Information technology — ~~Programming languages, their environments and system software interfaces — Language-independent~~ General purpose datatypes ~~(LI)~~ (GPD) [Working Draft 3, dated 2003-01-13]

1 Scope

This International Standard specifies the nomenclature and shared semantics for a collection of datatypes commonly occurring in programming languages and software interfaces, referred to as the ~~Language-Independent (LI)~~ General Purpose (GP) 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 "language-independent" in the sense that the datatypes specified are classes of datatypes of which the actual datatypes used in programming languages and other entities requiring the concept *datatype* are particular instances. These datatypes are general in nature because they serve a wide variety of information processing applications.

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

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

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

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

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

2 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.~~The following standards contain provisions which, through reference in this text, constitute provisions of this~~

~~International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of current valid International Standards.~~

ISO/IEC 8601:~~1988~~2000, *Data elements and interchange formats — Information interchange — Representation of dates and times.*

ISO/IEC 8824:~~1990~~2002, *Information technology — ~~Open Systems Interconnection — Specification of~~ Abstract Syntax Notation One (ASN.1).*

ISO/IEC 8825:2002, *Information technology — ASN.1 Encoding Rules.*

ISO/IEC 10646-1:~~1993~~2000, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

ISO/IEC 11179-3:2002, *Information technology — Metadata registries (MDR) — Part 3: Metamodel.*

3 Definitions

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

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

3.1 actual parametric datatype: a datatype appearing as a parametric datatype in a use of a datatype generator, as opposed to the *formal-parametric-types* appearing in the definition of the datatype generator.

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

3.3 aggregate datatype: a generated datatype each of whose values is made up of values of the component datatypes, in the sense that operations on all component values are meaningful.

3.4 annotation: a descriptive information unit attached to a datatype, or a component of a datatype, or a procedure (value), to characterize some aspect of the representations, variables, or operations associated with values of the datatype which goes beyond the scope of this International Standard.

3.5 approximate: a property of a datatype indicating that there is not a 1-to-1 relationship between values of the conceptual datatype and the values of a valid computational model of the datatype.

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

3.7 bounded above: a property of a datatype indicating that there is a value U in the value space such that, for all values s in the value space, $s \leq U$.

3.8 bounded below: a property of a datatype indicating that there is a value L in the value space such that, for all values *s* in the value space, $L \leq s$.

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

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

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

3.12 datatype declaration:
(1) the means provided by this International Standard for the definition of a LI datatype which is not itself defined by this International Standard;
(2) an instance of use of this means.

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

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

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

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

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

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

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

3.20 ~~LI~~ GP datatype:
(1) a datatype defined by this International Standard, or
(2) a datatype defined by the means of datatype definition provided by this International Standard.

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

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

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

3.24 generator declaration:
(1) the means provided by this International Standard for the definition of a datatype generator which is not itself defined by this International Standard;
(2) an instance of use of this means.

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

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

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

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

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

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

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

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

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

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

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

3.36 representation:
(of a LI datatype): the mapping from the value space of the ~~LI~~ GP datatype to the value space of some internal datatype of a computer system, file system or communications environment;
(of a value): the image of that value in the representation of the datatype.

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

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

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

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

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~~2~~.1), or indirectly, by means of mappings between internal datatypes used by the entity and the datatypes specified in this International Standard (4~~2~~.2).

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

4.1 Direct conformance

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

i) specify which of the datatypes and datatype generators specified in 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

ii) define the value spaces of the ~~LI~~ GP datatypes used by the entity to be identical to the value-spaces specified by this International Standard ; and

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

iv) to the extent that the entity provides operations other than movement or translation of values, define operations on the ~~LI~~ GP datatypes which can be derived from, or are otherwise consistent with, the characterizing operations specified by this International Standard.

NOTES

1. This International Standard defines a syntax for the denotation of values of each datatype it defines, but, in general, requirement *(iii)* does not require conformance to that syntax. Conformance to the value-syntax for a datatype is required only in those cases in which the value appears in a *type-specifier*, that is, only where the value is part of the identification of a datatype.

2. The requirements above prohibit the use of a *type-specifier* defined in this International Standard to designate any other datatype. They make no other limitation on the definition of additional datatypes in a conforming entity, although it is recommended that either the form in Clause 8 or the form in Clause 10 be used.

3. Requirement (*iv*) does not require all characterizing operations to be supported and permits additional operations to be provided. The intention is to permit addition of semantic interpretation to the ~~LI~~ GP datatypes and generators, as long as it does not conflict with the interpretations given in this International Standard. A conflict arises only when a given characterizing operation could not be implemented or would not be meaningful, given the entity-provided operations on the datatype.

4. Examples of entities which could conform directly are language definitions or interface specifications whose datatypes, and the notation for them, are those defined herein. In addition, the verbatim support by a software tool or application package of the datatype syntax and definition facilities herein should not be precluded.

4.2 Indirect conformance

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

i) provide mappings between its internal datatypes and the ~~LI~~ GP datatypes conforming to the specifications of Clause 11 of this International Standard; and

ii) specify for which of the datatypes in Clause 8 and Clause 10 an inward mapping is provided, for which an outward mapping is provided, and for which no mapping is provided.

NOTES

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

2. Examples of entities which could conform indirectly are language definitions and implementations, information exchange specifications and tools, software engineering tools and interface specifications, and many other entities which have a concept of datatype and an existing notation for it.

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.

NOTES

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

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

integer range (min..max),

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

4.4 Program conformance

A program that **conforms** to this International Standard shall:

i) conform to the syntax rules specified in Clauses 5, 7, 8, and 9 of this International Standard;

ii) conform to the datatyping provisions of Clauses 6, 7, 8, 9, and 10 of this Internatioinal Standard.

5 Conventions Used in this International Standard

5.1 Formal syntax

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

Table 5-1 — Metanotation Marks

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

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

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

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

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

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

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

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

non-terminal-symbol = valid-sequence .

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

5.2 Text conventions

Within the text:

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

6 Fundamental Notions

6.1 Datatype

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

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

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

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

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

6.2 Value space

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

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

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

6.3 Datatype properties

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

NOTES

1. It is important to differentiate between the values, relationships and operations for a datatype and the representations of those values, relationships and operations in computer systems. This International Standard specifies the characteristics of the conceptual datatypes, but it only provides a means for specification of characteristics of representations of the datatypes.

2. Some computational properties derive from the *need for the information units to be representable* in computers. Such properties are deemed to be appropriate to the abstract computational model, as opposed to purely *representational* properties, which derive from the *nature of specific representations of the information units.*

3. It is not proper to describe the datatype model used herein as "mathematical", because a truly mathematical model has no notions of "access to information units" or "invocation of processing elements", and these notions are important to the definition of characterizing operations for datatypes and datatype generators.

6.3.2 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 ≠ b;
- there is no pair of instances (a, b) of values from the value space such that both a = b and a ≠ 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.

6.3.3 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 ≤ b or b ≤ a, or both;
- for any two values (a, b) from the value space, if a ≤ b and b ≤ a, then a = b;
- for any three values (a, b, c) from the value space, if a ≤ b and b ≤ c, then a ≤ c.

For convenience, the notation a < b is used herein to denote the simultaneous relationships: a ≤ b and a ≠ 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 ≤ 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.

6.3.4 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 ≤ 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 ≤ 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 ≤ U2 and U2 ≤ 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.

6.3.5 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.

6.3.6 Exact and approximate

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

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

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

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

NOTE — The computational model described above allows a mathematically dense datatype to be mapped to a datatype with fixed-length representations and nonetheless evince intuitively acceptable mathematical behavior. When the real value *h* described above is constant over the value space, the computational model is characterized as having "bounded absolute error" and the result is a scaled datatype (8.1.9). When *h* has the form *c* • | 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.

6.3.7 Numeric

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

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

6.4 Primitive and non-primitive datatypes

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

- **primitive** datatypes, which are defined ~~*ab initio*~~ 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.

Aggregate datatypes may be:

- conceptually **structured**, having both degisnators (i.e., access methods) and datatypes known prior to use of the aggregate datatype, or
- conceptually **semi-structured**, have either designators and datatypes known prior to use of the aggregate datatype, or
- conceptually **unstructured**, having neither designators and adatatypes known prior to use of the aggregate datatype.

NOTE — For semi-structured datatypes and unstructured datatypes, the designators (i.e., access methods) and datatypes may be discovered via "introspection".

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 dataype 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.

NOTES

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

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

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

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

3. The characterizing operations on a datatype may be:

a) niladic operations which yield values of the given datatype,

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

c) dyadic operations which map a pair of values of the given datatype into a value of the given datatype or into a value of datatype Boolean,

d) n-adic operations which map ordered n-tuples of values, each of which is of a specified datatype, which may be the given datatype or a parametric datatype, into values of the given datatype or a parametric datatype.

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

5. Equality is always a characterizing operation on datatypes with the "equality" property.

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

6.7 Datatype families

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

6.8 Aggregate datatypes

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

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

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

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

The properties specific to an aggregate are independent of the properties of the component datatypes. (The fundamental properties of arrays, for example, do not depend on the nature of the elements.) In principle, any combination of the properties

specified in this subclause defines a particular form of aggregate datatype, although most are only meaningful for homogeneous aggregates (see 6.8.1) and there are implications of some direct access methods (see 6.8.5).

6.8.1 Homogeneity

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

NOTES

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

2. The aggregate datatype is homogeneous if its components all belong to the same datatype, even if the element datatype is itself an heterogeneous aggregate datatype. Consider the datatype label_list defined by:

type label = choice (state(name, handle)) of ((name): characterstring, (handle): integer);

type label_list = sequence of (label);

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

6.8.2 Size

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

6.8.3 Uniqueness

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

6.8.4 Aggregate -imposed designator uniqueness

An aggregate-value has the **designator uniqueness** property if and only if no designator (e.g., label, index) of the element datatype occurs more than once in the aggregate-value. The aggregate datatype has the designator 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, modelled as random selection.

NOTES

1. The access methods become characterizing operations on the aggregate types. It is preferable to define the types by their intrinsic properties and to see these access properties be derivable characterizing operations.

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

6.8.7 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 , recursivity is supported by the type-declaration facility (see 9.1), and recursive datatypes can be described using type-declaration in combination with choice datatypes (8.3.1) or pointer datatypes (8.3.2). Thus recursive structure is *not* considered to be a property of aggregate datatypes per se.

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

7 Elements of the Datatype Specification Language

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

7.1 IDN character-set

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

Table 7-1 — IDN Character Set

Syntax	Characters
letter	a b c d e f g h i j k l m n o p q r s t u v w x y z
digit	0 1 2 3 4 5 6 7 8 9

special	( ) (parentheses)	. (full stop)	, (comma)	: (colon)	; (semicolon)	- (hyphen minus)
	{ } (curly brackets)	/ (solidus)	* (asterisk)	^ (circumflex)	= (equals sign)	[ ] (square brackets)
underscore (low line)	_					
apostrophe (apostrophe)	'					
quote (quotation mark)	"					
escape (exclamation mark)	!					
space						

letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" .

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

special = "(" | ")" | "." | "," | ":" | ";" | "=" | "/" | "*" | "-" | "{" | "}" | "[" | "]" .

underscore = "_" .

apostrophe = "'" .

quote = '"' .

escape = "!" .

space = " " .

non-quote-character = letter | digit | underscore | special | apostrophe | space .

bound-character = non-quote-character | quote .

added-character = *not defined by this International Standard* .

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

1) The *bound-characters*, and the *escape* character, are required in any implementation to be associated with particular members of the implementation character set.

2) The character *space* is required to be bound to the "space" member of ISO/IEC 10646-1:~~1993~~2000, but it only has meaning within character-literals and string-literals.

3) A *bound-character* is required to be associated with the member having the corresponding symbol, if any, in any implementation character-set derived from ISO/IEC 10646-1:~~1993~~2000, except that no significance is attached to the "case" of letters.

4) An *added-character* is any other member of the implementation character-set which is bound to the member having the corresponding symbol in an ISO/IEC 10646-1 character-set.

7.2 Whitespace

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

A *comment* is either of:

- A~~a~~ny sequence of characters beginning with the sequence "/*" and terminating with the first occurrence thereafter of the sequence "*/".

- Any sequence of characters beginning with the sequence "//" and terminating with the occurrence thereafter of a newline 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 = letter { pseudo-letter } .~~

~~pseudo-letter = letter | digit | underscore .~~

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 | digit | underscore .

digit-like = digit | ISO/IEC-10176-extended-digit .

special-like = underscore | ISO/IEC-10176-extended-special .

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

1) An identifier X declared by a *type-declaration* or *value-declaration* shall not be declared in any other declaration.

2) The identifier X in a component of a *type-specifier* (Y) refers to that component of Y which Y declares X to identify, if any, or whatever X refers to in the *type-specifier* which immediately contains Y, if any, or else the datatype or value which X is declared to identify by a declaration.

7.3.2 Digit-string

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

digit-string = digit-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 = "'" any-character "'" .

any-character = bound-character | added-character | escape-character .

escape-character = escape character-name escape .

character-name = identifier { " " identifier } .

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

string-literal = quote { string-character } quote .

string-character = non-quote-character | added-character | escape-character .

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

7.3.4 Keywords

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

Table 7-2 — Reserved Keywords

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

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

7.4 Annotations

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

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

annotation-label = objectidentifiercomponent-list .

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

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

NOTE — Of the several forms of *objectidentifiercomponent-value* specified in 10.1.10, the *nameform* is the most convenient for labelling annotations. Following ISO/IEC 8824:~~1990~~2002, every value of the objectidentifier datatype must have as its first component one of "iso", "ccitt", or "joint-iso-ccitt", but an implementation or use is permitted to specify an identifier which represents a sequence of component values beginning with one of the above, as:

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

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

```
[rpc: discriminant = n].
```

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

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

7.5 Values

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

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

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

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

7.5.1 Independent values

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

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

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

NOTES

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).

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.

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

7.5.2 Dependent values

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

dependent-value = primary-dependency { "." component-reference } .

primary-dependency = field-identifier | parameter-name .

component-reference = field-identifier | "*" .

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

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

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

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

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

1) If the *n*th *component-reference* is the last *component-reference* of the *dependent-value*, the *dependent-value* shall refer to the value to which the *n*th *component-reference* refers.

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

3) If the *n*th *component-reference* is not the last *component-reference*, and the datatype of the *n*th *component-reference* is a record datatype, then the *(n+1)*th *component-reference* shall be a *field-identifier* which identifies a field of that record datatype; and the *(n+1)*th *component-reference* shall refer to the value of that field of the value referred to by the *n*th *component-reference*.

4) If the *n*th *component-reference* is not the last *component-reference*, and the datatype of the *n*th *component-reference* is a pointer datatype, then the *(n+1)*th *component-reference* shall be "*"; and the *(n+1)*th *component-reference* shall refer to the value resulting from Dereference applied to the value referred to by the *n*th *component-reference*.

NOTES

1. The datatype which involves a *dependent-value* must be a component of some generated datatype, but that generated datatype may itself be a component of another generated datatype, and so on. The subject datatype may be several levels up this hierarchy.

2. The primary component, and thus the subject datatype, cannot be ambiguous, even when the *primary-dependency* identifier appears more than once in such a hierarchy, according to the scope rules specified in 7.3.1.

3. In the same wise, an identifier which may be either a *value-identifier* or a *dependent-value* can be resolved by application of the same scope rules. If the identifier X is found to have a "declaration" anywhere within the outermost *type-specifier* which contains the reference to X, then that declaration is used. If no such declaration is found, then a declaration of X in a "global" context, e.g. as a *value-identifier*, applies.

8 Datatypes

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

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

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

- explicit specification of **primitive** datatypes, which have universal well-defined abstract notions, each independent of any other datatype.
- implicit specification of **generated** datatypes, which are syntactically and in some ways semantically dependent on other datatypes used in their specification. Generated datatypes are specified implicitly by means of explicit specification of datatype generators, which themselves embody independent abstract notions.

- specification of the means of **datatype declaration**, which permits the association of additional identifiers and refinements to primitive and generated datatypes and to datatype generators.
- specification of the means of defining **subtypes** of the datatypes defined by any of the foregoing methods.

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

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

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

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

NOTES

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

2. The names used in this International Standard to identify the datatypes are derived in many cases from common programming language usage, but nevertheless do not necessarily correspond to the names of equivalent datatypes in actual languages. The same applies to the names and symbols for the operations associated with the datatypes, and to the syntax for values of the datatypes.

8.2 Primitive datatypes

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

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

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

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

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

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

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

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

parameter-name : parameter-datatype, or

$parameter\text{-}name_1$, $parameter\text{-}name_2$: parameter-datatype.

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

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

8.2.1 Boolean

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

Syntax:

boolean-type = "boolean" .

Parametric Values: none.

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

Value-syntax:

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

Properties: unordered, exact, non-numeric.

Operations: Equal, Not, And, Or.

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

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

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

x	Not(x)
true	false
false	true

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

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

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

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

8.2.2 State

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

Syntax:

state-type = "state" "(" state-value~~-list~~ ")" .

state-value = state-value-list | state-value-source .

state-value-list = state-literal { "," state-literal } .

state-literal = identifier .

value-domain-source = "import" list-source-reference .

list-source-reference = identifier | """ URI-text """ .

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

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

Value-syntax:

state-literal = identifier .

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

Properties: unordered, exact, non-numeric.

Operations: Equal.

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

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

EXAMPLE — The declaration:

type switch = new state (on, off);

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

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

8.2.3 Enumerated

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

Syntax:

enumerated-type = "enumerated" "(" enumerated-value ~~list~~ ")" .

enumerated-value = enumerated-value-list |

enumerated-value-list = enumerated-literal { "," enumerated-literal } .

enumerated-literal = identifier .

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

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

Value-syntax:

enumerated-literal = identifier .

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

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

Operations: Equal, InOrder, Successor

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

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

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

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

8.2.4 Character

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

Syntax:

character-type = "character" ["(" repertoire-list ")"] .

repertoire-list = repertoire-identifier { "," repertoire-identifier } .

repertoire-identifier = value-expression .

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

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

Value-syntax:

character-literal = "'" any-character "'" .

any-character = bound-character | added-character | escape-character .

bound-character = non-quote-character | quote .

non-quote-character = letter | digit | underscore | special | apostrophe | space .

added-character = *not defined by this International Standard* .

escape-character = escape character-name escape .

character-name = identifier { " " identifier } .

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

Properties: unordered, exact, non-numeric.

Operations: Equal.

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

NOTES

1. The Character datatypes are distinct from the State datatypes in that the values of the datatype are defined by other standards rather than by this International Standard or by the application. This distinction is semantically unimportant, but it is of great significance in any use of these standards.

2. The standardization of *repertoire-identifier* values will be necessary for any use of this International Standard and will of necessity extend to character sets which are defined by other than international standards. Such standardization is beyond the scope of this International Standard . A partial list of the international standards defining such character-sets is included, for informative purposes only, in Annex A.

3. While an order relationship is important in many applications of character datatypes, there is no standard order for any of the International Standard character sets, and many applications require the order relationship to conform to rules which are particular to the application itself or its language environment. There will also be applications in which the order is unimportant. Since no standard order of character-sets can be defined by this International Standard, character datatypes are said to be "unordered", meaning, in this case, that the order relationship is an application-defined addition to the semantics of the datatype.

4. The terms *character-set*, *member*, *symbol* and *character-name* are those of ISO/IEC 10646-1:~~1993~~2000, but there should be analogous notions in any character set referenceable by a *repertoire-identifier*.

5. The value space of a Character datatype is the character *set*, not the character *codes*, as those terms are defined by ISO/IEC 10646-1:~~1993~~2000. The encoding of a character set is a representation issue and therefore out of the scope of this International Standard. Many uses of this International Standard , however, may require the association to codes implied by the *repertoire-identifier*.

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

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

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

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

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

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

8.2.5 Ordinal

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

Syntax:

ordinal-type = "ordinal" .

Parametric Values: none.

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

Value-syntax:

ordinal-literal = number .

number = digit-string .

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

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

Operations: Equal, InOrder, Successor

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

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

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

8.2.6 Date-and-Time

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

Syntax:

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

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

radix = value-expression .

factor = value-expression .

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

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

Value-syntax:

time-literal = string-literal .

A *time-literal* denotes a date-and-time value. The characterstring value represented by the *string-literal* shall conform to ISO 8601:~~1988~~2000, *Representation of dates and times*. The *time-literal* denotes the date-and-time value specified by the characterstring as interpreted under ISO 8601:~~1988~~2000.

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 x precedes that designated by y; 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-units of the specified resolution elapsing between the time x and the time y; else, let z be the number of time-units elapsing between the time y and the time x, then Negate(z).

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

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

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

8.2.7 Integer

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

Syntax:

integer-type = "integer" .

Parametric Values: none.

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

Value-syntax:

integer-literal = signed-number .

signed-number = ["-"] number .

number = digit-string .

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

Properties: ordered, exact, numeric, unbounded.

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

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

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

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

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

NonNegative(x: integer): boolean **is**
true if x = 0 or x can be developed by one or more iterations of adding 1,
i.e. if x = Add(1, Add(1, ... Add(1, Add(1,0)) ...));
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) ≤ x.

Remainder(x, y: integer): integer, where 0 ≤ x and 0 < y, = Add(x, Negate(Multiply(y, Quotient(x,y))));

~~1.1.8~~8.2.8 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, Negate, Add, Multiply, Reciprocal, Promote.

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

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

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

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

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

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

NonNegative(k: rational): boolean **is** defined by:
For every rational value *k*, there is a non-negative integer *n*, such that 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)))

~~1.1.9~~8.2.9 Scaled

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

Syntax:

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

radix = value-expression .

factor = value-expression .

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

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

Value-syntax:

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

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

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

Properties: ordered, exact, numeric, unbounded.

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

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

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

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

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

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

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

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

EXAMPLES

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

type currency = new scaled(10, 2);

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

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

NOTES

1. The case factor = 0, i.e. scaled(*r*, 0) for any *r*, has the same value-space as Integer, and is isomorphic to Integer under all operations except Divide, which is not defined on Integer in this International Standard, but could be defined consistent with the Divide operation for scaled(*r*, 0). It is recommended that the datatype scaled(*r*, 0) not be used explicitly.

2. Any reasonable rounding algorithm is equally acceptable. What is required is that any rational value *v* which is not a value of the scaled datatype is mapped into one of the two scaled values $n \bullet r^{(-f)}$ and $(n+1) \bullet r^{(-f)}$, such that in the Rational value space, $n \bullet r^{(-f)} < v < (n+1) \bullet r^{(-f)}$.

3. The proper definition of scaled arithmetic is complicated by the fact that scaled datatypes with the same radix can be combined arbitrarily in an arithmetic expression and the arithmetic is effectively Rational *until* a final result must be produced. At this point, rounding to the proper scale for the result operand occurs. Consequently, the given definition of arithmetic, for operands with a common scale factor, should not be considered a specification for arithmetic on the scaled datatype.

4. The values in any scaled value space are taken from the value space of the Rational datatype, and for that reason Scaled may appear to be a "subtype" of both Rational and Real (see 8.2). But scaled datatypes do not "inherit" the Rational or Real Multiply and Reciprocal operations. Therefore scaled datatypes are not proper subtypes of datatype Real or Rational. The concept of Round, and special Multiply and Divide operations, characterize the scaled datatypes. Unlike Rational, Real and Complex, however, Scaled is not a mathematical group under this definition of Multiply, although the results are intuitively acceptable.

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

6. Every scaled datatype is exact, because every value in its value space can be distinguished in the computational model. (The value space can be mapped 1-to-1 onto the integers.) It is only the *operations* on scaled datatypes which are approximate.

7. *Scaled-literals* are interpreted as decimal values regardless of the *radix* of the scaled datatype to which they belong. It was not found necessary for this International Standard to provide for representation of values in other radices, particularly since representation of values in radices greater than 10 introduces additional syntactic complexity.

8.2.10 Real

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

Syntax:

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

radix = value-expression .

factor = value-expression .

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

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

Let $\Re$ 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 $| r - r | \leq | r | \bullet \varepsilon$;

– 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 • 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 ≠ 0, = *M*(x') where x' is the real multiplicative inverse of x.

NOTES

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

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

EXAMPLES

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

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

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

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

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

8.2.11 Complex

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

Syntax:

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

radix = value-expression .

factor = value-expression .

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

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

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

– 0 is in *V*;
– for each *v* in *C* such that | *v* | ≥ ε, there exists at least one v in *V* such that | *v* - v | ≤ | *v* | • ε;
– for each *v* in *C* such that | *v* | < ε, there exists at least one v in *V* such that | *v* - v | ≤ ε^2;.

Value-syntax:

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

real-part = real-literal .

imaginary-part = real-literal .

A *complex-literal* denotes a value of a complex datatype. The *real-part* and the *imaginary-part* are interpreted as real values, and the complex value denoted is: *M*(*real-part* + (*imaginary-part* • *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 • 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 • 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 (future) Parts of ISO/IEC 10967 Language-Independent Arithmetic.

8.2.12 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;

NOTES

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.

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.

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".

4. The "empty set" is not a value of datatype Void, but rather a value of the appropriate set datatype (see 8.4.2).

8.3 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.

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.

8.3.1 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-values*, except in some occurrences in declarations (see 9.1).

Values: all values *v* from the base datatype such that *lowerbound* $\leq v$, if *lowerbound* is specified, and $v \leq$ *upperbound*, if *upperbound* 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.

8.3.2 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* $\leq v$, if *lowerbound* is specified, and $v \leq$ *upperbound*, if *upperbound* is specified.

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

8.3.3 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* $\leq v$, if a lower bound is specified, and $v \leq$ *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.

8.3.4 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] ")" .

maximum-size = value-expression | "*" .

minimum-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-values*, 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.

8.3.5 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 LI 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.

NOTES

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.

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)).

8.3.6 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" "(" 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*.

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.

NOTES

1. Extended produces a subtype relationship in which the base datatype is the subtype and the extended datatype has the larger value space.

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

8.3.7 Ordered

Description: Specifies that the components of aggregate type are Ordered.

Syntax:

ordered-subtype = "ordered" base .

base = type-specifier .

Components: *Base* shall designate an aggregate datatype.

Properties: The subtype is Ordered.

8.3.8 Unordered

Description: Specifies that the components of aggregate type are not Ordered.

Syntax:

unordered-subtype = "unordered" base .

base = type-specifier .

Components: *Base* shall designate an aggregate datatype.

Properties: The subtype is not Ordered.

8.3.9 Unique

Description: Specifies that the identifiers of aggregate type shall be distinct.

Syntax:

unique-subtype = "unique" base .

base = type-specifier .

Components: *Base* shall designate an aggregate datatype.

8.3.10 Non-unique

Description: Specifies that the identifiers of aggregate type are may not be distinct.

Syntax:

unique-subtype = "nonunique" base .

base = type-specifier .

Components: *Base* shall designate an aggregate datatype.

8.3.11 Extendable

Description: Specifies that additional components may be added, at run-time, to a state, enumerated, or aggregate type.

Syntax:

extendable-subtype = "extendable" base .

base = type-specifier .

Components: *Base* shall designate a state, enumerated, record datatype.

Properties: The same as the subtype.

8.3.12 Non-extendable

Description: Specifies that no additional components shall be added, at run-time, to a state, enumerated, or aggregate type.

Syntax:

non-extendable-subtype = "nonextendable" base .

base = type-specifier .

Components: *Base* shall designate a state, enumerated, record datatype.

Properties: The same as the subtype.

8.3.13 Override

Description: Specifies that the labeled class member definition that follows replaces the prior class member definition with the same label.

Syntax:

override-qualifier = "override" .

Components: Override may be used with a class member definition.

Properties: The same as the subtype.

~~8.3.13~~8.3.14 Obsolete

Description: Specifies that the base type is intended to be removed from future datatype specifications.

Syntax:

ordered-subtype = "obsolete" base .

base = type-specifier .

NOTE: The use of this subtype is intended to cause a diagnostic message.

~~8.3.14~~8.3.15 Reserved

Description: Specifies that the base type is invalid abd may be included from future datatype specifications. And use of the instance of the subtype is an error and shall cause a diagnostic message.

Syntax:

ordered-subtype = "reserved" base .

base = type-specifier .

NOTE: The use of this subtype is intende to cause a diagnostic message, possibly different than the diagnostic message that "obsolete" causes.

8.4 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.

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.

8.4.1 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* $\leq v$, if lowerbound is specified, and $v \leq$ *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 Discrminant(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.

NOTES

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 ).
```

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 *alterntaive-list*. The set of tag values appearing explictly or implicitly in the *alternative-list* is not required to cover the value space of the tag datatype.

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.

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 implemenations explicitly incorporate the tag-value.

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 operatoins. 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 parctice, 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 10.2.4.

8.4.2 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 ≠ null, **is** the value of the element datatype associated with the value x.

NOTES

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.

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.

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.

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.

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.

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 LI datatype "pointer to (T)" — or the "value of the variable" — the value of LI datatype T. And thus those are the only objects which are needed in the datatype model. This notion is further elaborated in ISO/IEC 13886:1995, *Language-independent procedure calling*, which relates pointer-values to the "boxes" (or "cells") which are elements of the *state* of a running program.

8.4.3 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-names* of *parameters* in the *parameter-list* shall be distinct from each other and from the *parameter-name* of the *return-parameter*, if any. The *termination-references* 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-types* 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: $I_1 \times I_2 \times ... \times I_n \rightarrow R_p \times R_1 \times R_2 \times ... \times R_m$

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-types* of the *parameters* in the *termination-parameter-list*. Let A^j be the alternative result space of the *j*th termination. Then:

$A^j = E_1^j \times E_2^j \times ... \times E_{mj}^j$,

where E_k^j 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), modelled as having *termination-identifier* "*normal". Consider the *termination-references*, and "*normal", to represent values of an unspecified state datatype S_T. Then the result space of the procedure datatype is:

$S_T \times (A^0 | A^1 | A^2 | ... | A^N)$,

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: $I_1 \times I_2 \times ... \times I_n \rightarrow S_T \times (A^0 | A^1 | A^2 | ... | A^N)$.

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 modelled 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 many 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-list*s 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-lists* of *P* and *Q*, if present, contain the same *termination-references*.
- 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.

NOTES

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 13886:1996, *Language-independent procedure calling*.

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. 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.

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.

4. In describing individual values of a procedure datatype, it is common in programming languages to specify parameter-names, 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.

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 outside 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.

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*).

8.5 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.

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.

NOTES

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.

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.

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 aggregrate 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.

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.

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 modelled 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.

8.5.2 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" "(" 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-identifiers* 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-identifiers* 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-values* in the *value-list* shall be equal to the number of fields 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, Aggregate.

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.

NOTES

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 fields of the Record value.

2. A record datatype can be modelled 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.

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).

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

8.5.3 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.

Syntax:

class-type = "class" "(" field-list ")" .

member-list = member { "," member } .

member = { "override" } member-identifier ":" member-type .

member-identifier = identifier .

member-type = type-specifier .

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

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

Value-syntax:

class-value = member-value-list | value-list .

member-value-list = "(" member-value { "," member-value } ")" .

member-value = member-identifier ":" independent-value .

value-list = "(" independent-value { "," independent-value } ")" .

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

Properties: non-numeric, unordered.

Aggregate properties: heterogeneous, no ordering, no uniqueness, access is keyed by *member-identifier*, one dimensional.

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

Operations: Equal, MemberSelect, Aggregate.

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

There is one MemberSelect and one MemberReplace operation for each member in the class datatype that is not a member procedure, of the forms:

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

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

There is one MemberSelect and one MemberReplace operation for each member in the class datatype that is a member procedure, of the forms:

MemberFunctionInvoke.*member-identifier*(x: class (*member-list*)): *member-type*(*parameter-list*) **is**
the value of the member function of class x whose member-identifier is *member-identifier*.

MemberFunctionOverride.*member-identifier*(x: class (*member-list*), y: *member-type*): class (*member-list*) **is** that function z: class(*member-list*) such that MemberFunctionInvoke.*member-identifier*(z) **is** y, and for all other members *f* in class(*member-list*), MemberFunctionInvoke.*f*(x) = MemberFunctionInvoke.*f*(z)
i.e. MemberFunctionOverride yields the class datatype in which the function of the designated *member* of x has been replaced by y.

NOTES

1. "Subtypes" of a "class" datatype which have *additional* members is an object-oriented notion.

~~8.5.3~~8.5.4 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" "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 modelled 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*, 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* _ 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 modelled 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.

~~1.1.4~~8.5.5 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" "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 modelled as a mathematical function whose domain is the value space of the element datatype and whose range is the nonnegative integers, i.e., if *b* is a value of datatype bag of (*E*), then *b*: *E* Z, 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* _ 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 x;

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* _ 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;

~~8.5.5~~8.5.6 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" "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.

NOTES

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.

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));

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.

~~8.5.6~~8.5.7 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" "(" index-type-list ")" "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-expressions* 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-values*, 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-values* 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: Equal, Select, 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) → z,
and for all values *p* of the index product space except (y_1, ..., y_n), *w*: *p* → x(*p*);
i.e. Replace yields the function which associates *z* with the value (y_1, ..., y_n) and is otherwise identical to x.

NOTES

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 LI 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.

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-values in a corresponding array-value.

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

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.

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.

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 many such functions. The one used in interpreting the *array-value* construct is as follows:

Let A be a value of datatype array(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 ≠ *upperbound*$_i$.

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:

where the non-existent *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 declaration:

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

declares the variable "screen" to have the LI datatype:

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

And 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).
```

The Fortran standard (ISO/IEC 1539:1991, *Information technology — Programming languages — Fortran*), however, requires a mapping function which gives a different sequence representation from that given in Note 6.

~~8.5.7~~8.5.8 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"  "("  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-identifiers* 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-identifiers* 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-values* 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));

NOTES

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.

2. This definition of Table is aligned with the notion of Table specified by ISO 9075:1990, Structured Query Language (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.

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.

8.5.9 Import

Description: Import retrieves the contents of a type definition.

Syntax:

import-type = "import" URI-or-type-identifier

{ "including" "(" select-list ")" | "excluding" "(" select-list ")" } .

URI-or-type-identifier = URI | identifier .

tag-type = type-specifier .

discriminant = value-expression .

select-list = select-item { "," select-item } .

select-item = identifier .

Components: Each datatype in the source, as specified by the URI or type idenfier is included as if it were presented as source text of the datatype specification. If the "including" keyword is used, then only those elements in the source. If the "excluding" keyword is used, then all other elements are included in the source.

NOTES

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",
).
```

~~1.~~2. The import datatype generator might be used to perform basic inheritence and subclassing:

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

8.6 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 thereby defined. The *actual-type-parameters*, if any, shall correspond in number and in type to the *formal-type-parameters* 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-parameters* 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*.

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).

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 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 .

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 LI 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* which 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 LI datatype, which is distinct from any other LI datatype. If the *formal-type-parameter-list* is not present, then the *type-identifier* is declared to identify a single LI datatype. If the *formal-type-parameter-list* is present, then the *type-identifier* is declared to identify a family of datatypes parametrized by the *formal-type-parameters.*

The *type-definition* defines the value space of the new datatype (family) — there is a one-to-one 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 ones.

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 parametrized 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 LI 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.

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).

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.

10.1.2 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).

10.1.3 Modulo

Description: Modulo is a family of dataypes derived from Integer by replacing the operations with arithmetic operations using the modulus characteristic.

Declaration:

type modulo (*modulus*: integer) = new integer range(0..*modulus*) 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.

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

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

Negate(x: modulo (*modulus*)): modulo (*modulus*) **is** the (unique) value y in the value space of modulo(*modulus*) such that Add(x, y) = 0.

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

10.1.4 Bit

Description: Bit is the datatype representing the finite field of two symbols designated "0", the additive identity, and "1", the multiplicative identity.

Declaration:

type bit = modulo(2);

Parametric Values: none.

Values: 0, 1

Properties: ordered, exact, numeric, bounded.

Operations: (Equal, InOrder, Add, Multiply) from Modulo.

10.1.5 Bit string

Description: Bitstring is the datatype of variable-length strings of binary digits.

Declaration:

type bitstring = new sequence of (bit);

Parametric Values: none.

Values: Each value of datatype bitstring is a finite sequence of values of datatype bit. The value-space comprises all such values.

Value-syntax:

bitstring-literal = quote { bit-literal } quote .

bit-literal = "0" | "1" .

The *bitstring-literal* denotes that value in which the first value in the sequence is that denoted by the leftmost *bit-literal*, the second value in the sequence is that denoted by the next *bit-literal*, etc. If there are no *bit-literal*s in the *bitstring-literal*, then the value denoted is the sequence of length zero.

Properties: unordered, exact, non-numeric.

Operations: (Head, Tail, Append, Equal, Empty, IsEmpty) from Sequence (8.4.4).

NOTES

1. Bitstring is assumed to be a Sequence, rather than an Array, in that the values may be of different lengths.

2. The description and properties of bitstring are identical to those of sequence of (bit). Bitstring is said to be "new" in order to facilitate mappings. Entities may need to attach special properties to the bitstring datatype.

10.1.6 Character string

Description: Characterstring is a family of datatypes which represent strings of symbols from standard character-sets.

Declaration:

type characterstring (*repertoire*: objectidentifier) = new sequence of (character (*repertoire*));

Parametric Values: *repertoire* is a "repertoire-identifier" (see 8.1.4).

Values: Each value of a characterstring datatype is a finite sequence of members of the character-set identified by *repertoire*. The value-space comprises the collection of all such values.

Value syntax:

string-literal = quote { string-character } quote .

string-character = non-quote-character | added-character | escape-character .

non-quote-character = letter | digit | underscore | special | apostrophe | space .

added-character = *not defined by this International Standard* .

escape-character = escape character-name escape .

character-name = identifier { " " identifier } .

Each *string-character* in the *string-literal* denotes a single member of the character-set identified by *repertoire*, as provided in 8.1.4. The *string-literal* denotes that value of the characterstring datatype in which the first value in the sequence is that denoted by the leftmost *string-character*, the second value in the sequence is that denoted by the next *string-character*, etc. If there are no *string-characters* in the *string-literal*, then the value denoted is the sequence of length zero.

Properties: unordered, exact, non-numeric, denumerable.

Operations: (Head, Tail, Append, Equal, Empty, IsEmpty) from Sequence (8.4.4).

NOTES

1. There is no general international standard for collating sequences, although certain international character-set standards require specific collating sequences. Applications which need the order relationship on characterstring, and which share a character-set for which there is no standard collating sequence, need to create a defined datatype or a repertoire-identifier which refers to the character-set and the agreed-upon collating sequence.

2. Characterstring is defined to be a Sequence, rather than an Array, to permit values to be of different lengths.

3. The description and properties of the characterstring(r) datatype are identical to those of sequence of (character(r)). Characterstring datatypes are said to be "new" in order to facilitate mappings. Entities may need to attach special properties to character string datatypes.

4. Many languages distinguish as separate datatypes objects represented by character strings with specific syntactic requirements. For example, LISP has dynamic evaluation of "s-expressions"; Prolog has a similar construct; COBOL represents currency as a "numeric edited string"; and several languages have an "identifier" datatype whose values are treated as user-defined objects to which properties will be attached. In a multi-language environment, such objects can probably be manipulated only as datatype characterstring, except in the language in which the special properties were intended to be interpreted. Thus, such datatypes should be declared as LI datatypes "derived from characterstring", e.g.:

type identifier = new characterstring(*repertoire*) size(1..maxidsize);

or:

type editcharacter = character({iso standard 646}) selecting ('0'..'9', '.', ',', '+', '-', '$', '#', '*');

type numericedited = new sequence of (editcharacter);

In each case, the keyword "new" should be used to indicate the presence of unusual characterizing operations, formation rules and interpretations (see 9.1.2).

10.1.7 Time interval

Description: Timeinterval is a family of datatypes representing elapsed time in seconds or fractions of a second (as opposed to Date-and-time, which represents a point in time, see 8.1.6). It is a generated datatype derived from a scaled datatype by limiting the operations.

Declaration:

type timeinterval(unit: timeunit, radix: integer, factor: integer) = new scaled (radix, factor);

type timeunit = state(year, month, day, hour, minute, second);

Parametric Values: *Radix* is a positive integer value, and *factor* is an integer value.

Values: all values which are integral multiples of one $radix^{(-factor)}$ unit of the specified timeunit.

Properties: ordered, exact, numeric, unbounded.

Operations: (Equal, Add, Negate) from Scaled; ScalarMultiply.

Let scaled.Multiply() be the Multiply operation defined on scaled datatypes. Then:

ScalarMultiply(x: scaled(*r*,*f*), y: timeinterval(*u*,*r*,*f*)): timeinterval(*u*,*r*,*f*) = scaled.Multiply(x,y).

EXAMPLE — timeinterval(second, 10, 3) is the datatype of elapsed time in milliseconds.

10.1.8 Octet

Description: Octet is the datatype of 8-bit codes, as used for character-sets and private encodings.

Declaration:

type octet = new integer range (0..255);

Parametric Values: none.

Values: Each value of datatype Octet is a code, represented by a non-negative integer value in the range [0, 255].

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

Operations: (Equal, InOrder) from Integer.

NOTES

1. Octet is a common datatype in communications protocols.

2. It is common to define "characterizing operations" that convert an octet value to a bitstring value or an array of bit value, but there is no agreement on which bit of the octet is first in the bit string, or equivalently, how the array indices map to the bits.

10.1.9 Octet string

Description: Octetstring is the datatype of variable-length encodings using 8-bit codes.

Declaration:

type octetstring = sequence of (octet);

Parametric Values: none.

Values: Each value of the octetstring datatype is a finite sequence of codes represented by octet values. The value-space comprises the collection of all such values, including the empty sequence.

Properties: unordered, exact, non-numeric, denumerable.

Operations: (Head, Tail, Append, Equal, Empty, IsEmpty) from Sequence (8.4.4).

NOTE — Among other uses, an octetstring value is the representation of a characterstring value, and is used when the characterstring is to be manipulated as codes. In particular, octetstring should be preferred when the values may contain codes which are not associated with characters in the repertoire.

10.1.10 Private

Description: A Private datatype represents an application-defined value-space and operation set which are intentionally concealed from certain processing entities.

Declaration:

type private(*length*: NaturalNumber) = new array (1..*length*) of (bit);

Parametric Values: *Length* shall have a positive integer value.

Values: application-defined.

Properties: unordered, exact, non-numeric.

Operations: none.

NOTES

1. There is no denotation for a value of a Private datatype.

2. The purpose of the Private datatype is to provide a means by which:
   a) an object of a non-standard datatype, having a complex internal structure, can be passed between two parties which understand the type through a standard-conforming service without the service having to interpret the internal structure, or
   b) values of a datatype which is meaningless to all parties but one, such as "handles", can be provided to an end-user for later use by the knowledgeable service, for example, as part of a package interface.

In either case, the length and ordering of the bits must be properly maintained by all intermediaries. In the former case, the Private datatype may be encoded by the provider (or his marshalling agent) and decoded by the recipient (or his marshalling agent). In the latter case the Private datatype will be encoded and decoded only by the knowledgeable agent, and all others, including end-users, will handle it as a bit-array.

10.1.11 Object identifier

Description: Objectidentifier is the datatype of "object identifiers", i.e. values which uniquely identify objects in a (Open Systems Interconnection) communications protocol, using the formal structure defined by Abstract Syntax Notation One (ISO/IEC 8824:1990).

Declaration:

type objectidentifier = new sequence of (objectidentifiercomponent) size(1..*);

type objectidentifiercomponent = new integer range(0..*);

Parametric Values: none.

Values: The value space of datatype objectidentifiercomponent is isomorphic to the cardinal numbers (10.1.1), but the meaning of each value is determined by its position in an objectidentifier value.

The value-space of datatype objectidentifier comprises all non-empty finite sequences of objectidentifiercomponent values. The meaning of each objectidentifiercomponent value within the objectidentifier value is determined by the sequence of values preceding it, as provided by ISO/IEC 8824:1990. The sequence constituting a single value of datatype objectidentifier uniquely identifies an object.

Value syntax:

objectidentifier-value = ASN-object-identifier | collection-identifier .

ASN-object-identifier = “{“ objectidentifiercomponent-list “}” .

objectidentifiercomponent-list = objectidentifiercomponent-value { objectidentifiercomponent-value } .

objectidentifiercomponent-value = nameform | numberform | nameandnumberform .

nameform = identifier .

numberform = number .

nameandnumberform = identifier “(“ numberform “)” .

collection-identifier = registry-name registry-index .

registry-name = "ISO_10646" | "ISO_2375" | "ISO_7350" | "ISO_10036" .

registry-index = number .

An *objectidentifier-value* denotes a value of datatype objectidentifier. An *objectidentifiercomponent-value* denotes a value of datatype objectidentifiercomponent. A *value-identifier* appearing in the *numberform* shall refer to a non-negative integer value. In all cases, the value denoted by an *ASN-object-identifier* is that prescribed by ISO/IEC 8824:1990 Abstract Syntax Notation One.

A *collection-identifier* denotes a value of datatype objectidentifier which refers to a registered character-set.

The keyword "ISO_10646" refers to the collections defined in Annex A of ISO/IEC 10646-1:1993 and the collection designated is that collection whose "collection-number" is the value of *registry-index*. The form of the object identifier value is:

{ iso(1) standard(0) 10646 part1(1) *registry-index* }.

A *collection-identifier* beginning with the keyword "ISO_2375" designates the collection registered under the provisions of ISO 2375:1985 whose registration-number is the value of *registry-index*. The form of the object identifier value is:

{ iso(1) standard(0) 2375 *registry-index* }.

A *collection-identifier* beginning with the keyword "ISO_7350" designates the collection registered under the provisions of ISO 7350:1991 whose registration-number is the value of *registry-index*. The form of the object identifier value is:

{ iso(1) standard(0) 7350 *registry-index* }.

A *collection-identifier* beginning with the keyword "ISO_10036" designates the collection registered under the provisions of ISO 10036:1991 whose registration-number is the value of *registry-index*. The form of the object identifier value is:

{ iso(1) standard(0) 10036 *registry-index* }.

Properties: unordered, exact, non-numeric.

Operations on objectidentifiercomponent: Equal from Integer;

Operations on objectidentifier: Append from Sequence; Equal, Length, Detach, Last.

Length(x: objectidentifier): integer is the number of objectidentifiercomponent values in the sequence x;

Detach(x: objectidentifier): objectidentifier, where Length(x) > 1, is the objectidentifier formed by removing the last objectidentifiercomponent value from the sequence x;

Last(x: objectidentifier): objectidentifiercomponent is the objectidentifiercomponent value which is the last element of the sequence x;

Equal(x,y: objectidentifier): boolean =
if Not(Length(x) = Length(y)) then false,
else if Not(objectidentifiercomponent.Equal(Last(x), Last(y))) then false,
else if Length(x) = 1 then true,
else Equal(Detach(x), Detach(y));

NOTES

1. IsEmpty, Head and Tail from Sequence are not meaningful on datatype objectidentifier. Therefore, Length and Equal are defined here, although they could be derived by using the Sequence operations.

2. ObjectIdentifier is treated as a primitive type by many applications, but the mechanism of definition of its value space, and the use of that mechanism by some applications, such as Directory Services for OSI, requires the values to be lists of an accessible element datatype (objectidentifiercomponent).

10.2 Defined generators

This clause specifies the declarations for a collection of commonly occurring datatype generators which can be derived from the datatypes and generators appearing in Clause 8.

The template for definition of such a datatype generator is:

Description:	prose description of the datatype generator.
Declaration:	a type-declaration for the datatype generator.
Components:	number of, and constraints on, the parametric datatypes and parametric values used by the generation procedure.
Values:	formal definition of the 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. When the generator generates an aggregate datatype, the aggregate properties described in clause 6.8 are also specified.
Operations:	characterizing operations for the resulting datatype which associate to the datatype generator. The definitions of operations have the form described in 8.1.

10.2.1 Stack

Description: Stack is a generator derived from Sequence by replacing the characterizing operation Append with the characterizing operation Push. That is, the insertion operation (Push) puts the values on the beginning of the sequence rather than the end of the sequence (Append).

Declaration:

type stack (*element*: type) = new sequence of (*element*);

Components: *element* may be any datatype.

Values: all finite sequences of values from the *element* datatype.

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).

Operations: (IsEmpty, Equal, Empty) from Sequence; Top, Pop, Push.

Top(x: stack (*element*)): *element* = sequence.Head(x).

Pop(x: stack (*element*)): stack (*element*) = sequence.Tail(x).

Push(x: stack (*element*), y: *element*): stack (*element*) is the sequence formed by adding the single value y to the beginning of the sequence x.

10.2.2 Tree

Description: Tree is a generator which generates recursive list structures.

Declaration:

```
type tree (leaf: type) = new sequence of (choice( state(atom, list) ) of (
                                                    (atom): leaf,
                                                    (list): tree(leaf)));
```

Components: *leaf* shall be any datatype.

Values: all finite recursive sequences in which every value is either a value of the *leaf* datatype, or a (sub-)tree itself. Ultimately, every "terminal" value is of the leaf datatype.

Properties: unordered, non-numeric, exact if and only if the *leaf* type is exact, denumerable.

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

Operations: (IsEmpty, Equal, Empty, Head, Tail) from Sequence; Join.

To facilitate definition of the operations, the datatype tree_member is introduced, with the declaration:

type tree_member(*leaf*: type) = choice(state(atom, list)) of ((atom): *leaf*, (list): tree(*leaf*));

tree_member(*leaf*) is then the element datatype of the sequence datatype underlying the tree datatype.

Join(x: tree(*leaf*), y: tree_member(*leaf*)): tree(*leaf*) **is** the sequence whose Head (first member) is the value y, and whose Tail is all members of the sequence x.

NOTE — Tree is an aggregate datatype which is formally an aggregate (sequence) of tree_members. Conceptually, tree is an aggregate datatype whose values are aggregates of *leaf* values. In either case, it is proper to consider Tree a homogeneous aggregate.

10.2.3 Cyclic enumerated

Description: Cyclic (enumerated) is a generator which redefines the successor operation on an enumerated datatype, so that the successor of the last value is the first value.

Declaration:

type cyclic of (base: type) = new base;

Components: *base* shall designate an enumerated datatype.

Values: all values v of the *base* datatype.

Properties: ordered, exact, non-numeric.

Operations: (Equal, InOrder) from the *base* datatype; Successor.

Let *base*.Successor denote the Successor operation defined on the *base* datatype; then:

Successor(x: cyclic of *(base)*): cyclic of *(base)* **is**
if for all y in the value space of *base*, Or(Not(InOrder(x,y)), Equal(x,y)), then that value z in the value space of *base* such that for all y in the value space of *base*, Or(Not(InOrder(y,z)), Equal(y,z));else *base*.Successor(x).

10.2.4 Optional

Description: Optional is a generator which effectively adds the "nil" value to the value space of a base datatype.

Declaration:

type optional(base: type) = new choice (boolean) of ((true): base, (false): void);

Components: *base* shall designate any datatype.

Values: all values *v* of the *base* datatype plus the "nil value" of void. This type is isomorphic to the set of pairs:
{ (true, v) | v in *base* } union { (false, nil) },
which is the modelled value space of the choice-type.

Properties: all properties of the *base* datatype, except for the value "nil".

Operations: IsPresent (= Discriminant from Choice); all operations on the *base* datatype, modified as indicated below.

IsPresent(x: optional(*base*)): boolean = Discriminant(x);

All unary operations of the form: Unary-op(x: *base*): result-type are defined on optional(*base*) by:

Unary-op(x: optional(*base*)): result-type **is** if IsPresent(x) then Unary-op(Cast.*base*(x)), else undefined.

All binary operations of the form: Binary-op(x, y: *base*): result-type are defined on optional(*base*) by:

Binary-op(x, y: optional(*base*)): result-type **is**:
if And(IsPresent(x), IsPresent(y)), then Binary-op(Cast.*base*(x), Cast.*base*(y)),
else undefined.

Other operations are defined similarly.

NOTE — An optional datatype is the proper type of an object, such as a parameter to a procedure or a field of a record, which in some instances may have no value.

EXAMPLES

1. A record-type containing optional (sometimes not present or "undefined") values can be declared:

```
record (
    required_name: characterstring,
    optional_value: optional(integer));
```

2. A procedure parameter which may only sometimes be provided can be declared:

```
procedure search(in t: T,
    in tableT: sequence of (T),
    in index: optional(procedure(in i: integer, in j: integer): integer)
): boolean;
```

The parameter index, which is an indexing function for tableT, need not always be provided. That is, it may have value "nil".

11 Mappings

This clause defines the general form of and requirements for mappings between the datatypes of a programming or specification language and the LI datatypes.

The internal datatypes of a language are considered to include the information type and structure notions which can be expressed in that language, particularly those which describe the nature of objects manipulated by the language primitives. Like the LI datatypes, the datatype notions of a language can be divided into primitive datatypes and datatype generators. The primitive datatypes of a language are those object types which are considered in the language semantics to be primitive, that is, not to be generated from other internal datatypes. The datatype generators of a language are those language constructs which can be used to produce new datatypes, objects with new datatypes, more elaborate information structures or static inter-object relationships.

This International Standard defines a neutral language for the formal identification of precise semantic datatype notions – the LI datatypes. The notion of a **mapping** between the internal datatypes of a language and the LI datatypes is the conceptual identification of semantically equivalent notions in the two languages. There are then two kinds of mappings between the internal datatypes of a language and the LI datatypes:

- a mapping from the internal datatypes of the language into the LI datatypes, referred to as an **outward mapping**, and
- a mapping from the LI datatypes to the internal datatypes of the language, referred to as an **inward mapping**.

This International Standard does not specify the precise form of a mapping, because many details of the form of a mapping are language-dependent. This clause specifies requirements for the information content of inward and outward mappings and conditions for the acceptability of such mappings.

NOTES

1. Mapping, in this sense, does not apply to program modules or service specifications directly, because they manipulate specific object-types, which have specific datatypes expressed in a specific language or languages. The datatypes of a program module or service specification can therefore be described in the LI datatypes language directly, or inferred from the inward and outward mappings of the language in which the module or specification is written.

2. The companion notion of *conversion of values* from an internal representation to a neutral representation associated with LI datatypes is not a part of this International Standard, but may be a part of standards which refer to this International Standard.

11.2 Outward Mappings

An outward mapping for a primitive internal datatype shall identify the syntactic and semantic constructs and relationships in the language which together uniquely represent that internal datatype and associate the internal datatype with a corresponding LI datatype expressed in the formal language defined by Clause 7 through Clause 10.

An outward mapping for an internal datatype generator shall identify the syntactic and semantic constructs and relationships in the language which together uniquely represent that internal datatype generator and associate the internal datatype generator with a corresponding LI datatype generator expressed in the formal language defined in this International Standard.

The collection of outward mappings for the datatypes and datatype generators of a language shall be said to constitute the *outward mapping of the language* and shall have the following properties:

i) to each primitive or generated internal datatype, the mapping shall associate a single corresponding LI datatype; and

ii) for each internal datatype, the mapping shall specify the relationship between each allowed value of the internal datatype and the equivalent value of the corresponding LI datatype; and

iii) for each value of each LI datatype appearing in the mapping, the mapping shall specify whether any value of any internal datatype is mapped onto it, and if so, which values of the internal datatypes are mapped onto it.

NOTES

1. There is no requirement for a primitive internal datatype to be mapped to a primitive LI datatype. This International Standard provides a variety of conceptual mechanisms for creating generated LI datatypes from primitive or previously-created datatypes, which are, inter alia, intended to facilitate mappings.

2. An internal datatype constructed by application of an internal datatype generator to a collection of internal parametric datatypes will be implicitly mapped to the LI datatype generated by application of the mapped datatype generator to the mapped parametric datatypes. In this way, property *(i)* above may be satisfied for internal generated datatypes.

3. The conceptual mapping to LI datatypes may not be either 1-to-1 or onto. A mapping must document the anomalies in the identification of internal datatypes with LI datatypes, specifically those values which are distinct in the language, but not distinct in the LI datatype, and those values of the LI datatype which are not accessible in the language.

4. Among other uses, an outward mapping may be used to identify an internal datatype with a particular LI datatype in order to require operation or representation definitions specified for LI datatypes by another standard to be properly applied to the internal datatype.

5. An outward mapping may be used to ensure that interfaces between two program units using a common programming language are properly provided by a third-party service which is ignorant of the language involved.

11.3 Inward Mappings

An inward mapping for a primitive LI datatype, or a single generated LI datatype, shall associate the LI datatype with a single internal datatype, defined by the syntactic and semantic constructs and relationships in the language which together uniquely represent that internal datatype. Such a mapping shall specify limitations on the parametric values of any LI datatype family which exclude members of that family from the mapping. Different members of a single LI datatype family may be mapped onto dissimilar internal datatypes.

An inward mapping for a LI datatype generator shall associate the LI datatype generator with an internal datatype generator, defined by the syntactic and semantic constructs and relationships in the language which together uniquely represent that internal datatype generator. Such a mapping shall specify limitations on the parametric datatypes of any LI datatype generator which exclude corresponding classes of generated datatypes from the mapping. The same LI datatype generator with different parametric datatypes may be mapped onto dissimilar internal datatype generators.

An inward mapping for a LI datatype shall associate the LI datatype with an internal datatype on which it is possible to implement all of the characterizing operations specified for that LI datatype.

The collection of inward mappings for the LI datatypes and datatype generators onto the internal datatypes and datatype generators of a language shall be said to constitute the **inward mapping of the language** and shall have the following properties:

i) for each LI datatype (primitive or generated), the mapping shall specify whether the LI datatype is supported by the language (as specified in 11.4), and if so, identify a single corresponding internal datatype; and

ii) for each LI datatype which is supported, the mapping shall specify the relationship between each allowed value of the LI datatype and the equivalent value of the corresponding internal datatype; and

iii) for each value of an internal datatype, the mapping shall specify whether that value is the image (under the mapping) of any value of any LI datatype, and if so, which values of which LI datatypes are mapped onto it.

NOTES

1. A LI generated datatype which is not specifically mapped by a primitive datatype mapping, and whose parametric datatypes are admissible under the constraints on the datatype generator mapping, will be implicitly mapped onto an internal datatype constructed by application of the mapped internal datatype generator to the mapped internal parametric datatypes.

2. When a LI datatype, primitive or generated, is mapped onto a language datatype, whether explicitly or implicitly by mapping the generators, the associated internal datatype should support the semantics of the LI datatype. The proof of this support is the ability to perform the characterizing operations on the internal datatype. It is not necessary for the language to support the characterizing operations directly (by operator or built-in function or anything the like), but it is necessary for the characterizing operations to be conceptually supported by the internal datatype. Either it should be possible to write procedures in the language which perform the characterizing operations on objects of the associated internal datatype, or the language standard should require this support in the further mappings of its internal datatypes, whether into representations or into programming languages.

3. The conceptual mapping onto internal datatypes may not be either 1-to-1 or onto. A mapping must document the anomalies in the association of internal datatypes with LI datatypes, specifically those values which are distinct in the LI datatype, but not distinct in the language, and those values of the internal datatype which are not accessible through interfaces using LI datatypes.

4. An inward mapping to a programming language may be used to ensure that an interface between two program units specified in terms of LI datatypes can be properly used by programs written in that language, with language-specific, but *not* application-specific, software tools providing conversions of information units.

11.4 Reverse Inward Mapping

An inward mapping from a LI datatype into the internal datatypes of a language defines a particular set of values of internal datatypes to be the *image* of the LI datatype in the language. The **reverse inward mapping** for a LI datatype maps those values of the internal datatypes which constitute its image to the corresponding values of that LI datatype using the correspondence which is established by the inward mapping. For the reverse inward mapping to be unambiguous, the inward mapping of each LI datatype must be 1-to-1. This is formalized as follows:

i) if *a* is a value of the LI datatype and the inward mapping maps *a* to a value *a'* of some internal datatype, then the inward mapping shall not map any value *b* of the same LI datatype into *a'*, unless *b* = *a*; and

ii) if *a* is a value of a LI datatype and the inward mapping maps *a* to a value *a'* of some internal datatype, then the reverse inward mapping maps *a'* to *a*; and

iii) if *c* is a value of a LI datatype which is excepted from the domain of the inward mapping, i.e. maps to no value of the corresponding internal datatype, then there is no value *c'* of any internal datatype such that the reverse inward mapping maps *c'* to *c*.

The **reverse inward mapping for a language** is the collection of the reverse inward mappings for the LI Datatypes.

NOTES

1. When an interface between two program units is specified in terms of LI datatypes, it is possible for the interface to be utilized by program units written in different languages and supported by a service which is ignorant of the languages involved. The inward mapping for each language is used by the programmer for that program unit to select appropriate internal datatypes and values to represent the information which is used in the interface. Information is then sent by one program unit, using the reverse inward mapping for its language to map the internal values to the intended values of the LI datatypes, and received by the other program unit, using the inward mapping to map the LI datatype values passed into suitable internal values. The actual transmission of the information may involve three software tools: one to perform the conversion between the sender form and the interchange form, automating the reverse inward mapping, one to transmit the interchange form based on LI datatypes, and one to perform the conversion between the interchange form and the receiving internal form, automating the inward mapping. None of these intermediate tools depends on the particular interface being used. Thus, it is possible to

implement an arbitrary interface using LI datatypes, in any programming language which supports those datatypes without interface-specific tools.

2. The reverse inward mapping for a language does not have useful formal properties. The same internal value can be mapped to several different values, as long as the different values belong to different LI datatypes. It is the per-datatype reverse inward mapping which is useful.

11.5 Support of Datatypes

An information processing entity is said to **support** a LI datatype if its mapping of that datatype into some internal datatype (see 11.2) preserves the properties of that datatype (see 6.3) as defined in this subclause.

NOTE — For aggregate datatypes, preservation of the "aggregate properties" defined in 6.8 is *not* required.

11.5.1 Support of equality

For a mapping to preserve the equality property, any two instances a, b of values of the internal datatype shall be considered equal if and only if the corresponding values a', b' of the LI datatype are equal.

11.5.2 Support of order

For a mapping to preserve the order property, the order relationship defined on the internal datatype shall be consistent with the order relationship defined on the LI datatype. That is, for any two instances *a*, *b* of values of the internal datatype, $a \leq b$ shall be true if and only if, for the corresponding values *a'*, *b'* of the LI datatype, $a' \leq b'$.

11.5.3 Support of bounds

For a mapping to preserve the bounds, the internal datatype shall be bounded above if and only if the LI datatype is bounded above, and the internal datatype shall be bounded below if and only if the LI datatype is bounded below.

NOTE — It follows that the values of the bounds must correspond.

11.5.4 Support of cardinality

For a mapping to preserve the cardinality of a finite datatype, the internal datatype shall have exactly the same number of values as the LI datatype. For a mapping to preserve the cardinality of an exact, denumerably infinite datatype, there shall be exactly one internal value for every value of the LI datatype and there shall be no *a priori* limitation on the values which can be represented. For a mapping to preserve the cardinality of an approximate datatype, it suffices that it preserve the approximate property, as provided in 6.3.5.

NOTES

1. There may be accidental limitations on the values of exact, denumerably infinite datatypes which can be represented, such as the total amount of storage available to a particular user, or the physical size of the machine. Such a limitation is not an intentional limitation on the datatype as implemented by a particular information processing entity, and is thus not considered to affect support.

2. An entity which *a priori* limits integer values to those which can be represented in 32 bits or characterstrings to a length of 256 characters, however, is *not* considered to support the mathematically infinite Integer and CharacterString datatypes. Rather such an entity supports describable subtypes of those datatypes (see 8.2).

11.5.5 Support for the exact or approximate property

To preserve the exact property, the mapping between values of the LI datatype and values of the internal datatype shall be 1-to-1.

For an inward mapping to preserve the approximate property, every value which is distinguishable in the LI datatype must be distinguishable in the internal datatype.

NOTE — The internal datatype may have *more values* than the LI datatype, i.e. a finer degree of approximation.

For an outward mapping to preserve the approximate property, every value which is distinguishable in the internal datatype must be distinguishable in the LI datatype.

11.5.6 Support for the numeric property

There are no requirements for support of the numeric property. Support for the numeric property is a requirement on representations of the values of the datatype, which is outside the scope of this International Standard.