
**Information technology — Programming
languages, their environments and system
software interfaces — Language-
independent datatypes**

*Technologies de l'information — Langages de programmation, leur
environnement et interfaces du logiciel système — Types de données
indépendants du langage*



Contents

Foreword v

Introduction vi

1 Scope 1

2 Conformance 1

2.1 Direct conformance 2

2.2 Indirect conformance 2

2.3 Conformance of a mapping standard 2

3 Normative References 3

4 Definitions 3

5 Conventions Used in this International Standard 5

5.1 Formal syntax 5

5.2 Text conventions 6

6 Fundamental Notions 6

6.1 Datatype 6

6.2 Value space 7

6.3 Datatype properties 7

6.3.1 Equality 7

6.3.2 Order 7

6.3.3 Bound 8

6.3.4 Cardinality 8

6.3.5 Exact and approximate 8

6.3.6 Numeric 9

6.4 Primitive and non-primitive datatypes 9

6.5 Datatype generator 9

6.6 Characterizing operations 9

6.7 Datatype families 10

6.8 Aggregate datatypes 10

6.8.1 Homogeneity 11

© ISO/IEC 1996

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

ISO/IEC Copyright Office ● Case Postale 56 ● CH-1211 Genève 20 ● Switzerland

Printed in Switzerland

6.8.2	Size	11
6.8.3	Uniqueness	11
6.8.4	(Aggregate-imposed) ordering	11
6.8.5	Access method	11
6.8.6	Recursive structure	12
7	Elements of the Datatype Specification Language	12
7.1	IDN character-set	12
7.2	Whitespace	13
7.3	Lexical objects	13
7.3.1	Identifiers	13
7.3.2	Digit-string	14
7.3.3	Character-literal and string-literal	14
7.3.4	Keywords	14
7.4	Annotations	14
7.5	Values	15
7.5.1	Independent values	15
7.5.2	Dependent values	16
8	Datatypes	17
8.1	Primitive datatypes	17
8.1.1	Boolean	18
8.1.2	State	19
8.1.3	Enumerated	19
8.1.4	Character	20
8.1.5	Ordinal	21
8.1.6	Date-and-Time	21
8.1.7	Integer	22
8.1.8	Rational	23
8.1.9	Scaled	23
8.1.10	Real	24
8.1.11	Complex	26
8.1.12	Void	27
8.2	Subtypes and extended types	27
8.2.1	Range	28
8.2.2	Selecting	28
8.2.3	Excluding	28
8.2.4	Size	29
8.2.5	Explicit subtypes	29
8.2.6	Extended	30
8.3	Generated datatypes	30
8.3.1	Choice	31
8.3.2	Pointer	33
8.3.3	Procedure	34
8.4	Aggregate Datatypes	36
8.4.1	Record	37
8.4.2	Set	38
8.4.3	Bag	39
8.4.4	Sequence	40
8.4.5	Array	41

8.4.6	Table	43
8.5	Defined Datatypes	44
9	Declarations	45
9.1	Type Declarations	45
9.1.1	Renaming declarations	46
9.1.2	New datatype declarations	46
9.1.3	New generator declarations	46
9.2	Value Declarations	46
9.3	Termination Declarations	47
10	Defined Datatypes and Generators	47
10.1	Defined datatypes	47
10.1.1	Natural number	47
10.1.2	Modulo	48
10.1.3	Bit	48
10.1.4	Bit string	48
10.1.5	Character string	49
10.1.6	Time interval	50
10.1.7	Octet	50
10.1.8	Octet string	50
10.1.9	Private	50
10.1.10	Object identifier	51
10.2	Defined generators	52
10.2.1	Stack	53
10.2.2	Tree	53
10.2.3	Cyclic enumerated	53
10.2.4	Optional	54
11	Mappings	54
11.1	Outward Mappings	55
11.2	Inward Mappings	56
11.3	Reverse Inward Mapping	56
11.4	Support of Datatypes	57
11.4.1	Support of equality	57
11.4.2	Support of order	57
11.4.3	Support of bounds	57
11.4.4	Support of cardinality	57
11.4.5	Support for the exact or approximate property	58
11.4.6	Support for the numeric property	58
Annex A.	Character-Set Standards	59
Annex B.	Recommended Placement of Annotations.....	62
Annex C.	Implementation Notions of Datatypes	64
Annex D.	Syntax for the Common Interface Definition Notation	67
Annex E.	Example Mapping to Pascal	72
Annex F.	Example Mapping to MUMPS	85
Annex G.	Resolved Issues	89

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 JTC1. 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 11404 was prepared by Joint Technical Committee ISO/IEC JTC1, Information technology, Subcommittee SC22, *Programming languages, their environments and system software interfaces*.

Annexes A to G of this International Standard are for information only.

Introduction

Many specifications of software services and applications libraries are, or are in the process of becoming, International Standards. The interfaces to these libraries are often described by defining the form of reference, e.g. the “procedure call”, to each of the separate functions or services in the library, as it must appear in a user program written in some standard programming language (Fortran, COBOL, Pascal, etc.). Such an interface specification is commonly referred to as the “<language> binding of <service>”, e.g. the “Fortran binding of PHIGS” (ISO/IEC 9593-1:1990, *Information processing systems — Computer Graphics — Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings — Part 1: FORTRAN*).

This approach leads directly to a situation in which the standardization of a new service library immediately requires the standardization of the interface bindings to every standard programming language whose users might reasonably be expected to use the service, and the standardization of a new programming language immediately requires the standardization of the interface binding to every standard service package which users of that language might reasonably be expected to use. To avoid this n-to-m binding problem, ISO/IEC JTC1 (Information Technology) assigned to SC22 the task of developing an International Standard for Language-Independent Procedure Calling and a parallel International Standard for Language-Independent Datatypes, which could be used to describe the parameters to such procedures.

This International Standard provides the specification for the Language-Independent Datatypes. It defines a set of datatypes, independent of any particular programming language specification or implementation, that is rich enough so that any common datatype in a standard programming language or service package can be mapped to some datatype in the set.

The purpose of this International Standard is to facilitate commonality and interchange of datatype notions, at the conceptual level, among different languages and language-related entities. Each datatype specified in this International Standard has a certain basic set of properties sufficient to set it apart from the others and to facilitate identification of the corresponding (or nearest corresponding) datatype to be found in other standards. Hence, this International Standard provides a single common reference model for all standards which use the concept *datatype*. It is expected that each programming language standard will define a mapping from the datatypes supported by that programming language into the datatypes specified herein, semantically identifying its datatypes with datatypes of the reference model, and thereby with corresponding datatypes in other programming languages.

It is further expected that each programming language standard will define a mapping from those Language-Independent (LI) Datatypes which that language can reasonably support into datatypes which may be specified in the programming language. At the same time, this International Standard will be used, among other applications, to define a “language-independent binding” of the parameters to the procedure calls constituting the principal elements of the standard interface to each of the standard services. The production of such service bindings and language mappings leads, in cooperation with the parallel Language-Independent Procedure Calling mechanism, to a situation in which no further “<language> binding of <service>” documents need to be produced: Each service interface, by defining its parameters using LI datatypes, effectively defines the binding of such parameters to any standard programming language; and each language, by its mapping from the LI datatypes into the language datatypes, effectively defines the binding to that language of parameters to any of the standard services.

Information technology — Programming languages, their environments and system software interfaces — Language-independent datatypes

1 Scope

This International Standard specifies the nomenclature and shared semantics for a collection of datatypes commonly occurring in programming languages and software interfaces, referred to as the Language-Independent (LI) Datatypes. It specifies both primitive datatypes, in the sense of being defined *ab initio* without reference to other datatypes, and non-primitive datatypes, in the sense of being wholly or partly defined in terms of other datatypes. The specification of datatypes in this International Standard is "language-independent" in the sense that the datatypes specified are classes of datatypes of which the actual datatypes used in programming languages and other entities requiring the concept *datatype* are particular instances.

This International Standard expressly distinguishes three notions of "datatype", namely:

- the conceptual, or abstract, notion of a datatype, which characterizes the datatype by its nominal values and properties;
- the structural notion of a datatype, which characterizes the datatype as a conceptual organization of specific component datatypes with specific functionalities; and
- the implementation notion of a datatype, which characterizes the datatype by defining the rules for representation of the datatype in a given environment.

This International Standard defines the abstract notions of many commonly used primitive and non-primitive datatypes which possess the structural notion of atomicity. This International Standard does not define all atomic datatypes; it defines only those which are common in programming languages and software interfaces. This International Standard defines structural notions for the specification of other non-primitive datatypes and provides a means by which datatypes not defined herein can be defined structurally in terms of the LI datatypes defined herein.

This International Standard defines a partial vocabulary for implementation notions of datatypes and provides for, but does not require, the use of this vocabulary in the definition of datatypes. The primary purpose of this vocabulary is to identify common implementation notions associated with datatypes and to distinguish them from conceptual notions. Specifications for the use of implementation notions are deemed to be outside the scope of this International Standard, which is concerned solely with the identification and distinction of datatypes.

This International Standard specifies the required elements of mappings between the LI datatypes and the datatypes of some other language. This International Standard does not specify the precise form of a mapping, but rather the required information content of a mapping.

2 Conformance

An information processing product, system, element or other entity may conform to this International Standard either directly, by utilizing datatypes specified in this International Standard in a conforming manner (2.1), or indirectly, by means of mappings between internal datatypes used by the entity and the datatypes specified in this International Standard (2.2).

NOTE — The general term **information processing entity** is used in this clause to include anything which processes information and contains the concept of *datatype*. Information processing entities for which conformance to this International Standard may be appropriate include other standards (e.g. standards for programming languages or language-related facilities), specifications, data handling facilities and services, etc.

2.1 Direct conformance

An information processing entity which **conforms directly** to this International Standard shall:

- i) specify which of the datatypes and datatype generators specified in Clauses 8 and 10 are provided by the entity and which are not, and which, if any, of the declaration mechanisms in Clause 9 it provides; and
- ii) define the value spaces of the LI datatypes used by the entity to be identical to the value-spaces specified by this International Standard; and
- iii) use the notation prescribed by clauses 7 through 10 of this International Standard to refer to those datatypes and to no others; and
- iv) to the extent that the entity provides operations other than movement or translation of values, define operations on the LI datatypes which can be derived from, or are otherwise consistent with, the characterizing operations specified by this International Standard.

NOTES

1. This International Standard defines a syntax for the denotation of values of each datatype it defines, but, in general, requirement (iii) does not require conformance to that syntax. Conformance to the value-syntax for a datatype is required only in those cases in which the value appears in a *type-specifier*, that is, only where the value is part of the identification of a datatype.
2. The requirements above prohibit the use of a *type-specifier* defined in this International Standard to designate any other datatype. They make no other limitation on the definition of additional datatypes in a conforming entity, although it is recommended that either the form in Clause 8 or the form in Clause 10 be used.
3. Requirement (iv) does not require all characterizing operations to be supported and permits additional operations to be provided. The intention is to permit addition of semantic interpretation to the LI datatypes and generators, as long as it does not conflict with the interpretations given in this International Standard. A conflict arises only when a given characterizing operation could not be implemented or would not be meaningful, given the entity-provided operations on the datatype.
4. Examples of entities which could conform directly are language definitions or interface specifications whose datatypes, and the notation for them, are those defined herein. In addition, the verbatim support by a software tool or application package of the datatype syntax and definition facilities herein should not be precluded.

2.2 Indirect conformance

An information processing entity which **conforms indirectly** to this International Standard shall:

- i) provide mappings between its internal datatypes and the LI datatypes conforming to the specifications of Clause 11 of this International Standard; and
- ii) specify for which of the datatypes in Clause 8 and Clause 10 an inward mapping is provided, for which an outward mapping is provided, and for which no mapping is provided.

NOTES

1. Standards for existing programming languages are expected to provide for indirect conformance rather than direct conformance.
2. Examples of entities which could conform indirectly are language definitions and implementations, information exchange specifications and tools, software engineering tools and interface specifications, and many other entities which have a concept of datatype and an existing notation for it.

2.3 Conformance of a mapping standard

In order to conform to this International Standard, a standard for a mapping shall include in its conformance requirements the requirement to conform to this International Standard.

NOTES

1. It is envisaged that this International Standard will be accompanied by other standards specifying mappings between the internal datatypes specified in language and language-related standards and the LI datatypes. Such mapping standards are required to comply with this Interna-

tional Standard.

2. Such mapping standards may define "generic" mappings, in the sense that for a given internal datatype the standard specifies a parametrized LI datatype in which the parametric values are not derived from parametric values of the internal datatype nor specified by the standard itself, but rather are required to be specified by a "user" or "implementor" of the mapping standard. That is, instead of specifying a particular LI datatype, the mapping specifies a family of LI datatypes and requires a further user or implementor to specify which member of the family applies to a particular use of the mapping standard. This is always necessary when the internal datatypes themselves are, in the intention of the language standard, either explicitly or implicitly parametrized. For example, a programming language standard may define a datatype INTEGER with the provision that a conforming processor will implement some range of Integer; hence the mapping standard may map the internal datatype INTEGER to the LI datatype :

integer range (min..max),

and require a conforming processor to provide values for "min" and "max".

3 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 on 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 current valid International Standards.

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

ISO/IEC 8824:1990, *Information technology — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1)*.

ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*.

4 Definitions

For the purposes of this International Standard, the following definitions apply.

NOTE — These definitions may not coincide with accepted mathematical or programming language definitions of the same terms.

- 4.1 actual parametric datatype:** a datatype appearing as a parametric datatype in a use of a datatype generator, as opposed to the *formal-parametric-types* appearing in the definition of the datatype generator.
- 4.2 actual parametric value:** a value appearing as a parametric value in a reference to a datatype family or datatype generator, as opposed to the *formal-parametric-values* appearing in the corresponding definitions.
- 4.3 aggregate datatype:** a generated datatype each of whose values is made up of values of the component datatypes, in the sense that operations on all component values are meaningful.
- 4.4 annotation:** a descriptive information unit attached to a datatype, or a component of a datatype, or a procedure (value), to characterize some aspect of the representations, variables, or operations associated with values of the datatype which goes beyond the scope of this International Standard.
- 4.5 approximate:** a property of a datatype indicating that there is not a 1-to-1 relationship between values of the conceptual datatype and the values of a valid computational model of the datatype.
- 4.6 bounded:** a property of a datatype, meaning both *bounded above* and *bounded below*.
- 4.7 bounded above:** a property of a datatype indicating that there is a value U in the value space such that, for all values s in the value space, $s \leq U$.
- 4.8 bounded below:** a property of a datatype indicating that there is a value L in the value space such that, for all values s in the value space, $L \leq s$.
- 4.9 characterizing operations:**
(of a datatype): a collection of operations on, or yielding, values of the datatype, which distinguish this datatype from

other datatypes with identical value spaces;

(of a datatype generator): a collection of operations on, or yielding, values of any datatype resulting from an application of the datatype generator, which distinguish this datatype generator from other datatype generators which produce identical value spaces from identical parametric datatypes.

4.10 component datatype: a datatype which is a parametric datatype to a datatype generator, i.e. a datatype on which the datatype generator operates.

4.11 datatype: a set of distinct values, characterized by properties of those values and by operations on those values.

4.12 datatype declaration:

(1) the means provided by this International Standard for the definition of a LI datatype which is not itself defined by this International Standard;

(2) an instance of use of this means.

4.13 datatype family: a collection of datatypes which have equivalent characterizing operations and relationships, but value spaces which differ in the number and identification of the individual values.

4.14 datatype generator: an operation on datatypes, as objects distinct from their values, which generates new datatypes.

4.15 defined datatype: a datatype defined by a type-declaration.

4.16 defined generator: a datatype generator defined by a type-declaration.

4.17 exact: a property of a datatype indicating that every value of the conceptual datatype is distinct from all others in any valid computational model of the datatype.

4.18 formal-parametric-type: an identifier, appearing in the definition of a datatype generator, for which a LI datatype will be substituted in any reference to a (defined) datatype resulting from the generator.

4.19 formal-parametric-value: an identifier, appearing in the definition of a datatype family or datatype generator, for which a value will be substituted in any reference to a (defined) datatype in the family or resulting from the generator.

4.20 generated datatype: a datatype defined by the application of a datatype generator to one or more previously-defined datatypes.

4.21 generated internal datatype: a datatype defined by the application of a datatype generator defined in a particular programming language to one or more previously-defined internal datatypes.

4.22 generator: a datatype generator (q.v.).

4.23 generator declaration:

(1) the means provided by this International Standard for the definition of a datatype generator which is not itself defined by this International Standard;

(2) an instance of use of this means.

4.24 internal datatype: a datatype whose syntax and semantics are defined by some other standard, language, product, service or other information processing entity.

4.25 inward mapping: a conceptual association between the internal datatypes of a language and the LI datatypes which assigns to each LI datatype either a single semantically equivalent internal datatype or no equivalent internal datatype.

4.26 LI datatype:

(1) a datatype defined by this International Standard, or

(2) a datatype defined by the means of datatype definition provided by this International Standard.

4.27 lower bound: in a datatype which is bounded below, the value L such that, for all values s in the value space, $L \leq s$.

4.28 mapping:

(of datatypes): a formal specification of the relationship between the (internal) datatypes which are notions of, and specifiable in, a particular programming language and the (LI) datatypes specified in this International Standard;

(of values): a corresponding specification of the relationships between values of the internal datatypes and values of the LI datatypes.

4.29 order: a mathematical relationship among values (see 6.3.2).

4.30 ordered: a property of a datatype which is determined by the existence and specification of an order relationship on its value space.

4.31 outward mapping: a conceptual association between the internal datatypes of a language and the LI datatypes which identifies each internal datatype with a single semantically equivalent LI datatype.

4.32 parametric datatype: a datatype on which a datatype generator operates to produce a generated-datatype.

4.33 parametric value:

- (1) a value which distinguishes one member of a datatype family from another, or
- (2) a value which is a parameter of a datatype or datatype generator defined by a *type-declaration* (see 9.1).

4.34 primitive datatype: an identifiable datatype that cannot be decomposed into other identifiable datatypes without loss of all semantics associated with the datatype.

4.35 primitive internal datatype: a datatype in a particular programming language whose values are not viewed as being constructed in any way from values of other datatypes in the language.

4.36 representation:

(of a LI datatype): the mapping from the value space of the LI datatype to the value space of some internal datatype of a computer system, file system or communications environment;

(of a value): the image of that value in the representation of the datatype.

4.37 subtype: a datatype derived from another datatype by restricting the value space to a subset whilst maintaining all characterizing operations.

4.38 upper bound: in a datatype which is bounded above, the value U such that, for all values s in the value space, $s \leq U$.

4.39 value space: the set of values for a given datatype.

4.40 variable: a computational object to which a value of a particular datatype is associated at any given time; and to which different values of the same datatype may be associated at different times.

5 Conventions Used in this International Standard

5.1 Formal syntax

This International Standard defines a formal datatype specification language. The following notation, derived from Backus-Naur form, is used in defining that language. In this clause, the word *mark* is used to refer to the characters used to define the syntax, while the word *character* is used to refer to the characters used in the actual datatype specification language. Table 5-1 summarizes the syntactic metanotation.

Table 5-1 — Metanotation Marks

"	(QUOTATION MARK)	delimits a terminal symbol
'	(APOSTROPHE)	delimits a terminal symbol
{ }	(CURLY BRACKETS)	delimit a repeated sequence (zero or more occurrences)
[]	(SQUARE BRACKETS)	delimit an optional sequence (zero or one occurrence)
	(VERTICAL LINE)	delimits an alternative sequence
=	(EQUALS SIGN)	separates a non-terminal symbol from its definition
.	(FULL STOP)	terminates a production

A **terminal symbol** is a sequence of marks beginning with either a QUOTATION MARK (") or an APOSTROPHE mark (') and terminated by the next occurrence of the same mark. The terminal symbol represents the occurrence of the sequence of characters in an implementation character-set corresponding to the marks enclosed by (but not including) the QUOTATION MARK or APOSTROPHE delimiters.

A **non-terminal symbol** is a sequence of marks, each of which is either a letter or the HYPHEN-MINUS (-) mark, terminated by the first mark which is neither a letter nor a HYPHEN-MINUS. A non-terminal symbol represents any sequence of terminal symbols which satisfies the *production* for that non-terminal symbol. For each non-terminal symbol there is exactly one production in clauses 7, 8, 9, and 10.

A **sequence** of symbols represents exactly one occurrence of a (group of) terminal symbol(s) represented by each symbol in the sequence in the order in which the symbols appear in the sequence, and no other symbols.

A **repeated sequence** is a sequence of terminal and/or non-terminal symbols enclosed between a LEFT CURLY BRACKET mark ({) and a RIGHT CURLY BRACKET mark (}). A repeated sequence represents any number of consecutive occurrences of the sequence of symbols so enclosed, including no occurrence.

An **optional sequence** is a sequence of terminal and/or non-terminal symbols enclosed between a LEFT SQUARE BRACKET mark ([) and a RIGHT SQUARE BRACKET mark (]). An optional sequence represents either exactly one occurrence of the sequence of symbols so enclosed or no symbols at all.

An **alternative sequence** is a sequence of terminal and/or non-terminal symbols preceded by a VERTICAL LINE (|) mark and followed by either a VERTICAL LINE mark or a FULL STOP mark (.). An alternative sequence represents the occurrence of either the sequence of symbols so delimited or the sequence of symbols preceding the (first) VERTICAL LINE mark.

A **production** defines the valid sequences of symbols which a non-terminal symbol represents. A production has the form:
non-terminal-symbol = valid-sequence .

where *valid-sequence* is any sequence of terminal symbols, non-terminal symbols, optional sequences, repeated sequences and alternative sequences. The EQUALS SIGN (=) mark separates the non-terminal symbol being defined from the valid-sequence which represents its definition. The FULL STOP mark terminates the valid-sequence.

5.2 Text conventions

Within the text:

- A reference to a terminal symbol syntactic object consists of the terminal symbol in quotation marks, e.g. "type".
- A reference to a non-terminal symbol syntactic object consists of the non-terminal-symbol in italic script, e.g. *type-declaration*.
- Non-italicized words which are identical or nearly identical in spelling to a non-terminal-symbol refer to the conceptual object represented by the syntactic object. In particular, *xxx-type* refers to the syntactic representation of an "xxx datatype" in all occurrences.

6 Fundamental Notions

6.1 Datatype

A **datatype** is a set of distinct values, characterized by properties of those values and by operations on those values. Characterizing operations are included in this International Standard solely in order to identify the datatype. In this International Standard, characterizing operations are purely informative and have no normative impact.

NOTE — Characterizing operations are included in order to assist in the identification of the appropriate datatypes for particular purposes, such as mapping to programming languages.

The term **LI datatype** (for Language-Independent datatype) is used to mean a datatype defined by this International Standard. **LI datatypes** (plural) refers to some or all of the datatypes defined by this International Standard.

The term **internal datatype** is used to mean a datatype whose syntax and semantics are defined by some other standard, language, product, service or other information processing entity.

NOTE — The datatypes included in this standard are "common", not in the sense that they are directly supported by, i.e. "built-in" to, many languages, but in the sense that they are common and useful generic concepts among users of datatypes, which include, but go well beyond, programming languages.

6.2 Value space

A **value space** is the collection of values for a given datatype. The value space of a given datatype can be defined in one of the following ways:

- enumerated outright, or
- defined axiomatically from fundamental notions, or
- defined as the subset of those values from some already defined value space which have a given set of properties, or
- defined as a combination of arbitrary values from some already defined value spaces by a specified construction procedure.

Every distinct value belongs to exactly one datatype, although it may belong to many subtypes of that datatype (see 8.2).

6.3 Datatype properties

The model of datatypes used in this International Standard is said to be an "abstract computational model". It is "computational" in the sense that it deals with the manipulation of information by computer systems and makes distinctions in the typing of information units which are appropriate to that kind of manipulation. It is "abstract" in the sense that it deals with the perceived properties of the information units themselves, rather than with the properties of their representations in computer systems.

NOTES

1. It is important to differentiate between the values, relationships and operations for a datatype and the representations of those values, relationships and operations in computer systems. This International Standard specifies the characteristics of the conceptual datatypes, but it only provides a means for specification of characteristics of representations of the datatypes.
2. Some computational properties derive from the *need for the information units to be representable* in computers. Such properties are deemed to be appropriate to the abstract computational model, as opposed to purely *representational* properties, which derive from the *nature of specific representations of the information units*.
3. It is not proper to describe the datatype model used herein as "mathematical", because a truly mathematical model has no notions of "access to information units" or "invocation of processing elements", and these notions are important to the definition of characterizing operations for datatypes and datatype generators.

6.3.1 Equality

In every value space there is a notion of **equality**, for which the following rules hold:

- for any two instances (a, b) of values from the value space, either *a is equal to b*, denoted $a = b$, or *a is not equal to b*, denoted $a \neq b$;
- there is no pair of instances (a, b) of values from the value space such that both $a = b$ and $a \neq b$;
- for every value a from the value space, $a = a$;
- for any two instances (a, b) of values from the value space, $a = b$ if and only if $b = a$;
- for any three instances (a, b, c) of values from the value space, if $a = b$ and $b = c$, then $a = c$.

On every datatype, the operation Equal is defined in terms of the equality property of the value space, by:

- for any values a, b drawn from the value space, Equal(a,b) is *true* if $a = b$, and *false* otherwise.

6.3.2 Order

A value space is said to be **ordered** if there exists for the value space an **order** relation, denoted $\leq$, with the following rules:

- for every pair of values (a, b) from the value space, either $a \leq b$ or $b \leq a$, or both;
- for any two values (a, b) from the value space, if $a \leq b$ and $b \leq a$, then $a = b$;
- for any three values (a, b, c) from the value space, if $a \leq b$ and $b \leq c$, then $a \leq c$.

For convenience, the notation $a < b$ is used herein to denote the simultaneous relationships: $a \leq b$ and $a \neq b$.

A datatype is said to be **ordered** if an order relation is defined on its value space. A corresponding characterizing operation, called *InOrder*, is then defined by:

- for any two values (a, b) from the value space, *InOrder* (a, b) is *true* if $a \leq b$, and *false* otherwise.

NOTE — There may be several possible orderings of a given value space. And there may be several different datatypes which have a common value space, each using a different order relationship. The chosen order relationship is a characteristic of an ordered datatype and may affect the definition of other operations on the datatype.

6.3.3 Bound

A datatype is said to be **bounded above** if it is ordered and there is a value U in the value space such that, for all values s in the value space, $s \leq U$. The value U is then said to be an **upper bound** of the value space. Similarly, a datatype is said to be **bounded below** if it is ordered and there is a value L in the space such that, for all values s in the value space, $L \leq s$. The value L is then said to be a **lower bound** of the value space. A datatype is said to be **bounded** if its value space has both an upper bound and a lower bound.

NOTE — The upper bound of a value space, if it exists, must be unique under the equality relationship. For if U_1 and U_2 are both upper bounds of the value space, then $U_1 \leq U_2$ and $U_2 \leq U_1$, and therefore $U_1 = U_2$, following the second rule for the order relationship. And similarly the lower bound, if it exists, must also be unique.

On every datatype which is bounded below, the niladic operation *Lowerbound* is defined to yield that value which is the lower bound of the value space, and, on every datatype which is bounded above the niladic operation *Upperbound* is defined to yield that value which is the upper bound of the value space.

6.3.4 Cardinality

A value space has the mathematical concept of cardinality: it may be finite, denumerably infinite (countable), or non-denumerably infinite (uncountable). A datatype is said to have the cardinality of its value space. In the computational model, there are three significant cases:

- datatypes whose value spaces are finite,
- datatypes whose value spaces are exact (see 6.3.5) and denumerably infinite,
- datatypes whose value spaces are approximate (see 6.3.5), and therefore have a finite or denumerably infinite computational model, although the conceptual value space may be non-denumerably infinite.

Every conceptually finite datatype is necessarily exact. No computational datatype is non-denumerably infinite.

NOTE — For a denumerably infinite value space, there always exist representation algorithms such that no two distinct values have the same representation and the representation of any given value is of finite length. Conversely, in a non-denumerably infinite value space there always exist values which do not have finite representations.

6.3.5 Exact and approximate

The computational model of a datatype may limit the degree to which values of the datatype can be distinguished. If every value in the value space of the conceptual datatype is distinguishable in the computational model from every other value in the value space, then the datatype is said to be **exact**.

Certain mathematical datatypes having values which do not have finite representations are said to be **approximate**, in the following sense:

Let M be the mathematical datatype and C be the corresponding computational datatype, and let P be the mapping from the value space of M to the value space of C . Then for every value v' in C , there is a corresponding value v in M and a real value h such that $P(x) = v'$ for all x in M such that $|v - x| < h$. That is, v' is the approximation in C to all values in M which are "within distance h of value v ". Furthermore, for at least one value v' in C , there is more than one value y in M such that $P(y) = v'$. And thus C is *not* an exact model of M .

In this International Standard, all approximate datatypes have computational models which specify, via parametric values, a **degree of approximation**, that is, they require a certain minimum set of values of the mathematical datatype to be distinguishable in the computational datatype.

NOTE — The computational model described above allows a mathematically dense datatype to be mapped to a datatype with fixed-length representations and nonetheless evince intuitively acceptable mathematical behavior. When the real value h described above is constant over the

value space, the computational model is characterized as having "bounded absolute error" and the result is a scaled datatype (8.1.9). When h has the form $c \cdot |v|$, where c is constant over the value space, the computational model is characterized as having "bounded relative error", which is the model used for the Real (8.1.10) and Complex (8.1.11) datatypes.

6.3.6 Numeric

A datatype is said to be **numeric** if its values are conceptually quantities (in some mathematical number system). A datatype whose values do not have this property is said to be **non-numeric**.

NOTE — The significance of the numeric property is that the representations of the values depend on some *radix*, but can be algorithmically transformed from one radix to another.

6.4 Primitive and non-primitive datatypes

In this International Standard, datatypes are categorized, for syntactic convenience, into:

- **primitive** datatypes, which are defined ab initio without reference to other datatypes, and
- **generated** datatypes, which are specified, and partly defined, in terms of other datatypes.

In addition, this International Standard identifies structural and abstract notions of datatypes. The structural notion of a datatype characterizes the datatype as either:

- conceptually **atomic**, having values which are intrinsically indivisible, or
- conceptually **aggregate**, having values which can be seen as an organization of specific component datatypes with specific functionalities.

All primitive datatypes are conceptually atomic, and therefore have, and are defined in terms of, well-defined abstract notions. Some generated datatypes are conceptually atomic but are dependent on specifications which involve other datatypes. These too are defined in terms of their abstract notions. Many other datatypes may represent objects which are conceptually atomic, but are themselves conceptually aggregates, being organized collections of accessible component values. For aggregate datatypes, this International Standard defines a set of basic structural notions (see 6.8) which can be recursively applied to produce the value space of a given generated datatype. The only abstract semantics assigned to such a datatype by this International Standard are those which characterize the aggregate value structure itself.

NOTE — The abstract notion of a datatype is the semantics of the values of the datatype itself, as opposed to its utilization to represent values of a particular information unit or a particular abstract object. The abstract and structural notions provided by this International Standard are sufficient to define its role in the universe of discourse between two languages, but *not* to define its role in the universe of discourse between two *programs*. For example, Array datatypes are supported as such by both Fortran and Pascal, so that Array of Real has sufficient semantics for procedure calls between the two languages. By comparison, both linear operators and lists of Cartesian points may be represented by Array of Real, and Array of Real is insufficient to distinguish those meanings in the programs.

6.5 Datatype generator

A **datatype generator** is a conceptual operation on one or more datatypes which yields a datatype. A datatype generator operates on datatypes to generate a datatype, rather than on values to generate a value. Specifically, a datatype generator is the combination of:

- a collection of criteria for the number and characteristics of the datatypes to be operated upon,
- a construction procedure which, given a collection of datatypes meeting those criteria, creates a new value space from the value spaces of those datatypes, and
- a collection of characterizing operations which attach to the resulting value space to complete the definition of a new datatype.

The application of a datatype generator to a specific collection of datatypes meeting the criteria for the datatype generator forms a **generated datatype**. The generated datatype is sometimes called the **resulting** datatype, and the collection of datatypes to which the datatype generator was applied are called its **parametric datatypes**.

6.6 Characterizing operations

The set of **characterizing operations for a datatype** comprises those operations on or yielding values of the datatype which distinguish this datatype from other datatypes having value spaces which are identical except possibly for substitution of symbols.

The set of **characterizing operations for a datatype generator** comprises those operations on or yielding values of any datatype resulting from an application of the datatype generator which distinguish this datatype generator from other datatype generators which produce identical value spaces from identical parametric datatypes.

NOTES

1. Characterizing operations are needed to distinguish datatypes whose value spaces differ only in what the values are called. For example, the value spaces (one, two, three, four), (1, 2, 3, 4), and (red, yellow, green, blue) all have four distinct values and all the names (symbols) are different. But one can claim that the first two support the characterizing operation Add, while the last does not:

Add(one, two) = three; and Add(1,2) = 3; but Add(red, yellow) ≠ green.

It is this characterizing operation (Add) which enables one to recognize that the first two datatypes are the same datatype, while the last is a different datatype.

2. The characterizing operations for an aggregate datatype are compositions of characterizing operations for its datatype generator with characterizing operations for its component datatypes. Such operations are, of course, only sufficient to identify the datatype as a structure.

3. The characterizing operations on a datatype may be:

- a) niladic operations which yield values of the given datatype,
- b) monadic operations which map a value of the given datatype into a value of the given datatype or into a value of datatype Boolean,
- c) dyadic operations which map a pair of values of the given datatype into a value of the given datatype or into a value of datatype Boolean,
- d) n-adic operations which map ordered n-tuples of values, each of which is of a specified datatype, which may be the given datatype or a parametric datatype, into values of the given datatype or a parametric datatype.

4. In general, there is no unique collection of characterizing operations for a given datatype. This International Standard specifies one collection of characterizing operations for each datatype (or datatype generator) which is sufficient to distinguish the (resulting) datatype from all other datatypes with value spaces of the same cardinality. While some effort has been made to minimize the collection of characterizing operations for each datatype, no assertion is made that any of the specified collections is minimal.

5. InOrder is always a characterizing operation on ordered datatypes (see 6.3.2).

6.7 Datatype families

If there is a one-to-one symbol substitution which maps the entire value space of one datatype (the **domain**) into a subset of the value space of another datatype (the **range**) in such a way that the value relationships and characterizing operations of the domain datatype are preserved in the corresponding value relationships and characterizing operations of the range datatype, and if there are no additional characterizing operations on the range datatype, then the two datatypes are said to belong to the same **family of datatypes**. An individual member of a family of datatypes is distinguished by the symbol set making up its value space. In this International Standard, the symbol set for an individual member of a datatype family is specified by one or more values, called the **parametric values** of the datatype family.

6.8 Aggregate datatypes

An **aggregate datatype** is a generated datatype, each of whose values is, in principle, made up of values of the parametric datatypes. The parametric datatypes of an aggregate datatype or its generator are also called **component datatypes**. An aggregate datatype generator generates a datatype by

- applying an algorithmic procedure to the value spaces of its component datatypes to yield the value space of the aggregate datatype, and
- providing a set of characterizing operations specific to the generator.

Unlike other generated datatypes, it is characteristic of aggregate datatypes that the component values of an aggregate value are accessible through characterizing operations.

Aggregate datatypes of various kinds are distinguished one from another by properties which characterize relationships among the component datatypes and relationships between each component and the aggregate value. This subclause defines those properties.

The properties specific to an aggregate are independent of the properties of the component datatypes. (The fundamental properties of arrays, for example, do not depend on the nature of the elements.) In principle, any combination of the properties specified in this subclause defines a particular form of aggregate datatype, although most are only meaningful for homogeneous aggregates (see 6.8.1) and there are implications of some direct access methods (see 6.8.5).

6.8.1 Homogeneity

An aggregate datatype is **homogeneous**, if and only if all components must belong to a single datatype. If different components may belong to different datatypes, the aggregate datatype is said to be **heterogeneous**. The component datatype of a homogeneous aggregate is also called the **element datatype**.

NOTES

1. Homogeneous aggregates view all their elements as serving the same role or purpose. Heterogeneous aggregates divide their elements into different roles.
2. The aggregate datatype is homogeneous if its components all belong to the same datatype, even if the element datatype is itself an heterogeneous aggregate datatype. Consider the datatype `label_list` defined by:
`type label = choice (state(name, handle)) of ((name): characterstring, (handle): integer);`
`type label_list = sequence of (label);`

Formally, a `label_list` value is a homogeneous series of `label` values. One could argue that it is really a series of heterogeneous values, because every `label` value is of a choice datatype (see 8.3.1). Choice is clearly heterogeneous because it is *capable of introducing variation* in element type. But Sequence (see 8.4.4) is homogeneous because it itself *introduces no variation* in element type.

6.8.2 Size

The **size** of an aggregate-value is the number of component values it contains. The size of the aggregate datatype is **fixed**, if and only if all values in its value space contain the same number of component values. The size is **variable**, if different values of the aggregate datatype may have different numbers of component values. Variability is the more general case; fixed-size is a constraint.

6.8.3 Uniqueness

An aggregate-value has the **uniqueness** property if and only if no value of the element datatype occurs more than once in the aggregate-value. The aggregate datatype has the uniqueness property, if and only if all values in its value space do.

6.8.4 (Aggregate-imposed) ordering

An aggregate datatype has the **ordering** property, if and only if there is a canonical first element of each non-empty value in its value-space. This ordering is (externally) imposed by the aggregate value, as distinct from the value-space of the element datatype itself being (internally) **ordered** (see 6.3.2). It is also distinct from the value-space of the aggregate datatype being **ordered**.

EXAMPLE — The type-generator `sequence` has the ordering property. The datatype `characterstring` is defined as `sequence of (character(repertoire))`. The ordering property of `sequence` means that in every value of type `characterstring`, there is a first character value. For example, the first element value of the `characterstring` value “computation” is ‘c’. This is different from the question of whether the element datatype `character(repertoire)` is ordered: is ‘a’ < ‘c’? It is also different from the question of whether the value space of datatype `characterstring` is ordered by some collating-sequence: is “computation” < “Computer”?

6.8.5 Access method

The **access method** for an aggregate datatype is the property which determines how component values can be extracted from a given aggregate-value.

An aggregate datatype has a **direct access method**, if and only if there is an aggregate-imposed mapping between values of one or more “index” (or “key”) datatypes and the component values of each aggregate value. Such a mapping is required to be single-valued, i.e. there is at most one element of each aggregate value which corresponds to each (composite) value of the index datatype(s). The **dimension** of an aggregate datatype is the number of index or key datatypes the aggregate has.

An aggregate datatype is said to be **indexed**, if and only if it has a direct access method, every index datatype is ordered, and an element of the aggregate value is actually present and defined for every (composite) value in the value space of the index datatype(s). Every indexed aggregate datatype has a fixed size, because of the 1-to-1 mapping from the index value space. In addition, an indexed datatype has a “partial ordering” in each dimension imposed by the order relationship on the index datatype for that dimension; in particular, an aggregate datatype with a single ordered index datatype implicitly has the **ordering** imposed by sequential indexing.

An aggregate datatype is said to be **keyed**, if and only if it has a direct access method, but either the index datatypes or the mapping do not meet the requirements for **indexed**. That is, the “index” (or “key”) datatypes need not be ordered, and a value of the

aggregate datatype need not have elements corresponding to all of the key values.

An aggregate datatype is said to have only **indirect access methods** if there is no aggregate-imposed index mapping. Indirect access may be by position (if the aggregate datatype has **ordering**), by value of the element (if the aggregate datatype has **unique-ness**), or by some implementation-dependent selection mechanism, modelled as random selection.

NOTES

1. The access methods become characterizing operations on the aggregate types. It is preferable to define the types by their intrinsic properties and to see these access properties be derivable characterizing operations.
2. Sequence (see 8.4.4) is said to have *indirect access* because the only way a given element value (or an element value satisfying some given condition) can be found is to traverse the list in order until the desired element is the “Head”. In general, therefore, one cannot access the desired element without first accessing all (undesired) elements appearing earlier in the sequence. On the other hand, Array (see 8.4.5) has *direct access* because the access operation for a given element is “find the element whose index is i” – the ith element can be accessed without accessing any other element in the given Array. Of course, if the Array element which satisfies a condition not related to the index value is wanted, access would be indirect.

6.8.6 Recursive structure

A datatype is said to be **recursive** if a value of the datatype can contain (or refer to) another value of the datatype. In this International Standard, recursivity is supported by the type-declaration facility (see 9.1), and recursive datatypes can be described using type-declaration in combination with choice datatypes (8.3.1) or pointer datatypes (8.3.2). Thus recursive structure is *not* considered to be a property of aggregate datatypes per se.

EXAMPLE — LISP has several "atomic" datatypes, collected under the generic datatype "atom", and a "list" datatype which is a sequence of elements each of which can be an atom or a list. This datatype can be described using the Tree datatype generator defined in 10.2.2.

7 Elements of the Datatype Specification Language

This International Standard defines a datatype specification language, in order to formalize the identification and declaration of datatypes conforming to this International Standard. The language is a subset of the Interface Definition Notation defined in ISO/IEC 13886:1996, *Information technology — Programming languages — Language-independent procedure calling*, which is completely specified in Annex D. This clause defines the basic syntactic objects used in that language.

7.1 IDN character-set

The following productions define the character-set of the datatype specification language, summarized in Table 7-1.

Table 7-1 — IDN Character Set

Syntax	Characters
letter	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
digit	0 1 2 3 4 5 6 7 8 9
special	() (parentheses) . (full stop) , (comma) : (colon) ; (semicolon) - (hyphen minus) { } / * ^ = [] (curly brackets) (solidus) (asterisk) (circumflex) (equals sign) (square brackets)
underscore	_ (low line)
apostrophe	' (apostrophe)
quote	" (quotation mark)
escape	! (exclamation mark)
space	

```

letter =      "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" .
digit =      "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
special =    "(" | ")" | "." | "," | ":" | ";" | "=" | "/" | "*" | "-" | "{" | "}" | "[" | "]" .
underscore = "_" .
apostrophe = "'" .
quote =      '"' .
escape =     "\"" .
space =      " " .
non-quote-character = letter | digit | underscore | special | apostrophe | space .
bound-character = non-quote-character | quote .
added-character = not defined by this International Standard .

```

These productions are nominal. Lexical productions are always subject to minor changes from implementation to implementation, in order to handle the vagaries of available character-sets. The following rules, however, always apply:

- 1) The *bound-characters*, and the *escape* character, are required in any implementation to be associated with particular members of the implementation character set.
- 2) The character *space* is required to be bound to the "space" member of ISO/IEC 10646-1: 1993, but it only has meaning within character-literals and string-literals.
- 3) A *bound-character* is required to be associated with the member having the corresponding symbol, if any, in any implementation character-set derived from ISO/IEC 10646-1:1993, except that no significance is attached to the "case" of letters.
- 4) An *added-character* is any other member of the implementation character-set which is bound to the member having the corresponding symbol in an ISO/IEC 10646-1 character-set.

7.2 Whitespace

A sequence of one or more *space* characters, except within a character-literal or string-literal (see 7.3), shall be considered *whitespace*. Any use of this International Standard may define any other characters or sequences of characters not in the above character set to be whitespace as well, such as horizontal and vertical tabulators, end of line and end of page indicators, etc.

A *comment* is any sequence of characters beginning with the sequence *"/** and terminating with the first occurrence thereafter of the sequence **/*. Every character of a comment shall be considered whitespace.

With respect to interpretation of a syntactic object under this International Standard, any annotation (see 7.4) is considered whitespace.

Any two lexical objects which occur consecutively may be separated by whitespace, without effect on the interpretation of the syntactic construction. Whitespace shall not appear *within* lexical objects.

Any two consecutive keywords or identifiers, or a keyword preceded or followed by an identifier, shall be separated by whitespace.

7.3 Lexical objects

The lexical objects are all terminal symbols except those defined in 7.1, and the objects *identifier*, *digit-string*, *character-literal*, *string-literal*.

7.3.1 Identifiers

An *identifier* is a terminal symbol used to name a datatype or datatype generator, a component of a generated datatype, or a value of some datatype.

```

identifier = letter { pseudo-letter } .
pseudo-letter = letter | digit | underscore .

```

Table 7-2 — Reserved Keywords

array	choice	default	excluding	from	in	inout
new	of	out	plus	pointer	procedure	raises
range	record	returns	selecting	size	subtype	table
termination	to	type	value			

Multiple identifiers with the same spelling are permitted, as long as the object to which the identifier refers can be determined by the following rules:

- 1) An identifier X declared by a *type-declaration* or *value-declaration* shall not be declared in any other declaration.
- 2) The identifier X in a component of a *type-specifier* (Y) refers to that component of Y which Y declares X to identify, if any, or whatever X refers to in the *type-specifier* which immediately contains Y, if any, or else the datatype or value which X is declared to identify by a declaration.

7.3.2 Digit-string

A **digit-string** is a terminal-symbol consisting entirely of digits. It is used to designate a value of some datatype, with the interpretation specified by that datatype definition.

digit-string = digit { digit } .

7.3.3 Character-literal and string-literal

A **character-literal** is a terminal-symbol delimited by *apostrophe* characters. It is used to designate a value of a character datatype, as specified in 8.1.4.

character-literal = `''` any-character `''` .
any-character = bound-character | added-character | escape-character .
escape-character = escape character-name escape .
character-name = identifier { `"` identifier `"` } .

A **string-literal** is a terminal-symbol delimited by *quote* characters. It is used to designate values of time datatypes (8.1.6), bitstring datatypes (10.1.4), and characterstring datatypes (10.1.5), with the interpretation specified for each of those datatypes.

string-literal = quote { string-character } quote .
string-character = non-quote-character | added-character | escape-character .

Every character appearing in a *character-literal* or *string-literal* shall be a part of the literal, even when that character would otherwise be whitespace.

7.3.4 Keywords

The term **keyword** refers to any terminal symbol which also satisfies the production for *identifier*, i.e. is not composed of special characters. The keywords appearing in Table 7-2 are "reserved", in the sense that none of them shall be interpreted as an identifier. All other keywords appearing in this International Standard shall be interpreted as predefined identifiers for the datatype or type-generator to which this International Standard defines them to refer.

NOTE — All of the above keywords are reserved because they introduce (or are part of) syntax which cannot validly follow an *identifier* for a datatype or type-generator. Most datatype identifiers defined in Clause 8 are syntactically equivalent to a *type-reference* (see 8.5), except for their appearance in Clause 8.

7.4 Annotations

An *annotation*, or *extension*, is a syntactic object defined by a standard or information processing entity which uses this International Standard. All annotations shall have the form:

annotation = `"["` annotation-label `":"` annotation-text `"]"` .

annotation-label = objectidentifiercomponent-list .
 annotation-text = *not defined by this International Standard* .

The *annotation-label* shall identify the standard or information processing entity which defines the meaning of the *annotation-text*. The entity identified by the *annotation-label* shall also define the allowable syntactic placement of a given type of annotation and the syntactic object(s), if any, to which the annotation applies. The *objectidentifiercomponent-list* shall have the structure and meaning prescribed by clause 10.1.10.

NOTE — Of the several forms of *objectidentifiercomponent-value* specified in 10.1.10, the *nameform* is the most convenient for labelling annotations. Following ISO/IEC 8824:1990, every value of the objectidentifier datatype must have as its first component one of "iso", "ccitt", or "joint-iso-ccitt", but an implementation or use is permitted to specify an identifier which represents a sequence of component values beginning with one of the above, as:

value rpc : objectidentifier = { iso(1) standard(0) 11578 };

and that identifier may then be used as the first (or only) component of an *annotation-label*, as in:

[rpc: discriminant = n].

(This example is fictitious. ISO/IEC 11578:1995 does not define any annotations.)

Non-standard annotations, defined by vendors or user organizations, for example, can acquire such labels through one of the { iso member-body <nation> ... } or { iso identified-organization <organization> ... } paths, using the appropriate national or international registration authority.

7.5 Values

The identification of members of a datatype family, subtypes of a datatype, and the resulting datatypes of datatype generators may require the syntactic designation of specific values of a datatype. For this reason, this International Standard provides a notation for values of every datatype that is defined herein or can be defined using the features provided by clause 10, except for datatypes for which designation of specific values is not appropriate.

A *value-expression* designates a value of a datatype. Syntax:

value-expression = independent-value | dependent-value | formal-parametric-value .

An *independent-value* is a syntactic construction which resolves to a fixed value of some LI datatype. A *dependent-value* is a syntactic construction which refers to the value possessed by another component of the same datatype. A *formal-parametric-value* refers to the value of a *formal-type-parameter* in a *type-declaration*, as provided in 9.1.

7.5.1 Independent values

An *independent-value* designates a specific fixed value of a datatype. Syntax:

independent-value = explicit-value | value-reference .

explicit-value = boolean-literal | state-literal | enumerated-literal | character-literal
 | ordinal-literal | time-literal | integer-literal | rational-literal
 | scaled-literal | real-literal | complex-literal | void-literal
 | extended-literal | pointer-literal | procedure-reference | string-literal
 | bitstring-literal | objectidentifier-value | choice-value | record-value
 | set-value | sequence-value | bag-value | array-value | table-value .

value-reference = value-identifier .

procedure-reference = procedure-identifier .

An *explicit-value* uses an explicit syntax for values of the datatype, as defined in clauses 8 and 10. A *value-reference* designates the value associated with the *value-identifier* by a *value-declaration*, as provided in 9.2. A *procedure-reference* designates the value of a procedure datatype associated with a *procedure-identifier*, as described in 8.3.3.

NOTES

- Two syntactically different *explicit-values* may designate the same value, such as *rational-literals* 3/4 and 6/8, or set of (integer) values (1,3,4) and (4,3,1).
- The same *explicit-value* syntax may designate values of two different datatypes, as 19940101 can be an Integer value, or an Ordinal value. In general, the syntax requires that the intended datatype of a *value-expression* can be determined from context when the *value-expression* is encountered.

3. The IDN productions for *value-reference* and *procedure-reference* appearing in Annex D are more general. The above productions are sufficient for the purposes of this International Standard.

7.5.2 Dependent values

When a parameterized datatype appears within a procedure parameter (see 8.3.3) or a record datatype (see 8.4.1), it is possible to specify that the parametric value is always identical to the value of another parameter to the procedure or another component within the record. Such a value is referred to as a **dependent-value**. Syntax:

```
dependent-value = primary-dependency { "." component-reference } .
primary-dependency = field-identifier | parameter-name .
component-reference = field-identifier | "*" .
```

A *type-specifier* *x* is said to **involve** a *dependent-value* if *x* contains the *dependent-value* and no component of *x* contains the *dependent-value*. Thus, exactly one *type-specifier* involves a given *dependent-value*. A *type-specifier* which involves a *dependent-value* is said to be a **data-dependent type**. Every data-dependent type shall be the datatype of a component of some generated datatype.

The *primary-dependency* shall be the identifier of a (different) component of a procedure or record datatype which (also) contains the data-dependent type. The component so identified will be referred to in the following as the **primary component**; the generated datatype of which it is a component will be referred to as the **subject datatype**. That is, the subject datatype shall have an immediate component to which the *primary-dependency* refers, and a different immediate component which, *at some level*, contains the data-dependent type.

When the subject datatype is a procedure datatype, the *primary-dependency* shall be a *parameter-name* and shall identify a parameter of the subject datatype. If the *direction* of the parameter (component) which contains the data-dependent type is "in" or "inout", then the *direction* of the parameter designated by the *primary-dependency* shall also be "in" or "inout". If the parameter which contains the data-dependent type is the *return-parameter* or has *direction* "out", then the *primary-dependency* may designate any parameter in the *parameter-list*. If the parameter which contains the data-dependent type is a *termination* parameter, then the *primary-dependency* shall designate another parameter in the same *termination-parameter-list*.

When the subject datatype is a record datatype, the *primary-dependency* shall be a *field-identifier* and shall identify a field of the subject datatype.

When the *dependent-value* contains no *component-references*, it refers to the value of the primary component. Otherwise, the primary component shall be considered the "0th *component-reference*", and the following rules shall apply:

- 1) If the *n*th *component-reference* is the last *component-reference* of the *dependent-value*, the *dependent-value* shall refer to the value to which the *n*th *component-reference* refers.
- 2) If the *n*th *component-reference* is not the last *component-reference*, then the datatype of the *n*th *component-reference* shall be a record datatype or a pointer datatype.
- 3) If the *n*th *component-reference* is not the last *component-reference*, and the datatype of the *n*th *component-reference* is a record datatype, then the (*n*+1)th *component-reference* shall be a *field-identifier* which identifies a field of that record datatype; and the (*n*+1)th *component-reference* shall refer to the value of that field of the value referred to by the *n*th *component-reference*.
- 4) If the *n*th *component-reference* is not the last *component-reference*, and the datatype of the *n*th *component-reference* is a pointer datatype, then the (*n*+1)th *component-reference* shall be "*"; and the (*n*+1)th *component-reference* shall refer to the value resulting from Dereference applied to the value referred to by the *n*th *component-reference*.

NOTES

1. The datatype which involves a *dependent-value* must be a component of some generated datatype, but that generated datatype may itself be a component of another generated datatype, and so on. The subject datatype may be several levels up this hierarchy.
2. The primary component, and thus the subject datatype, cannot be ambiguous, even when the *primary-dependency* identifier appears more than once in such a hierarchy, according to the scope rules specified in 7.3.1.
3. In the same wise, an identifier which may be either a *value-identifier* or a *dependent-value* can be resolved by application of the same scope rules. If the identifier *X* is found to have a "declaration" anywhere within the outermost *type-specifier* which contains the reference to *X*, then that declaration is used. If no such declaration is found, then a declaration of *X* in a "global" context, e.g. as a *value-identifier*, applies.

8 Datatypes

This clause defines the collection of LI datatypes. A LI datatype is either:

- a datatype defined in this clause, or
- a datatype defined by a datatype declaration, as defined in 9.1.

Since this collection is unbounded, there are four formal methods used in the definition of the datatypes:

- explicit specification of **primitive** datatypes, which have universal well-defined abstract notions, each independent of any other datatype.
- implicit specification of **generated** datatypes, which are syntactically and in some ways semantically dependent on other datatypes used in their specification. Generated datatypes are specified implicitly by means of explicit specification of datatype generators, which themselves embody independent abstract notions.
- specification of the means of **datatype declaration**, which permits the association of additional identifiers and refinements to primitive and generated datatypes and to datatype generators.
- specification of the means of defining **subtypes** of the datatypes defined by any of the foregoing methods.

A reference to a LI datatype is a *type-specifier*, with the following syntax:

type-specifier = primitive-type | subtype | generated-type | type-reference | formal-parametric-type .

A *type-specifier* shall not be a *formal-parametric-type*, except in some cases in *type-declarations*, as provided by clause 9.1.3.

This clause also provides syntax for the identification of values of LI datatypes. Notations for values of datatypes are required in the syntactic designations for subtypes and for some primitive datatypes.

NOTES

1. For convenience, or correctness, some datatypes and characterizing operations are defined in terms of other LI datatypes. The use of a LI datatype defined in this clause always refers to the datatype so defined.
2. The names used in this International Standard to identify the datatypes are derived in many cases from common programming language usage, but nevertheless do not necessarily correspond to the names of equivalent datatypes in actual languages. The same applies to the names and symbols for the operations associated with the datatypes, and to the syntax for values of the datatypes.

8.1 Primitive datatypes

A datatype whose value space is defined either axiomatically or by enumeration is said to be a **primitive datatype**. All primitive LI datatypes shall be defined by this International Standard.

primitive-type = boolean-type | state-type | enumerated-type | character-type
 | ordinal-type | time-type | integer-type | rational-type
 | scaled-type | real-type | complex-type | void-type .

Each primitive datatype, or datatype family, is defined by a separate subclause. The title of each such subclause gives the informal name for the datatype, and the datatype is defined by a single occurrence of the following template:

Description:	prose description of the conceptual datatype.
Syntax:	the syntactic productions for the type-specifier for the datatype.
Parametric values:	identification of any parametric values which are necessary for the complete identification of a distinct member of a datatype family.
Values:	enumerated or axiomatic definition of the value space.
Value-syntax:	the syntactic productions for denotation of a value of the datatype, and the identification of the value denoted.
Properties:	properties of the datatype which indicate its admissibility as a component datatype of certain datatype generators: numeric or non-numeric, approximate or exact, unordered or ordered and, if ordered, bounded or unbounded.

Operations: definitions of characterizing operations.

The definition of an operation herein has one of the forms:

operation-name (parameters) : result-datatype = formal-definition; or

operation-name (parameters) : result-datatype is prose-definition.

In either case, "parameters" may be empty, or be a list, separated by commas, of one or more formal parameters of the operation in the form:

parameter-name : parameter-datatype, or

parameter-name₁ , parameter-name₂ : parameter-datatype.

The *operation-name* is an identifier unique only within the datatype being defined. The *parameter-names* are formal identifiers appearing in the *formal-* or *prose-definition*. Each is understood to represent an arbitrary value of the datatype designated by *parameter-datatype*, and all occurrences of the formal identifier represent the same value in any application of the operation. The *result-datatype* indicates the datatype of the value resulting from an application of the operation. A *formal-definition* defines the operation in terms of other operations and constants. A *prose-definition* defines the operation in somewhat formalized natural language. When there are constraints on the parameter values, they are expressed by a phrase beginning "where" immediately before the = or is.

In some operation definitions, characterizing operations of a previously defined datatype are referenced with the form: *datatype.operation(parameters)*, where *datatype* is the *type-specifier* for the referenced datatype and *operation* is the name of a characterizing operation defined for that datatype.

8.1.1 Boolean

Description: Boolean is the mathematical datatype associated with two-valued logic.

Syntax:

boolean-type = "boolean" .

Parametric Values: none.

Values: "true", "false", such that true ≠ false.

Value-syntax:

boolean-literal = "true" | "false" .

Properties: unordered, exact, non-numeric.

Operations: Equal, Not, And, Or.

Equal(x, y: boolean): boolean is defined by tabulation:

x	y	Equal(x,y)
true	true	true
true	false	false
false	true	false
false	false	true

Not(x: boolean): boolean is defined by tabulation:

x	Not(x)
true	false
false	true

Or(x,y: boolean): boolean is defined by tabulation:

x	y	Or(x,y)
true	true	true
true	false	true
false	true	true
false	false	false

And(x, y: boolean): boolean = Not(Or(Not(x), Not(y))).

NOTE — Either And or Or is sufficient to characterize the boolean datatype, and given one, the other can be defined in terms of it. They are both defined here because both of them are used in the definitions of operations on other datatypes.

8.1.2 State

Description: State is a family of datatypes, each of which comprises a finite number of distinguished but unordered values.

Syntax:

```
state-type = "state" "(" state-value-list ")" .
state-value-list = state-literal { "," state-literal } .
state-literal = identifier .
```

Parametric Values: Each *state-literal* identifier shall be distinct from all other *state-literal* identifiers of the same *state-type*.

Values: The value space of a state datatype is the set comprising exactly the named values in the *state-value-list*, each of which is designated by a unique *state-literal*.

Value-syntax:

```
state-literal = identifier .
```

A *state-literal* denotes that value of the state datatype which has the same identifier.

Properties: unordered, exact, non-numeric.

Operations: Equal.

Equal(x, y: state(*state-value-list*)): boolean **is** true if x and y designate the same value in the *state-value-list*, and false otherwise.

NOTE — Other uses of the IDN syntax make stronger requirements on the uniqueness of *state-literal* identifiers.

EXAMPLE — The declaration:

```
type switch = new state (on, off);
```

defines a state datatype comprising two distinguished but unordered values, which supports the characterizing operation:

```
Invert(x: switch): switch is if x = off then on, else off.
```

8.1.3 Enumerated

Description: Enumerated is a family of datatypes, each of which comprises a finite number of distinguished values having an intrinsic order.

Syntax:

```
enumerated-type = "enumerated" "(" enumerated-value-list ")" .
enumerated-value-list = enumerated-literal { "," enumerated-literal } .
enumerated-literal = identifier .
```

Parametric Values: Each *enumerated-literal* identifier shall be distinct from all other *enumerated-literal* identifiers of the same *enumerated-type*.

Values: The value space of an enumerated datatype is the set comprising exactly the named values in the *enumerated-value-list*, each of which is designated by a unique *enumerated-literal*. The order of these values is given by the sequence of their occurrence in the *enumerated-value-list*, designated the **naming sequence**.

Value-syntax:

```
enumerated-literal = identifier .
```

An *enumerated-literal* denotes that value of the enumerated datatype which has the same identifier.

Properties: ordered, exact, non-numeric, bounded.

Operations: Equal, InOrder, Successor

Equal(x, y: enumerated(*enum-value-list*)): boolean **is** true if x and y designate the same value in the *enum-value-list*, and false otherwise.

InOrder(x, y: enumerated(*enum-value-list*)): boolean, denoted $x \leq y$, **is** true if $x = y$ or if x precedes y in the naming sequence, else false.

Successor(x: enumerated(*enum-value-list*)): enumerated(*enum-value-list*) **is**
 if for all y: enumerated(*enum-value-list*), $x \leq y$ implies $x = y$, then undefined;
 else the value y: enumerated(*enum-value-list*), such that $x < y$ and for all $z \neq x$, $x \leq z$ implies $y \leq z$.

NOTE — Other uses of the IDN syntax make stronger requirements on the uniqueness of *enumerated-literal* identifiers.

8.1.4 Character

Description: Character is a family of datatypes whose value spaces are character-sets.

Syntax:

```
character-type = "character" [ "(" repertoire-list ")" ] .
repertoire-list = repertoire-identifier { "," repertoire-identifier } .
repertoire-identifier = value-expression .
```

Parametric Values: The *value-expression* for a *repertoire-identifier* shall designate a value of the objectidentifier datatype (see 10.1.10), and that value shall refer to a character-set. A *repertoire-identifier* shall not be a *formal-parametric-value*, except in some cases in declarations (see 9.1). All *repertoire-identifiers* in a given *repertoire-list* shall designate subsets of the same reference character-set. When *repertoire-list* is not specified, it shall have a default value. The means for specification of the default is outside the scope of this International Standard.

Values: The value space of a character datatype comprises exactly the members of the character-sets identified by the *repertoire-list*. In cases where the character-sets identified by the individual *repertoire-identifiers* have members in common, the value space of the character datatype is the (set) union of the character-sets (without duplication).

Value-syntax:

```
character-literal = "" any-character "" .
any-character = bound-character | added-character | escape-character .
bound-character = non-quote-character | quote .
non-quote-character = letter | digit | underscore | special | apostrophe | space .
added-character = not defined by this International Standard .
escape-character = escape character-name escape .
character-name = identifier { " " identifier } .
```

Every *character-literal* denotes a single member of the character-set identified by *repertoire-list*. A *bound-character* denotes that member which is associated with the symbol for the *bound-character* per 7.1. An *added-character* denotes that member which is associated with the symbol for the *added-character* by the implementation, as provided in 7.1. An *escape-character* denotes that member whose "character name" in the (reference) character-set identified by *repertoire-list* is the same as *character-name*.

Properties: unordered, exact, non-numeric.

Operations: Equal.

Equal(*x*, *y*: character(*repertoire-list*)): boolean is true if *x* and *y* designate the same member of the character-set given by *repertoire-list*, and false otherwise.

NOTES

1. The Character datatypes are distinct from the State datatypes in that the values of the datatype are defined by other standards rather than by this International Standard or by the application. This distinction is semantically unimportant, but it is of great significance in any use of these standards.
2. The standardization of *repertoire-identifier* values will be necessary for any use of this International Standard and will of necessity extend to character sets which are defined by other than international standards. Such standardization is beyond the scope of this International Standard. A partial list of the international standards defining such character-sets is included, for informative purposes only, in Annex A.
3. While an order relationship is important in many applications of character datatypes, there is no standard order for any of the International Standard character sets, and many applications require the order relationship to conform to rules which are particular to the application itself or its language environment. There will also be applications in which the order is unimportant. Since no standard order of character-sets can be defined by this International Standard, character datatypes are said to be "unordered", meaning, in this case, that the order relationship is an application-defined addition to the semantics of the datatype.
4. The terms *character-set*, *member*, *symbol* and *character-name* are those of ISO/IEC 10646-1:1993, but there should be analogous notions in any character set referenceable by a *repertoire-identifier*.
5. The value space of a Character datatype is the character *set*, not the character *codes*, as those terms are defined by ISO/IEC 10646-1:1993. The encoding of a character set is a representation issue and therefore out of the scope of this International Standard. Many uses of this International Standard, however, may require the association to codes implied by the *repertoire-identifier*.

6. An occurrence of three consecutive APOSTROPHE characters (''') is a valid *character-literal* denoting the APOSTROPHE character.

EXAMPLE — `character({ iso standard 8859 part 1 })` denotes a character datatype whose values are the members of the character-set specified by ISO 8859-1 (Latin alphabet No. 1). It is possible to give this datatype a convenient name, by means of a *type-declaration* (see 9.1), e.g.:

`type Latin1 = character({ iso standard 8859 1 });`

or by means of a *value-declaration* (see 9.2):

`value latin : objectidentifier = { iso(1) standard(0) 8859 part(1) };`

Now, the colon mark (:) is a member of the ISO 8859-1 character set and therefore a value of datatype Latin1, or equivalently, of datatype `character(latin)`. Thus, ':', '!' and '!colon!', among others, are valid *character-literals* denoting that value.

8.1.5 Ordinal

Description: Ordinal is the datatype of the ordinal numbers, as distinct from the quantifying numbers (datatype Integer). Ordinal is the infinite enumerated datatype.

Syntax:

`ordinal-type = "ordinal" .`

Parametric Values: none.

Values: the mathematical ordinal numbers: "first", "second", "third", etc., (a denumerably infinite list).

Value-syntax:

`ordinal-literal = number .`

`number = digit-string .`

An *ordinal-literal* denotes that ordinal value which corresponds to the cardinal number identified by the *digit-string*, interpreted as a decimal number. An *ordinal-literal* shall not be zero.

Properties: ordered, exact, non-numeric, unbounded above, bounded below.

Operations: Equal, InOrder, Successor

Equal(x, y: ordinal): boolean is true if x and y designate the same ordinal number, and false otherwise.

InOrder(x,y: ordinal): boolean, denoted $x \leq y$, is true if $x = y$ or if x precedes y in the ordinal numbers, else false.

Successor(x: ordinal): ordinal is the value y: ordinal, such that $x < y$ and for all $z \neq x$, $x \leq z$ implies $y \leq z$.

8.1.6 Date-and-Time

Description: Date-and-Time is a family of datatypes whose values are points in time to various common resolutions: year, month, day, hour, minute, second, and fractions thereof.

Syntax:

`time-type = "time" "(" time-unit [ "," radix "," factor ] ")" .`

`time-unit = "year" | "month" | "day" | "hour" | "minute" | "second" | formal-parametric-value .`

`radix = value-expression .`

`factor = value-expression .`

Parametric Values: *Time-unit* shall be a value of the datatype `state(year, month, day, hour, minute, second)`, designating the unit to which the point in time is resolved. If *radix* and *factor* are omitted, the resolution is to one of the specified *time-unit*. If present, *radix* shall have an integer value greater than 1, and *factor* shall have an integer value. When *radix* and *factor* are present, the resolution is to one $radix^{(-factor)}$ of the specified *time-unit*. *Time-unit*, and *radix* and *factor* if present, shall not be *formal-parametric-values* except in some occurrences in declarations (see 9.1).

Values: The value-space of a date-and-time datatype is the denumerably infinite set of all possible points in time with the resolution (*time-unit*, *radix*, *factor*).

Value-syntax:

`time-literal = string-literal .`

A *time-literal* denotes a date-and-time value. The characterstring value represented by the *string-literal* shall conform to ISO 8601:1988, *Representation of dates and times*. The *time-literal* denotes the date-and-time value specified by the characterstring as interpreted under ISO 8601:1988.

Properties: ordered, exact, non-numeric, unbounded.

Operations: Equal, InOrder, Difference, Round, Extend.

Equal(x, y : time($time\text{-}unit, radix, factor$)): boolean **is** true if x and y designate the same point in time to the resolution ($time\text{-}unit, radix, factor$), and false otherwise.

InOrder(x, y : time($time\text{-}unit, radix, factor$)): boolean **is** true if the point in time designated by x precedes that designated by y ; else false.

Difference(x, y : time($time\text{-}unit, radix, factor$)): timeinterval($time\text{-}unit, radix, factor$) **is**:
if InOrder(x, y), then the number of time-units of the specified resolution elapsing between the time x and the time y ;
else, let z be the number of time-units elapsing between the time y and the time x , then Negate(z).

Extend.resItores2(x : time($unit1, radix1, factor1$)): time($unit2, radix2, factor2$), where the resolution ($res2$) specified by ($unit2, radix2, factor2$) is more precise than the resolution ($res1$) specified by ($unit1, radix1, factor1$), **is** that value of time($unit2, radix2, factor2$) which designates the first instant of time occurring within the span of time($unit2, radix2, factor2$) identified by the instant x .

Round.resItores2(x : time($unit1, radix1, factor1$)): time($unit2, radix2, factor2$), where the resolution ($res2$) specified by ($unit2, radix2, factor2$) is less precise than the resolution ($res1$) specified by ($unit1, radix1, factor1$), **is** the largest value y of time($unit2, radix2, factor2$) such that InOrder(Extend.res2tores1(y), x).

NOTE — The operations yielding specific time-unit elements from a time($unit, radix, factor$) value, e.g. Year, Month, DayofYear, DayofMonth, TimeofDay, Hour, Minute, Second, can be derived from Round, Extend, and Difference.

EXAMPLE — time(second, 10, 0) designates a date-and-time datatype whose values are points in time with accuracy to the second. "19910401T120000" specifies the value of that datatype which is exactly noon on April 1, 1991, universal time.

8.1.7 Integer

Description: Integer is the mathematical datatype comprising the exact integral values.

Syntax:

integer-type = "integer" .

Parametric Values: none.

Values: Mathematically, the infinite ring produced from the additive identity (0) and the multiplicative identity (1) by requiring $0 \leq 1$ and $\text{Add}(x, 1) \neq y$ for any $y \leq x$. That is: ..., -2, -1, 0, 1, 2, ... (a denumerably infinite list).

Value-syntax:

integer-literal = signed-number .

signed-number = ["-"] number .

number = digit-string .

An *integer-literal* denotes an integer value. If the negative-sign ("-") is not present, the value denoted is that of the *digit-string* interpreted as a decimal number. If the negative-sign is present, the value denoted is the negative of that value.

Properties: ordered, exact, numeric, unbounded.

Operations: Equal, InOrder, NonNegative, Negate, Add, Multiply.

Equal(x, y : integer): boolean **is** true if x and y designate the same integer value, and false otherwise.

Add(x, y : integer): integer **is** the mathematical additive operation.

Multiply(x, y : integer): integer **is** the mathematical multiplicative operation.

Negate(x : integer): integer **is** the value y : integer such that $\text{Add}(x, y) = 0$.

NonNegative(x : integer): boolean **is**
true if $x = 0$ or x can be developed by one or more iterations of adding 1,
i.e. if $x = \text{Add}(1, \text{Add}(1, \dots \text{Add}(1, \text{Add}(1, 0)) \dots))$;
else false.

InOrder(x, y : integer): boolean = NonNegative(Add(x , Negate(y))).

The following operations are defined solely in order to facilitate other datatype definitions:

Quotient(x, y : integer): integer, where $0 < y$, is the upperbound of the set of all integers z such that $\text{Multiply}(y, z) \leq x$.

Remainder(x, y : integer): integer, where $0 \leq x$ and $0 < y$, = $\text{Add}(x, \text{Negate}(\text{Multiply}(y, \text{Quotient}(x, y))))$;

8.1.8 Rational

Description: Rational is the mathematical datatype comprising the "rational numbers".

Syntax:

rational-type = "rational" .

Parametric Values: none.

Values: Mathematically, the infinite field produced by closing the Integer ring under multiplicative-inverse.

Value-syntax:

rational-literal = signed-number ["/" number] .

Signed-number and *number* shall denote the corresponding integer values. *Number* shall not designate the value 0. The rational value denoted by the form *signed-number* is:

Promote(*signed-number*),

and the rational value denoted by the form *signed-number/number* is:

$\text{Multiply}(\text{Promote}(\text{signed-number}), \text{Reciprocal}(\text{Promote}(\text{number})))$.

Properties: ordered, exact, numeric, unbounded.

Operations: Equal, NonNegative, InOrder, Negate, Add, Multiply, Reciprocal, Promote.

Equal(x, y : rational): boolean is true if x and y designate the same rational number, and false otherwise.

Promote(x : integer): rational is the embedding isomorphism between the integers and the integral rational values.

Add(x, y : rational): rational is the mathematical additive operation.

Multiply(x, y : rational): rational is the mathematical multiplicative operation.

Negate(x : rational): rational is the value y : rational such that $\text{Add}(x, y) = 0$.

Reciprocal(x : rational): rational, where $x \neq 0$, is the value y : rational such that $\text{Multiply}(x, y) = 1$.

NonNegative(k : rational): boolean is defined by:

For every rational value k , there is a non-negative integer n , such that $\text{Multiply}(n, k)$ is an integral value, and:

$\text{NonNegative}(k) = \text{integer.NonNegative}(\text{Multiply}(n, k))$.

InOrder(x, y : rational): boolean = $\text{NonNegative}(\text{Add}(x, \text{Negate}(y)))$

8.1.9 Scaled

Description: Scaled is a family of datatypes whose value spaces are subsets of the rational value space, each individual datatype having a fixed denominator, but the scaled datatypes possess the concept of approximate value.

Syntax:

scaled-type = "scaled" "(" radix "," factor ")" .

radix = value-expression .

factor = value-expression .

Parametric Values: *Radix* shall have an integer value greater than 1, and *factor* shall have an integer value. *Radix* and *factor* shall not be *formal-parametric-values* except in some occurrences in declarations (see 9.1).

Values: The value space of a scaled datatype is that set of values of the rational datatype which are expressible as a value of datatype Integer divided by *radix* raised to the power *factor*.

Value-syntax:

scaled-literal = integer-literal ["*" scale-factor] .

scale-factor = number "^" signed-number .

A *scaled-literal* denotes a value of a scaled datatype. The *integer-literal* is interpreted as a decimal integer value, and the *scale-factor*, if present, is interpreted as *number* raised to the power *signed-number*, where *number* and *signed-number* are expressed as decimal integers. *Number* should be the same as the *radix* of the datatype. If the *scale-factor* is not present,

the value is that denoted by *integer-literal*. If the *scale-factor* is present, the value denoted is the rational value `Multiply(integer-literal, scale-factor)`.

Properties: ordered, exact, numeric, unbounded.

Operations: Equal, InOrder, Negate, Add, Round, Multiply, Divide

`Equal(x, y: scaled(r,f))`: boolean is true if *x* and *y* designate the same rational number, and false otherwise.

`InOrder(x,y: scaled (r,f))`: boolean = `rational.InOrder(x,y)`

`Negate(x: scaled (r,f))`: `scaled (r,f) = rational.Negate(x)`

`Add(x,y: scaled (r,f))`: `scaled (r,f) = rational.Add(x,y)`

`Round(x: rational)`: `scaled(r,f)` is the value *y*: `scaled(r,f)` such that `rational.InOrder(y, x)` and for all *z*: `scaled(r,f)`, `rational.InOrder(z,x)` implies `rational.InOrder(z,y)`.

`Multiply(x,y: scaled(r,f))`: `scaled(r,f) = Round(rational.Multiply(x,y))`

`Divide(x,y: scaled(r,f))`: `scaled(r,f) = Round(rational.Multiply(x, Reciprocal(y)))`

EXAMPLES

1. A datatype representing monetary values exact to two decimal places can be defined by:

```
type currency = new scaled(10, 2);
```

where the keyword "new" is used because currency does not support the Multiply and Divide operations characterizing `scaled(10,2)`.

2. The value 39.50 (or 39,50), i.e. thirty-nine and fifty one-hundredths, is represented by: $3950 * 10^{-2}$, while the value 10.00 (or 10,00) may be represented by: 10.

NOTES

1. The case factor = 0, i.e. `scaled(r, 0)` for any *r*, has the same value-space as Integer, and is isomorphic to Integer under all operations except Divide, which is not defined on Integer in this International Standard, but could be defined consistent with the Divide operation for `scaled(r, 0)`. It is recommended that the datatype `scaled(r, 0)` not be used explicitly.
2. Any reasonable rounding algorithm is equally acceptable. What is required is that any rational value *v* which is not a value of the scaled datatype is mapped into one of the two scaled values $n \cdot r^{(-f)}$ and $(n+1) \cdot r^{(-f)}$, such that in the Rational value space, $n \cdot r^{(-f)} < v < (n+1) \cdot r^{(-f)}$.
3. The proper definition of scaled arithmetic is complicated by the fact that scaled datatypes with the same radix can be combined arbitrarily in an arithmetic expression and the arithmetic is effectively Rational *until* a final result must be produced. At this point, rounding to the proper scale for the result operand occurs. Consequently, the given definition of arithmetic, for operands with a common scale factor, should not be considered a specification for arithmetic on the scaled datatype.
4. The values in any scaled value space are taken from the value space of the Rational datatype, and for that reason Scaled may appear to be a "subtype" of both Rational and Real (see 8.2). But scaled datatypes do not "inherit" the Rational or Real Multiply and Reciprocal operations. Therefore scaled datatypes are not proper subtypes of datatype Real or Rational. The concept of Round, and special Multiply and Divide operations, characterize the scaled datatypes. Unlike Rational, Real and Complex, however, Scaled is not a mathematical group under this definition of Multiply, although the results are intuitively acceptable.
5. The value space of a scaled datatype contains the multiplicative identity (1) if and only if factor ≥ 0 .
6. Every scaled datatype is exact, because every value in its value space can be distinguished in the computational model. (The value space can be mapped 1-to-1 onto the integers.) It is only the *operations* on scaled datatypes which are approximate.
7. *Scaled-literals* are interpreted as decimal values regardless of the *radix* of the scaled datatype to which they belong. It was not found necessary for this International Standard to provide for representation of values in other radices, particularly since representation of values in radices greater than 10 introduces additional syntactic complexity.

8.1.10 Real

Description: Real is a family of datatypes which are computational approximations to the mathematical datatype comprising the "real numbers". Specifically, each real datatype designates a collection of mathematical real values which are known to certain applications to some finite precision and must be distinguishable to at least that precision in those applications.

Syntax:

```
real-type = "real" [ "(" radix "," factor ")" ] .
```

radix = value-expression .

factor = value-expression .

Parametric Values: *Radix* shall have an integer value greater than 1, and *factor* shall have an integer value. *Radix* and *factor* shall not be *formal-parametric-values* except in some occurrences in declarations (see 9.1). When *radix* and *factor* are not specified, they shall have default values. The means for specification of these defaults is outside the scope of this International Standard.

Values: The value space of the mathematical real type comprises all values which are the limits of convergent sequences of rational numbers. The value space of a computational real datatype shall be a subset of the mathematical real type, characterized by two parametric values, *radix* and *factor*, which, taken together, describe the precision to which values of the datatype are distinguishable, in the following sense:

Let $\mathfrak{R}$ denote the mathematical real value space and for v in $\mathfrak{R}$, let $|v|$ denote the absolute value of v . Let V denote the value space of datatype *real(radix, factor)*, and let $\epsilon = \text{radix}^{(-factor)}$. Then V shall be a subset of $\mathfrak{R}$ with the following properties:

- 0 is in V ;
- for each r in $\mathfrak{R}$ such that $|r| \geq \epsilon$, there exists at least one $\bar{r}$ in V such that $|r - \bar{r}| \leq |r| \cdot \epsilon$;
- for each r in $\mathfrak{R}$ such that $|r| < \epsilon$, there exists at least one $\bar{r}$ in V such that $|r - \bar{r}| \leq \epsilon^2$.

Value-syntax:

real-literal = integer-literal ["*" scale-factor] .

scale-factor = number "^" signed-number .

A *real-literal* denotes a value of a real datatype. The *integer-literal* is interpreted as a decimal integer value, and the *scale-factor*, if present, is interpreted as *number* raised to the power *signed-number*, where *number* and *signed-number* are expressed as decimal integers. If the *scale-factor* is not present, the value is that denoted by *integer-literal*. If the *scale-factor* is present, the value denoted is the rational value *Multiply(integer-literal, scale-factor)*.

Properties: ordered, approximate, numeric, unbounded.

Operations: Equal, InOrder, Promote, Negate, Add, Multiply, Reciprocal.

In the following operation definitions, let M designate an approximation function which maps each r in $\mathfrak{R}$ into a corresponding $\bar{r}$ in V with the properties given above and the further requirement that for each v in V , $M(v) = v$.

Equal(x, y : *real(radix, factor)*): boolean is true if x and y designate the same value, and false otherwise.

InOrder(x, y : *real(radix, factor)*): boolean is true if $x \leq y$, where $\leq$ designates the order relationship on $\mathfrak{R}$, and false otherwise.

Promote(x : rational): *real(radix, factor)* = $M(x)$.

Add(x, y : *real(radix, factor)*): *real(radix, factor)* = $M(x + y)$, where $+$ designates the additive operation on the mathematical reals.

Multiply(x, y : *real(radix, factor)*): *real(radix, factor)* = $M(x \cdot y)$, where $\cdot$ designates the multiplicative operation on the mathematical reals.

Negate(x : *real(radix, factor)*): *real(radix, factor)* = $M(-x)$, where $-x$ is the real additive inverse of x .

Reciprocal(x : *real(radix, factor)*): *real(radix, factor)*, where $x \neq 0$, = $M(x')$ where x' is the real multiplicative inverse of x .

NOTES

1. The LI datatype Real is not the abstract mathematical real datatype, nor is it an abstraction of floating-point implementations. It is a computational model of the mathematical reals which is similar to the "scientific number" model used in many sciences. Details of the relationship of a real datatype to floating-point implementations may be specified by the use of annotations (see 7.4). For languages whose semantics in some way assumes a floating-point representation, the use of such annotations in the datatype mappings may be necessary. On the other hand, for some applications, the representation of a real datatype may be something other than floating-point, which the application would specify by different annotations.

2. Detailed requirements for the approximation function, its relationship to the characterizing operations, and the implementation of the characterizing operations in languages are provided by ISO/IEC 10967-1:1994, Information technology — Programming languages, their environments and system software interfaces — Language-Independent arithmetic — Part 1: Integer and real arithmetic. IEC 559:1988 Floating-Point Arithmetic for Microprocessors specifies the requirements for floating-point implementations thereof.

EXAMPLES

real(10, 7) denotes a real datatype with values which are accurate to 7 significant decimal figures.

`real(2, 48)` denotes a real datatype whose values have at least 48 bits of precision.

$1 * 10^9$ denotes the value 1 000 000 000, i.e. 10 raised to the ninth power.

$15 * 10^{-4}$ denotes the value 0,0015, i.e. fifteen ten-thousandths.

$3 * 2^{-1}$ denotes the value 1.5, i.e. $3/2$.

8.1.11 Complex

Description: Complex is a family of datatypes, each of which is a computational approximation to the mathematical datatype comprising the "complex numbers". Specifically, each complex datatype designates a collection of mathematical complex values which are known to certain applications to some finite precision and must be distinguishable to at least that precision in those applications.

Syntax:

`complex-type` = "complex" ["(" radix "," factor ")"] .

`radix` = value-expression .

`factor` = value-expression .

Parametric Values: *Radix* shall have an integer value greater than 1, and *factor* shall have an integer value. *Radix* and *factor* shall not be *formal-parametric-values* except in some occurrences in declarations (see 9.1). When *radix* and *factor* are not specified, they shall have default values. The means for specification of these defaults is outside the scope of this International Standard.

Values: The value space of the mathematical complex type is the field which is the solution space of all polynomial equations having real coefficients. The value space of a computational complex datatype shall be a subset of the mathematical complex type, characterized by two parametric values, *radix* and *factor*, which, taken together, describe the precision to which values of the datatype are distinguishable, in the following sense:

Let C denote the mathematical complex value space and for v in C , let $|v|$ denote the absolute value of v . Let V denote the value space of datatype `complex(radix, factor)`, and let $\epsilon = \text{radix}^{-\text{factor}}$. Then V shall be a subset of C with the following properties:

- 0 is in V ;
- for each v in C such that $|v| \geq \epsilon$, there exists at least one $\bar{v}$ in V such that $|v - \bar{v}| \leq |v| \cdot \epsilon$;
- for each v in C such that $|v| < \epsilon$, there exists at least one $\bar{v}$ in V such that $|v - \bar{v}| \leq \epsilon^2$.

Value-syntax:

`complex-literal` = "(" real-part "," imaginary-part ")" .

`real-part` = real-literal .

`imaginary-part` = real-literal .

A *complex-literal* denotes a value of a complex datatype. The *real-part* and the *imaginary-part* are interpreted as real values, and the complex value denoted is: $M(\text{real-part} + (\text{imaginary-part} \cdot i))$, where $+$ is the additive operation on the mathematical complex numbers and $\cdot$ is the multiplicative operation on the mathematical complex numbers, and i is the "principal square root" of -1 (one of the two solutions to $x^2 + 1 = 0$).

Properties: approximate, numeric, unordered.

Operations: Equal, Promote, Negate, Add, Multiply, Reciprocal, SquareRoot.

In the following operation definitions, let M designate an approximation function which maps each v in C into a corresponding $\bar{v}$ in V with the properties given above and the further requirement that for each v in V , $M(v) = v$.

Equal(x, y : `complex(radix, factor)`): boolean is true if x and y designate the same value, and false otherwise.

Promote(x : `real(radix, factor)`): `complex(radix, factor)` = $M(x)$, considering x as a mathematical real value.

Add(x, y : `complex(radix, factor)`): `complex(radix, factor)` = $M(x + y)$, where $+$ designates the additive operation on the mathematical complex numbers.

Multiply(x, y : `complex(radix, factor)`): `complex(radix, factor)` = $M(x \cdot y)$, where $\cdot$ designates the multiplicative operation on the mathematical complex numbers.

Negate(x : `complex(radix, factor)`): `complex(radix, factor)` = $M(-x)$, where $-x$ is the complex additive inverse of x .

Reciprocal(x : `complex(radix, factor)`): `complex(radix, factor)`, where $x \neq 0$, = $M(x')$ where x' is the complex multiplicative inverse of x .