INTERNATIONAL STANDARD

ISO/IEC 11404

Second edition
2007-12-15

Information technology — General-Purpose Datatypes (GPD)

Technologies de l'information — Types de données

Reference number
ISO/IEC 11404:2007(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword vi

0 Introduction vii

1 Scope 1

2 Normative references 1

3 Terms and definitions 2

4 Conformance 8
4.1 Direct conformance 8
4.2 Indirect conformance 9
4.3 Conformance of a mapping standard 9
4.4 GPD program conformance 10

5 Conventions used in this International Standard 10
5.1 Formal syntax 10
5.2 Text conventions 11

6 Fundamental notions 11
6.1 Datatype 11
6.2 Value space 12
6.3 Datatype properties 12
6.3.1 Equality 13
6.3.2 Order 13
6.3.3 Bound 13
6.3.4 Cardinality 14
6.3.5 Exact and approximate 14
6.3.6 Numeric 14
6.4 Primitive and non-primitive datatypes 15
6.5 Datatype generator 15
6.6 Characterizing operations 15
6.7 Datatype families 16
6.8 Aggregate datatypes 17
6.8.1 Homogeneity 17
6.8.2 Size 17
6.8.3 Uniqueness 17
6.8.4 Aggregate-imposed identifier uniqueness 18
6.8.5 Aggregate-imposed ordering 18
6.8.6 Access method 18
6.8.7 Recursive structure 19
6.8.8 Structured and unstructured 19
6.8.9 Mandatory and optional components 19
6.9 Provisions associated with datatypes 19

7 Elements of the Datatype Specification Language 21
7.1 IDN character-set 21
7.2 Whitespace 22
7.3 Lexical objects 23
7.3.1 Identifiers 23
7.3.2 Digit-string 23
7.3.3 Character-literal and string-literal 23
7.3.4 Keywords 24
7.4 Annotations 24
7.5 Values 25

7.5.1 Independent values 25
7.5.2 Dependent values 26
7.6 GPD program text 27

8 Datatypes 27
8.1 Primitive datatypes 28
8.1.1 Boolean 29
8.1.2 State 30
8.1.3 Enumerated 31
8.1.4 Character 32
8.1.5 Ordinal 33
8.1.6 Date-and-Time 34
8.1.7 Integer 35
8.1.8 Rational 36
8.1.9 Scaled 37
8.1.10 Real 38
8.1.11 Complex 40
8.1.12 Void 41
8.2 Subtypes and extended types 42
8.2.1 Range 43
8.2.2 Selecting 43
8.2.3 Excluding 44
8.2.4 Size 44
8.2.5 Explicit subtypes 45
8.2.6 Extended 45
8.3 Generated datatypes 46
8.3.1 Choice 47
8.3.2 Pointer 49
8.3.3 Procedure 50
8.4 Aggregate Datatypes 53
8.4.1 Record 55
8.4.2 Class 56
8.4.3 Set 58
8.4.4 Bag 59
8.4.5 Sequence 60
8.4.6 Array 61
8.4.7 Table 64
8.5 Defined datatypes 66
8.6 Provisions 66
8.6.1 General parameters for provisions 67
8.6.2 Aggregate-specific features 70
8.6.3 Aggregate-component-identifier uniqueness 70
8.6.4 Usage-specific features 71

9 Declarations 72
9.1 Type declarations 72
9.1.1 Renaming declarations 73
9.1.2 New datatype declarations 73
9.1.3 New generator declarations 73
9.2 Value declarations 73
9.3 Termination declarations 74
9.4 Normative datatype declarations 74
9.5 Lexical operations 74
9.5.1 Import 74
9.5.2 Macro 75

10 Defined datatypes and generators 75
10.1 Defined datatypes 75
10.1.1 Natural number 76
10.1.2 Modulo 76
10.1.3 Bit 77

10.1.4 Bit string 77
10.1.5 Character string 77
10.1.6 Time interval 79
10.1.7 Octet 79
10.1.8 Octet string 79
10.1.9 Private 80
10.1.10 Object identifier 80
10.2 Defined generators 82
10.2.1 Stack 82
10.2.2 Tree 83
10.2.3 Optional 83

11 Mappings 84
11.1 Outward Mappings 85
11.2 Inward Mappings 86
11.3 Reverse Inward Mapping 87
11.4 Support of Datatypes 87
11.4.1 Support of equality 87
11.4.2 Support of order 88
11.4.3 Support of bounds 88
11.4.4 Support of cardinality 88
11.4.5 Support for the exact or approximate property 88
11.4.6 Support for the numeric property 88
11.4.7 Support for the mandatory components 88

Annex A (informative) **Character-set standards** 89

Annex B (informative) **Recommendation for the placement of annotations** 91

Annex C (informative) **Implementation notions of datatypes** 93

Bibliography 96

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. 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.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 11404 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This second edition cancels and replaces the first edition (ISO/IEC 11404:1996), which has been technically revised.

0 Introduction

0.1 Introduction to the second edition

This second edition of ISO/IEC 11404 incorporates recent technologies and improvements since the first edition (ISO/IEC 11404:1996). The following improvements have been incorporated into the second edition.

— *Title change to reflect actual usage*. The use of ISO/IEC 11404 is no longer simply a tool for communicating among programming languages (old title: *Language-independent datatypes*). ISO/IEC 11404 is used for formal description of conceptual datatypes in binding (or binding-independent) standards and used as formalization of metadata for data elements, data element concepts, and value domains (see ISO/IEC 11179-3). The old title was potentially misleading because readers might believe that ISO/IEC 11404 is only useful for programming languages. The new title, *General-Purpose Datatypes* captures the essence of ISO/IEC 11404 and its use.

— *Incorporation of latest technologies*. Provide enhancements to the use of ISO/IEC 11404 as a datatype nomenclature reference for current programming languages, interface languages and data representation languages, specifically Java, IDL, Express, and XML.

— *Support for semi-structured and unstructured data aggregates*. Semi-structured data and unstructured data includes aggregates where datatyping and navigation may be unknown or unspecified in advance. For example, some systems permit “discovery” (or “introspection”) of data. In some cases, the datatype may be unknown in advance (e.g. at compilation time), but may be discovered and processed at runtime (e.g. via datatype libraries or metadata registries).

— *Support for data longevity, versioning, and migration*. There is a need to support, from a datatyping perspective, obsolete and reserved features, such as data elements and permissible values (enumerations and states). Marking features as “obsolete” allows processing, compilation, and runtime systems to “flag” or diagnose old (deprecated) features, while still maintaining compatibility, so that it is possible to support transitions from past to present. Similarly, marking features as “reserved” allows processing, compilation, and runtime systems to “flag” or diagnose potential incompatibilities with future systems, so that it is possible to support transitions from present to future.

— *Extensibility of datatypes and value spaces*. There is a need to support some kind of extensibility concept. For example: (1) a GPD specification of an aggregate contains the elements A and B. (2) An application creates an aggregate with the elements A, B, and C. (3) Are the application's “extensions” of the aggregate acceptable/in conformity with the GPD specification in (1)? The answer to (3) is dependent upon the intent and design of the specification in (1): in some cases extensions are permitted, in some cases extensions are not permitted. The extensibility concept would allow the user of GPD datatypes to describe the kind of extensions permitted. This feature is particularly important in (a) data conformance and (b) application runtime environments that permit “discovery” or “introspection”. This feature is available via the “provision()” capability.

Features that are not incorporated within GPD include the following:

— *Namespace capability*. Given the larger number of declarations, a namespace capability is necessary.

— *Data representation*. Although these features are a part of GPD annotations, there is no standardization of data representation in these annotations. This step is an important link for data interoperability.

0.2 Introduction to the first edition (ISO/IEC 11404:1996)

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 JTC 1, *Information technology* assigned to SC 22 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 — General-Purpose Datatypes (GPD)

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 General-Purpose Datatypes (GPD). 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 "general-purpose" in the sense that the datatypes specified are classes of datatype of which the actual datatypes used in programming languages and other entities requiring the concept "datatype" are particular instances. These datatypes are general in nature; thus, they serve a wide variety of information processing applications.

This International Standard expressly distinguishes three notions of datatype:

— 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 GPDs defined herein.

This International Standard defines a partial terminology for implementation notions of datatypes and provides for the use of this terminology in the definition of datatypes. The primary purpose of this terminology is to identify common implementation notions associated with datatypes and to distinguish them from conceptual notions.

This International Standard specifies the required elements of mappings between the GPDs 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 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

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

ISO/IEC 8824 (all parts), *Information technology — Abstract Syntax Notation One (ASN.1)*

ISO/IEC 10646, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*

ISO/IEC 14977, *Information technology — Syntactic metalanguage — Extended BNF*

IETF RFC 2396, *Uniform Resource Identifiers (URI): Generic Syntax*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

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

3.1
actual parametric datatype
datatype appearing as a parametric datatype in a use of a datatype generator, in contrast to the formal-parametric-types appearing in the definition of the datatype generator

3.2
actual parametric value
value appearing as a parametric value in a reference to a datatype family or datatype generator, in contrast to the formal-parametric-values appearing in the corresponding definitions

3.3
aggregate datatype
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

3.4
annotation
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

3.5
approximate
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

3.6
bounded
property of a datatype, meaning both bounded above and bounded below

3.7
bounded above
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, $s \leq U$

3.8
bounded below
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, $s \geq L$

3.9
characterizing operations
⟨datatype⟩[1)] collection of operations on, or yielding, values of the datatype that distinguish this datatype from other datatypes with identical value spaces

3.10
characterizing operations
⟨datatype generator⟩ collection of operations on, or yielding, values of any datatype resulting from an application of the datatype generator that distinguish this datatype generator from other datatype generators and produce identical value spaces from identical parametric datatypes

3.11
component datatype
datatype which is a parametric datatype to a datatype generator

NOTE A component datatype is a datatype on which the datatype generator operates.

3.12
datatype
set of distinct values, characterized by properties of those values, and by operations on those values

3.13
datatype declaration
means provided by this International Standard for the definition of a datatype which is not itself defined by this International Standard

3.14
datatype family
collection of datatypes which have equivalent characterizing operations and relationships, but value spaces that differ in the number and identification of the individual values

3.15
datatype generator
generator
operation on datatypes, as objects distinct from their values, that generates new datatypes

3.16
defined datatype
datatype defined by a type-declaration

3.17
defined generator
datatype generator defined by a type-declaration

3.18
exact
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

3.19
formal-parametric-type
identifier, appearing in the definition of a datatype generator, for which a datatype will be substituted in any reference to a (defined) datatype resulting from the generator

1) Angle brackets indicate the subject field to which the concept belongs, in accordance with ISO 10241.

3.20
formal-parametric-value
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

3.21
general-purpose datatype
GPD
datatype defined by this International Standard

3.22
GPD-generated datatype
GPD datatype
datatype defined by the means of datatype definition provided by this International Standard

NOTE Although "GPD datatype" expands to "general-purpose datatype datatype" and might appear redundant, it is to be read as "general-purpose-datatype datatype", where GPD is an adjective and datatype (standalone) is a noun.

3.23
generated datatype
datatype defined by the application of a datatype generator to one or more previously-defined datatypes

3.24
generated internal datatype
datatype defined by the application of a datatype generator defined in a particular programming language to one or more previously-defined internal datatypes

3.25
generator declaration
means provided by this International Standard for the definition of a datatype generator which is not itself defined by this International Standard

3.26
instruction
provision that conveys an action to be performed

[ISO/IEC Guide 2]

3.27
internal datatype
datatype whose syntax and semantics are defined by some other standard, specification, language, product, service or other information processing entity

3.28
inward mapping
conceptual association between the internal datatypes of a language and the general-purpose datatypes which assigns to each general-purpose datatype either a single semantically equivalent internal datatype or no equivalent internal datatype

3.29
lower bound
value L such that, for all values s in the value space in a datatype which is bounded below, $L \leq s$

3.30
mandatory requirement
requirement of a normative document that must necessarily be fulfilled in order to comply with that document

NOTE 1 Adapted from the definition of "exclusive requirement" in ISO/IEC Guide 2.

NOTE 2 A "mandatory requirement" is also known as an "exclusive requirement".

3.31
mapping
⟨datatypes⟩ formal specification of the relationship between the internal datatypes that are notions of, and specifiable in, a particular programming language and the general-purpose datatypes specified in this International Standard

3.32
mapping
⟨values⟩ corresponding specification of the relationships between values of the internal datatypes and values of the general-purpose datatypes

3.33
meta-identifier
name of a non-terminal symbol

[ISO/IEC 14977]

NOTE See note in 5.1 concerning the context of the specialized usage of this term for describing the syntax of ISO/IEC 11404 program text.

3.34
non-terminal symbol
⟨EBNF⟩ syntactic part of the language being defined

[ISO/IEC 14977]

NOTE See note in 5.1 concerning the context of the specialized usage of this term.

3.35
normative datatype
collection of specifications for datatype properties that may be simultaneously satisfied by more than one actual datatype

3.36
normative document
document that provides rules, guidelines or characteristics for activities or their results

[ISO/IEC Guide 2]

NOTE 1 The term “normative document” is a generic term that covers such documents as standards and technical specifications.

NOTE 2 A “document” is to be understood as any medium with information recorded on or in it, such as a paper document or program code.

3.37
optional requirement
requirement of a normative document that must be fulfilled in order to comply with a particular option permitted by that document

[ISO/IEC Guide 2]

NOTE An optional requirement may be either (1) one of two or more alternative requirements; or (2) an additional requirement that must be fulfilled only if applicable and that may otherwise be disregarded.

3.38
order
mathematical relationship among values

NOTE See 6.3.2.

3.39
ordered
property of a datatype which is determined by the existence and specification of an order relationship on its value space

3.40
outward mapping
conceptual association between the internal datatypes of a language and the general-purpose datatypes that identifies each internal datatype with a single semantically equivalent general-purpose datatype

3.41
parametric datatype
datatype on which a datatype generator operates to produce a generated datatype

3.42
parametric value (1)
value which distinguishes one member of a datatype family from another

3.43
parametric value (2)
value which is a parameter of a datatype or datatype generator defined by a type-declaration

NOTE See 9.1.

3.44
primitive datatype
identifiable datatype that cannot be decomposed into other identifiable datatypes without loss of all semantics associated with the datatype

3.45
primitive internal datatype
datatype in a particular programming language whose values, conceptually, are not constructed in any way from values of other datatypes in the language

3.46
provision
expression of normative wording that takes the form of a statement, an instruction, a recommendation or a requirement

NOTE 1 Adapted from ISO/IEC Guide 2.

NOTE 2 These types of provision are distinguished by the form of wording they employ; e.g. instructions are expressed in the imperative mood, recommendations by the use of the auxiliary “should” and requirements by the use of the auxiliary “shall”.

3.47
recommendation
provision that conveys advice or guidance

[ISO/IEC Guide 2]

3.48
regular value
element of a value space that is consistent with a datatype's properties and characterizing operations

3.49
representation
⟨general-purpose datatype⟩ mapping from the value space of the general-purpose datatype to the value space of some internal datatype of a computer system, file system or communications environment

3.50
representation
⟨value⟩ sign(s) of that value in the representation of the datatype

NOTE In this context, the term “sign” is used in its terminological sense (i.e. a symbol) and not in its mathematical sense (i.e. positive or negative).

3.51
requirement
provision that conveys criteria to be fulfilled

[ISO/IEC Guide 2]

3.52
sentence
⟨EBNF⟩ sequence of symbols that represents the start symbol

[ISO/IEC 14977]

NOTE See note in 5.1 concerning the context of the specialized usage of this term.

3.53
sentinel value
element of a value space that is not completely consistent with a datatype's properties and characterizing operations

NOTE A numeric datatype, which includes characterizing operations such as **Equal** and **InOrder**, may include sentinel values such as **not-a-number**, **indeterminate**, **not-applicable**, **+infinity**, **-infinity** and so on. These characterizing operations are not defined for sentinel values.

3.54
sequence
⟨EBNF⟩ ordered list of zero or more items

[ISO/IEC 14977]

NOTE See note in 5.1 concerning the context of the specialized usage of this term.

3.55
start symbol
⟨EBNF⟩ non-terminal symbol that is defined by one or more syntax rules but does not occur in any other syntax rule

[ISO/IEC 14977]

NOTE See note in 5.1 concerning the context of the specialized usage of this term.

3.56
statement
provision that conveys information

[ISO/IEC Guide 2]

3.57
subsequence
⟨EBNF⟩ sequence within a sequence

[ISO/IEC 14977]

NOTE See note in 5.1 concerning the context of the specialized usage of this term.

3.58
subtype
datatype derived from another datatype by restricting the value space to a subset whilst maintaining all characterizing operations

3.59
terminal symbol
⟨EBNF⟩ sequence of one or more characters forming an irreducible element of a language

[ISO/IEC 14977]

NOTE See note in 5.1 on the context of the specialized usage of this term.

3.60
upper bound
value U such that, for all values s in the value space in a datatype which is bounded above, $s \leq U$

3.61
value space
set of values for a given datatype

3.62
variable
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

4 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 (4.1), or indirectly, by means of mappings between internal datatypes used by the entity and the datatypes specified in this International Standard (4.2).

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.

4.1 Direct conformance

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

1. specify which of the datatypes and datatype generators specified in Clause 8 and Clause 10 are provided by the entity and which are not, and which, if any, of the declaration mechanisms in Clause 9 it provides; and

2. define the value spaces of the general-purpose datatypes used by the entity to be identical to the value spaces specified by this International Standard; and

3. use the notation prescribed by Clause 7 through Clause 10 of this International Standard to refer to those datatypes and to no others; and

4. to the extent that the entity provides operations other than movement or transformation of values, define operations on the general-purpose datatypes which can be derived from, or are otherwise consistent with, the characterizing operations specified by this International Standard.

NOTE 1 This International Standard defines a syntax for the denotation of values of each datatype it defines, but, in general, requirement 3 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.

NOTE 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.

NOTE 3 Requirement 4 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 general-purpose 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.

NOTE 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.

4.2 Indirect conformance

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

1. provide mappings between its internal datatypes and the general-purpose datatypes conforming to the specifications of Clause 11 of this International Standard; and

2. 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.

NOTE 1 Standards for existing programming languages are expected to provide for indirect conformance rather than direct conformance.

NOTE 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.

4.3 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.

NOTE 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 general-purpose datatypes. Such mapping standards are required to comply with this International Standard.

NOTE 2 Such mapping standards may define “generic” mappings, in the sense that for a given internal datatype the standard specifies a parameterized general-purpose 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 general-purpose datatype, the mapping specifies a family of general-purpose 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 parameterized. 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 general-purpose datatype:

```
integer range (min..max)
```

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

4.4 GPD program conformance

A GDP conforming program [2)] is a specification that uses datatypes and datatype values and their syntaxes as specified in this International Standard. Such a specification may be self-contained: there is no requirement for the existence of a GDP conformant implementation that can produce or operate on the specification. The requirements of this International Standard to which the specification conforms shall be clearly identified, either in the specification itself, or in documentation that is unambiguously identified in the specification.

NOTE A GDP conforming program is a special case of directly conforming entity.

5 Conventions used in this International Standard

5.1 Formal syntax

This International Standard defines a formal datatype specification language. The notation defined in ISO/IEC 14977, Extended Backus-Naur Form (EBNF), is used in defining that language. Table 5-1 summarizes the ISO/IEC 14977 EBNF syntactic metalanguage.

NOTE The terms meta-identifier, non-terminal symbol, sentence, sequence, start symbol, subsequence, and terminal symbol have special meaning in the context of EBNF notation (see Clause 3).

Table 5-1 — Summary of ISO/IEC 14977 EBNF Syntactic Metalanguage Notation

Representation	ISO/IEC 10646 Character Names	Metalanguage Symbol
' '	apostrophe	first quote symbol
" "	quotation mark	second quote symbol
(* *)	left parenthesis with asterisk, asterisk with right parenthesis	start/end comment symbols
( )	left parenthesis, right parenthesis	start/end group symbols
[ ]	left square bracket, right square bracket	start/end option symbols
{ }	left curly bracket, right curly bracket	start/end repeat symbols
? ?	question mark	special sequence symbol
-	hyphen-minus	except symbol
,	comma	concatenate symbol
=	equals sign	defining symbol
|	vertical line	definition separator symbol
*	asterisk	repetition symbol
;	semicolon	terminator symbol

EXAMPLE 1 The following syntax rules illustrate repetition (asterisk and curly brackets) and option square brackets:

```
aa = "A" ;
bb = 3 * aa, "B" ;
cc = 3 * [aa], "C" ;
dd = {aa}, "D" ;
ee = aa, {aa}, "E" ;
ff = 3 * aa, 3 * [aa], "F" ;
```

Terminal strings defined by these rules are as follows:

```
aa: A
bb: AAAB
```

2) A GPD conforming program might be an 11404 GPD datatype definition or a data declaration based upon an 11404 GPD datatype.

```
cc: C AC AAC AAAC
dd: D AD AAD AAAD AAAAD  etc.
ee: AE AAE AAAE AAAAE AAAAAE  etc.
ff: AAAF AAAAF AAAAAF AAAAAAF
```

EXAMPLE 2 The following syntax rules illustrate a definitions list (vertical line), an exception (hyphen-minus), and comments (parentheses and asterisks):

```
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" ;
vowel = "A" | "E" | "I" | "O" | "U" ;
consonant = letter - vowel ;        (* These examples are from ISO/IEC 14977 *)
```

Terminal strings defined by these rules are as follows:

```
letter:    A B C D E F G H I J  etc.
vowel:     A E I O U
consonant: B C D F G H J K L M  etc.
```

5.2 Text conventions

Within the text:

— A reference to a terminal symbol syntactic object consists of the terminal symbol in fixed width courier, e.g. `type`.

— A reference to a non-terminal symbol syntactic object consists of the non-terminal-symbol in fixed width italic courier, e.g. *`type-declaration`*.

— Mathematical notation, properties, and characterizing operations are in bold, e.g., **InOrder(x,y)**.

— 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.

The term *general-purpose datatype* is used to mean a datatype defined by this International Standard. The term *general-purpose datatypes* (plural) refers to some or all of the datatypes defined by this International Standard. The term *GPD datatype* refers to datatypes generated or defined by means specified in 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.

NOTE 1 This International Standard defines the concept “value space”, which is just a set of values. It extends that notion to “datatype” by adding computational properties supported by characterizing operations. ISO/IEC 11179, *Information technology — Metadata registries (MDR)*, introduces the concept “value domain”. A “value domain” is a set of <*value*, *meaning*> pairs, each pair consisting of a value and its conceptual interpretation. That is, ISO/IEC 11179 extends the notion value space to “value domain” by adding its meaning for users and applications.

A distinct value may belong to the value space of more than one datatype, so long as it properly supports the properties and characterizing operations of each of them (see 6.6).

A value space contains regular values (elements of a value space that are consistent with a datatype's properties and characterizing operations). A datatype may also have sentinel values: elements that can be said to 'belong' to the datatype but that may not be completely consistent with the properties and characterizing operations of the datatype. For the purpose of this International Standard, sentinel values do not belong to the value space of the datatype. If a datatype has sentinel values, then there shall always be a form of the Equal operator to distinguish these sentinel values from regular values (see also 8.2.6).

NOTE 2 A numeric datatype, which includes characterizing operations such as **Equal** and **InOrder**, may include sentinel values such as **not-a-number**, **indeterminate**, **not-applicable**, **+infinity**, **-infinity**, and so on. These characterizing operations are not defined for sentinel values.

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 data units which are appropriate to that kind of manipulation. It is “abstract” in the sense that it deals with the perceived properties of the data units themselves, rather than with the properties of their representations in computer systems.

NOTE 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.

NOTE 2 Some computational properties derive from the need *for the data 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 data units*.

NOTE 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 data 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 ≤, 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 $U1$ and $U2$ are both upper bounds of the value space, then $U1 \leq U2$ and $U2 \leq U1$, and therefore $U1 = U2$, 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 evinces 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 axiomatically 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 that 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 that distinguish this datatype generator from other datatype generators which produce identical value spaces from identical parametric datatypes.

NOTE 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 $Add(red, yellow) \neq green$

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.

NOTE 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.

NOTE 3 The characterizing operations on a datatype may be:

— niladic operations which yield values of the given datatype,

— monadic operations which map a value of the given datatype into a value of the given datatype or into a value of datatype Boolean,

— 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,

— n-adic operations [3] 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.

NOTE 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.

NOTE 5 **Equal** is always a characterizing operation on datatypes with the equality property.

NOTE 6 **InOrder** is always a characterizing operation on ordered datatypes (see 6.3.2).

6.7 Datatype families

If there is a 1-to-1 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.

3) The term "n-adic" is a general term, which includes niladic, monadic, and dyadic.

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 6.8.6).

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*.

NOTE 1 Homogeneous aggregates view all their elements as serving the same role or purpose. Heterogeneous aggregates divide their elements into different roles.

NOTE 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). The `choice` datatype generator 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 identifier uniqueness

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

6.8.5 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, e.g. is `"computation" < "Computer"`?

6.8.6 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 *uniqueness*), or by some implementation-dependent selection mechanism, modeled as random selection.

NOTE 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.

NOTE 2 The `sequence` datatype generator (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.

6.8.7 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 , a recursive capability 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.

6.8.8 Structured and unstructured

Aggregate datatypes are:

— conceptually structured, having both the component datatypes and the access method specified, or

— conceptually semi-structured, having either the component datatypes or the access method specified, but not both, or

— conceptually unstructured, having neither the component datatype nor the access method specified.

6.8.9 Mandatory and optional components

The components of an aggregate datatype may not all be required to have a valid value of the datatype, i.e., the actual value space of the datatype may include values for which some of the component values are unspecified.

When a component of the datatype is required to have a valid value in order for the aggregate value to be a valid value of the datatype, the component is said to be a *mandatory component*.

When a component of the datatype is not required to have a valid value in order for the aggregate value to be a valid value of the datatype, the component is said to be an *optional component*.

NOTE 1 This property applies to fields of records, members of classes, and elements of sequences, tables, and arrays.

NOTE 2 See examples in 6.9.

6.9 Provisions associated with datatypes

A *provision* is the fundamental unit of normative wording [4)] in a normative document, such as a standard or specification. A provision is an “expression of normative wording that takes the form of a statement, an instruction, a recommendation or a requirement”. Auxiliary verbs such as “shall” (mandatory requirement), “should” (recommendation), and “may” (optional requirement) are used in normative wording to express provisions.

This International Standard contains many provisions. Some provisions apply to datatypes in general, e.g., a datatype consists of a value space, properties, and characterizing operations — a “statement” provision. Some provisions apply to specific datatypes, e.g., a mapping to the GPD **`integer`** datatype shall be a datatype that is numeric — a “requirement” provision. Declarations may contain provisions described via annotations (outside the scope of this International Standard). Declarations may contain provisions associated with datatype families, as described by the **`provision()`** ***`type-attribute`***.

4) Provisions, in general, may be expressed in natural language text and/or specialized notation. For the GPD "provision()", the provision is expressed as a set of name-value pairs.

A *normative datatype* is a collection of specifications for datatype properties that may be simultaneously satisfied by more than one actual datatype. A related concept concerns *conformity* to a normative datatype: a datatype conforms to a normative datatype if it satisfies all of the properties specified by the normative datatype, i.e., a normative datatype does not have a specific value space, but it may specify properties that any conforming value space must have. Similarly, a normative datatype may specify operations that must be supported by a conforming datatype, without that set of operations itself being sufficient to characterize any one datatype.

EXAMPLE 1 The normative datatype **Any** can be satisfied by any GPD datatype, with any value space. The only requirement is that **Equal** is defined on the value space.

EXAMPLE 2 The normative datatype `address_label_standard` is a `record` that contains 6 components.

```
// shorthand for "mandatory data element" provision
normative MDE = provision(obligation=require, target=type, scope=identifier, subset=defined),

// shorthand for "optional data element" provision
normative ODE = provision(obligation=permit, target=type, scope=identifier, subset=defined),

// shorthand for "extended data element" provision
normative XDE = provision(obligation=permit, target=type, scope=identifier, subset=undefined),

normative address_label_standard =
record XDE
(
      name MDE: characterstring,
      address MDE: characterstring,
      city MDE: characterstring,
      state_province MDE: characterstring,
      postal_code MDE: characterstring,
      country_code ODE: characterstring,
),
```

It is not possible to instantiate a normative datatype directly, but it is possible to instantiate an implementation (of the normative datatype) that conforms to the normative datatype. The following are examples of datatypes (implementations) that conform to the normative datatype `address_label_standard`. It is possible to instantiate the following datatypes.

```
// address_label_1 conforms because it has all the mandatory data elements
type address_label_1 =
record
(
      name: characterstring,  // mandatory data element
      address: characterstring,  // mandatory data element
      city: characterstring,  // mandatory data element
      state_province: characterstring,  // mandatory data element
      postal_code: characterstring,  // mandatory data element
),

// address_label_2 conforms because it has all the mandatory data elements,
// and the optional data element (present in address label 2) conforms to
// the requirements in the normative datatype
type address_label_2 =
record
(
      name: characterstring,
      address: characterstring,
      city: characterstring,
      state_province: characterstring,
      postal_code: characterstring,
      country_code: characterstring,  // optional data element
),

// address_label_3 conforms because it has the data elements
// of address_label_2 and the XDE permits the definition of
// additional data elements
type address_label_3 =
record
(
      name: characterstring,
      address: characterstring,
      city: characterstring,
      state_province: characterstring,
```

```
        postal_code: characterstring,
        country_code: characterstring,
        telephone_number: characterstring, // extended data element
),
```

The following are examples of datatypes that do not conform to the datatype.

```
// address_label_4 does not conform because it is missing
// mandatory data elements "state_province" and "postal_code"
type address_label_4 =
record
(
        name: characterstring,
        address: characterstring,
        city: characterstring,
),

// address_label_5 does not conform because its optional data element
// conflicts with the definition of the normative datatype
type address_label_5 =
record
(
        name: characterstring,
        address: characterstring,
        city: characterstring,
        state_province: characterstring,
        postal_code: characterstring,
        country_code: integer,
),
```

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 — Language-Independent Procedure Calling (LIPC)*.[5] 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.

```
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"  |  "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             =  "("  |   (* left parenthesis *)
                       ")"  |   (* right parenthesis *)
                       "."  |   (* full stop *)
                       ","  |   (* comma *)
                       ":"  |   (* colon *)
                       ";"  |   (* semicolon *)
                       "="  |   (* equals sign *)
                       "/"  |   (* solidus *)
                       "*"  |   (* asterisk *)
```

5) The IDN is only one feature of ISO/IEC 13886. The primary purpose of ISO/IEC 13886 is to specify a technique for language-independent procedure calling.

```
                                "-"  |   (* hyphen-minus *)
                                "{"  |   (* left curly bracket *)
                                "}"  |   (* right curly bracket *)
                                "["  |   (* left square bracket *)
                                "]"  ;   (* right square bracket *)
underscore                  =   "_"  ;   (* low line *)
apostrophe                  =   "'"  ;   (* apostrophe *)
quote                       =   '""'  ;   (* quotation mark *)
escape                      =   "!"  ;   (* exclamation point *)
space                       =   " "      (* space *)
non-quote-character         =   letter  |
                                digit  |
                                special  |
                                underscore  |
                                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, 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.

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 character-set. An added-character may be referenced by name, by 8-digit short UCS identifier, or by 4-digit short UCS identifier, as specified by ISO/IEC 10646. For example, `"!QUOTATION MARK!"`, `"!U00000022!"`, and `"!U+0022!"` are all equivalent: a string literal that contains the one character, a quotation mark.

7.2 Whitespace

A sequence of one or more space characters, horizontal tabs, end of line characters, or newline 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 vertical tabulators, end of page indicators, etc.

A comment is either of:

— Any sequence of characters beginning with the sequence `/*` (solidus, asterisk) and terminating with the first occurrence thereafter of the sequence `*/` (asterisk solidus).

— Any sequence of characters beginning with the sequence `//` (solidus, solidus) and terminating with the occurrence thereafter of end-of-line character 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              =  initial-letter-like,  {  pseudo-letter-like  } ;
initial-letter-like     =  letter-like  |
                           special-like ;
letter-like             =  letter  |
                           ISO/IEC-10176-extended-letter ;
pseudo-letter-like      =  letter-like  |
                           digit-like  |
                           underscore ;
digit-like              =  digit  |
                           ISO/IEC-10176-extended-digit ;
special-like            =  underscore |
                           ISO/IEC-10176-extended-special ;
```

NOTE ISO/IEC 10176 describes the notion of classes of letter-like characters (outside of ISO/IEC 646 letters A through Z), digit-like characters (outside of ISO/IEC 646 digits 0 through 9), and special characters that are all used in identifiers.

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, say, 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-like,  {  digit-like  }  ;
digit-like              =  digit  |
                           ISO/IEC-10176-extended-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       =  apostrophe,  any-character,  apostrophe ;
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 below 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.

```
reserved-keywords       =  "array"  |
                           "choice"  |
                           "default"  |
                           "excluding"  |
                           "from"  |
                           "in"  |
                           "inout"  |
                           "new"  |
                           "of"  |
                           "out"  |
                           "plus"  |
                           "pointer"  |
                           "procedure"  |
                           "raises"  |
                           "record"  |
                           "returns"  |
                           "selecting"  |
                           "size"  |
                           "subtype"  |
                           "table"  |
                           "termination"  |
                           "to"  |
                           "type"  |
                           "value"  ;
```

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 labeling annotations. Following ISO/IEC 8824, every value of the **objectidentifier** datatype must have

as its first component one of `iso`, `itu-t`, or `joint-iso-itu-t`, 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 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 general-purpose 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   |
                           class-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 Clause 8 and Clause 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.

NOTE 1 Two syntactically different ***explicit-value***s may designate the same value, such as ***rational-literal***s 3/4 and 6/8, or **set of (integer)** values (1,3,4) and (4,3,1).

NOTE 2 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.

NOTE 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-reference***s, 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 nth ***component-reference*** is the last ***component-reference*** of the ***dependent-value***, the ***dependent-value*** shall refer to the value to which the nth ***component-reference*** refers.

2. If the nth ***component-reference*** is not the last ***component-reference***, then the datatype of the nth ***component-reference*** shall be a **record** datatype or a **pointer** datatype.

3. If the nth ***component-reference*** is not the last ***component-reference***, and the datatype of the nth ***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 nth ***component-reference***.

4. If the nth ***component-reference*** is not the last ***component-reference***, and the datatype of the nth ***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 nth ***component-reference***.

NOTE 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.

NOTE 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.

NOTE 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.

7.6 GPD program text

A ***program-text*** designates a collection of GPD statements.

Syntax:

```
program-text           = { program-statement, "," };
program-statement      = type-specifier  |
                         declaration  |
                         normative-datatype-declaration  ;
```

8 Datatypes

This clause defines the collection of general-purpose datatypes. A general-purpose 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 general-purpose 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-declaration***s, as provided by clause 9.1.3.

This clause also provides syntax for the identification of values of general-purpose datatypes and their generated datatypes. Notations for values of datatypes are required in the syntactic designations for subtypes and for some primitive datatypes.

NOTE 1 For convenience, or correctness, some datatypes and characterizing operations are defined in terms of other general-purpose datatypes. The use of a general-purpose datatype defined in this clause always refers to the datatype so defined.

NOTE 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 general-purpose datatypes shall be defined by this International Standard.

Syntax:

```
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-definition*** 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*)
false	**false**	**true**
false	**true**	**false**
true	**false**	**false**
true	**true**	**true**

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

x	**Not**(*x*)
false	**true**
true	**false**

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

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

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,  ")"  ;
state-value             =  state-value-list  |
                           value-space-source  ;
state-value-list        =  state-literal,  {  ",",  state-literal  }  ;
state-literal           =  identifier  ;
value-space-source      =  "import",  list-source-reference  ;
list-source-reference   =  identifier  |
                           '"', URI-text, '"'  ;
URI-text                =  '"', URI defined by IETF RFC2396, '"'  ;
```

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

Values: When the state-value-list form of state-values is used, 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`***. When the ***`value-space-source`*** form is used, the value set shall be exactly the set of code values specified in the document identified by the list-source value. When the list-source is a URI-value, it shall denote a valid value of the URI datatype. When the list-source is an **`objectidentifier`**-value, it shall denote a valid value of the **`objectidentifier`** datatype, as defined in 10.1.10. In either case, the list-source value shall identify a document that explicitly defines a set of code values and their denotations.

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,  ")" ;
enumerated-value        =  enumerated-value-list  |
                           URI-to-value-space ;
enumerated-value-list   =  enumerated-literal,  {  ",",  enumerated-literal  }  ;
enumerated-literal      =  identifier  ;
URI-to-value-space      =  (* URI defined by RFC 2396 *)  ;
```

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***, which shall be referred to as the naming sequence of the enumerated datatype.

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

InOrder(*x***,** *y***: enumerated(***enumerated-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(***enumerated-value-list***)): enumerated(***enumerated-value-list***)** is

if for all *y*: **enumerated(***enumerated-value-list***)**, $x \leq y$ implies $x = y$, then **undefined**;

else the value *y*: **enumerated(***enumerated-value-list***)**, such that $x < y$ and for all $z \neq x$, $x \leq z$ implies $y \leq z$.

NOTE 1 Other uses of the IDN syntax make stronger requirements on the uniqueness of ***enumerated-literal*** identifiers.

NOTE 2 The ordering on enumerated types imposed by programming languages is a convenience that allows programs to reference all the values via for-loops and enables the compiler to use integer encodings to simplify implementation. Properly, the enumerated type should be chosen over the state type only when the ordering has semantic value. However, it may be necessary to declare the datatype of an object to be an enumerated GPD when the purpose is to ensure the correct interpretation of an integer-based implementation.

EXAMPLE Enumerated types **(short, medium, tall)** and **(light, medium, heavy)** are distinct types of the family "enumerated", even though they have exactly the same number of elements, and the same characterizing operations: **Equal** and **InOrder**. Enumerated types **(short, medium, tall)** and **(short, moderate, medium, tall)** are distinct types. It is outside the scope of this International Standard whether or not the value medium is the same in both enumerated types.

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  |
                           special  |
                           underscore  |
                           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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 4 The terms character-set, member, symbol and character-name are those of ISO/IEC 10646, but there should be analogous notions in any character set referenceable by a repertoire-identifier.

NOTE 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. 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.

NOTE 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/IEC 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 part 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/IEC 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-literal***s 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**, that $x < y$ and for all $z \neq x$, $x \leq z$ implies $y \leq z$.

8.1.6 Date-and-Time

Description: **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 $\mathbf{radix}^{(-\mathbf{factor})}$ of the specified ***time-unit***. ***time-unit***, and ***radix*** and ***factor*** if present, shall not be ***formal-parametric-value***s 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. The ***time-literal*** denotes the date-and-time value specified by the characterstring as interpreted under ISO 8601.

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

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

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

InOrder(*x***, *y*: time(*time-unit*, *radix*, *factor*)): boolean** is **true** if the point in time designated by y does not precede that designated by x; else **false**.

Difference(*x*, *y*: **time**(*time-unit*, *radix*, *factor*)): **timeinterval**(*time-unit*, *radix*, *factor*) is:

if **InOrder**(*x*, *y*), then the number of *time-unit*s of the specified resolution elapsing between the time x and the time y; else, let z be the number of *time-unit*s elapsing between the time y and the time x, then **Negate**(*z*).

Extend.res1tores2(*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.res1tores2(*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**, **Dayof-Month**, **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 $Add(x,1) \neq y$ 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 $Add(x, y) = 0$.

NonNegative(*x*: integer): boolean is

true if $x = 0$ or x can be developed by one or more iterations of adding 1 to 0,

i.e. if $x = Add(1, Add(1, \ldots Add(1, Add(1,0))\ldots))$;

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 **Multiply(*y*,*z*)** $\leq x$.

Remainder(*x*, *y*: integer): integer, where $0 \leq x$ and $0 < y$, **= Add(*x*, Negate(Multiply(*y*, 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:

Multiply(Promote(*signed-number*), Reciprocal(Promote(*number*))).

Properties: **ordered**, **exact**, **numeric**, **unbounded**.

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

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

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

For every rational value k, there is a non-negative integer n, such that **Multiply(*n*,*k*)** is an integral value, and:

NonNegative(*k*) = integer.NonNegative(Multiply(*n*,*k*)).

InOrder(*x*, *y*: rational): boolean = NonNegative(Add(*x*, Negate(*y*)))

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.

Negate(*x*: **rational**): **rational** is the value *y*: **rational** such that $Add(x, y) = 0$.

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

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

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-value`*s 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*)))

EXAMPLE 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)**.

EXAMPLE 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.

NOTE 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.

NOTE 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, $n \cdot r^{(-f)} < v < (n+1) \cdot r^{(-f)}$.

NOTE 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.

NOTE 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.

NOTE 5 The value space of a scaled datatype contains the multiplicative identity (1) if and only if $factor \geq 0$.

NOTE 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.

NOTE 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 expressed to some finite precision and must be distinguishable to at least that precision.

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 greater than 0. ***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 $\Re$. denote the mathematical real value space and for v in $\Re$, let $|v|$ denote the absolute value of v. Let V denote the value space of datatype **`real(radix, factor)`**, and let $\varepsilon = radix^{(-factor)}$. Then V shall be a subset of $\Re$. with the following properties:

– **0** is in V;

– for each r in $\Re$. such that $|r| \geq \varepsilon$, there exists at least one r' in V such that $|r - r'| \leq |r| \cdot \varepsilon$;

– for each r in $\Re$. such that $|r| < \varepsilon$, there exists at least one r' in V such that $|r - r'| \leq \varepsilon^2$.

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 $\Re$ into a corresponding 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 $\Re$, 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 • 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.

NOTE 1 The general-purpose 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.

NOTE 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, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*.[6] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems*, 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 greater than 0. **`radix`** and **`factor`** shall not be ***`formal-parametric-value`***s 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 $\varepsilon = radix^{(-factor)}$. Then V shall be a subset of C with the following properties:

– **0** is in V;

– for each v in C such that $|v| \geq \varepsilon$, there exists at least one v' in V such that $|v - v'| \leq |v| \cdot \varepsilon$;

– for each v in C such that $|v| < \varepsilon$, there exists at least one v' in V such that $|v - v'| \leq \varepsilon^2$.

6) The ISO/IEC 10967 series provides a common model of arithmetic in programming and database languages.

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: $M(realpart + (imaginarypart \cdot i))$, where $+$ is the additive operation on the mathematical complex numbers and • 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 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 • 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.

SquareRoot(*x*: **complex**(*radix*, *factor*)): **complex**(*radix*, *factor*) = $M(y)$, where y is one of the two mathematical complex values such that $y \cdot y = x$. Every complex number can be uniquely represented in the form $a + b\,i$, where i is the "principal square root" of -1, in which a is designated the real part and b is designated the imaginary part. The y value used is that in which the real part of y is positive, if any, else that in which the real part of y is zero and the imaginary part is non-negative.

NOTE Detailed requirements for the approximation function, its relationship to the characterizing operations, and the implementation of the characterizing operations in languages are to be provided by Parts of ISO/IEC 10967, *Information technology — Language independent arithmetic*.

8.1.12 Void

Description: **void** is the datatype representing an object whose presence is syntactically or semantically required, but carries no information in a given instance.

Syntax:

```
void-type                =  "void"  ;
```

Parametric Values: none.

Values: Conceptually, the value space of the void datatype is empty, but a single nominal value is necessary to perform the “presence required” function.

Value-syntax:

```
void-literal            =  "nil"  ;
```

“nil” is the syntactic representation of an occurrence of void as a value.

Properties: none.

Operations: **Equal**.

Equal(x, y: void) = true;

NOTE 1 The `void` datatype is used as the implicit type of the result parameter of a procedure datatype (8.3.3) which returns no value, or as an alternative of a choice datatype (8.3.1) when that alternative has no content.

NOTE 2 The `void` datatype is represented in some languages as a record datatype (see 8.4.1) which has no fields. In this International Standard, the void datatype is not a record datatype, because it has none of the properties or operations of a record datatype.

NOTE 3 Like the motivation for the `void` datatype itself, Equal is required in order to support the comparison of aggregate values containing `void` and it must yield **true**.

NOTE 4 The “empty set” is not a value of datatype `void`, but rather a value of the appropriate set datatype (see 8.4.2).

8.2 Subtypes and extended types

A subtype is a datatype derived from an existing datatype, designated the base datatype, by restricting the value space to a subset of that of the base datatype whilst maintaining all characterizing operations. Subtypes are created by a kind of datatype generator which is unusual in that its only function is to define the relationship between the value spaces of the base datatype and the subtype.

Syntax:

```
subtype                 =  range-subtype  |
                           selecting-subtype  |
                           excluding-subtype  |
                           size-subtype  |
                           explicit-subtype  |
                           extended-type  ;
```

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

Description: prose description of the subtype value space.

Syntax: the syntactic production for a subtype resulting from the subtype generator, including identification of all parametric values which are necessary for the complete identification of a distinct subtype.

Components: constraints on the base datatype and parametric values.

Values: formal definition of resulting value space.

Properties: all datatype properties are the same in the subtype as in the base datatype, except possibly the presence and values of the bounds. This entry therefore defines only the effects of the subtype generator on the bounds.

All characterizing operations are the same in the subtype as in the base datatype, but the domain of a characterizing operation in the subtype may not be identical to the domain in the base datatype. Those values from the value space of the subtype which, under the operation on the base datatype, produce result values which lie outside the value space of the subtype, are deleted from the domain of the operation in the subtype.

8.2.1 Range

Description: **range** creates a subtype of any ordered datatype by placing new upper and/or lower bounds on the value space.

Syntax:

```
range-subtype          =  base,  "range",  "(",  select-range,  ")"  ;
select-range           =  lowerbound,  "..",  upperbound  ;
lowerbound             =  value-expression  |
                          "*"  ;
upperbound             =  value-expression  |
                          "*"  ;
base                   =  type-specifier  ;
```

Components: Base shall designate an ordered datatype. When *lowerbound* and *upperbound* are *value-expressions*, they shall have values of the base datatype such that **InOrder(lowerbound, upperbound)**. When *lowerbound* is "*", it indicates that no lower bound is being specified, and when *upperbound* is "*", it indicates that no upper bound is being specified. *lowerbound* and *upperbound* shall not be *formal-parametric-value*s, except in some occurrences in declarations (see 9.1).

Values: all values **v** from the base datatype such that **lowerbound** ≤ **v**, if **lowerbound** is specified, and **v** ≤ **upperbound**, if upper-bound is specified.

Properties: The subtype is bounded (above, below, both) if the base datatype is so bounded or if the *select-range* specifies the corresponding bounds.

8.2.2 Selecting

Description: *selecting* creates a subtype of any exact datatype by enumerating the values in the subtype value-space.

Syntax:

```
selecting-subtype      =  base,  "selecting",  "(",  select-list,  ")"  ;
select-list            =  select-item,  {  ",",  select-item  }  ;
select-item            =  value-expression  |
                          select-range  ;
select-range           =  lowerbound,  "..",  upperbound  ;
lowerbound             =  value-expression  |
                          "*"  ;
upperbound             =  value-expression  |
                          "*"  ;
base                   =  type-specifier  ;
```

Components: *base* shall designate an exact datatype. When the *select-items* are *value-expressions*, they shall have values of the *base* datatype, and each value shall be distinct from all others in the *select-list*. A *select-item* shall not be a *select-range* unless the *base* datatype is ordered. When *lowerbound* and *upperbound* are *value-expressions*, they shall have values of the *base* datatype such that **InOrder(lowerbound, upperbound)**. When *lowerbound* is *"*"*, it indicates that no lower bound is being specified, and when *upperbound* is *"*"*, it indicates that no upper bound is being specified. No *value-expression* occurring in the *select-list* shall be a *formal-parametric-value*, except in some occurrences in declarations (see 9.1).

Values: The values specified by the *select-list* designate those values from the *value-space* of the *base* datatype which comprise the value-space of the *selecting* subtype. A *select-item* which is a *value-expression* specifies the single value designated by that *value-expression*. A *select-item* which is a

select-range specifies all values **v** of the *base* datatype such that **lowerbound** ≤ **v**, if *lowerbound* is specified, and **v** ≤ **upperbound**, if *upperbound* is specified.

Properties: The subtype is bounded (above, below, both) (1) if the *base* datatype is so bounded, or (2) if no *select-range* appears in the *select-list*, or (3) if all *select-range*s in the *select-list* specify the corresponding bounds.

8.2.3 Excluding

Description: *excluding* creates a subtype of any exact datatype by enumerating the values which are to be excluded in constructing the subtype value-space.

Syntax:

```
excluding-subtype       =  base,  "excluding",  "(",  select-list,  ")"  ;
select-list             =  select-item,  {  ",",  select-item  }  ;
select-item             =  value-expression  |
                           select-range  ;
select-range            =  lowerbound,  "..",  upperbound  ;
lowerbound              =  value-expression  |
                           "*"  ;
upperbound              =  value-expression  |
                           "*"  ;
base                    =  type-specifier  ;
```

Components: *base* shall designate an exact datatype. A *select-item* shall not be a *select-range* unless the *base* datatype is ordered. When *lowerbound* and *upperbound* are *value-expressions*, they shall have values of the **base** datatype such that **InOrder(lowerbound, upperbound)**. When *lowerbound* is `"*"`, it indicates that no lower bound is being specified, and when *upperbound* is `"*"`, it indicates that no upper bound is being specified. No *value-expression* occurring in the *select-list* shall be a *formal-parametric-value*, except in some occurrences in declarations (see 9.1).

Values: The value space of the `excluding` subtype comprises all values of the *base* datatype except for those specified by the *select-list*. A *select-item* which is a *value-expression* specifies the single value designated by that *value-expression*. A *select-item* which is a *select-range* specifies all values **v** of the *base* datatype such that **lowerbound** ≤ **v**, if a lower bound is specified, and **v** ≤ **upperbound**, if an upper bound is specified.

Properties: The subtype is bounded (above, below, both) if the *base* datatype is so bounded or if some *select-range* appears in the *select-list* and does not specify the corresponding bound.

8.2.4 Size

Description: `size` creates a subtype of any `sequence`, `set`, `bag`, or `table` datatype by specifying bounds on the number of elements any value of the `base` datatype may contain.

Syntax:

```
size-subtype            =  base,  "size",  "(",  minimum-size,
                               [  "..",  maximum-size  ],  ")"  ;
minimum-size            =  value-expression  ;
maximum-size            =  value-expression  |
                           "*"  ;
base                    =  type-specifier  ;
```

Components: `base` shall designate a generated datatype resulting from the `sequence`, `set`, `bag`, or `table` generator, or from a `new` datatype generator whose value space is constructed by such a generator (see 9.1.3). *minimum-size* shall have an integer value greater than or equal to zero, and *maximum-size*, if it is a *value-expression*, shall have an integer value such that *minimum-size* ≤ *maximum-size*. If *maximum-size* is omitted, the maximum size is taken to be equal to the *minimum-size*, and if *maximum-size* is `"*"`, the maximum size is taken to be unlimited. *minimum-size* and *maximum-size* shall not be *formal-parametric-value*s, except in some occurrences in declarations (see 9.1).

Values: The value space of the subtype consists of all values of the ***base*** datatype which contain at least ***minimum-size*** values and at most ***maximum-size*** values of the element datatype.

Subtypes: Any ***size*** subtype of the same base datatype, such that ***base-minimum-size*** ≤ ***subtype-minimum-size***, and ***subtype-maximum-size*** ≤ ***base-maximum-size***.

Properties: those of the base datatype; the aggregate subtype has fixed size if the maximum size is (explicitly or implicitly) equal to the minimum size.

8.2.5 Explicit subtypes

Description: Explicit subtyping identifies a datatype as a subtype of the base datatype and defines the construction procedure for the subset value space in terms of general-purpose datatypes or datatype generators.

Syntax:

```
explicit-subtype        =  base,  "subtype",  "(",  subtype-definition,  ")"  ;
base                    =  type-specifier  ;
subtype-definition      =  type-specifier  ;
```

Components: **base** may designate any datatype. The ***subtype-definition*** shall designate a datatype whose value space is (isomorphic to) a subset of the value space of the ***base*** datatype.

Values: The subtype value space is identical to the value space of the datatype designated by the ***subtype-definition***.

Properties: exactly those of the subtype-definition datatype.

NOTE 1 When the ***base*** datatype is generated by a datatype generator, the ways in which a subset value space can be constructed are complex and dependent on the nature of the ***base*** datatype itself. Clause 8.3 specifies the subtyping possibilities associated with each datatype generator.

NOTE 2 It is redundant, but syntactically acceptable, for the subtype-definition to be an occurrence of a subtype-generator, e.g.

```
integer subtype (integer selecting(0..5))
```

8.2.6 Extended

Description: Extended creates a datatype whose value-space contains the value-space of the base datatype as a proper subset.

Syntax:

```
extended-type           =  base,  [ "plus",  "sentinel" ],
                               "(",  extended-value-list,  ")"  ;
extended-value-list     =  extended-value,  {  ",",  extended-value  }  ;
extended-value          =  extended-literal  |
                           formal-parametric-value  ;
extended-literal        =  identifier  ;
base                    =  type-specifier  ;
```

Components: ***base*** may designate any datatype. An ***extended-value*** shall be an ***extended-literal***, except in some occurrences in declarations (see 9.1). Each ***extended-literal*** shall be distinct from all ***value-literals*** and ***value-identifiers***, if any, of the ***base*** datatype and distinct from all others in the ***extended-value-list***.

Values: The value space of the **extended** datatype comprises all values in the value-space of the **base** datatype plus those additional values specified in the ***extended-value-list***. If **sentinel** is included in the type specification, the additional values are sentinel values in the value space.

NOTE 1 The value space of a datatype is the set of values specified in the definition of the datatype. Sentinel values are values that can occur wherever values of the value space can occur; they can be distinguished by **Equal** from values in the value space. Sentinel values must be specified explicitly even for a datatype that is defined axiomatically. For example, it follows that **{short, medium, tall}** and **{short, medium, tall, sentinels = Unknown, Unspecified}** are two distinct datatypes with the same value space.

Properties: The subtype is bounded (above, below, both) if the base datatype is so bounded or if the additional values are upper or lower bounds.

The definition of an **extended** datatype shall include specification of the characterizing operations on the ***base*** datatype as applied to, or yielding, the added values in the ***extended-value-list***. In particular, when the ***base*** datatype is ordered, the behavior of the **InOrder** operation on the added values shall be specified.

NOTE 2 **extended** produces a subtype relationship in which the ***base*** datatype is the subtype and the **extended** datatype has the larger value space.

NOTE 3 Other uses of the IDN syntax make stronger requirements on the uniqueness of extended-literal identifiers.

8.3 Generated datatypes

A *generated datatype* is a datatype resulting from an application of a 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. The datatypes on which a datatype generator operates are said to be its *parametric* or *component datatypes*. The generated datatype is semantically dependent on the parametric datatypes, but has its own characterizing operations. An important characteristic of all datatype generators is that the generator can be applied to many different parametric datatypes. The Pointer and Procedure generators generate datatypes whose values are atomic, while Choice and the generators of aggregate datatypes generate datatypes whose values admit of decomposition. A generated-type designates a generated datatype.

Syntax:

```
generated-type          =  pointer-type   |
                           procedure-type |
                           choice-type    |
                           aggregate-type |
                           import-type    ;
```

This International Standard defines common datatype generators by which an application of this International Standard may define generated datatypes. (An application may also define “new” generators, as provided in clause 9.1.3.) Each datatype generator is defined by a separate subclause. The title of each such subclause gives the informal name for the datatype generator, and the datatype generator is defined by a single occurrence of the following template:

Description: prose description of the datatypes resulting from the generator.

Syntax: the syntactic production for a generated datatype resulting from the datatype generator, including identification of all parametric datatypes which are necessary for the complete identification of a distinct datatype.

Components: number of and constraints on the parametric datatypes and parametric values used by the generator.

Values: formal definition of resulting value space.

Properties: properties of the resulting datatype which indicate its admissibility as a component datatype of certain datatype generators: numeric or non-numeric, approximate or exact, ordered or unordered, and if ordered, bounded or unbounded.

Subtypes: generators, subtype-generators and parametric values which produce subset value spaces.

Operations: characterizing operations for the resulting datatype which associate to the datatype generator. The definitions of operations have the form described in 8.1.

NOTE Unlike subtype generators, datatype generators yield resulting datatypes whose value spaces are entirely distinct from those of the component datatypes of the datatype generator.

8.3.1 Choice

Description: Choice generates a datatype called a choice datatype, each of whose values is a single value from any of a set of alternative datatypes. The alternative datatypes of a choice datatype are logically distinguished by their correspondence to values of another datatype, called the tag datatype.

Syntax:

```
choice-type             =  "choice",  "(",  [  field-identifier  ":"  ],
                               tag-type,  [  "="  discriminant  ],  ")"
                               "of",  "("  alternative-list  ")"  ;
field-identifier        =  identifier  ;
tag-type                =  type-specifier  ;
discriminant            =  value-expression  ;
alternative-list        =  alternative,  {  ",",  alternative  },
                               [  ",",  default-alternative  ]  ;
alternative             =  tag-value-list,  [  field-identifier  ],
                               ":",  alternative-type  ;
default-alternative     =  "default",  ":",  alternative-type  ;
alternative-type        =  type-specifier ;
tag-value-list          =  "(",  select-list,  ")"  ;
select-list             =  select-item,  {  ",",  select-item  }  ;
select-item             =  value-expression  |
                               select-range  ;
select-range            =  lowerbound,  "..",  upperbound  ;
lowerbound              =  value-expression  |
                               "*"  ;
upperbound              =  value-expression  |
                               "*"  ;
```

Components: Each *alternative-type* in the *alternative-list* may be any datatype. The *tag-type* shall be an exact datatype. The *tag-value-list* of each alternative shall specify values in the value space of the (tag) datatype designated by *tag-type*. A *select-item* shall not be a select-range unless the tag datatype is ordered. When *lowerbound* and *upperbound* are *value-expressions*, they shall have values of the tag datatype such that **InOrder(lowerbound, upperbound)**. When *lowerbound* is "*", it indicates that no *lowerbound* is being specified, and when *upperbound* is "*", it indicates that no *upperbound* is being specified. No *value-expression* in the *select-list* shall be a parametric value, except in some occurrences in declarations (see 9.1).

A **choice** datatype defines an association from the value space of the tag datatype to the set of alternative datatypes in the *alternative-list*, such that each value of the tag datatype associates with exactly one alternative datatype. The *tag-value-list* of an alternative specifies those values of the tag datatype which are associated with the alternative datatype designated by the *alternative-type* in the alternative. A *select-item* which is a *value-expression* specifies the single value of the tag datatype designated by that *value-expression*. A *select-item* which is a *select-range* specifies all values **v** of the tag datatype such that **lowerbound** ≤ **v**, if *lowerbound* is specified, and **v** ≤ **upperbound**, if *upperbound* is specified. The *default-alternative*, if present, specifies that all values of the tag datatype which do not appear in any other alternative are associated with the alternative datatype designated by its *alternative-type*.

No value of the tag datatype shall appear in the *tag-value-list* of more than one alternative.

The occurrence of a *field-identifier* before the *tag-type* or in an alternative has no meaning in the resulting *choice-type*. Its purpose is to facilitate mappings to programming languages.

The *discriminant*, if present, shall designate a value of the tag datatype. It identifies the tag value, or the source of the tag value, to be used in a particular occurrence of the choice datatype.

Values: all values having the conceptual form **(tag-value, alternative-value)**, where *tag-value* is a value of the tag datatype which occurs (explicitly or implicitly) in some alternative in the *alternative-list* and is uniquely mapped to an alternative datatype thereby, and alternative-value is any value of that alternative datatype.

Value-syntax:

```
choice-value            =  "(",  tag-value,  ":",  alternative-value,  ")"  ;
tag-value               =  independent-value  ;
alternative-value       =  independent-value  ;
```

A ***choice-value*** denotes a value of a choice datatype. The tag-value of a choice-value shall be a value of the tag datatype of the choice datatype, and the alternative-value shall designate a value of the corresponding alternative datatype. The value denoted shall be that value having the conceptual form (tag-value, alternative-value).

Properties: unordered, exact if and only if all alternative datatypes are exact, non-numeric.

Subtypes: any choice datatype in which the tag datatype is the same as, or a subtype of, the tag datatype of the base datatype, and the alternative datatype corresponding to each value of the tag datatype in the subtype is the same as, or a subtype of, the alternative datatype corresponding to that value in the base datatype.

Operations: **Equal**, **Tag**, **Cast**, **Discriminant**.

Discriminant(*x***: choice (***tag-type***) of (***alternative-list***)): tag-type** is the **tag-value** of the value x

Tag.type(*x***: type, *s*: tag-type): choice (***tag-type***) of (***alternative-list***)**, where **type** is that alternative datatype in **alternative-list** which corresponds to the value **s**, is that value of the choice datatype which has **tag-value** s and **alternative-value** x.

Cast.type(*x***: choice (***tag-type***) of (***alternative-list***)): type**, where **type** is an alternative datatype in **alternative-list**, is:

if the tag value of x selects an alternative whose alternative-type is **type**, then that value of **type** which is the (alternative) value of x, else **undefined**.

Equal(*x***, *y*: choice (***tag-type***) of (***alternative-list***)): boolean** is:

if **Discriminant(***x***) and Discriminant(***y***)** select the same alternative, then

type.Equal(Cast.type(*x***), Cast.type(***y***))**,

where **type** is the alternative datatype of the selected alternative and **type.Equal** is the **Equal** operation on the datatype type,

else false.

NOTE 1 The **choice** datatype generator is referred to in some programming languages as a "(discriminated) union" datatype, and in others as a datatype with "variants". The generator defined here represents the Pascal/Ada "variant-record" concept, but it allows the C-language "union", and similar discriminated union concepts, to be supported by a slight subterfuge. E.g. the C datatype:

```
union
{
      float a1;
      int a2;
      char* a3;
};
```

may be represented by:

```
choice ( state(a1, a2, a3) ) of
(
      (a1): real,
      (a2): integer,
      (a3): characterstring
)
```

NOTE 2 The actual value space of the tag datatype from which tag-values may be drawn is actually a subtype of the value space of the designated tag datatype, namely that subtype consisting exactly of the values which are mapped into alternative datatypes by the alternative-list. The set of tag values appearing explicitly or implicitly in the alternative-list is not required to cover the value space of the tag datatype.

NOTE 3 The subtypes of a choice datatype are typically choice datatypes with a smaller list of alternatives, and in the simplest case, the list is reduced to a single datatype.

NOTE 4 The operation **Discriminant** is a conceptual operation which reflects the ability to determine which alternative of a choice-type is selected in a given value. When a choice-value is moved between two contexts, as between a program and a data repository, representation of the chosen alternative is required, and most implementations explicitly incorporate the tag-value.

NOTE 5 Another useful model of `choice` is choice (field-list), where exactly one field is present in any given value, and the means of discrimination is not specified. In this model, the operation:

IsField.field(*x*: choice (*field-list*)): boolean = true if the designated **field** is present in the value x, otherwise **false**;

replaces **Discriminant**, with corresponding changes to the other characterizing operations. It is recognized that this model is mathematically more elegant (the Or-graph to match the And-graph of the fields in Record), but in practice, either IsField is not provided (which makes all operations user-defined) or IsField is implemented by tag-value (which makes IsField equivalent to Discriminant).

EXAMPLES See 10.2.2 and 10.2.3.

8.3.2 Pointer

Description: `pointer` generates a datatype, called a pointer datatype, each of whose values constitutes a means of reference to values of another datatype, designated the element datatype. The values of a pointer datatype are atomic.

Syntax:

```
pointer-type        =  "pointer", "to", "(", element-type, ")" ;
element-type        =  type-specifier ;
```

Components: Any single datatype, designated the element-type.

Values: The value space is that of an unspecified state datatype, each of whose values, save one, is associated with a value of the element datatype. The single value null may belong to the value space but it is never associated with any value of the element datatype.

Value-syntax:

```
pointer-literal     = "null" ;
```

`"null"` denotes the null value. There is no denotation for any other value of a pointer datatype.

Properties: unordered, exact, non-numeric.

Subtypes: any pointer datatype for which the element datatype is a subtype of the element datatype of the base pointer datatype.

Operations: **Equal**, **Dereference**.

Equal(*x*, *y*: pointer(*element*)): boolean is true if the values x and y are identical values of the unspecified state datatype, else **false**;

Dereference(*x*: pointer(element)): element, where $x \neq null$, is the value of the element datatype associated with the value x.

NOTE 1 A **pointer** datatype defines an association from the "unspecified state datatype" into the element datatype. There may be many values of the pointer datatype which are associated with the same value of the element datatype; and there may be members of the element datatype which are not associated with any value of the pointer datatype. The notion that there may be values of the "unspecified state datatype" to which no element value is associated, however, is an artifact of implementations – conceptually, except for null, those values of the (universal) "unspecified state datatype" which are not associated with values of the element datatype are not in the value space of the pointer datatype.

NOTE 2 Two pointer values are equal only if they are identical; it does not suffice that they are associated with the same value of the element datatype. The operation which compares the associated values is

Equal.element(Dereference(*x*), Dereference(*y*)),

where **Equal.element** is the **Equal** operation on the element datatype.

NOTE 3 The computational model of the pointer datatype often allows the association to vary over time. E.g., if *x* is a value of datatype pointer to (integer), then *x* may be associated with the value *0* at one time and with the value *1* at another. This implies that such pointer datatypes also support an operation, called assignment, which associates a (new) value of datatype e to a value of datatype pointer(e), thus changing the value returned by the **Dereference** operation on the value of datatype pointer to e. This assignment operation was not found to be necessary to characterize the pointer datatype, and listing it as a characterizing operation would imply that support of the pointer datatype requires it, which is not the intention.

NOTE 4 The term lvalue appears in some language standards, meaning "a value which refers to a storage object or area". Since the storage object is a means of association, an lvalue is therefore a value of some pointer datatype. Similarly, the implementation notion machine-address, to the extent that it can be manipulated by a programming language, is often a value of some pointer datatype.

NOTE 5 The hardware implementation of the "means of reference to" a value of the element-type is usually a memory cell or cells which contain a value of the ***element-type***. The memory cell has an "address", which is the "value of the unspecified state datatype". The memory cell physically maintains the association between the address (pointer-value) and the element-value which is stored in the cell. The **Dereference** operation is conceptually applied to the "address", but is implemented by a "fetch" from the memory cell. Thus in the computational model used here, the "address" and the "memory cell" are not distinguished: a pointer-value is both the cell and its address, because the cell can only be manipulated through its address. The cell, which is the pointer-value, is distinguished from its contents, which is the element-value.

NOTE 6 The notion "variable of datatype T" appears in programming languages and is usually implemented as a cell which contains a value of type T. Language standards often distinguish between the "address of the variable" and the "value of the variable" and the "name of the variable", and one might conclude that the "variable" is the cell itself. But all operations on such a "variable" actually operate on either the "address of the variable" — the value of general-purpose datatype "pointer to (T)" — or the "value of the variable" — the value of general-purpose datatype T. And thus those are the only objects which are needed in the datatype model. This notion is further elaborated in ISO/IEC 13886, which relates pointer-values to the "boxes" (or "cells") which are elements of the state of a running program.

8.3.3 Procedure

Description: **procedure** generates a datatype, called a procedure datatype, each of whose values is an operation on values of other datatypes, designated the parameter datatypes. That is, a procedure datatype comprises the set of all operations on values of a particular collection of datatypes. All values of a procedure datatype are conceptually atomic.

Syntax:

```
procedure-type          =  "procedure",  "(",  [  parameter-list  ],  ")",
                               [  "returns",  "(",  return-parameter,  ")",  ],
                               [  "raises",  "(",  termination-list,  ")"  ]  ;
parameter-list          =  parameter-declaration,
                               {  ",",  parameter-declaration  }  ;
parameter-declaration   =  direction  parameter  ;
direction               =  "in"  |
                           "out"  |
                           "inout"  ;
parameter               =  [  parameter-name,  ":"  ],  parameter-type  ;
parameter-type          =  type-specifier  ;
parameter-name          =  identifier  ;
```



```
return-parameter        =  [  parameter-name,  ":"  ],  parameter-type  ;
termination-list        =  termination-reference,
                               {  ",",  termination-reference  }  ;
termination-reference   =  termination-identifier  ;
```

Components: A ***parameter-type*** may designate any datatype. The ***parameter-name***s of ***parameter***s in the ***parameter-list*** shall be distinct from each other and from the parameter-name of the ***return-parameter***, if any. The ***termination-reference***s in the ***termination-list***, if any, shall be distinct.

Values: Conceptually, a value of a procedure datatype is a function which maps an input space to a result space. A ***parameter*** in the ***parameter-list*** is said to be an input parameter if its ***parameter-declaration*** contains the direction **"in"** or **"inout"**. The input space is the cross-product of the value spaces of the datatypes designated by the parameter-types of all the input parameters. A ***parameter*** is said to be a result parameter if it is the ***return-parameter*** or it appears in the ***parameter-list*** and its ***parameter-declaration*** contains the direction **"out"** or **"inout"**. The normal result space is the cross-product of the value spaces of the datatypes designated by the ***parameter-type***s of all the result parameters, if any, and otherwise the value space of the **void** datatype. When there is no ***termination-list***, the result space of the procedure datatype is the normal result space, and every value p of the procedure datatype is a function of the mathematical form:

$$p : I_1 \times I_2 \times \ldots \times I_n \rightarrow R_p \times R_1 \times R_2 \times \ldots \times R_m$$

where I_k is the value space of the parameter datatype of the k th input parameter, R_k is the value space of the parameter datatype of the k th result parameter, and R_p is the value space of the ***return-parameter***.

When a ***termination-list*** is present, each ***termination-reference*** shall be associated, by some ***termination-declaration*** (see 9.3), with an alternative result space which is the cross-product of the value spaces of the datatypes designated by the ***parameter-type***s of the ***parameter***s in the ***termination-parameter-list***. Let A_j be the alternative result space of the j th termination. Then:

$$A_j = E_{1j} \times E_{2j} \times \ldots \times E_{mj} ,$$

where E_{kj} is the value space of the parameter datatype of the k th parameter in the ***termination-parameter-list*** of the j th termination. The normal result space then becomes the alternative result space associated with normal termination (A_0), modeled as having ***termination-identifier*** **"*normal"**. Consider the ***termination-reference***s, and **"*normal"**, to represent values of an unspecified state datatype **ST**. Then the result space of the procedure datatype is:

$$S_T \times (A_0 \mid A_1 \mid A_2 \mid \ldots \mid A_N) ,$$

where A_0 is the normal result space and A_k is the alternative result space of the k th termination; and every value of the procedure datatype is a function of the form:

$$p : I_1 \times I_2 \times \ldots \times I_n \rightarrow S_T \times (A_0 \mid A_1 \mid A_2 \mid \ldots \mid A_N) .$$

Any of the input space, the normal result space and the alternative result space corresponding to a given ***termination-identifier*** may be empty. An empty space can be modeled mathematically by substituting for the empty space the value space of the datatype **void** (see 8.1.12).

The value space of a procedure datatype conceptually comprises all operations which conform to the above model, i.e. those which operate on a collection of values whose datatypes correspond to the input parameter datatypes and yield a collection of values whose datatypes correspond to the parameter datatypes of the normal result space or the appropriate alternative result space. The term corresponding in this regard means that to each parameter datatype in the respective product space the “collection of values” shall associate exactly one value of that datatype. When the input space is empty, the value space of the procedure datatype comprises all niladic operations yielding values in the result space.

When the result space is empty, the mathematical value space contains only one value, but the value space of the computational procedure datatype may contain many distinct values which differ in their effects on the “real world”, i.e. physical operations outside of the information space.

Value-syntax:

```
procedure-declaration   =  "procedure",  procedure-identifier,  "(",
                               [  parameter-list  ],  ")",
                               [  "returns",  "(",  return-parameter,  ")"  ],
                               [  "raises",  "(",  termination-list,  ")"  ]  ;
procedure-identifier    =  identifier  ;
```

A ***procedure-declaration*** declares the ***procedure-identifier*** to refer to a (specific) value of the **procedure** datatype whose ***type-specifier*** is identical to the ***procedure-declaration*** after deletion of the ***procedure-identifier***. The means of association of the ***procedure-identifier*** with a particular value of the procedure datatype is outside the scope of this International Standard.

Properties: unordered, exact, non-numeric.

Subtypes: For two procedure datatypes **P** and **Q**:

— **P** is said to be formally compatible with **Q** if their parameter-lists are of the same length, the direction of each parameter in the ***parameter-list*** of **P** is the same as the corresponding parameter in the ***parameter-list*** of **Q**, both have a ***return-parameter*** or neither does, and the ***termination-list***s of **P** and **Q**, if present, contain the same ***termination-reference***s.

— If **P** is formally compatible with **Q**, and for every result parameter of **Q**, the parameter datatype of the corresponding parameter of **P** is a (not necessarily proper) subtype of the parameter datatype of the parameter of **Q**, then **P** is said to be a result-subtype of Q. If the return parameter datatype and all of the parameter datatypes in the parameter-list of P and Q are identical (none are proper subtypes), then each is a result-subtype of the other.

— If **P** is formally compatible with **Q**, and for every input parameter of **Q** , the parameter datatype of the corresponding parameter of **P** is a (not necessarily proper) subtype of the parameter datatype of the parameter of **Q**, then **Q** is said to be an input-subtype of **P**. If all of the input parameter datatypes in the parameter-lists of **P** and **Q** are identical (none are proper subtypes), then each is an input-subtype of the other.

Every subtype of a procedure datatype shall be both an input-subtype of that procedure datatype and a result-subtype of that procedure datatype.

Operations: **Equal**, **Invoke**.

The definitions of **Invoke** and **Equals** below are templates for the definition of specific Invoke and Equals operators for each individual procedure datatype. Each procedure datatype has its own Invoke operator whose first parameter is a value of the procedure datatype, and whose remaining input parameters, if any, have the datatypes in the input space of that procedure datatype, and whose result-list has the datatypes of the result space of the procedure datatype.

Invoke(*x*: **procedure**(*parameter-list*), v_1: I_1, ..., v_n: I_n): **record** (r_1: R_1, ..., r_m: R_m)) is that value in the result space which is produced by the procedure *x* operating on the value of the input space which corresponds to values (v_1, ..., v_n).

Equal(*x*, *y*: **procedure**(*parameter-list*)): **boolean** is:

true if for each collection of values (v_1: I_1, ..., v_n: I_n), corresponding to a value in the input space of *x* and *y*, either:

neither *x* nor *y* is defined on (v_1, ..., v_n), or

Invoke(*x*, v_1, ..., v_n) = **Invoke**(*y*, v_1, ..., v_n);

and **false** otherwise.

NOTE 1 The definition of **Invoke** above is simplistic and ignores the concept of alternative terminations, the implications of procedure and pointer datatypes appearing in the **parameter-list**, etc. The true definition of **Invoke** is beyond the scope of this International Standard and forms a principal part of ISO/IEC 13886.

NOTE 2 Considered as a function, a given value of a procedure datatype may not be defined on the entire input space, that is, it may not yield a value for every possible input. In describing a specific value of the procedure datatype it is necessary to specify limitations on the input domain on which the procedure value is defined ("procedure value" means conceptual functionality, and not a specific body). In the general case, these limitations are on combinations of values which go beyond specifying proper subtypes of the individual parameter datatypes. Such limitations are therefore not considered to affect the admissibility of a given procedure as a value of the procedure datatype.

NOTE 3 The subtyping of procedure datatypes may be counterintuitive. Assume the declarations:

```
type P = procedure (in a: integer range (0..100), out b: typeX);
type Q = procedure (in a: integer range (0..100), out b: typeY);
type R = procedure (in a: integer, out b: typeX);
```

If **typeX** is a subtype of **typeY** then **P** is a subtype of **Q**, as one might expect. But integer range **(0..100)** is a subtype of `integer`, which makes **R** a subtype of **P**, and not the reverse! In general, the collection of procedures which can accept an arbitrary input from the larger input datatype (integer) is a subset of the collection of procedures which can accept an input from the more restricted input datatype (`integer range (0..100)`). If a procedure is required to be of type **P**, then it is presumed to be applicable to values in `integer range (0..100)`. If a procedure of type `R` is actually used, it can indeed be safely applied to any value in `integer range (0..100)`, because integer `range (0..100)` is a subtype of the domain of the procedures in **R**. But the converse is not true. If a procedure is required to be of type **R**, then it is presumed to be applicable to an arbitrary integer value, for example, **-1**, and therefore a procedure of type **P**, which is not necessarily defined at **-1**, cannot be used.

NOTE 4 In describing individual values of a procedure datatype, it is common in programming languages to specify ***parameter-name***s, in addition to parameter datatypes, for the parameters. These identifiers provide a means of distinguishing the functionality of the individual parameter values. But while this functionality is important in distinguishing one value of a procedure datatype from another, it has no meaning at all for the procedure datatype itself. For example, `Subtract(in a:real, in b:real, out diff: real)` and `Multiply(in a:real, in b:real, out prod: real)` are both values of the procedure datatype `procedure(in real, in real, out real)`, but the functionality of the parameters a and b in the two procedure values is unrelated.

NOTE 5 In describing procedures in programming languages, it is common to distinguish parameters as input, output, and input/output, to import information from common interchange areas, and to distinguish returning a single result value from returning values through the parameters and/or the interchange areas. These distinctions are supported by the syntax, but conceptually, a procedure operates on an set of input values to produce a set of output values. The syntactic distinctions relate to the methods of moving values between program elements, which are out-side the scope of this International Standard. This syntax is used in other international standards which define such mechanisms. It is used here to facilitate the mapping to programming language constructs. ISO/IEC 13886 explains the model of procedures.

NOTE 6 As may be apparent from the definition of Invoke above, there is a natural isomorphism between the normal result space of a procedure datatype and the value space of some record datatype (see 8.4.1). Similarly, there is an isomorphism between the general form of the result space and the value space of a choice datatype (see 8.3.1) in which the tag datatype is the unspecified state datatype and each alternative, including "*normal", has the form:

```
termination-name: alternative-result-space (record-type)
```


8.4 Aggregate Datatypes

An aggregate datatype is a generated datatype each of whose values is, in principle, made up of values of the 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.

Thus, many of the properties of aggregate datatypes are those of the generator, independent of the datatypes of the components. Unlike other generated datatypes, it is characteristic of aggregate datatypes that the component values of an aggregate value are accessible through characterizing operations.

This clause describes commonly encountered aggregate datatype generators, attaching to them only the semantics which derive from the construction procedure.

Syntax:

```
aggregate-type          =  record-type  |
                           class-type |
                           set-type  |
                           sequence-type  |
                           bag-type  |
                           array-type  |
                           table-type  ;
```

The definition template for an aggregate datatype is that used for all datatype generators (see 8.3), with an addition of the Properties paragraph to describe which of the aggregate properties described in clause 6.8 are possessed by that generator.

NOTE 1 In general, an aggregate-value contains more than one component value. This does not, however, preclude degenerate cases where the “aggregate” value has only one component, or even none at all.

NOTE 2 Many characterizing operations on aggregate datatypes are “constructors”, which construct a value of the aggregate datatype from a collection of values of the component datatypes, or “selectors”, which select a value of a component datatype from a value of the aggregate datatype. Since composition is inherent in the concept of aggregate, the existence of construction and selection operations is not in itself characterizing. However, the nature of such operations, together with other operations on the aggregate as a whole, is characterizing.

NOTE 3 In principle, from each aggregate it is possible to extract a single component, using selection operations of some form. But some languages may specify that particular (logical) aggregates must be treated as atomic values, and hence not provide such operations for them. For example, a character string may be regarded as an atomic value or as an aggregate of Character components. This international standard models characterstring (10.1.5) as an aggregate, in order to support languages whose fundamental datatype is (single) Character. But Basic, for example, sees the characterstring as the primitive type, and defines operations on it which yield other characterstring values, wherein 1-character strings are not even a special case. This difference in viewpoint does not prevent a meaningful mapping between the characterstring datatype and Basic strings.

NOTE 4 Some characterizations of aggregate datatypes are essentially implementations, whereas others convey essential semantics of the datatype. For example, an object which is conceptually a tree may be defined by either:

```
type tree = record
(
      label: character_string ({ iso standard 8859 1 }),
      branches: set of (tree)
),
```

or:

```
type tree = record
(
      label: character_string ({ iso standard 8859 1 }),
      son: pointer to (tree),
      sibling: pointer to (tree)
),
```

The first is a proper conceptual definition, while the second is clearly the definition of a particular implementation of a tree. Which of these datatype definitions is appropriate to a given usage, however, depends on the purpose to which this International Standard is being employed in that usage.

NOTE 5 There is no “generic” aggregate datatype. There is no “generic” construction algorithm, and the “generic” form of aggregate has no characterizing operations on the aggregate values. Every aggregate is, in a purely mathematical sense, at least a `bag` (see 8.4.3). And thus the ability to “select one” from any aggregate value is a mathematical requirement given by the axiom of choice. The ability to perform any particular operation on each element of an aggregate is sometimes cited as characterizing. But in this International Standard, this capability is modeled as a composition of more primitive functions, viz.:

```
Applytoall(A: aggregate-type, P: procedure-type) is:
    if not IsEmpty(A) begin
      e := Select(A);
      Invoke (P, e);
      Applytoall (Delete(A, e), P);
    end;
```

and the particular **Select** operations available, as well as the need for **IsEmpty** and **Delete**, are characterizing.

8.4.1 Record

Description: **record** generates a datatype, called a record datatype, whose values are heterogeneous aggregations of values of component datatypes, each aggregation having one value for each component datatype, keyed by a fixed ***field-identifier***.

Syntax:

```
record-type             =  "record",  { provision-statement }, (* see 8.6 *)
                                  "("  field-list  ")"  ;
field-list              =  field  {  ","  field  }  ;
field                   =  field-identifier  ":"  field-type  ;
field-identifier        =  identifier  ;
field-type              =  type-specifier  ;
```

Components: A list of ***field***s, each of which associates a ***field-identifier*** with a single field datatype, designated by the ***field-type***, which may be any datatype. All ***field-identifier***s of fields in the ***field-list*** shall be distinct.

Values: all collections of named values, one per field in the ***field-list***, such that the datatype of each value is the field datatype of the field to which it corresponds.

Value-syntax:

```
record-value            =  field-value-list  |
                           value-list  ;
field-value-list        =  "(",  field-value,  { ",",  field-value  },  ")"  ;
field-value             =  field-identifier, ":",  independent-value  ;
value-list              =  "(",  independent-value,
                               {  ",",  independent-value  },  ")"  ;
```

A ***record-value*** denotes a value of a **record** datatype. When the ***record-value*** is a ***field-value-list***, each ***field-identifier*** in the ***field-list*** of the **record** datatype to which the ***record-value*** belongs shall occur exactly once in the ***field-value-list***, each ***field-identifier*** in the ***record-value*** shall be one of the ***field-identifier***s in the ***field-list*** of the record-type, and the corresponding independent-value shall designate a value of the corresponding field datatype. When the ***record-value*** is a ***value-list***, the number of ***independent-value***s in the ***value-list*** shall be equal to the number of ***field***s in the ***field-list*** of the **record** datatype to which the value belongs, each ***independent-value*** shall be associated with the field in the corresponding position, and each ***independent-value*** shall designate a value of the field datatype of the associated field.

Properties: non-numeric, unordered, exact if and only if all component datatypes are exact.

Aggregate properties: heterogeneous, fixed size, no ordering, no uniqueness, access is keyed by field-identifier, one dimensional.

Subtypes: any record datatype with exactly the same field-identifiers as the base datatype, such that the field datatype of each field of the subtype is the same as, or is a subtype of, the corresponding field datatype of the base datatype.

Operations: **Equal**, **FieldSelect**, **FieldReplace**.

Equal(*x*, *y*: **record** (*field-list*)): **boolean** is **true** if for every field-identifier f of the record datatype,

field-type.Equal(FieldSelect.f(*x*), **FieldSelect.f**(*y*)), else **false**

(where **field-type.Equal** is the equality relationship on the field datatype corresponding to f).

There is one **FieldSelect** and one **FieldReplace** operation for each field in the record datatype, of the forms:

FieldSelect.field-identifier(*x***: record (***field-list***)): field-type** is the value of the field of record x whose **field-identifier** is **field-identifier**.

FieldReplace.field-identifier(*x***: record (***field-list***), *y*: field-type): record (***field-list***)** is that value z **: record(***field-list***)** such that **FieldSelect.field-identifier(*z*) =** y , and for all other fields f in **record(***field-list***), FieldSelect.f(*x*) = FieldSelect.f(*z*)**

i.e. **FieldReplace** yields the record value in which the value of the designated field of x has been replaced by y .

NOTE 1 The sequence of fields in a **record** datatype is not semantically significant in the definition of the ***record*** datatype generator. An implementation of a **record** datatype may define a representation convention which is an ordering of physically distinct fields, but that is a pragmatic consideration and not a part of the conceptual notion of the datatype. Indeed, the optimal representation for certain **record** values might be a bit string, and then **FieldReplace** would be an encoding operation and **FieldSelect** would be a decoding operation. Note that in a ***record-value*** which is a ***value-list***, however, the physical sequence of fields is significant: it is the convention used to associate the component values in the ***value-list*** with the ***field***s of the record value.

NOTE 2 A **record** datatype can be modeled as a heterogeneous aggregate of fixed size which is accessed by key, where the key datatype is a state datatype whose values are the field identifiers. But in a value of a **record** datatype, totality of the mapping is required: no field (keyed value) can be missing.

NOTE 3 A **record** datatype with a subset of the fields of a base record datatype (a "sub-record" or "projection" of the record datatype) is not a subtype of the base record datatype: none of the values in the sub-record value space appears in the base value-space. And there are, in general, a great many different "embeddings" which map the sub-record datatype into the base datatype, each of which supplies different values for the missing fields. Supplying **void** values for the missing fields is only possible if the datatypes of those fields are of the form

```
choice (tag-type) of (..., v: void)
```

NOTE 4 "Subtypes" of a "record" datatype which have additional fields is an object-oriented notion which goes beyond the scope of this International Standard.

8.4.2 Class

Description: **class** generates a datatype, called a class datatype, whose values are heterogeneous aggregations of values of component datatypes, each aggregation having one value for each component datatype, keyed by a fixed ***field-identifier***. Components of a class may include procedure definitions. The ***override*** type qualifier specifies that the labeled class attribute definition that follows replaces the prior class attribute definition with the same label.

Syntax:

```
class-type              =  "class",  { provision-statement }, (* see 8.6 *)
                               "(",  attribute-list,  ")"  ;
attribute-list          =  attribute,  {  ",",  attribute  }  ;
attribute               =  [ override-qualifier ], attribute-identifier,  ":",
                               attribute-type  ;
override-qualifier      =  "override" ;
attribute-identifier    =  identifier  ;
attribute-type          =  type-specifier  ;
```

Components: A list of attributes, each of which associates an ***attribute-identifier*** with a single ***attribute*** datatype, designated by the ***attribute-type***, which may be any datatype. All ***attribute-identifier***s of ***attribute***s in the ***attribute-list*** shall be distinct. The keyword ***override*** shall not appear unless the class is being defined as an explicit subtype (see 8.2.5). The ***attribute-identifier*** following the keyword ***override*** shall be the identifier for an attribute of the base datatype for the explicit subtype. The ***attribute-type*** following the keyword ***override*** shall designate a subtype of the attribute-type that was declared for that attribute of the base datatype.

Values: The value space is that of an unspecified state datatype, each of whose values denotes a single object that supports the interface represented by the class. The values of a class datatype are atomic.

Value-syntax: None. In general, values of a class datatype have no external representation.

Properties: non-numeric, unordered.

Subtypes: any class datatype whose attributes include one attribute corresponding to each attribute of the base datatype, such that:

— the attribute-identifier for the subtype attribute is identical to the attribute-identifier for the attribute of the base datatype, and

— the attribute datatype of the attribute of the subtype is the same as, or is a subtype of, the attribute datatype of the attribute of the base datatype.

Operations: **Equal**, **AttributeSelect, AttributeReplace**.

Equal(*x***,** *y***: class (***attribute-list***)): boolean** If there exists an **Equal** procedure for the class, then is **Equal(***x***,***y***)**. Otherwise if there are no procedure definitions then is **true** if for every attribute-identifier f of the class datatype, **attribute-type.Equal(AttributeSelect.f(***x***), AttributeSelect.f(***y***))**, else **false** (where **attribute-type.Equal** is the equality relationship on the attribute datatype corresponding to f). Otherwise is **indeterminate**.

There is one **AttributeSelect** and one **AttributeReplace** operation for each attribute in the class datatype that is not an attribute procedure, of the form:

AttributeSelect.attribute-identifier(*x***: class (***attribute-list***)): attribute-type** is the value of the attribute of class **x** whose **attribute-identifier** is **attribute-identifier**.

AttributeReplace.attribute-identifier(*x***: class(***attribute-list***),** *y***: attribute-type): class (***attribute-list***)** is that value *z***: class(***attribute-list***)** such that **AttributeSelect.attribute-identifier(***z***) =** y, and for all other attributes f in **class(***attribute-list***), AttributeSelect.f(***x***) = AttributeSelect.f(***z***)**, i.e. **AttributeReplace** yields the class value in which the value of the designated attribute of x has been replaced by y.

There is one **AttributeFunctionInvoke** and one **AttributeFunctionOverride** operation for each attribute in the class datatype that is an attribute procedure, of the forms:

AttributeFunctionInvoke.attribute-identifier(*x***: class (***attribute-list***)): attribute-type(***parameter-list***)** is the value of the attribute procedure of class x whose **attribute-identifier** is **attribute-identifier**.

AttributeFunctionOverride.attribute-identifier(*x***: class (attribute-list),** *y***: attribute-type): class (***attribute-list***)** is that function *z***: class(***attribute-list***)** such that **AttributeFunctionInvoke.attribute-identifier(***z***)** is y, and for all other attributes f in **class(***attribute-list***), AttributeFunctionInvoke.f(***x***) = AttributeFunctionInvoke.f(***z***)**

i.e. **AttributeFunctionOverride** yields the class datatype in which the function of the designated attribute of x has been replaced by y.

NOTE 1 Class models the object-oriented programming language concept with the same name.

NOTE 2 The characterization of class that distinguishes it from Pointer to Record, which is the typical implementation of Class, is the characterization of the allowable subtypes. A subtype of a Class datatype models the object-oriented notion of “subtype” or “subclass”. A subtype of a Class datatype can have *additional* attributes; a subtype of a Record cannot.

NOTE 3 An operation is represented by an attribute whose attribute-type is a procedure datatype. Invoking an operation associated with a value of a class datatype can be derived from the characterizing operations as: **Invoke(AttributeSelect(...))**.

8.4.3 Set

Description: **set** generates a datatype, called a set datatype, whose value-space is the set of all subsets of the value space of the element datatype, with operations appropriate to the mathematical set.

Syntax:

```
set-type              =  "set", { provision-statement },   (* see 8.6 *)
                                "of", "(",  element-type,   ")"  ;
element-type          =  type-specifier  ;
```

Components: The *element-type* shall designate an exact datatype, called the element datatype.

Values: every set of distinct values from the value space of the element datatype, including the set of no values, called the empty-set. A value of a set datatype can be modeled as a mathematical function whose domain is the value space of the element datatype and whose range is the value space of the boolean datatype (true, false), i.e., if **s** is a value of datatype set of **(E)**, then **s: E→B**, where B is the set of Boolean values **true** and **false**, and for any value **e** in the value space of **E**, **s(e) = true** means **e** "is a member of" the set-value **s**, and **s(e) = false** means **e** "is not a member of" the set-value **s**. The value-space of the set datatype then comprises all functions **s** which are distinct (different at some value e of the element datatype).

Value-syntax:

```
set-value             =  empty-value  |
                         value-list  ;
empty-value           =  "(",  ")"  ;
value-list            =  "(",  independent-value,
                             {  ",",  independent-value  },  ")"  ;
```

Each *independent-value* in the *value-list* shall designate a value of the element datatype. A *set-value* denotes a value of a **set** datatype, namely the set containing exactly the distinct values of the element datatype which appear in the *value-list*, or equivalently the function **s** which yields true at every value in the *value-list* and false at all other values in the element value space.

Properties: non-numeric, unordered, exact.

Aggregate properties: homogeneous, variable size, uniqueness, no ordering, access indirect (by value).

Subtypes:

a) any set datatype in which the element datatype of the subtype is the same as, or a subtype of, the element datatype of the base set datatype; or

b) any datatype derived from a base set datatype conforming to (a) by use of the Size subtype-generator (see 8.2.4).

Operations: **IsIn**, **Subset**, **Equal**, **Difference**, **Union**, **Intersection**, **Empty**, **Setof**, **Select**

IsIn(*x*: **element-type,** *y*: **set of** (*element-type*)): **boolean = y**(*x*), i.e. **true** if the value x is a member of the set y, else **false**;

Subset(*x*, *y*: **set of** (*element-type*)): **boolean** is **true** if for every value v of the element datatype, **Or(Not(IsIn**(*v*,*x*)**), IsIn**(*v*,*y*)**) = true**, else **false**; i.e. **true** if and only if every member of x is a member of y;

Equal(*x*, *y*: **set of** (*element-type*)): **boolean = And(Subset**(*x*,*y*)**, Subset**(*y*,*x*)**)**;

Difference(*x*, *y*: **set of** (*element-type*)): **set of** (*element-type*) is the set consisting of all values v of the element datatype such that **And(IsIn**(*v*, *x*)**, Not(IsIn**(*v*,*y*)**))**;

Union(*x***,** *y***: set of (***element-type***)): set of (***element-type***)** is the set consisting of all values v of the element datatype such that **Or(IsIn(***v***,***x***), IsIn(***v***,***y***))**;

Intersection(*x***,** *y***: set of (***element-type***)): set of (***element-type***)** is the set consisting of all values v of the element datatype such that **And(IsIn(***v***,***x***), IsIn(***v***,***y***))**;

Empty(): set of (*element-type***)** is the function s such that for all values v of the element datatype, $s(v) =$ **false**; i.e. the set which consists of no values of the element datatype;

Setof(*y***:** *element-type***): set of (***element-type***)** is the function s such that $s(y) =$ **true** and for all values $v \neq y$, $s(v) =$ **false**; i.e. the set consisting of the single value y;

Select(*x***: set of (***element-type***)):** *element-type*, where **Not(Equal(***x***, Empty()))**, is some one value from the value space of element datatype which appears in the set x.

NOTE Set is modeled as having only the (undefined) Select operation derived from the axiom of choice. In another sense, the access method for an element of a set value is "find the element (if any) with value v", which actually uses the characterizing "IsIn" operation, and the uniqueness property.

8.4.4 Bag

Description: **bag** generates a datatype, called a bag datatype, whose values are collections of instances of values from the element datatype. Multiple instances of the same value may occur in a given collection; and the ordering of the value instances is not significant.

Syntax:

```
bag-type            = "bag",  { provision-statement },  (* see 8.6 *)
                              "of", "(",  element-type,  ")"  ;
element-type        =  type-specifier  ;
```

Components: The ***element-type*** shall designate an exact datatype, called the element datatype.

Values: all finite collections of instances of values from the element datatype, including the empty collection. A value of a bag datatype can be modeled as a mathematical function whose domain is the value space of the element datatype and whose range is the non-negative integers, i.e., if **b** is a value of datatype **bag of (E)**, then **b: E→Z**, where **Z** is the set of integers, and for any value **e** in the value space of **E**, **b(e) = 0** means **e** "does not occur in" the bag-value **b**, and **b(e) = n**, where **n** is a positive integer, means **e** "occurs **n** times in" the bag-value **b**. The value-space of the bag datatype then comprises all functions **b** which are distinct.

Value-syntax:

```
bag-value           =  empty-value  |
                       value-list  ;
empty-value         =  "(",  ")"  ;
value-list          =  "(",  independent-value,
                          {  ",",  independent-value  },  ")"  ;
```

Each ***independent-value*** in the ***value-list*** shall designate a value of the element datatype. A ***bag-value*** denotes a value of a bag datatype, namely that function which at each value **e** of the element datatype yields the number of occurrences of **e** in the ***value-list***.

Properties: non-numeric, unordered, exact.

Aggregate properties: homogeneous, variable size, no uniqueness, no ordering, access indirect.

Subtypes:

a) any bag datatype in which the element datatype of the subtype is the same as, or a subtype of, the element datatype of the base bag datatype; or

b) any datatype derived from a base bag datatype conforming to (a) by use of the Size subtype-generator (see 8.2.4).

Operations: **IsEmpty**, **Equal**, **Empty**, **Serialize**, **Select**, **Delete**, **Insert**

IsEmpty(*x***: bag of (***element-type***)): boolean** is true if for all e in the element value space, $x(e) = 0$, else **false**;

Equal(*x***, *y*: bag of (***element-type***)): boolean** is true if for all e in the element value space, $x(e) = y(e)$, else **false**;

Empty(): bag of (*element-type***)** is that function x such that for all e in the element value space, $x(e) = 0$;

Serialize(*x***: bag of (***element-type***)): sequence of (***element-type***)** is:

if **IsEmpty(***x***)**, then (),

else any sequence value s such that for each e in the element value space, e occurs exactly $x(e)$ times in s;

Select(*x***: bag of (***element-type***)):** *element-type* **= Sequence.Head(Serialize(***x***))**;

Delete(*x***: bag of (***element-type***), *y*:** *element-type***): bag of (***element-type***)** is that function z in bag of (*element-type*) such that:

for all $e \neq y$, $z(e) = x(e)$, and

if $x(y) > 0$ then $z(y) = x(y) - 1$ and if $x(y) = 0$ then $z(y) = 0$;

i.e. the collection formed by deleting one instance of the value y, if any, from the collection e;

Insert(*x***: bag of (***element-type***), *y*:** *element-type***): bag of (***element-type***)** is that function z in bag of (*element-type*) such that:

for all $e \neq y$, $z(e) = x(e)$, and $z(y) = x(y) + 1$;

i.e. the collection formed by adding one instance of the value y to the collection x;

8.4.5 Sequence

Description: Sequence generates a datatype, called a sequence datatype, whose values are ordered sequences of values from the element datatype. The ordering is imposed on the values and not intrinsic in the underlying datatype; the same value may occur more than once in a given sequence.

Syntax:

```
sequence-type          =  "sequence",  { provision-statement }, (* see 8.6 *)
                                "of", "(",  element-type,  ")"  ;
element-type           =  type-specifier  ;
```

Components: The ***element-type*** shall designate any datatype, called the element datatype.

Values: all finite sequences of values from the element datatype, including the empty sequence.

Value-syntax:

```
sequence-value         =  empty-value  |
                          value-list  ;
empty-value            =  "(",  ")"  ;
```

```
value-list              =  "(",  independent-value,
                              {  ",",  independent-value  },  ")"  ;
```

Each ***independent-value*** in the ***value-list*** shall designate a value of the element datatype. A ***sequence-value*** denotes a value of a **sequence** datatype, namely the sequence containing exactly the values in the ***value-list***, in the order of their occurrence in the ***value-list***.

Properties: non-numeric, unordered, exact if and only if the element datatype is exact.

Aggregate properties: homogeneous, variable size, no uniqueness, imposed ordering, access indirect (by position).

Subtypes:

a) any sequence datatype in which the element datatype of the subtype is the same as, or a subtype of, the element datatype of the base sequence datatype; or

b) any datatype derived from a base sequence datatype conforming to (a) by use of the Size subtype-generator (see 8.2.4).

Operations: **IsEmpty**, **Head**, **Tail**, **Equal**, **Empty**, **Append**.

IsEmpty(*x***: sequence of (***element-type***)): boolean** is **true** if the sequence x contains no values, else **false**;

Head(*x***: sequence of (***element-type***)):** *element-type***,** where **Not(IsEmpty(***x***))**, is the first value in the sequence x;

Tail(*x***: sequence of (***element-type***)): sequence of (***element-type***)** is the sequence of values formed by deleting the first value, if any, from the sequence x;

Equal(*x***,** *y***: sequence of (***element-type***)): boolean** is:

if **IsEmpty(***x***)**, then **IsEmpty(***y***)**;

else if $Head(x) = Head(y)$, then **Equal(Tail(***x***), Tail(***y***))**;

else, **false**;

Empty(): sequence of (*element-type***)** is the sequence containing no values;

Append(x: sequence of (*element-type***),** *y*: *element-type***): sequence of (***element-type***)** is

the sequence formed by adding the single value y to the end of the sequence x.

NOTE 1 **sequence** differs from **bag** in that the ordering of the values is significant and therefore the operations **Head**, **Tail**, and **Append**, which depend on position, are provided instead of **Select**, **Delete** and **Insert**, which depend on value.

NOTE 2 The extended operation **Concatenate(***x***,** *y***: sequence of (***E***)): sequence of (***E***)** is:

if **IsEmpty(***y***)** then x; else **Concatenate(Append(***x***, Head(***y***)), Tail(***y***))**;

NOTE 3 The notion sequential file, meaning “a sequence of values of a given datatype, usually stored on some external medium”, is an implementation of datatype **sequence**.

8.4.6 Array

Description: **array** generates a datatype, called an array datatype, whose values are associations between the product space of one or more finite datatypes, designated the index datatypes, and the value space

of the element datatype, such that every value in the product space of the index datatypes associates to exactly one value of the element datatype.

Syntax:

```
array-type              =  "array",  { provision-statement }, (* see 8.6 *)
                               "(",  index-type-list,  ")", { provision-statement },
                               "of",
                               "(",  element-type,  ")"  ;
index-type-list         =  index-type,  {  ",",  index-type  }  ;
index-type              =  type-specifier  |
                           index-lowerbound,  "..",  index-upperbound  ;
index-lowerbound        =  value-expression  ;
index-upperbound        =  value-expression  ;
element-type            =  type-specifier  ;
```

Components: The ***element-type*** shall designate any datatype, called the element datatype. Each ***index-type*** shall designate an ordered and finite exact datatype, called an ***index*** datatype. When the ***index-type*** has the form:

```
index-lowerbound .. index-upperbound
```

the implied index datatype is:

```
integer range(index-lowerbound .. index-upperbound),
```

and *index-lowerbound* and *index-upperbound* shall have integer values, such that **index-lowerbound ≤ index-upperbound**.

The ***value-expression***s for ***index-lowerbound*** and ***index-upperbound*** may be dependent-values when the array datatype appears as a parameter-type, or in a component of a parameter-type, of a procedure datatype, or in a component of a record datatype. Neither ***index-lowerbound*** nor ***index-upperbound*** shall be dependent-values in any other case. Neither ***index-lowerbound*** nor ***index-upperbound*** shall be ***formal-parametric-value***s, except in certain cases in declarations (see 9.1).

Values: all functions from the cross-product of the value spaces of the index datatypes appearing in the ***index-type-list***, designated the index product space, into the value space of the element datatype, such that each value in the index product space associates to exactly one value of the element datatype.

Value-syntax:

```
array-value             =  value-list  ;
value-list              =  "(",  independent-value,
                           {  ",",  independent-value  },  ")"  ;
```

An ***array-value*** denotes a value of an array datatype. The number of ***independent-value***s in the ***value-list*** shall be equal to the cardinality of the index product space, and each independent-value shall designate a value of the element datatype. To define the associations, the index product space is first ordered lexically, with the last-occurring index datatype varying most rapidly, then the second-last, etc., with the first-occurring index datatype varying least rapidly. The first independent-value in the array-value associates to the first value in the product space thus ordered, the second to the second, etc. The array-value denotes that value of the array datatype which makes exactly those associations.

Properties: non-numeric, unordered, exact if and only if the element datatype is exact.

Aggregate properties: homogeneous, fixed size, no uniqueness, no ordering, access is indexed, dimensionality is equal to the number of index-types in the index-type-list.

Subtypes: any array datatype having the same index datatypes as the base datatype and an element datatype which is a subtype of the base element datatype.

Operations: **Select**, **Equal**, **Replace**.

Select(*x*: **array** (*index*$_1$, ..., *index*$_n$) **of** (*element-type*), y_1: *index*$_1$, ..., y_n: **index**$_n$): *element-type* is that value of the element datatype which **x** associates with the **value** (y_1, ..., y_n) in the index product space;

Equal(*x*, *y*: **array** (*index*$_1$, ..., *index*$_n$) **of** (*element-type*)): **boolean** is **true** if for every **value** (v_1, ..., v_n) in the index product space, **Select**(*x*, v_1, ..., v_n) = **Select**(*y*, v_1, ..., v_n), else **false**;

Replace(*x*: **array** (*index*$_1$, ..., *index*$_n$) **of** (*element-type*), y_1: *index*$_1$, ..., y_n: *index*$_n$, *z*: *element-type*): **array** (*index*$_1$, ..., *index*$_n$) **of** (*element-type*) is that value w of the array datatype such that *w*: (y_1, ..., y_n) $\rightarrow z$, and for all values p of the index product space except (y_1, ..., y_n), *w*: $p \rightarrow x(p)$; i.e. **Replace** yields the function which associates z with the **value** (y_1, ..., y_n) and is otherwise identical to x.

NOTE 1 The general array datatype is "multidimensional", where the number of dimensions and the index datatypes themselves are part of the conceptual datatype. The index space is an unordered product space, although it is necessarily ordered in each "dimension", that is, within each index datatype. This model was chosen in lieu of the "array of array" model, in which an array has a single ordered index datatype, in the belief that it facilitates the mappings to programming languages. Note that:

```
type arrayA = array (1..m, 1..n) of (integer);
```

defines arrayA to be a 2-dimensional datatype, whereas

```
type arrayB = array (1..m) of (array (1..n) of (integer));
```

defines arrayB to be a 1-dimensional (with element datatype `array (1..n) of (integer)`, rather than `integer`). This allows languages in which A[i][j] is distinguished from A[i, j] to maintain the distinction in mappings to the general-purpose datatypes. Similarly, languages which disallow the A[i][j] construct can properly state the limitation in the mapping or treat it as the same as A[i, j], as appropriate.

NOTE 2 The array of a single dimension is simply the case in which the number of index datatypes is 1 and the index product space is the value space of that datatype. The order of the index datatype then determines the association to the ***independent-value***s in a corresponding array-value.

NOTE 3 Support for index datatypes other than integer is necessary to model certain Pascal and Ada datatypes (and possibly others) with equivalent semantics.

NOTE 4 It is not required that the specific index values be preserved in any mapping of an array datatype, but rather that each index datatype be mapped 1-to-1 onto a corresponding index datatype and the corresponding indexing functions be preserved.

NOTE 5 Since the values of an array datatype are functions, the array datatype is conceptually a special case of the procedure datatype (see 8.3.3). In most programming languages, however, arrays are conceptually aggregates, not procedures, and have such constraints as to ensure that the function can be represented by a sequence of values of the element datatype, where the size of the sequence is fixed and equal to the cardinality of the index product space.

NOTE 6 In order to define an interchangeable representation of the Array as a sequence of element values, it is first necessary to define the function which maps the index product space to the ordinal datatype. There are various possible such functions. The one used in interpreting the array-value construct is as follows:

Let **A** be a value of datatype `array(index`$_1$`, ..., index`$_n$`) of (element-type)`. For each index datatype `index`$_i$, let `lowerbound`$_i$ and `upperbound`$_i$ be the lower and upper bounds on its value space. Define the operation **Map**$_i$ to map the index datatype **index**$_i$ into a range of integer by:

Map$_i$(*x*: **index**$_i$): **integer** is

Map$_i$(**lowerbound**$_i$) = **0**; and

Map$_i$(**Successor**$_i$(*x*)) = **Map**$_i$(*x*) + **1**, for all $x \neq upperbound$.

And define the constant: **size**$_i$ = **Map**$_i$(**upperbound**$_i$) - **Map**$_i$(**lowerbound**$_i$) + **1**. Then

Ord(x_1: **index**$_1$, ..., x_n: **index**$_n$): **ordinal**

is the **ordinal** value corresponding to the integer value:

$$1 + \sum_{i=1}^{n} Map_i(x_i) \bullet (\prod_{j=1}^{n} size_{j+1})$$

where the non-existent $\mathbf{size}_{n+1}$ is taken to be **1**. And the **Ord(**x_1**, ..., **x_n**)**th position in the sequence representation is occupied by **A(**x_1**, ..., **x_n**)**.

EXAMPLE The Fortran standard (ISO/IEC 1539-1:2004, *Information technology — Programming languages — Fortran — Part 1: Base language*) specifies that multidimensional arrays are stored with the left-most varying most rapidly. Thus in the following declaration:

```
CHARACTER*1  SCREEN (80, 24)
```

which declares the variable “screen” to have the general-purpose datatype:

```
array (1..80, 1..24) of character (unspecified)
```

The Fortran subscript operation:

```
S = SCREEN (COLUMN, ROW)
```

is equivalent to the characterizing operation:

Select (screen, column, row)

while

```
SCREEN(COLUMN, ROW) = S
```

is equivalent to the characterizing operation:

Replace(screen, column, row, S)

8.4.7 Table

Description: `table` generates a datatype, called a table datatype, whose values are collections of values in the product space of one or more field datatypes, such that each value in the product space represents an association among the values of its fields. Although the field datatypes may be infinite, any given value of a table datatype contains a finite number of associations.

Syntax:

```
table-type          =  "table",  { provision-statement }, (* see 8.6 *)
                           "(",  field-list,  ")"  ;
field-list          =  field,  {  ",",  field  }  ;
field               =  field-identifier,  ":",  field-type  ;
field-identifier    =  identifier  ;
field-type          =  type-specifier  ;
```

Components: A list of fields, each of which associates a ***field-identifier*** with a single field datatype, designated by the ***field-type***, which may be any datatype. All ***field-identifier***s of fields in the ***field-list*** shall be distinct.

Values: The value space of table (***field-list***) is isomorphic to the value space of `bag of (record(field-list))`, that is, all finite collections of associations represented by values from the cross-product of the value spaces of all the field datatypes in the ***field-list***.

Value-syntax:

```
table-value         =  empty-value  |
                          "(",  table-entry,  {  ",",  table-entry,  },  ")"  ;
table-entry         =  field-value-list  |
                       value-list  ;
```

```
field-value-list        =  "(",  field-value,  {  ",",  field-value  },  ")"  ;
field-value             =  field-identifier,  ":",  independent-value  ;
value-list              =  "(",  independent-value,
                              {  ",",  independent-value  },  ")"  ;
```

A ***table-value*** denotes a value of a table datatype, namely the collection comprising exactly the associations designated by the ***table-entry***s appearing in the ***table-value***. A ***table-entry*** denotes a value in the product space of the field datatypes in the ***field-list*** of the ***table-type***. When the ***table-entry*** is a ***field-value-list***, each ***field-identifier*** in the ***field-list*** of the ***table*** datatype to which the ***table-value*** belongs shall occur exactly once in the ***field-value-list***, each ***field-identifier*** in the ***table-entry*** shall be one of the ***field-identifier***s in the ***field-list*** of the ***table-type***, and the corresponding ***independent-value*** shall designate a value of the corresponding field datatype. When the ***table-entry*** is a ***value-list***, the number of ***independent-value***s in the ***value-list*** shall be equal to the number of fields in the ***field-list*** of the table datatype to which the value belongs, each ***independent-value*** shall be associated with the field in the corresponding position, and each ***independent-value*** shall designate a value of the field datatype of the associated field.

Properties: non-numeric, unordered, exact if and only if all field datatypes are exact.

Aggregate properties: heterogeneous, variable size, no uniqueness, no ordering, dimensionality is two.

Subtypes:

a) any table datatype which has exactly the same field-identifiers in the field-list, and the field datatype of each field of the subtype is the same as, or is a subtype of, the corresponding field datatype of the base datatype; or

b) any table datatype derived from a base table datatype conforming to (a) by use of the Size subtype-generator (see 8.2.4).

Operations: **MaptoBag**, **MaptoTable**, **Serialize**, **IsEmpty**, **Equal**, **Empty**, **Delete**, **Insert**, **Select**, **Fetch**.

MaptoBag(*x***: table(***field-list***)): bag of (record(***field-list***))** is the isomorphism which maps the table to a bag of records.

MaptoTable(*x***: bag of (record(***field-list***))): table(***field-list***)** is the inverse of the **MaptoBag** isomorphism.

Serialize(*x***: table(***field-list***)): sequence of (record(***field-list***)) = Bag.Serialize(MaptoBag(***x***))**;

IsEmpty(*x***: table(***field-list***)): boolean = Bag.IsEmpty(MaptoBag(***x***))**;

Equal(*x***, *y*: table(***field-list***)): boolean = Bag.Equal(MaptoBag(***x***), MaptoBag(*y*))**;

Empty(): table(*field-list***) = ()**;

Delete(*x***: table(***field-list***), *y*: record(***field-list***)): table(***field-list***) = MaptoTable(Bag.Delete(MaptoBag(***x***), *y*))**;

Insert(*x***: table(***field-list***), *y*: record(***field-list***)): table(***field-list***) = MaptoTable(Bag.Insert(MaptoBag(***x***), *y*))**;

Select(*x***: table (***field-list***), criterion: procedure(in row: record(***field-list***)): boolean): table(***field-list***) = MaptoTable(*z*)**, where z is the bag value whose elements are exactly those record values r in **MaptoBag(***x***)** for which **criterion(*r*) = true**.

Fetch(*x***: table(***field-list***)): record(***field-list***), where Not(IsEmpty(***x***)), = Sequence.Head(Serialize(***x***))**;

NOTE 1 Table would be a defined-generator (as in 10.2), but the type (generator) declaration syntax (see 9.1) does not permit the parametric element list to be a variable length list of field-specifiers.

NOTE 2 This definition of Table is aligned with the notion of Table specified by ISO 9075, *Information technology — Database languages — SQL*. In SQL, the "select procedure" may take as input rows from more than one table, but this is a generalization of the characterizing operation Select, rather than an extension to the Table datatype concept.

NOTE 3 In general, access to a Table is indirect, via Fetch or MaptoBag. Access to a Table is sometimes said to be "keyed" because the common utilization of this data structure represents "relationships" in which some field or fields are designated "keys" on which the values of all other fields are said to be "dependent", thus creating a mapping between the product space of the key value spaces and the value spaces of the other fields. (In database terminology, such a relationship is said to be of the "third normal form".) The specification of this mapping, when present, is a complex part of the SQL language standard and goes beyond the scope of this International Standard.

8.5 Defined datatypes

A defined datatype is a datatype defined by a type-declaration (see 9.1). It is denoted syntactically by a type-reference, with the following syntax:

```
type-reference          =  type-identifier,
                           [  "(",  actual-type-parameter-list,  ")"  ]  ;
type-identifier         =  identifier  ;
actual-type-parameter-list  =  actual-type-parameter,
                               {  ",",  actual-type-parameter  }  ;
actual-type-parameter   =  value-expression  |
                           type-specifier  ;
```

The ***type-identifier*** shall be the ***type-identifier*** of some ***type-declaration*** and shall refer to the datatype or datatype generator there-by defined. The ***actual-type-parameter***s, if any, shall correspond in number and in type to the ***formal-type-parameter***s of the ***type-declaration***. That is, each ***actual-type-parameter*** corresponds to the ***formal-type-parameter*** in the corresponding position in the ***formal-type-parameter-list***. If the ***formal-parameter-type*** is a ***type-specifier***, then the ***actual-type-parameter*** shall be a ***value-expression*** designating a value of the datatype specified by the ***formal-parameter-type***. If the ***formal-parameter-type*** is "type", then the ***actual-type-parameter*** shall be a ***type-specifier*** and shall have the properties required of that parametric datatype in the generator-declaration.

The ***type-declaration*** identifies the ***type-identifier*** in the ***type-reference*** with a single datatype, a family of datatypes, or a datatype generator. If the ***type-identifier*** designates a single datatype, then the ***type-reference*** refers to that datatype. If the ***type-identifier*** designates a datatype family, then the ***type-reference*** refers to that member of the family whose value space is identified by the ***type-definition*** after substitution of each ***actual-type-parameter*** value for all occurrences of the corresponding ***formal-parametric-value***. If the ***type-identifier*** designates a datatype generator, then the ***type-reference*** designates the datatype resulting from application of the datatype generator to the actual parametric datatypes, that is, the datatype whose value space is identified by the ***type-definition*** after substitution of each ***actual-type-parameter*** datatype for all occurrences of the corresponding ***formal-parametric-type***. In all cases, the defined datatype has the values, properties and characterizing operations defined, explicitly or implicitly, by the ***type-declaration***.

When a ***type-reference*** occurs in a ***type-declaration***, the requirements for its ***actual-type-parameter***s are as specified by clause 9.1. In any other occurrence of a ***type-reference***, no ***actual-type-parameter*** shall be a ***formal-parametric-value*** or a ***formal-parametric-type***.

8.6 Provisions

Provisions may be attached to a datatype or aggregate keyword.

Syntax:

```
provision-statement     =  "provision", "(", actual-parameter-list, ")"  ;
actual-parameter-list   =  actual-parameter, { ",", actual-parameter }  ;
actual-parameter        =  identifier, "=", identifier  ;
```

The following features may be included in a parameter list. The obligation parameter shall be included. The obligation parameter should be the first element of the list to improve reading clarity.

NOTE Typically, obligation, target, and scope are required as parameters.

A normative datatype includes a characterizing operation **IsConforming(NDT,DT)** that determines if a datatype **DT** conforms to the provisions of **NDT**.

8.6.1 General parameters for provisions

This subclause describes the general parameters for provisions.

EXAMPLE 1 The following provision specifies that for all aggregates (and subcomponents, recursively) their data elements are optional:

```
normative all_data_elements_optional =
      provision(obligation=permit, target=type, scope=recursiveidentifier, subset=defined),
normative R1 =
record all_data_elements_optional
(
      // ...
),
```

EXAMPLE 2 The following provisions combine Example 1 above with the additional provision that the datatype may be extended with additional data elements:

```
normative extended_data_elements_permitted =
      provision(obligation=permit, target=type, scope=recursiveidentifier, subset=undefined),
normative R2 =
record all_data_elements_optional extended_data_elements_permitted
(
      // ...
),
```

EXAMPLE 3 The following provision specifies that the datatype for data element `B` has a smallest of array size 17:

```
// SPM: smallest permitted maximum
normative SPM(limit) =
      provision(obligation=require, target=type, scope=size, value=range(limit..*)),

normative R =
record
(
      a: type_a,
      b: array (0..maxsize) SPM(17) of integer,
)
```

8.6.1.1 Obligation

Description: Describes the kind of obligation for the provision.

Syntax:

```
obligation-kind        =  "obligation",  "=",  obligation-kind-value ;
obligation-kind-value  =  "require" |
                          "recommend" |
                          "permit" |
                          "permitnot" |
                          "recommendnot" |
                          "requirenot" |
                          "unspecified" |
                          "default" ;
```

The values have the following meaning:

— **`require`**: the provision is a mandatory requirement, i.e., “shall” (the implementation is required to satisfy ...)

— **`recommend`**: the provision is a recommendation, i.e., “should” (the implementation is recommended to satisfy ,,,)

— **permit**: the provision is an optional [7] requirement, i.e., “may” (the implementation is permitted to satisfy ...)

— **permitnot**: the provision is an optional requirement in the negative, i.e., “may not” (the implementation is permitted not to satisfy ...)

— **recommendnot**: the provision is a recommendation, i.e., “should not” (the implementation is recommended not to satisfy ...)

— **requirenot**: the provision is a mandatory requirement, i.e., “shall not” (the implementation is required not to satisfy ...)

— **unspecified**: there is no further specification of the provision

— **default**: the default value

8.6.1.2 Target

Description: Describes the target of the provision, i.e., what is intended to satisfy the provision.

Syntax:

```
target-kind            =  "target",  "=",  target-kind-value ;
target-kind-value      =  "value" |
                          "valuespace" |
                          "properties" |
                          "charops" |
                          "type" |
                          "runtimetype" |
                          "access" |
                          "runtimeaccess" ;
```

The values have the following meaning:

— **value**: the provision is associated with the instantiation of a datatype [8]

— **valuespace**: the provision is associated with the value space of datatype

— **properties**: the provision is associated with the properties of datatype

— **charops**: the provision is associated with the characterizing operations of datatype

— **type**: the provision is associated with a datatype

— **runtimetype**: the provision is associated with the datatype at execution time

— **access**: the provision is associated with the access methods of a datatype

— **runtimeaccess**: the provision is associated with the access methods of a datatype at execution time

NOTE Except for **value**, **runtimetype**, and **runtimeaccess**, all others concern provisions of datatypes.

7) The “optional” feature described here concerns the requirements for accessing components, while the “optional” feature of 10.2.4 concerns the support of the **nil** sentinel value within a datatype.

8) Supplying **target=value** means that the provisions apply to the value itself, in contrast to the properties (**properties**) or characterizing operations (**charops**).

8.6.1.3 Scope

Description: Describes the scope of the provision, i.e., what is affected by the provision.

Syntax:

```
scope-kind              =  "scope",  "=",  scope-kind-value ;
scope-kind-value        =  "identifier" |
                           "allidentifier" |
                           "recursiveidentifier" |
                           "size" |
                           "allsize" |
                           "recursivesize" ;
```

The values have the following meaning:

— **identifier**: the provision is associated with a single identifier

— **allidentifier**: the provision is associated with all identifiers in an aggregate type

— **recursiveidentifier**: the provision is associated with the all identifiers in all aggregate types, recursively

— **size**: the provision is associated with a single sizing parameter

— **allsize**: the provision is associated with all sizing parameters in an aggregate type

— **recursivesize**: the provision is associated with the sizing parameters in all aggregate types, recursively

8.6.1.4 Subset

Description: Describes the subset scope of the provision, i.e., a pattern that describes the subset.

Syntax:

```
subset-kind             =  "subset",  "=",  subset-kind-value ;
subset-kind-value       =  "defined" |
                           "undefined" |
                           "*" |
                           "(", select-list, ")" |
                           value-expression ;
```

The values have the following meaning:

— **defined**: chooses those elements that are defined, e.g., for identifiers, if the identifier is defined; for values, if the value is defined

— **undefined**: chooses those elements that are undefined, e.g., neither the identifier nor the value is defined

— *****: chooses all elements

— **select-list**: a selecting expression that limits the selection

— **value-expression**: a value expression that describes a pattern for the selection

8.6.1.5 Value

Description: Describes the subset scope of the provision, i.e., a pattern that describes the subset.

Syntax:

```
value-spec              =  "value",  "=",  value-spec-value ;
value-spec-value        =  "nil" |
                           select-range |
                           "(", select-list, ")" |
                           value-expression ;
```

The values have the following meaning:

— **nil**: the value nil

— **select-range**: a range of values

— **select-list**: a selecting expression that limits the range

— **value-expression**: a value expression that specifies the value

8.6.2 Aggregate-specific features

This subclause describes features that are specific to aggregate values, datatypes, and normative datatypes.

8.6.2.1 Aggregate-component ordering

Description: Specifies that the components of record or class type are ordered, unordered, or unspecified.

Syntax:

```
aggregate-order         =  "aggregateorder",  "=",  aggregate-order-value ;
aggregate-order-value   =  "ordered" |
                           "notordered" |
                           "unspecified" |
                           "default" ;
```

The values have the following meaning:

— **ordered**: the aggregate's components are ordered

— **notordered**: the aggregate's component's ordering is indeterminate

— **unspecified**: it is not specified whether the aggregate's components are ordered or unordered

— **default**: the ordering is the default value

8.6.3 Aggregate-component-identifier uniqueness

Description: Specifies that the components of record or class type whose identifiers are unique or not (see 6.8.4).

Syntax:

```
aggregate-uniqueness    =  "aggregateuniqueness",  "=",  aggregate-uniqueness-value ;
aggregate-uniqueness-value =
                           "unique" |
                           "notunique" |
                           "unspecified" |
                           "default" ;
```

The values have the following meaning:

— **unique**: the aggregate's components' identifiers are unique

— **notunique**: the aggregate's components' identifiers may be non-unique

— **unspecified**: it is not specified whether the aggregate's components' identifiers are unique or not

— **default**: the uniqueness is the default value

8.6.4 Usage-specific features

This subclause describes features that are specific to the use of values, datatypes, and normative datatypes.

EXAMPLE The following provision specifies that a diagnostic message occurs every time **R** is instantiated.

```
normative obsolete =
      provision(obligation=require, target=type,
         scope=identifier, trigger=oninstantiation, action=diagnostic)

normative R =
record obsolete
(
      // ...
)
```

8.6.4.1 Usage triggers

Description: Specifies usage triggers for the provisions features.

Syntax:

```
usage-trigger          =  "onuse",  "=",  usage-trigger-value ;
usage-trigger-value    =  "ondeclaration" |
                          "oninstantiation" |
                          "onaccess" ;
```

The values have the following meaning:

— **ondeclaration**: the action is triggered on a declaration that uses this provision

— **oninstantiation**: the action is triggered on instantiation of a value

— **onaccess**: the action is triggered on the use of a value

8.6.4.2 Usage actions

Description: Specifies the action to take if a provision is triggered.

Syntax:

```
action-trigger         =  "action",  "=",  action-trigger-value ;
action-trigger-value   =  "diagnostic" |
                          "none" ;
```

The values have the following meaning:

— **diagnostic**: an implementation-defined diagnostic message occurs

— **none**: no action is taken

9 Declarations

This International Standard specifies an indefinite number of generated datatypes, implicitly, as recursive applications of the datatype generators to the primitive datatypes. This clause defines declaration mechanisms by which new datatypes and generators can be derived from the datatypes and generators of Clause 8, named and constrained. It also specifies a declaration mechanism for naming values and a mechanism for declaring alternative terminations of procedure datatypes (see 8.3.3).

Syntax:

```
declaration              =  type-declaration  |
                            value-declaration  |
                            procedure-declaration  |
                            termination-declaration  ;
```

NOTE This clause provides the mechanisms by which the facilities of this International Standard can be extended to meet the needs of a particular application. These mechanisms are intended to facilitate mappings by allowing for definition of datatypes and subtypes appropriate to a particular language, and to facilitate definition of application services by allowing the definition of more abstract datatypes.

9.1 Type declarations

A type-declaration defines a new type-identifier to refer to a datatype or a datatype generator. A datatype declaration may be used to accomplish any of the following:

— to rename an existing datatype or to name an existing datatype which has a complex syntax, or

— as the syntactic component of the definition of a new datatype, or

— as the syntactic component of the definition of a new datatype generator.

Syntax:

```
type-declaration         =  "type",  type-identifier,
                                [  "("  formal-type-parameter-list,  ")"  ],
                                "=",  [  "new"  ],  type-definition  |
                                normative-datatype-declaration  ;
type-identifier          =  identifier  ;
formal-type-parameter-list  =  formal-type-parameter,
                                {  ",",  formal-type-parameter  }  ;
formal-type-parameter    =  formal-parameter-name,  ":",  formal-parameter-type  ;
formal-parameter-name    =  identifier  ;
formal-parameter-type    =  type-specifier  |
                            "type"  ;
type-definition          =  type-specifier  ;
formal-parametric-value  =  formal-parameter-name  ;
formal-parametric-type   =  formal-parameter-name  ;
```

Every ***formal-parameter-name*** appearing in the ***formal-type-parameter-list*** shall appear at least once in the ***type-definition***. Each ***formal-parameter-name*** whose ***formal-parameter-type*** is a ***type-specifier*** shall appear as a ***formal-parametric-value*** and each ***formal-parameter-name*** whose ***formal-parameter-type*** is **type** shall appear as a ***formal-parametric-type***. Except for such occurrences, no ***value-expression*** appearing in the ***type-definition*** shall be a ***formal-parametric-value*** and no ***type-specifier*** appearing in the ***type-definition*** shall be a ***formal-parametric-type***.

The ***type-identifier*** declared in a ***type-declaration*** may be referenced in a subsequent use of a ***type-reference*** (see 8.5). The ***formal-type-parameter-list*** declares the number and required nature of the ***actual-type-parameters*** which must appear in a ***type-reference*** which references this ***type-identifier***. A ***type-reference*** which references this ***type-identifier*** may appear in an ***alternative-type*** of a ***choice-type*** or in the ***element-type*** of a ***pointer-type*** in the ***type-definition*** of this or any preceding ***type-declaration***. In any other case, the ***type-declaration*** for the ***type-identifier*** shall appear before the first reference to it in a ***type-reference***.

No ***type-identifier*** shall be declared more than once in a given context.

What the *type-identifier* is actually declared to refer to depends on whether the keyword **new** is present and whether the *formal-parameter-type* **type** is present.

9.1.1 Renaming declarations

A *type-declaration* which does not contain the keyword **new** declares the *type-identifier* to be a synonym for the *type-definition*. A *type-reference* referencing the *type-identifier* refers to the general-purpose datatype identified by the *type-definition*, after substitution of the actual datatype parameters for the corresponding formal datatype parameters.

9.1.2 New datatype declarations

A *type-declaration* that contains the keyword **new** and does not contain the *formal-parameter-type* **type** is said to be a datatype declaration. It defines the value-space of a new general-purpose datatype, which is distinct from any other general-purpose datatype. If the *formal-type-parameter-list* is not present, then the *type-identifier* is declared to identify a single general-purpose datatype. If the *formal-type-parameter-list* is present, then the *type-identifier* is declared to identify a family of datatypes parameterized by the *formal-type-parameters*.

The *type-definition* defines the value space of the new datatype (family) — there is a 1-to-1 correspondence between values of the new datatype and values of the datatype described by the *type-definition*. The characterizing operations, and any other property of the new datatype which cannot be deduced from the value space, shall be provided along with the *type-declaration* to complete the definition of the new datatype (family). The characterizing operations may be taken from those of the datatype (family) described by the *type-definition* directly, or defined by some algorithmic means using those operations.

NOTE The purpose of the **new** declaration is to allow both syntactic and semantic distinction between datatypes with identical value spaces. It is not required that the characterizing operations on the new datatype be different from those of the *type-definition*. A semantic distinction based on application concerns too complex to appear in the basic characterizing operations is possible. For example, acceleration and velocity may have identical computational value spaces and operations (datatype **real**) but quite different physical meanings.

9.1.3 New generator declarations

A *type-declaration* which contains the keyword **new** and at least one *formal-type-parameter* whose *formal-parameter-type* is **type** is said to be a generator declaration. A generator declaration declares the *type-identifier* to be a new datatype generator parameterized by the *formal-type-parameters*, and the associated value space construction algorithm to be that specified by the *type-definition*. The characterizing operations, and other properties of the datatypes resulting from the generator which cannot be deduced from the value space, shall be provided along with the generator declaration to complete the definition of the new datatype generator.

The *formal-type-parameters* whose *formal-parameter-type* is **type** are said to be parametric datatypes. A generator declaration shall be accompanied by a statement of the constraints on the parametric datatypes and on the values of the other *formal-type-parameters*, if any.

9.2 Value declarations

A *value-declaration* declares an identifier to refer to a specific value of a specific datatype.

Syntax:

```
value-declaration       =  "value",  value-identifier,  ":",  type-specifier,
                               "=",  independent-value  ;
value-identifier        =  identifier  ;
```

The *value-declaration* declares the identifier *value-identifier* to denote that value of the datatype designated by the *type-specifier* which is denoted by the given *independent-value* (see 7.5.1). The

independent-value shall (be interpreted to) designate a value of the designated general-purpose datatype, as specified by Clause 8 or Clause 10.

No *independent-value* appearing in a *value-declaration* shall be a *formal-parametric-value* and no *type-specifier* appearing in a *value-declaration* shall be a *formal-parametric-type*.

9.3 Termination declarations

A *termination-declaration* declares a *termination-identifier* to refer to an alternate termination common to multiple procedures or procedure datatypes (see 8.3.3) and declares the collection of procedure parameters returned by that termination.

Syntax:

```
termination-declaration     =  "termination",  termination-identifier,
                               [  "(",  termination-parameter-list,  ")"  ]  ;
termination-identifier      =  identifier  ;
termination-parameter-list  =  parameter,  {  ",",  parameter  }  ;
parameter                   =  [  parameter-name,  ":"  ],  parameter-type  ;
parameter-type              =  type-specifier  ;
parameter-name              =  identifier  ;
```

The *parameter-names* of the parameters in a *termination-parameter-list* shall be distinct. No *termination-identifier* shall be declared more than once, nor shall it be the same as any *type-identifier*.

The termination-declaration is a purely syntactic object. All semantics are derived from the use of the *termination-identifier* as a *termination-reference* in a procedure or procedure datatype (see 8.3.3).

9.4 Normative datatype declarations

A normative datatype declaration defines a new type-identifier to refer to a family of datatypes.[9)]

Syntax:

```
normative-datatype-declaration =
                            "normative",  identifier,
                                [  "(",  formal-type-parameter-list,  ")"  ],
                                "=",  type-definition  ;
```

9.5 Lexical operations

This section describes declarations that relate to construction of a program-text from other program-texts. A defined datatype is a datatype defined by a type-declaration (see 9.1). It is denoted syntactically by a type-reference, with the following syntax:

9.5.1 Import

Description: Import retrieves the contents of a type definition.

Syntax:

```
import-type             =  "import",  source-value,
                               { "including", "(", select-list, ")"  |
                           "excluding", "(", select-list, ")" }  ;
source-value            =  URI  |
                           identifier  ;
tag-type                =  type-specifier  ;
discriminant            =  value-expression  ;
select-list             =  select-item,  {  ",",  select-item  }  ;
select-item             =  identifier  ;
URI                     =  (* described by RFC 2396 *)
```

9) See 6.9.

Components: The source value identifies a resource that contains a program-text. Each declaration in that program-text is included in the current program-text as if it appeared verbatim in the current program-text. Exceptions: If the **`including`** keyword is used, then only those elements are included in the source. If the **`excluding`** keyword is used, then all other elements are included in the source.

NOTE 1 The import datatype generator is referred to in some programming languages as #include operator:

```
record
(
      import "http://headers.org/my-public-api-definition/record.txt",
)
```

NOTE 2 The import datatype generator might be used to perform basic inheritance and subclassing:

```
class
(
      import superclass,
      override method1: procedure // ...,
)
```

9.5.2 Macro

Description: Macro transforms string parameter value to declaration text. The macro capability permits the definition of textual replacements within 11404 program text, but does not declare new datatypes.

Syntax:

```
macro-definition        =  "macro", identifier, "(", parameter-list, ")" ,
                               "{", text, "}";
```

EXAMPLE A parameter is used to insert declaration text:

```
type X(extra) = record
(
      name: characterstring,
      address: characterstring,
      city: characterstring,
      eval(extra)
}

Y: X("country: characterstring, postalcode: characterstring")
```

In this example, the datatype of **`Y`** includes the three elements in the definition of **`X`** (**`name`**, **`address`**, **`city`**) and two additional elements specified as parameters (**`country`**, **`postalcode`**).

10 Defined datatypes and generators

This clause specifies the declarations for commonly occurring datatypes and generators which can be derived from the datatypes and generators defined in Clause 8 using the declaration mechanisms defined in Clause 9. They are included in this International Standard in order to standardize their designations and definitions for interchange purposes.

10.1 Defined datatypes

This clause specifies the declarations for a collection of commonly occurring datatypes which are treated as primitive datatypes by some common programming languages, but can be derived from the datatypes and generators defined in Clause 8.

The template for definition of such a datatype is:

Description: prose description of the datatype.

Declaration: a type-declaration for the datatype.

Parametric values: when the defined datatype is a family of datatypes, identification of and constraints on the parametric values of the family.

Values: formal definition of the value space.

Value-syntax: when there is a special notation for values of this datatype, the requisite syntactic productions, and identification of the values denoted thereby.

Properties: properties of the datatype which indicate its admissibility as a component datatype of certain datatype generators: numeric or non-numeric, approximate or exact, ordered or unordered, and if ordered, bounded or unbounded.

Operations: characterizing operations for the datatype.

The notation for values of a defined datatype may be of two kinds:

1. If the datatype is declared to have a specific value syntax, then that value syntax is a valid notation for values of the datatype, and has the interpretation given in this clause.

2. If the datatype is not declared to have a specific value syntax, then the syntax for explicit-values of the datatype identified by the type-definition is a valid notation for values of the defined datatype.

10.1.1 Natural number

Description: `naturalnumber` is the datatype of the cardinal or natural numbers.

Declaration:

```
type naturalnumber = integer range (0..*)
```

Parametric Values: none.

Values: the non-negative subset of the value-space of datatype Integer.

Properties: **ordered**, **exact**, **numeric**, **unbounded above**, **bounded below**.

Operations: all those of datatype Integer, except **Negate** (which is undefined everywhere).

10.1.2 Modulo

Description: `modulo` is a family of datatypes derived from Integer by replacing the operations with arithmetic operations using the modulus characteristic.

Declaration:

```
type modulo (modulus: integer) = new integer range(0..modulus-1) excluding(modulus)
```

Parametric Values: modulus is an integer value, such that **1 ≤ modulus**, designated the **modulus** of the Modulo datatype.

Values: all Integer values **v** such that **0 ≤ v** and **v < modulus**.

Properties: **ordered**, **exact**, **numeric**, **bounded**.

Operations: **Equal**, **InOrder** from Integer; **Add**, **Multiply**, **Negate**.

Add(*x***, *y*: modulo (*modulus*)): modulo(*modulus*) = Integer.Remainder(integer.Add(***x***,*y*), *modulus*)**