INTERNATIONAL STANDARD

ISO/IEC 1539

Second edition
1991-07-01

Information technology — Programming languages — Fortran

Technologies de l'information — Langages de programmation — Fortran

Reference number
ISO/IEC 1539 : 1991 (E)

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Contents

Foreword xii

1. Overview 1

1.1 Scope 1
1.2 Processor 1
1.3 Inclusions and exclusions 1
1.3.1 Inclusions 1
1.3.2 Exclusions 1
1.4 Conformance 2
1.4.1 FORTRAN 77 compatibility 3
1.5 Notation used in this International Standard 3
1.5.1 Syntax rules 3
1.5.2 Assumed syntax rules 4
1.5.3 Syntax conventions and characteristics 5
1.5.4 Text conventions 5
1.6 Deleted and obsolescent features 5
1.6.1 Nature of deleted features 5
1.6.2 Nature of obsolescent features 5
1.7 Modules 6
1.8 Normative references 6

2. Fortran terms and concepts 7

2.1 High level syntax 7
2.2 Program unit concepts 9
2.2.1 Executable program 10
2.2.2 Main program 10
2.2.3 Procedure 10
2.2.4 Module 10
2.3 Execution concepts 11
2.3.1 Executable/nonexecutable statements 11
2.3.2 Statement order 11
2.3.3 The END statement 12
2.3.4 Execution sequence 12
2.4 Data concepts 13
2.4.1 Data type 13
2.4.2 Data value 13
2.4.3 Data entity 13
2.4.4 Scalar 14
2.4.5 Array 15
2.4.6 Pointer 15
2.4.7 Storage 15
2.5 Fundamental terms 15
2.5.1 Name and designator 15
2.5.2 Keyword 16

© ISO/IEC 1991
All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

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

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

2.5.3 Declaration 16
2.5.4 Definition 16
2.5.5 Reference 16
2.5.6 Association 16
2.5.7 Intrinsic 16
2.5.8 Operator 16
2.5.9 Sequence 17

3. Characters, lexical tokens, and source form 18

3.1 Processor character set 18
3.1.1 Letters 18
3.1.2 Digits 18
3.1.3 Underscore 18
3.1.4 Special characters 19
3.1.5 Other characters 19
3.2 Low-level syntax 19
3.2.1 Keywords 19
3.2.2 Names 19
3.2.3 Constants 20
3.2.4 Operators 20
3.2.5 Statement labels 21
3.2.6 Delimiters 21
3.3 Source form 21
3.3.1 Free source form 22
3.3.2 Fixed source form 23
3.4 Including source text 24

4. Intrinsic and derived data types 25

4.1 The concept of data type 25
4.1.1 Set of values 25
4.1.2 Constants 25
4.1.3 Operations 26
4.2 Relationship of types and values to objects 26
4.3 Intrinsic data types 26
4.3.1 Numeric types 26
4.3.2 Nonnumeric types 30
4.4 Derived types 32
4.4.1 Derived-type definition 32
4.4.2 Determination of derived types 35
4.4.3 Derived-type values 36
4.4.4 Construction of derived-type values 37
4.4.5 Derived-type operations and assignment 37
4.5 Construction of array values 37

5. Data object declarations and specifications 39

5.1 Type declaration statements 39
5.1.1 Type specifiers 41
5.1.2 Attributes 43
5.2 Attribute specification statements 48
5.2.1 INTENT statement 48

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

5.2.2 OPTIONAL statement 49
5.2.3 Accessibility statements 49
5.2.4 SAVE statement 50
5.2.5 DIMENSION statement 50
5.2.6 ALLOCATABLE statement 50
5.2.7 POINTER statement 51
5.2.8 TARGET statement 51
5.2.9 DATA statement 51
5.2.10 PARAMETER statement 53
5.3 IMPLICIT statement 54
5.4 NAMELIST statement 56
5.5 Storage association of data objects 56
5.5.1 EQUIVALENCE statement 56
5.5.2 COMMON statement 58

6. Use of data objects 61

6.1 Scalars 62
6.1.1 Substrings 62
6.1.2 Structure components 62
6.2 Arrays 63
6.2.1 Whole arrays 63
6.2.2 Array elements and array sections 63
6.3 Dynamic association 67
6.3.1 ALLOCATE statement 67
6.3.2 NULLIFY statement 68
6.3.3 DEALLOCATE statement 68

7. Expressions and assignment 70

7.1 Expressions 70
7.1.1 Form of an expression 70
7.1.2 Intrinsic operations 74
7.1.3 Defined operations 75
7.1.4 Data type, type parameters, and shape of an expression 75
7.1.5 Conformability rules for intrinsic operations 77
7.1.6 Scalar and array expressions 77
7.1.7 Evaluation of operations 79
7.2 Interpretation of intrinsic operations 83
7.2.1 Numeric intrinsic operations 83
7.2.2 Character intrinsic operation 84
7.2.3 Relational intrinsic operations 85
7.2.4 Logical intrinsic operations 86
7.3 Interpretation of defined operations 86
7.3.1 Unary defined operation 86
7.3.2 Binary defined operation 87
7.4 Precedence of operators 87
7.5 Assignment 89
7.5.1 Assignment statement 89
7.5.2 Pointer assignment 92
7.5.3 Masked array assignment—WHERE 92

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

8. Execution control 95

8.1 Executable constructs containing blocks 95
8.1.1 Rules governing blocks 95
8.1.2 IF construct 96
8.1.3 CASE construct 97
8.1.4 DO construct 100
8.2 Branching 107
8.2.1 Statement labels 107
8.2.2 GO TO statement 107
8.2.3 Computed GO TO statement 107
8.2.4 ASSIGN and assigned GO TO statement 107
8.2.5 Arithmetic IF statement 108
8.3 CONTINUE statement 108
8.4 STOP statement 108
8.5 PAUSE statement 108

9. Input/output statements 109

9.1 Records 109
9.1.1 Formatted record 109
9.1.2 Unformatted record 109
9.1.3 Endfile record 110
9.2 Files 110
9.2.1 External files 110
9.2.2 Internal files 112
9.3 File connection 113
9.3.1 Unit existence 114
9.3.2 Connection of a file to a unit 114
9.3.3 Preconnection 115
9.3.4 The OPEN statement 115
9.3.5 The CLOSE statement 118
9.4 Data transfer statements 119
9.4.1 Control information list 119
9.4.2 Data transfer input/output list 123
9.4.3 Error, end-of-record, and end-of-file conditions 124
9.4.4 Execution of a data transfer input/output statement 125
9.4.5 Printing of formatted records 128
9.4.6 Termination of data transfer statements 128
9.5 File positioning statements 128
9.5.1 BACKSPACE statement 129
9.5.2 ENDFILE statement 129
9.5.3 REWIND statement 129
9.6 File inquiry 130
9.6.1 Inquiry specifiers 130
9.6.2 Restrictions on inquiry specifiers 134
9.6.3 Inquire by output list 134
9.7 Restrictions on function references and list items 134
9.8 Restriction on input/output statements 134

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

10. Input/output editing 135

10.1 Explicit format specification methods 135
10.1.1 FORMAT statement 135
10.1.2 Character format specification 135
10.2 Form of a format item list 136
10.2.1 Edit descriptors 136
10.2.2 Fields 137
10.3 Interaction between input/output list and format 137
10.4 Positioning by format control 138
10.5 Data edit descriptors 139
10.5.1 Numeric editing 139
10.5.2 Logical editing 143
10.5.3 Character editing 144
10.5.4 Generalized editing 144
10.6 Control edit descriptors 145
10.6.1 Position editing 145
10.6.2 Slash editing 146
10.6.3 Colon editing 146
10.6.4 S, SP, and SS editing 146
10.6.5 P editing 147
10.6.6 BN and BZ editing 147
10.7 Character string edit descriptors 147
10.7.1 Character constant edit descriptor 147
10.7.2 H editing 148
10.8 List-directed formatting 148
10.8.1 List-directed input 148
10.8.2 List-directed output 150
10.9 Namelist formatting 151
10.9.1 Namelist input 151
10.9.2 Namelist output 154

11. Program units 156

11.1 Main program 156
11.1.1 Main program specifications 156
11.1.2 Main program executable part 156
11.1.3 Main program internal procedures 157
11.2 External subprograms 157
11.3 Modules 157
11.3.1 Module reference 157
11.3.2 The USE statement and use association 158
11.3.3 Examples of the use of modules 159
11.4 Block data program units 161

12. Procedures 163

12.1 Procedure classifications 163
12.1.1 Procedure classification by reference 163
12.1.2 Procedure classification by means of definition 163
12.2 Characteristics of procedures 165
12.2.1 Characteristics of dummy arguments 165
12.2.2 Characteristics of function results 166

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

12.3 Procedure interface 166
12.3.1 Implicit and explicit interfaces 166
12.3.2 Specification of the procedure interface 167
12.4 Procedure reference 171
12.4.1 Actual argument list 171
12.4.2 Function reference 174
12.4.3 Elemental intrinsic function reference 174
12.4.4 Subroutine reference 174
12.4.5 Elemental intrinsic subroutine reference 175
12.5 Procedure definition 175
12.5.1 Intrinsic procedure definition 175
12.5.2 Procedures defined by subprograms 175
12.5.3 Definition of procedures by means other than Fortran 181
12.5.4 Statement function 182

13. Intrinsic procedures 183

13.1 Intrinsic functions 183
13.2 Elemental intrinsic procedures 183
13.2.1 Elemental intrinsic function arguments and results 183
13.2.2 Elemental intrinsic subroutine arguments 183
13.3 Positional arguments or argument keywords 183
13.4 Argument presence inquiry function 184
13.5 Numeric, mathematical, character, kind, logical, and bit procedures 184
13.5.1 Numeric functions 184
13.5.2 Mathematical functions 184
13.5.3 Character functions 184
13.5.4 Character inquiry function 184
13.5.5 Kind functions 184
13.5.6 Logical function 184
13.5.7 Bit manipulation and inquiry procedures 184
13.6 Transfer function 185
13.7 Numeric manipulation and inquiry functions 185
13.7.1 Models for integer and real data 185
13.7.2 Numeric inquiry functions 186
13.7.3 Floating point manipulation functions 186
13.8 Array intrinsic functions 186
13.8.1 The shape of array arguments 186
13.8.2 Mask arguments 186
13.8.3 Vector and matrix multiplication functions 186
13.8.4 Array reduction functions 187
13.8.5 Array inquiry functions 187
13.8.6 Array construction functions 187
13.8.7 Array reshape function 187
13.8.8 Array manipulation functions 187
13.8.9 Array location functions 187
13.8.10 Pointer association status inquiry functions 187
13.9 Intrinsic subroutines 187
13.9.1 Date and time subroutines 188
13.9.2 Pseudorandom numbers 188
13.9.3 Bit copy subroutine 188

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

13.10 Generic intrinsic functions 188
13.10.1 Argument presence inquiry function 188
13.10.2 Numeric functions 188
13.10.3 Mathematical functions 189
13.10.4 Character functions 189
13.10.5 Character inquiry function 189
13.10.6 Kind functions 190
13.10.7 Logical function 190
13.10.8 Numeric inquiry functions 190
13.10.9 Bit inquiry function 190
13.10.10 Bit manipulation functions 190
13.10.11 Transfer function 190
13.10.12 Floating-point manipulation functions 190
13.10.13 Vector and matrix multiply functions 191
13.10.14 Array reduction functions 191
13.10.15 Array inquiry functions 191
13.10.16 Array construction functions 191
13.10.17 Array reshape function 191
13.10.18 Array manipulation functions 192
13.10.19 Array location functions 192
13.10.20 Pointer association status inquiry function 192
13.11 Intrinsic subroutines 192
13.12 Specific names for intrinsic functions 192
13.13 Specifications of the intrinsic procedures 194

14. Scope, association, and definition 241

14.1 Scope of names 241
14.1.1 Global entities 241
14.1.2 Local entities 241
14.1.3 Statement entities 245
14.2 Scope of labels 245
14.3 Scope of external input/output units 245
14.4 Scope of operators 245
14.5 Scope of the assignment symbol 245
14.6 Association 245
14.6.1 Name association 246
14.6.2 Pointer association 246
14.6.3 Storage association 247
14.7 Definition and undefinition of variables 249
14.7.1 Definition of objects and subobjects 249
14.7.2 Variables that are always defined 249
14.7.3 Variables that are initially defined 249
14.7.4 Variables that are initially undefined 250
14.7.5 Events that cause variables to become defined 250
14.7.6 Events that cause variables to become undefined 251
14.8 Allocation status 252

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Annexes

A. Glossary of technical terms 254

B. Decremental features 262

B.1 Deleted features 262
B.2 Obsolescent features 262
B.2.1 Alternate return 262
B.2.2 PAUSE statement 263
B.2.3 ASSIGN and assigned GO TO statements 263
B.2.4 Assigned FORMAT specifiers 263
B.2.5 H editing 263

C. Section notes 264

C.1 Section 1 notes 264
C.1.1 Conformance (1.4) 264
C.2 Section 2 notes 264
C.2.1 Keywords 264
C.3 Section 3 notes 264
C.3.1 Representable characters (3.1.5) 264
C.3.2 Comment lines (3.3.1.1, 3.3.2.1) 264
C.3.3 Statement labels (3.2.5) 265
C.3.4 Source form (3.3) 265
C.4 Section 4 notes 265
C.4.1 Zero (4.3.1) 265
C.4.2 Characters (4.2) 265
C.4.3 Intrinsic and derived data types (4.3, 4.4) 266
C.4.4 Selection of the approximation methods 266
C.4.5 Storage of derived types (4.4.1) 267
C.4.6 Pointers 267
C.5 Section 5 notes 268
C.5.1 Type declaration statements (5.1) 268
C.5.2 The POINTER attribute (5.1.2.7) 268
C.5.3 The TARGET attribute (5.1.2.8) 269
C.5.4 PARAMETER statements and IMPLICIT NONE (5.2.10, 5.3) 270
C.5.5 EQUIVALENCE statement extensions (5.5.1) 270
C.5.6 COMMON statement extensions (5.5.2) 270
C.6 Section 6 notes 270
C.6.1 Substrings (6.1.1) 270
C.6.2 Array element references (6.2.2) 270
C.6.3 Structure components (6.1.2) 270
C.6.4 Pointer allocation and association 271
C.7 Section 7 notes 272
C.7.1 Character assignment 272
C.7.2 Evaluation of function references 272
C.7.3 Pointers in expressions 272
C.7.4 Pointers on the left side of an assignment 273
C.8 Section 8 notes 273
C.8.1 Loop control 273

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

	C.8.2	The CASE construct	274
	C.8.3	Examples of invalid DO constructs	274
C.9	Section 9 notes		274
	C.9.1	Input/output records (9.1)	274
	C.9.2	Files (9.2)	275
	C.9.3	OPEN statement (9.3.4)	276
	C.9.4	Connection properties (9.3.2)	278
	C.9.5	CLOSE statement (9.3.5)	279
	C.9.6	INQUIRE statement (9.6)	280
	C.9.7	Keyword specifiers	280
	C.9.8	Format specifications (9.4.1.1)	280
	C.9.9	Unformatted input/output (9.4.4.4.1)	280
	C.9.10	Input/output restrictions	280
	C.9.11	Pointers in an input/output list	280
	C.9.12	Derived type objects in an input/output list (9.4.2)	280
C.10	Section 10 notes		281
	C.10.1	Character constant format specification (10.1.2, 10.7.1)	281
	C.10.2	T edit descriptor (10.6.1.1)	281
	C.10.3	Length of formatted records	281
	C.10.4	Number of records (10.3, 10.4, 10.6.2)	281
	C.10.5	List-directed input/output (10.8)	282
	C.10.6	List-directed input (10.8.1)	282
	C.10.7	Namelist list items for character input (10.9.1.3)	282
	C.10.8	Namelist output records (10.9.2.2)	282
C.11	Section 11 notes		283
	C.11.1	Main program and block data program unit (11.1, 11.4)	283
	C.11.2	Dependent compilation (11.3)	283
	C.11.3	Pointers in modules	285
	C.11.4	Example of a module (11.3)	285
C.12	Section 12 notes		288
	C.12.1	Examples of host association (12.1.2.2.1)	288
	C.12.2	External procedures (12.3.2.2)	289
	C.12.3	Procedures defined by means other than Fortran (12.5.3)	289
	C.12.4	Procedure interfaces (12.3)	290
	C.12.5	Argument association and evaluation (12.4.1)	290
	C.12.6	Argument intent specification (12.4.1.1)	291
	C.12.7	Dummy argument restrictions (12.5.2.9)	291
	C.12.8	Pointers and targets as arguments	291
	C.12.9	The ASSOCIATED function (13.13.13)	292
	C.12.10	Internal procedure restrictions	292
	C.12.11	The result variable (12.5.2.2)	293
C.13	Section 13 notes		293
	C.13.1	Summary of features	293
	C.13.2	Examples	295
	C.13.3	FORmula TRANslation and array processing	299
	C.13.4	Sum of squared residuals	300
	C.13.5	Vector norms: infinity-norm and one-norm	300
	C.13.6	Matrix norms: infinity-norm and one-norm	300
	C.13.7	Logical queries	300
	C.13.8	Parallel computations	301
	C.13.9	Example of element-by-element computation	301

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

C.13.10 Bit manipulation and inquiry procedures 302
C.14 Section 14 notes 302
C.14.1 Storage association of zero-sized objects 302

D. Syntax rules 303

D.1 Syntax rules and constraints 303
D.1.1 Overview 303
D.1.2 Fortran terms and concepts 303
D.1.3 Characters, lexical tokens, and source form 306
D.1.4 Intrinsic and derived data types 307
D.1.5 Data object declarations and specifications 310
D.1.6 Use of data objects 315
D.1.7 Expressions and assignment 317
D.1.8 Execution control 320
D.1.9 Input/output statements 323
D.1.10 Input/output editing 326
D.1.11 Program units 328
D.1.12 Procedures 329
D.1.13 Intrinsic procedures 331
D.1.14 Scope, association, and definition 332
D.2 Cross references 332
D.2.1 Nonterminal symbols that are defined 332
D.2.2 Nonterminal symbols that are not defined 338
D.2.3 Terminal symbols 339

E. Permuted index for headings 344

F. Index 363

Figures

Figure 2.1 Requirements on statement ordering 11

Tables

Table 2.1 Statements allowed in scoping units 12
Table 3.1 Special characters 19
Table 6.1 Subscript order value 65
Table 7.1 Type of operands and result for the intrinsic operation $[x_1]$ *op* x_2 74
Table 7.2 Interpretation of the numeric intrinsic operators 84
Table 7.3 Interpretation of the character intrinsic operator // 84
Table 7.4 Interpretation of the relational intrinsic operators 85
Table 7.5 Interpretation of the logical intrinsic operators 86
Table 7.6 The values of operations involving logical intrinsic operators 86
Table 7.7 Categories of operations and relative precedences 87
Table 7.8 Type conformance for the intrinsic assignment statement *variable* = *expr* 89
Table 7.9 Numeric conversion and assignment statement *variable* = *expr* 90
Table C.1 Values assigned to INQUIRE specifier variables 279

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Foreword

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

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

International Standard ISO/IEC 1539 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

This second edition cancels and replaces the first edition (ISO 1539 : 1980), which has been technically revised.

Annexes A, B, C, D, E and F are for information only.

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Introduction

Standard programming language Fortran

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

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

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

Overview

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

(1) Array operations

(2) Improved facilities for numerical computation

(3) Parameterized intrinsic data types

(4) User-defined data types

(5) Facilities for modular data and procedure definitions

(6) Pointers

(7) The concept of language evolution

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

Array operations

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

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Numerical computation

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

Parameterized character data type

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

Derived types

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

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

Modular definitions

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

Pointers

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

Language evolution

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

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

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

Organization of this International Standard

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

High/Low Level Concepts	Sections 1, 2, 3
Data Concepts	Sections 4, 5, 6
Computations	Sections 7, 13
Execution Control	Section 8
Input/Output	Sections 9, 10
Program Units	Sections 11, 12
Scoping and Association Rules	Section 14

High/low level concepts

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

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

Data concepts

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

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

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

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

Section 6 (Use of Data Objects) deals mainly with the concept of a variable, and describes the various forms that variables may take. Scalar variables include character strings and substrings, structured (derived-type) objects, structure components, and array elements. Arrays are considered to be variables, as are array sections. Among the array facilities described here are array sections (subarrays), and array

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

allocation and deallocation (user controlled dynamic arrays). The section concludes with a summary of the allowed appearances of array names.

Computations

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

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

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

Execution control

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

Input/output

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

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

Program units

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

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

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Scoping and association rules

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Information technology — Programming languages — Fortran

Section 1 : Overview

1.1 Scope

This International Standard specifies the form and establishes the interpretation of programs expressed in the Fortran language. The purpose of this International Standard is to promote portability, reliability, maintainability, and efficient execution of Fortran programs for use on a variety of computing systems.

1.2 Processor

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

1.3 Inclusions and exclusions

This International Standard specifies the bounds of the Fortran language by identifying both those items included and those items excluded.

1.3.1 Inclusions

This International Standard specifies:

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

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

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

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

1.3.2 Exclusions

This International Standard does not specify:

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

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

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

(4) The program and processor behavior when the rules of this International Standard fail to establish an interpretation except for the processor detection and reporting requirements in items (2) to (8) of 1.4

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

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

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

(8) The physical properties and implementation of storage

Documents 1
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

1.4 Conformance

The requirements, prohibitions, and options specified in this International Standard refer primarily to permissible forms and relationships for a standard-conforming program rather than for a processor.

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

A processor conforms to this International Standard if:

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

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

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

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

(5) It contains the capability to detect and report the use within a submitted program unit of source form or characters not permitted by Section 3.

(6) It contains the capability to detect and report the use within a submitted program of name usage not consistent with the scope rules for names, labels, operators, and assignment symbols in Section 14.

(7) It contains the capability to detect and report the use within a submitted program unit of intrinsic procedures whose names are not defined in Section 13.

(8) It contains the capability to detect and report the reason for rejecting a submitted program.

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

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

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

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

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

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

1.4.1 FORTRAN 77 compatibility

Except as noted in this section, this International Standard is an upward compatible extension to the preceding Fortran International Standard, ISO 1539:1980, informally referred to as FORTRAN 77, and a standard-conforming processor for this International Standard is a standard-conforming processor for FORTRAN 77. Any standard-conforming FORTRAN 77 program remains standard conforming under this International Standard; however, see item (4) below regarding intrinsic procedures. This International Standard restricts the behavior for some features that were processor dependent in FORTRAN 77. Therefore, a standard-conforming FORTRAN 77 program that uses one of these processor-dependent features may have a different interpretation under this International Standard, yet remain a standard-conforming program. The following FORTRAN 77 features have different interpretations in this International Standard:

(1) FORTRAN 77 permitted a processor to supply more precision derived from a real constant than can be contained in a real datum when the constant is used to initialize a DOUBLE PRECISION data object in a DATA statement. This International Standard does not permit a processor this option.

(2) If a named variable that is not in a common block is initialized in a DATA statement and does not have the SAVE attribute specified, FORTRAN 77 left its SAVE attribute processor dependent. This International Standard specifies (5.2.9) that this named variable has the SAVE attribute.

(3) FORTRAN 77 required that the number of characters required by the input list must be less than or equal to the number of characters in the record during formatted input. This International Standard specifies (9.4.4.4.2) that the input record is logically padded with blanks if there are not enough characters in the record, unless the PAD= 'NO' option is specified in an appropriate OPEN statement.

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

1.5 Notation used in this International Standard

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

1.5.1 Syntax rules

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

(1) Characters from the Fortran character set (3.1) are to be written as shown, except where otherwise noted.

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Some common abbreviations used in syntactic terms are:

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

(3) The syntactic metasymbols used are:

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

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

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

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

An example of the use of the syntax rules is:

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

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

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

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

4
67
1999
10243852

1.5.2 Assumed syntax rules

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

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

xyz-name **is** *name*

scalar-xyz **is** *xyz*

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

Constraint: *scalar-xyz* must be scalar.

1.5.3 Syntax conventions and characteristics

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

(2) The rules on statement ordering are described rigorously in the definition of *program-unit* (R202-R216). Expression hierarchy is described rigorously in the definition of *expr* (R723).

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

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

(5) The term *"subscript"* (for example, R617, R618, and R619) is used consistently in array definitions.

1.5.4 Text conventions

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

1.6 Deleted and obsolescent features

This International Standard protects the users' investment in existing software by including all of the language elements of FORTRAN 77 that are not processor dependent. This document identifies two categories of outmoded features. There are none in the first category, **deleted features**, which consists of features considered to have been redundant in FORTRAN 77 and largely unused. Those in the second category, **obsolescent features**, are considered to have been redundant in FORTRAN 77, but are still used frequently.

1.6.1 Nature of deleted features

(1) Better methods existed in FORTRAN 77.

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

1.6.2 Nature of obsolescent features

(1) Better methods existed in FORTRAN 77.

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

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

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

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

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

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

1.7 Modules

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

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

1.8 Normative references

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

ISO 8601:1988, *Data Elements and Interchange Formats—Information interchange—Representation of dates and times.*
ISO 646:1983, *Information processing—ISO 7-bit coded character set for information interchange.*
CCIR Recommendation 460-2.

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

Section 2 : Fortran terms and concepts

2.1 High level syntax

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

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

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

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

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

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

R1215 *function-subprogram* **is** *function-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1219 *subroutine-subprogram* **is** *subroutine-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-subroutine-stmt

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

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

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

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

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

R206 *implicit-part-stmt*
is *implicit-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*

R207 *declaration-construct*
is *derived-type-def*
or *interface-block*
or *type-declaration-stmt*
or *specification-stmt*
or *parameter-stmt*
or *format-stmt*
or *entry-stmt*
or *stmt-function-stmt*

R208 *execution-part*
is *executable-construct*
[*execution-part-construct*] ...

R209 *execution-part-construct*
is *executable-construct*
or *format-stmt*
or *data-stmt*
or *entry-stmt*

R210 *internal-subprogram-part*
is *contains-stmt*
internal-subprogram
[*internal-subprogram*] ...

R211 *internal-subprogram*
is *function-subprogram*
or *subroutine-subprogram*

R212 *module-subprogram-part*
is *contains-stmt*
module-subprogram
[*module-subprogram*] ...

R213 *module-subprogram*
is *function-subprogram*
or *subroutine-subprogram*

R214 *specification-stmt*
is *access-stmt*
or *allocatable-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 *save-stmt*
or *target-stmt*

R215 *executable-construct*
is *action-stmt*
or *case-construct*
or *do-construct*
or *if-construct*
or *where-construct*

AVA Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

R216 *action-stmt* **is** *allocate-stmt*
or *assignment-stmt*
or *backspace-stmt*
or *call-stmt*
or *close-stmt*
or *computed-goto-stmt*
or *continue-stmt*
or *cycle-stmt*
or *deallocate-stmt*
or *endfile-stmt*
or *end-function-stmt*
or *end-program-stmt*
or *end-subroutine-stmt*
or *exit-stmt*
or *goto-stmt*
or *if-stmt*
or *inquire-stmt*
or *nullify-stmt*
or *open-stmt*
or *pointer-assignment-stmt*
or *print-stmt*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*
or *stop-stmt*
or *where-stmt*
or *write-stmt*
or *arithmetic-if-stmt*
or *assign-stmt*
or *assigned-goto-stmt*
or *pause-stmt*

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

2.2 Program unit concepts

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

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

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

(2) A procedure interface body, excluding any derived-type definitions and procedure interface bodies contained within it (12.3.2.1), or

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

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

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

2.2.1 Executable program

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

2.2.2 Main program

The main program is described in 11.1.

2.2.3 Procedure

A **procedure** encapsulates an arbitrary sequence of computations that may be invoked directly during program execution. Procedures are either functions or subroutines. A **function** is a procedure that is invoked in an expression; its invocation causes a value to be computed which is then used in evaluating the expression. The variable that returns the value of a function is called the **result variable**. A **subroutine** is a procedure that is invoked in a CALL statement or by a defined assignment statement (12.4.4, 7.5.1.3). A subroutine may be used to change the program state by changing the values of any of the data objects accessible to the subroutine; a function may do this in addition to computing the function value.

Procedures are described further in Section 12.

2.2.3.1 External procedure

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

2.2.3.2 Module procedure

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

2.2.3.3 Internal procedure

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

2.2.3.4 Procedure interface block

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

2.2.4 Module

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

2.3 Execution concepts

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

2.3.1 Executable/nonexecutable statements

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

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

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE,
MODULE, or BLOCK DATA Statement</td></tr>
<tr><td colspan="3">USE Statements</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
Statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER
Statements</td><td>IMPLICIT
Statements</td></tr>
<tr><td>PARAMETER
and DATA
Statements</td><td>Derived-Type Definitions,
Interface Blocks,
Type Declaration Statements,
Statement Function Statements,
and Specification Statements</td></tr>
<tr><td>DATA
Statements</td><td>Executable
Constructs</td></tr>
<tr><td colspan="3">CONTAINS Statement</td></tr>
<tr><td colspan="3">Internal Subprograms
or Module Subprograms</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

Figure 2.1 Requirements on statement ordering

2.3.2 Statement order

The syntax rules of Section 2.1 specify the statement order within program units and subprograms. These rules are illustrated in Figure 2.1 and Table 2.1. Figure 2.1 shows the ordering rules for statements and applies to all program units and subprograms. Vertical lines delineate varieties of statements that may be interspersed and horizontal lines delineate varieties of statements that must not be interspersed. USE statements, if any, must appear immediately after the program unit heading. Internal or module

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

subprograms must follow a CONTAINS statement. Between USE and CONTAINS statements in a subprogram, nonexecutable statements generally precede executable statements, though the FORMAT statement, DATA statement, and ENTRY statement may appear among the executable statements. Table 2.1 shows which statements are allowed in a scoping unit.

Table 2.1 Statements allowed in scoping units

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

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

2.3.3 The END statement

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

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

2.3.4 Execution sequence

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

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

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

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

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

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

2.4 Data concepts

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

2.4.1 Data type

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

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

2.4.1.1 Intrinsic type

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

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

2.4.1.2 Derived type

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

Derived types are described further in 4.4.

2.4.2 Data value

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

2.4.3 Data entity

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

2.4.3.1 Data object

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

Subobjects are portions of certain named objects that may be referenced and defined (variables only) independently of the other portions. These include portions of arrays (array elements and array sections),

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

portions of character strings (substrings), and portions of structures (components). Subobjects are themselves data objects, but subobjects are referenced only by subobject designators. A subobject of a variable is a variable. Subobjects are described in Section 6.

Objects referenced by a name are:

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

Subobjects referenced by a subobject designator are:

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

2.4.3.1.1 Variable

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

2.4.3.1.2 Constant

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

2.4.3.1.3 Constant subobject

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

2.4.3.2 Expression

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

2.4.3.3 Function reference

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

2.4.4 Scalar

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

AVA Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

2.4.5 Array

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

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

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

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

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

2.4.6 Pointer

A **pointer** is a variable that has the POINTER attribute. A pointer is **associated** with a target by allocation (6.3.1) or pointer assignment (7.5.2). A pointer must not be referenced of defined until it is associated. A pointer is **disassociated** following execution of a NULLIFY or DEALLOCATE statement or following pointer association with a disassociated pointer. A disassociated pointer is not currently associated with a target (14.6.2). If the pointer is an array, the rank is declared, but the extents are determined when the pointer is associated with a target.

2.4.7 Storage

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

2.5 Fundamental terms

The following terms are defined here and used throughout this International Standard.

2.5.1 Name and designator

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

2.5.2 Keyword

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

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

2.5.3 Declaration

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

2.5.4 Definition

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

2.5.5 Reference

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

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

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

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

2.5.6 Association

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

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

2.5.7 Intrinsic

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

2.5.8 Operator

An **operator** specifies a particular computation involving one (unary operator) or two (binary operator) data values (operands). Fortran contains a number of intrinsic operators (e.g., the arithmetic operators

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

+, -, *, /, and ** with numeric operands and the logical operators .AND., .OR., etc. with logical operands). Additional operators also may be defined within an executable program (7.1.3).

2.5.9 Sequence

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

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Section 3 : Characters, lexical tokens, and source form

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

3.1 Processor character set

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

(1) **Control characters** ("newline", for example)

(2) **Graphic characters**

- (a) Letters (3.1.1)
- (b) Digits (3.1.2)
- (c) Underscore (3.1.3)
- (d) Special characters (3.1.4)
- (e) Other characters (3.1.5)

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

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

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

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

3.1.1 Letters

The twenty-six **letters** are:

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

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

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

3.1.2 Digits

The ten **digits** are:

0 1 2 3 4 5 6 7 8 9

The ten digits define the syntactic class *digit*.

3.1.3 Underscore

R303 *underscore* **is** _

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

3.1.4 Special characters

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

Table 3.1 Special characters

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

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

3.1.5 Other characters

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

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

3.2 Low-level syntax

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

3.2.1 Keywords

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

3.2.2 Names

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

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

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Examples of names:

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

3.2.3 Constants

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

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

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

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

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

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

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

3.2.4 Operators

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

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

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

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

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

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

AVA Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

		or >
		or >=
R719	*not-op*	**is** .NOT.
R720	*and-op*	**is** .AND.
R721	*or-op*	**is** .OR.
R722	*equiv-op*	**is** .EQV. **or** .NEQV.
R311	*defined-operator*	**is** *defined-unary-op* **or** *defined-binary-op* **or** *extended-intrinsic-op*
R704	*defined-unary-op*	**is** . *letter* [*letter*] ... .
R724	*defined-binary-op*	**is** . *letter* [*letter*] ... .
R312	*extended-intrinsic-op*	**is** *intrinsic-operator*

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

3.2.5 Statement labels

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

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

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

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

```
99999
10
 010
```

are all statement labels. The last two are equivalent.

3.2.6 Delimiters

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

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

3.3 Source form

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

A **character context** means characters within a character literal constant (4.3.2.1) or within a character string edit descriptor (10.7).

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

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

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

3.3.1 Free source form

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

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

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

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

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

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

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

3.3.1.1 Free form commentary

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

3.3.1.2 Free form statement separation

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

";" separator. A ";" separator that is the last nonblank character on a line, or the last nonblank character ahead of commentary, is ignored.

3.3.1.3 Free form statement continuation

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

3.3.1.3.1 Noncharacter context continuation

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

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

3.3.1.3.2 Character context continuation

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

3.3.1.4 Free form statements

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

3.3.2 Fixed source form

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

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

3.3.2.1 Fixed form commentary

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

3.3.2.2 Fixed form statement separation

The character ";" separates statements, or partial statements, on a single source line except when it appears in a character context or in a comment. If a ";" separator is followed by zero or more blanks and one or more ";" separators, the sequence from the first ";" to the last, inclusive, is interpreted as a single ";" separator. A ";" separator that is the last nonblank character on a line, or the last nonblank character ahead of commentary, is ignored.

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

3.3.2.3 Fixed form statement continuation

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

3.3.2.4 Fixed form statements

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

3.4 Including source text

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

INCLUDE *char-literal-constant*

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

An INCLUDE line is not a Fortran statement.

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

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

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

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

Section 4 : Intrinsic and derived data types

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

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

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

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

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

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

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

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

4.1 The concept of data type

A data type has (1) a name, (2) a set of valid values, (3) a means to denote such values (constants), and (4) a set of operations to manipulate the values.

4.1.1 Set of values

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

4.1.2 Constants

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

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

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

4.1.3 Operations

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

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

4.2 Relationship of types and values to objects

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

An array is a collection of subobjects.

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

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

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

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

4.3 Intrinsic data types

The intrinsic data types are:

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

4.3.1 Numeric types

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

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

4.3.1.1 Integer type

The set of values for the **integer type** is a subset of the mathematical integers. A processor must provide one or more **representation methods** that define sets of values for data of type integer. Each such method is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.13.51). Among the kind values that provide a given range, one providing the smallest range is returned by the intrinsic function SELECTED_INT_KIND (13.13.92). The decimal exponent range of a representation method is returned by the intrinsic function RANGE (13.13.85).

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

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

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

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

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

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

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

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

R406 *sign* **is** +
or -

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

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

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

Examples of unsigned and signed integer literal constants are:

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

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

An integer constant is interpreted as a decimal value.

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

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

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

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

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

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

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

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

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

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

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

4.3.1.2 Real type

The **real type** has values that approximate the mathematical real numbers. A processor must provide two or more **approximation methods** that define sets of values for data of type real. Each such method has a **representation method** and is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of an approximation method is returned by the intrinsic inquiry function KIND (13.13.51). Among the kind values that provide a given precision and a given exponent range, one providing the smallest decimal precision is returned by the intrinsic function SELECTED_REAL_KIND (13.13.93). The decimal precision and decimal exponent range of an approximation method are returned by the intrinsic functions PRECISION (13.13.79) and RANGE (13.13.85).

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

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

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

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

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

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

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

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

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

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

A real literal constant without a kind type parameter is a default real constant if it is without an exponent part or has exponent letter E, and is a double precision real constant if it has exponent letter D. A real literal constant written with a kind type parameter is a real constant with the specified kind type parameter.

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

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

Examples of signed real literal constants are:

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

In 10.93E7_QUAD, the named constant QUAD must have been defined and its value must be a valid kind type parameter value for the real type.

4.3.1.3 Complex type

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

Each approximation method used to represent data entities of type real must be available for both the real and imaginary parts of a data entity of type complex. A **kind** type parameter may be specified for a complex entity and selects for both parts the real approximation method characterized by this kind type parameter value.

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

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

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

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

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

Documents
Document Sharing Hub
Full document details: ISO_IEC 1539_1991

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

Examples of complex literal constants are:

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

4.3.2 Nonnumeric types

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

4.3.2.1 Character type

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

A processor must provide one or more **representation methods** that define sets of values for data of type character. Each such method is characterized by a value for a type parameter called the **kind** type parameter. The kind type parameter of a representation method is returned by the intrinsic inquiry function KIND (13.13.51). Any character of a particular representation method representable in the processor may occur in a character string of that representation method.

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

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

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

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

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

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

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

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

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

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

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

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

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

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

Examples of character literal constants are:

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

both of which have the value DON'T and

```
''
```

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

Examples of nondefault character literal constants, where the processor supports the corresponding character sets, are:

```
CYRILLIC_' Без неё ничто невозможно '
HINDI_' उसके बिना कुछ थी संथव नही '
MAGYAR_' Őnélküle, nem csinálhatom'
NIHONGO_' 彼女なしでは何もできない。'
```

where CYRILLIC, HINDI, MAGYAR, and NIHONGO are named constants whose values are the kind type parameters for Cyrillic (Russian and other Slavic languages), Hindi, Magyar (Hungarian), and Nihongo (Japanese) characters, respectively. Note that Nihongo is a very large (ideographic) character set, whereas Cyrillic, Hindi, and Magyar are alphabetic.

4.3.2.1.1 Collating sequence

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

ICHAR (*'character'*)

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

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

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

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

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

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

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

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

ISO 646:1983 (International Reference Version) assigns numerical codes to a set of characters that includes the letters, digits, underscore, and special characters; the sequence of such codes is called in this standard

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991

the **ASCII collating sequence**. Note that ISO 646:1983 is the international equivalent of ANSI X3.4-1986, commonly known as ASCII. The intrinsic functions ACHAR and IACHAR provide conversions between these characters and the integers of the ASCII collating sequence. The intrinsic functions LGT, LGE, LLE, and LLT provide comparisons between strings based on the ASCII collating sequence. International portability is guaranteed if the set of characters used is limited to the letters, digits, underscore, and special characters.

4.3.2.2 Logical type

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

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

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

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

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

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

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

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

4.4 Derived types

Additional data types may be derived from the intrinsic data types. A type definition is required to define the name of the type and the names and types of its components. Ultimately, a derived type is resolved into **ultimate components** that are either of intrinsic type or are pointers.

By default, derived types defined in the specification part of a module are accessible (5.1.2.2, 5.2.3) in any scoping unit that accesses the module. This default may be changed to restrict the accessibility of such types to the host module itself. A particular type definition may be declared to be public or private regardless of the default accessibility declared for the module. In addition, a type may be accessible while its components are private.

By default, no storage sequence is implied by the order of the component definitions. However, if the definition of a derived type contains a SEQUENCE statement, the type is a **sequence type**. The order of the component definitions in a sequence type specifies a storage sequence for objects of that type.

The type specifier for derived types is the keyword TYPE followed by the name of the type in parentheses (R502).

4.4.1 Derived-type definition

R422 *derived-type-def* **is** *derived-type-stmt*
[*private-sequence-stmt*] ...
component-def-stmt
[*component-def-stmt*] ...
end-type-stmt

Documents
Document Sharing Hub

Full document details: ISO_IEC 1539_1991