efficient means of expressing the same computation.

### 36 B.2.3 Statement functions

37 Statement functions are subject to a number of nonintuitive restrictions and are a potential source of  
38 error because their syntax is easily confused with that of an assignment statement.

39 The internal function is a more generalized form of the statement function and completely supersedes  
40 it.

#### 1 **B.2.4 DATA statements among executables**

2 The statement ordering rules of FORTRAN 66, and hence of FORTRAN 77 and Fortran 90 for compatibility,  
3 allowed DATA statements to appear anywhere in a program unit after the specification statements. The  
4 ability to position DATA statements amongst executable statements is very rarely used, is unnecessary  
5 and is a potential source of error.

#### 6 **B.2.5 Assumed character length functions**

7 Assumed character length for functions is an irregularity in the language in that elsewhere in Fortran  
8 the philosophy is that the attributes of a function result depend only on the actual arguments of the  
9 invocation and on any data accessible by the function through host or use association. Some uses of this  
10 facility can be replaced with an automatic character length function, where the length of the function  
11 result is declared in a specification expression. Other uses can be replaced by the use of a subroutine  
12 whose arguments correspond to the function result and the function arguments.

13 Note that dummy arguments of a function may be assumed character length.

#### 14 **B.2.6 Fixed form source**

15 Fixed form source was designed when the principal machine-readable input medium for new programs  
16 was punched cards. Now that new and amended programs are generally entered via keyboards with  
17 screen displays, it is an unnecessary overhead, and is potentially error-prone, to have to locate positions  
18 6, 7, or 72 on a line. Free form source was designed expressly for this more modern technology.

19 It is a simple matter for a software tool to convert from fixed to free form source.

#### 20 **B.2.7 CHARACTER\* form of CHARACTER declaration**

21 Fortran 90 had two different forms of specifying the length selector in CHARACTER declarations. The  
22 older form (CHARACTER\*char-length) is redundant.



# Annex C

(Informative)

## Extended notes

### C.1 Section 4 notes

#### C.1.1 Intrinsic and derived types (4.4, 4.5)

FORTRAN 77 provided only types explicitly defined in the standard (logical, integer, real, double precision, complex, and character). This standard provides those intrinsic types and provides derived types to allow the creation of new types. A derived-type definition specifies a data structure consisting of components of intrinsic types and of derived types. Such a type definition does not represent a data object, but rather, a template for declaring named objects of that derived type. For example, the definition

```
11 TYPE POINT
12     INTEGER X_COORD
13     INTEGER Y_COORD
14 END TYPE POINT
```

specifies a new derived type named POINT which is composed of two components of intrinsic type integer (X\_COORD and Y\_COORD). The statement TYPE (POINT) FIRST, LAST declares two data objects, FIRST and LAST, that can hold values of type POINT.

FORTRAN 77 provided REAL and DOUBLE PRECISION intrinsic types as approximations to mathematical real numbers. This standard generalizes REAL as an intrinsic type with a type parameter that selects the approximation method. The type parameter is named kind and has values that are processor dependent. DOUBLE PRECISION is treated as a synonym for REAL (*k*), where *k* is the implementation-defined kind type parameter value KIND (0.0D0).

Real literal constants may be specified with a kind type parameter to ensure that they have a particular kind type parameter value (4.4.2).

For example, with the specifications

```
26 INTEGER Q
27 PARAMETER (Q = 8)
28 REAL (Q) B
```

the literal constant 10.93\_Q has the same precision as the variable B.

FORTRAN 77 did not allow zero-length character strings. They are permitted by this standard (4.4.4).

Objects are of different derived type if they are declared using different derived-type definitions. For example,

```
33 TYPE APPLES
```

```
1     INTEGER NUMBER
2 END TYPE APPLES
3 TYPE ORANGES
4     INTEGER NUMBER
5 END TYPE ORANGES
6 TYPE (APPLES) COUNT1
7 TYPE (ORANGES) COUNT2
8 COUNT1 = COUNT2 ! Erroneous statement mixing apples and oranges
```

9 Even though all components of objects of type APPLES and objects of type ORANGES have identical  
10 intrinsic types, the objects are of different types.

### 11 **C.1.2 Selection of the approximation methods (4.4.2)**

12 One can select the real approximation method for an entire program through the use of a module and  
13 the parameterized real type. This is accomplished by defining a named integer constant to have a  
14 particular kind type parameter value and using that named constant in all real, complex, and derived-  
15 type declarations. For example, the specification statements

```
16 INTEGER, PARAMETER :: LONG_FLOAT = 8
17 REAL (LONG_FLOAT) X, Y
18 COMPLEX (LONG_FLOAT) Z
```

19 specify that the approximation method corresponding to a kind type parameter value of 8 is supplied for  
20 the data objects X, Y, and Z in the program unit. The kind type parameter value LONG\_FLOAT can  
21 be made available to an entire program by placing the INTEGER specification statement in a module  
22 and accessing the named constant LONG\_FLOAT with a USE statement. Note that by changing 8 to 4  
23 once in the module, a different approximation method is selected.

24 To avoid the use of the processor-dependent values 4 or 8, replace 8 by KIND (0.0) or KIND (0.0D0).  
25 Another way to avoid these processor-dependent values is to select the kind value using the intrinsic  
26 inquiry function SELECTED\_REAL\_KIND. This function, given integer arguments P and R specifying  
27 minimum requirements for decimal precision and decimal exponent range, respectively, returns the kind  
28 type parameter value of the approximation method that has at least P decimal digits of precision and  
29 at least a range for positive numbers of  $10^{-R}$  to  $10^R$ . In the above specification statement, the 8 may be  
30 replaced by, for instance, SELECTED\_REAL\_KIND (10, 50), which requires an approximation method  
31 to be selected with at least 10 decimal digits of precision and a range from  $10^{-50}$  to  $10^{50}$ . There are no  
32 magnitude or ordering constraints placed on kind values, in order that implementers may have flexibility  
33 in assigning such values and may add new kinds without changing previously assigned kind values.

34 As kind values have no portable meaning, a good practice is to use them in programs only through  
35 named constants as described above (for example, SINGLE, IEEE\_SINGLE, DOUBLE, and QUAD),  
36 rather than using the kind values directly.

### 37 **C.1.3 Type extension and component accessibility (4.5.1.1, 4.5.3)**

38 The default accessibility of an extended type may be specified in the type definition. The accessibility  
39 of its components may be specified individually.

40 module types

```

1   type base_type
2     private          !-- Sets default accessibility
3     integer :: i     !-- a private component
4     integer, private :: j !-- another private component
5     integer, public :: k !-- a public component
6   end type base_type
7
8   type, extends(base_type) :: my_type
9     private          !-- Sets default for components declared in my_type
10    integer :: l     !-- A private component.
11    integer, public :: m !-- A public component.
12  end type my_type
13
14  end module types
15
16  subroutine sub
17    use types
18    type (my_type) :: x
19
20    ....
21
22    call another_sub( &
23      x%base_type, & !-- ok because base_type is a public subobject of x
24      x%base_type%k, & !-- ok because x%base_type is ok and has k as a
25                      !-- public component.
26      x%k, & !-- ok because it is shorthand for x%base_type%k
27      x%base_type%i, & !-- Invalid because i is private.
28      x%i) !-- Invalid because it is shorthand for x%base_type%i
29  end subroutine sub

```

### 30 C.1.4 Abstract types

31 The following defines an object that can be displayed in an X window:

```

32   TYPE, ABSTRACT :: DRAWABLE_OBJECT
33     REAL, DIMENSION(3) :: RGB_COLOR=(/1.0,1.0,1.0/) ! White
34     REAL, DIMENSION(2) :: POSITION=(/0.0,0.0/) ! Centroid
35   CONTAINS
36     PROCEDURE(RENDER_X), PASS(OBJECT), DEFERRED :: RENDER
37   END TYPE DRAWABLE_OBJECT
38
39   ABSTRACT INTERFACE
40     SUBROUTINE RENDER_X(OBJECT, WINDOW)
41     CLASS(DRAWABLE_OBJECT), INTENT(IN) :: OBJECT
42     CLASS(X_WINDOW), INTENT(INOUT) :: WINDOW

```

```

1         END SUBROUTINE RENDER_X
2     END INTERFACE

```

3 We can declare a nonabstract type by extending the abstract type:

```

4     TYPE, EXTENDS(DRAWABLE_OBJECT) :: DRAWABLE_TRIANGLE ! Not ABSTRACT
5     REAL, DIMENSION(2,3) :: VERTICES ! In relation to centroid
6     CONTAINS
7     PROCEDURE, PASS(TRIANGLE) :: RENDER=>RENDER_TRIANGLE_X
8     END TYPE DRAWABLE_TRIANGLE

```

9 The actual drawing procedure will draw a triangle in WINDOW with vertices at x coordinates  
10 TRIANGLE%POSITION(1)+TRIANGLE%VERTICES(1,:) and y coordinates  
11 TRIANGLE%POSITION(2)+TRIANGLE%VERTICES(2,:):

```

12     SUBROUTINE RENDER_TRIANGLE_X(TRIANGLE, WINDOW)
13     CLASS(DRAWABLE_TRIANGLE), INTENT(IN) :: TRIANGLE
14     CLASS(X_WINDOW), INTENT(INOUT) :: WINDOW
15     ...
16     END SUBROUTINE RENDER_TRIANGLE_X

```

### 17 C.1.5 Pointers (4.5.1)

18 Pointers are names that can change dynamically their association with a target object. In a sense, a  
19 normal variable is a name with a fixed association with a particular object. A normal variable name  
20 refers to the same storage space throughout the lifetime of the variable. A pointer name may refer  
21 to different storage space, or even no storage space, at different times. A variable may be considered  
22 to be a descriptor for space to hold values of the appropriate type, type parameters, and array rank  
23 such that the values stored in the descriptor are fixed when the variable is created. A pointer also may  
24 be considered to be a descriptor, but one whose values may be changed dynamically so as to describe  
25 different pieces of storage. When a pointer is declared, space to hold the descriptor is created, but the  
26 space for the target object is not created.

27 A derived type may have one or more components that are defined to be pointers. It may have a  
28 component that is a pointer to an object of the same derived type. This “recursive” data definition  
29 allows dynamic data structures such as linked lists, trees, and graphs to be constructed. For example:

```

30 TYPE NODE          ! Define a ''recursive'' type
31     INTEGER :: VALUE = 0
32     TYPE (NODE), POINTER :: NEXT_NODE => NULL ( )
33 END TYPE NODE
34
35 TYPE (NODE), TARGET :: HEAD          ! Automatically initialized
36 TYPE (NODE), POINTER :: CURRENT, TEMP ! Declare pointers
37 INTEGER :: IOEM, K
38
39 CURRENT => HEAD          ! CURRENT points to head of list

```

```

1
2 DO
3   READ (*, *, IOSTAT = IOEM) K ! Read next value, if any
4   IF (IOEM /= 0) EXIT
5   ALLOCATE (TEMP)             ! Create new cell each iteration
6   TEMP % VALUE = K           ! Assign value to cell
7   CURRENT % NEXT_NODE => TEMP ! Attach new cell to list
8   CURRENT => TEMP            ! CURRENT points to new end of list
9 END DO

```

10 A list is now constructed and the last linked cell contains a disassociated pointer. A loop can be used  
 11 to “walk through” the list.

```

12 CURRENT => HEAD
13 DO
14   IF (.NOT. ASSOCIATED (CURRENT % NEXT_NODE)) EXIT
15   CURRENT => CURRENT % NEXT_NODE
16   WRITE (*, *) CURRENT % VALUE
17 END DO

```

### 18 C.1.6 Structure constructors and generic names

19 A generic name may be the same as a type name. This can be used to emulate user-defined structure  
 20 constructors for that type, even if the type has private components. For example:

```

21 MODULE mytype_module
22   TYPE mytype
23     PRIVATE
24     COMPLEX value
25     LOGICAL exact
26   END TYPE
27   INTERFACE mytype
28     MODULE PROCEDURE int_to_mytype
29   END INTERFACE
30   ! Operator definitions etc.
31   ...
32 CONTAINS
33   TYPE(mytype) FUNCTION int_to_mytype(i)
34     INTEGER, INTENT(IN) :: i
35     int_to_mytype%value = i
36     int_to_mytype%exact = .TRUE.
37   END FUNCTION
38   ! Procedures to support operators etc.
39   ...
40 END

```

```
1
2 PROGRAM example
3   USE mytype_module
4   TYPE(mytype) x
5   x = mytype(17)
6 END
```

7 The type name may still be used as a generic name if the type has type parameters. For example:

```
8 MODULE m
9   TYPE t(kind)
10    INTEGER, KIND :: kind
11    COMPLEX(kind) value
12  END TYPE
13  INTEGER, PARAMETER :: single = KIND(0.0), double = KIND(0d0)
14  INTERFACE t
15    MODULE PROCEDURE real_to_t1, dble_to_t2, int_to_t1, int_to_t2
16  END INTERFACE
17  ...
18 CONTAINS
19  TYPE(t(single)) FUNCTION real_to_t1(x)
20    REAL(single) x
21    real_to_t1%value = x
22  END FUNCTION
23  TYPE(t(double)) FUNCTION dble_to_t2(x)
24    REAL(double) x
25    dble_to_t2%value = x
26  END FUNCTION
27  TYPE(t(single)) FUNCTION int_to_t1(x,mold)
28    INTEGER x
29    TYPE(t(single)) mold
30    int_to_t1%value = x
31  END FUNCTION
32  TYPE(t(double)) FUNCTION int_to_t2(x,mold)
33    INTEGER x
34    TYPE(t(double)) mold
35    int_to_t2%value = x
36  END FUNCTION
37  ...
38 END
39
40 PROGRAM example
41   USE m
42   TYPE(t(single)) x
43   TYPE(t(double)) y
```

```

1  x = t(1.5)                ! References real_to_t1
2  x = t(17,mold=x)         ! References int_to_t1
3  y = t(1.5d0)             ! References dble_to_t2
4  y = t(42,mold=y)        ! References int_to_t2
5  y = t(kind(0d0)) ((0,1)) ! Uses the structure constructor for type t
6  END

```

### 7 C.1.7 Generic type-bound procedures

8 Example of a derived type with generic type-bound procedures:

9 The only difference between this example and the same thing rewritten to use generic interface blocks  
10 is that with type-bound procedures,

```
11     USE(rational_numbers),ONLY :: rational
```

12 does not block the type-bound procedures; the user still gets access to the defined assignment and  
13 extended operations.

```

14 MODULE rational_numbers
15     IMPLICIT NONE
16     PRIVATE
17     TYPE,PUBLIC :: rational
18     PRIVATE
19     INTEGER n,d
20     CONTAINS
21     ! ordinary type-bound procedure
22     PROCEDURE :: real => rat_to_real
23     ! specific type-bound procedures for generic support
24     PROCEDURE,PRIVATE :: rat_asgn_i, rat_plus_rat, rat_plus_i
25     PROCEDURE,PRIVATE,PASS(b) :: i_plus_rat
26     ! generic type-bound procedures
27     GENERIC :: ASSIGNMENT(=) => rat_asgn_i
28     GENERIC :: OPERATOR(+) => rat_plus_rat, rat_plus_i, i_plus_rat
29     END TYPE
30     CONTAINS
31     ELEMENTAL REAL FUNCTION rat_to_real(this) RESULT(r)
32     CLASS(rational),INTENT(IN) :: this
33     r = REAL(this%n)/this%d
34     END FUNCTION
35     ELEMENTAL SUBROUTINE rat_asgn_i(a,b)
36     CLASS(rational),INTENT(OUT) :: a
37     INTEGER,INTENT(IN) :: b
38     a%n = b
39     a%d = 1
40     END SUBROUTINE

```

```

1  ELEMENTAL TYPE(rational) FUNCTION rat_plus_i(a,b) RESULT(r)
2    CLASS(rational),INTENT(IN) :: a
3    INTEGER,INTENT(IN) :: b
4    r%n = a%n + b*a%d
5    r%d = a%d
6  END FUNCTION
7  ELEMENTAL TYPE(rational) FUNCTION i_plus_rat(a,b) RESULT(r)
8    INTEGER,INTENT(IN) :: a
9    CLASS(rational),INTENT(IN) :: b
10   r%n = b%n + a*b%d
11   r%d = b%d
12  END FUNCTION
13  ELEMENTAL TYPE(rational) FUNCTION rat_plus_rat(a,b) RESULT(r)
14   CLASS(rational),INTENT(IN) :: a,b
15   r%n = a%n*b%d + b%n*a%d
16   r%d = a%d*b%d
17  END FUNCTION
18  END

```

### 19 **C.1.8 Final subroutines (4.5.5, 4.5.5.1, 4.5.5.2, 4.5.5.3)**

20 Example of a parameterized derived type with final subroutines:

```

21  MODULE m
22    TYPE t(k)
23      INTEGER, KIND :: k
24      REAL(k),POINTER :: vector(:) => NULL()
25    CONTAINS
26      FINAL :: finalize_t1s, finalize_t1v, finalize_t2e
27    END TYPE
28  CONTAINS
29    SUBROUTINE finalize_t1s(x)
30      TYPE(t(KIND(0.0))) x
31      IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
32    END SUBROUTINE
33    SUBROUTINE finalize_t1v(x)
34      TYPE(t(KIND(0.0))) x(:)
35      DO i=LBOUND(x,1),UBOUND(x,1)
36        IF (ASSOCIATED(x(i)%vector)) DEALLOCATE(x(i)%vector)
37      END DO
38    END SUBROUTINE
39    ELEMENTAL SUBROUTINE finalize_t2e(x)
40      TYPE(t(KIND(0.0d0))),INTENT(INOUT) :: x
41      IF (ASSOCIATED(x%vector)) DEALLOCATE(x%vector)
42    END SUBROUTINE

```

```

1  END MODULE
2
3  SUBROUTINE example(n)
4    USE m
5    TYPE(t(KIND(0.0))) a,b(10),c(n,2)
6    TYPE(t(KIND(0.0d0))) d(n,n)
7    ...
8    ! Returning from this subroutine will effectively do
9    !   CALL finalize_t1s(a)
10   !   CALL finalize_t1v(b)
11   !   CALL finalize_t2e(d)
12   ! No final subroutine will be called for variable C because the user
13   ! omitted to define a suitable specific procedure for it.
14  END SUBROUTINE

```

15 Example of extended types with final subroutines:

```

16  MODULE m
17    TYPE t1
18      REAL a,b
19    END TYPE
20    TYPE,EXTENDS(t1) :: t2
21      REAL,POINTER :: c(:),d(:)
22    CONTAINS
23      FINAL :: t2f
24    END TYPE
25    TYPE,EXTENDS(t2) :: t3
26      REAL,POINTER :: e
27    CONTAINS
28      FINAL :: t3f
29    END TYPE
30    ...
31  CONTAINS
32    SUBROUTINE t2f(x) ! Finalizer for TYPE(t2)'s extra components
33      TYPE(t2) :: x
34      IF (ASSOCIATED(x%c)) DEALLOCATE(x%c)
35      IF (ASSOCIATED(x%d)) DEALLOCATE(x%d)
36    END SUBROUTINE
37    SUBROUTINE t3f(y) ! Finalizer for TYPE(t3)'s extra components
38      TYPE(t3) :: y
39      IF (ASSOCIATED(y%e)) DEALLOCATE(y%e)
40    END SUBROUTINE
41  END MODULE
42
43  SUBROUTINE example

```

```

1  USE m
2  TYPE(t1) x1
3  TYPE(t2) x2
4  TYPE(t3) x3
5  ...
6  ! Returning from this subroutine will effectively do
7  !   ! Nothing to x1; it is not finalizable
8  !   CALL t2f(x2)
9  !   CALL t3f(x3)
10 !   CALL t2f(x3%t2)
11 END SUBROUTINE

```

## 12 C.2 Section 5 notes

### 13 C.2.1 The POINTER attribute (5.1.2.11)

14 The POINTER attribute shall be specified to declare a pointer. The type, type parameters, and rank,  
15 which may be specified in the same statement or with one or more attribute specification statements,  
16 determine the characteristics of the target objects that may be associated with the pointers declared  
17 in the statement. An obvious model for interpreting declarations of pointers is that such declarations  
18 create for each name a descriptor. Such a descriptor includes all the data necessary to describe fully  
19 and locate in memory an object and all subobjects of the type, type parameters, and rank specified.  
20 The descriptor is created empty; it does not contain values describing how to access an actual memory  
21 space. These descriptor values will be filled in when the pointer is associated with actual target space.

22 The following example illustrates the use of pointers in an iterative algorithm:

```

23 PROGRAM DYNAM_ITER
24   REAL, DIMENSION (:, :), POINTER :: A, B, SWAP ! Declare pointers
25   ...
26   READ (*, *) N, M
27   ALLOCATE (A (N, M), B (N, M)) ! Allocate target arrays
28   ! Read values into A
29   ...
30   ITER: DO
31     ...
32     ! Apply transformation of values in A to produce values in B
33     ...
34     IF (CONVERGED) EXIT ITER
35     ! Swap A and B
36     SWAP => A; A => B; B => SWAP
37   END DO ITER
38   ...
39 END PROGRAM DYNAM_ITER

```

## 1 C.2.2 The TARGET attribute (5.1.2.14)

2 The TARGET attribute shall be specified for any nonpointer object that may, during the execution of the  
 3 program, become associated with a pointer. This attribute is defined primarily for optimization purposes.  
 4 It allows the processor to assume that any nonpointer object not explicitly declared as a target may  
 5 be referred to only by way of its original declared name. It also means that implicitly-declared objects  
 6 shall not be used as pointer targets. This will allow a processor to perform optimizations that otherwise  
 7 would not be possible in the presence of certain pointers.

8 The following example illustrates the use of the TARGET attribute in an iterative algorithm:

```

9 PROGRAM ITER
10   REAL, DIMENSION (1000, 1000), TARGET :: A, B
11   REAL, DIMENSION (:, :), POINTER      :: IN, OUT, SWAP
12   ...
13   ! Read values into A
14   ...
15   IN => A           ! Associate IN with target A
16   OUT => B          ! Associate OUT with target B
17   ...
18   ITER:DO
19     ...
20     ! Apply transformation of IN values to produce OUT
21     ...
22     IF (CONVERGED) EXIT ITER
23     ! Swap IN and OUT
24     SWAP => IN; IN => OUT; OUT => SWAP
25   END DO ITER
26   ...
27 END PROGRAM ITER

```

## 28 C.2.3 The VOLATILE attribute (5.1.2.16)

29 The following example shows the use of a variable with the VOLATILE attribute to communicate with  
 30 an asynchronous process, in this case the operating system. The program detects a user keystroke on  
 31 the terminal and reacts at a convenient point in its processing.

32 The VOLATILE attribute is necessary to prevent an optimizing compiler from storing the communication  
 33 variable in a register or from doing flow analysis and deciding that the EXIT statement can never be  
 34 executed.

```

35 SUBROUTINE TERMINATE_ITERATIONS
36
37   LOGICAL, VOLATILE :: USER_HIT_ANY_KEY
38
39   ! Have the OS start to look for a user keystroke and set the variable
40   ! "USER_HIT_ANY_KEY" to TRUE as soon as it detects a keystroke.
41   ! This pseudo call is operating system dependent.

```

```
1
2 CALL OS_BEGIN_DETECT_USER_KEYSTROKE( USER_HIT_ANY_KEY )
3
4 USER_HIT_ANY_KEY = .FALSE.      ! This will ignore any recent keystrokes
5
6 PRINT *, " Hit any key to terminate iterations!"
7
8 DO I = 1,100
9     ...                          ! Compute a value for R
10    PRINT *, I, R
11    IF (USER_HIT_ANY_KEY)        EXIT
12 ENDDO
13
14 ! Have the OS stop looking for user keystrokes
15 CALL OS_STOP_DETECT_USER_KEYSTROKE
16
17 END SUBROUTINE TERMINATE_ITERATIONS
```

## 18 C.3 Section 6 notes

### 19 C.3.1 Structure components (6.1.2)

20 Components of a structure are referenced by writing the components of successive levels of the structure  
21 hierarchy until the desired component is described. For example,

```
22 TYPE ID_NUMBERS
23     INTEGER SSN
24     INTEGER EMPLOYEE_NUMBER
25 END TYPE ID_NUMBERS
26
27 TYPE PERSON_ID
28     CHARACTER (LEN=30) LAST_NAME
29     CHARACTER (LEN=1) MIDDLE_INITIAL
30     CHARACTER (LEN=30) FIRST_NAME
31     TYPE (ID_NUMBERS) NUMBER
32 END TYPE PERSON_ID
33
34 TYPE PERSON
35     INTEGER AGE
36     TYPE (PERSON_ID) ID
37 END TYPE PERSON
38
39 TYPE (PERSON) GEORGE, MARY
40
41 PRINT *, GEORGE % AGE          ! Print the AGE component
```

```

1 PRINT *, MARY % ID % LAST_NAME      ! Print LAST_NAME of MARY
2 PRINT *, MARY % ID % NUMBER % SSN ! Print SSN of MAR
3 PRINT *, GEORGE % ID % NUMBER      ! Print SSN and EMPLOYEE_NUMBER of GEORGE

```

4 A structure component may be a data object of intrinsic type as in the case of GEORGE % AGE or it  
5 may be of derived type as in the case of GEORGE % ID % NUMBER. The resultant component may  
6 be a scalar or an array of intrinsic or derived type.

```

7 TYPE LARGE
8     INTEGER ELT (10)
9     INTEGER VAL
10 END TYPE LARGE
11
12 TYPE (LARGE) A (5)      ! 5 element array, each of whose elements
13                        ! includes a 10 element array ELT and
14                        ! a scalar VAL.
15 PRINT *, A (1)         ! Prints 10 element array ELT and scalar VAL.
16 PRINT *, A (1) % ELT (3) ! Prints scalar element 3
17                        ! of array element 1 of A.
18 PRINT *, A (2:4) % VAL ! Prints scalar VAL for array elements
19                        ! 2 to 4 of A.

```

20 Components of an object of extensible type that are inherited from the parent type may be accessed as  
21 a whole by using the parent component name, or individually, either with or without qualifying them  
22 by the parent component name.

23 For example:

```

24 TYPE POINT              ! A base type
25     REAL :: X, Y
26 END TYPE POINT
27 TYPE, EXTENDS(POINT) :: COLOR_POINT ! An extension of TYPE(POINT)
28     ! Components X and Y, and component name POINT, inherited from parent
29     INTEGER :: COLOR
30 END TYPE COLOR_POINT
31
32 TYPE(POINT) :: PV = POINT(1.0, 2.0)
33 TYPE(COLOR_POINT) :: CPV = COLOR_POINT(PV, 3) ! Nested form constructor
34
35 PRINT *, CPV%POINT          ! Prints 1.0 and 2.0
36 PRINT *, CPV%POINT%X, CPV%POINT%Y ! And this does, too
37 PRINT *, CPV%X, CPV%Y      ! And this does, too

```

### 38 C.3.2 Allocation with dynamic type (6.3.1)

39 The following example illustrates the use of allocation with the value and dynamic type of the allocated  
40 object given by another object. The example copies a list of objects of any type. It copies the list

1 starting at IN\_LIST. After copying, each element of the list starting at LIST\_COPY has a polymorphic  
 2 component, ITEM, for which both the value and type are taken from the ITEM component of the  
 3 corresponding element of the list starting at IN\_LIST.

```

4 TYPE :: LIST ! A list of anything
5   TYPE(LIST), POINTER :: NEXT => NULL()
6   CLASS(*), ALLOCATABLE :: ITEM
7 END TYPE LIST
8 ...
9 TYPE(LIST), POINTER :: IN_LIST, LIST_COPY => NULL()
10 TYPE(LIST), POINTER :: IN_WALK, NEW_TAIL
11 ! Copy IN_LIST to LIST_COPY
12 IF (ASSOCIATED(IN_LIST)) THEN
13   IN_WALK => IN_LIST
14   ALLOCATE(LIST_COPY)
15   NEW_TAIL => LIST_COPY
16   DO
17     ALLOCATE(NEW_TAIL%ITEM, SOURCE=IN_WALK%ITEM)
18     IN_WALK => IN_WALK%NEXT
19     IF (.NOT. ASSOCIATED(IN_WALK)) EXIT
20     ALLOCATE(NEW_TAIL%NEXT)
21     NEW_TAIL => NEW_TAIL%NEXT
22   END DO
23 END IF

```

### 24 C.3.3 Pointer allocation and association

25 The effect of ALLOCATE, DEALLOCATE, NULLIFY, and pointer assignment is that they are inter-  
 26 preted as changing the values in the descriptor that is the pointer. An ALLOCATE is assumed to create  
 27 space for a suitable object and to “assign” to the pointer the values necessary to describe that space.  
 28 A NULLIFY breaks the association of the pointer with the space. A DEALLOCATE breaks the asso-  
 29 ciation and releases the space. Depending on the implementation, it could be seen as setting a flag in  
 30 the pointer that indicates whether the values in the descriptor are valid, or it could clear the descriptor  
 31 values to some (say zero) value indicative of the pointer not pointing to anything. A pointer assignment  
 32 copies the values necessary to describe the space occupied by the target into the descriptor that is the  
 33 pointer. Descriptors are copied, values of objects are not.

34 If PA and PB are both pointers and PB is associated with a target, then

35 PA => PB

36 results in PA being associated with the same target as PB. If PB was disassociated, then PA becomes  
 37 disassociated.

38 The standard is specified so that such associations are direct and independent. A subsequent statement

39 PB => D

40 or

41 ALLOCATE (PB)

1 has no effect on the association of PA with its target. A statement

2 DEALLOCATE (PB)

3 deallocates the space that is associated with both PA and PB. PB becomes disassociated, but there is  
4 no requirement that the processor make it explicitly recognizable that PA no longer has a target. This  
5 leaves PA as a “dangling pointer” to space that has been released. The program shall not use PA again  
6 until it becomes associated via pointer assignment or an ALLOCATE statement.

7 DEALLOCATE may only be used to release space that was created by a previous ALLOCATE. Thus  
8 the following is invalid:

9 REAL, TARGET :: T

10 REAL, POINTER :: P

11 ...

12 P = > T

13 DEALLOCATE (P) ! Not allowed: P's target was not allocated

14 The basic principle is that ALLOCATE, NULLIFY, and pointer assignment primarily affect the pointer  
15 rather than the target. ALLOCATE creates a new target but, other than breaking its connection with  
16 the specified pointer, it has no effect on the old target. Neither NULLIFY nor pointer assignment has  
17 any effect on targets. A piece of memory that was allocated and associated with a pointer will become  
18 inaccessible to a program if the pointer is nullified or associated with a different target and no other  
19 pointer was associated with this piece of memory. Such pieces of memory may be reused by the processor  
20 if this is expedient. However, whether such inaccessible memory is in fact reused is entirely processor  
21 dependent.

## 22 C.4 Section 7 notes

### 23 C.4.1 Character assignment

24 The FORTRAN 77 restriction that none of the character positions being defined in the character assign-  
25 ment statement may be referenced in the expression has been removed (7.4.1.3).

### 26 C.4.2 Evaluation of function references

27 If more than one function reference appears in a statement, they may be executed in any order (subject to  
28 a function result being evaluated after the evaluation of its arguments) and their values shall not depend  
29 on the order of execution. This lack of dependence on order of evaluation permits parallel execution of  
30 the function references (7.1.8.1).

### 31 C.4.3 Pointers in expressions

32 A pointer is basically considered to be like any other variable when it is used as a primary in an expression.  
33 If a pointer is used as an operand to an operator that expects a value, the pointer will automatically  
34 deliver the value stored in the space described by the pointer, that is, the value of the target object  
35 associated with the pointer.

### 36 C.4.4 Pointers on the left side of an assignment

37 A pointer that appears on the left of an intrinsic assignment statement also is dereferenced and is taken  
38 to be referring to the space that is its current target. Therefore, the assignment statement specifies the

1 normal copying of the value of the right-hand expression into this target space. All the normal rules of  
 2 intrinsic assignment hold; the type and type parameters of the expression and the pointer target shall  
 3 agree and the shapes shall be conformable.

4 For intrinsic assignment of derived types, nonpointer components are assigned and pointer components  
 5 are pointer assigned. Dereferencing is applied only to entire scalar objects, not selectively to pointer  
 6 subobjects.

7 For example, suppose a type such as

```
8 TYPE CELL
9     INTEGER :: VAL
10    TYPE (CELL), POINTER :: NEXT_CELL
11 END TYPE CELL
```

12 is defined and objects such as HEAD and CURRENT are declared using

```
13 TYPE (CELL), TARGET :: HEAD
14 TYPE (CELL), POINTER :: CURRENT
```

15 If a linked list has been created and attached to HEAD and the pointer CURRENT has been allocated  
 16 space, statements such as

```
17 CURRENT = HEAD
18 CURRENT = CURRENT % NEXT_CELL
```

19 cause the contents of the cells referenced on the right to be copied to the cell referred to by CURRENT.  
 20 In particular, the right-hand side of the second statement causes the pointer component in the cell,  
 21 CURRENT, to be selected. This pointer is dereferenced because it is in an expression context to produce  
 22 the target's integer value and a pointer to a cell that is in the target's NEXT\_CELL component. The  
 23 left-hand side causes the pointer CURRENT to be dereferenced to produce its present target, namely  
 24 space to hold a cell (an integer and a cell pointer). The integer value on the right is copied to the integer  
 25 space on the left and the pointer component is pointer assigned (the descriptor on the right is copied  
 26 into the space for a descriptor on the left). When a statement such as

```
27 CURRENT => CURRENT % NEXT_CELL
```

28 is executed, the descriptor value in CURRENT % NEXT\_CELL is copied to the descriptor named  
 29 CURRENT. In this case, CURRENT is made to point at a different target.

30 In the intrinsic assignment statement, the space associated with the current pointer does not change but  
 31 the values stored in that space do. In the pointer assignment, the current pointer is made to associate  
 32 with different space. Using the intrinsic assignment causes a linked list of cells to be moved up through  
 33 the current "window"; the pointer assignment causes the current pointer to be moved down through the  
 34 list of cells.

### 35 **C.4.5 An example of a FORALL construct containing a WHERE construct**

```
36 INTEGER :: A(5,5)
37 ...
38 FORALL (I = 1:5)
39     WHERE (A(I,:) == 0)
```

```

1      A(:,I) = I
2      ELSEWHERE (A(I,:) > 2)
3      A(I,:) = 6
4      END WHERE
5  END FORALL

```

6 If prior to execution of the FORALL, A has the value

```

7  A =      1  0  0  0  0
8          2  1  1  1  0
9          1  2  2  0  2
10         2  1  0  2  3
11         1  0  0  0  0

```

12 After execution of the assignment statements following the WHERE statement A has the value A'. The  
13 mask created from row one is used to mask the assignments to column one; the mask from row two is  
14 used to mask assignments to column two; etc.

```

15 A' =      1  0  0  0  0
16          1  1  1  1  5
17          1  2  2  4  5
18          1  1  3  2  5
19          1  2  0  0  5

```

20 The masks created for assignments following the ELSEWHERE statement are

```

21 .NOT. (A(I,:) == 0) .AND. (A'(I,:) > 2)

```

22 Thus the only elements affected by the assignments following the ELSEWHERE statement are A(3, 5)  
23 and A(4, 5). After execution of the FORALL construct, A has the value

```

24 A =      1  0  0  0  0
25          1  1  1  1  5
26          1  2  2  4  6
27          1  1  3  2  6
28          1  2  0  0  5

```

## 29 C.4.6 Examples of FORALL statements

30 Example 1:

```

31 FORALL (J=1:M, K=1:N) X(K, J) = Y(J, K)
32 FORALL (K=1:N) X(K, 1:M) = Y(1:M, K)

```

33 These statements both copy columns 1 through N of array Y into rows 1 through N of array X. They  
34 are equivalent to

```

35 X(1:N, 1:M) = TRANSPOSE (Y(1:M, 1:N) )

```

1 Example 2:

2 The following FORALL statement computes five partial sums of subarrays of J.

3  $J = (/ 1, 2, 3, 4, 5 /)$

4 `FORALL (K = 1:5) J(K) = SUM (J(1:K) )`

5 SUM is allowed in a FORALL because intrinsic functions are pure (12.6). After execution of the FORALL  
6 statement,  $J = (/ 1, 3, 6, 10, 15 /)$ .

7 Example 3:

8 `FORALL (I = 2:N-1) X(I) = (X(I-1) + 2*X(I) + X(I+1) ) / 4`

9 has the same effect as

10  $X(2:N-1) = (X(1:N-2) + 2*X(2:N-1) + X(3:N) ) / 4$

## 11 C.5 Section 8 notes

### 12 C.5.1 Loop control

13 Fortran provides several forms of loop control:

14 (1) With an iteration count and a DO variable. This is the classic Fortran DO loop.

15 (2) Test a logical condition before each execution of the loop (DO WHILE).

16 (3) DO “forever”.

### 17 C.5.2 The CASE construct

18 At most one case block is selected for execution within a CASE construct, and there is no fall-through  
19 from one block into another block within a CASE construct. Thus there is no requirement for the user  
20 to exit explicitly from a block.

### 21 C.5.3 Examples of DO constructs

22 The following are all valid examples of block DO constructs.

23 Example 1:

```

24     SUM = 0.0
25     READ (IUN) N
26     OUTER: DO L = 1, N           ! A DO with a construct name
27         READ (IUN) IQUAL, M, ARRAY (1:M)
28         IF (IQUAL < IQUAL_MIN) CYCLE OUTER    ! Skip inner loop
29         INNER: DO 40 I = 1, M       ! A DO with a label and a name
30             CALL CALCULATE (ARRAY (I), RESULT)
31             IF (RESULT < 0.0) CYCLE
32             SUM = SUM + RESULT
33             IF (SUM > SUM_MAX) EXIT OUTER
34     40     END DO INNER
35     END DO OUTER

```

1 The outer loop has an iteration count of MAX (N, 0), and will execute that number of times or until  
 2 SUM exceeds SUM\_MAX, in which case the EXIT OUTER statement terminates both loops. The inner  
 3 loop is skipped by the first CYCLE statement if the quality flag, IQUAL, is too low. If CALCULATE  
 4 returns a negative RESULT, the second CYCLE statement prevents it from being summed. Both loops  
 5 have construct names and the inner loop also has a label. A construct name is required in the EXIT  
 6 statement in order to terminate both loops, but is optional in the CYCLE statements because each  
 7 belongs to its innermost loop.

8 Example 2:

```

9      N = 0
10     DO 50, I = 1, 10
11         J = I
12         DO K = 1, 5
13             L = K
14             N = N + 1 ! This statement executes 50 times
15         END DO      ! Nonlabeled DO inside a labeled DO
16     50 CONTINUE

```

17 After execution of the above program fragment, I = 11, J = 10, K = 6, L = 5, and N = 50.

18 Example 3:

```

19     N = 0
20     DO I = 1, 10
21         J = I
22         DO 60, K = 5, 1 ! This inner loop is never executed
23             L = K
24             N = N + 1
25     60 CONTINUE      ! Labeled DO inside a nonlabeled DO
26 END DO

```

27 After execution of the above program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by  
 28 these statements.

29 The following are all valid examples of nonblock DO constructs:

30 Example 4:

```

31     DO 70
32         READ (IUN, '(1X, G14.7)', IOSTAT = IOS) X
33         IF (IOS /= 0) EXIT
34         IF (X < 0.) GOTO 70
35         CALL SUBA (X)
36         CALL SUBB (X)
37         ...
38         CALL SUBY (X)
39         CYCLE
40     70 CALL SUBNEG (X) ! SUBNEG called only when X < 0.

```

1 This is not a block DO construct because it ends with a statement other than END DO or CONTINUE. The loop will  
2 continue to execute until an end-of-file condition or input/output error occurs.

3 Example 5:

```

4     SUM = 0.0
5     READ (IUN) N
6     DO 80, L = 1, N
7         READ (IUN) IQUAL, M, ARRAY (1:M)
8         IF (IQUAL < IQUAL_MIN) M = 0 ! Skip inner loop
9         DO 80 I = 1, M
10            CALL CALCULATE (ARRAY (I), RESULT)
11            IF (RESULT < 0.) CYCLE
12            SUM = SUM + RESULT
13            IF (SUM > SUM_MAX) GOTO 81
14 80    CONTINUE ! This CONTINUE is shared by both loops
15 81    CONTINUE

```

16 This example is similar to Example 1 above, except that the two loops are not block DO constructs because they share  
17 the CONTINUE statement with the label 80. The terminal construct of the outer DO is the entire inner DO construct.  
18 The inner loop is skipped by forcing M to zero. If SUM grows too large, both loops are terminated by branching to the  
19 CONTINUE statement labeled 81. The CYCLE statement in the inner loop is used to skip negative values of RESULT.

20 Example 6:

```

21     N = 0
22     DO 100 I = 1, 10
23         J = I
24         DO 100 K = 1, 5
25             L = K
26 100    N = N + 1 ! This statement executes 50 times

```

27 In this example, the two loops share an assignment statement. After execution of this program fragment, I = 11, J = 10,  
28 K = 6, L = 5, and N = 50.

29 Example 7:

```

30     N = 0
31     DO 200 I = 1, 10
32         J = I
33         DO 200 K = 5, 1 ! This inner loop is never executed
34             L = K
35 200    N = N + 1

```

36 This example is very similar to the previous one, except that the inner loop is never executed. After execution of this  
37 program fragment, I = 11, J = 10, K = 5, N = 0, and L is not defined by these statements.

### 38 C.5.4 Examples of invalid DO constructs

39 The following are all examples of invalid skeleton DO constructs:

40 Example 1:

```
1 DO I = 1, 10
2   ...
3 END DO LOOP    ! No matching construct name
```

4 Example 2:

```
5 LOOP: DO 1000 I = 1, 10    ! No matching construct name
6   ...
7 1000 CONTINUE
```

8 Example 3:

```
9 LOOP1: DO
10   ...
11 END DO LOOP2    ! Construct names don't match
```

12 Example 4:

```
13 DO I = 1, 10    ! Label required or ...
14   ...
15 1010 CONTINUE  ! ... END DO required
```

16 Example 5:

```
17 DO 1020 I = 1, 10
18   ...
19 1021 END DO      ! Labels don't match
```

20 Example 6:

```
21 FIRST: DO I = 1, 10
22   SECOND: DO J = 1, 5
23   ...
24   END DO FIRST    ! Improperly nested DOs
25 END DO SECOND
```

## 26 C.6 Section 9 notes

### 27 C.6.1 External files (9.2)

28 This standard accommodates, but does not require, file cataloging. To do this, several concepts are  
29 introduced.

1 **C.6.1.1 File connection (9.4)**

2 Before any input/output may be performed on a file, it shall be connected to a unit. The unit then serves  
 3 as a designator for that file as long as it is connected. To be connected does not imply that “buffers”  
 4 have or have not been allocated, that “file-control tables” have or have not been filled, or that any other  
 5 method of implementation has been used. Connection means that (barring some other fault) a READ  
 6 or WRITE statement may be executed on the unit, hence on the file. Without a connection, a READ  
 7 or WRITE statement shall not be executed.

8 **C.6.1.2 File existence (9.2.1)**

9 Totally independent of the connection state is the property of existence, this being a file property. The  
 10 processor “knows” of a set of files that exist at a given time for a given program. This set would include  
 11 tapes ready to read, files in a catalog, a keyboard, a printer, etc. The set may exclude files inaccessible  
 12 to the program because of security, because they are already in use by another program, etc. This  
 13 standard does not specify which files exist, hence wide latitude is available to a processor to implement  
 14 security, locks, privilege techniques, etc. Existence is a convenient concept to designate all of the files  
 15 that a program can potentially process.

16 All four combinations of connection and existence may occur:

Connect	Exist	Examples
Yes	Yes	A card reader loaded and ready to be read
Yes	No	A printer before the first line is written
No	Yes	A file named 'JOAN' in the catalog
No	No	A file on a reel of tape, not known to the processor

17 Means are provided to create, delete, connect, and disconnect files.

18 **C.6.1.3 File names (9.4.5.8)**

19 A file may have a name. The form of a file name is not specified. If a system does not have some form of  
 20 cataloging or tape labeling for at least some of its files, all file names will disappear at the termination  
 21 of execution. This is a valid implementation. Nowhere does this standard require names to survive for  
 22 any period of time longer than the execution time span of a program. Therefore, this standard does not  
 23 impose cataloging as a prerequisite. The naming feature is intended to allow use of a cataloging system  
 24 where one exists.

25 **C.6.1.4 File access (9.2.2)**

26 This standard does not address problems of security, protection, locking, and many other concepts that  
 27 may be part of the concept of “right of access”. Such concepts are considered to be in the province of  
 28 an operating system.

29 The OPEN and INQUIRE statements can be extended naturally to consider these things.

30 Possible access methods for a file are: sequential and direct. The processor may implement two different  
 31 types of files, each with its own access method. It might also implement one type of file with two different  
 32 access methods.

33 Direct access to files is of a simple and commonly available type, that is, fixed-length records. The key  
 34 is a positive integer.

## 1 C.6.2 Nonadvancing input/output (9.2.3.1)

2 Data transfer statements affect the positioning of an external file. In FORTRAN 77, if no error or end-of-  
3 file condition exists, the file is positioned after the record just read or written and that record becomes  
4 the preceding record. This standard contains the record positioning ADVANCE= specifier in a data  
5 transfer statement that provides the capability of maintaining a position within the current record from  
6 one formatted data transfer statement to the next data transfer statement. The value NO provides this  
7 capability. The value YES positions the file after the record just read or written. The default is YES.

8 The tab edit descriptor and the slash are still appropriate for use with this type of record access but the  
9 tab will not reposition before the left tab limit.

10 A BACKSPACE of a file that is positioned within a record causes the specified unit to be positioned  
11 before the current record.

12 If the last data transfer statement was WRITE and the file is positioned within a record, the file will be  
13 positioned implicitly after the current record before an ENDFILE record is written to the file, that is, a  
14 REWIND, BACKSPACE, or ENDFILE statement following a nonadvancing WRITE statement causes  
15 the file to be positioned at the end of the current output record before the endfile record is written to  
16 the file.

17 This standard provides a SIZE= specifier to be used with nonadvancing data transfer statements. The  
18 variable in the SIZE= specifier will contain the count of the number of characters that make up the  
19 sequence of values read by the data edit descriptors in this input statement.

20 The count is especially helpful if there is only one list item in the input list because it will contain the  
21 number of characters that were present for the item.

22 The EOR= specifier is provided to indicate when an end-of-record condition has been encountered during  
23 a nonadvancing data transfer statement. The end-of-record condition is not an error condition. If this  
24 specifier is present, the current input list item that required more characters than the record contained  
25 will be padded with blanks if PAD= 'YES' is in effect. This means that the input list item was successfully  
26 completed. The file will then be positioned after the current record. The IOSTAT= specifier, if present,  
27 will be defined with the value of the named constant IOSTAT\_EOR from the ISO\_FORTRAN\_ENV  
28 module and the data transfer statement will be terminated. Program execution will continue with the  
29 statement specified in the EOR= specifier. The EOR= specifier gives the capability of taking control  
30 of execution when the end-of-record has been found. The *do-variables* in *io-implied-dos* retain their  
31 last defined value and any remaining items in the *input-item-list* retain their definition status when an  
32 end-of-record condition occurs. The SIZE= specifier, if present, will contain the number of characters  
33 read with the data edit descriptors during this READ statement.

34 For nonadvancing input, the processor is not required to read partial records. The processor may read  
35 the entire record into an internal buffer and make successive portions of the record available to successive  
36 input statements.

37 In an implementation of nonadvancing input/output in which a nonadvancing write to a terminal device  
38 causes immediate display of the output, such a write can be used as a mechanism to output a prompt.  
39 In this case, the statement

```
40 WRITE (*, FMT='(A)', ADVANCE='NO') 'CONTINUE?(Y/N): '
```

41 would result in the prompt

```
42 CONTINUE?(Y/N) :
```

43 being displayed with no subsequent line feed.

1 The response, which might be read by a statement of the form

2 `READ (*, FMT='(A)') ANSWER`

3 can then be entered on the same line as the prompt as in

4 `CONTINUE?(Y/N): Y`

5 The standard does not require that an implementation of nonadvancing input/output operate in this  
6 manner. For example, an implementation of nonadvancing output in which the display of the output is  
7 deferred until the current record is complete is also standard conforming. Such an implementation will  
8 not, however, allow a prompting mechanism of this kind to operate.

### 9 **C.6.3 Asynchronous input/output**

10 Rather than limit support for asynchronous input/output to what has been traditionally provided by  
11 facilities such as BUFFERIN/BUFFEROUT, this standard builds upon existing Fortran syntax. This  
12 permits alternative approaches for implementing asynchronous input/output, and simplifies the task of  
13 adapting existing standard conforming programs to use asynchronous input/output.

14 Not all processors will actually perform input/output asynchronously, nor will every processor that does  
15 be able to handle data transfer statements with complicated input/output item lists in an asynchronous  
16 manner. Such processors can still be standard conforming. Hopefully, the documentation for each  
17 Fortran processor will describe when, if ever, input/output will be performed asynchronously.

18 This standard allows for at least two different conceptual models for asynchronous input/output.

19 Model 1: the processor will perform asynchronous input/output when the item list is simple (perhaps  
20 one contiguous named array) and the input/output is unformatted. The implementation cost is reduced,  
21 and this is the scenario most likely to be beneficial on traditional “big-iron” machines.

22 Model 2: The processor is free to do any of the following:

- 23 (1) on output, create a buffer inside the input/output library, completely formatted, and then  
24 start an asynchronous write of the buffer, and immediately return to the next statement in  
25 the program. The processor is free to wait for previously issued WRITES, or not, or  
26 (2) pass the input/output list addresses to another processor/process, which will process the  
27 list items independently of the processor that executes the user’s code. The addresses of the  
28 list items must be computed before the asynchronous READ/WRITE statement completes.  
29 There is still an ordering requirement on list item processing to handle things like READ  
30 (... ) N,(a(i),i=1,N).

31 The standard allows a user to issue a large number of asynchronous input/output requests, without  
32 waiting for any of them to complete, and then wait for any or all of them. It may be impossible, and  
33 undesirable to keep track of each of these input/output requests individually.

34 It is not necessary for all requests to be tracked by the runtime library. If an ID= specifier does not  
35 appear in on a READ or WRITE statement, the runtime is free to forget about this particular request  
36 once it has successfully completed. If it gets an ERR or END condition, the processor is free to report  
37 this during any input/output operation to that unit. When an ID= specifier is present, the processor’s  
38 runtime input/output library is required to keep track of any END or ERR conditions for that particular  
39 input/output request. However, if the input/output request succeeds without any exceptional conditions  
40 occurring, then the runtime can forget that ID= value if it wishes. Typically, a runtime might only keep  
41 track of the last request made, or perhaps a very few. Then, when a user WAITs for a particular request,  
42 either the library knows about it (and does the right thing with respect to error handling, etc.), or will  
43 assume it is one of those requests that successfully completed and was forgotten about (and will just  
44 return without signaling any end or error conditions). It is incumbent on the user to pass valid ID=

1 values. There is no requirement on the processor to detect invalid ID= values. There is of course,  
2 a processor dependent limit on how many outstanding input/output requests that generate an end or  
3 error condition can be handled before the processor runs out of memory to keep track of such conditions.  
4 The restrictions on the SIZE= variables are designed to allow the processor to update such variables at  
5 any time (after the request has been processed, but before the WAIT operation), and then forget about  
6 them. That's why there is no SIZE= specifier allowed in the various WAIT operations. Only exceptional  
7 conditions (errors or ends of files) are expected to be tracked by individual request by the runtime, and  
8 then only if an ID= specifier was present. The END= and EOR= specifiers have not been added to all  
9 statements that can be WAIT operations. Instead, the IOSTAT variable will have to be queried after a  
10 WAIT operation to handle this situation. This choice was made because we expect the WAIT statement  
11 to be the usual method of waiting for input/output to complete (and WAIT does support the END=  
12 and EOR= specifiers). This particular choice is philosophical, and was not based on significant technical  
13 difficulties.

14 Note that the requirement to set the IOSTAT variable correctly requires an implementation to remember  
15 which input/output requests got an EOR condition, so that a subsequent wait operation will return the  
16 correct IOSTAT value. This means there is a processor defined limit on the number of outstanding  
17 nonadvancing input/output requests that got an EOR condition (constrained by available memory to  
18 keep track of this information, similar to END/ERR conditions).

#### 19 **C.6.4 OPEN statement (9.4.5)**

20 A file may become connected to a unit either by preconnection or by execution of an OPEN statement.  
21 Preconnection is performed prior to the beginning of execution of a program by means external to For-  
22 tran. For example, it may be done by job control action or by processor-established defaults. Execution  
23 of an OPEN statement is not required to access preconnected files (9.4.4).

24 The OPEN statement provides a means to access existing files that are not preconnected. An OPEN  
25 statement may be used in either of two ways: with a file name (open-by-name) and without a file name  
26 (open-by-unit). A unit is given in either case. Open-by-name connects the specified file to the specified  
27 unit. Open-by-unit connects a processor-dependent default file to the specified unit. (The default file  
28 might or might not have a name.)

29 Therefore, there are three ways a file may become connected and hence processed: preconnection, open-  
30 by-name, and open-by-unit. Once a file is connected, there is no means in standard Fortran to determine  
31 how it became connected.

32 An OPEN statement may also be used to create a new file. In fact, any of the foregoing three connection  
33 methods may be performed on a file that does not exist. When a unit is preconnected, writing the first  
34 record creates the file. With the other two methods, execution of the OPEN statement creates the file.

35 When an OPEN statement is executed, the unit specified in the OPEN might or might not already be  
36 connected to a file. If it is already connected to a file (either through preconnection or by a prior OPEN),  
37 then omitting the FILE= specifier in the OPEN statement implies that the file is to remain connected  
38 to the unit. Such an OPEN statement may be used to change the values of the blank interpretation  
39 mode, decimal edit mode, pad mode, I/O rounding mode, delimiter mode, and sign mode.

40 If the value of the ACTION= specifier is WRITE, then READ statements shall not refer to this connec-  
41 tion. ACTION = 'WRITE' does not restrict positioning by a BACKSPACE statement or positioning  
42 specified by the POSITION= specifier with the value APPEND. However, a BACKSPACE statement  
43 or an OPEN statement containing POSITION = 'APPEND' may fail if the processor requires reading  
44 of the file to achieve the positioning.

45 The following examples illustrate these rules. In the first example, unit 10 is preconnected to a SCRATCH  
46 file; the OPEN statement changes the value of PAD= to YES.

```

1 CHARACTER (LEN = 20) CH1
2 WRITE (10, '(A)') 'THIS IS RECORD 1'
3 OPEN (UNIT = 10, STATUS = 'OLD', PAD = 'YES')
4 REWIND 10
5 READ (10, '(A20)') CH1 ! CH1 now has the value
6 ! 'THIS IS RECORD 1 '

```

7 In the next example, unit 12 is first connected to a file named FRED, with a status of OLD. The second  
8 OPEN statement then opens unit 12 again, retaining the connection to the file FRED, but changing the  
9 value of the DELIM= specifier to QUOTE.

```

10 CHARACTER (LEN = 25) CH2, CH3
11 OPEN (12, FILE = 'FRED', STATUS = 'OLD', DELIM = 'NONE')
12 CH2 = '''THIS STRING HAS QUOTES.'''
13 ! Quotes in string CH2
14 WRITE (12, *) CH2 ! Written with no delimiters
15 OPEN (12, DELIM = 'QUOTE') ! Now quote is the delimiter
16 REWIND 12
17 READ (12, *) CH3 ! CH3 now has the value
18 ! 'THIS STRING HAS QUOTES. '

```

19 The next example is invalid because it attempts to change the value of the STATUS= specifier.

```

20 OPEN (10, FILE = 'FRED', STATUS = 'OLD')
21 WRITE (10, *) A, B, C
22 OPEN (10, STATUS = 'SCRATCH') ! Attempts to make FRED
23 ! a SCRATCH file

```

24 The previous example could be made valid by closing the unit first, as in the next example.

```

25 OPEN (10, FILE = 'FRED', STATUS = 'OLD')
26 WRITE (10, *) A, B, C
27 CLOSE (10)
28 OPEN (10, STATUS = 'SCRATCH') ! Opens a different
29 ! SCRATCH file

```

### 30 **C.6.5 Connection properties (9.4.3)**

31 When a unit becomes connected to a file, either by execution of an OPEN statement or by preconnection,  
32 the following connection properties, among others, may be established:

- 33 (1) An access method, which is sequential, direct, or stream, is established for the connection  
34 (9.4.5.1).
- 35 (2) A form, which is formatted or unformatted, is established for a connection to a file that  
36 exists or is created by the connection. For a connection that results from execution of an  
37 OPEN statement, a default form (which depends on the access method, as described in

1           9.2.2) is established if no form is specified. For a preconnected file that exists, a form is  
 2 established by preconnection. For a preconnected file that does not exist, a form may be  
 3 established, or the establishment of a form may be delayed until the file is created (for  
 4 example, by execution of a formatted or unformatted WRITE statement) (9.4.5.9).

5           (3) A record length may be established. If the access method is direct, the connection establishes  
 6 a record length that specifies the length of each record of the file. An existing file with records  
 7 that are not all of equal length shall not be connected for direct access.

8           If the access method is sequential, records of varying lengths are permitted. In this case,  
 9 the record length established specifies the maximum length of a record in the file (9.4.5.12).

10 A processor has wide latitude in adapting these concepts and actions to its own cataloging and job  
 11 control conventions. Some processors may require job control action to specify the set of files that  
 12 exist or that will be created by a program. Some processors may require no job control action prior to  
 13 execution. This standard enables processors to perform dynamic open, close, or file creation operations,  
 14 but it does not require such capabilities of the processor.

15 The meaning of “open” in contexts other than Fortran may include such things as mounting a tape,  
 16 console messages, spooling, label checking, security checking, etc. These actions may occur upon job  
 17 control action external to Fortran, upon execution of an OPEN statement, or upon execution of the first  
 18 read or write of the file. The OPEN statement describes properties of the connection to the file and  
 19 might or might not cause physical activities to take place. It is a place for an implementation to define  
 20 properties of a file beyond those required in standard Fortran.

## 21 C.6.6 CLOSE statement (9.4.6)

22 Similarly, the actions of dismounting a tape, protection, etc. of a “close” may be implicit at the end of  
 23 a run. The CLOSE statement might or might not cause such actions to occur. This is another place to  
 24 extend file properties beyond those of standard Fortran. Note, however, that the execution of a CLOSE  
 25 statement on a unit followed by an OPEN statement on the same unit to the same file or to a different  
 26 file is a permissible sequence of events. The processor shall not deny this sequence solely because the  
 27 implementation chooses to do the physical act of closing the file at the termination of execution of the  
 28 program.

## 29 C.7 Section 10 notes

### 30 C.7.1 Number of records (10.3, 10.4, 10.7.2)

31 The number of records read by an explicitly formatted advancing input statement can be determined  
 32 from the following rule: a record is read at the beginning of the format scan (even if the input list is  
 33 empty), at each slash edit descriptor encountered in the format, and when a format rescan occurs at the  
 34 end of the format.

35 The number of records written by an explicitly formatted advancing output statement can be determined  
 36 from the following rule: a record is written when a slash edit descriptor is encountered in the format,  
 37 when a format rescan occurs at the end of the format, and at completion of execution of the output  
 38 statement (even if the output list is empty). Thus, the occurrence of  $n$  successive slashes between two  
 39 other edit descriptors causes  $n - 1$  blank lines if the records are printed. The occurrence of  $n$  slashes at  
 40 the beginning or end of a complete format specification causes  $n$  blank lines if the records are printed.  
 41 However, a complete format specification containing  $n$  slashes ( $n > 0$ ) and no other edit descriptors  
 42 causes  $n + 1$  blank lines if the records are printed. For example, the statements

```
43 PRINT 3
44 3 FORMAT (/)
```

1 will write two records that cause two blank lines if the records are printed.

## 2 **C.7.2 List-directed input (10.9.1)**

3 The following examples illustrate list-directed input. A blank character is represented by b.

4 Example 1:

5 Program:

```
6 J = 3
7 READ *, I
8 READ *, J
```

9 Sequential input file:

```
10 record 1: b1b,4bbbb
11 record 2: ,2bbbbbbb
```

12 Result: I = 1, J = 3.

13 Explanation: The second READ statement reads the second record. The initial comma in the record  
14 designates a null value; therefore, J is not redefined.

15 Example 2:

16 Program:

```
17 CHARACTER A *8, B *1
18 READ *, A, B
```

19 Sequential input file:

```
20 record 1: 'bbbbbbb'
21 record 2: 'QXY'b'Z'
```

22 Result: A = 'bbbbbbb', B = 'Q'

23 Explanation: In the first record, the rightmost apostrophe is interpreted as delimiting the constant (it  
24 cannot be the first of a pair of embedded apostrophes representing a single apostrophe because this  
25 would involve the prohibited “splitting” of the pair by the end of a record); therefore, A is assigned  
26 the character constant 'bbbbbbb'. The end of a record acts as a blank, which in this case is a value  
27 separator because it occurs between two constants.

## 28 **C.8 Section 11 notes**

### 29 **C.8.1 Main program and block data program unit (11.1, 11.3)**

30 The name of the main program or of a block data program unit has no explicit use within the Fortran  
31 language. It is available for documentation and for possible use by a processor.

1 A processor may implement an unnamed main program or unnamed block data program unit by assigning  
2 it a default name. However, this name shall not conflict with any other global name in a standard-  
3 conforming program. This might be done by making the default name one that is not permitted in a  
4 standard-conforming program (for example, by including a character not normally allowed in names)  
5 or by providing some external mechanism such that for any given program the default name can be  
6 changed to one that is otherwise unused.

## 7 **C.8.2 Dependent compilation (11.2)**

8 This standard, like its predecessors, is intended to permit the implementation of conforming processors  
9 in which a program can be broken into multiple units, each of which can be separately translated in  
10 preparation for execution. Such processors are commonly described as supporting separate compilation.  
11 There is an important difference between the way separate compilation can be implemented under this  
12 standard and the way it could be implemented under the FORTRAN 77 standard. Under the FORTRAN  
13 77 standard, any information required to translate a program unit was specified in that program unit.  
14 Each translation was thus totally independent of all others. Under this standard, a program unit can use  
15 information that was specified in a separate module and thus may be dependent on that module. The  
16 implementation of this dependency in a processor may be that the translation of a program unit may  
17 depend on the results of translating one or more modules. Processors implementing the dependency this  
18 way are commonly described as supporting dependent compilation.

19 The dependencies involved here are new only in the sense that the Fortran processor is now aware  
20 of them. The same information dependencies existed under the FORTRAN 77 standard, but it was the  
21 programmer's responsibility to transport the information necessary to resolve them by making redundant  
22 specifications of the information in multiple program units. The availability of separate but dependent  
23 compilation offers several potential advantages over the redundant textual specification of information:

- 24 (1) Specifying information at a single place in the program ensures that different program units  
25 using that information will be translated consistently. Redundant specification leaves the  
26 possibility that different information will erroneously be specified. Even if an INCLUDE line  
27 is used to ensure that the text of the specifications is identical in all involved program units,  
28 the presence of other specifications (for example, an IMPLICIT statement) may change the  
29 interpretation of that text.
- 30 (2) During the revision of a program, it is possible for a processor to assist in determining  
31 whether different program units have been translated using different (incompatible) versions  
32 of a module, although there is no requirement that a processor provide such assistance.  
33 Inconsistencies in redundant textual specification of information, on the other hand, tend  
34 to be much more difficult to detect.
- 35 (3) Putting information in a module provides a way of packaging it. Without modules, redun-  
36 dant specifications frequently shall be interleaved with other specifications in a program  
37 unit, making convenient packaging of such information difficult.
- 38 (4) Because a processor may be implemented such that the specifications in a module are  
39 translated once and then repeatedly referenced, there is the potential for greater efficiency  
40 than when the processor shall translate redundant specifications of information in multiple  
41 program units.

42 The exact meaning of the requirement that the public portions of a module be available at the time of  
43 reference is processor dependent. For example, a processor could consider a module to be available only  
44 after it has been compiled and require that if the module has been compiled separately, the result of  
45 that compilation shall be identified to the compiler when compiling program units that use it.

**1 C.8.2.1 USE statement and dependent compilation (11.2.1)**

2 Another benefit of the USE statement is its enhanced facilities for name management. If one needs to  
3 use only selected entities in a module, one can do so without having to worry about the names of all  
4 the other entities in that module. If one needs to use two different modules that happen to contain  
5 entities with the same name, there are several ways to deal with the conflict. If none of the entities with  
6 the same name are to be used, they can simply be ignored. If the name happens to refer to the same  
7 entity in both modules (for example, if both modules obtained it from a third module), then there is no  
8 confusion about what the name denotes and the name can be freely used. If the entities are different  
9 and one or both is to be used, the local renaming facility in the USE statement makes it possible to give  
10 those entities different names in the program unit containing the USE statements.

11 A benefit of using the ONLY specifier consistently, as compared to USE without it, is that the module  
12 from which each accessed entity is accessed is explicitly specified in each program unit. This means that  
13 one need not search other program units to find where each one is defined. This reduces maintenance  
14 costs.

15 A typical implementation of dependent but separate compilation may involve storing the result of trans-  
16 lating a module in a file (or file element) whose name is derived from the name of the module. Note,  
17 however, that the name of a module is limited only by the Fortran rules and not by the names allowed  
18 in the file system. Thus the processor may have to provide a mapping between Fortran names and file  
19 system names.

20 The result of translating a module could reasonably either contain only the information textually specified  
21 in the module (with “pointers” to information originally textually specified in other modules) or contain  
22 all information specified in the module (including copies of information originally specified in other  
23 modules). Although the former approach would appear to save on storage space, the latter approach  
24 can greatly simplify the logic necessary to process a USE statement and can avoid the necessity of  
25 imposing a limit on the logical “nesting” of modules via the USE statement.

26 Variables declared in a module retain their definition status on much the same basis as variables in  
27 a common block. That is, saved variables retain their definition status throughout the execution of a  
28 program, while variables that are not saved retain their definition status only during the execution of  
29 scoping units that reference the module. In some cases, it may be appropriate to put a USE statement  
30 such as

31 `USE MY_MODULE, ONLY:`

32 in a scoping unit in order to assure that other procedures that it references can communicate through  
33 the module. In such a case, the scoping unit would not access any entities from the module, but the  
34 variables not saved in the module would retain their definition status throughout the execution of the  
35 scoping unit.

36 There is an increased potential for undetected errors in a scoping unit that uses both implicit typing  
37 and the USE statement. For example, in the program fragment

```
38 SUBROUTINE SUB  
39     USE MY_MODULE  
40     IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)  
41     X = F (B)  
42     A = G (X) + H (X + 1)  
43 END SUBROUTINE SUB
```

44 X could be either an implicitly typed real variable or a variable obtained from the module MY\_MODULE

1 and might change from one to the other because of changes in MY\_MODULE unrelated to the action  
2 performed by SUB. Logic errors resulting from this kind of situation can be extremely difficult to locate.  
3 Thus, the use of these features together is discouraged.

#### 4 **C.8.2.2 Accessibility attributes**

5 The PUBLIC and PRIVATE attributes, which can be declared only in modules, divide the entities in a  
6 module into those that are actually relevant to a scoping unit referencing the module and those that are  
7 not. This information may be used to improve the performance of a Fortran processor. For example,  
8 it may be possible to discard much of the information about the private entities once a module has  
9 been translated, thus saving on both storage and the time to search it. Similarly, it may be possible  
10 to recognize that two versions of a module differ only in the private entities they contain and avoid  
11 retranslating program units that use that module when switching from one version of the module to the  
12 other.

### 13 **C.8.3 Examples of the use of modules**

#### 14 **C.8.3.1 Identical common blocks**

15 A common block and all its associated specification statements may be placed in a module named, for  
16 example, MY\_COMMON and accessed by a USE statement of the form

```
17 USE MY_COMMON
```

18 that accesses the whole module without any renaming. This ensures that all instances of the common  
19 block are identical. Module MY\_COMMON could contain more than one common block.

#### 20 **C.8.3.2 Global data**

21 A module may contain only data objects, for example:

```
22 MODULE DATA_MODULE  
23   SAVE  
24   REAL A (10), B, C (20,20)  
25   INTEGER :: I=0  
26   INTEGER, PARAMETER :: J=10  
27   COMPLEX D (J,J)  
28 END MODULE DATA_MODULE
```

29 Data objects made global in this manner may have any combination of data types.

30 Access to some of these may be made by a USE statement with the ONLY option, such as:

```
31 USE DATA_MODULE, ONLY: A, B, D
```

32 and access to all of them may be made by the following USE statement:

```
33 USE DATA_MODULE
```

34 Access to all of them with some renaming to avoid name conflicts may be made by:

```
35 USE DATA_MODULE, AMODULE => A, DMODULE => D
```

### 1 C.8.3.3 Derived types

2 A derived type may be defined in a module and accessed in a number of program units. For example:

```
3 MODULE SPARSE
4     TYPE NONZERO
5         REAL A
6         INTEGER I, J
7     END TYPE NONZERO
8 END MODULE SPARSE
```

9 defines a type consisting of a real component and two integer components for holding the numerical  
10 value of a nonzero matrix element and its row and column indices.

### 11 C.8.3.4 Global allocatable arrays

12 Many programs need large global allocatable arrays whose sizes are not known before program execution.  
13 A simple form for such a program is:

```
14 PROGRAM GLOBAL_WORK
15     CALL CONFIGURE_ARRAYS      ! Perform the appropriate allocations
16     CALL COMPUTE              ! Use the arrays in computations
17 END PROGRAM GLOBAL_WORK
18 MODULE WORK_ARRAYS          ! An example set of work arrays
19     INTEGER N
20     REAL, ALLOCATABLE, SAVE :: A (:), B (:, :), C (:, :, :)
21 END MODULE WORK_ARRAYS
22 SUBROUTINE CONFIGURE_ARRAYS  ! Process to set up work arrays
23     USE WORK_ARRAYS
24     READ (*, *) N
25     ALLOCATE (A (N), B (N, N), C (N, N, 2 * N))
26 END SUBROUTINE CONFIGURE_ARRAYS
27 SUBROUTINE COMPUTE
28     USE WORK_ARRAYS
29     ... ! Computations involving arrays A, B, and C
30 END SUBROUTINE COMPUTE
```

31 Typically, many subprograms need access to the work arrays, and all such subprograms would contain  
32 the statement

```
33 USE WORK_ARRAYS
```

### 34 C.8.3.5 Procedure libraries

35 Interface bodies for external procedures in a library may be gathered into a module. This permits the  
36 use of argument keywords and optional arguments, and allows static checking of the references. Different  
37 versions may be constructed for different applications, using argument keywords in common use in each  
38 application.

1 An example is the following library module:

```

2 MODULE LIBRARY_LLS
3   INTERFACE
4     SUBROUTINE LLS (X, A, F, FLAG)
5       REAL X (:, :)
6       ! The SIZE in the next statement is an intrinsic function
7       REAL, DIMENSION (SIZE (X, 2)) :: A, F
8       INTEGER FLAG
9     END SUBROUTINE LLS
10    ...
11  END INTERFACE
12  ...
13 END MODULE LIBRARY_LLS

```

14 This module allows the subroutine LLS to be invoked:

```

15 USE LIBRARY_LLS
16 ...
17 CALL LLS (X = ABC, A = D, F = XX, FLAG = IFLAG)
18 ...

```

### 19 C.8.3.6 Operator extensions

20 In order to extend an intrinsic operator symbol to have an additional meaning, an interface block  
 21 specifying that operator symbol in the OPERATOR option of the INTERFACE statement may be  
 22 placed in a module.

23 For example, // may be extended to perform concatenation of two derived-type objects serving as varying  
 24 length character strings and + may be extended to specify matrix addition for type MATRIX or interval  
 25 arithmetic addition for type INTERVAL.

26 A module might contain several such interface blocks. An operator may be defined by an external  
 27 function (either in Fortran or some other language) and its procedure interface placed in the module.

### 28 C.8.3.7 Data abstraction

29 In addition to providing a portable means of avoiding the redundant specification of information in  
 30 multiple program units, a module provides a convenient means of “packaging” related entities, such as  
 31 the definitions of the representation and operations of an abstract data type. The following example  
 32 of a module defines a data abstraction for a SET type where the elements of each set are of type  
 33 integer. The standard set operations of UNION, INTERSECTION, and DIFFERENCE are provided.  
 34 The CARDINALITY function returns the cardinality of (number of elements in) its set argument.  
 35 Two functions returning logical values are included, ELEMENT and SUBSET. ELEMENT defines the  
 36 operator .IN. and SUBSET extends the operator <=. ELEMENT determines if a given scalar integer  
 37 value is an element of a given set, and SUBSET determines if a given set is a subset of another given  
 38 set. (Two sets may be checked for equality by comparing cardinality and checking that one is a subset  
 39 of the other, or checking to see if each is a subset of the other.)

1 The transfer function SETF converts a vector of integer values to the corresponding set, with duplicate  
2 values removed. Thus, a vector of constant values can be used as set constants. An inverse transfer  
3 function VECTOR returns the elements of a set as a vector of values in ascending order. In this SET  
4 implementation, set data objects have a maximum cardinality of 200.

```
5 MODULE INTEGER_SETS
6 ! This module is intended to illustrate use of the module facility
7 ! to define a new type, along with suitable operators.
8
9 INTEGER, PARAMETER :: MAX_SET_CARD = 200
10
11 TYPE SET                                ! Define SET type
12     PRIVATE
13     INTEGER CARD
14     INTEGER ELEMENT (MAX_SET_CARD)
15 END TYPE SET
16
17 INTERFACE OPERATOR (.IN.)
18     MODULE PROCEDURE ELEMENT
19 END INTERFACE OPERATOR (.IN.)
20
21 INTERFACE OPERATOR (<=)
22     MODULE PROCEDURE SUBSET
23 END INTERFACE OPERATOR (<=)
24
25 INTERFACE OPERATOR (+)
26     MODULE PROCEDURE UNION
27 END INTERFACE OPERATOR (+)
28
29 INTERFACE OPERATOR (-)
30     MODULE PROCEDURE DIFFERENCE
31 END INTERFACE OPERATOR (-)
32
33 INTERFACE OPERATOR (*)
34     MODULE PROCEDURE INTERSECTION
35 END INTERFACE OPERATOR (*)
36
37 CONTAINS
38
39 INTEGER FUNCTION CARDINALITY (A)        ! Returns cardinality of set A
40     TYPE (SET), INTENT (IN) :: A
41     CARDINALITY = A % CARD
42 END FUNCTION CARDINALITY
43
44 LOGICAL FUNCTION ELEMENT (X, A)        ! Determines if
```

```

1   INTEGER, INTENT(IN) :: X           ! element X is in set A
2   TYPE (SET), INTENT(IN) :: A
3   ELEMENT = ANY (A % ELEMENT (1 : A % CARD) == X)
4   END FUNCTION ELEMENT
5
6   FUNCTION UNION (A, B)              ! Union of sets A and B
7   TYPE (SET) UNION
8   TYPE (SET), INTENT(IN) :: A, B
9   INTEGER J
10  UNION = A
11  DO J = 1, B % CARD
12     IF (.NOT. (B % ELEMENT (J) .IN. A)) THEN
13         IF (UNION % CARD < MAX_SET_CARD) THEN
14             UNION % CARD = UNION % CARD + 1
15             UNION % ELEMENT (UNION % CARD) = &
16                 B % ELEMENT (J)
17         ELSE
18             ! Maximum set size exceeded . . .
19         END IF
20     END IF
21  END DO
22  END FUNCTION UNION
23
24  FUNCTION DIFFERENCE (A, B)         ! Difference of sets A and B
25  TYPE (SET) DIFFERENCE
26  TYPE (SET), INTENT(IN) :: A, B
27  INTEGER J, X
28  DIFFERENCE % CARD = 0             ! The empty set
29  DO J = 1, A % CARD
30     X = A % ELEMENT (J)
31     IF (.NOT. (X .IN. B)) DIFFERENCE = DIFFERENCE + SET (1, X)
32  END DO
33  END FUNCTION DIFFERENCE
34
35  FUNCTION INTERSECTION (A, B)      ! Intersection of sets A and B
36  TYPE (SET) INTERSECTION
37  TYPE (SET), INTENT(IN) :: A, B
38  INTERSECTION = A - (A - B)
39  END FUNCTION INTERSECTION
40
41  LOGICAL FUNCTION SUBSET (A, B)    ! Determines if set A is
42  TYPE (SET), INTENT(IN) :: A, B  ! a subset of set B
43  INTEGER I
44  SUBSET = A % CARD <= B % CARD
45  IF (.NOT. SUBSET) RETURN         ! For efficiency

```

```

1   DO I = 1, A % CARD
2       SUBSET = SUBSET .AND. (A % ELEMENT (I) .IN. B)
3   END DO
4 END FUNCTION SUBSET
5
6 TYPE (SET) FUNCTION SETF (V)      ! Transfer function between a vector
7   INTEGER V (:)                  ! of elements and a set of elements
8   INTEGER J                       ! removing duplicate elements
9   SETF % CARD = 0
10  DO J = 1, SIZE (V)
11      IF (.NOT. (V (J) .IN. SETF)) THEN
12          IF (SETF % CARD < MAX_SET_CARD) THEN
13              SETF % CARD = SETF % CARD + 1
14              SETF % ELEMENT (SETF % CARD) = V (J)
15          ELSE
16              ! Maximum set size exceeded . . .
17          END IF
18      END IF
19  END DO
20 END FUNCTION SETF
21
22 FUNCTION VECTOR (A)              ! Transfer the values of set A
23   TYPE (SET), INTENT (IN) :: A ! into a vector in ascending order
24   INTEGER, POINTER :: VECTOR (:)
25   INTEGER I, J, K
26   ALLOCATE (VECTOR (A % CARD))
27   VECTOR = A % ELEMENT (1 : A % CARD)
28   DO I = 1, A % CARD - 1         ! Use a better sort if
29       DO J = I + 1, A % CARD     ! A % CARD is large
30           IF (VECTOR (I) > VECTOR (J)) THEN
31               K = VECTOR (J); VECTOR (J) = VECTOR (I); VECTOR (I) = K
32           END IF
33       END DO
34   END DO
35 END FUNCTION VECTOR
36
37 END MODULE INTEGER_SETS

38 Examples of using INTEGER_SETS (A, B, and C are variables of type SET; X
39 is an integer variable):
40 ! Check to see if A has more than 10 elements
41 IF (CARDINALITY (A) > 10) ...
42
43 ! Check for X an element of A but not of B
44 IF (X .IN. (A - B)) ...

```

```

1
2 ! C is the union of A and the result of B intersected
3 ! with the integers 1 to 100
4 C = A + B * SETF ((/ (I, I = 1, 100) /))
5
6 ! Does A have any even numbers in the range 1:100?
7 IF (CARDINALITY (A * SETF ((/ (I, I = 2, 100, 2) /))) > 0) ...
8
9 PRINT *, VECTOR (B) ! Print out the elements of set B, in ascending order

```

### 10 C.8.3.8 Public entities renamed

11 At times it may be necessary to rename entities that are accessed with USE statements. Care should be  
 12 taken if the referenced modules also contain USE statements.

13 The following example illustrates renaming features of the USE statement.

```

14 MODULE J; REAL JX, JY, JZ; END MODULE J
15 MODULE K
16     USE J, ONLY : KX => JX, KY => JY
17     ! KX and KY are local names to module K
18     REAL KZ      ! KZ is local name to module K
19     REAL JZ      ! JZ is local name to module K
20 END MODULE K
21 PROGRAM RENAME
22     USE J; USE K
23     ! Module J's entity JX is accessible under names JX and KX
24     ! Module J's entity JY is accessible under names JY and KY
25     ! Module K's entity KZ is accessible under name KZ
26     ! Module J's entity JZ and K's entity JZ are different entities
27     ! and shall not be referenced
28     ...
29 END PROGRAM RENAME

```

## 30 C.9 Section 12 notes

### 31 C.9.1 Portability problems with external procedures (12.3.2.2)

32 There is a potential portability problem in a scoping unit that references an external procedure without  
 33 explicitly declaring it to have the EXTERNAL attribute (5.1.2.6). On a different processor, the name  
 34 of that procedure may be the name of a nonstandard intrinsic procedure and the processor would  
 35 be permitted to interpret those procedure references as references to that intrinsic procedure. (On that  
 36 processor, the program would also be viewed as not conforming to the standard because of the references  
 37 to the nonstandard intrinsic procedure.) Declaration of the EXTERNAL attribute causes the references  
 38 to be to the external procedure regardless of the availability of an intrinsic procedure with the same  
 39 name. Note that declaration of the type of a procedure is not enough to make it external, even if the  
 40 type is inconsistent with the type of the result of an intrinsic of the same name.

## 1 C.9.2 Procedures defined by means other than Fortran (12.5.3)

2 A processor is not required to provide any means other than Fortran for defining external procedures.  
3 Among the means that might be supported are the machine assembly language, other high level lan-  
4 guages, the Fortran language extended with nonstandard features, and the Fortran language as supported  
5 by another Fortran processor (for example, a previously existing FORTRAN 77 processor).

6 Procedures defined by means other than Fortran are considered external procedures because their def-  
7 initions are not in a Fortran program unit and because they are referenced using global names. The  
8 use of the term external should not be construed as any kind of restriction on the way in which these  
9 procedures may be defined. For example, if the means other than Fortran has its own facilities for  
10 internal and external procedures, it is permissible to use them. If the means other than Fortran can  
11 create an “internal” procedure with a global name, it is permissible for such an “internal” procedure  
12 to be considered by Fortran to be an external procedure. The means other than Fortran for defining  
13 external procedures, including any restrictions on the structure for organization of those procedures, are  
14 entirely processor dependent.

15 A Fortran processor may limit its support of procedures defined by means other than Fortran such that  
16 these procedures may affect entities in the Fortran environment only on the same basis as procedures  
17 written in Fortran. For example, it might prohibit the value of a local variable from being changed by  
18 a procedure reference unless that variable were one of the arguments to the procedure.

## 19 C.9.3 Procedure interfaces (12.3)

20 In FORTRAN 77, the interface to an external procedure was always deduced from the form of references  
21 to that procedure and any declarations of the procedure name in the referencing program unit. In this  
22 standard, features such as argument keywords and optional arguments make it impossible to deduce  
23 sufficient information about the dummy arguments from the nature of the actual arguments to be  
24 associated with them, and features such as array function results and pointer function results make  
25 necessary extensions to the declaration of a procedure that cannot be done in a way that would be  
26 analogous with the handling of such declarations in FORTRAN 77. Hence, mechanisms are provided  
27 through which all the information about a procedure’s interface may be made available in a scoping  
28 unit that references it. A procedure whose interface shall be deduced as in FORTRAN 77 is described  
29 as having an implicit interface. A procedure whose interface is fully known is described as having an  
30 explicit interface.

31 A scoping unit is allowed to contain an interface body for a procedure that does not exist in the program,  
32 provided the procedure described is never referenced or used in any other way. The purpose of this rule is  
33 to allow implementations in which the use of a module providing interface bodies describing the interface  
34 of every routine in a library would not automatically cause each of those library routines to be a part of  
35 the program referencing the module. Instead, only those library procedures actually referenced would  
36 be a part of the program. (In implementation terms, the mere presence of an interface body would not  
37 generate an external reference in such an implementation.)

## 38 C.9.4 Abstract interfaces (12.3) and procedure pointer components (4.5)

39 This is an example of a library module providing lists of callbacks that the user may register and invoke.

```
40 MODULE callback_list_module
41     !
42     ! Type for users to extend with their own data, if they so desire
43     !
44     TYPE callback_data
```

```

1  END TYPE
2  !
3  ! Abstract interface for the callback procedures
4  !
5  ABSTRACT INTERFACE
6    SUBROUTINE callback_procedure(data)
7      IMPORT callback_data
8      CLASS(callback_data),OPTIONAL :: data
9    END SUBROUTINE
10 END INTERFACE
11 !
12 ! The callback list type.
13 !
14 TYPE callback_list
15   PRIVATE
16   CLASS(callback_record),POINTER :: first => NULL()
17 END TYPE
18 !
19 ! Internal: each callback registration creates one of these
20 !
21 TYPE,PRIVATE :: callback_record
22   PROCEDURE(callback_procedure),POINTER,NOPASS :: proc
23   CLASS(callback_record),POINTER :: next
24   CLASS(callback_data),POINTER :: data => NULL();
25 END TYPE
26 PRIVATE invoke,forward_invoke
27 CONTAINS
28 !
29 ! Register a callback procedure with optional data
30 !
31 SUBROUTINE register_callback(list, entry, data)
32   TYPE(callback_list),INTENT(INOUT) :: list
33   PROCEDURE(callback_procedure) :: entry
34   CLASS(callback_data),OPTIONAL :: data
35   TYPE(callback_record),POINTER :: new,last
36   ALLOCATE(new)
37   new%proc => entry
38   IF (PRESENT(data)) ALLOCATE(new%data,SOURCE=data)
39   new%next => list%first
40   list%first => new
41 END SUBROUTINE
42 !
43 ! Internal: Invoke a single callback and destroy its record
44 !
45 SUBROUTINE invoke(callback)

```

```

1     TYPE(callback_record),POINTER :: callback
2     IF (ASSOCIATED(callback%data) THEN
3         CALL callback%proc(list%first%data)
4         DEALLOCATE(callback%data)
5     ELSE
6         CALL callback%proc
7     END IF
8     DEALLOCATE(callback)
9     END SUBROUTINE
10    !
11    ! Call the procedures in reverse order of registration
12    !
13    SUBROUTINE invoke_callback_reverse(list)
14        TYPE(callback_list),INTENT(INOUT) :: list
15        TYPE(callback_record),POINTER :: next,current
16        current => list%first
17        NULLIFY(list%first)
18        DO WHILE (ASSOCIATED(current))
19            next => current%next
20            CALL invoke(current)
21            current => next
22        END DO
23    END SUBROUTINE
24    !
25    ! Internal: Forward mode invocation
26    !
27    RECURSIVE SUBROUTINE forward_invoke(callback)
28        IF (ASSOCIATED(callback%next)) CALL forward_invoke(callback%next)
29        CALL invoke(callback)
30    END SUBROUTINE
31    !
32    ! Call the procedures in forward order of registration
33    !
34    SUBROUTINE invoke_callback_forward(list)
35        TYPE(callback_list),INTENT(INOUT) :: list
36        IF (ASSOCIATED(list%first)) CALL forward_invoke(list%first)
37    END SUBROUTINE
38    END

```

### 39 C.9.5 Argument association and evaluation (12.4.1.2)

40 There is a significant difference between the argument association allowed in this standard and that  
41 supported by FORTRAN 77 and FORTRAN 66. In FORTRAN 77 and 66, actual arguments were limited  
42 to consecutive storage units. With the exception of assumed length character dummy arguments, the  
43 structure imposed on that sequence of storage units was always determined in the invoked procedure and  
44 not taken from the actual argument. Thus it was possible to implement FORTRAN 66 and FORTRAN 77

1 argument association by supplying only the location of the first storage unit (except for character argu-  
2 ments, where the length would also have to be supplied). However, this standard allows arguments that  
3 do not reside in consecutive storage locations (for example, an array section), and dummy arguments that  
4 assume additional structural information from the actual argument (for example, assumed-shape dummy  
5 arguments). Thus, the mechanism to implement the argument association allowed in this standard needs  
6 to be more general.

7 Because there are practical advantages to a processor that can support references to and from pro-  
8 cedures defined by a FORTRAN 77 processor, requirements for explicit interfaces make it possible to  
9 determine whether a simple (FORTRAN 66/FORTRAN 77) argument association implementation mecha-  
10 nism is sufficient or whether the more general mechanism is necessary (12.3.1.1). Thus a processor can  
11 be implemented whose procedures expect the simple mechanism to be used whenever the procedure's  
12 interface is one that uses only FORTRAN 77 features and that expects the more general mechanism  
13 otherwise (for example, if there are assumed-shape or optional arguments). At the point of reference,  
14 the appropriate mechanism can be determined from the interface if it is explicit and can be assumed to  
15 be the simple mechanism if it is not. Note that if the simple mechanism is determined to be what the  
16 procedure expects, it may be necessary for the processor to allocate consecutive temporary storage for  
17 the actual argument, copy the actual argument to the temporary storage, reference the procedure using  
18 the temporary storage in place of the actual argument, copy the contents of temporary storage back to  
19 the actual argument, and deallocate the temporary storage.

20 While this is the particular implementation method these rules were designed to support, it is not  
21 the only one possible. For example, on some processors, it may be possible to implement the general  
22 argument association in such a way that the information involved in FORTRAN 77 argument association  
23 may be found in the same places and the "extra" information is placed so it does not disturb a procedure  
24 expecting only FORTRAN 77 argument association. With such an implementation, argument association  
25 could be translated without regard to whether the interface is explicit or implicit.

26 The provisions for expression evaluation give the processor considerable flexibility for obtaining expres-  
27 sion values in the most efficient way possible. This includes not evaluating or only partially evaluating  
28 an operand, for example, if the value of the expression can be determined otherwise (7.1.8.1). This  
29 flexibility applies to function argument evaluation, including the order of argument evaluation, delay-  
30 ing argument evaluation, and omitting argument evaluation. A processor may delay the evaluation of  
31 an argument in a procedure reference until the execution of the procedure refers to the value of that  
32 argument, provided delaying the evaluation of the argument does not otherwise affect the results of  
33 the program. The processor may, with similar restrictions, entirely omit the evaluation of an argument  
34 not referenced in the execution of the procedure. This gives processors latitude for optimization (for  
35 example, for parallel processing).

### 36 **C.9.6 Pointers and targets as arguments (12.4.1.2)**

37 If a dummy argument is declared to be a pointer, it may be matched only by an actual argument that  
38 also is a pointer, and the characteristics of both arguments shall agree. A model for such an association is  
39 that descriptor values of the actual pointer are copied to the dummy pointer. If the actual pointer has an  
40 associated target, this target becomes accessible via the dummy pointer. If the dummy pointer becomes  
41 associated with a different target during execution of the procedure, this target will be accessible via the  
42 actual pointer after the procedure completes execution. If the dummy pointer becomes associated with  
43 a local target that ceases to exist when the procedure completes, the actual pointer will be left dangling  
44 in an undefined state. Such dangling pointers shall not be used.

45 When execution of a procedure completes, any pointer that remains defined and that is associated with  
46 a dummy argument that has the TARGET attribute and is either a scalar or an assumed-shape array,  
47 remains associated with the corresponding actual argument if the actual argument has the TARGET  
48 attribute and is not an array section with a vector subscript.

```

1 REAL, POINTER      :: PBEST
2 REAL, TARGET      :: B (10000)
3 CALL BEST (PBEST, B)      ! Upon return PBEST is associated
4   ...                  ! with the ‘best’ element of B
5 CONTAINS
6   SUBROUTINE BEST (P, A)
7     REAL, POINTER, INTENT (OUT) :: P
8     REAL, TARGET, INTENT (IN)  :: A (:)
9     ...                      ! Find the ‘best’ element A(I)
10    P => A (I)
11  RETURN
12  END SUBROUTINE BEST
13 END

```

14 When procedure BEST completes, the pointer PBEST is associated with an element of B.

15 An actual argument without the TARGET attribute can become associated with a dummy argument  
16 with the TARGET attribute. This permits pointers to become associated with the dummy argument  
17 during execution of the procedure that contains the dummy argument. For example:

```

18 INTEGER LARGE(100,100)
19 CALL SUB (LARGE)
20   ...
21 CALL SUB ()
22 CONTAINS
23   SUBROUTINE SUB(ARG)
24     INTEGER, TARGET, OPTIONAL :: ARG(100,100)
25     INTEGER, POINTER, DIMENSION(:, :) :: PARG
26     IF (PRESENT(ARG)) THEN
27       PARG => ARG
28     ELSE
29       ALLOCATE (PARG(100,100))
30       PARG = 0
31     ENDIF
32     ... ! Code with lots of references to PARG
33     IF (.NOT. PRESENT(ARG)) DEALLOCATE(PARG)
34   END SUBROUTINE SUB
35 END

```

36 Within subroutine SUB the pointer PARG is either associated with the dummy argument ARG or it is  
37 associated with an allocated target. The bulk of the code can reference PARG without further calls to  
38 the PRESENT intrinsic.

### 39 **C.9.7 Polymorphic Argument Association (12.4.1.3)**

40 The following example illustrates polymorphic argument association rules using the derived types defined  
41 in Note 4.54.

```

1  TYPE(POINT) :: T2
2  TYPE(COLOR_POINT) :: T3
3  CLASS(POINT) :: P2
4  CLASS(COLOR_POINT) :: P3
5  ! Dummy argument is polymorphic and actual argument is of fixed type
6  SUBROUTINE SUB2 ( X2 ); CLASS(POINT) :: X2; ...
7  SUBROUTINE SUB3 ( X3 ); CLASS(COLOR_POINT) :: X3; ...
8
9  CALL SUB2 ( T2 ) ! Valid -- The declared type of T2 is the same as the
10                   !           declared type of X2.
11  CALL SUB2 ( T3 ) ! Valid -- The declared type of T3 is extended from
12                   !           the declared type of X2.
13  CALL SUB3 ( T2 ) ! Invalid -- The declared type of T2 is neither the
14                   !           same as nor extended from the declared type
15                   !           type of X3.
16  CALL SUB3 ( T3 ) ! Valid -- The declared type of T3 is the same as the
17                   !           declared type of X3.
18  ! Actual argument is polymorphic and dummy argument is of fixed type
19  SUBROUTINE TUB2 ( D2 ); TYPE(POINT) :: D2; ...
20  SUBROUTINE TUB3 ( D3 ); TYPE(COLOR_POINT) :: D3; ...
21
22  CALL TUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
23                   !           declared type of D2.
24  CALL TUB2 ( P3 ) ! Invalid -- The declared type of P3 differs from the
25                   !           declared type of D2.
26  CALL TUB2 ( P3%POINT ) ! Valid alternative to the above
27  CALL TUB3 ( P2 ) ! Invalid -- The declared type of P2 differs from the
28                   !           declared type of D3.
29  SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
30  CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the same
31     CALL TUB3 ( P2 )      ! as the declared type of D3, or a type
32                           ! extended therefrom.
33  CLASS DEFAULT
34                           ! Cannot work if not.
35  END SELECT
36  CALL TUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
37                   !           declared type of D3.
38  ! Both the actual and dummy arguments are of polymorphic type.
39  CALL SUB2 ( P2 ) ! Valid -- The declared type of P2 is the same as the
40                   !           declared type of X2.
41  CALL SUB2 ( P3 ) ! Valid -- The declared type of P3 is extended from
42                   !           the declared type of X2.
43  CALL SUB3 ( P2 ) ! Invalid -- The declared type of P2 is neither the
44                   !           same as nor extended from the declared
45                   !           type of X3.

```

```

1  SELECT TYPE ( P2 ) ! Valid conditional alternative to the above
2  CLASS IS ( COLOR_POINT ) ! Works if the dynamic type of P2 is the
3  CALL SUB3 ( P2 )      ! same as the declared type of X3, or a
4                        ! type extended therefrom.
5  CLASS DEFAULT
6                        ! Cannot work if not.
7  END SELECT
8  CALL SUB3 ( P3 ) ! Valid -- The declared type of P3 is the same as the
9                    ! declared type of X3.

```

## 10 C.10 Section 15 notes

### 11 C.10.1 Runtime environments

12 This standard allows programs to contain procedures defined by means other than Fortran. That raises  
 13 the issues of initialization of and interaction between the runtime environments involved.

14 Implementations are free to solve these issues as they see fit, provided that:

- 15 (1) Heap allocation/deallocation (e.g., (DE)ALLOCATE in a Fortran subprogram and mal-  
 16 loc/free in a C function) can be performed without interference.
- 17 (2) I/O to and from external files can be performed without interference, as long as procedures  
 18 defined by different means do not do I/O to/from the same external file.
- 19 (3) I/O preconnections exist as required by the respective standards.
- 20 (4) Initialized data is initialized according to the respective standards.

### 21 C.10.2 Examples of Interoperation between Fortran and C Functions

22 The following examples illustrate the interoperation of Fortran and C functions. Two examples are  
 23 shown: one of Fortran calling C, and one of C calling Fortran. In each of the examples, the correspon-  
 24 dences of Fortran actual arguments, Fortran dummy arguments, and C formal parameters are described.

#### 25 C.10.2.1 Example of Fortran calling C

26 C Function Prototype:

```

27     int C_Library_Function(void* sendbuf, int sendcount,
28                           int *recvcounts);

```

29 Fortran Modules:

```

30     MODULE FTN_C_1
31     USE, INTRINSIC :: ISO_C_BINDING
32     END MODULE FTN_C_1
33
34     MODULE FTN_C_2
35     INTERFACE

```

```

1      INTEGER (C_INT) FUNCTION C_LIBRARY_FUNCTION &
2      (SENDBUF, SENDCOUNT, RECVCOUNTS),    &
3      BIND(C, NAME='C_Library_Function')
4      USE FTN_C_1
5      IMPLICIT NONE
6      TYPE (C_PTR), VALUE :: SENDBUF
7      INTEGER (C_INT), VALUE :: SENDCOUNT
8      TYPE (C_PTR), VALUE :: RECVCOUNTS
9      END FUNCTION C_LIBRARY_FUNCTION
10     END INTERFACE
11     END MODULE FTN_C_2

```

12 The module FTN\_C.2 contains the declaration of the Fortran dummy arguments, which correspond to  
13 the C formal parameters. The intrinsic module ISO\_C\_BINDING is referenced in the module FTN\_C.1.  
14 The NAME specifier is used in the BIND attribute in order to handle the case-sensitive name change  
15 between Fortran and C from 'C\_LIBRARY\_FUNCTION' to 'C\_Library\_Function'. See also Note [12.39](#).

16 The first C formal parameter is the pointer to void `sendbuf`, which corresponds to the Fortran dummy  
17 argument `SENDBUF`, which has the type `C_PTR` and the `VALUE` attribute.

18 The second C formal parameter is the int `sendcount`, which corresponds to the Fortran dummy argument  
19 `SENDCOUNT`, which has the type `INTEGER(C_INT)` and the `VALUE` attribute.

20 The third C formal parameter is the pointer to int `recvcunts`, which corresponds to the Fortran dummy  
21 argument `RECVCOUNTS`, which has the type `C_PTR` and the `VALUE` attribute.

22 Fortran Calling Sequence:

```

23     USE, INTRINSIC :: ISO_C_BINDING, ONLY: C_INT, C_FLOAT, C_LOC
24     USE FTN_C_2
25     ...
26     REAL (C_FLOAT), TARGET :: SEND(100)
27     INTEGER (C_INT)      :: SENDCOUNT
28     INTEGER (C_INT), ALLOCATABLE, TARGET :: RECVCOUNTS(100)
29     ...
30     ALLOCATE( RECVCOUNTS(100) )
31     ...
32     CALL C_LIBRARY_FUNCTION(C_LOC(SEND), SENDCOUNT, &
33     C_LOC(RECVCOUNTS))
34     ...

```

35 The preceding code contains the declaration of the Fortran actual arguments associated with the above-  
36 listed Fortran dummy arguments.

37 The first Fortran actual argument is the address of the first element of the array `SEND`, which has the  
38 type `REAL(C_FLOAT)` and the `TARGET` attribute. This address is returned by the intrinsic function  
39 `C_LOC`. This actual argument is associated with the Fortran dummy argument `SENDBUF`, which has  
40 the type `C_PTR` and the `VALUE` attribute.

1 The second Fortran actual argument is SENDCOUNT of type INTEGER(C\_INT), which is associated  
 2 with the Fortran dummy argument SENDCOUNT, which has the type INTEGER(C\_INT) and the  
 3 VALUE attribute.

4 The third Fortran actual argument is the address of the first element of the allocatable array RECV-  
 5 COUNTS, with has the type REAL(C\_FLOAT) and the TARGET attribute. This address is returned  
 6 by the intrinsic function C\_LOC. This actual argument is associated with the Fortran dummy argument  
 7 RECVCOUNTS, which has the type C\_PTR and the VALUE attribute.

### 8 C.10.2.2 Example of C calling Fortran

9 Fortran Code:

```
10 SUBROUTINE SIMULATION(ALPHA, BETA, GAMMA, DELTA, ARRAYS), BIND(C)
11   USE, INTRINSIC :: ISO_C_BINDING
12   IMPLICIT NONE
13   INTEGER (C_LONG), VALUE           :: ALPHA
14   REAL (C_DOUBLE), INTENT(INOUT)   :: BETA
15   INTEGER (C_LONG), INTENT(OUT)    :: GAMMA
16   REAL (C_DOUBLE), DIMENSION(*), INTENT(IN) :: DELTA
17   TYPE, BIND(C) :: PASS
18     INTEGER (C_INT) :: LENC, LENF
19     TYPE (C_PTR)    :: C, F
20   END TYPE PASS
21   TYPE (PASS), INTENT(INOUT) :: ARRAYS
22   REAL (C_FLOAT), ALLOCATABLE, TARGET, SAVE :: ETA(:)
23   REAL (C_FLOAT), POINTER :: C_ARRAY(:)
24   ...
25   ! Associate C_ARRAY with an array allocated in C
26   CALL C_F_POINTER (ARRAYS%C, C_ARRAY, (/ARRAYS%LENC/))
27   ...
28   ! Allocate an array and make it available in C
29   ARRAYS%LENF = 100
30   ALLOCATE (ETA(ARRAYS%LENF))
31   ARRAYS%F = C_LOC(ETA)
32   ...
33 END SUBROUTINE SIMULATION
```

34 C Struct Declaration

```
35     struct pass {int lenc, lenf; float* f, c};
```

36 C Function Prototype:

```
37     void simulation(long alpha, double *beta, long *gamma,
38         double delta[], pass *arrays);
```

1 C Calling Sequence:

```
2 simulation(alpha, &beta, &gamma, delta, &arrays);
```

3 The above-listed Fortran code specifies a subroutine SIMULATION. This subroutine corresponds to the  
4 C void function `simulation`.

5 The Fortran subroutine references the intrinsic module ISO\_C\_BINDING.

6 The first Fortran dummy argument of the subroutine is ALPHA, which has the type INTEGER(C\_-  
7 LONG) and the attribute VALUE. This dummy argument corresponds to the C formal parameter  
8 `alpha`, which is a long. The actual C parameter is also a long.

9 The second Fortran dummy argument of the subroutine is BETA, which has the type REAL(C\_-  
10 DOUBLE) and the INTENT(INOUT) attribute. This dummy argument corresponds to the C formal  
11 parameter `beta`, which is a pointer to double. An address is passed as the actual parameter in the C  
12 calling sequence.

13 The third Fortran dummy argument of the subroutine is GAMMA, which has the type INTEGER(C\_-  
14 LONG) and the INTENT(OUT) attribute. This dummy argument corresponds to the C formal param-  
15 eter `gamma`, which is a pointer to long. An address is passed as the actual parameter in the C calling  
16 sequence.

17 The fourth Fortran dummy argument is the assumed-size array DELTA, which has the type REAL  
18 (C\_DOUBLE) and the attribute INTENT(IN). This dummy argument corresponds to the C formal  
19 parameter `delta`, which is a double array. The actual C parameter is also a double array.

20 The fifth Fortran dummy argument is ARRAYS, which is a structure for accessing an array allocated  
21 in C and an array allocated in Fortran. The lengths of these arrays are held in the components LENC  
22 and LENF; their C addresses are help in components C and F.

### 23 C.10.2.3 Example of calling C functions with non-interoperable data

24 Many Fortran processors support 16-byte real numbers, which might not be supported by the C processor.  
25 Assume a Fortran programmer wants to use a C procedure from a message passing library for an array  
26 of these reals. The C prototype of this procedure is

```
27 void ProcessBuffer(void *buffer, int n_bytes);
```

28 with the corresponding Fortran interface

```
29 USE, INTRINSIC :: ISO_C_BINDING
```

```
30
```

```
31 INTERFACE
```

```
32 SUBROUTINE PROCESS_BUFFER(BUFFER,N_BYTES), BIND(C,NAME="ProcessBuffer")
```

```
33 IMPORT :: C_PTR, C_INT
```

```
34 TYPE(C_PTR), VALUE :: BUFFER ! The 'C address' of the array buffer
```

```
35 INTEGER(C_INT), VALUE :: N_BYTES ! Number of bytes in buffer
```

```
36 END SUBROUTINE PROCESS_BUFFER
```

```
37 END INTERFACE
```

38 This may be done using C\_LOC if the particular Fortran processor specifies that C\_LOC returns an  
39 appropriate address:

```

1 REAL(R_QUAD), DIMENSION(:), ALLOCATABLE, TARGET :: QUAD_ARRAY
2 ...
3 CALL PROCESS_BUFFER(C_LOC(QUAD_ARRAY), INT(16*SIZE(QUAD_ARRAY),C_INT))
4 ! One quad real takes 16 bytes on this processor

```

#### 5 C.10.2.4 Example of opaque communication between C and Fortran

6 The following example demonstrates how a Fortran processor can make a modern OO random number  
7 generator written in Fortran available to a C program:

```

8 USE, INTRINSIC :: ISO_C_BINDING
9 ! Assume this code is inside a module
10
11 TYPE RANDOM_STREAM
12 ! A (uniform) random number generator (URNG)
13 CONTAINS
14 PROCEDURE(RANDOM_UNIFORM), DEFERRED, PASS(STREAM) :: NEXT
15 ! Generates the next number from the stream
16 END TYPE RANDOM_STREAM
17
18 ABSTRACT INTERFACE
19 ! Abstract interface of Fortran URNG
20 SUBROUTINE RANDOM_UNIFORM(STREAM, NUMBER)
21 IMPORT :: RANDOM_STREAM, C_DOUBLE
22 CLASS(RANDOM_STREAM), INTENT(INOUT) :: STREAM
23 REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
24 END SUBROUTINE RANDOM_UNIFORM
25 END INTERFACE

```

26 A polymorphic object of base type RANDOM\_STREAM is not interoperable with C. However, we can  
27 make such a random number generator available to C by packaging it inside another nonpolymorphic,  
28 nonparameterized derived type:

```

29 TYPE :: URNG_STATE ! No BIND(C), as this type is not interoperable
30 CLASS(RANDOM_STREAM), ALLOCATABLE :: STREAM
31 END TYPE URNG_STATE

```

32 The following two procedures will enable a C program to use our Fortran uniform random number  
33 generator:

```

34 ! Initialize a uniform random number generator:
35 SUBROUTINE INITIALIZE_URNG(STATE_HANDLE, METHOD), &
36 BIND(C, NAME="InitializeURNG")
37 TYPE(C_PTR), INTENT(OUT) :: STATE_HANDLE
38 ! An opaque handle for the URNG
39 CHARACTER(C_CHAR), DIMENSION(*), INTENT(IN) :: METHOD

```

```

1      ! The algorithm to be used
2
3      TYPE(URNG_STATE), POINTER :: STATE
4      ! An actual URNG object
5
6      ALLOCATE(STATE)
7      ! There needs to be a corresponding finalization
8      ! procedure to avoid memory leaks, not shown in this example
9      ! Allocate STATE%STREAM with a dynamic type depending on METHOD
10     ...
11     STATE_HANDLE=C_LOC(STATE)
12     ! Obtain an opaque handle to return to C
13 END SUBROUTINE INITIALIZE_URNG
14
15 ! Generate a random number:
16 SUBROUTINE GENERATE_UNIFORM(STATE_HANDLE, NUMBER), &
17     BIND(C, NAME="GenerateUniform")
18     TYPE(C_PTR), INTENT(IN), VALUE :: STATE_HANDLE
19     ! An opaque handle: Obtained via a call to INITIALIZE_URNG
20     REAL(C_DOUBLE), INTENT(OUT) :: NUMBER
21
22     TYPE(URNG_STATE), POINTER :: STATE
23     ! A pointer to the actual URNG
24
25     CALL C_F_POINTER(CPTR=STATE_HANDLE, FPTR=STATE)
26     ! Convert the opaque handle into a usable pointer
27     CALL STATE%STREAM%NEXT(NUMBER)
28     ! Use the type-bound procedure NEXT to generate NUMBER
29 END SUBROUTINE GENERATE_UNIFORM

```

## 30 C.11 Section 16 notes

### 31 C.11.1 Examples of host association (16.4.1.3)

32 The first two examples are examples of valid host association. The third example is an example of invalid  
33 host association.

34 Example 1:

```

35 PROGRAM A
36     INTEGER I, J
37     ...
38 CONTAINS
39     SUBROUTINE B
40         INTEGER I ! Declaration of I hides
41                 ! program A's declaration of I

```

```

1      ...
2      I = J      ! Use of variable J from program A
3                ! through host association
4      END SUBROUTINE B
5  END PROGRAM A

6  Example 2:

7  PROGRAM A
8      TYPE T
9      ...
10     END TYPE T
11     ...
12  CONTAINS
13     SUBROUTINE B
14         IMPLICIT TYPE (T) (C) ! Refers to type T declared below
15                                     ! in subroutine B, not type T
16                                     ! declared above in program A
17         ...
18         TYPE T
19         ...
20     END TYPE T
21     ...
22  END SUBROUTINE B
23  END PROGRAM A

```

24 Example 3:

```

25  PROGRAM Q
26      REAL (KIND = 1) :: C
27      ...
28  CONTAINS
29      SUBROUTINE R
30          REAL (KIND = KIND (C)) :: D ! Invalid declaration
31                                      ! See below
32          REAL (KIND = 2) :: C
33          ...
34      END SUBROUTINE R
35  END PROGRAM Q

```

36 In the declaration of D in subroutine R, the use of C would refer to the declaration of C in subroutine  
37 R, not program Q. However, it is invalid because the declaration of C is required to occur before it is  
38 used in the declaration of D (7.1.7).

### 39 C.11.2 Rules ensuring unambiguous generics (16.2.3)

40 The rules in 16.2.3 are intended to ensure

- 1     • that it is possible to reference each specific procedure in the generic collection,
- 2     • that for any valid reference to the generic procedure, the determination of the specific procedure
- 3       referenced is unambiguous, and
- 4     • that the determination of the specific procedure referenced can be made before execution of the
- 5       program begins (during compilation).

6 Specific procedures are distinguished by fixed properties of their arguments, specifically type, kind type  
7 parameters, and rank. A valid reference to one procedure in a generic collection will differ from another  
8 because it has an argument that the other cannot accept, because it is missing an argument that the  
9 other requires, or because one of these fixed properties is different.

10 Although the declared type of a data entity is a fixed property, polymorphic variables allow for a  
11 limited degree of type mismatch between dummy arguments and actual arguments, so the requirement  
12 for distinguishing two dummy arguments is type incompatibility, not merely different types. (This is  
13 illustrated in the BAD6 example later in this note.)

14 That same limited type mismatch means that two dummy arguments that are not type incompatible  
15 can be distinguished on the basis of the values of the kind type parameters they have in common; if one  
16 of them has a kind type parameter that the other does not, that is irrelevant in distinguishing them.

17 Rank is a fixed property, but some forms of array dummy arguments allow rank mismatches when a  
18 procedure is referenced by its specific name. In order to allow rank to always be usable in distinguishing  
19 generics, such rank mismatches are disallowed for those arguments when the procedure is referenced as  
20 part of a generic. Additionally, the fact that elemental procedures can accept array arguments is not  
21 taken into account when applying these rules, so apparent ambiguity between elemental and nonelemental  
22 procedures is possible; in such cases, the reference is interpreted as being to the nonelemental procedure.

23 For procedures referenced as operators or defined-assignment, syntactically distinguished arguments are  
24 mapped to specific positions in the argument list, so the rule for distinguishing such procedures is that  
25 it be possible to distinguish the arguments at one of the argument positions.

26 For user-defined derived-type input/output procedures, only the `dtv` argument corresponds to something  
27 explicitly written in the program, so it is the `dtv` that is required to be distinguished. Because `dtv`  
28 arguments are required to be scalar, they cannot differ in rank. Thus this rule effectively involves only  
29 type and kind type parameters.

30 For generic procedures identified by names, the rules are more complicated because optional arguments  
31 may be omitted and because arguments may be specified either positionally or by name.

32 In the special case of type-bound procedures with passed-object dummy arguments, the passed-object  
33 argument is syntactically distinguished in the reference, so rule (2) can be applied. The type of passed-  
34 object arguments is constrained in ways that prevent passed-object arguments in the same scoping unit  
35 from being type incompatible. Thus this rule effectively involves only kind type parameters and rank.

36 The primary means of distinguishing named generics is rule (3). The most common application of that  
37 rule is a single argument satisfying both (3a) and (3b):

```
38     INTERFACE GOOD1
39         FUNCTION F1A(X)
40             REAL :: F1A,X
41         END FUNCTION F1A
42         FUNCTION F1B(X)
43             INTEGER :: F1B,X
```

```

1         END FUNCTION F1B
2         END INTERFACE GOOD1

```

3 Whether one writes `GOOD1(1.0)` or `GOOD1(X=1.0)`, the reference is to F1A because F1B would require an  
4 integer argument whereas these references provide the real constant 1.0.

5 This example and those that follow are expressed using interface bodies, with type as the distinguishing  
6 property. This was done to make it easier to write and describe the examples. The principles being  
7 illustrated are equally applicable when the procedures get their explicit interfaces in some other way or  
8 when kind type parameters or rank are the distinguishing property.

9 Another common variant is the argument that satisfies (3a) and (3b) by being required in one specific  
10 and completely missing in the other:

```

11        INTERFACE GOOD2
12            FUNCTION F2A(X)
13                REAL :: F2A,X
14            END FUNCTION F2A
15            FUNCTION F2B(X,Y)
16                COMPLEX :: F2B
17                REAL :: X,Y
18            END FUNCTION F2B
19        END INTERFACE GOOD2

```

20 Whether one writes `GOOD2(0.0,1.0)`, `GOOD2(0.0,Y=1.0)`, or `GOOD2(Y=1.0,X=0.0)`, the reference is to  
21 F2B, because F2A has no argument in the second position or with the name Y. This approach is used as  
22 an alternative to optional arguments when one wants a function to have different result type, kind type  
23 parameters, or rank, depending on whether the argument is present. In many of the intrinsic functions,  
24 the DIM argument works this way.

25 It is possible to construct cases where different arguments are used to distinguish positionally and by  
26 name:

```

27        INTERFACE GOOD3
28            SUBROUTINE S3A(W,X,Y,Z)
29                REAL :: W,Y
30                INTEGER :: X,Z
31            END SUBROUTINE S3A
32            SUBROUTINE S3B(X,W,Z,Y)
33                REAL :: W,Z
34                INTEGER :: X,Y
35            END SUBROUTINE S3B
36        END INTERFACE GOOD3

```

37 If one writes `GOOD3(1.0,2,3.0,4)` to reference S3A, then the third and fourth arguments are consistent  
38 with a reference to S3B, but the first and second are not. If one switches to writing the first two  
39 arguments as keyword arguments in order for them to be consistent with a reference to S3B, the latter  
40 two arguments must also be written as keyword arguments, `GOOD3(X=2,W= 1.0,Z=4,Y=3.0)`, and the  
41 named arguments Y and Z are distinguished.

1 The ordering requirement in rule (3) is critical:

```

2     INTERFACE BAD4 ! this interface is invalid !
3     SUBROUTINE S4A(W,X,Y,Z)
4         REAL :: W,Y
5         INTEGER :: X,Z
6     END SUBROUTINE S4A
7     SUBROUTINE S4B(X,W,Z,Y)
8         REAL :: X,Y
9         INTEGER :: W,Z
10    END SUBROUTINE S4B
11    END INTERFACE BAD4

```

12 In this example, the positionally distinguished arguments are Y and Z, and it is W and X that are  
13 distinguished by name. In this order it is possible to write BAD4(1.0,2,Y=3.0,Z=4), which is a valid  
14 reference for both S4A and S4B.

15 Rule (1) can be used to distinguish some cases that are not covered by rule (3):

```

16    INTERFACE GOOD5
17    SUBROUTINE S5A(X)
18        REAL :: X
19    END SUBROUTINE S5A
20    SUBROUTINE S5B(Y,X)
21        REAL :: Y,X
22    END SUBROUTINE S5B
23    END INTERFACE GOOD5

```

24 In attempting to apply rule (3), position 2 and name Y are distinguished, but they are in the wrong  
25 order, just like the BAD4 example. However, when we try to construct a similarly ambiguous reference,  
26 we get GOOD5(1.0,X=2.0), which can't be a reference to S5A because it would be attempting to associate  
27 two different actual arguments with the dummy argument X. Rule (3) catches this case by recognizing  
28 that S5B requires two real arguments, and S5A cannot possibly accept more than one.

29 The application of rule (1) becomes more complicated when extensible types are involved. If FRUIT is  
30 an extensible type, PEAR and APPLE are extensions of FRUIT, and BOSC is an extension of PEAR, then

```

31    INTERFACE BAD6 ! this interface is invalid !
32    SUBROUTINE S6A(X,Y)
33        CLASS(PEAR) :: X,Y
34    END SUBROUTINE S6A
35    SUBROUTINE S6B(X,Y)
36        CLASS(FRUIT) :: X
37        CLASS(BOSC) :: Y
38    END SUBROUTINE S6B
39    END INTERFACE BAD6

```

1 might, at first glance, seem distinguishable this way, but because of the limited type mismatching allowed,  
 2 BAD6(A\_PEAR,A\_BOSC) is a valid reference to both S6A and S6B.

3 It is important to try rule (1) for each type present:

```

4     INTERFACE GOOD7
5         SUBROUTINE S7A(X,Y,Z)
6             CLASS(PEAR) :: X,Y,Z
7         END SUBROUTINE S7A
8         SUBROUTINE S7B(X,Z,W)
9             CLASS(FRUIT) :: X
10            CLASS(BOSC) :: Z
11            CLASS(APPLE),OPTIONAL :: W
12        END SUBROUTINE S7B
13    END INTERFACE GOOD7

```

14 Looking at the most general type, S7A has a minimum and maximum of 3 FRUIT arguments, while S7B  
 15 has a minimum of 2 and a maximum of three. Looking at the most specific, S7A has a minimum of 0  
 16 and a maximum of 3 BOSC arguments, while S7B has a minimum of 1 and a maximum of 2. However,  
 17 when we look at the intermediate, S7A has a minimum and maximum of 3 PEAR arguments, while S7B  
 18 has a minimum of 1 and a maximum of 2. Because S7A's minimum exceeds S7B's maximum, they can  
 19 be distinguished.

20 In identifying the minimum number of arguments with a particular set of properties, we exclude optional  
 21 arguments and test TKR compatibility, so the corresponding actual arguments are required to have  
 22 those properties. In identifying the maximum number of arguments with those properties, we include  
 23 the optional arguments and test not distinguishable, so we include actual arguments which could have  
 24 those properties but are not required to have them.

25 These rules are sufficient to ensure that references to procedures that meet them are unambiguous, but  
 26 there remain examples that fail to meet these rules but which can be shown to be unambiguous:

```

27     INTERFACE BAD8 ! this interface is invalid !
28     ! despite the fact that it is unambiguous !
29     SUBROUTINE S8A(X,Y,Z)
30         REAL,OPTIONAL :: X
31         INTEGER :: Y
32         REAL :: Z
33     END SUBROUTINE S8A
34     SUBROUTINE S8B(X,Z,Y)
35         INTEGER,OPTIONAL :: X
36         INTEGER :: Z
37         REAL :: Y
38     END SUBROUTINE S8B
39 END INTERFACE BAD8

```

40 This interface fails rule (3) because there are no required arguments that can be distinguished from the  
 41 positionally corresponding argument, but in order for the mismatch of the optional arguments not to  
 42 be relevant, the later arguments must be specified as keyword arguments, so distinguishing by name

1 does the trick. This interface is nevertheless invalid so a standard- conforming Fortran processor is not  
2 required to do such reasoning. The rules to cover all cases are too complicated to be useful.

3 In addition to not recognizing distinguishable patterns like the one in BAD8, the rules do not distinguish  
4 on the basis of any properties other than type, kind type parameters, and rank:

```

5     INTERFACE BAD9 ! this interface is invalid !
6     ! despite the fact that it is unambiguous !
7     SUBROUTINE S9A(X)
8     REAL :: X
9     END SUBROUTINE S9A
10    SUBROUTINE S9B(X)
11    INTERFACE
12    FUNCTION X(A)
13    REAL :: X,A
14    END FUNCTION X
15    END INTERFACE
16    END SUBROUTINE S9B
17    SUBROUTINE S9C(X)
18    INTERFACE
19    FUNCTION X(A)
20    REAL :: X
21    INTEGER :: A
22    END FUNCTION X
23    END INTERFACE
24    END SUBROUTINE S9C
25    END INTERFACE BAD9

```

26 The real data objects that would be valid arguments for S9A are entirely disjoint from procedures that  
27 are valid arguments to S9B and S9C, and the procedures that valid arguments for S9B are disjoint from  
28 the procedures that are valid arguments to S9C because the former are required to accept real arguments  
29 and the latter integer arguments. Again, this interface is invalid, so a standard-conforming Fortran  
30 processor need not examine such properties when deciding whether a generic collection is valid. Again,  
31 the rules to cover all cases are too complicated to be useful.

## 32 C.12 Array feature notes

### 33 C.12.1 Summary of features

34 This section is a summary of the principal array features.

#### 35 C.12.1.1 Whole array expressions and assignments (7.4.1.2, 7.4.1.3)

36 An important feature is that whole array expressions and assignments are permitted. For example, the  
37 statement

```
38 A = B + C * SIN (D)
```

39 where A, B, C, and D are arrays of the same shape, is permitted. It is interpreted element-by-element;

1 that is, the sine function is taken on each element of D, each result is multiplied by the corresponding  
2 element of C, added to the corresponding element of B, and assigned to the corresponding element of  
3 A. Functions, including user-written functions, may be arrays and may be generic with scalar versions.  
4 All arrays in an expression or across an assignment shall conform; that is, have exactly the same shape  
5 (number of dimensions and extents in each dimension), but scalars may be included freely and these are  
6 interpreted as being broadcast to a conforming array. Expressions are evaluated before any assignment  
7 takes place.

#### 8 **C.12.1.2 Array sections (2.4.5, 6.2.2.3)**

9 Whenever whole arrays may be used, it is also possible to use subarrays called “sections”. For example:

```
10 A (:, 1:N, 2, 3:1:-1)
```

11 consists of a subarray containing the whole of the first dimension, positions 1 to N of the second dimen-  
12 sion, position 2 of the third dimension and positions 1 to 3 in reverse order of the fourth dimension.  
13 This is an artificial example chosen to illustrate the different forms. Of course, a common use may be  
14 to select a row or column of an array, for example:

```
15 A (:, J)
```

#### 16 **C.12.1.3 WHERE statement (7.4.3)**

17 The WHERE statement applies a conforming logical array as a mask on the individual operations in the  
18 expression and in the assignment. For example:

```
19 WHERE (A > 0) B = LOG (A)
```

20 takes the logarithm only for positive components of A and makes assignments only in these positions.

21 The WHERE statement also has a block form (WHERE construct).

#### 22 **C.12.1.4 Automatic arrays and allocatable variables (5.1, 5.1.2.5.3)**

23 Two features useful for writing modular software are automatic arrays, created on entry to a subprogram  
24 and destroyed on return, and allocatable variables, including arrays whose rank is fixed but whose actual  
25 size and lifetime is fully under the programmer’s control through explicit ALLOCATE and DEALLO-  
26 CATE statements. The declarations

```
27 SUBROUTINE X (N, A, B)
```

```
28 REAL WORK (N, N); REAL, ALLOCATABLE :: HEAP (:, :)
```

29 specify an automatic array WORK and an allocatable array HEAP. Note that a stack is an adequate  
30 storage mechanism for the implementation of automatic arrays, but a heap will be needed for some  
31 allocatable variables.

#### 32 **C.12.1.5 Array constructors (4.7)**

33 Arrays, and in particular array constants, may be constructed with array constructors exemplified by:

```
34 (/ 1.0, 3.0, 7.2 /)
```

35 which is a rank-one array of size 3,

```
36 (/ (1.3, 2.7, L = 1, 10), 7.1 /)
```

1 which is a rank-one array of size 21 and contains the pair of real constants 1.3 and 2.7 repeated 10 times  
 2 followed by 7.1, and

3 (/ (I, I = 1, N) /)

4 which contains the integers 1, 2, ..., N. Only rank-one arrays may be constructed in this way, but higher  
 5 dimensional arrays may be made from them by means of the intrinsic function RESHAPE.

## 6 C.12.2 Examples

7 The array features have the potential to simplify the way that almost any array-using program is con-  
 8 ceived and written. Many algorithms involving arrays can now be written conveniently as a series of  
 9 computations with whole arrays.

### 10 C.12.2.1 Unconditional array computations

11 At the simplest level, statements such as

12 A = B + C

13 or

14 S = SUM (A)

15 can take the place of entire DO loops. The loops were required to perform array addition or to sum all  
 16 the elements of an array.

17 Further examples of unconditional operations on arrays that are simple to write are:

matrix multiply	P = MATMUL (Q, R)
largest array element	L = MAXVAL (P)
factorial N	F = PRODUCT ((/ (K, K = 2, N) /))

18 The Fourier sum  $F = \sum_{i=1}^N a_i \times \cos x_i$  may also be computed without writing a DO loop if one makes  
 19 use of the element-by-element definition of array expressions as described in Section 7. Thus, we can  
 20 write

21 F = SUM (A \* COS (X))

22 The successive stages of calculation of F would then involve the arrays:

A	=	(/ A (1), ..., A (N) /)
X	=	(/ X (1), ..., X (N) /)
COS (X)	=	(/ COS (X (1)), ..., COS (X (N)) /)
A * COS (X)	=	(/ A (1) * COS (X (1)), ..., A (N) * COS (X (N)) /)

23 The final scalar result is obtained simply by summing the elements of the last of these arrays. Thus, the  
 24 processor is dealing with arrays at every step of the calculation.

### 25 C.12.2.2 Conditional array computations

26 Suppose we wish to compute the Fourier sum in the above example, but to include only those terms  
 27  $a(i) \cos x(i)$  that satisfy the condition that the coefficient  $a(i)$  is less than 0.01 in absolute value. More  
 28 precisely, we are now interested in evaluating the conditional Fourier sum

$$CF = \sum_{|a_i| < 0.01} a_i \times \cos x_i$$

1 where the index runs from 1 to N as before.

2 This can be done by using the MASK parameter of the SUM function, which restricts the summation  
 3 of the elements of the array A \* COS (X) to those elements that correspond to true elements of MASK.  
 4 Clearly, the mask required is the logical array expression ABS (A) < 0.01. Note that the stages of  
 5 evaluation of this expression are:

$$\begin{aligned} A &= (/ A (1), \dots, A (N) /) \\ \text{ABS (A)} &= (/ \text{ABS (A (1))}, \dots, \text{ABS (A (N))} /) \\ \text{ABS (A) < 0.01} &= (/ \text{ABS (A (1)) < 0.01}, \dots, \text{ABS (A (N)) < 0.01} /) \end{aligned}$$

6 The conditional Fourier sum we arrive at is:

7 CF = SUM (A \* COS (X), MASK = ABS (A) < 0.01)

8 If the mask is all false, the value of CF is zero.

9 The use of a mask to define a subset of an array is crucial to the action of the WHERE statement. Thus  
 10 for example, to zero an entire array, we may write simply A = 0; but to set only the negative elements  
 11 to zero, we need to write the conditional assignment

12 WHERE (A .LT. 0) A = 0

13 The WHERE statement complements ordinary array assignment by providing array assignment to any  
 14 subset of an array that can be restricted by a logical expression.

15 In the Ising model described below, the WHERE statement predominates in use over the ordinary array  
 16 assignment statement.

### 17 C.12.2.3 A simple program: the Ising model

18 The Ising model is a well-known Monte Carlo simulation in 3-dimensional Euclidean space which is  
 19 useful in certain physical studies. We will consider in some detail how this might be programmed. The  
 20 model may be described in terms of a logical array of shape N by N by N. Each gridpoint is a single  
 21 logical variable which is to be interpreted as either an up-spin (true) or a down-spin (false).

22 The Ising model operates by passing through many successive states. The transition to the next state is  
 23 governed by a local probabilistic process. At each transition, all gridpoints change state simultaneously.  
 24 Every spin either flips to its opposite state or not according to a rule that depends only on the states  
 25 of its 6 nearest neighbors in the surrounding grid. The neighbors of gridpoints on the boundary faces of  
 26 the model cube are defined by assuming cubic periodicity. In effect, this extends the grid periodically  
 27 by replicating it in all directions throughout space.

28 The rule states that a spin is flipped to its opposite parity for certain gridpoints where a mere 3 or  
 29 fewer of the 6 nearest neighbors have the same parity as it does. Also, the flip is executed only with  
 30 probability P (4), P (5), or P (6) if as many as 4, 5, or 6 of them have the same parity as it does. (The  
 31 rule seems to promote neighborhood alignments that may presumably lead to equilibrium in the long  
 32 run.)

1 **C.12.2.3.1 Problems to be solved**

2 Some of the programming problems that we will need to solve in order to translate the Ising model into  
3 Fortran statements using entire arrays are

- 4 (1) Counting nearest neighbors that have the same spin;  
5 (2) Providing an array function to return an array of random numbers; and  
6 (3) Determining which gridpoints are to be flipped.

7 **C.12.2.3.2 Solutions in Fortran**

8 The arrays needed are:

9 LOGICAL ISING (N, N, N), FLIPS (N, N, N)  
10 INTEGER ONES (N, N, N), COUNT (N, N, N)  
11 REAL THRESHOLD (N, N, N)

12 The array function needed is:

13 FUNCTION RAND (N)  
14 REAL RAND (N, N, N)

15 The transition probabilities are specified in the array

16 REAL P (6)

17 The first task is to count the number of nearest neighbors of each gridpoint  $g$  that have the same spin  
18 as  $g$ .

19 Assuming that ISING is given to us, the statements

20 ONES = 0  
21 WHERE (ISING) ONES = 1

22 make the array ONES into an exact analog of ISING in which 1 stands for an up-spin and 0 for a  
23 down-spin.

24 The next array we construct, COUNT, will record for every gridpoint of ISING the number of spins to  
25 be found among the 6 nearest neighbors of that gridpoint. COUNT will be computed by adding together  
26 6 arrays, one for each of the 6 relative positions in which a nearest neighbor is found. Each of the 6  
27 arrays is obtained from the ONES array by shifting the ONES array one place circularly along one of  
28 its dimensions. This use of circular shifting imparts the cubic periodicity.

29 COUNT = CSHIFT (ONES, SHIFT = -1, DIM = 1) &  
30 + CSHIFT (ONES, SHIFT = 1, DIM = 1) &  
31 + CSHIFT (ONES, SHIFT = -1, DIM = 2) &  
32 + CSHIFT (ONES, SHIFT = 1, DIM = 2) &  
33 + CSHIFT (ONES, SHIFT = -1, DIM = 3) &  
34 + CSHIFT (ONES, SHIFT = 1, DIM = 3)

35 At this point, COUNT contains the count of nearest neighbor up-spins even at the gridpoints where  
36 the Ising model has a down-spin. But we want a count of down-spins at those gridpoints, so we correct  
37 COUNT at the down (false) points of ISING by writing:

1 WHERE (.NOT. ISING) COUNT = 6 - COUNT

2 Our object now is to use these counts of what may be called the “like-minded nearest neighbors” to  
 3 decide which gridpoints are to be flipped. This decision will be recorded as the true elements of an array  
 4 FLIPS. The decision to flip will be based on the use of uniformly distributed random numbers from the  
 5 interval  $0 \leq p < 1$ . These will be provided at each gridpoint by the array function RAND. The flip will  
 6 occur at a given point if and only if the random number at that point is less than a certain threshold  
 7 value. In particular, by making the threshold value equal to 1 at the points where there are 3 or fewer  
 8 like-minded nearest neighbors, we guarantee that a flip occurs at those points (because  $p$  is always less  
 9 than 1). Similarly, the threshold values corresponding to counts of 4, 5, and 6 are assigned  $P(4)$ ,  $P(5)$ ,  
 10 and  $P(6)$  in order to achieve the desired probabilities of a flip at those points ( $P(4)$ ,  $P(5)$ , and  $P(6)$ )  
 11 are input parameters in the range 0 to 1).

12 The thresholds are established by the statements:

13 THRESHOLD = 1.0

14 WHERE (COUNT == 4) THRESHOLD = P (4)

15 WHERE (COUNT == 5) THRESHOLD = P (5)

16 WHERE (COUNT == 6) THRESHOLD = P (6)

17 and the spins that are to be flipped are located by the statement:

18 FLIPS = RAND (N) <= THRESHOLD

19 All that remains to complete one transition to the next state of the ISING model is to reverse the spins  
 20 in ISING wherever FLIPS is true:

21 WHERE (FLIPS) ISING = .NOT. ISING

### 22 C.12.2.3.3 The complete Fortran subroutine

23 The complete code, enclosed in a subroutine that performs a sequence of transitions, is as follows:

24 SUBROUTINE TRANSITION (N, ISING, ITERATIONS, P)

25

26 LOGICAL ISING (N, N, N), FLIPS (N, N, N)

27 INTEGER ONES (N, N, N), COUNT (N, N, N)

28 REAL THRESHOLD (N, N, N), P (6)

29

30 DO I = 1, ITERATIONS

31 ONES = 0

32 WHERE (ISING) ONES = 1

33 COUNT = CSHIFT (ONES, -1, 1) + CSHIFT (ONES, 1, 1) &

34 + CSHIFT (ONES, -1, 2) + CSHIFT (ONES, 1, 2) &

35 + CSHIFT (ONES, -1, 3) + CSHIFT (ONES, 1, 3)

36 WHERE (.NOT. ISING) COUNT = 6 - COUNT

37 THRESHOLD = 1.0

38 WHERE (COUNT == 4) THRESHOLD = P (4)

39 WHERE (COUNT == 5) THRESHOLD = P (5)

40 WHERE (COUNT == 6) THRESHOLD = P (6)

```

1      FLIPS = RAND (N) <= THRESHOLD
2      WHERE (FLIPS) ISING = .NOT. ISING
3  END DO
4
5  CONTAINS
6      FUNCTION RAND (N)
7          REAL RAND (N, N, N)
8          CALL RANDOM_NUMBER (HARVEST = RAND)
9          RETURN
10     END FUNCTION RAND
11 END

```

#### 12 **C.12.2.3.4 Reduction of storage**

13 The array ISING could be removed (at some loss of clarity) by representing the model in ONES all the  
14 time. The array FLIPS can be avoided by combining the two statements that use it as:

```
15 WHERE (RAND (N) <= THRESHOLD) ISING = .NOT. ISING
```

16 but an extra temporary array would probably be needed. Thus, the scope for saving storage while  
17 performing whole array operations is limited. If N is small, this will not matter and the use of whole  
18 array operations is likely to lead to good execution speed. If N is large, storage may be very important  
19 and adequate efficiency will probably be available by performing the operations plane by plane. The  
20 resulting code is not as elegant, but all the arrays except ISING will have size of order  $N^2$  instead of  $N^3$ .

### 21 **C.12.3 FORMula TRANslation and array processing**

22 Many mathematical formulas can be translated directly into Fortran by use of the array processing  
23 features.

24 We assume the following array declarations:

```
25 REAL X (N), A (M, N)
```

26 Some examples of mathematical formulas and corresponding Fortran expressions follow.

#### 27 **C.12.3.1 A sum of products**

The expression

$$\sum_{j=1}^N \prod_{i=1}^M a_{ij}$$

28 can be formed using the Fortran expression

```
29 SUM (PRODUCT (A, DIM=1))
```

30 The argument DIM=1 means that the product is to be computed down each column of A. If A had the  
31 value  $\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$  the result of this expression is BE + CF + DG.

1 **C.12.3.2 A product of sums**

The expression

$$\prod_{i=1}^M \sum_{j=1}^N a_{ij}$$

2 can be formed using the Fortran expression

3 `PRODUCT (SUM (A, DIM = 2))`

4 The argument `DIM = 2` means that the sum is to be computed along each row of A. If A had the  
 5 value  $\begin{bmatrix} B & C & D \\ E & F & G \end{bmatrix}$  the result of this expression is  $(B+C+D)(E+F+G)$ .

6 **C.12.3.3 Addition of selected elements**

The expression

$$\sum_{x_i > 0.0} x_i$$

7 can be formed using the Fortran expression

8 `SUM (X, MASK = X > 0.0)`

9 The mask locates the positive elements of the array of rank one. If X has the vector value (0.0, -0.1,  
 10 0.2, 0.3, 0.2, -0.1, 0.0), the result of this expression is 0.7.

11 **C.12.4 Sum of squared residuals**

The expression

$$\sum_{i=1}^N (x_i - x_{\text{mean}})^2$$

12 can be formed using the Fortran statements

13 `XMEAN = SUM (X) / SIZE (X)`

14 `SS = SUM ((X - XMEAN) ** 2)`

15 Thus, SS is the sum of the squared residuals.

16 **C.12.5 Vector norms: infinity-norm and one-norm**

17 The infinity-norm of vector  $X = (X(1), \dots, X(N))$  is defined as the largest of the numbers  $ABS(X(1)),$   
 18  $\dots, ABS(X(N))$  and therefore has the value `MAXVAL (ABS(X))`.

19 The one-norm of vector X is defined as the *sum* of the numbers  $ABS(X(1)), \dots, ABS(X(N))$  and  
 20 therefore has the value `SUM (ABS(X))`.

21 **C.12.6 Matrix norms: infinity-norm and one-norm**

22 The infinity-norm of the matrix  $A = (A(I, J))$  is the largest row-sum of the matrix  $ABS(A(I, J))$  and  
 23 therefore has the value `MAXVAL (SUM (ABS(A), DIM = 2))`.

24 The one-norm of the matrix  $A = (A(I, J))$  is the largest column-sum of the matrix  $ABS(A(I, J))$  and  
 25 therefore has the value `MAXVAL (SUM (ABS(A), DIM = 1))`.

### 1 C.12.7 Logical queries

2 The intrinsic functions allow quite complicated questions about tabular data to be answered without  
3 use of loops or conditional constructs. Consider, for example, the questions asked below about a simple  
4 tabulation of students' test scores.

5 Suppose the rectangular table T (M, N) contains the test scores of M students who have taken N different  
6 tests. T is an integer matrix with entries in the range 0 to 100.

7 Example: The scores on 4 tests made by 3 students are held as the table

$$T = \begin{bmatrix} 85 & 76 & 90 & 60 \\ 71 & 45 & 50 & 80 \\ 66 & 45 & 21 & 55 \end{bmatrix}$$

8 Question: What is each student's top score?

9 Answer: MAXVAL (T, DIM = 2); in the example: [90, 80, 66].

10 Question: What is the average of all the scores?

11 Answer: SUM (T) / SIZE (T); in the example: 62.

12 Question: How many of the scores in the table are above average?

13 Answer: ABOVE = T > SUM (T) / SIZE (T); N = COUNT (ABOVE); in the example: ABOVE is the  
14 logical array (t = true, . = false):  $\begin{bmatrix} t & t & t & . \\ t & . & . & t \\ t & . & . & . \end{bmatrix}$  and COUNT (ABOVE) is 6.

15 Question: What was the lowest score in the above-average group of scores?

16 Answer: MINVAL (T, MASK = ABOVE), where ABOVE is as defined previously; in the example: 66.

17 Question: Was there a student whose scores were all above average?

18 Answer: With ABOVE as previously defined, the answer is yes or no according as the value of the  
19 expression ANY (ALL (ABOVE, DIM = 2)) is true or false; in the example, the answer is no.

### 20 C.12.8 Parallel computations

21 The most straightforward kind of parallel processing is to do the same thing at the same time to many  
22 operands. Matrix addition is a good example of this very simple form of parallel processing. Thus, the  
23 array assignment A = B + C specifies that corresponding elements of the identically-shaped arrays B  
24 and C be added together in parallel and that the resulting sums be assigned in parallel to the array A.

25 The process being done in parallel in the example of matrix addition is of course the process of addi-  
26 tion; the array feature that implements matrix addition as a parallel process is the element-by-element  
27 evaluation of array expressions.

28 These observations lead us to look to element-by-element computation as a means of implementing other  
29 simple parallel processing algorithms.

### 1 C.12.9 Example of element-by-element computation

2 Several polynomials of the same degree may be evaluated at the same point by arranging their coefficients  
 3 as the rows of a matrix and applying Horner's method for polynomial evaluation to the columns of the  
 4 matrix so formed.

5 The procedure is illustrated by the code to evaluate the three cubic polynomials

$$P(t) = 1 + 2t - 3t^2 + 4t^3$$

$$Q(t) = 2 - 3t + 4t^2 - 5t^3$$

$$R(t) = 3 + 4t - 5t^2 + 6t^3$$

6 in parallel at the point  $t = X$  and to place the resulting vector of numbers  $[P(X), Q(X), R(X)]$  in the  
 7 real array RESULT (3).

8 The code to compute RESULT is just the one statement

9 `RESULT = M (:, 1) + X * (M (:, 2) + X * (M (:, 3) + X * M (:, 4)))`

10 where M represents the matrix M (3, 4) with value  $\begin{bmatrix} 1 & 2 & -3 & 4 \\ 2 & -3 & 4 & -5 \\ 3 & 4 & -5 & 6 \end{bmatrix}$ .

### 11 C.12.10 Bit manipulation and inquiry procedures

12 The procedures IOR, IAND, NOT, IEOR, ISHFT, ISHFTC, IBITS, MVBITS, BTEST, IBSET, and  
 13 IBCLR are defined by MIL-STD 1753 for scalar arguments and are extended in this standard to accept  
 14 array arguments and to return array results.

## Annex D

(Informative)

### Syntax rules

#### D.1 Extract of all syntax rules

##### Section 1:

R101 *xyz-list* is *xyz* [ , *xyz* ] ...

R102 *xyz-name* is *name*

R103 *scalar-xyz* is *xyz*

C101 (R103) *scalar-xyz* shall be scalar.

##### Section 2:

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

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

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

R204 *specification-part* is [ *use-stmt* ] ...  
[ *import-stmt* ] ...  
[ *implicit-part* ]  
[ *declaration-construct* ] ...

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

R206 *implicit-part-stmt* is *implicit-stmt*  
or *parameter-stmt*  
or *format-stmt*  
or *entry-stmt*

R207 *declaration-construct* is *derived-type-def*  
or *entry-stmt*  
or *enum-def*  
or *format-stmt*  
or *interface-block*  
or *parameter-stmt*  
or *procedure-declaration-stmt*  
or *specification-stmt*  
or *type-declaration-stmt*  
or *stmt-function-stmt*

R208 *execution-part* is *executable-construct*  
[ *execution-part-construct* ] ...

- R209 *execution-part-construct* is *executable-construct*  
or *format-stmt*  
or *entry-stmt*  
or *data-stmt*
- R210 *internal-subprogram-part* is *contains-stmt*  
*internal-subprogram*  
[ *internal-subprogram* ] ...
- R211 *internal-subprogram* is *function-subprogram*  
or *subroutine-subprogram*
- R212 *specification-stmt* is *access-stmt*  
or *allocatable-stmt*  
or *asynchronous-stmt*  
or *bind-stmt*  
or *common-stmt*  
or *data-stmt*  
or *dimension-stmt*  
or *equivalence-stmt*  
or *external-stmt*  
or *intent-stmt*  
or *intrinsic-stmt*  
or *namelist-stmt*  
or *optional-stmt*  
or *pointer-stmt*  
or *protected-stmt*  
or *save-stmt*  
or *target-stmt*  
or *volatile-stmt*  
or *value-stmt*
- R213 *executable-construct* is *action-stmt*  
or *associate-construct*  
or *case-construct*  
or *do-construct*  
or *forall-construct*  
or *if-construct*  
or *select-type-construct*  
or *where-construct*
- R214 *action-stmt* is *allocate-stmt*  
or *assignment-stmt*  
or *backspace-stmt*  
or *call-stmt*  
or *close-stmt*  
or *continue-stmt*  
or *cycle-stmt*  
or *deallocate-stmt*  
or *endfile-stmt*  
or *end-function-stmt*

or *end-program-stmt*  
 or *end-subroutine-stmt*  
 or *exit-stmt*  
 or *flush-stmt*  
 or *forall-stmt*  
 or *goto-stmt*  
 or *if-stmt*  
 or *inquire-stmt*  
 or *nullify-stmt*  
 or *open-stmt*  
 or *pointer-assignment-stmt*  
 or *print-stmt*  
 or *read-stmt*  
 or *return-stmt*  
 or *rewind-stmt*  
 or *stop-stmt*  
 or *wait-stmt*  
 or *where-stmt*  
 or *write-stmt*  
 or *arithmetic-if-stmt*  
 or *computed-goto-stmt*

C201 (R208) An *execution-part* shall not contain an *end-function-stmt*, *end-program-stmt*, or *end-subroutine-stmt*.

R215 *keyword* is *name*

### Section 3:

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

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

R303 *underscore* is *-*

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

C301 (R304) The maximum length of a *name* is 63 characters.

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

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

R307 *named-constant* is *name*

R308 *int-constant* is *constant*

C302 (R308) *int-constant* shall be of type integer.

R309 *char-constant* is *constant*

C303 (R309) *char-constant* shall be of type character.

- R310 *intrinsic-operator* is *power-op*  
or *mult-op*  
or *add-op*  
or *concat-op*  
or *rel-op*  
or *not-op*  
or *and-op*  
or *or-op*  
or *equiv-op*
- R311 *defined-operator* is *defined-unary-op*  
or *defined-binary-op*  
or *extended-intrinsic-op*
- R312 *extended-intrinsic-op* is *intrinsic-operator*
- R313 *label* is *digit* [ *digit* [ *digit* [ *digit* [ *digit* ] ] ] ]
- C304 (R313) At least one digit in a *label* shall be nonzero.

**Section 4:**

- R401 *type-spec* is *intrinsic-type-spec*  
or *derived-type-spec*
- C401 (R401) The *derived-type-spec* shall not specify an abstract type (4.5.6).
- R402 *type-param-value* is *scalar-int-expr*  
or \*  
or :
- C402 (R402) The *type-param-value* for a kind type parameter shall be an initialization expression.
- C403 (R402) A colon may be used as a *type-param-value* only in the declaration of an entity or component that has the POINTER or ALLOCATABLE attribute.
- R403 *intrinsic-type-spec* is INTEGER [ *kind-selector* ]  
or REAL [ *kind-selector* ]  
or DOUBLE PRECISION  
or COMPLEX [ *kind-selector* ]  
or CHARACTER [ *char-selector* ]  
or LOGICAL [ *kind-selector* ]
- R404 *kind-selector* is ( [ KIND = ] *scalar-int-initialization-expr* )
- C404 (R404) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.
- R405 *signed-int-literal-constant* is [ *sign* ] *int-literal-constant*
- R406 *int-literal-constant* is *digit-string* [ \_ *kind-param* ]
- R407 *kind-param* is *digit-string*  
or *scalar-int-constant-name*
- R408 *signed-digit-string* is [ *sign* ] *digit-string*
- R409 *digit-string* is *digit* [ *digit* ] ...
- R410 *sign* is +  
or -
- C405 (R407) A *scalar-int-constant-name* shall be a named constant of type integer.
- C406 (R407) The value of *kind-param* shall be nonnegative.
- C407 (R406) The value of *kind-param* shall specify a representation method that exists on the processor.

- R411 *boz-literal-constant* is *binary-constant*  
or *octal-constant*  
or *hex-constant*
- R412 *binary-constant* is B ' *digit* [ *digit* ] ... '  
or B " *digit* [ *digit* ] ... "
- C408 (R412) *digit* shall have one of the values 0 or 1.
- R413 *octal-constant* is O ' *digit* [ *digit* ] ... '  
or O " *digit* [ *digit* ] ... "
- C409 (R413) *digit* shall have one of the values 0 through 7.
- R414 *hex-constant* is Z ' *hex-digit* [ *hex-digit* ] ... '  
or Z " *hex-digit* [ *hex-digit* ] ... "
- R415 *hex-digit* is *digit*  
or A  
or B  
or C  
or D  
or E  
or F
- C410 (R411) A *boz-literal-constant* shall appear only as a *data-stmt-constant* in a DATA statement, as the actual argument associated with the dummy argument A of the numeric intrinsic functions DBLE, REAL or INT, or as the actual argument associated with the X or Y dummy argument of the intrinsic function CMPLX.
- R416 *signed-real-literal-constant* is [ *sign* ] *real-literal-constant*
- R417 *real-literal-constant* is *significand* [ *exponent-letter* *exponent* ] [ *\_ kind-param* ]  
or *digit-string* *exponent-letter* *exponent* [ *\_ kind-param* ]
- R418 *significand* is *digit-string* . [ *digit-string* ]  
or . *digit-string*
- R419 *exponent-letter* is E  
or D
- R420 *exponent* is *signed-digit-string*
- C411 (R417) If both *kind-param* and *exponent-letter* are present, *exponent-letter* shall be E.
- C412 (R417) The value of *kind-param* shall specify an approximation method that exists on the processor.
- R421 *complex-literal-constant* is ( *real-part* , *imag-part* )
- R422 *real-part* is *signed-int-literal-constant*  
or *signed-real-literal-constant*  
or *named-constant*
- R423 *imag-part* is *signed-int-literal-constant*  
or *signed-real-literal-constant*  
or *named-constant*
- C413 (R421) Each named constant in a complex literal constant shall be of type integer or real.
- R424 *char-selector* is *length-selector*  
or ( LEN = *type-param-value* , ■  
■ KIND = *scalar-int-initialization-expr* )  
or ( *type-param-value* , ■  
■ [ KIND = ] *scalar-int-initialization-expr* )  
or ( KIND = *scalar-int-initialization-expr* ■

- R425 *length-selector*                    **is** ( [ LEN = ] *type-param-value* )  
R426 *char-length*                        **is** ( *type-param-value* )  
R427 *char-literal-constant*            **is** [ *kind-param* \_ ] ' [ *rep-char* ] ... '  
R428 *logical-literal-constant*        **is** .TRUE. [ \_ *kind-param* ]  
R429 *derived-type-def*                    **is** *derived-type-stmt*  
R430 *derived-type-stmt*                    **is** TYPE [ [ , *type-attr-spec-list* ] :: ] *type-name* ■  
R431 *type-attr-spec*                        **is** *access-spec*  
R432 *private-or-sequence*                **is** *private-components-stmt*
- R425 *length-selector*                    **is** ( [ LEN = ] *type-param-value* )  
R426 *char-length*                        **is** ( *type-param-value* )  
R427 *char-literal-constant*            **is** [ *kind-param* \_ ] ' [ *rep-char* ] ... '  
R428 *logical-literal-constant*        **is** .TRUE. [ \_ *kind-param* ]  
R429 *derived-type-def*                    **is** *derived-type-stmt*  
R430 *derived-type-stmt*                    **is** TYPE [ [ , *type-attr-spec-list* ] :: ] *type-name* ■  
R431 *type-attr-spec*                        **is** *access-spec*  
R432 *private-or-sequence*                **is** *private-components-stmt*
- C414 (R424) The value of *scalar-int-initialization-expr* shall be nonnegative and shall specify a representation method that exists on the processor.  
 C415 (R426) The *scalar-int-literal-constant* shall not include a *kind-param*.  
 C416 (R424 R425 R426) A *type-param-value* of \* may be used only in the following ways:  
 C417 A function name shall not be declared with an asterisk *type-param-value* unless it is of type CHARACTER and is the name of the result of an external function or the name of a dummy function.  
 C418 A function name declared with an asterisk *type-param-value* shall not be an array, a pointer, recursive, or pure.  
 C419 (R425) The optional comma in a *length-selector* is permitted only in a *declaration-type-spec* in a *type-declaration-stmt*.  
 C420 (R425) The optional comma in a *length-selector* is permitted only if no double-colon separator appears in the *type-declaration-stmt*.  
 C421 (R424) The length specified for a character statement function or for a statement function dummy argument of type character shall be an initialization expression.  
 C422 (R427) The value of *kind-param* shall specify a representation method that exists on the processor.  
 C423 (R428) The value of *kind-param* shall specify a representation method that exists on the processor.  
 C424 (R430) A derived type *type-name* shall not be DOUBLEPRECISION or the same as the name of any intrinsic type defined in this standard.  
 C425 (R430) The same *type-attr-spec* shall not appear more than once in a given *derived-type-stmt*.  
 C426 (R431) A *parent-type-name* shall be the name of an accessible extensible type (4.5.6).  
 C427 (R429) If the type definition contains or inherits (4.5.6.1) a deferred binding (4.5.4), ABSTRACT shall appear.  
 C428 (R429) If ABSTRACT appears, the type shall be extensible.  
 C429 (R429) If EXTENDS appears, SEQUENCE shall not appear.

- or** *sequence-stmt*
- C430 (R429) The same *private-or-sequence* shall not appear more than once in a given *derived-type-def*.
- R433 *end-type-stmt* **is** END TYPE [ *type-name* ]
- C431 (R433) If END TYPE is followed by a *type-name*, the *type-name* shall be the same as that in the corresponding *derived-type-stmt*.
- R434 *sequence-stmt* **is** SEQUENCE
- C432 (R438) If SEQUENCE appears, all derived types specified in component definitions shall be sequence types.
- C433 (R429) If SEQUENCE appears, a *type-bound-procedure-part* shall not appear.
- R435 *type-param-def-stmt* **is** INTEGER [ *kind-selector* ], *type-param-attr-spec* :: ■  
 ■ *type-param-decl-list*
- R436 *type-param-decl* **is** *type-param-name* [ = *scalar-int-initialization-expr* ]
- C434 (R435) A *type-param-name* in a *type-param-def-stmt* in a *derived-type-def* shall be one of the *type-param-names* in the *derived-type-stmt* of that *derived-type-def*.
- C435 (R435) Each *type-param-name* in the *derived-type-stmt* in a *derived-type-def* shall appear as a *type-param-name* in a *type-param-def-stmt* in that *derived-type-def*.
- R437 *type-param-attr-spec* **is** KIND  
**or** LEN
- R438 *component-part* **is** [ *component-def-stmt* ] ...
- R439 *component-def-stmt* **is** *data-component-def-stmt*  
**or** *proc-component-def-stmt*
- R440 *data-component-def-stmt* **is** *declaration-type-spec* [ [ , *component-attr-spec-list* ] :: ] ■  
 ■ *component-decl-list*
- R441 *component-attr-spec* **is** POINTER  
**or** DIMENSION ( *component-array-spec* )  
**or** ALLOCATABLE  
**or** *access-spec*
- R442 *component-decl* **is** *component-name* [ ( *component-array-spec* ) ] ■  
 ■ [ \* *char-length* ] [ *component-initialization* ]
- R443 *component-array-spec* **is** *explicit-shape-spec-list*  
**or** *deferred-shape-spec-list*
- R444 *component-initialization* **is** = *initialization-expr*  
**or** => *null-init*
- C436 (R440) No *component-attr-spec* shall appear more than once in a given *component-def-stmt*.
- C437 (R440) A component declared with the CLASS keyword (5.1.1.2) shall have the ALLOCATABLE or POINTER attribute.
- C438 (R440) If the POINTER attribute is not specified for a component, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type or a previously defined derived type.
- C439 (R440) If the POINTER attribute is specified for a component, the *declaration-type-spec* in the *component-def-stmt* shall specify an intrinsic type or any accessible derived type including the type being defined.
- C440 (R440) If the POINTER or ALLOCATABLE attribute is specified, each *component-array-spec* shall be a *deferred-shape-spec-list*.
- C441 (R440) If neither the POINTER attribute nor the ALLOCATABLE attribute is specified, each *component-array-spec* shall be an *explicit-shape-spec-list*.
- C442 (R443) Each bound in the *explicit-shape-spec* shall either be an initialization expression or be a specification expression that does not contain references to specification functions or any object



procedure that has an explicit interface.

- R452** *generic-binding*                    **is** **GENERIC** ■  
    ■ [ *access-spec* ] :: *generic-spec* => *binding-name-list*
- C459 (R452) Within the *specification-part* of a module, each *generic-binding* shall specify, either implicitly or explicitly, the same accessibility as every other *generic-binding* with that *generic-spec* in the same derived type.
- C460 (R452) Each *binding-name* in *binding-name-list* shall be the name of a specific binding of the type.
- C461 (R452) If *generic-spec* is not *generic-name*, each of its specific bindings shall have a passed-object dummy argument (4.5.3.3).
- C462 (R452) If *generic-spec* is OPERATOR ( *defined-operator* ), the interface of each binding shall be as specified in 12.3.2.1.1.
- C463 (R452) If *generic-spec* is ASSIGNMENT ( = ), the interface of each binding shall be as specified in 12.3.2.1.2.
- C464 (R452) If *generic-spec* is *dtio-generic-spec*, the interface of each binding shall be as specified in 9.5.3.7. The type of the *dtv* argument shall be *type-name*.
- R453** *binding-attr*                        **is** PASS [ (*arg-name*) ]  
    **or** NOPASS  
    **or** NON\_OVERRIDABLE  
    **or** DEFERRED  
    **or** *access-spec*
- C465 (R453) The same *binding-attr* shall not appear more than once in a given *binding-attr-list*.
- C466 (R451) If the interface of the binding has no dummy argument of the type being defined, NOPASS shall appear.
- C467 (R451) If PASS (*arg-name*) appears, the interface of the binding shall have a dummy argument named *arg-name*.
- C468 (R453) PASS and NOPASS shall not both appear in the same *binding-attr-list*.
- C469 (R453) NON\_OVERRIDABLE and DEFERRED shall not both appear in the same *binding-attr-list*.
- C470 (R453) DEFERRED shall appear if and only if *interface-name* appears.
- C471 (R451) An overriding binding (4.5.6.2) shall have the DEFERRED attribute only if the binding it overrides is deferred.
- C472 (R451) A binding shall not override an inherited binding (4.5.6.1) that has the NON\_OVERRIDABLE attribute.
- R454** *final-binding*                        **is** FINAL [ :: ] *final-subroutine-name-list*
- C473 (R454) A *final-subroutine-name* shall be the name of a module procedure with exactly one dummy argument. That argument shall be nonoptional and shall be a nonpointer, nonallocatable, nonpolymorphic variable of the derived type being defined. All length type parameters of the dummy argument shall be assumed. The dummy argument shall not be INTENT(OUT).
- C474 (R454) A *final-subroutine-name* shall not be one previously specified as a final subroutine for that type.
- C475 (R454) A final subroutine shall not have a dummy argument with the same kind type parameters and rank as the dummy argument of another final subroutine of that type.
- R455** *derived-type-spec*                    **is** *type-name* [ ( *type-param-spec-list* ) ]
- R456** *type-param-spec*                    **is** [ *keyword* = ] *type-param-value*
- C476 (R455) *type-name* shall be the name of an accessible derived type.
- C477 (R455) *type-param-spec-list* shall appear only if the type is parameterized.
- C478 (R455) There shall be at most one *type-param-spec* corresponding to each parameter of the type. If a type parameter does not have a default value, there shall be a *type-param-spec* corresponding

to that type parameter.

- C479 (R456) The *keyword=* may be omitted from a *type-param-spec* only if the *keyword=* has been omitted from each preceding *type-param-spec* in the *type-param-spec-list*.
- C480 (R456) Each *keyword* shall be the name of a parameter of the type.
- C481 (R456) An asterisk may be used as a *type-param-value* in a *type-param-spec* only in the declaration of a dummy argument or associate name or in the allocation of a dummy argument.
- R457 *structure-constructor*        **is** *derived-type-spec* ( [ *component-spec-list* ] )
- R458 *component-spec*                **is** [ *keyword =* ] *component-data-source*
- R459 *component-data-source*       **is** *expr*  
    **or** *data-target*  
    **or** *proc-target*
- C482 (R457) The *derived-type-spec* shall not specify an abstract type (4.5.6).
- C483 (R457) At most one *component-spec* shall be provided for a component.
- C484 (R457) If a *component-spec* is provided for a component, no *component-spec* shall be provided for any component with which it is inheritance associated.
- C485 (R457) A *component-spec* shall be provided for a component unless it has default initialization or is inheritance associated with another component for which a *component-spec* is provided or that has default initialization.
- C486 (R458) The *keyword=* may be omitted from a *component-spec* only if the *keyword=* has been omitted from each preceding *component-spec* in the constructor.
- C487 (R458) Each *keyword* shall be the name of a component of the type.
- C488 (R457) The type name and all components of the type for which a *component-spec* appears shall be accessible in the scoping unit containing the structure constructor.
- C489 (R457) If *derived-type-spec* is a type name that is the same as a generic name, the *component-spec-list* shall not be a valid *actual-arg-spec-list* for a function reference that is resolvable as a generic reference (12.4.4.1).
- C490 (R459) A *data-target* shall correspond to a nonprocedure pointer component; a *proc-target* shall correspond to a procedure pointer component.
- C491 (R459) A *data-target* shall have the same rank as its corresponding component.
- R460 *enum-def*                        **is** *enum-def-stmt*  
    *enumerator-def-stmt*  
    [ *enumerator-def-stmt* ] ...  
    *end-enum-stmt*
- R461 *enum-def-stmt*                    **is** ENUM, BIND(C)
- R462 *enumerator-def-stmt*            **is** ENUMERATOR [ :: ] *enumerator-list*
- R463 *enumerator*                      **is** *named-constant* [ = *scalar-int-initialization-expr* ]
- R464 *end-enum-stmt*                   **is** END ENUM
- C492 (R462) If = appears in an *enumerator*, a double-colon separator shall appear before the *enumerator-list*.
- R465 *array-constructor*              **is** (/ *ac-spec* /)  
    **or** *left-square-bracket ac-spec right-square-bracket*
- R466 *ac-spec*                            **is** *type-spec* ::  
    **or** [*type-spec* ::] *ac-value-list*
- R467 *left-square-bracket*              **is** [  
 R468 *right-square-bracket*            **is** ]
- R469 *ac-value*                          **is** *expr*  
    **or** *ac-implied-do*
- R470 *ac-implied-do*                   **is** ( *ac-value-list* , *ac-implied-do-control* )

- R471 *ac-implied-do-control* is *ac-do-variable* = *scalar-int-expr* , *scalar-int-expr* ■  
 ■ [ , *scalar-int-expr* ]
- R472 *ac-do-variable* is *scalar-int-variable*
- C493 (R472) *ac-do-variable* shall be a named variable.
- C494 (R466) If *type-spec* is omitted, each *ac-value* expression in the *array-constructor* shall have the same type and kind type parameters.
- C495 (R466) If *type-spec* specifies an intrinsic type, each *ac-value* expression in the *array-constructor* shall be of an intrinsic type that is in type conformance with a variable of type *type-spec* as specified in Table 7.8.
- C496 (R466) If *type-spec* specifies a derived type, all *ac-value* expressions in the *array-constructor* shall be of that derived type and shall have the same kind type parameter values as specified by *type-spec*.
- C497 (R470) The *ac-do-variable* of an *ac-implied-do* that is in another *ac-implied-do* shall not appear as the *ac-do-variable* of the containing *ac-implied-do*.

### Section 5:

- R501 *type-declaration-stmt* is *declaration-type-spec* [ [ , *attr-spec* ] ... :: ] *entity-decl-list*
- R502 *declaration-type-spec* is *intrinsic-type-spec*  
 or TYPE ( *derived-type-spec* )  
 or CLASS ( *derived-type-spec* )  
 or CLASS ( \* )
- C501 (R502) In a *declaration-type-spec*, every *type-param-value* that is not a colon or an asterisk shall be a *specification-expr*.
- C502 (R502) In a *declaration-type-spec* that uses the CLASS keyword, *derived-type-spec* shall specify an extensible type.
- C503 (R502) The TYPE(*derived-type-spec*) shall not specify an abstract type (4.5.6).
- R503 *attr-spec* is *access-spec*  
 or ALLOCATABLE  
 or ASYNCHRONOUS  
 or DIMENSION ( *array-spec* )  
 or EXTERNAL  
 or INTENT ( *intent-spec* )  
 or INTRINSIC  
 or *language-binding-spec*  
 or OPTIONAL  
 or PARAMETER  
 or POINTER  
 or PROTECTED  
 or SAVE  
 or TARGET  
 or VALUE  
 or VOLATILE
- R504 *entity-decl* is *object-name* [ ( *array-spec* ) ] [ \* *char-length* ] [ *initialization* ]  
 or *function-name* [ \* *char-length* ]
- C504 (R504) If a *type-param-value* in a *char-length* in an *entity-decl* is not a colon or an asterisk, it shall be a *specification-expr*.
- R505 *object-name* is *name*
- C505 (R505) The *object-name* shall be the name of a data object.

- R506 *initialization*                    **is** = *initialization-expr*  
    **or** => *null-init*
- R507 *null-init*                            **is** *function-reference*
- C506 (R507) The *function-reference* shall be a reference to the NULL intrinsic function with no arguments.
- C507 (R501) The same *attr-spec* shall not appear more than once in a given *type-declaration-stmt*.
- C508 An entity shall not be explicitly given any attribute more than once in a scoping unit.
- C509 (R501) An entity declared with the CLASS keyword shall be a dummy argument or have the ALLOCATABLE or POINTER attribute.
- C510 (R501) An array that has the POINTER or ALLOCATABLE attribute shall be specified with an *array-spec* that is a *deferred-shape-spec-list* (5.1.2.5.3).
- C511 (R501) An *array-spec* for an *object-name* that is a function result that does not have the ALLOCATABLE or POINTER attribute shall be an *explicit-shape-spec-list*.
- C512 (R501) If the POINTER attribute is specified, the ALLOCATABLE, TARGET, EXTERNAL, or INTRINSIC attribute shall not be specified.
- C513 (R501) If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.
- C514 (R501) The PARAMETER attribute shall not be specified for a dummy argument, a pointer, an allocatable entity, a function, or an object in a common block.
- C515 (R501) The INTENT, VALUE, and OPTIONAL attributes may be specified only for dummy arguments.
- C516 (R501) The INTENT attribute shall not be specified for a dummy procedure without the POINTER attribute.
- C517 (R501) The SAVE attribute shall not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, an automatic data object, or an object with the PARAMETER attribute.
- C518 An entity shall not have both the EXTERNAL attribute and the INTRINSIC attribute.
- C519 (R501) An entity in an *entity-decl-list* shall not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.
- C520 (R504) The \* *char-length* option is permitted only if the type specified is character.
- C521 (R504) The *function-name* shall be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.
- C522 (R501) The *initialization* shall appear if the statement contains a PARAMETER attribute (5.1.2.10).
- C523 (R501) If *initialization* appears, a double-colon separator shall appear before the *entity-decl-list*.
- C524 (R504) *initialization* shall not appear if *object-name* is a dummy argument, a function result, an object in a named common block unless the type declaration is in a block data program unit, an object in blank common, an allocatable variable, an external name, an intrinsic name, or an automatic object.
- C525 (R504) If => appears in *initialization*, the object shall have the POINTER attribute. If = appears in *initialization*, the object shall not have the POINTER attribute.
- C526 (R501) If the VOLATILE attribute is specified, the PARAMETER, INTRINSIC, EXTERNAL, or INTENT(IN) attribute shall not be specified.
- C527 (R501) If the VALUE attribute is specified, the PARAMETER, EXTERNAL, POINTER, ALLOCATABLE, DIMENSION, VOLATILE, INTENT(INOUT), or INTENT(OUT) attribute shall not be specified.
- C528 (R501) If the VALUE attribute is specified for a dummy argument of type character, the length parameter shall be omitted or shall be specified by an initialization expression with the value

- one.
- C529 (R501) The VALUE attribute shall not be specified for a dummy procedure.
- C530 (R501) The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy argument of a procedure that has a *proc-language-binding-spec*.
- C531 (R503) A *language-binding-spec* shall appear only in the specification part of a module.
- C532 (R501) If a *language-binding-spec* is specified, the entity declared shall be an interoperable variable (15.2).
- C533 (R501) If a *language-binding-spec* with a NAME= specifier appears, the *entity-decl-list* shall consist of a single *entity-decl*.
- C534 (R503) The PROTECTED attribute is permitted only in the specification part of a module.
- C535 (R501) The PROTECTED attribute is permitted only for a procedure pointer or named variable that is not in a common block.
- C536 (R501) If the PROTECTED attribute is specified, the EXTERNAL, INTRINSIC, or PARAMETER attribute shall not be specified.
- C537 A nonpointer object that has the PROTECTED attribute and is accessed by use association shall not appear in a variable definition context (16.5.7) or as the *data-target* or *proc-target* in a *pointer-assignment-stmt*.
- C538 A pointer object that has the PROTECTED attribute and is accessed by use association shall not appear as
- (1) A *pointer-object* in a *pointer-assignment-stmt* or *nullify-stmt*,
  - (2) An *allocate-object* in an *allocate-stmt* or *deallocate-stmt*, or
  - (3) An actual argument in a reference to a procedure if the associated dummy argument is a pointer with the INTENT(OUT) or INTENT(INOUT) attribute.
- R508 *access-spec*                               is PUBLIC  
  or PRIVATE
- C539 (R508) An *access-spec* shall appear only in the *specification-part* of a module.
- R509 *language-binding-spec*               is BIND (C [, NAME = *scalar-char-initialization-expr* ])
- C540 (R509) The *scalar-char-initialization-expr* shall be of default character kind.
- R510 *array-spec*                               is *explicit-shape-spec-list*  
  or *assumed-shape-spec-list*  
  or *deferred-shape-spec-list*  
  or *assumed-size-spec*
- C541 (R510) The maximum rank is seven.
- R511 *explicit-shape-spec*                   is [ *lower-bound* : ] *upper-bound*
- R512 *lower-bound*                            is *specification-expr*
- R513 *upper-bound*                            is *specification-expr*
- C542 (R511) An explicit-shape array whose bounds are not initialization expressions shall be a dummy argument, a function result, or an automatic array of a procedure.
- R514 *assumed-shape-spec*                   is [ *lower-bound* ] :
- R515 *deferred-shape-spec*                   is :
- R516 *assumed-size-spec*                    is [ *explicit-shape-spec-list* , ] [ *lower-bound* : ] \*
- C543 An *assumed-size-spec* shall not appear except as the declaration of the array bounds of a dummy data argument.
- C544 An assumed-size array with INTENT (OUT) shall not be of a type for which default initialization is specified.
- R517 *intent-spec*                           is IN  
  or OUT

or INOUT

- C545 (R517) A nonpointer object with the INTENT (IN) attribute shall not appear in a variable definition context (16.5.7).
- C546 (R517) A pointer object with the INTENT (IN) attribute shall not appear as
- C547 (R503) (R1216) If the name of a generic intrinsic procedure is explicitly declared to have the INTRINSIC attribute, and it is also the generic name in one or more generic interfaces (12.3.2.1) accessible in the same scoping unit, the procedures in the interfaces and the specific intrinsic procedures shall all be functions or all be subroutines, and the characteristics of the specific intrinsic procedures and the procedures in the interfaces shall differ as specified in 16.2.3.
- R518 *access-stmt* is *access-spec* [ [ :: ] *access-id-list* ]
- R519 *access-id* is *use-name*  
or *generic-spec*
- C548 (R518) An *access-stmt* shall appear only in the *specification-part* of a module. Only one accessibility statement with an omitted *access-id-list* is permitted in the *specification-part* of a module.
- C549 (R519) Each *use-name* shall be the name of a named variable, procedure, derived type, named constant, or namelist group.
- R520 *allocatable-stmt* is ALLOCATABLE [ [ :: ] ■  
■ *object-name* [ ( *deferred-shape-spec-list* ) ] ■  
■ [ , *object-name* [ ( *deferred-shape-spec-list* ) ] ] ...
- R521 *asynchronous-stmt* is ASYNCHRONOUS [ [ :: ] *object-name-list*
- R522 *bind-stmt* is *language-binding-spec* [ [ :: ] *bind-entity-list*
- R523 *bind-entity* is *entity-name*  
or / *common-block-name* /
- C550 (R522) If any *bind-entity* in a *bind-stmt* is an *entity-name*, the *bind-stmt* shall appear in the specification part of a module and the entity shall be an interoperable variable (15.2.4, 15.2.5).
- C551 (R522) If the *language-binding-spec* has a NAME= specifier, the *bind-entity-list* shall consist of a single *bind-entity*.
- C552 (R522) If a *bind-entity* is a common block, each variable of the common block shall be interoperable (15.2.4, 15.2.5).
- R524 *data-stmt* is DATA *data-stmt-set* [ [ , ] *data-stmt-set* ] ...
- R525 *data-stmt-set* is *data-stmt-object-list* / *data-stmt-value-list* /
- R526 *data-stmt-object* is *variable*  
or *data-implied-do*
- R527 *data-implied-do* is ( *data-i-do-object-list* , *data-i-do-variable* = ■  
■ *scalar-int-expr* , *scalar-int-expr* [ , *scalar-int-expr* ] )
- R528 *data-i-do-object* is *array-element*  
or *scalar-structure-component*  
or *data-implied-do*
- R529 *data-i-do-variable* is *scalar-int-variable*
- C553 (R526) In a *variable* that is a *data-stmt-object*, any subscript, section subscript, substring starting point, and substring ending point shall be an initialization expression.
- C554 (R526) A variable whose designator is included in a *data-stmt-object-list* or a *data-i-do-object-list* shall not be: a dummy argument, made accessible by use association or host association, in a named common block unless the DATA statement is in a block data program unit, in a blank common block, a function name, a function result name, an automatic object, or an allocatable variable.
- C555 (R526) A *data-i-do-object* or a *variable* that appears as a *data-stmt-object* shall not be an object

designator in which a pointer appears other than as the entire rightmost *part-ref*.

- C556 (R529) The *data-i-do-variable* shall be a named variable.
- C557 (R527) A *scalar-int-expr* of a *data-implied-do* shall involve as primaries only constants, subobjects of constants, or DO variables of the containing *data-implied-dos*, and each operation shall be intrinsic.
- C558 (R528) The *array-element* shall be a variable.
- C559 (R528) The *scalar-structure-component* shall be a variable.
- C560 (R528) The *scalar-structure-component* shall contain at least one *part-ref* that contains a *subscript-list*.
- C561 (R528) In an *array-element* or a *scalar-structure-component* that is a *data-i-do-object*, any subscript shall be an expression whose primaries are either constants, subobjects of constants, or DO variables of this *data-implied-do* or the containing *data-implied-dos*, and each operation shall be intrinsic.
- R530 *data-stmt-value* is [ *data-stmt-repeat* \* ] *data-stmt-constant*
- R531 *data-stmt-repeat* is *scalar-int-constant*  
or *scalar-int-constant-subobject*
- C562 (R531) The *data-stmt-repeat* shall be positive or zero. If the *data-stmt-repeat* is a named constant, it shall have been declared previously in the scoping unit or made accessible by use association or host association.
- R532 *data-stmt-constant* is *scalar-constant*  
or *scalar-constant-subobject*  
or *signed-int-literal-constant*  
or *signed-real-literal-constant*  
or *null-init*  
or *structure-constructor*
- C563 (R532) If a DATA statement constant value is a named constant or a structure constructor, the named constant or derived type shall have been declared previously in the scoping unit or made accessible by use or host association.
- C564 (R532) If a *data-stmt-constant* is a *structure-constructor*, it shall be an initialization expression.
- R533 *int-constant-subobject* is *constant-subobject*
- C565 (R533) *int-constant-subobject* shall be of type integer.
- R534 *constant-subobject* is *designator*
- C566 (R534) *constant-subobject* shall be a subobject of a constant.
- C567 (R534) Any subscript, substring starting point, or substring ending point shall be an initialization expression.
- R535 *dimension-stmt* is DIMENSION [ :: ] *array-name* ( *array-spec* ) ■  
■ [ , *array-name* ( *array-spec* ) ] ...
- R536 *intent-stmt* is INTENT ( *intent-spec* ) [ :: ] *dummy-arg-name-list*
- R537 *optional-stmt* is OPTIONAL [ :: ] *dummy-arg-name-list*
- R538 *parameter-stmt* is PARAMETER ( *named-constant-def-list* )
- R539 *named-constant-def* is *named-constant* = *initialization-expr*
- R540 *pointer-stmt* is POINTER [ :: ] *pointer-decl-list*
- R541 *pointer-decl* is *object-name* [ ( *deferred-shape-spec-list* ) ]  
or *proc-entity-name*
- C568 (R541) A *proc-entity-name* shall also be declared in a *procedure-declaration-stmt*.
- R542 *protected-stmt* is PROTECTED [ :: ] *entity-name-list*
- R543 *save-stmt* is SAVE [ [ :: ] *saved-entity-list* ]
- R544 *saved-entity* is *object-name*

- or** *proc-pointer-name*  
**or** / *common-block-name* /
- R545 *proc-pointer-name* **is** *name*
- C569 (R545) A *proc-pointer-name* shall be the name of a procedure pointer.
- C570 (R543) If a SAVE statement with an omitted saved entity list occurs in a scoping unit, no other explicit occurrence of the SAVE attribute or SAVE statement is permitted in the same scoping unit.
- R546 *target-stmt* **is** TARGET [ :: ] *object-name* [ ( *array-spec* ) ] ■  
 ■ [ , *object-name* [ ( *array-spec* ) ] ] ...
- R547 *value-stmt* **is** VALUE [ :: ] *dummy-arg-name-list*
- R548 *volatile-stmt* **is** VOLATILE [ :: ] *object-name-list*
- R549 *implicit-stmt* **is** IMPLICIT *implicit-spec-list*  
**or** IMPLICIT NONE
- R550 *implicit-spec* **is** *declaration-type-spec* ( *letter-spec-list* )
- R551 *letter-spec* **is** *letter* [ - *letter* ]
- C571 (R549) If IMPLICIT NONE is specified in a scoping unit, it shall precede any PARAMETER statements that appear in the scoping unit and there shall be no other IMPLICIT statements in the scoping unit.
- C572 (R551) If the minus and second letter appear, the second letter shall follow the first letter alphabetically.
- R552 *namelist-stmt* **is** NAMELIST ■  
 ■ / *namelist-group-name* / *namelist-group-object-list* ■  
 ■ [ [ , ] / *namelist-group-name* / ■  
 ■ *namelist-group-object-list* ] ...
- C573 (R552) The *namelist-group-name* shall not be a name made accessible by use association.
- R553 *namelist-group-object* **is** *variable-name*
- C574 (R553) A *namelist-group-object* shall not be an assumed-size array.
- C575 (R552) A *namelist-group-object* shall not have the PRIVATE attribute if the *namelist-group-name* has the PUBLIC attribute.
- R554 *equivalence-stmt* **is** EQUIVALENCE *equivalence-set-list*
- R555 *equivalence-set* **is** ( *equivalence-object* , *equivalence-object-list* )
- R556 *equivalence-object* **is** *variable-name*  
**or** *array-element*  
**or** *substring*
- C576 (R556) An *equivalence-object* shall not be a designator with a base object that is a dummy argument, a pointer, an allocatable variable, a derived-type object that has an allocatable ultimate component, an object of a nonsequence derived type, an object of a derived type that has a pointer at any level of component selection, an automatic object, a function name, an entry name, a result name, a variable with the BIND attribute, a variable in a common block that has the BIND attribute, or a named constant.
- C577 (R556) An *equivalence-object* shall not be a designator that has more than one *part-ref*.
- C578 (R556) An *equivalence-object* shall not have the TARGET attribute.
- C579 (R556) Each subscript or substring range expression in an *equivalence-object* shall be an integer initialization expression (7.1.7).
- C580 (R555) If an *equivalence-object* is of type default integer, default real, double precision real, default complex, default logical, or numeric sequence type, all of the objects in the equivalence set shall be of these types.
- C581 (R555) If an *equivalence-object* is of type default character or character sequence type, all of the

objects in the equivalence set shall be of these types.

- C582 (R555) If an *equivalence-object* is of a sequence derived type that is not a numeric sequence or character sequence type, all of the objects in the equivalence set shall be of the same type with the same type parameter values.
- C583 (R555) If an *equivalence-object* is of an intrinsic type other than default integer, default real, double precision real, default complex, default logical, or default character, all of the objects in the equivalence set shall be of the same type with the same kind type parameter value.
- C584 (R556) If an *equivalence-object* has the PROTECTED attribute, all of the objects in the equivalence set shall have the PROTECTED attribute.
- C585 (R556) The name of an *equivalence-object* shall not be a name made accessible by use association.
- C586 (R556) A *substring* shall not have length zero.
- R557 *common-stmt* is COMMON ■  
 ■ [ / [ *common-block-name* ] / ] *common-block-object-list* ■  
 ■ [ [ , ] / [ *common-block-name* ] / ■  
 ■ *common-block-object-list* ] ...
- R558 *common-block-object* is *variable-name* [ ( *explicit-shape-spec-list* ) ]  
 or *proc-pointer-name*
- C587 (R558) Only one appearance of a given *variable-name* or *proc-pointer-name* is permitted in all *common-block-object-lists* within a scoping unit.
- C588 (R558) A *common-block-object* shall not be a dummy argument, an allocatable variable, a derived-type object with an ultimate component that is allocatable, an automatic object, a function name, an entry name, a variable with the BIND attribute, or a result name.
- C589 (R558) If a *common-block-object* is of a derived type, it shall be a sequence type (4.5.1) or a type with the BIND attribute and it shall have no default initialization.
- C590 (R558) A *variable-name* or *proc-pointer-name* shall not be a name made accessible by use association.

## Section 6:

- R601 *variable* is *designator*
- C601 (R601) *designator* shall not be a constant or a subobject of a constant.
- R602 *variable-name* is *name*
- C602 (R602) A *variable-name* shall be the name of a variable.
- R603 *designator* is *object-name*  
 or *array-element*  
 or *array-section*  
 or *structure-component*  
 or *substring*
- R604 *logical-variable* is *variable*
- C603 (R604) *logical-variable* shall be of type logical.
- R605 *default-logical-variable* is *variable*
- C604 (R605) *default-logical-variable* shall be of type default logical.
- R606 *char-variable* is *variable*
- C605 (R606) *char-variable* shall be of type character.
- R607 *default-char-variable* is *variable*
- C606 (R607) *default-char-variable* shall be of type default character.
- R608 *int-variable* is *variable*
- C607 (R608) *int-variable* shall be of type integer.
- R609 *substring* is *parent-string* ( *substring-range* )

- R610 *parent-string* is *scalar-variable-name*  
or *array-element*  
or *scalar-structure-component*  
or *scalar-constant*
- R611 *substring-range* is [ *scalar-int-expr* ] : [ *scalar-int-expr* ]
- C608 (R610) *parent-string* shall be of type character.
- R612 *data-ref* is *part-ref* [ % *part-ref* ] ...
- R613 *part-ref* is *part-name* [ ( *section-subscript-list* ) ]
- C609 (R612) Each *part-name* except the rightmost shall be of derived type.
- C610 (R612) Each *part-name* except the leftmost shall be the name of a component of the declared type of the preceding *part-name*.
- C611 (R612) If the rightmost *part-name* is of abstract type, *data-ref* shall be polymorphic.
- C612 (R612) The leftmost *part-name* shall be the name of a data object.
- C613 (R613) If a *section-subscript-list* appears, the number of *section-subscripts* shall equal the rank of *part-name*.
- C614 (R612) There shall not be more than one *part-ref* with nonzero rank. A *part-name* to the right of a *part-ref* with nonzero rank shall not have the ALLOCATABLE or POINTER attribute.
- R614 *structure-component* is *data-ref*
- C615 (R614) There shall be more than one *part-ref* and the rightmost *part-ref* shall be of the form *part-name*.
- R615 *type-param-inquiry* is *designator* % *type-param-name*
- C616 (R615) The *type-param-name* shall be the name of a type parameter of the declared type of the object designated by the *designator*.
- R616 *array-element* is *data-ref*
- C617 (R616) Every *part-ref* shall have rank zero and the last *part-ref* shall contain a *subscript-list*.
- R617 *array-section* is *data-ref* [ ( *substring-range* ) ]
- C618 (R617) Exactly one *part-ref* shall have nonzero rank, and either the final *part-ref* shall have a *section-subscript-list* with nonzero rank or another *part-ref* shall have nonzero rank.
- C619 (R617) If a *substring-range* appears, the rightmost *part-name* shall be of type character.
- R618 *subscript* is *scalar-int-expr*
- R619 *section-subscript* is *subscript*  
or *subscript-triplet*  
or *vector-subscript*
- R620 *subscript-triplet* is [ *subscript* ] : [ *subscript* ] [ : *stride* ]
- R621 *stride* is *scalar-int-expr*
- R622 *vector-subscript* is *int-expr*
- C620 (R622) A *vector-subscript* shall be an integer array expression of rank one.
- C621 (R620) The second subscript shall not be omitted from a *subscript-triplet* in the last dimension of an assumed-size array.
- R623 *allocate-stmt* is ALLOCATE ( [ *type-spec* :: ] *allocation-list* ■  
■ [ , *alloc-opt-list* ] )
- R624 *alloc-opt* is STAT = *stat-variable*  
or ERRMSG = *errmsg-variable*  
or SOURCE = *source-expr*
- R625 *stat-variable* is *scalar-int-variable*
- R626 *errmsg-variable* is *scalar-default-char-variable*
- R627 *source-expr* is *expr*

- R628 *allocation* is *allocate-object* [ ( *allocate-shape-spec-list* ) ]
- R629 *allocate-object* is *variable-name*  
or *structure-component*
- R630 *allocate-shape-spec* is [ *lower-bound-expr* : ] *upper-bound-expr*
- R631 *lower-bound-expr* is *scalar-int-expr*
- R632 *upper-bound-expr* is *scalar-int-expr*
- C622 (R629) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.
- C623 (R623) If any *allocate-object* in the statement has a deferred type parameter, either *type-spec* or SOURCE= shall appear.
- C624 (R623) If a *type-spec* appears, it shall specify a type with which each *allocate-object* is type compatible.
- C625 (R623) If any *allocate-object* is unlimited polymorphic, either *type-spec* or SOURCE= shall appear.
- C626 (R623) A *type-param-value* in a *type-spec* shall be an asterisk if and only if each *allocate-object* is a dummy argument for which the corresponding type parameter is assumed.
- C627 (R623) If a *type-spec* appears, the kind type parameter values of each *allocate-object* shall be the same as the corresponding type parameter values of the *type-spec*.
- C628 (R628) An *allocate-shape-spec-list* shall appear if and only if the *allocate-object* is an array.
- C629 (R628) The number of *allocate-shape-specs* in an *allocate-shape-spec-list* shall be the same as the rank of the *allocate-object*.
- C630 (R624) No *alloc-opt* shall appear more than once in a given *alloc-opt-list*.
- C631 (R623) If SOURCE= appears, *type-spec* shall not appear and *allocation-list* shall contain only one *allocate-object*, which shall be type compatible (5.1.1.2) with *source-expr*.
- C632 (R623) The *source-expr* shall be a scalar or have the same rank as *allocate-object*.
- C633 (R623) Corresponding kind type parameters of *allocate-object* and *source-expr* shall have the same values.
- R633 *nullify-stmt* is NULLIFY ( *pointer-object-list* )
- R634 *pointer-object* is *variable-name*  
or *structure-component*  
or *proc-pointer-name*
- C634 (R634) Each *pointer-object* shall have the POINTER attribute.
- R635 *deallocate-stmt* is DEALLOCATE ( *allocate-object-list* [ , *dealloc-opt-list* ] )
- C635 (R635) Each *allocate-object* shall be a nonprocedure pointer or an allocatable variable.
- R636 *dealloc-opt* is STAT = *stat-variable*  
or ERRMSG = *errmsg-variable*
- C636 (R636) No *dealloc-opt* shall appear more than once in a given *dealloc-opt-list*.

**Section 7:**

- R701 *primary* is *constant*  
or *designator*  
or *array-constructor*  
or *structure-constructor*  
or *function-reference*  
or *type-param-inquiry*  
or *type-param-name*  
or ( *expr* )
- C701 (R701) The *type-param-name* shall be the name of a type parameter.
- C702 (R701) The *designator* shall not be a whole assumed-size array.

- R702 *level-1-expr* is [ *defined-unary-op* ] *primary*
- R703 *defined-unary-op* is . *letter* [ *letter* ] ... .
- C703 (R703) A *defined-unary-op* shall not contain more than 63 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.
- R704 *mult-operand* is *level-1-expr* [ *power-op mult-operand* ]
- R705 *add-operand* is [ *add-operand mult-op* ] *mult-operand*
- R706 *level-2-expr* is [ [ *level-2-expr* ] *add-op* ] *add-operand*
- R707 *power-op* is \*\*
- R708 *mult-op* is \*  
or /
- R709 *add-op* is +  
or -
- R710 *level-3-expr* is [ *level-3-expr concat-op* ] *level-2-expr*
- R711 *concat-op* is //
- R712 *level-4-expr* is [ *level-3-expr rel-op* ] *level-3-expr*
- R713 *rel-op* is .EQ.  
or .NE.  
or .LT.  
or .LE.  
or .GT.  
or .GE.  
or ==  
or /=  
or <  
or <=  
or >  
or >=
- R714 *and-operand* is [ *not-op* ] *level-4-expr*
- R715 *or-operand* is [ *or-operand and-op* ] *and-operand*
- R716 *equiv-operand* is [ *equiv-operand or-op* ] *or-operand*
- R717 *level-5-expr* is [ *level-5-expr equiv-op* ] *equiv-operand*
- R718 *not-op* is .NOT.
- R719 *and-op* is .AND.
- R720 *or-op* is .OR.
- R721 *equiv-op* is .EQV.  
or .NEQV.
- R722 *expr* is [ *expr defined-binary-op* ] *level-5-expr*
- R723 *defined-binary-op* is . *letter* [ *letter* ] ... .
- C704 (R723) A *defined-binary-op* shall not contain more than 63 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.
- R724 *logical-expr* is *expr*
- C705 (R724) *logical-expr* shall be of type logical.
- R725 *char-expr* is *expr*
- C706 (R725) *char-expr* shall be of type character.
- R726 *default-char-expr* is *expr*
- C707 (R726) *default-char-expr* shall be of type default character.

- R727 *int-expr* is *expr*
- C708 (R727) *int-expr* shall be of type integer.
- R728 *numeric-expr* is *expr*
- C709 (R728) *numeric-expr* shall be of type integer, real, or complex.
- R729 *specification-expr* is *scalar-int-expr*
- C710 (R729) The *scalar-int-expr* shall be a restricted expression.
- R730 *initialization-expr* is *expr*
- C711 (R730) *initialization-expr* shall be an initialization expression.
- R731 *char-initialization-expr* is *char-expr*
- C712 (R731) *char-initialization-expr* shall be an initialization expression.
- R732 *int-initialization-expr* is *int-expr*
- C713 (R732) *int-initialization-expr* shall be an initialization expression.
- R733 *logical-initialization-expr* is *logical-expr*
- C714 (R733) *logical-initialization-expr* shall be an initialization expression.
- R734 *assignment-stmt* is *variable = expr*
- C715 (R734) The *variable* in an *assignment-stmt* shall not be a whole assumed-size array.
- R735 *pointer-assignment-stmt* is *data-pointer-object* [ (*bounds-spec-list*) ] => *data-target*  
or *data-pointer-object* (*bounds-remapping-list*) => *data-target*  
or *proc-pointer-object* => *proc-target*
- R736 *data-pointer-object* is *variable-name*  
or *variable* % *data-pointer-component-name*
- C716 (R735) If *data-target* is not unlimited polymorphic, *data-pointer-object* shall be type compatible (5.1.1.2) with it, and the corresponding kind type parameters shall be equal.
- C717 (R735) If *data-target* is unlimited polymorphic, *data-pointer-object* shall be unlimited polymorphic, of a sequence derived type, or of a type with the BIND attribute.
- C718 (R735) If *bounds-spec-list* is specified, the number of *bounds-specs* shall equal the rank of *data-pointer-object*.
- C719 (R735) If *bounds-remapping-list* is specified, the number of *bounds-remappings* shall equal the rank of *data-pointer-object*.
- C720 (R735) If *bounds-remapping-list* is specified, *data-target* shall have rank one; otherwise, the ranks of *data-pointer-object* and *data-target* shall be the same.
- C721 (R736) A *variable-name* shall have the POINTER attribute.
- C722 (R736) A *data-pointer-component-name* shall be the name of a component of *variable* that is a data pointer.
- R737 *bounds-spec* is *lower-bound-expr* :
- R738 *bounds-remapping* is *lower-bound-expr* : *upper-bound-expr*
- R739 *data-target* is *variable*  
or *expr*
- C723 (R739) A *variable* shall have either the TARGET or POINTER attribute, and shall not be an array section with a vector subscript.
- C724 (R739) An *expr* shall be a reference to a function whose result is a data pointer.
- R740 *proc-pointer-object* is *proc-pointer-name*  
or *proc-component-ref*
- R741 *proc-component-ref* is *variable* % *procedure-component-name*
- C725 (R741) the *procedure-component-name* shall be the name of a procedure pointer component of the declared type of *variable*.
- R742 *proc-target* is *expr*

- or *procedure-name*  
or *proc-component-ref*
- C726 (R742) An *expr* shall be a reference to a function whose result is a procedure pointer.
- C727 (R742) A *procedure-name* shall be the name of an external, module, or dummy procedure, a specific intrinsic function listed in 13.6 and not marked with a bullet (●), or a procedure pointer.
- C728 (R742) The *proc-target* shall not be a nonintrinsic elemental procedure.
- R743 *where-stmt* is WHERE ( *mask-expr* ) *where-assignment-stmt*
- R744 *where-construct* is *where-construct-stmt*  
[ *where-body-construct* ] ...  
[ *masked-elsewhere-stmt*  
[ *where-body-construct* ] ... ] ...  
[ *elsewhere-stmt*  
[ *where-body-construct* ] ... ]  
*end-where-stmt*
- R745 *where-construct-stmt* is [*where-construct-name*:] WHERE ( *mask-expr* )
- R746 *where-body-construct* is *where-assignment-stmt*  
or *where-stmt*  
or *where-construct*
- R747 *where-assignment-stmt* is *assignment-stmt*
- R748 *mask-expr* is *logical-expr*
- R749 *masked-elsewhere-stmt* is ELSEWHERE ( *mask-expr* ) [*where-construct-name*]
- R750 *elsewhere-stmt* is ELSEWHERE [*where-construct-name*]
- R751 *end-where-stmt* is END WHERE [*where-construct-name*]
- C729 (R747) A *where-assignment-stmt* that is a defined assignment shall be elemental.
- C730 (R744) If the *where-construct-stmt* is identified by a *where-construct-name*, the corresponding *end-where-stmt* shall specify the same *where-construct-name*. If the *where-construct-stmt* is not identified by a *where-construct-name*, the corresponding *end-where-stmt* shall not specify a *where-construct-name*. If an *elsewhere-stmt* or a *masked-elsewhere-stmt* is identified by a *where-construct-name*, the corresponding *where-construct-stmt* shall specify the same *where-construct-name*.
- C731 (R746) A statement that is part of a *where-body-construct* shall not be a branch target statement.
- R752 *forall-construct* is *forall-construct-stmt*  
[*forall-body-construct* ] ...  
*end-forall-stmt*
- R753 *forall-construct-stmt* is [*forall-construct-name* :] FORALL *forall-header*
- R754 *forall-header* is ( *forall-triplet-spec-list* [ , *scalar-mask-expr* ] )
- R755 *forall-triplet-spec* is *index-name* = *subscript* : *subscript* [ : *stride* ]
- R756 *forall-body-construct* is *forall-assignment-stmt*  
or *where-stmt*  
or *where-construct*  
or *forall-construct*  
or *forall-stmt*
- R757 *forall-assignment-stmt* is *assignment-stmt*  
or *pointer-assignment-stmt*
- R758 *end-forall-stmt* is END FORALL [*forall-construct-name* ]
- C732 (R758) If the *forall-construct-stmt* has a *forall-construct-name*, the *end-forall-stmt* shall have the same *forall-construct-name*. If the *end-forall-stmt* has a *forall-construct-name*, the *forall-*

*construct-stmt* shall have the same *forall-construct-name*.

- C733 (R754) The *scalar-mask-expr* shall be scalar and of type logical.
- C734 (R754) Any procedure referenced in the *scalar-mask-expr*, including one referenced by a defined operation, shall be a pure procedure (12.6).
- C735 (R755) The *index-name* shall be a named scalar variable of type integer.
- C736 (R755) A *subscript* or *stride* in a *forall-triplet-spec* shall not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.
- C737 (R756) A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-construct*.
- C738 (R756) Any procedure referenced in a *forall-body-construct*, including one referenced by a defined operation, assignment, or finalization, shall be a pure procedure.
- C739 (R756) A *forall-body-construct* shall not be a branch target.
- R759 *forall-stmt* is FORALL *forall-header forall-assignment-stmt*

### Section 8:

- R801 *block* is [ *execution-part-construct* ] ...
- R802 *if-construct* is *if-then-stmt*  
                                   *block*  
                                   [ *else-if-stmt*  
                                   *block* ] ...  
                                   [ *else-stmt*  
                                   *block* ]  
                                   *end-if-stmt*
- R803 *if-then-stmt* is [ *if-construct-name* : ] IF ( *scalar-logical-expr* ) THEN
- R804 *else-if-stmt* is ELSE IF ( *scalar-logical-expr* ) THEN [ *if-construct-name* ]
- R805 *else-stmt* is ELSE [ *if-construct-name* ]
- R806 *end-if-stmt* is END IF [ *if-construct-name* ]
- C801 (R802) If the *if-then-stmt* of an *if-construct* specifies an *if-construct-name*, the corresponding *end-if-stmt* shall specify the same *if-construct-name*. If the *if-then-stmt* of an *if-construct* does not specify an *if-construct-name*, the corresponding *end-if-stmt* shall not specify an *if-construct-name*. If an *else-if-stmt* or *else-stmt* specifies an *if-construct-name*, the corresponding *if-then-stmt* shall specify the same *if-construct-name*.
- R807 *if-stmt* is IF ( *scalar-logical-expr* ) *action-stmt*
- C802 (R807) The *action-stmt* in the *if-stmt* shall not be an *if-stmt*, *end-program-stmt*, *end-function-stmt*, or *end-subroutine-stmt*.
- R808 *case-construct* is *select-case-stmt*  
                                   [ *case-stmt*  
                                   *block* ] ...  
                                   *end-select-stmt*
- R809 *select-case-stmt* is [ *case-construct-name* : ] SELECT CASE ( *case-expr* )
- R810 *case-stmt* is CASE *case-selector* [ *case-construct-name* ]
- R811 *end-select-stmt* is END SELECT [ *case-construct-name* ]
- C803 (R808) If the *select-case-stmt* of a *case-construct* specifies a *case-construct-name*, the corresponding *end-select-stmt* shall specify the same *case-construct-name*. If the *select-case-stmt* of a *case-construct* does not specify a *case-construct-name*, the corresponding *end-select-stmt* shall not specify a *case-construct-name*. If a *case-stmt* specifies a *case-construct-name*, the corresponding *select-case-stmt* shall specify the same *case-construct-name*.
- R812 *case-expr* is *scalar-int-expr*  
                                   or *scalar-char-expr*

- R813** *case-selector*                    **or** *scalar-logical-expr*  
     **is** ( *case-value-range-list* )  
     **or** DEFAULT
- C804 (R808) No more than one of the selectors of one of the CASE statements shall be DEFAULT.
- R814** *case-value-range*                **is** *case-value*  
     **or** *case-value :*  
     **or** *: case-value*  
     **or** *case-value : case-value*
- R815** *case-value*                      **is** *scalar-int-initialization-expr*  
     **or** *scalar-char-initialization-expr*  
     **or** *scalar-logical-initialization-expr*
- C805 (R808) For a given *case-construct*, each *case-value* shall be of the same type as *case-expr*. For character type, the kind type parameters shall be the same; character length differences are allowed.
- C806 (R808) A *case-value-range* using a colon shall not be used if *case-expr* is of type logical.
- C807 (R808) For a given *case-construct*, the *case-value-ranges* shall not overlap; that is, there shall be no possible value of the *case-expr* that matches more than one *case-value-range*.
- R816** *associate-construct*            **is** *associate-stmt*  
     *block*  
     *end-associate-stmt*
- R817** *associate-stmt*                    **is** [ *associate-construct-name :* ] ASSOCIATE ■  
     ■ ( *association-list* )
- R818** *association*                      **is** *associate-name => selector*
- R819** *selector*                         **is** *expr*  
     **or** *variable*
- C808 (R818) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.5.7).
- C809 (R818) An *associate-name* shall not be the same as another *associate-name* in the same *associate-stmt*.
- R820** *end-associate-stmt*            **is** END ASSOCIATE [ *associate-construct-name* ]
- C810 (R820) If the *associate-stmt* of an *associate-construct* specifies an *associate-construct-name*, the corresponding *end-associate-stmt* shall specify the same *associate-construct-name*. If the *associate-stmt* of an *associate-construct* does not specify an *associate-construct-name*, the corresponding *end-associate-stmt* shall not specify an *associate-construct-name*.
- R821** *select-type-construct*        **is** *select-type-stmt*  
     [ *type-guard-stmt*  
     *block* ] ...  
     *end-select-type-stmt*
- R822** *select-type-stmt*                **is** [ *select-construct-name :* ] SELECT TYPE ■  
     ■ ( [ *associate-name =>* ] *selector* )
- C811 (R822) If *selector* is not a named *variable*, *associate-name =>* shall appear.
- C812 (R822) If *selector* is not a *variable* or is a *variable* that has a vector subscript, *associate-name* shall not appear in a variable definition context (16.5.7).
- C813 (R822) The *selector* in a *select-type-stmt* shall be polymorphic.
- R823** *type-guard-stmt*                **is** TYPE IS ( *type-spec* ) [ *select-construct-name* ]  
     **or** CLASS IS ( *type-spec* ) [ *select-construct-name* ]

- or CLASS DEFAULT [ *select-construct-name* ]
- C814 (R823) The *type-spec* shall specify that each length type parameter is assumed.
- C815 (R823) The *type-spec* shall not specify a sequence derived type or a type with the BIND attribute.
- C816 (R823) If *selector* is not unlimited polymorphic, the *type-spec* shall specify an extension of the declared type of *selector*.
- C817 (R823) For a given *select-type-construct*, the same type and kind type parameter values shall not be specified in more than one TYPE IS *type-guard-stmt* and shall not be specified in more than one CLASS IS *type-guard-stmt*.
- C818 (R823) For a given *select-type-construct*, there shall be at most one CLASS DEFAULT *type-guard-stmt*.
- R824 *end-select-type-stmt* is END SELECT [ *select-construct-name* ]
- C819 (R821) If the *select-type-stmt* of a *select-type-construct* specifies a *select-construct-name*, the corresponding *end-select-type-stmt* shall specify the same *select-construct-name*. If the *select-type-stmt* of a *select-type-construct* does not specify a *select-construct-name*, the corresponding *end-select-type-stmt* shall not specify a *select-construct-name*. If a *type-guard-stmt* specifies a *select-construct-name*, the corresponding *select-type-stmt* shall specify the same *select-construct-name*.
- R825 *do-construct* is *block-do-construct*  
or *nonblock-do-construct*
- R826 *block-do-construct* is *do-stmt*  
*do-block*  
*end-do*
- R827 *do-stmt* is *label-do-stmt*  
or *nonlabel-do-stmt*
- R828 *label-do-stmt* is [ *do-construct-name* : ] DO *label* [ *loop-control* ]
- R829 *nonlabel-do-stmt* is [ *do-construct-name* : ] DO [ *loop-control* ]
- R830 *loop-control* is [ , ] *do-variable* = *scalar-int-expr*, *scalar-int-expr* ■  
■ [ , *scalar-int-expr* ]  
or [ , ] WHILE ( *scalar-logical-expr* )
- R831 *do-variable* is *scalar-int-variable*
- C820 (R831) The *do-variable* shall be a named scalar variable of type integer.
- R832 *do-block* is *block*
- R833 *end-do* is *end-do-stmt*  
or *continue-stmt*
- R834 *end-do-stmt* is END DO [ *do-construct-name* ]
- C821 (R826) If the *do-stmt* of a *block-do-construct* specifies a *do-construct-name*, the corresponding *end-do* shall be an *end-do-stmt* specifying the same *do-construct-name*. If the *do-stmt* of a *block-do-construct* does not specify a *do-construct-name*, the corresponding *end-do* shall not specify a *do-construct-name*.
- C822 (R826) If the *do-stmt* is a *nonlabel-do-stmt*, the corresponding *end-do* shall be an *end-do-stmt*.
- C823 (R826) If the *do-stmt* is a *label-do-stmt*, the corresponding *end-do* shall be identified with the same *label*.
- R835 *nonblock-do-construct* is *action-term-do-construct*  
or *outer-shared-do-construct*
- R836 *action-term-do-construct* is *label-do-stmt*  
*do-body*  
*do-term-action-stmt*
- R837 *do-body* is [ *execution-part-construct* ] ...

- R838 *do-term-action-stmt* is *action-stmt*
- C824 (R838) A *do-term-action-stmt* shall not be a *continue-stmt*, a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.
- C825 (R835) The *do-term-action-stmt* shall be identified with a label and the corresponding *label-do-stmt* shall refer to the same label.
- R839 *outer-shared-do-construct* is *label-do-stmt*  
*do-body*  
*shared-term-do-construct*
- R840 *shared-term-do-construct* is *outer-shared-do-construct*  
or *inner-shared-do-construct*
- R841 *inner-shared-do-construct* is *label-do-stmt*  
*do-body*  
*do-term-shared-stmt*
- R842 *do-term-shared-stmt* is *action-stmt*
- C826 (R842) A *do-term-shared-stmt* shall not be a *goto-stmt*, a *return-stmt*, a *stop-stmt*, an *exit-stmt*, a *cycle-stmt*, an *end-function-stmt*, an *end-subroutine-stmt*, an *end-program-stmt*, or an *arithmetic-if-stmt*.
- C827 (R840) The *do-term-shared-stmt* shall be identified with a label and all of the *label-do-stmts* of the *inner-shared-do-construct* and *outer-shared-do-construct* shall refer to the same label.
- R843 *cycle-stmt* is CYCLE [ *do-construct-name* ]
- C828 (R843) If a *cycle-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.
- R844 *exit-stmt* is EXIT [ *do-construct-name* ]
- C829 (R844) If an *exit-stmt* refers to a *do-construct-name*, it shall be within the range of that *do-construct*; otherwise, it shall be within the range of at least one *do-construct*.
- R845 *goto-stmt* is GO TO *label*
- C830 (R845) The *label* shall be the statement label of a branch target statement that appears in the same scoping unit as the *goto-stmt*.
- R846 *computed-goto-stmt* is GO TO ( *label-list* ) [ , ] *scalar-int-expr*
- C831 (R846) Each *label* in *label-list* shall be the statement label of a branch target statement that appears in the same scoping unit as the *computed-goto-stmt*.
- R847 *arithmetic-if-stmt* is IF ( *scalar-numeric-expr* ) *label* , *label* , *label*
- C832 (R847) Each *label* shall be the label of a branch target statement that appears in the same scoping unit as the *arithmetic-if-stmt*.
- C833 (R847) The *scalar-numeric-expr* shall not be of type complex.
- R848 *continue-stmt* is CONTINUE
- R849 *stop-stmt* is STOP [ *stop-code* ]
- R850 *stop-code* is *scalar-char-constant*  
or *digit* [ *digit* [ *digit* [ *digit* ] ] ] ]
- C834 (R850) *scalar-char-constant* shall be of type default character.

**Section 9:**

- R901 *io-unit* is *file-unit-number*  
or \*  
or *internal-file-variable*
- R902 *file-unit-number* is *scalar-int-expr*

- R903 *internal-file-variable* is *char-variable*
- C901 (R903) The *char-variable* shall not be an array section with a vector subscript.
- C902 (R903) The *char-variable* shall be of type default character, ASCII character, or ISO 10646 character.
- R904 *open-stmt* is OPEN ( *connect-spec-list* )
- R905 *connect-spec* is [ UNIT = ] *file-unit-number*
- or ACCESS = *scalar-default-char-expr*
- or ACTION = *scalar-default-char-expr*
- or ASYNCHRONOUS = *scalar-default-char-expr*
- or BLANK = *scalar-default-char-expr*
- or DECIMAL = *scalar-default-char-expr*
- or DELIM = *scalar-default-char-expr*
- or ENCODING = *scalar-default-char-expr*
- or ERR = *label*
- or FILE = *file-name-expr*
- or FORM = *scalar-default-char-expr*
- or IOMSG = *iomsg-variable*
- or IOSTAT = *scalar-int-variable*
- or PAD = *scalar-default-char-expr*
- or POSITION = *scalar-default-char-expr*
- or RECL = *scalar-int-expr*
- or ROUND = *scalar-default-char-expr*
- or SIGN = *scalar-default-char-expr*
- or STATUS = *scalar-default-char-expr*
- R906 *file-name-expr* is *scalar-default-char-expr*
- R907 *iomsg-variable* is *scalar-default-char-variable*
- C903 (R905) No specifier shall appear more than once in a given *connect-spec-list*.
- C904 (R905) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *connect-spec-list*.
- C905 (R905) The *label* used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the OPEN statement.
- R908 *close-stmt* is CLOSE ( *close-spec-list* )
- R909 *close-spec* is [ UNIT = ] *file-unit-number*
- or IOSTAT = *scalar-int-variable*
- or IOMSG = *iomsg-variable*
- or ERR = *label*
- or STATUS = *scalar-default-char-expr*
- C906 (R909) No specifier shall appear more than once in a given *close-spec-list*.
- C907 (R909) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *close-spec-list*.
- C908 (R909) The *label* used in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the CLOSE statement.
- R910 *read-stmt* is READ ( *io-control-spec-list* ) [ *input-item-list* ]
- or READ *format* [ , *input-item-list* ]
- R911 *write-stmt* is WRITE ( *io-control-spec-list* ) [ *output-item-list* ]
- R912 *print-stmt* is PRINT *format* [ , *output-item-list* ]
- R913 *io-control-spec* is [ UNIT = ] *io-unit*

- or [ FMT = ] *format*
- or [ NML = ] *namelist-group-name*
- or ADVANCE = *scalar-default-char-expr*
- or ASYNCHRONOUS = *scalar-char-initialization-expr*
- or BLANK = *scalar-default-char-expr*
- or DECIMAL = *scalar-default-char-expr*
- or DELIM = *scalar-default-char-expr*
- or END = *label*
- or EOR = *label*
- or ERR = *label*
- or ID = *scalar-int-variable*
- or IOMSG = *iomsg-variable*
- or IOSTAT = *scalar-int-variable*
- or PAD = *scalar-default-char-expr*
- or POS = *scalar-int-expr*
- or REC = *scalar-int-expr*
- or ROUND = *scalar-default-char-expr*
- or SIGN = *scalar-default-char-expr*
- or SIZE = *scalar-int-variable*

- C909 (R913) No specifier shall appear more than once in a given *io-control-spec-list*.
- C910 (R913) An *io-unit* shall be specified; if the optional characters UNIT= are omitted, the *io-unit* shall be the first item in the *io-control-spec-list*.
- C911 (R913) A DELIM= or SIGN= specifier shall not appear in a *read-stmt*.
- C912 (R913) A BLANK=, PAD=, END=, EOR=, or SIZE= specifier shall not appear in a *write-stmt*.
- C913 (R913) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the data transfer statement.
- C914 (R913) A *namelist-group-name* shall be the name of a namelist group.
- C915 (R913) A *namelist-group-name* shall not appear if an *input-item-list* or an *output-item-list* appears in the data transfer statement.
- C916 (R913) An *io-control-spec-list* shall not contain both a *format* and a *namelist-group-name*.
- C917 (R913) If *format* appears without a preceding FMT=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.
- C918 (R913) If *namelist-group-name* appears without a preceding NML=, it shall be the second item in the *io-control-spec-list* and the first item shall be *io-unit*.
- C919 (R913) If *io-unit* is not a *file-unit-number*, the *io-control-spec-list* shall not contain a REC= specifier or a POS= specifier.
- C920 (R913) If the REC= specifier appears, an END= specifier shall not appear, a *namelist-group-name* shall not appear, and the *format*, if any, shall not be an asterisk.
- C921 (R913) An ADVANCE= specifier may appear only in a formatted sequential or stream input/output statement with explicit format specification (10.1) whose control information list does not contain an *internal-file-variable* as the *io-unit*.
- C922 (R913) If an EOR= specifier appears, an ADVANCE= specifier also shall appear.
- C923 (R913) If a SIZE= specifier appears, an ADVANCE= specifier also shall appear.
- C924 (R913) The *scalar-char-initialization-expr* in an ASYNCHRONOUS= specifier shall be of type default character and shall have the value YES or NO.
- C925 (R913) An ASYNCHRONOUS= specifier with a value YES shall not appear unless *io-unit* is a *file-unit-number*.
- C926 (R913) If an ID= specifier appears, an ASYNCHRONOUS= specifier with the value YES shall

also appear.

- C927 (R913) If a POS= specifier appears, the *io-control-spec-list* shall not contain a REC= specifier.
- C928 (R913) If a DECIMAL=, BLANK=, PAD=, SIGN=, or ROUND= specifier appears, a *format* or *namelist-group-name* shall also appear.
- C929 (R913) If a DELIM= specifier appears, either *format* shall be an asterisk or *namelist-group-name* shall appear.
- R914 *format*                                    **is** *default-char-expr*  
   **or** *label*  
   **or** \*
- C930 (R914) The *label* shall be the label of a FORMAT statement that appears in the same scoping unit as the statement containing the FMT= specifier.
- R915 *input-item*                               **is** *variable*  
   **or** *io-implied-do*
- R916 *output-item*                             **is** *expr*  
   **or** *io-implied-do*
- R917 *io-implied-do*                           **is** ( *io-implied-do-object-list* , *io-implied-do-control* )
- R918 *io-implied-do-object*                   **is** *input-item*  
   **or** *output-item*
- R919 *io-implied-do-control*                 **is** *do-variable = scalar-int-expr* , ■  
   ■ *scalar-int-expr* [ , *scalar-int-expr* ]
- C931 (R915) A variable that is an *input-item* shall not be a whole assumed-size array.
- C932 (R915) A variable that is an *input-item* shall not be a procedure pointer.
- C933 (R919) The *do-variable* shall be a named scalar variable of type integer.
- C934 (R918) In an *input-item-list*, an *io-implied-do-object* shall be an *input-item*. In an *output-item-list*, an *io-implied-do-object* shall be an *output-item*.
- C935 (R916) An expression that is an *output-item* shall not have a value that is a procedure pointer.
- R920 *dtv-type-spec*                           **is** TYPE( *derived-type-spec* )  
   **or** CLASS( *derived-type-spec* )
- C936 (R920) If *derived-type-spec* specifies an extensible type, the CLASS keyword shall be used; otherwise, the TYPE keyword shall be used.
- C937 (R920) All length type parameters of *derived-type-spec* shall be assumed.
- R921 *wait-stmt*                                **is** WAIT ( *wait-spec-list* )
- R922 *wait-spec*                                **is** [ UNIT = ] *file-unit-number*  
   **or** END = *label*  
   **or** EOR = *label*  
   **or** ERR = *label*  
   **or** ID = *scalar-int-variable*  
   **or** IOMSG = *iomsg-variable*  
   **or** IOSTAT = *scalar-int-variable*
- C938 (R922) No specifier shall appear more than once in a given *wait-spec-list*.
- C939 (R922) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *wait-spec-list*.
- C940 (R922) The *label* in the ERR=, EOR=, or END= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the WAIT statement.
- R923 *backspace-stmt*                         **is** BACKSPACE *file-unit-number*  
   **or** BACKSPACE ( *position-spec-list* )
- R924 *endfile-stmt*                            **is** ENDFILE *file-unit-number*

- R925 *rewind-stmt*                   or ENDFILE ( *position-spec-list* )  
                                       is REWIND *file-unit-number*  
                                       or REWIND ( *position-spec-list* )
- R926 *position-spec*                is [ UNIT = ] *file-unit-number*  
                                       or IOMSG = *iomsg-variable*  
                                       or IOSTAT = *scalar-int-variable*  
                                       or ERR = *label*
- C941 (R926) No specifier shall appear more than once in a given *position-spec-list*.
- C942 (R926) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *position-spec-list*.
- C943 (R926) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the file positioning statement.
- R927 *flush-stmt*                   is FLUSH *file-unit-number*  
                                       or FLUSH ( *flush-spec-list* )
- R928 *flush-spec*                   is [UNIT =] *file-unit-number*  
                                       or IOSTAT = *scalar-int-variable*  
                                       or IOMSG = *iomsg-variable*  
                                       or ERR = *label*
- C944 (R928) No specifier shall appear more than once in a given *flush-spec-list*.
- C945 (R928) A *file-unit-number* shall be specified; if the optional characters UNIT= are omitted from the unit specifier, the *file-unit-number* shall be the first item in the *flush-spec-list*.
- C946 (R928) The *label* in the ERR= specifier shall be the statement label of a branch target statement that appears in the same scoping unit as the flush statement.
- R929 *inquire-stmt*                is INQUIRE ( *inquire-spec-list* )  
                                       or INQUIRE ( IOLENGTH = *scalar-int-variable* ) ■  
                                       ■ *output-item-list*
- R930 *inquire-spec*                is [ UNIT = ] *file-unit-number*  
                                       or FILE = *file-name-expr*  
                                       or ACCESS = *scalar-default-char-variable*  
                                       or ACTION = *scalar-default-char-variable*  
                                       or ASYNCHRONOUS = *scalar-default-char-variable*  
                                       or BLANK = *scalar-default-char-variable*  
                                       or DECIMAL = *scalar-default-char-variable*  
                                       or DELIM = *scalar-default-char-variable*  
                                       or DIRECT = *scalar-default-char-variable*  
                                       or ENCODING = *scalar-default-char-variable*  
                                       or ERR = *label*  
                                       or EXIST = *scalar-default-logical-variable*  
                                       or FORM = *scalar-default-char-variable*  
                                       or FORMATTED = *scalar-default-char-variable*  
                                       or ID = *scalar-int-variable*  
                                       or IOMSG = *iomsg-variable*  
                                       or IOSTAT = *scalar-int-variable*  
                                       or NAME = *scalar-default-char-variable*  
                                       or NAMED = *scalar-default-logical-variable*  
                                       or NEXTREC = *scalar-int-variable*  
                                       or NUMBER = *scalar-int-variable*

- or OPENED = *scalar-default-logical-variable*
- or PAD = *scalar-default-char-variable*
- or PENDING = *scalar-default-logical-variable*
- or POS = *scalar-int-variable*
- or POSITION = *scalar-default-char-variable*
- or READ = *scalar-default-char-variable*
- or READWRITE = *scalar-default-char-variable*
- or RECL = *scalar-int-variable*
- or ROUND = *scalar-default-char-variable*
- or SEQUENTIAL = *scalar-default-char-variable*
- or SIGN = *scalar-default-char-variable*
- or SIZE = *scalar-int-variable*
- or STREAM = *scalar-default-char-variable*
- or UNFORMATTED = *scalar-default-char-variable*
- or WRITE = *scalar-default-char-variable*

- C947 (R930) No specifier shall appear more than once in a given *inquire-spec-list*.
- C948 (R930) An *inquire-spec-list* shall contain one FILE= specifier or one UNIT= specifier, but not both.
- C949 (R930) In the inquire by unit form of the INQUIRE statement, if the optional characters UNIT= are omitted, the *file-unit-number* shall be the first item in the *inquire-spec-list*.
- C950 (R930) If an ID= specifier appears, a PENDING= specifier shall also appear.

### Section 10:

- R1001 *format-stmt* is FORMAT *format-specification*
- R1002 *format-specification* is ( [ *format-item-list* ] )
- C1001 (R1001) The *format-stmt* shall be labeled.
- C1002 (R1002) The comma used to separate *format-items* in a *format-item-list* may be omitted
- R1003 *format-item* is [ *r* ] *data-edit-desc*  
or *control-edit-desc*  
or *char-string-edit-desc*  
or [ *r* ] ( *format-item-list* )
- R1004 *r* is *int-literal-constant*
- C1003 (R1004) *r* shall be positive.
- C1004 (R1004) *r* shall not have a kind parameter specified for it.
- R1005 *data-edit-desc* is I *w* [ . *m* ]  
or B *w* [ . *m* ]  
or O *w* [ . *m* ]  
or Z *w* [ . *m* ]  
or F *w* . *d*  
or E *w* . *d* [ E *e* ]  
or EN *w* . *d* [ E *e* ]  
or ES *w* . *d* [ E *e* ]  
or G *w* . *d* [ E *e* ]  
or L *w*  
or A [ *w* ]  
or D *w* . *d*  
or DT [ *char-literal-constant* ] [ ( *v-list* ) ]

- R1006 *w* is *int-literal-constant*
- R1007 *m* is *int-literal-constant*
- R1008 *d* is *int-literal-constant*
- R1009 *e* is *int-literal-constant*
- R1010 *v* is *signed-int-literal-constant*
- C1005 (R1009) *e* shall be positive.
- C1006 (R1006) *w* shall be zero or positive for the I, B, O, Z, and F edit descriptors. *w* shall be positive for all other edit descriptors.
- C1007 (R1005) *w*, *m*, *d*, *e*, and *v* shall not have kind parameters specified for them.
- C1008 (R1005) The *char-literal-constant* in the DT edit descriptor shall not have a kind parameter specified for it.
- R1011 *control-edit-desc* is *position-edit-desc*  
or [ *r* ] /  
or :  
or *sign-edit-desc*  
or *k P*  
or *blank-interp-edit-desc*  
or *round-edit-desc*  
or *decimal-edit-desc*
- R1012 *k* is *signed-int-literal-constant*
- C1009 (R1012) *k* shall not have a kind parameter specified for it.
- R1013 *position-edit-desc* is T *n*  
or TL *n*  
or TR *n*  
or *n X*
- R1014 *n* is *int-literal-constant*
- C1010 (R1014) *n* shall be positive.
- C1011 (R1014) *n* shall not have a kind parameter specified for it.
- R1015 *sign-edit-desc* is SS  
or SP  
or S
- R1016 *blank-interp-edit-desc* is BN  
or BZ
- R1017 *round-edit-desc* is RU  
or RD  
or RZ  
or RN  
or RC  
or RP
- R1018 *decimal-edit-desc* is DC  
or DP
- R1019 *char-string-edit-desc* is *char-literal-constant*
- C1012 (R1019) The *char-literal-constant* shall not have a kind parameter specified for it.

**Section 11:**

- R1101 *main-program* is [ *program-stmt* ]  
[ *specification-part* ]

- [ *execution-part* ]  
 [ *internal-subprogram-part* ]  
*end-program-stmt*
- R1102 *program-stmt*            **is** PROGRAM *program-name*  
 R1103 *end-program-stmt*       **is** END [ PROGRAM [ *program-name* ] ]
- C1101 (R1101) In a *main-program*, the *execution-part* shall not contain a RETURN statement or an ENTRY statement.
- C1102 (R1101) The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, shall be identical to the *program-name* specified in the *program-stmt*.
- C1103 (R1101) An automatic object shall not appear in the *specification-part* (R204) of a main program.
- R1104 *module*                   **is** *module-stmt*  
                                   [ *specification-part* ]  
                                   [ *module-subprogram-part* ]  
                                   *end-module-stmt*
- R1105 *module-stmt*             **is** MODULE *module-name*  
 R1106 *end-module-stmt*       **is** END [ MODULE [ *module-name* ] ]
- R1107 *module-subprogram-part* **is** *contains-stmt*  
                                   *module-subprogram*  
                                   [ *module-subprogram* ] ...
- R1108 *module-subprogram*       **is** *function-subprogram*  
                                   **or** *subroutine-subprogram*
- C1104 (R1104) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-name* specified in the *module-stmt*.
- C1105 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a *format-stmt*.
- C1106 (R1104) An automatic object shall not appear in the *specification-part* of a module.
- C1107 (R1104) If an object of a type for which *component-initialization* is specified (R444) appears in the *specification-part* of a module and does not have the ALLOCATABLE or POINTER attribute, the object shall have the SAVE attribute.
- R1109 *use-stmt*                 **is** USE [ [ , *module-nature* ] :: ] *module-name* [ , *rename-list* ]  
                                   **or** USE [ [ , *module-nature* ] :: ] *module-name* , ■  
                                   ■ ONLY : [ *only-list* ]
- R1110 *module-nature*           **is** INTRINSIC  
                                   **or** NON\_INTRINSIC
- R1111 *rename*                  **is** *local-name* => *use-name*  
                                   **or** OPERATOR (*local-defined-operator*) => ■  
                                   ■ OPERATOR (*use-defined-operator*)
- R1112 *only*                    **is** *generic-spec*  
                                   **or** *only-use-name*  
                                   **or** *rename*
- R1113 *only-use-name*          **is** *use-name*
- C1108 (R1109) If *module-nature* is INTRINSIC, *module-name* shall be the name of an intrinsic module.
- C1109 (R1109) If *module-nature* is NON\_INTRINSIC, *module-name* shall be the name of a nonintrinsic module.
- C1110 (R1109) A scoping unit shall not access an intrinsic module and a nonintrinsic module of the

same name.

- C1111 (R1111) OPERATOR(*use-defined-operator*) shall not identify a *generic-binding*.
- C1112 (R1112) The *generic-spec* shall not identify a *generic-binding*.
- C1113 (R1112) Each *generic-spec* shall be a public entity in the module.
- C1114 (R1113) Each *use-name* shall be the name of a public entity in the module.
- R1114 *local-defined-operator*           is *defined-unary-op*  
  or *defined-binary-op*
- R1115 *use-defined-operator*           is *defined-unary-op*  
  or *defined-binary-op*
- C1115 (R1115) Each *use-defined-operator* shall be a public entity in the module.
- R1116 *block-data*                    is *block-data-stmt*  
  [ *specification-part* ]  
  *end-block-data-stmt*
- R1117 *block-data-stmt*               is BLOCK DATA [ *block-data-name* ]
- R1118 *end-block-data-stmt*           is END [ BLOCK DATA [ *block-data-name* ] ]
- C1116 (R1116) The *block-data-name* shall be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-data-stmt*.
- C1117 (R1116) A *block-data specification-part* shall contain only derived-type definitions and ASYNCHRONOUS, BIND, COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, INTRINSIC, PARAMETER, POINTER, SAVE, TARGET, USE, VOLATILE, and type declaration statements.
- C1118 (R1116) A type declaration statement in a *block-data specification-part* shall not contain ALLOCATABLE, EXTERNAL, or BIND attribute specifiers.

## Section 12:

- R1201 *interface-block*               is *interface-stmt*  
  [ *interface-specification* ] ...  
  *end-interface-stmt*
- R1202 *interface-specification*       is *interface-body*  
  or *procedure-stmt*
- R1203 *interface-stmt*                is INTERFACE [ *generic-spec* ]  
  or ABSTRACT INTERFACE
- R1204 *end-interface-stmt*           is END INTERFACE [ *generic-spec* ]
- R1205 *interface-body*                is *function-stmt*  
  [ *specification-part* ]  
  *end-function-stmt*  
  or *subroutine-stmt*  
  [ *specification-part* ]  
  *end-subroutine-stmt*
- R1206 *procedure-stmt*                is [ MODULE ] PROCEDURE *procedure-name-list*
- R1207 *generic-spec*                   is *generic-name*  
  or OPERATOR ( *defined-operator* )  
  or ASSIGNMENT ( = )  
  or *dtio-generic-spec*
- R1208 *dtio-generic-spec*            is READ (FORMATTED)  
  or READ (UNFORMATTED)  
  or WRITE (FORMATTED)

- or WRITE (UNFORMATTED)
- R1209 *import-stmt* is IMPORT [ [ :: ] *import-name-list*
- C1201 (R1201) An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure defined by that subprogram.
- C1202 (R1201) The *generic-spec* shall be included in the *end-interface-stmt* only if it is provided in the *interface-stmt*. If the *end-interface-stmt* includes *generic-name*, the *interface-stmt* shall specify the same *generic-name*. If the *end-interface-stmt* includes ASSIGNMENT(=), the *interface-stmt* shall specify ASSIGNMENT(=). If the *end-interface-stmt* includes *dtio-generic-spec*, the *interface-stmt* shall specify the same *dtio-generic-spec*. If the *end-interface-stmt* includes OPERATOR(*defined-operator*), the *interface-stmt* shall specify the same *defined-operator*. If one *defined-operator* is .LT., .LE., .GT., .GE., .EQ., or .NE., the other is permitted to be the corresponding operator <, <=, >, >=, ==, or /=.
- C1203 (R1203) If the *interface-stmt* is ABSTRACT INTERFACE, then the *function-name* in the *function-stmt* or the *subroutine-name* in the *subroutine-stmt* shall not be the same as a keyword that specifies an intrinsic type.
- C1204 (R1202) A *procedure-stmt* is allowed only in an interface block that has a *generic-spec*.
- C1205 (R1205) An *interface-body* of a pure procedure shall specify the intents of all dummy arguments except pointer, alternate return, and procedure arguments.
- C1206 (R1205) An *interface-body* shall not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-function-stmt*.
- C1207 (R1206) A *procedure-name* shall have an explicit interface and shall refer to an accessible procedure pointer, external procedure, dummy procedure, or module procedure.
- C1208 (R1206) If MODULE appears in a *procedure-stmt*, each *procedure-name* in that statement shall be accessible in the current scope as a module procedure.
- C1209 (R1206) A *procedure-name* shall not specify a procedure that is specified previously in any *procedure-stmt* in any accessible interface with the same generic identifier.
- C1210 (R1209) The IMPORT statement is allowed only in an *interface-body*.
- C1211 (R1209) Each *import-name* shall be the name of an entity in the host scoping unit.
- R1210 *external-stmt* is EXTERNAL [ [ :: ] *external-name-list*
- R1211 *procedure-declaration-stmt* is PROCEDURE ( [ *proc-interface* ] ) ■  
 ■ [ [ , *proc-attr-spec* ] ... :: ] *proc-decl-list*
- R1212 *proc-interface* is *interface-name*  
 or *declaration-type-spec*
- R1213 *proc-attr-spec* is *access-spec*  
 or *proc-language-binding-spec*  
 or INTENT ( *intent-spec* )  
 or OPTIONAL  
 or POINTER  
 or SAVE
- R1214 *proc-decl* is *procedure-entity-name*[ => *null-init* ]
- R1215 *interface-name* is *name*
- C1212 (R1215) The *name* shall be the name of an abstract interface or of a procedure that has an explicit interface. If *name* is declared by a *procedure-declaration-stmt* it shall be previously declared. If *name* denotes an intrinsic procedure it shall be one that is listed in 13.6 and not marked with a bullet (●).
- C1213 (R1215) The *name* shall not be the same as a keyword that specifies an intrinsic type.
- C1214 If a procedure entity has the INTENT attribute or SAVE attribute, it shall also have the POINTER attribute.
- C1215 (R1211) If a *proc-interface* describes an elemental procedure, each *procedure-entity-name* shall

specify an external procedure.

- C1216 (R1214) If => appears in *proc-decl*, the procedure entity shall have the POINTER attribute.
- C1217 (R1211) If *proc-language-binding-spec* with a NAME= is specified, then *proc-decl-list* shall contain exactly one *proc-decl*, which shall neither have the POINTER attribute nor be a dummy procedure.
- C1218 (R1211) If *proc-language-binding-spec* is specified, the *proc-interface* shall appear, it shall be an *interface-name*, and *interface-name* shall be declared with a *proc-language-binding-spec*.
- R1216 *intrinsic-stmt*                    **is**   INTRINSIC [ : : ] *intrinsic-procedure-name-list*
- C1219 (R1216) Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.
- R1217 *function-reference*           **is**   *procedure-designator* ( [ *actual-arg-spec-list* ] )
- C1220 (R1217) The *procedure-designator* shall designate a function.
- C1221 (R1217) The *actual-arg-spec-list* shall not contain an *alt-return-spec*.
- R1218 *call-stmt*                       **is**   CALL *procedure-designator* [ ( [ *actual-arg-spec-list* ] ) ]
- C1222 (R1218) The *procedure-designator* shall designate a subroutine.
- R1219 *procedure-designator*       **is**   *procedure-name*  
   **or**   *proc-component-ref*  
   **or**   *data-ref* % *binding-name*
- C1223 (R1219) A *procedure-name* shall be the name of a procedure or procedure pointer.
- C1224 (R1219) A *binding-name* shall be a binding name (4.5.4) of the declared type of *data-ref*.
- R1220 *actual-arg-spec*               **is**   [ *keyword* = ] *actual-arg*
- R1221 *actual-arg*                      **is**   *expr*  
   **or**   *variable*  
   **or**   *procedure-name*  
   **or**   *proc-component-ref*  
   **or**   *alt-return-spec*
- R1222 *alt-return-spec*               **is**   \* *label*
- C1225 (R1220) The *keyword* = shall not appear if the interface of the procedure is implicit in the scoping unit.
- C1226 (R1220) The *keyword* = shall not be omitted from an *actual-arg-spec* unless it has been omitted from each preceding *actual-arg-spec* in the argument list.
- C1227 (R1220) Each *keyword* shall be the name of a dummy argument in the explicit interface of the procedure.
- C1228 (R1221) A nonintrinsic elemental procedure shall not be used as an actual argument.
- C1229 (R1221) A *procedure-name* shall be the name of an external procedure, a dummy procedure, a module procedure, a procedure pointer, or a specific intrinsic function that is listed in 13.6 and not marked with a bullet(●).
- C1230 (R1221) In a reference to a pure procedure, a *procedure-name actual-arg* shall be the name of a pure procedure (12.6).
- C1231 (R1222) The *label* used in the *alt-return-spec* shall be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.
- C1232 (R1221) If an actual argument is an array section or an assumed-shape array, and the corresponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be an assumed-shape array.
- C1233 (R1221) If an actual argument is a pointer array, and the corresponding dummy argument has either the VOLATILE or ASYNCHRONOUS attribute, that dummy argument shall be an assumed-shape array or a pointer array.
- R1223 *function-subprogram*       **is**   *function-stmt*  
   [ *specification-part* ]

- [ *execution-part* ]  
 [ *internal-subprogram-part* ]  
*end-function-stmt*
- R1224 *function-stmt*            **is** [ *prefix* ] FUNCTION *function-name* ■  
 ■ ( [ *dummy-arg-name-list* ] ) [ *suffix* ]
- C1234 (R1224) If RESULT is specified, *result-name* shall not be the same as *function-name* and shall not be the same as the *entry-name* in any ENTRY statement in the subprogram.
- C1235 (R1224) If RESULT is specified, the *function-name* shall not appear in any specification statement in the scoping unit of the function subprogram.
- R1225 *proc-language-binding-spec*   **is** *language-binding-spec*
- C1236 (R1225) A *proc-language-binding-spec* with a NAME= specifier shall not be specified in the *function-stmt* or *subroutine-stmt* of an interface body for an abstract interface or a dummy procedure.
- C1237 (R1225) A *proc-language-binding-spec* shall not be specified for an internal procedure.
- C1238 (R1225) If *proc-language-binding-spec* is specified for a procedure, each of the procedure's dummy arguments shall be a nonoptional interoperable variable (15.2.4, 15.2.5) or an interoperable procedure (15.2.6). If *proc-language-binding-spec* is specified for a function, the function result shall be an interoperable variable.
- R1226 *dummy-arg-name*            **is** *name*
- C1239 (R1226) A *dummy-arg-name* shall be the name of a dummy argument.
- R1227 *prefix*                    **is** *prefix-spec* [ *prefix-spec* ] ...
- R1228 *prefix-spec*                **is** *declaration-type-spec*  
**or** RECURSIVE  
**or** PURE  
**or** ELEMENTAL
- C1240 (R1227) A *prefix* shall contain at most one of each *prefix-spec*.
- C1241 (R1227) A *prefix* shall not specify both ELEMENTAL and RECURSIVE.
- C1242 (R1227) A *prefix* shall not specify ELEMENTAL if *proc-language-binding-spec* appears in the *function-stmt* or *subroutine-stmt*.
- R1229 *suffix*                    **is** *proc-language-binding-spec* [ RESULT ( *result-name* ) ]  
**or** RESULT ( *result-name* ) [ *proc-language-binding-spec* ]
- R1230 *end-function-stmt*        **is** END [ FUNCTION [ *function-name* ] ]
- C1243 (R1230) FUNCTION shall appear in the *end-function-stmt* of an internal or module function.
- C1244 (R1223) An internal function subprogram shall not contain an ENTRY statement.
- C1245 (R1223) An internal function subprogram shall not contain an *internal-subprogram-part*.
- C1246 (R1230) If a *function-name* appears in the *end-function-stmt*, it shall be identical to the *function-name* specified in the *function-stmt*.
- R1231 *subroutine-subprogram*    **is** *subroutine-stmt*  
   [ *specification-part* ]  
   [ *execution-part* ]  
   [ *internal-subprogram-part* ]  
   *end-subroutine-stmt*
- R1232 *subroutine-stmt*            **is** [ *prefix* ] SUBROUTINE *subroutine-name* ■  
 ■ [ ( [ *dummy-arg-list* ] ) [ *proc-language-binding-spec* ] ]
- C1247 (R1232) The *prefix* of a *subroutine-stmt* shall not contain a *declaration-type-spec*.
- R1233 *dummy-arg*                **is** *dummy-arg-name*  
**or** \*

- R1234 *end-subroutine-stmt*                    **is** END [ SUBROUTINE [ *subroutine-name* ] ]
- C1248 (R1234) SUBROUTINE shall appear in the *end-subroutine-stmt* of an internal or module subroutine.
- C1249 (R1231) An internal subroutine subprogram shall not contain an ENTRY statement.
- C1250 (R1231) An internal subroutine subprogram shall not contain an *internal-subprogram-part*.
- C1251 (R1234) If a *subroutine-name* appears in the *end-subroutine-stmt*, it shall be identical to the *subroutine-name* specified in the *subroutine-stmt*.
- R1235 *entry-stmt*                                **is** ENTRY *entry-name* [ ( [ *dummy-arg-list* ] ) [ *suffix* ] ]
- C1252 (R1235) If RESULT is specified, the *entry-name* shall not appear in any specification or type-declaration statement in the scoping unit of the function program.
- C1253 (R1235) An *entry-stmt* shall appear only in an *external-subprogram* or *module-subprogram*. An *entry-stmt* shall not appear within an *executable-construct*.
- C1254 (R1235) RESULT shall appear only if the *entry-stmt* is in a function subprogram.
- C1255 (R1235) Within the subprogram containing the *entry-stmt*, the *entry-name* shall not appear as a dummy argument in the FUNCTION or SUBROUTINE statement or in another ENTRY statement nor shall it appear in an EXTERNAL, INTRINSIC, or PROCEDURE statement.
- C1256 (R1235) A *dummy-arg* shall not be an alternate return indicator if the ENTRY statement is in a function subprogram.
- C1257 (R1235) If RESULT is specified, *result-name* shall not be the same as the *function-name* in the FUNCTION statement and shall not be the same as the *entry-name* in any ENTRY statement in the subprogram.
- R1236 *return-stmt*                                **is** RETURN [ *scalar-int-expr* ]
- C1258 (R1236) The *return-stmt* shall be in the scoping unit of a function or subroutine subprogram.
- C1259 (R1236) The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.
- R1237 *contains-stmt*                            **is** CONTAINS
- R1238 *stmt-function-stmt*                    **is** *function-name* ( [ *dummy-arg-name-list* ] ) = *scalar-expr*
- C1260 (R1238) The *primaries* of the *scalar-expr* shall be constants (literal and named), references to variables, references to functions and function dummy procedures, and intrinsic operations. If *scalar-expr* contains a reference to a function or a function dummy procedure, the reference shall not require an explicit interface, the function shall not require an explicit interface unless it is an intrinsic, the function shall not be a transformational intrinsic, and the result shall be scalar. If an argument to a function or a function dummy procedure is an array, it shall be an array name. If a reference to a statement function appears in *scalar-expr*, its definition shall have been provided earlier in the scoping unit and shall not be the name of the statement function being defined.
- C1261 (R1238) Named constants in *scalar-expr* shall have been declared earlier in the scoping unit or made accessible by use or host association. If array elements appear in *scalar-expr*, the array shall have been declared as an array earlier in the scoping unit or made accessible by use or host association.
- C1262 (R1238) If a *dummy-arg-name*, variable, function reference, or dummy function reference is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm this implied type and the values of any implied type parameters.
- C1263 (R1238) The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.
- C1264 (R1238) A given *dummy-arg-name* shall not appear more than once in any *dummy-arg-name-list*.
- C1265 (R1238) Each variable reference in *scalar-expr* may be either a reference to a dummy argument of the statement function or a reference to a variable accessible in the same scoping unit as the statement function statement.
- C1266 The *specification-part* of a pure function subprogram shall specify that all its nonpointer dummy data objects have INTENT(IN).
- C1267 The *specification-part* of a pure subroutine subprogram shall specify the intents of all its nonpointer dummy data objects.
- C1268 A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure subprogram shall not have the SAVE attribute.
- C1269 The *specification-part* of a pure subprogram shall specify that all its dummy procedures are

pure.

- C1270 If a procedure that is neither an intrinsic procedure nor a statement function is used in a context that requires it to be pure, then its interface shall be explicit in the scope of that use. The interface shall specify that the procedure is pure.
- C1271 All internal subprograms in a pure subprogram shall be pure.
- C1272 In a pure subprogram any designator with a base object that is in common or accessed by host or use association, is a dummy argument of a pure function, is a dummy argument with INTENT (IN) of a pure subroutine, or an object that is storage associated with any such variable, shall not be used in the following contexts:
- C1273 Any procedure referenced in a pure subprogram, including one referenced via a defined operation, assignment, or finalization, shall be pure.
- C1274 A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-stmt*, *rewind-stmt*, *flush-stmt*, *wait-stmt*, or *inquire-stmt*.
- C1275 A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is a *file-unit-number* or *\**.
- C1276 A pure subprogram shall not contain a *stop-stmt*.
- C1277 All dummy arguments of an elemental procedure shall be scalar dummy data objects and shall not have the POINTER or ALLOCATABLE attribute.
- C1278 The result variable of an elemental function shall be scalar and shall not have the POINTER or ALLOCATABLE attribute.
- C1279 In the scoping unit of an elemental subprogram, an object designator with a dummy argument as the base object shall not appear in a *specification-expr* except as the argument to one of the intrinsic functions BIT\_SIZE, KIND, LEN, or the numeric inquiry functions (13.5.6).

### Section 13:

### Section 14:

### Section 15:

- C1501 (R429) A derived type with the BIND attribute shall not be a SEQUENCE type.
- C1502 (R429) A derived type with the BIND attribute shall not have type parameters.
- C1503 (R429) A derived type with the BIND attribute shall not have the EXTENDS attribute.
- C1504 (R429) A derived type with the BIND attribute shall not have a *type-bound-procedure-part*.
- C1505 (R429) Each component of a derived type with the BIND attribute shall be a nonpointer, nonallocatable data component with interoperable type and type parameters.

### Section 16:

## D.2 Syntax rule cross-reference

R472	<i>ac-do-variable</i>	R471, C493, C497
R470	<i>ac-IMPLIED-DO</i>	R469, C497
R471	<i>ac-IMPLIED-DO-CONTROL</i>	R470
R466	<i>ac-spec</i>	R465
R469	<i>ac-value</i>	R466, R470, C494, C495, C496
R519	<i>access-id</i>	R518, C548
R508	<i>access-spec</i>	R431, R441, R446, R452, R453, R503, C539, R518, R1213
R518	<i>access-stmt</i>	R212, C548
R214	<i>action-stmt</i>	R213, R807, C802
R836	<i>action-term-do-construct</i>	R835
R1221	<i>actual-arg</i>	R1220, C1230

R1220	<i>actual-arg-spec</i>	C489, R1217, C1221, R1218, C1226
R709	<i>add-op</i>	R310, R706
R705	<i>add-operand</i>	R705, R706
R624	<i>alloc-opt</i>	R623, C630
R520	<i>allocatable-stmt</i>	R212
R629	<i>allocate-object</i>	R628, C622, C623, C624, C625, C626, C627, C628, C629, C631, C632, C633, R635, C635
R630	<i>allocate-shape-spec</i>	R628, C628, C629
R623	<i>allocate-stmt</i>	R214
R628	<i>allocation</i>	R623, C631
R302	<i>alphanumeric-character</i>	R301, R304
R1222	<i>alt-return-spec</i>	C1221, R1221, C1231
R719	<i>and-op</i>	R310, R715
R714	<i>and-operand</i>	R715
—	<i>arg-name</i>	R446, C451, R453, C467
R847	<i>arithmetic-if-stmt</i>	R214, C824, C826, C832
R465	<i>array-constructor</i>	C494, C495, C496, R701
R616	<i>array-element</i>	R528, C558, C561, R556, R603, R610
—	<i>array-name</i>	R535
R617	<i>array-section</i>	R603
R510	<i>array-spec</i>	R503, R504, C510, C511, R535, R546
R734	<i>assignment-stmt</i>	R214, C715, R747, R757
R816	<i>associate-construct</i>	R213, C810
—	<i>associate-construct-name</i>	R817, R820, C810
—	<i>associate-name</i>	R818, C808, C809, R822, C811, C812
R817	<i>associate-stmt</i>	R816, C809, C810
R818	<i>association</i>	R817
R514	<i>assumed-shape-spec</i>	R510
R516	<i>assumed-size-spec</i>	R510
R521	<i>asynchronous-stmt</i>	R212
R503	<i>attr-spec</i>	R501, C507
R923	<i>backspace-stmt</i>	R214, C1274
R412	<i>binary-constant</i>	R411
R523	<i>bind-entity</i>	R522, C550, C551, C552
R522	<i>bind-stmt</i>	R212, C550
R453	<i>binding-attr</i>	R451, C465, C468, C469
—	<i>binding-name</i>	R451, R452, C460, R1219, C1224
R449	<i>binding-private-stmt</i>	R448, C455
R1016	<i>blank-interp-edit-desc</i>	R1011
R801	<i>block</i>	R802, R808, R816, R821, R832
R1116	<i>block-data</i>	R202, C1117, C1118
—	<i>block-data-name</i>	R1117, R1118, C1116
R1117	<i>block-data-stmt</i>	R1116, C1116
R826	<i>block-do-construct</i>	R825, C821
R738	<i>bounds-remapping</i>	R735, C719, C720
R737	<i>bounds-spec</i>	R735, C718
R411	<i>boz-literal-constant</i>	R306, C410

R1218	<i>call-stmt</i>	R214, C1231
R808	<i>case-construct</i>	R213, C803, C805, C807
—	<i>case-construct-name</i>	R809, R810, R811, C803
R812	<i>case-expr</i>	R809, C805, C806, C807
R813	<i>case-selector</i>	R810
R810	<i>case-stmt</i>	R808, C803
R815	<i>case-value</i>	R814, C805
R814	<i>case-value-range</i>	R813, C806, C807
R309	<i>char-constant</i>	C303, R850, C834
R725	<i>char-expr</i>	C706, R731, R812
R731	<i>char-initialization-expr</i>	R509, C540, C712, R815, R913, C924
R426	<i>char-length</i>	R425, R442, C444, R504, C504, C520
R427	<i>char-literal-constant</i>	R306, R1005, C1008, R1019, C1012
R424	<i>char-selector</i>	R403
R1019	<i>char-string-edit-desc</i>	R1003
R606	<i>char-variable</i>	C605, R903, C901, C902
R909	<i>close-spec</i>	R908, C906, C907
R908	<i>close-stmt</i>	R214, C1274
—	<i>common-block-name</i>	R523, R544, R557
R558	<i>common-block-object</i>	R557, C587, C588, C589
R557	<i>common-stmt</i>	R212
R421	<i>complex-literal-constant</i>	R306
R443	<i>component-array-spec</i>	R441, R442, C440, C441
R441	<i>component-attr-spec</i>	R440, C436, C447
R459	<i>component-data-source</i>	R458
R442	<i>component-decl</i>	R440, C446
R439	<i>component-def-stmt</i>	R438, C436, C438, C439, C445
R444	<i>component-initialization</i>	R442, C446, C447, C1107
—	<i>component-name</i>	R442
R438	<i>component-part</i>	R429
R458	<i>component-spec</i>	R457, C483, C484, C485, C486, C488, C489
R846	<i>computed-goto-stmt</i>	R214, C831
R711	<i>concat-op</i>	R310, R710
R905	<i>connect-spec</i>	R904, C903, C904
R305	<i>constant</i>	R308, R309, R532, R610, R701
R534	<i>constant-subobject</i>	R532, R533, C566
R1237	<i>contains-stmt</i>	R210, R448, R1107
R848	<i>continue-stmt</i>	R214, R833, C824
R1011	<i>control-edit-desc</i>	R1003
R843	<i>cycle-stmt</i>	R214, C824, C826, C828
R1008	<i>d</i>	R1005, C1007
R440	<i>data-component-def-stmt</i>	R439
R1005	<i>data-edit-desc</i>	R1003
R528	<i>data-i-do-object</i>	R527, C554, C555, C561
R529	<i>data-i-do-variable</i>	R527, C556
R527	<i>data-implied-do</i>	R526, R528, C557, C561

—	<i>data-pointer-component-name</i>	R736, C722
R736	<i>data-pointer-object</i>	R735, C716, C717, C718, C719, C720
R612	<i>data-ref</i>	C611, R614, R616, R617, R1219, C1224
R524	<i>data-stmt</i>	R209, R212, C1206
R532	<i>data-stmt-constant</i>	C410, R530, C564
R526	<i>data-stmt-object</i>	R525, C553, C554, C555
R531	<i>data-stmt-repeat</i>	R530, C562
R525	<i>data-stmt-set</i>	R524
R530	<i>data-stmt-value</i>	R525
R739	<i>data-target</i>	R459, C490, C491, C537, R735, C716, C717, C720
R636	<i>dealloc-opt</i>	R635, C636
R635	<i>deallocate-stmt</i>	R214
R1018	<i>decimal-edit-desc</i>	R1011
R207	<i>declaration-construct</i>	R204
R502	<i>declaration-type-spec</i>	C419, R440, C438, C439, R501, C501, C502, R550, R1212, R1228, C1247
R726	<i>default-char-expr</i>	C707, R905, R906, R909, R913, R914
R607	<i>default-char-variable</i>	C606, R626, R907, R930
R605	<i>default-logical-variable</i>	C604, R930
R515	<i>deferred-shape-spec</i>	R443, C440, C510, R510, R520, R541
R723	<i>defined-binary-op</i>	R311, R722, C704, R1114, R1115
R311	<i>defined-operator</i>	C462, R1207, C1202
R703	<i>defined-unary-op</i>	R311, R702, C703, R1114, R1115
R429	<i>derived-type-def</i>	R207, C430, C434, C435
R455	<i>derived-type-spec</i>	R401, C401, R457, C482, C489, R502, C502, C503, R920, C936, C937
R430	<i>derived-type-stmt</i>	R429, C425, C431, C434, C435
R603	<i>designator</i>	R534, R601, C601, R615, C616, R701, C702
—	<i>digit</i>	R302, R313, R409, R412, C408, R413, C409, R415, R850
R409	<i>digit-string</i>	R406, R407, R408, R417, R418
R535	<i>dimension-stmt</i>	R212
R832	<i>do-block</i>	R826
R837	<i>do-body</i>	R836, R839, R841
R825	<i>do-construct</i>	R213, C828, C829
—	<i>do-construct-name</i>	R828, R829, R834, C821, R843, C828, R844, C829
R827	<i>do-stmt</i>	R826, C821, C822, C823
R838	<i>do-term-action-stmt</i>	R836, C824, C825
R842	<i>do-term-shared-stmt</i>	R841, C826, C827
R831	<i>do-variable</i>	R830, C820, R919, C933
R1208	<i>dtio-generic-spec</i>	C464, R1207, C1202
R1233	<i>dummy-arg</i>	R1232, R1235, C1256
R1226	<i>dummy-arg-name</i>	R536, R537, R547, R1224, C1239, R1233, R1238, C1262, C1263, C1264
R1009	<i>e</i>	R1005, C1005, C1007
R804	<i>else-if-stmt</i>	R802, C801
R805	<i>else-stmt</i>	R802, C801
R750	<i>elsewhere-stmt</i>	R744, C730

R820	<i>end-associate-stmt</i>	R816, C810
R1118	<i>end-block-data-stmt</i>	R1116, C1116
R833	<i>end-do</i>	R826, C821, C822, C823
R834	<i>end-do-stmt</i>	R833, C821, C822
R464	<i>end-enum-stmt</i>	R460
R758	<i>end-forall-stmt</i>	R752, C732
R1230	<i>end-function-stmt</i>	R214, C201, C802, C824, C826, R1205, R1223, C1243, C1246
R806	<i>end-if-stmt</i>	R802, C801
R1204	<i>end-interface-stmt</i>	R1201, C1202
R1106	<i>end-module-stmt</i>	R1104, C1104
R1103	<i>end-program-stmt</i>	R214, C201, C802, C824, C826, R1101, C1102
R811	<i>end-select-stmt</i>	R808, C803
R824	<i>end-select-type-stmt</i>	R821, C819
R1234	<i>end-subroutine-stmt</i>	R214, C201, C802, C824, C826, R1205, R1231, C1248, C1251
R433	<i>end-type-stmt</i>	R429
R751	<i>end-where-stmt</i>	R744, C730
R924	<i>endfile-stmt</i>	R214, C1274
R504	<i>entity-decl</i>	R501, C504, C519, C523, C533
—	<i>entity-name</i>	R523, C550, R542
—	<i>entry-name</i>	C1234, R1235, C1252, C1255, C1257
R1235	<i>entry-stmt</i>	R206, R207, R209, C1105, C1206, C1253, C1254, C1255
R460	<i>enum-def</i>	R207
R461	<i>enum-def-stmt</i>	R460
R463	<i>enumerator</i>	R462, C492
R462	<i>enumerator-def-stmt</i>	R460
R721	<i>equiv-op</i>	R310, R717
R716	<i>equiv-operand</i>	R716, R717
R556	<i>equivalence-object</i>	R555, C576, C577, C578, C579, C580, C581, C582, C583, C584, C585
R555	<i>equivalence-set</i>	R554
R554	<i>equivalence-stmt</i>	R212
R626	<i>errmsg-variable</i>	R624, R636
R213	<i>executable-construct</i>	R208, R209, C1253
R208	<i>execution-part</i>	C201, R1101, C1101, R1223, R1231
R209	<i>execution-part-construct</i>	R208, R801, R837
R844	<i>exit-stmt</i>	R214, C824, C826, C829
R511	<i>explicit-shape-spec</i>	R443, C441, C442, C511, R510, R516, R558
R420	<i>exponent</i>	R417
R419	<i>exponent-letter</i>	R417, C411
R722	<i>expr</i>	R459, R469, R627, R701, R722, R724, R725, R726, R727, R728, R730, R734, R739, C724, R742, C726, R819, R916, R1221, R1238, C1260, C1261, C1265
R312	<i>extended-intrinsic-op</i>	R311
—	<i>external-name</i>	R1210
R1210	<i>external-stmt</i>	R212
R203	<i>external-subprogram</i>	R202, C1253

R906	<i>file-name-expr</i>	R905, R930
R902	<i>file-unit-number</i>	R901, R905, C904, R909, C907, C919, C925, R922, C939, R923, R924, R925, R926, C942, R927, R928, C945, R930, C949, C1275
R454	<i>final-binding</i>	R450
—	<i>final-subroutine-name</i>	R454, C473, C474
R928	<i>flush-spec</i>	R927, C944, C945
R927	<i>flush-stmt</i>	R214, C1274
R757	<i>forall-assignment-stmt</i>	R756, R759
R756	<i>forall-body-construct</i>	R752, C737, C738, C739
R752	<i>forall-construct</i>	R213, R756, C737
—	<i>forall-construct-name</i>	R753, R758, C732
R753	<i>forall-construct-stmt</i>	R752, C732
R754	<i>forall-header</i>	R753, R759
R759	<i>forall-stmt</i>	R214, R756
R755	<i>forall-triplet-spec</i>	R754, C736
R914	<i>format</i>	R910, R912, R913, C916, C917, C920, C928, C929
R1003	<i>format-item</i>	R1002, C1002, R1003
R1002	<i>format-specification</i>	R1001
R1001	<i>format-stmt</i>	R206, R207, R209, C1001, C1105, C1206
—	<i>function-name</i>	R504, C521, C1203, R1224, C1234, C1235, R1230, C1246, C1257, R1238, C1263
R1217	<i>function-reference</i>	R507, C506, R701
R1224	<i>function-stmt</i>	R1205, C1203, R1223, C1236, C1242, C1246
R1223	<i>function-subprogram</i>	R203, R211, R1108
R452	<i>generic-binding</i>	R450, C459, C1111, C1112
—	<i>generic-name</i>	C461, R1207, C1202
R1207	<i>generic-spec</i>	R452, C459, C461, C462, C463, C464, R519, R1112, C1112, C1113, R1203, R1204, C1202, C1204
R845	<i>goto-stmt</i>	R214, C824, C826, C830
R414	<i>hex-constant</i>	R411
R415	<i>hex-digit</i>	R414
R802	<i>if-construct</i>	R213, C801
—	<i>if-construct-name</i>	R803, R804, R805, R806, C801
R807	<i>if-stmt</i>	R214, C802
R803	<i>if-then-stmt</i>	R802, C801
R423	<i>imag-part</i>	R421
R205	<i>implicit-part</i>	R204
R206	<i>implicit-part-stmt</i>	R205
R550	<i>implicit-spec</i>	R549
R549	<i>implicit-stmt</i>	R205, R206
—	<i>import-name</i>	R1209, C1211
R1209	<i>import-stmt</i>	R204
—	<i>index-name</i>	R755, C735, C736, C737
R506	<i>initialization</i>	R504, C522, C523, C524, C525
R730	<i>initialization-expr</i>	R444, R506, R539, C711
R841	<i>inner-shared-do-construct</i>	R840, C827

R915	<i>input-item</i>	R910, C915, R918, C931, C932, C934
R930	<i>inquire-spec</i>	R929, C947, C948, C949
R929	<i>inquire-stmt</i>	R214, C1274
R308	<i>int-constant</i>	C302, R531
—	<i>int-constant-name</i>	R407, C405
R533	<i>int-constant-subobject</i>	R531, C565
R727	<i>int-expr</i>	R402, R471, R527, C557, R611, R618, R621, R622, R631, R632, C708, R729, C710, R732, R812, R830, R846, R902, R905, R913, R919, R1236, C1259
R732	<i>int-initialization-expr</i>	R404, C404, R424, C414, R436, R463, C713, R815
R406	<i>int-literal-constant</i>	R306, R405, R426, C415, R1004, R1006, R1007, R1008, R1009, R1014
R608	<i>int-variable</i>	R472, R529, C607, R625, R831, R905, R909, R913, R922, R926, R928, R929, R930
R517	<i>intent-spec</i>	R503, R536, R1213
R536	<i>intent-stmt</i>	R212
R1201	<i>interface-block</i>	R207, C1201
R1205	<i>interface-body</i>	R1202, C1201, C1205, C1206, C1210
R1215	<i>interface-name</i>	R451, C457, C470, R1212, C1218
R1202	<i>interface-specification</i>	R1201
R1203	<i>interface-stmt</i>	R1201, C1202, C1203
R903	<i>internal-file-variable</i>	R901, C921
R211	<i>internal-subprogram</i>	R210
R210	<i>internal-subprogram-part</i>	R1101, R1223, C1245, R1231, C1250, C1268
R310	<i>intrinsic-operator</i>	R312, C703, C704
—	<i>intrinsic-procedure-name</i>	R1216, C1219
R1216	<i>intrinsic-stmt</i>	R212
R403	<i>intrinsic-type-spec</i>	R401, R502
R913	<i>io-control-spec</i>	R910, R911, C909, C910, C916, C917, C918, C919, C927
R917	<i>io-implicit-do</i>	R915, R916
R919	<i>io-implicit-do-control</i>	R917
R918	<i>io-implicit-do-object</i>	R917, C934
R901	<i>io-unit</i>	R913, C910, C917, C918, C919, C921, C925, C1275
R907	<i>iomsg-variable</i>	R905, R909, R913, R922, R926, R928, R930
R1012	<i>k</i>	R1011, C1009
R215	<i>keyword</i>	R456, C479, C480, R458, C486, C487, R1220, C1225, C1226, C1227
R407	<i>kind-param</i>	R406, C406, C407, R417, C411, C412, C415, R427, C422, R428, C423
R404	<i>kind-selector</i>	R403, R435
R313	<i>label</i>	C304, R828, C823, R845, C830, R846, C831, R847, C832, R905, C905, R909, C908, R913, C913, R914, C930, R922, C940, R926, C943, R928, C946, R930, R1222, C1231
R828	<i>label-do-stmt</i>	R827, C823, R836, C825, R839, R841, C827
R509	<i>language-binding-spec</i>	R503, C531, C532, C533, R522, C551, R1225
R467	<i>left-square-bracket</i>	R465
R425	<i>length-selector</i>	R424, C419, C420
—	<i>letter</i>	R302, R304, R551, R703, R723

R551	<i>letter-spec</i>	R550
R702	<i>level-1-expr</i>	R704
R706	<i>level-2-expr</i>	R706, R710
R710	<i>level-3-expr</i>	R710, R712
R712	<i>level-4-expr</i>	R714
R717	<i>level-5-expr</i>	R717, R722
R306	<i>literal-constant</i>	R305
R1114	<i>local-defined-operator</i>	R1111
—	<i>local-name</i>	R1111
R724	<i>logical-expr</i>	C705, R733, R748, R803, R804, R807, R812, R830
R733	<i>logical-initialization-expr</i>	C714, R815
R428	<i>logical-literal-constant</i>	R306, C703, C704
R604	<i>logical-variable</i>	C603
R830	<i>loop-control</i>	R828, R829
R512	<i>lower-bound</i>	R511, R514, R516
R631	<i>lower-bound-expr</i>	R630, R737, R738
R1007	<i>m</i>	R1005, C1007
R1101	<i>main-program</i>	R202, C1101
R748	<i>mask-expr</i>	R743, R745, R749, R754, C733, C734
R749	<i>masked-elsewhere-stmt</i>	R744, C730
R1104	<i>module</i>	R202
—	<i>module-name</i>	R1105, R1106, C1104, R1109, C1108, C1109
R1110	<i>module-nature</i>	R1109, C1108, C1109
R1105	<i>module-stmt</i>	R1104, C1104
R1108	<i>module-subprogram</i>	R1107, C1253
R1107	<i>module-subprogram-part</i>	R1104
R708	<i>mult-op</i>	R310, R705
R704	<i>mult-operand</i>	R704, R705
R1014	<i>n</i>	R1013, C1010, C1011
R304	<i>name</i>	R102, R215, C301, R307, R505, R545, R602, R1215, C1212, C1213, R1226
R307	<i>named-constant</i>	R305, R422, R423, R463, R539
R539	<i>named-constant-def</i>	R538
—	<i>namelist-group-name</i>	R552, C573, C575, R913, C914, C915, C916, C918, C920, C928, C929
R553	<i>namelist-group-object</i>	R552, C574, C575
R552	<i>namelist-stmt</i>	R212
R835	<i>nonblock-do-construct</i>	R825
R829	<i>nonlabel-do-stmt</i>	R827, C822
R718	<i>not-op</i>	R310, R714
R507	<i>null-init</i>	R444, R506, R532, R1214
R633	<i>nullify-stmt</i>	R214
R728	<i>numeric-expr</i>	C709, R847, C833
R505	<i>object-name</i>	R504, C505, C511, C524, R520, R521, R541, R544, R546, R548, R603
R413	<i>octal-constant</i>	R411
R1112	<i>only</i>	R1109

R1113	<i>only-use-name</i>	R1112
R904	<i>open-stmt</i>	R214, C1274
R537	<i>optional-stmt</i>	R212
R720	<i>or-op</i>	R310, R716
R715	<i>or-operand</i>	R715, R716
R839	<i>outer-shared-do-construct</i>	R835, R840, C827
R916	<i>output-item</i>	R911, R912, C915, R918, C934, C935, R929
R538	<i>parameter-stmt</i>	R206, R207
R610	<i>parent-string</i>	R609, C608
—	<i>parent-type-name</i>	R431, C426
—	<i>part-name</i>	R613, C609, C610, C611, C612, C613, C614, C615, C619
R613	<i>part-ref</i>	C555, C560, C577, R612, C614, C615, C617, C618
R735	<i>pointer-assignment-stmt</i>	R214, C537, R757
R541	<i>pointer-decl</i>	R540
R634	<i>pointer-object</i>	R633, C634
R540	<i>pointer-stmt</i>	R212
R1013	<i>position-edit-desc</i>	R1011
R926	<i>position-spec</i>	R923, R924, R925, C941, C942
R707	<i>power-op</i>	R310, R704
R1227	<i>prefix</i>	R1224, C1240, C1241, C1242, R1232, C1247
R1228	<i>prefix-spec</i>	R1227, C1240
—	<i>primaries</i>	C1260
R701	<i>primary</i>	R702
R912	<i>print-stmt</i>	R214, C1274
R447	<i>private-components-stmt</i>	R432, C454
R432	<i>private-or-sequence</i>	R429, C430
R1213	<i>proc-attr-spec</i>	R1211
R450	<i>proc-binding-stmt</i>	R448
R446	<i>proc-component-attr-spec</i>	R445, C448, C449, C452
R445	<i>proc-component-def-stmt</i>	R439, C448
R741	<i>proc-component-ref</i>	R740, R742, R1219, R1221
R1214	<i>proc-decl</i>	R445, R1211, C1216, C1217
—	<i>proc-entity-name</i>	R541, C568
R1212	<i>proc-interface</i>	R445, R1211, C1215, C1218
R1225	<i>proc-language-binding-spec</i>	C530, R1213, C1217, C1218, C1236, C1237, C1238, C1242, R1229, R1232
R545	<i>proc-pointer-name</i>	R544, C569, R558, C587, C590, R634, R740
R740	<i>proc-pointer-object</i>	R735
R742	<i>proc-target</i>	R459, C490, C537, R735, C728
—	<i>procedure-component-name</i>	R741, C725
R1211	<i>procedure-declaration-stmt</i>	R207, C568, C1212
R1219	<i>procedure-designator</i>	R1217, C1220, R1218, C1222
—	<i>procedure-entity-name</i>	R1214, C1215
—	<i>procedure-name</i>	R451, C456, C457, C458, R742, C727, R1206, C1207, C1208, C1209, R1219, C1223, R1221, C1229, C1230
R1206	<i>procedure-stmt</i>	R1202, C1204, C1208, C1209
—	<i>program-name</i>	R1102, R1103, C1102

R1102	<i>program-stmt</i>	R1101, C1102
R202	<i>program-unit</i>	R201
R542	<i>protected-stmt</i>	R212
R1004	<i>r</i>	R1003, C1003, C1004, R1011
R910	<i>read-stmt</i>	R214, C911, C1275
R417	<i>real-literal-constant</i>	R306, R416
R422	<i>real-part</i>	R421
R713	<i>rel-op</i>	R310, R712
R1111	<i>rename</i>	R1109, R1112
—	<i>rep-char</i>	R427
—	<i>result-name</i>	C1234, R1229, C1257
R1236	<i>return-stmt</i>	R214, C824, C826, C1258
R925	<i>rewind-stmt</i>	R214, C1274
R468	<i>right-square-bracket</i>	R465
R1017	<i>round-edit-desc</i>	R1011
R543	<i>save-stmt</i>	R212
R544	<i>saved-entity</i>	R543
R103	<i>scalar-xyz</i>	C101
R619	<i>section-subscript</i>	R613, C613, C618
R809	<i>select-case-stmt</i>	R808, C803
—	<i>select-construct-name</i>	R822, R823, R824, C819
R821	<i>select-type-construct</i>	R213, C817, C818, C819
R822	<i>select-type-stmt</i>	R821, C813, C819
R819	<i>selector</i>	R818, C808, R822, C811, C812, C813, C816
R434	<i>sequence-stmt</i>	R432
R840	<i>shared-term-do-construct</i>	R839
R410	<i>sign</i>	R405, R408, R416
R1015	<i>sign-edit-desc</i>	R1011
R408	<i>signed-digit-string</i>	R420
R405	<i>signed-int-literal-constant</i>	R422, R423, R532, R1010, R1012
R416	<i>signed-real-literal-constant</i>	R422, R423, R532
R418	<i>significand</i>	R417
R627	<i>source-expr</i>	R624, C631, C632, C633
—	<i>special-character</i>	R301
R451	<i>specific-binding</i>	R450
R729	<i>specification-expr</i>	C501, C504, R512, R513, C1279
R204	<i>specification-part</i>	C459, C539, C548, R1101, C1103, R1104, C1105, C1106, C1107, R1116, C1117, C1118, R1205, R1223, R1231, C1266, C1267, C1268, C1269
R212	<i>specification-stmt</i>	R207
R625	<i>stat-variable</i>	R624, R636
R1238	<i>stmt-function-stmt</i>	R207, C1105, C1206
R850	<i>stop-code</i>	R849
R849	<i>stop-stmt</i>	R214, C824, C826, C1276
R621	<i>stride</i>	R620, R755, C736
R614	<i>structure-component</i>	R528, C559, C560, C561, R603, R610, R629, R634
R457	<i>structure-constructor</i>	R532, C564, R701

—	<i>subroutine-name</i>	C1203, R1232, R1234, C1251
R1232	<i>subroutine-stmt</i>	R1205, C1203, C1236, C1242, R1231, C1247, C1251
R1231	<i>subroutine-subprogram</i>	R203, R211, R1108
R618	<i>subscript</i>	C560, C617, R619, R620, R755, C736
R620	<i>subscript-triplet</i>	R619, C621
R609	<i>substring</i>	R556, C586, R603
R611	<i>substring-range</i>	R609, R617, C619
R1229	<i>suffix</i>	R1224, R1235
R546	<i>target-stmt</i>	R212
R431	<i>type-attr-spec</i>	R430, C425
R448	<i>type-bound-procedure-part</i>	R429, C433, C1504
R501	<i>type-declaration-stmt</i>	R207, C419, C420, C507
R823	<i>type-guard-stmt</i>	R821, C817, C818, C819
—	<i>type-name</i>	R430, C424, R433, C431, C464, R455, C476
R437	<i>type-param-attr-spec</i>	R435
R436	<i>type-param-decl</i>	R435
R435	<i>type-param-def-stmt</i>	R429, C434, C435
R615	<i>type-param-inquiry</i>	R701
—	<i>type-param-name</i>	R430, R436, C434, C435, R615, C616, R701, C701
R456	<i>type-param-spec</i>	R455, C477, C478, C479, C481
R402	<i>type-param-value</i>	C402, C403, R424, R425, R426, C416, C417, C418, C445, R456, C481, C501, C504, C626
R401	<i>type-spec</i>	R466, C494, C495, C496, R623, C623, C624, C625, C626, C627, C631, R823, C814, C815, C816
R303	<i>underscore</i>	R302
R513	<i>upper-bound</i>	R511
R632	<i>upper-bound-expr</i>	R630, R738
R1115	<i>use-defined-operator</i>	R1111, C1111, C1115
—	<i>use-name</i>	R519, C549, R1111, R1113, C1114
R1109	<i>use-stmt</i>	R204
R1010	<i>v</i>	R1005, C1007
R547	<i>value-stmt</i>	R212
R601	<i>variable</i>	R526, C553, C555, R604, R605, R606, R607, R608, R734, C715, R736, C722, R739, C723, R741, C725, R819, C808, C811, C812, R915, R1221
R602	<i>variable-name</i>	R553, R556, R558, C587, C590, C602, R610, R629, R634, R736, C721
R622	<i>vector-subscript</i>	R619, C620
R548	<i>volatile-stmt</i>	R212
R1006	<i>w</i>	R1005, C1006, C1007
R922	<i>wait-spec</i>	R921, C938, C939
R921	<i>wait-stmt</i>	R214, C1274
R747	<i>where-assignment-stmt</i>	R743, R746, C729
R746	<i>where-body-construct</i>	R744, C731
R744	<i>where-construct</i>	R213, R746, R756
—	<i>where-construct-name</i>	R745, R749, R750, R751, C730
R745	<i>where-construct-stmt</i>	R744, C730
R743	<i>where-stmt</i>	R214, R746, R756

R911 *write-stmt*

R214, C912, C1275

# Annex E

(Informative)

## Index

In this index, entries in *italics* denote BNF terms, entries in **bold face** denote language keywords, and page numbers in **bold face** denote primary or defining text.

### Symbols

<, 134  
 <=, 134  
 >, 134  
 >=, 134  
 \*, 29, 34, 35, 40, 75, 80, 88, 133, 195, 196, 227, 238, 242, 267, 282  
 \*\*, 133  
 +, 133  
 -, 133  
 .AND., 135  
 .EQ., 134  
 .EQV., 136  
 .GE., 134  
 .GT., 134  
 .LE., 134  
 .LT., 134  
 .NE., 134  
 .NEQV., 135  
 .NOT., 135  
 .OR., 135  
 /, 133  
 //, 42, 134  
 /=, 135  
 ;, 29  
 ==, 134  
 &, 28  
 &, 243

### A

abstract interface, 257  
 abstract interface block, 259  
 abstract type, 60, 425  
*ac-do-variable* (R472), 67, 67, 68, 125, 127, 409  
*ac-implied-do* (R470), 67, 67, 68, 128  
*ac-implied-do-control* (R471), 67, 67, 125–128  
*ac-spec* (R466), 67, 67  
*ac-value* (R469), 67, 67, 68  
 access methods, 172  
*access-id* (R519), 86, 86  
*access-spec* (R508), 45, 46, 50, 51, 55–58, 71, 76, 76, 86, 264  
*access-stmt* (R518), 10, 86, 86  
 ACCESS= specifier, 182, 211

accessibility attribute, 76  
 accessibility statements, 86  
 action statement, 425  
*action-stmt* (R214), 10, 11, 157, 169  
*action-term-do-construct* (R836), 165, 165  
 ACTION= specifier, 182, 211  
 actions, 172  
 active, 166  
 active combination of, 151  
 actual argument, 255, 425  
*actual-arg* (R1221), 267, 267  
*actual-arg-spec* (R1220), 64, 266, 267, 267  
*add-op* (R709), 25, 118, 118  
*add-operand* (R705), 118, 118, 137  
 ADVANCE= specifier, 189  
 advancing input/output statement, 175  
 affector, 190  
*alloc-opt* (R624), 110, 110, 111  
 allocatable array, 79  
 ALLOCATABLE attribute, 77, 86  
 ALLOCATABLE statement, 86  
 allocatable variable, 425  
*allocatable-stmt* (R520), 10, 86, 411  
 ALLOCATE statement, 110  
*allocate-object* (R629), 41, 74, 81, 110, 111, 111, 112, 114, 423  
*allocate-shape-spec* (R630), 110, 111, 111  
*allocate-stmt* (R623), 11, 74, 81, 110, 423  
 allocated, 112  
*allocation* (R628), 110, 110, 111, 112  
*alphanumeric-character* (R302), 23, 23, 25  
*alt-return-spec* (R1222), 169, 266, 267, 267  
 ancestor component, 61  
*and-op* (R719), 26, 120, 120  
*and-operand* (R714), 120, 120  
 APOSTROPHE (DELIM value), 247  
 approximation methods, 37  
*arg-name*, 51, 52, 57  
 argument, 425  
 argument association, 268, 410, 425  
 argument keyword, 19, 268  
 argument keywords, 291, 409  
 arithmetic IF statement, 170  
*arithmetic-if-stmt* (R847), 11, 165, 166, 170, 170

- array, **18**, 78–80, **106**, 106–110, **425**
    - assumed-shape, **79**
    - assumed-size, **80**
    - automatic, **78**
    - deferred-shape, **79**
    - explicit-shape, **78**
  - array constructor, **67**, **67**
  - array element, **18**, 108, **425**
  - array element order, **108**
  - array elements, **106**
  - array intrinsic assignment statement, **139**
  - array pointer, **79**, **425**
  - array section, **18**, **108**, **425**
    - array-constructor* (R465), **67**, **67**, 68, 117
    - array-element* (R616), 87, 88, 96, 103, 104, **107**
    - array-name*, 90, 411
    - array-section* (R617), 103, 107, **107**, 108
    - array-spec* (R510), 6, 71, 72, 74, 78, **78**, 90, 92
  - ASCII, **349**
  - ASCII character set, **40**
  - ASCII character type, **40**
  - ASCII collating sequence, **43**
  - assignment, 138–154
    - defined, **142**
    - elemental array (FORALL), **148**
    - intrinsic, **138**
    - masked array (WHERE), **145**
    - pointer, **143**
  - assignment statement, 138, **425**
    - assignment-stmt* (R734), 11, 138, **138**, 146, 148, 151, 287, **423**
  - ASSOCIATE construct, **160**
  - associate name, **425**
  - associate names, **160**
    - associate-construct* (R816), 10, **160**, 161
    - associate-construct-name*, 160, 161
    - associate-name*, 160, 162, 164, 409, 434
    - associate-stmt* (R817), 160, **160**, 161, 169
  - associated, **18**
  - associating entity, **418**
  - association, **19**, **425**
    - argument, 268, 410
    - host, 411
    - name, 410
    - pointer, 414
    - sequence, 272
    - storage, 415
    - use, 410
  - association* (R818), 160, **160**
  - association status
    - pointer, 414
  - assumed type parameter, **35**
  - assumed-shape array, **79**, **425**
    - assumed-shape-spec* (R514), 78, 79, **79**
  - assumed-size array, **80**, **425**
    - assumed-size-spec* (R516), 78, 80, **80**
  - ASYNCHRONOUS attribute, **77**, **86**
  - ASYNCHRONOUS statement, **86**
    - asynchronous-stmt* (R521), 10, **86**
  - ASYNCHRONOUS= specifier**, 182, 189, 211
    - attr-spec* (R503), 71, **71**, 72, 78
  - attribute, **425**
    - accessibility, **76**
    - ALLOCATABLE, **77**, **86**
    - ASYNCHRONOUS, **77**, **86**
    - BIND, 44, 47, 87
    - DIMENSION, 78, 90
    - EXTERNAL, **80**
    - INTENT, 81, 90
    - INTRINSIC, **82**
    - OPTIONAL, 83, 90
    - PARAMETER, 83, 90
    - POINTER, 83, 91
    - PRIVATE, 46, 76, 86
    - PROTECTED, 84, 91
    - PUBLIC, 46, 76, 86
    - SAVE, 84, 87, 91
    - TARGET, 84, 92
    - VALUE, 85, 92
    - VOLATILE, 85, 92
  - attribute specification statements, 85–100
  - attributes, **71**, 76–85
  - automatic array, **78**
  - automatic data object, **74**, **426**
- ## B
- BACKSPACE statement, **207**
    - backspace-stmt* (R923), 11, **207**, 287
  - base object, **105**
  - base type, **60**, **426**
  - belong, **426**
  - belongs, **155**, **426**
    - binary-constant* (R412), 37, **37**
  - BIND attribute, 44, 47, 87
  - BIND statement, **87**
  - BIND(C)**, 256, 258, 398, 400, 403
    - bind-entity* (R523), 87, **87**
    - bind-stmt* (R522), 10, 87, **87**
  - binding, **57**
  - binding label, **403**, **426**
  - binding name, **57**
    - binding-attr* (R453), 56, 57, **57**
    - binding-name*, 56, 57, 266, 278, 408, 435
    - binding-private-stmt* (R449), 56, **56**, 58
  - bit model, **292**
  - blank common, **98**
    - blank-interp-edit-desc* (R1016), 223, **224**
  - BLANK= specifier**, 182, 190, 211

block, **155, 426**  
*block* (R801), **155, 156, 158, 160, 162, 165**  
 block data program unit, **253, 426**  
*block-data* (R1116), **9, 253, 254**  
*block-data-name*, **253, 254**  
*block-data-stmt* (R1117), **9, 253, 253, 254**  
*block-do-construct* (R826), **165, 165**  
 bounds, **426**  
*bounds-remapping* (R738), **143, 143, 144**  
*bounds-spec* (R737), **143, 143, 144**  
*boz-literal-constant* (R411), **25, 37, 37, 89, 307, 308, 312, 323, 346**  
 branch target statement, **169**  
 Branching, **169**

## C

C address, **392**  
 C character kind, **392**  
 C\_(C type), **391–401**  
 C\_LOC function, **395**  
 CALL statement, **266**  
*call-stmt* (R1218), **11, 266, 267, 268**  
 CASE construct, **158**  
 case index, **158**  
*case-construct* (R808), **10, 158, 158**  
*case-construct-name*, **158**  
*case-expr* (R812), **158, 158**  
*case-selector* (R813), **158, 158**  
*case-stmt* (R810), **158, 158**  
*case-value* (R815), **158, 158**  
*case-value-range* (R814), **158, 158**  
 CHAR intrinsic, **43**  
*char-constant* (R309), **25, 25, 170**  
*char-expr* (R725), **123, 123, 127, 158**  
*char-initialization-expr* (R731), **77, 127, 127, 158, 186, 187**  
*char-length* (R426), **40, 40, 41, 50, 72–74**  
*char-literal-constant* (R427), **25, 30, 42, 201, 223, 224**  
*char-selector* (R424), **36, 40, 41**  
*char-string-edit-desc* (R1019), **222, 224**  
*char-variable* (R606), **103, 103, 178**  
**CHARACTER, 40**  
 character, **426**  
*character* (R301), **23**  
 character context, **27**  
 CHARACTER declaration, **439**  
 character intrinsic assignment statement, **139**  
 character intrinsic operation, **121, 134**  
 character intrinsic operator, **121**  
 character length parameter, **16, 426**  
 character literal constant, **41**  
 character relational intrinsic operation, **121**  
 character sequence type, **46, 96, 417, 520**

character set, **23**  
 character storage unit, **416, 426**  
 character string, **40, 426**  
 character string edit descriptor, **222**  
 character type, **40, 40–43**  
 characteristics, **426**  
 characteristics of a procedure, **256**  
 child data transfer statement, **199, 198–203, 219**  
**CLASS, 75**  
 class, **426**  
 CLOSE statement, **185**  
*close-spec* (R909), **185, 185**  
*close-stmt* (R908), **11, 185, 287**  
 collating sequence, **43, 43, 426**  
 comment, **28, 29**  
 common association, **99**  
 common block, **98, 407, 427, 471**  
 common block storage sequence, **99**  
 COMMON statement, **98, 98–101**  
*common-block-name*, **87, 91, 98, 253**  
*common-block-object* (R558), **98, 98, 253, 411**  
*common-stmt* (R557), **10, 98, 411**  
 companion processor, **21, 427**  
 compatibility  
   FORTRAN 77, **4**  
   Fortran 90, **3**  
   Fortran 95, **3**  
**COMPLEX, 39**  
 complex type, **39, 39**  
*complex-literal-constant* (R421), **25, 39**  
 component, **427**  
 component keyword, **19**  
 Component order, **54**  
 component order, **427**  
 component value, **62**  
*component-array-spec* (R443), **50, 50, 51**  
*component-attr-spec* (R441), **50, 50, 52**  
*component-data-source* (R459), **63, 63, 64, 65**  
*component-decl* (R442), **41, 50, 50, 51**  
*component-def-stmt* (R439), **49, 50, 50**  
*component-initialization* (R444), **50, 50, 250**  
*component-name*, **50**  
*component-part* (R438), **45, 49, 55, 58**  
*component-spec* (R458), **63, 63, 64, 126**  
 components, **408**  
 computed GO TO statement, **170**  
*computed-goto-stmt* (R846), **11, 170, 170**  
*concat-op* (R711), **25, 119, 119**  
 concatenation, **42**  
 conform, **138**  
 conformable, **18, 427**  
 conformance, **125, 427**  
*connect-spec* (R905), **181, 181**  
 connected, **179, 427**

connected files, 179  
 constant, **17**, 25, 33, **427**  
   character, 41  
   integer, 36  
   named, 90  
*constant* (R305), 25, **25**, 88, 104, 117  
 constant subobject, **17**  
*constant-subobject* (R534), 88, **88**, 117  
 construct, **427**  
 Construct association, 413  
 construct entity, **405**, **427**  
 constructor  
   array, 67  
   derived-type, 63  
   structure, 63  
 CONTAINS statement, 284  
*contains-stmt* (R1237), 10, 56, 250, **284**  
 continuation, 28, 29  
 CONTINUE statement, 170  
*continue-stmt* (R848), 11, 165, **170**  
 Control characters, **23**  
 control edit descriptor, **222**  
 control edit descriptors, 235  
 control information list, **186**  
 control mask, **427**  
*control-edit-desc* (R1011), 222, **223**  
 conversion  
   numeric, 141  
 current record, **175**  
 CYCLE statement, 164, 167  
*cycle-stmt* (R843), 11, 165–167, **167**

## D

*d* (R1008), 223, **223**, 228–231, 233, 234, 241  
 data, **427**  
 data edit descriptor, **222**  
 data edit descriptors, 226–234  
 data entity, **16**, **427**  
 data object, **16**, **427**  
 data object reference, **20**  
 data pointer, **18**  
 DATA statement, 87, 419  
 data transfer, 196  
 data transfer input statement, **171**  
 data transfer output statements, **171**  
 data transfer statements, 186  
 data type, *see* type, **427**  
*data-component-def-stmt* (R440), 50, **50**, 51  
*data-edit-desc* (R1005), 222, **223**  
*data-i-do-object* (R528), 87, **87**, 88  
*data-i-do-variable* (R529), 87, **87**, 88, 89, 409  
*data-implied-do* (R527), 87, **87**, 88, 89, 409  
*data-pointer-component-name*, 143  
*data-pointer-object* (R736), 143, **143**, 144, 151

*data-ref* (R612), 105, **105**, 106, 107, 189, 266, 268, 278, 279  
*data-stmt* (R524), 10, **87**, 259, 286, 411  
*data-stmt-constant* (R532), 37, 88, **88**, 89  
*data-stmt-object* (R526), 87, **87**, 88, 89  
*data-stmt-repeat* (R531), 88, **88**, 89  
*data-stmt-set* (R525), 87, **87**  
*data-stmt-value* (R530), 87, **88**, 89  
*data-target* (R739), 63–65, 73, 143, **143**, 144, 151, 272, 286, 304, 415  
 datum, **427**  
*dealloc-opt* (R636), 114, **114**, 116  
 DEALLOCATE statement, **114**  
*deallocate-stmt* (R635), 11, 74, 81, **114**, 423  
 decimal symbol, **226**, **427**  
*decimal-edit-desc* (R1018), 223, **224**  
**DECIMAL= specifier**, 183, 190, 212  
 declaration, **19**  
*declaration-construct* (R207), 9, **10**  
*declaration-type-spec* (R502), 41, 50, 71, **71**, 74, 75, 92, 94, 264, 265, 279, 282  
 declarations, 71–101  
 declared type, **75**, **427**  
 default character, **40**  
 default complex, **39**  
 default initialization, **428**  
 default integer, **36**  
 default logical, **44**  
 default real, **38**  
*default-char-expr* (R726), 123, **123**, 181–191  
*default-char-variable* (R607), 103, **103**, 110, 181, 210–216  
 default-initialized, **53**, **428**  
*default-logical-variable* (R605), 103, **103**, 210, 212, 213  
 deferred binding, **57**, **428**  
 deferred type parameter, **35**, **428**  
 deferred-shape array, **79**  
*deferred-shape-spec* (R515), 50, 72, 78, 79, **79**, 86, 91  
 definable, **428**  
 defined, **20**, **428**  
 defined assignment, 263  
 defined assignment statement, **142**, **428**  
 defined binary operation, **122**  
 defined elemental assignment statement, **142**  
 defined elemental operation, **123**  
 defined operation, **122**, 262, **428**  
 defined unary operation, **122**  
*defined-binary-op* (R723), 26, 120, **120**, 122, 123, 252  
*defined-operator* (R311), 26, 56, 252, 258, 259  
*defined-unary-op* (R703), 26, 118, **118**, 122, 123, 136, 252

- definition, **19**
  - definition of variables, **419**
  - deleted feature, **428**
  - deleted features, **7**
  - DELIM= specifier**, **183, 190, 212**
  - Delimiters, **27**
  - derived type, **16, 428**
  - derived type determination, **47**
  - derived types, **44–65**
  - derived-type intrinsic assignment statement, **139**
  - derived-type type specifier, **75**
  - derived-type-def* (R429), **10, 45, 45, 48, 49, 75**
  - derived-type-spec* (R455), **33, 63, 63, 71, 199, 408**
  - derived-type-stmt* (R430), **45, 45, 48, 411**
  - designator, **19, 428**
  - designator* (R603), **88, 103, 103, 106, 117**
  - designator*, **117**
  - digit*, **5, 23, 24, 26, 36, 37, 170, 240**
  - digit-string* (R409), **36, 36, 38, 227, 228**
  - digit-string*, **36**
  - digits, **24**
  - DIMENSION attribute, **78, 90**
  - DIMENSION statement, **90**
  - dimension-stmt* (R535), **10, 90, 411**
  - direct access, **173**
  - direct access input/output statement, **191**
  - DIRECT= specifier**, **212**
  - disassociated, **18, 428**
  - distinguishable, **407**
  - DO construct, **164**
  - DO statement, **164**
  - DO termination, **166**
  - DO WHILE statement, **164**
  - do-block* (R832), **165, 165, 166, 167**
  - do-body* (R837), **165, 165, 166**
  - do-construct* (R825), **11, 165, 167, 168**
  - do-construct-name*, **165, 167, 168**
  - do-stmt* (R827), **165, 165, 169, 423**
  - do-term-action-stmt* (R838), **165, 165, 166, 167, 169**
  - do-term-shared-stmt* (R842), **166, 166, 167–169**
  - do-variable* (R831), **165, 165, 166, 191, 192, 217–219, 240, 420, 422, 423, 463**
  - DOUBLE PRECISION**, **38**
  - double precision real, **38**
  - dtio-generic-spec* (R1208), **56, 62, 198–200, 205, 258, 258, 259, 262, 407**
  - dtv-type-spec* (R920), **199**
  - dummy argument, **255, 428**
  - dummy arguments
    - restrictions, **273**
  - dummy array, **428**
  - dummy data object, **428**
  - dummy pointer, **428**
  - dummy procedure, **256, 428**
  - dummy-arg* (R1233), **282, 282, 283**
  - dummy-arg-name* (R1226), **90, 92, 279, 279, 282, 285, 411**
  - dynamic type, **75, 428**
- ## E
- e* (R1009), **223, 223, 229–231, 233, 241**
  - edit descriptor, **222**
  - edit descriptors, *see* format descriptors
  - effective item, **193, 429**
  - effective position, **408**
  - element array assignment (FORALL), **148**
  - element sequence, **272**
  - elemental, **18, 255, 429**
  - elemental intrinsic function, **291**
  - elemental operation, **125**
  - elemental procedure, **287**
  - else-if-stmt* (R804), **156, 156**
  - else-stmt* (R805), **156, 156**
  - elsewhere-stmt* (R750), **146, 146**
  - ENCODING= specifier**, **183, 212**
  - END statement, **14, 14**
  - end-associate-stmt* (R820), **160, 161, 161, 169**
  - end-block-data-stmt* (R1118), **9, 14, 15, 253, 253**
  - end-do* (R833), **165, 165, 166–168**
  - end-do-stmt* (R834), **165, 165, 169**
  - end-enum-stmt* (R464), **66, 66**
  - end-forall-stmt* (R758), **148, 148**
  - end-function-stmt* (R1230), **9, 11, 14, 157, 165, 166, 258, 279, 280, 280, 284**
  - end-if-stmt* (R806), **156, 156, 169**
  - end-interface-stmt* (R1204), **258, 258, 259**
  - end-module-stmt* (R1106), **9, 14, 15, 250, 250**
  - end-of-file condition, **216**
  - end-of-record condition, **216**
  - end-program-stmt* (R1103), **9, 11, 14, 15, 157, 165, 166, 249, 249**
  - end-select-stmt* (R811), **158, 158, 159, 169**
  - end-select-type-stmt* (R824), **162, 162, 163, 169**
  - end-subroutine-stmt* (R1234), **9, 11, 14, 157, 165, 166, 258, 282, 282, 284**
  - end-type-stmt* (R433), **45, 45**
  - end-where-stmt* (R751), **146, 146**
  - END= specifier**, **217**
  - endfile record, **172**
  - ENDFILE statement, **172, 208**
  - endfile-stmt* (R924), **11, 207, 287**
  - ending point, **104**
  - entity, **429**
  - entity-decl* (R504), **41, 71, 72, 72, 73, 74, 78, 126, 127, 411**
  - entity-name*, **87, 91**
  - ENTRY statement, **283, 283**

*entry-name*, 279, 283, 407  
*entry-stmt* (R1235), 10, 250, 259, 283, **283**, 407, 411  
*enum-def* (R460), 10, 66, **66**, 67  
*enum-def-stmt* (R461), 66, **66**  
enumeration, 66  
enumerator, 66  
*enumerator* (R463), 66, **66**  
*enumerator-def-stmt* (R462), 66, **66**  
**EOR= specifier**, 218  
*equiv-op* (R721), 26, 120, **120**  
*equiv-operand* (R716), 120, **120**  
EQUIVALENCE statement, **95**, 95–98  
*equivalence-object* (R556), 96, **96**, 253  
*equivalence-set* (R555), 96, **96**, 97  
*equivalence-stmt* (R554), 10, **96**, 411  
**ERR= specifier**, 217  
*errmsg-variable* (R626), 110, **110**, 111, 113, 114, 421, 423  
**ERRMSG= specifier**, 113  
ERROR\_UNIT, 179, 360  
evaluation

- optional, 129
- operations, 128
- parentheses, 129

executable construct, **429**  
executable constructs, 155  
executable statement, **13**, **429**  
*executable-construct* (R213), 10, **10**, 13, 283  
execution control, 155–170  
execution cycle, **167**  
*execution-part* (R208), 9, **10**, 11, 249, 279, 280, 282  
*execution-part-construct* (R209), 10, **10**, 155, 165  
exist, **173**  
**EXIST= specifier**, 212  
EXIT statement, 168  
*exit-stmt* (R844), 11, 165, 166, 168, **168**  
explicit, **257**  
explicit formatting, 221–238  
explicit initialization, **74**, 87, **429**  
explicit interface, **257**, **429**  
explicit-shape array, **78**, **429**  
*explicit-shape-spec* (R511), 50, 72, 78, **78**, 80, 98, 99  
*exponent* (R420), 38, **38**  
*exponent-letter* (R419), 38, **38**  
*expr* (R722), 6, 59, 63, 64, 67, 110, 117, 118, 120, **120**, 123, 124, 127, 138–144, 147, 148, 151, 160, 191, 267, 285, 286  
expression, **117**, **429**  
expressions, **17**, 117–137  
extended type, **60**, **429**  
*extended-intrinsic-op* (R312), 26, **26**  
extensible type, **60**, **429**

extension operation, **123**, 136  
extension operator, **123**  
extension type, **60**, **429**  
extent, **18**, **429**  
EXTERNAL attribute, **80**  
external file, **172**, **429**  
external linkage, **429**  
external procedure, **12**, **255**, **429**  
EXTERNAL statement, **263**  
external subprogram, **11**, **429**  
external unit, **178**, **429**  
*external-name*, 263, 264  
*external-stmt* (R1210), 10, **263**, 411  
*external-subprogram* (R203), 9, **9**, 283

## F

field, **224**  
field width, **224**  
file, **430**  
file access, 173  
file connection, 178  
file connection statements, **171**  
file inquiry, 209  
file inquiry statement, **171**  
file position, 175  
file positioning statements, **171**, 207  
file storage unit, **177**, **430**  
*file-name-expr* (R906), 181, **181**, 183, 210, 211, 213  
*file-unit-number* (R902), 178, **178**, 181, 185, 187, 201, 206, 207, 209–211, 287  
**FILE= specifier**, 183, 211  
files

- connected, 179
- external, 172
- internal, 177
- preconnected, 180

final subroutine, **430**  
final subroutines, **58**  
*final-binding* (R454), 56, **58**  
*final-subroutine-name*, 58  
finalizable, **58**, **430**  
finalization, 59, **430**  
finalized, **59**  
fixed source form, 29, **29**  
FLUSH statement, 209  
*flush-spec* (R928), 209, **209**  
*flush-stmt* (R927), 11, **209**, 287  
**FMT= specifier**, 188  
FORALL construct, 148  
*forall-assignment-stmt* (R757), 148, **148**, 151, 153, 287  
*forall-body-construct* (R756), 148, **148**, 149, 151, 153  
*forall-construct* (R752), 11, 148, **148**, 149, 150, 152

*forall-construct-name*, 148, 149  
*forall-construct-stmt* (R753), 148, 148, 149, 169  
*forall-header* (R754), 148, 148, 153, 154  
*forall-stmt* (R759), 11, 148, 152, 153, 153  
*forall-triplet-spec* (R755), 148, 148, 149–152  
**FORM= specifier**, 183, 212  
*format* (R914), 186–188, 188, 196, 221, 222  
 format control, 224  
 format descriptors  
   /, 236  
   :, 236  
   A, 232  
   B, 227  
   BN, 237  
   BZ, 237  
   control edit descriptors, 235  
   D, 229  
   data edit descriptors, 226–234  
   E, 229  
   EN, 230  
   ES, 230  
   F, 228  
   G, 232, 233  
   I, 227  
   L, 232  
   O, 227  
   P, 237  
   S, 237  
   SP, 237  
   SS, 237  
   TL, 235  
   TR, 235  
   X, 236  
   Z, 227  
 FORMAT statement, 188, 221, 533  
*format-item* (R1003), 221, 222, 222  
*format-specification* (R1002), 221, 221  
*format-stmt* (R1001), 10, 221, 221, 250, 259  
 formatted data transfer, 197  
 formatted input/output statement, 188  
 formatted record, 171  
**FORMATTED= specifier**, 212  
 formatting  
   explicit, 221–238  
   list-directed, 198, 238–242  
   namelist, 198, 242–247  
 forms, 172  
 FORTRAN 77 compatibility, 4  
 Fortran 90 compatibility, 3  
 Fortran 95 compatibility, 3  
 Fortran character set, 23  
 free source form, 27, 27  
 function, 12, 430  
 function reference, 17, 276

function result, 430  
 FUNCTION statement, 279  
 function subprogram, 279, 430  
*function-name*, 72, 73, 259, 279, 280, 283, 285, 407, 411  
*function-reference* (R1217), 64, 72, 117, 266, 268, 276  
*function-stmt* (R1224), 9, 258, 259, 279, 279, 280, 407, 411  
*function-subprogram* (R1223), 9, 10, 250, 279

## G

generic identifier, 261, 430  
 generic interface, 57, 261, 430  
 generic interface block, 259, 430  
 generic name, 261  
 Generic names, 291  
 generic procedure references, 407  
*generic-binding* (R452), 56, 56, 57, 58, 251, 252  
*generic-name*, 56, 57, 258, 408, 411  
*generic-spec* (R1207), 56–58, 62, 86, 122, 142, 251, 252, 258, 258, 259, 261, 408, 411  
 global entities, 405  
 global entity, 430  
 global identifier, 405  
 GO TO statement, 169  
*goto-stmt* (R845), 11, 165, 166, 169, 169  
 Graphic characters, 23

## H

*hex-constant* (R414), 37, 37  
*hex-digit* (R415), 37, 37  
 host, 12, 430  
 host association, 411, 430  
 host scoping unit, 12, 430

## I

ICHAR intrinsic, 43  
**ID= specifier**, 190, 213  
 IEEE\_, 300, 363–389  
 IF construct, 156, 156  
 IF statement, 156, 157  
*if-construct* (R802), 11, 156, 156  
*if-construct-name*, 156  
*if-stmt* (R807), 11, 157, 157  
*if-then-stmt* (R803), 156, 156, 169  
*imag-part* (R423), 39, 39  
 imaginary part, 39  
 implicit, 257  
 implicit interface, 266, 430  
 IMPLICIT statement, 92, 92  
*implicit-part* (R205), 9, 10  
*implicit-part-stmt* (R206), 10, 10  
*implicit-spec* (R550), 92, 92

- implicit-stmt* (R549), 10, **92**
- import-name*, 258, 259
- import-name-list*, 260
- import-stmt* (R1209), 9, **258**
- inactive, **166**
- INCLUDE**, 30
- INCLUDE line, **30**
- index-name*, 148–154, 409, 410, 421, 423
- inherit, **430**
- inheritance associated, **61**
- inheritance association, 418, **430**
- inherited, **60**
- initial point, **175**
- initialization, 53, 73, 74, 419, 516
- initialization* (R506), 72, **72**, 73, 74
- initialization expression, **126**
- initialization-expr* (R730), 50, 53, 72, 74, 83, 90, 127, **127**
- inner-shared-do-construct* (R841), 165, **165**, 166
- Input statements, **171**
- input-item* (R915), 186, 187, 191, **191**, 192, 205, 219, 423
- input/output editing, 221–247
- input/output list, 191
- input/output statements, 171–219
- INPUT\_UNIT, 179, 360
- inquire by file, **209**
- inquire by output list, **209**
- inquire by unit, **209**
- INQUIRE statement, 209
- inquire-spec* (R930), 210, **210**, 211, 216, 219
- inquire-stmt* (R929), 11, **210**, 287
- inquiry function, **291**, **430**
- inquiry, type parameter, 106
- instance, **282**
- instance of a subprogram, **430**
- int-constant* (R308), 25, **25**, 88
- int-constant-name*, 36
- int-constant-subobject* (R533), 88, **88**
- int-expr* (R727), 15, 34, 67, 87, 88, 104, 107, 111, 123, **123**, 125–128, 148, 158, 165, 166, 170, 178, 181, 187, 191, 192, 194, 284
- int-initialization-expr* (R732), 36, 40, 41, 48, 49, 66, 67, 127, **127**, 158
- int-literal-constant* (R406), 25, 36, **36**, 41, 89, 222, 223, 323
- int-variable* (R608), 67, 87, 103, **103**, 110, 165, 181, 185, 187, 206, 207, 209, 210, 213–218
- INTEGER**, 36
- integer constant, 36
- integer editing, 227
- integer model, 293
- integer type, **36**, **36**
- INTENT**, 292
- intent, **430**
- INTENT attribute, **81**, 90
- INTENT statement, 90
- intent-spec* (R517), 71, **81**, 90, 264
- intent-stmt* (R536), 10, **90**
- interface, **257**
  - (procedure), 257
  - explicit, 257
  - generic, 261
  - implicit, 266
- interface block, **13**, **430**
- interface body, **13**, **431**
- interface of a procedure, **431**
- interface-block* (R1201), 10, 258, **258**
- interface-body* (R1205), 258, **258**, 259, 411
- interface-name* (R1215), 264, **264**, 265
- interface-name*, 56, 57
- interface-specification* (R1202), 258, **258**
- interface-stmt* (R1203), 258, **258**, 259, 261, 262, 411
- internal file, **431**
- internal files, 177
- internal procedure, **12**, **255**, **431**
- internal subprogram, **11**, **431**
- internal unit, **178**
- internal-file-variable* (R903), 178, **178**, 187, 423
- internal-subprogram* (R211), 10, **10**
- internal-subprogram-part* (R210), 9, **10**, 249, 279, 280, 282, 286
- interoperable, 395, **396**, **398–400**, 401, **431**
- intrinsic, **20**, **431**
  - elemental, 291
  - inquiry function, 291
  - transformational, 291
- intrinsic assignment statement, **138**
- INTRINSIC attribute, **82**
- intrinsic binary operation, **121**
- intrinsic module, **250**
- intrinsic operation, **121**
- intrinsic operations, 132–136
  - logical, 44
- intrinsic procedure, **255**
- intrinsic procedures, 300, 359
- INTRINSIC statement, **266**
- intrinsic type, **15**
- intrinsic types, 35–44
- intrinsic unary operation, **121**
- intrinsic-operator* (R310), **25**, 26, 118, 120–122, 262
- intrinsic-procedure-name*, 266, 411
- intrinsic-stmt* (R1216), 10, **266**, 411
- intrinsic-type-spec* (R403), 33, **36**, 41, 71, 75
- invoke, **431**
- io-control-spec* (R913), 186, **186**, 187, 188, 201, 219

*io-implied-do* (R917), 191, **191**, 192, 193, 197, 219, 420, 422, 423, 463  
*io-implied-do-control* (R919), 191, **191**, 194  
*io-implied-do-object* (R918), 191, **191**, 192  
*io-unit* (R901), 178, **178**, 186, 187, 287  
*iomsg-variable* (R907), 181, **181**, 185, 187, 206, 207, 209, 210, 217–219, 421  
**IOMSG= specifier**, 219  
**IOSTAT= specifier**, 218  
ISO 10646, 349  
ISO 10646 character set, **40**  
ISO 10646 character type, **40**  
ISO\_C\_BINDING module, 391  
ISO\_FORTRAN\_ENV module, 178, 179, 360  
iteration count, **167**

## K

*k* (R1012), 223, **223**, 229, 230, 234, 237  
keyword, **19**, 268, **431**  
*keyword* (R215), 19, **19**, 63, 267  
kind, **36**, **37**, **39**, **40**, **44**  
KIND intrinsic, 36, 37, 39, 40, 44  
kind type parameter, **16**, 34, 36, 37, 39, 40, 44, **431**  
*kind-param* (R407), 36, **36**, 38, 41, 42, 44, 89, 323  
*kind-selector* (R404), 6, 36, **36**, 48

## L

label, **431**  
*label* (R313), 26, **26**, 165, 169, 170, 181, 185–188, 206, 207, 209, 210, 217, 218, 267  
*label-do-stmt* (R828), 165, **165**, 166  
*language-binding-spec* (R509), 71, 73, **77**, 87, 279  
left tab limit, **235**  
*left-square-bracket* (R467), 67, **67**  
length, **40**  
length of a character string, **431**  
length type parameter, 34  
*length-selector* (R425), 6, 40, **40**, 41  
*letter*, 23, **23**, 25, 26, 92, 118, 120  
*letter-spec* (R551), 92, **92**, 93  
letters, **23**  
*level-1-expr* (R702), 118, **118**, 137  
*level-2-expr* (R706), 118, **118**, 119, 137  
*level-3-expr* (R710), 119, **119**  
*level-4-expr* (R712), 119, **119**, 120  
*level-5-expr* (R717), 120, **120**  
lexical token, **431**  
Lexical tokens, **25**  
line, **27**, **431**  
linked, **431**  
list-directed formatting, 198, 238–242  
list-directed input/output statement, **188**  
literal constant, **17**, 103, **431**  
*literal-constant* (R306), 25, **25**

local entity, **431**  
local identifier, **405**  
local identifiers, 406  
local variable, **17**, **431**  
*local-defined-operator* (R1114), 251, 252, **252**  
*local-name*, 251–253  
**LOGICAL**, 44  
logical intrinsic assignment statement, **139**  
logical intrinsic operation, **121**  
logical intrinsic operations, 44, 135  
logical intrinsic operator, **121**  
logical type, 43, **43**  
*logical-expr* (R724), 123, **123**, 127, 146, 156–158, 165, 167, 168  
*logical-initialization-expr* (R733), 127, **127**, 158  
*logical-literal-constant* (R428), 25, **44**, 118, 120  
*logical-variable* (R604), 103, **103**  
loop, **164**  
*loop-control* (R830), 165, **165**, 166–168  
low-level syntax, **25**  
*lower-bound* (R512), 78, **78**, 79, 80  
*lower-bound-expr* (R631), 111, **111**, 143

## M

*m* (R1007), 223, **223**, 227  
main program, 12, **249**, **432**  
*main-program* (R1101), 9, 249, **249**  
many-one array section, **109**, **432**  
*mask-expr* (R748), 146, **146**, 147–153  
masked array assignment, **146**  
masked array assignment (WHERE), 145  
*masked-elsewhere-stmt* (R749), 146, **146**, 147, 152  
model  
    bit, 292  
    integer, 293  
    real, 293  
module, **13**, **250**, **432**  
*module* (R1104), 9, **250**  
module procedure, **12**, **256**, **432**  
module reference, **20**, **251**  
module subprogram, **11**, **432**  
*module-name*, 250–252, 411  
*module-nature* (R1110), 251, **251**, 252  
*module-stmt* (R1105), 9, 250, **250**  
*module-subprogram* (R1108), 10, 250, **250**, 283  
*module-subprogram-part* (R1107), 9, 57, 62, 250, **250**  
*mult-op* (R708), 25, 118, **118**  
*mult-operand* (R704), 118, **118**, 137

## N

*n* (R1014), 223, **223**  
name, **19**, **432**

- name* (R304), 6, 19, 25, **25**, 72, 91, 103, 162, 196, 264, 279
- name association, **410**, **432**
- name-value subsequences, **242**
- NAME= specifier**, 213
- named, **432**
- named common block, **98**
- named constant, **17**, 83, 90, 103, **432**
- named file, **172**
- named-constant* (R307), 25, **25**, 30, 39, 66, 90, 411
- named-constant-def* (R539), 90, **90**, 411
- NAMED= specifier**, 213
- namelist formatting, 198, 242–247
- namelist input/output statement, **189**
- NAMELIST** statement, 95, **95**
- namelist-group-name*, 95, 186–189, 195–197, 221, 243, 247, 253, 411, 423
- namelist-group-object* (R553), 95, **95**, 196, 198, 205, 219, 242, 243, 253
- namelist-stmt* (R552), 10, **95**, 411, 423
- Names, **25**
- NaN, 300, 368, **432**
- next record, **175**
- NEXTREC= specifier**, 213
- NML= specifier**, 189
- nonadvancing input/output statement, **175**
- nonblock-do-construct* (R835), 165, **165**
- NONE (DELIM value)**, 247
- nonexecutable statement, **432**
- Nonexecutable statements, **13**
- nonintrinsic module, **250**
- nonlabel-do-stmt* (R829), 165, **165**
- normal, **368**
- not-op* (R718), 26, 120, **120**
- null-init* (R507), 50, 53, 72, **72**, 74, 88, 89, 264, 265
- NULLIFY** statement, **113**
- nullify-stmt* (R633), 11, 74, 81, **113**, 423
- NUMBER= specifier**, 213
- numeric conversion, 141
- numeric editing, 226
- numeric intrinsic assignment statement, **139**
- numeric intrinsic operation, **121**
- numeric intrinsic operations, 133
- numeric intrinsic operator, **121**
- numeric relational intrinsic operation, **121**
- numeric sequence type, 46, 96, 417, 520
- numeric storage unit, **416**, **432**
- numeric type, **432**
- numeric types, **35**
- numeric-expr* (R728), 38, 123, **123**, 170
- O**
- object, *see* data object, **16**, **432**
- object designator, **19**, **432**
- object-name* (R505), 72, **72**, 73, 74, 83, 86, 91, 92, 103, 411
- obsolescent feature, **432**
- obsolescent features, **7**
- octal-constant* (R413), 37, **37**
- only* (R1112), 251, **251**, 252
- only-use-name* (R1113), 251, **251**, 252
- OPEN** statement, 180, **180**
- open-stmt* (R904), 11, **181**, 287
- OPENED= specifier**, 213
- operand, **432**
- operands, **20**
- operation, **432**
- operations, 34
- character intrinsic, 134
- logical intrinsic, 135
- numeric intrinsic, 133
- relational intrinsic, 134
- operator, **20**, **432**
- operator precedence, 136
- operators, 25
- OPTIONAL** attribute, **83**, 90
- optional dummy argument, 272
- OPTIONAL** statement, 90
- optional-stmt* (R537), 10, **90**
- or-op* (R720), 26, 120, **120**
- or-operand* (R715), 120, **120**
- outer-shared-do-construct* (R839), 165, **165**, 166
- Output statements, **171**
- output-item* (R916), 186, 187, 191, **191**, 192, 205, 210
- OUTPUT\_UNIT**, 179, 360
- override, 61, **432**
- overrides, **53**
- P**
- PAD= specifier**, 183, 190, 213
- PARAMETER**, 17
- PARAMETER** attribute, **83**, 90
- PARAMETER** statement, 90, **90**
- parameter-stmt* (R538), 10, **90**, 411
- parent component, **61**, **432**
- parent data transfer statement, **199**, 198–203, 219
- parent type, **60**, **432**
- parent-string* (R610), 104, **104**
- parent-type-name*, 45
- parentheses, 129
- part-name*, 105, 107
- part-ref* (R613), 88, 96, 105, **105**, 107, 108
- partially [storage] associated, **417**
- PASS** attribute, 268
- passed-object dummy argument, **52**, **433**
- PENDING= specifier**, 214
- pointer, **18**, **433**

- pointer assignment, **143, 433**
  - pointer assignment statement, **433**
  - pointer associated, **433**
  - pointer association, **414, 433**
  - pointer association status, **414**
  - POINTER attribute, **83, 91**
  - POINTER statement, **91**
  - pointer-assignment-stmt* (R735), **11, 74, 81, 143, 148, 151, 286, 423**
  - pointer-decl* (R541), **91, 91**
  - pointer-object* (R634), **74, 81, 113, 113, 114, 304, 423**
  - pointer-stmt* (R540), **10, 91, 411**
  - polymorphic, **75, 433**
  - POS= specifier**, **190, 214**
  - position, **172**
  - position-edit-desc* (R1013), **223, 223**
  - position-spec* (R926), **207, 207**
  - POSITION= specifier**, **184, 214**
  - positional arguments, **291**
  - power-op* (R707), **25, 118, 118**
  - pre-existing, **418**
  - precedence of operators, **136**
  - preceding record, **175**
  - PRECISION intrinsic, **37**
  - preconnected, **433**
  - preconnected files, **180**
  - Preconnection, **180**
  - prefix* (R1227), **279, 279, 280, 282**
  - prefix-spec* (R1228), **279, 279, 280–282, 286, 287**
  - present, **272**
  - present (dummy argument), **272**
  - PRESENT intrinsic, **83**
  - primaries*, **285**
  - primary, **117**
  - primary* (R701), **117, 117, 118**
  - PRINT statement, **186**
  - print-stmt* (R912), **11, 186, 287**
  - PRIVATE attribute, **46, 76**
  - PRIVATE statement, **86**
  - private-components-stmt* (R447), **45, 55, 55**
  - private-or-sequence* (R432), **45, 45**
  - proc-attr-spec* (R1213), **264, 264, 265**
  - proc-binding-stmt* (R450), **56, 56, 57**
  - proc-component-attr-spec* (R446), **51, 51**
  - proc-component-def-stmt* (R445), **50, 51, 51**
  - proc-component-ref* (R741), **143, 143, 144, 266, 267**
  - proc-decl* (R1214), **51, 264, 264, 265**
  - proc-entity-name*, **91**
  - proc-interface* (R1212), **51, 264, 264, 265**
  - proc-language-binding-spec* (R1225), **73, 264, 265, 279, 279, 280, 282, 285, 400**
  - proc-pointer-name* (R545), **91, 91, 98, 114, 143, 411**
  - proc-pointer-object* (R740), **143, 143, 144, 145, 151**
  - proc-target* (R742), **63–65, 73, 143, 144, 144, 145, 151, 272, 304, 415**
  - procedure, **12, 433**
    - characteristics of, **256**
    - dummy, **256**
    - elemental, **287**
    - external, **255**
    - internal, **255**
    - intrinsic, **291–359**
    - non-Fortran, **285**
    - pure, **286**
  - procedure designator, **19, 433**
  - procedure interface, **257, 433**
  - procedure pointer, **18, 264**
  - procedure reference, **20, 266**
  - procedure references
    - generic, **407**
    - resolving, **276**
  - procedure-component-name*, **143**
  - procedure-declaration-stmt* (R1211), **10, 80, 91, 264, 264, 265, 411**
  - procedure-designator* (R1219), **266, 266, 278**
  - procedure-entity-name*, **264**
  - procedure-name*, **56, 57, 144, 145, 258, 259, 266, 267**
  - procedure-stmt* (R1206), **258, 258, 259**
  - processor, **1, 433**
  - processor dependent, **3, 433**
  - program, **12, 433**
  - program* (R201), **9, 9**
  - program unit, **11, 433**
  - program-name*, **249**
  - program-stmt* (R1102), **9, 249, 249**
  - program-unit* (R202), **6, 9, 9**
  - PROTECTED attribute, **84, 91**
  - PROTECTED statement, **91, 91**
  - protected-stmt* (R542), **10, 91**
  - prototype, **433**
  - PUBLIC attribute, **46, 76**
  - PUBLIC statement, **86**
  - pure procedure, **286, 433**
- ## Q
- QUOTE (DELIM value), **247**
- ## R
- r* (R1004), **222, 222, 223, 224**
  - range, **166**
  - RANGE intrinsic, **36, 37**
  - rank, **17, 18, 433**
  - READ statement, **186**
  - read-stmt* (R910), **11, 186, 187, 287, 423**
  - READ= specifier**, **214**
  - reading, **171**
  - READWRITE= specifier**, **215**

**REAL**, 38

real and complex editing, 228

real model, 293

real part, 39

real type, 37, 37–39

*real-literal-constant* (R417), 25, 38, 38*real-part* (R422), 39, 39**REC= specifier**, 191**RECL= specifier**, 184, 215

record, 171, 433

record file, 171

record lengths, 172

record number, 174

**RECURSIVE**, 282

recursive input/output statement, 219

reference, 433

*rel-op* (R713), 26, 119, 119, 135

relational intrinsic operation, 121

relational intrinsic operations, 134

relational intrinsic operator, 121

*rename* (R1111), 251, 251, 252, 406*rep-char*, 42, 42, 224, 239, 244

repeat specification., 222

representable character, 42

representation method, 37

representation methods, 36, 40, 44

resolving procedure references, 276

derived-type input/output, 205

restricted expression, 125

result variable, 12, 434

*result-name*, 279, 280, 283, 411

RETURN statement, 284

*return-stmt* (R1236), 11, 15, 165, 166, 284, 284

REWIND statement, 208

*rewind-stmt* (R925), 11, 207, 287*right-square-bracket* (R468), 67, 67*round-edit-desc* (R1017), 223, 224**ROUND= specifier**, 184, 191, 215

rounding mode, 434

**S**

SAVE attribute, 84, 87, 91

SAVE statement, 91

*save-stmt* (R543), 10, 91, 411

saved, 84

*saved-entity* (R544), 74, 91, 91

scalar, 17, 104, 434

*scalar-xyz* (R103), 6, 6

scale factor, 223

scope, 405, 434

scoping unit, 11, 434

section subscript, 434

*section-subscript* (R619), 105, 107, 107, 108

SELECT CASE statement, 158

SELECT TYPE construct, 162

**SELECT TYPE construct**, 410, 413*select-case-stmt* (R809), 158, 158, 169*select-construct-name*, 162*select-type-construct* (R821), 11, 162, 162*select-type-stmt* (R822), 162, 162, 169

SELECTED\_INT\_KIND intrinsic, 36

SELECTED\_REAL\_KIND intrinsic, 37

selector, 434

*selector* (R819), 160, 160, 161–163, 273, 413, 424

sequence, 20

sequence association, 272

SEQUENCE property, 47

SEQUENCE statement, 46

sequence structure, 75

sequence type, 46

*sequence-stmt* (R434), 45, 46

sequential access, 173

sequential access input/output statement, 191

**SEQUENTIAL= specifier**, 215

shape, 18, 434

shape conformance, 125

*shared-term-do-construct* (R840), 165, 165, 166*sign* (R410), 36, 36, 38, 228*sign-edit-desc* (R1015), 223, 223**SIGN= specifier**, 184, 191, 215*signed-digit-string* (R408), 36, 38, 227, 228*signed-int-literal-constant* (R405), 36, 36, 39, 88, 223*signed-real-literal-constant* (R416), 38, 39, 88*significand* (R418), 38, 38

size, 18, 434

size of a common block, 99

size of a storage sequence, 416

**SIZE= specifier**, 191, 215

source forms, 27

*source-expr* (R627), 110, 110, 111, 112

special characters, 24

*special-character*, 23, 24

specific interface, 259

specific interface block, 259

Specific names, 291

*specific-binding* (R451), 56, 56

specification expression, 125, 434

specification function, 126, 434

specification inquiry, 125

*specification-expr* (R729), 71, 72, 74, 78, 125, 288*specification-part* (R204), 9, 9, 56, 76, 86, 90, 126, 127, 249, 250, 253, 254, 258, 279, 282, 286*specification-stmt* (R212), 10, 10

specifications, 71–101

standard-conforming program, 2, 434

starting point, 104

*stat-variable* (R625), 110, 110, 111, 112, 114, 423

- statement, **27, 434**
- statement entity, **405, 434**
- statement function, **256, 285, 434**
- Statement functions, **438**
- statement label, **26, 26, 434**
- statement order, **13**
- statements
  - accessibility, **86**
  - ALLOCATABLE, **86**
  - ALLOCATE, **110**
  - arithmetic IF, **170**
  - assignment, **138**
  - ASYNCHRONOUS, **86**
  - attribute specification, **85–100**
  - BACKSPACE, **207**
  - BIND, **87**
  - CALL, **266**
  - CASE, **158**
  - CLOSE, **185**
  - COMMON, **98–101**
  - computed GO TO, **170**
  - CONTAINS, **284**
  - CONTINUE, **170**
  - CYCLE, **167**
  - DATA, **87**
  - data transfer, **186**
  - DEALLOCATE, **114**
  - DIMENSION, **90**
  - DO, **164**
  - DO WHILE, **164**
  - END, **14**
  - ENDFILE, **208**
  - ENTRY, **283**
  - EQUIVALENCE, **95–98**
  - EXIT, **168**
  - EXTERNAL, **263**
  - file positioning, **207**
  - FLUSH, **209**
  - FORALL, **148, 153**
  - FORMAT, **221**
  - formatted input/output, **188**
  - FUNCTION, **279**
  - GO TO, **169**
  - IF, **157**
  - IMPLICIT, **92**
  - input/output, **171–219**
  - INQUIRE, **209**
  - INTENT, **90**
  - INTRINSIC, **266**
  - list-directed input/output, **188**
  - NAMELIST, **95**
  - namelist input/output, **189**
  - NULLIFY, **113**
  - OPEN, **180**
  - OPTIONAL, **90**
  - PARAMETER, **90**
  - POINTER, **91**
  - PRINT, **186**
  - PRIVATE, **86**
  - PROTECTED, **91**
  - PUBLIC, **86**
  - READ, **186**
  - RETURN, **284**
  - REWIND, **208**
  - SAVE, **91**
  - SELECT CASE, **158**
  - STOP, **170**
  - TARGET, **92**
  - type declaration, **71–85**
  - unformatted input/output, **188**
  - VALUE, **92**
  - VOLATILE, **92**
  - WHERE, **146**
  - WRITE, **186**
- STATUS= specifier, 184, 186**
- stmt-function-stmt* (R1238), **10, 250, 259, 285, 411**
- STOP statement, **170**
- stop-code* (R850), **170, 170**
- stop-stmt* (R849), **11, 165, 166, 170, 287**
- storage associated, **416**
- Storage association, **415**
- storage association, **95–101, 415, 435**
- storage sequence, **99, 416, 435**
- storage unit, **416, 435**
- stream access, **174**
- stream access input/output statement, **191**
- stream file, **171**
- STREAM= specifier, 216**
- stride, **109, 435**
- stride* (R621), **107, 107, 148, 149, 151, 152, 194**
- string, *see* character string
- struct, **435**
- structure, **16, 75, 435**
- structure component, **104, 435**
- structure constructor, **63, 435**
- structure-component* (R614), **87, 88, 103, 104, 105, 111, 114**
- structure-constructor* (R457), **63, 64, 88, 117, 286**
- subcomponent, **106, 435**
- subobject, **435**
- subobjects, **16**
- subprogram, **435**
- subroutine, **12, 435**
- subroutine reference, **276**
- subroutine subprogram, **282, 435**
- subroutine-name*, **259, 282, 407**
- subroutine-stmt* (R1232), **9, 258, 259, 279, 280, 282, 282, 407, 411**

*subroutine-subprogram* (R1231), 9, 10, 250, **282**  
 subscript, **435**  
*subscript* (R618), 6, 88, 107, **107**, 148, 149, 152, 194  
 subscript triplet, 109, **435**  
*subscript-triplet* (R620), 107, **107**, 108  
 substring, **104**, **435**  
*substring* (R609), 96, 103, **104**  
*substring-range* (R611), 104, **104**, 106, 107, 194  
*suffix* (R1229), 279, **280**, 283

## T

target, **18**, **435**  
 TARGET attribute, **84**, 92  
 TARGET statement, 92  
*target-stmt* (R546), 10, **92**, 411  
 terminal point, **175**  
 TKR compatible, **76**  
 totally [storage] associated, **417**  
 transfer of control, 156, 169, 217, 218  
 transformational function, **435**  
 transformational functions, **291**  
 type, **15**, 33–68, **435**  
   base, 60  
   character, 40–43  
   complex, 39  
   concept, **33**  
   declared, 75  
   derived types, 65  
   dynamic, 75  
   expression, 123  
   extended, 60  
   extensible, 60  
   extension, 60  
   integer, 36  
   intrinsic types, 35–44  
   logical, 43  
   operation, 124  
   parent, 60  
   primary, 123  
   real, 37–39  
 type compatible, **75**, **436**  
 type conformance, 139  
 type declaration statement, **436**  
 type declaration statements, 71–85  
 type equality, 47  
 type incompatible, **76**  
 type parameter, **15**, 34, 36, 37, **436**  
 type parameter inquiry, **106**  
 type parameter keyword, **19**  
 Type parameter order, **49**  
 type parameter order, **436**  
 type specifier, **33**  
   derived type, 75

TYPE, 75

TYPE type specifier, 75  
*type-attr-spec* (R431), 45, **45**, 95  
 type-bound procedure, **57**, **435**  
*type-bound-procedure-part* (R448), 45, 46, **56**, 58, 398  
*type-declaration-stmt* (R501), 10, 41, **71**, 72, 286, 411  
*type-guard-stmt* (R823), 162, **162**  
*type-name*, 45, 48, 56, 63, 436  
*type-param-attr-spec* (R437), 48, **48**  
*type-param-decl* (R436), 48, **48**, 49  
*type-param-def-stmt* (R435), 45, 48, **48**  
*type-param-inquiry* (R615), 106, **106**, 117, 408  
*type-param-name*, 45, 48, 50, 106, 117, 408, 411  
*type-param-spec* (R456), 63, **63**  
*type-param-value* (R402), 34, **34**, 35, 40, 41, 50, 63, 71, 72, 74, 111  
*type-spec* (R401), **33**, 41, 67, 68, 71, 110, 111, 162

## U

ultimate component, **436**  
 ultimate components, **44**  
 unallocated, **113**  
 undefined, **20**, **436**  
 undefinition of variables, 419  
*underscore* (R303), 23, **24**  
 unformatted data transfer, 197  
 unformatted input/output statement, **188**  
 unformatted record, **172**  
**UNFORMATTED=** specifier, 216  
 Unicode, **183**  
 unit, **178**  
 unlimited polymorphic, **75**  
 unsaved, **84**  
 unsigned, **436**  
 unspecified storage unit, **416**, **436**  
*upper-bound* (R513), 78, **78**, 79  
*upper-bound-expr* (R632), 111, **111**, 143  
 use associated, **251**  
 Use association, **410**  
 use association, 410, **436**  
 USE statement, **251**  
*use-defined-operator* (R1115), 251, 252, **252**  
*use-name*, 86, 251, 252, 406  
*use-stmt* (R1109), 9, **251**, 252, 411

## V

*v* (R1010), 202, 223, **223**, 235  
 value, **151**  
 VALUE attribute, **85**, 92  
 value separator, **239**  
 VALUE statement, 92  
*value-stmt* (R547), 10, **92**

variable, **436**  
*variable* (R601), 59, 64, 87, 88, 103, **103**, 116, 117,  
 138–143, 146–148, 151, 160, 162, 163, 191,  
 267, 286, 423, 424  
*variable-name* (R602), 95, 96, 98, 99, 103, **103**, 104,  
 111, 113, 143, 411, 423  
 variables, **17**  
   definition & undefinition, 419  
 vector subscript, **109**, **436**  
*vector-subscript* (R622), 107, **107**, 108  
 void, **436**  
 VOLATILE attribute, **85**, 92  
 VOLATILE statement, 92  
*volatile-stmt* (R548), 10, **92**

## W

*w* (R1006), 223, **223**, 227–233, 239, 241  
 wait operation, 185, 194, 205, 206  
**WAIT statement**, 206  
*wait-spec* (R922), 206, **206**  
*wait-stmt* (R921), 11, **206**, 287  
 WHERE construct, 146  
 WHERE statement, 146  
*where-assignment-stmt* (R747), 146, **146**, 147, 148,  
 152, 427  
*where-body-construct* (R746), 146, **146**, 147  
*where-construct* (R744), 11, 146, **146**, 148, 152  
*where-construct-name*, 146  
*where-construct-stmt* (R745), 146, **146**, 152, 169  
*where-stmt* (R743), 11, 146, **146**, 148, 152  
 whole array, **107**, **436**  
 WRITE statement, **186**  
*write-stmt* (R911), 11, **186**, 187, 287, 423  
**WRITE= specifier**, 216  
 writing, **171**

## X

*xyz-list* (R101), **6**  
*xyz-name* (R102), **6**

## Z

zero-size array, 18, 79, 89