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

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

# Contents

## Foreword

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

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

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

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

# Introduction

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

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

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

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

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

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

## 1 Scope

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

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

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

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

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

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

## 2 Conformance

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

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

## 2.1 Direct conformance

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

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

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

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

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

NOTES

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

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

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

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

## 2.2 Indirect conformance

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

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

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

NOTES

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

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

## 2.3 Conformance of a mapping standard

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

NOTES

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

tional Standard.

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

integer range (min..max),

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

## 3 Normative References

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

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

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

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

## 4 Definitions

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

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

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

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

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

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

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

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

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

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

**4.9 characterizing operations**:
(of a datatype): a collection of operations on, or yielding, values of the datatype, which distinguish this datatype from

other datatypes with identical value spaces;

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

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

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

**4.12 datatype declaration**:

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

(2) an instance of use of this means.

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

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

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

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

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

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

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

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

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

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

**4.23 generator declaration**:

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

(2) an instance of use of this means.

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

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

**4.26 LI datatype**:

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

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

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

**4.28 mapping**:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# 5 Conventions Used in this International Standard

## 5.1 Formal syntax

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

**Table 5-1 — Metanotation Marks**

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

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

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

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

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

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

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

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

non-terminal-symbol = valid-sequence .

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

### 5.2 Text conventions

Within the text:

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

## 6 Fundamental Notions

### 6.1 Datatype

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

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

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

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

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

## 6.2 Value space

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

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

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

## 6.3 Datatype properties

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

NOTES

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

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

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

### 6.3.1 Equality

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

- for any two instances (a, b) of values from the value space, either a *is equal to* b, denoted a = b, or a *is not equal to* b, denoted a ≠ 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.

### 6.3.2 Order

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

- for every pair of values (a, b) from the value space, either a ≤ 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.

### 6.3.3 Bound

A datatype is said to be **bounded above** if it is ordered and there is a value U in the value space such that, for all values s in the value space, s ≤ 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.

### 6.3.4 Cardinality

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

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

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

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

### 6.3.5 Exact and approximate

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

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

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

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

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

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

### 6.3.6 Numeric

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

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

## 6.4 Primitive and non-primitive datatypes

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

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

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

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

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

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

## 6.5 Datatype generator

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

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

The application of a datatype generator to a specific collection of datatypes meeting the criteria for the datatype generator forms a **generated datatype**. The generated 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 which distinguish this datatype from other datatypes having value spaces which are identical except possibly for substitution of symbols.

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

NOTES

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

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

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

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

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

## 6.7 Datatype families

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

## 6.8 Aggregate datatypes

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

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

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

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

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

### 6.8.1 Homogeneity

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

NOTES

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

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

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

type label_list = sequence of (label);

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

### 6.8.2 Size

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

### 6.8.3 Uniqueness

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

### 6.8.4 (Aggregate-imposed) ordering

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

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

### 6.8.5 Access method

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

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

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

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

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

An aggregate datatype is said to have only **indirect access methods** if there is no aggregate-imposed index mapping. Indirect access may be by position (if the aggregate datatype has **ordering**), by value of the element (if the aggregate datatype has **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.

### 6.8.6 Recursive structure

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

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

## 7 Elements of the Datatype Specification Language

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

### 7.1 IDN character-set

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

**Table 7-1 — IDN Character Set**

| Syntax | Characters | | | | | |
|---|---|---|---|---|---|---|
| letter | a b c d e f g h i j k l m n o p q r s t u v w x y z | | | | | |
| digit | 0 1 2 3 4 5 6 7 8 9 | | | | | |
| special | ( )<br>(parentheses) | .<br>(full stop) | ,<br>(comma) | :<br>(colon) | ;<br>(semicolon) | -<br>(hyphen minus) |
| | { }<br>(curly brackets) | /<br>(solidus) | *<br>(asterisk) | ^<br>(circumflex) | =<br>(equals sign) | [ ]<br>(square brackets) |
| underscore | _<br>(low line) | | | | | |
| apostrophe | '<br>(apostrophe) | | | | | |
| quote | "<br>(quotation mark) | | | | | |
| escape | !<br>(exclamation mark) | | | | | |
| space | | | | | | |

```
letter =            "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
                    "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" .
digit =             "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
special =           "(" | ")" | "." | "," | ":" | ";" | "=" | "/" | "*" | "-" | "{" | "}" | "[" | "]" .
underscore =        "_" .
apostrophe =        "'" .
quote =             "" .
escape =            "!" .
space =             " " .
non-quote-character = letter | digit | underscore | special | apostrophe | space .
bound-character = non-quote-character | quote .
added-character = not defined by this International Standard .
```

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

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

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

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

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

## 7.2 Whitespace

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

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

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

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

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

## 7.3 Lexical objects

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

### 7.3.1 Identifiers

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

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

**Table 7-2 — Reserved Keywords**

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

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

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

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

### 7.3.2 Digit-string

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

digit-string = digit { digit } .

### 7.3.3 Character-literal and string-literal

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

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

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

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

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

### 7.3.4 Keywords

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

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

## 7.4 Annotations

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

annotation = "[" annotation-label ":" annotation-text "]" .
annotation-label = objectidentifiercomponent-list .
annotation-text = *not defined by this International Standard* .

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

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

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

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

[rpc: discriminant = n].

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

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

## 7.5 Values

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

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

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

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

### 7.5.1 Independent values

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

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

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

NOTES

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 datatypes. A LI datatype is either:

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

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

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

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

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

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

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

NOTES

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

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

## 8.1 Primitive datatypes

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

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

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

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

Operations: definitions of characterizing operations.

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

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

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

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

parameter-name : parameter-datatype, or

parameter-name$_1$ , parameter-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.

### 8.1.1 Boolean

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

Syntax:

boolean-type = "boolean" .

Parametric Values: none.

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

Value-syntax:

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

Properties: unordered, exact, non-numeric.

Operations: Equal, Not, And, Or.

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

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

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

| x | Not(x) |
|---|---|
| true | false |
| false | true |

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

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

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

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

### 8.1.2 State

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

Syntax:

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

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

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

Value-syntax:

state-literal = identifier .

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

Properties: unordered, exact, non-numeric.

Operations: Equal.

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

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

EXAMPLE — The declaration:

type switch = new state (on, off);

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

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

### 8.1.3 Enumerated

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

Syntax:

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

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

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

Value-syntax:

enumerated-literal = identifier .

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

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

Operations: Equal, InOrder, Successor

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

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

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

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

### 8.1.4 Character

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

Syntax:

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

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

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

Value-syntax:

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

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

Properties: unordered, exact, non-numeric.

Operations: Equal.

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

NOTES

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

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

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

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

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

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

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

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

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

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

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

### 8.1.5 Ordinal

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

Syntax:

ordinal-type = "ordinal" .

Parametric Values: none.

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

Value-syntax:

ordinal-literal = number .

number = digit-string .

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

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

Operations: Equal, InOrder, Successor

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

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

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

### 8.1.6 Date-and-Time

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

Syntax:

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

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

radix = value-expression .

factor = value-expression .

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

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

Value-syntax:

time-literal = string-literal .

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

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

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

Equal(x, y: time(*time-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, Dayof-Month, TimeofDay, Hour, Minute, Second, can be derived from Round, Extend, and Difference.

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

### 8.1.7 Integer

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

Syntax:

integer-type = "integer" .

Parametric Values: none.

Values: Mathematically, the infinite ring produced from the additive identity (0) and the multiplicative identity (1) by requiring $0 \leq 1$ and $\text{Add}(x,1) \neq y$ 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))));

### 8.1.8 Rational

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

Syntax:

rational-type = "rational" .

Parametric Values: none.

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

Value-syntax:

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

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

Promote(*signed-number*),

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

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

Properties: ordered, exact, numeric, unbounded.

Operations: Equal, NonNegative, InOrder, 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)))

### 8.1.9 Scaled

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

Syntax:

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

radix = value-expression .

factor = value-expression .

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

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

Value-syntax:

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

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

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

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

Properties: ordered, exact, numeric, unbounded.

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

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

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

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

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

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

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

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

EXAMPLES

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

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

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

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

NOTES

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

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

### 8.1.10 Real

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

Syntax:

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

radix = value-expression .
factor = value-expression .

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

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

Let $\Re$ 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 $\bar{r}$ in $V$ such that $| r - \bar{r} | \leq | r | \bullet \varepsilon$;
– for each $r$ in $\Re$ such that $| r | < \varepsilon$, there exists at least one $\bar{r}$ in $V$ such that $| r - \bar{r} | \leq \varepsilon^2$;.

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 $\bar{r}$ in $V$ with the properties given above and the further requirement that for each $v$ in $V$, $M(v) = v$.

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

InOrder(x,y: real(*radix, factor*)): boolean **is** true if $x \leq y$, where $\leq$ designates the order relationship on $\Re$, 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 $x \neq 0$, = $M$(x') where x' is the real multiplicative inverse of x.

NOTES

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

2. Detailed requirements for the approximation function, its relationship to the characterizing operations, and the implementation of the characterizing operations in languages are provided by ISO/IEC 10967-1:1994, Information technology — Programming languages, their 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.

### 8.1.11 Complex

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

Syntax:

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

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

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

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

– 0 is in *V*;
– for each *v* in *C* such that $|v| \geq \varepsilon$, there exists at least one $\overline{v}$ in *V* such that $|v - \overline{v}| \leq |v| \bullet \varepsilon$;
– for each *v* in *C* such that $|v| < \varepsilon$, there exists at least one $\overline{v}$ in *V* such that $|v - \overline{v}| \leq \varepsilon^2$;.

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 $\overline{v}$ 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.

### 8.1.12 Void

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

Syntax:

void-type = "void" .

Parametric Values: none.

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

Value-syntax:

void-literal = "nil" .

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

Properties: none.

Operations: Equal.

Equal(x, y: void) = true;

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).

## 8.2 Subtypes and extended types

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

subtype = range-subtype | selecting-subtype | excluding-subtype
| size-subtype | explicit-subtype | extended-type .

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

| | |
|---|---|
| Description: | prose description of the subtype value space. |
| Syntax: | the syntactic production for a subtype resulting from the subtype generator, including identification of all parametric values which are necessary for the complete identification of a distinct subtype. |
| Components: | constraints on the base datatype and parametric values. |
| Values: | formal definition of resulting value space. |

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

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

### 8.2.1 Range

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

Syntax:

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

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

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

### 8.2.2 Selecting

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

Syntax:

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

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

Values: The values specified by the *select-list* designate those values from the value-space of the base datatype which comprise the value-space of the selecting subtype. A *select-item* which is a *value-expression* specifies the single value designated by that *value-expression*. A *select-item* which is a *select-range* specifies all values *v* of the base datatype such that *lowerbound* ≤ *v*, if *lowerbound* is specified, and *v* ≤ *upperbound*, if *upperbound* is specified.

Properties: The subtype is bounded (above, below, both) 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.

### 8.2.3 Excluding

Description: Excluding creates a subtype of any exact datatype by enumerating the values which are to be excluded in construct-

ing the subtype value-space.

Syntax:

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

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

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

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

### 8.2.4 Size

Description: Size creates a subtype of any Sequence, Set, Bag or Table datatype by specifying bounds on the number of elements any value of the base datatype may contain.

Syntax:

size-subtype = base "size" "(" minimum-size [ ".." maximum-size ] ")" .
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.

### 8.2.5 Explicit subtypes

Description: Explicit subtyping identifies a datatype as a subtype of the base datatype and defines the construction procedure for the subset value space in terms of 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 (isomor-

phic 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)).

### 8.2.6 Extended

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

Syntax:

```
extended-type = base "plus" "(" 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 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 .
```

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

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

| | |
|---|---|
| Syntax: | the syntactic production for a generated datatype resulting from the datatype generator, including identification of all parametric datatypes which are necessary for the complete identification of a distinct datatype. |
| Components: | number of and constraints on the parametric datatypes and parametric values used by the generator. |
| Values: | formal definition of resulting value space. |
| Properties: | properties of the resulting datatype which indicate its admissibility as a component datatype of certain datatype generators: numeric or non-numeric, approximate or exact, ordered or unordered, and if ordered, bounded or unbounded. |
| Subtypes: | generators, subtype-generators and parametric values which produce subset value spaces. |
| Operations: | characterizing operations for the resulting datatype which associate to the datatype generator. The definitions of operations have the form described in 8.1. |

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

### 8.3.1 Choice

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

Syntax:

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

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

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

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

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

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

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

Value-syntax:

choice-value = "(" tag-value ":" alternative-value ")" .

tag-value = independent-value .

alternative-value = independent-value .

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

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

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

Operations: Equal, Tag, Cast, Discriminant.

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

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

Cast.*type*(x: choice (*tag-type*) of (*alternative-list*)): *type*, where *type* is an alternative datatype in *alternative-list*, **is**:
if the tag value of x selects an alternative whose *alternative-type* is *type*, then that value of *type* which is the (alternative) value of x, else undefined.

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

### 8.3.2 Pointer

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

Syntax:

pointer-type = "pointer" "to" "(" element-type ")" .
element-type = type-specifier .

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

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

Value-syntax:

pointer-literal = "null" .

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

Properties: unordered, exact, non-numeric.

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

Operations: Equal, Dereference.

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

Dereference(x: pointer(*element*)): *element*, where x ≠ 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.

### 8.3.3 Procedure

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

Syntax:

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

Components: A *parameter-type* may designate any datatype. The *parameter-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:

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

where $A^0$ is the normal result space and $A^k$ is the alternative result space of the *k*th termination; and every value of the procedure datatype is a function of the form:

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

Any of the input space, the normal result space and the alternative result space corresponding to a given *termination-iden-*

*tifier* 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*).

## 8.4 Aggregate Datatypes

An **aggregate datatype** is a generated datatype each of whose values is, in principle, made up of values of the component datatypes. An aggregate datatype generator generates a datatype by

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

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

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

```
aggregate-type = record-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 characterising 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.

### 8.4.1 Record

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

Syntax:

```
record-type = "record" "(" 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.

### 8.4.2 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*: $E \rightarrow B$, 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 $v \neq$ y, *s(v)* = false; i.e. the set consisting of the single value y;

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

NOTE — Set is 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.

### 8.4.3 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 $b: E \rightarrow Z$, 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 $e \neq y$, z(*e*) = x(*e*), and
if x(y) > 0 then z(y) = x(y) - 1 and if x(y) = 0 then z(y) = 0;
i.e. the collection formed by deleting one instance of the value y, if any, from the collection 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 $e \neq y$, z(*e*) = x(*e*), and z(y) = x(y) + 1;
i.e. the collection formed by adding one instance of the value y to the collection x;

### 8.4.4 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.

### 8.4.5 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 $x \neq upperbound_i$.

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:

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

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.

### 8.4.6 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 val-

ue 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 extention 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 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.

#### 10.1.1 Natural number

Description: Naturalnumber is the datatype of the cardinal or natural numbers.

Declaration:

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

Parametric Values: none.

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

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

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

### 10.1.2 Modulo

Description: Modulo is a family of 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 $1 \leq$ *modulus*, designated the *modulus* of the Modulo datatype.

Values: all Integer values *v* such that $0 \leq v$ 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*).

### 10.1.3 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.

### 10.1.4 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.

### 10.1.5 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).

### 10.1.6 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.

### 10.1.7 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.

### 10.1.8 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.

### 10.1.9 Private

Description: A Private datatype represents an application-defined value-space and operation set which are intentionally con-

cealed 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.

### 10.1.10 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.<br><br>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.

## 11.1 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 Clauses 7 through 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.

## 11.2 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.

## 11.3 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.

## 11.4 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.

### 11.4.1 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.

### 11.4.2 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, $a' \leq b'$.

### 11.4.3 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.

### 11.4.4 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).

### 11.4.5 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.

### 11.4.6 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.

# Annex A
(informative)

# Character-Set Standards

The following is a partial list of International Standards which define character-sets. Character sets defined by such standards are suitable for reference by a "repertoire-identifier" in the Character and CharacterString datatypes.

These standards define character-sets, in the sense of repertoires of characters. Most of them also define "character codes" — integer values used to represent the character values for certain computational purposes. Whether "character(*repertoire*)" is interpreted as requiring the characters to be represented by the codes defined by the *repertoire* is outside of the scope of this International Standard.

None of these standards defines a collating sequence or order relationship on the character-sets. The definition of such an order relationship requires additional standards or application agreements. Order relationships commonly supported by programming languages are based on the integer ordering of the code values used in a particular implementation of the language. Such orderings have no semantics with respect to the character-set itself and are outside the scope of this International Standard.

| | |
|---|---|
| ISO/IEC 646:1991 | *Information technology — ISO 7-bit coded character set for information interchange* |
| ISO 2047:1975 | *Information processing — Graphical representations for the control characters of the 7-bit coded character set* |
| ISO 9036:1987 | *Information processing — Arabic 7-bit coded character set for information interchange* |
| ISO/IEC 2022:1994 | *Information technology — Character code structure and extension techniques* |
| ISO/IEC 6937:1994 | *Information technology — Coded graphic character set for text communication — Latin alphabet* |
| ISO/IEC 4873:1991 | *Information technology — ISO 8-bit code for information interchange — Structure and rules for implementation* |
| ISO 8859-1:1987 | *Information processing — 8-bit single byte coded graphic character sets — Part 1: Latin alphabet No. 1* |
| ISO 8859-2:1987 | *Information processing — 8-bit single byte coded graphic character sets — Part 2: Latin alphabet No. 2* |
| ISO 8859-3:1988 | *Information processing — 8-bit single byte coded graphic character sets — Part 3: Latin alphabet No. 3* |
| ISO 8859-4:1988 | *Information processing — 8-bit single byte coded graphic character sets — Part 4: Latin alphabet No. 4* |
| ISO/IEC 8859-5:1988 | *Information processing — 8-bit single byte coded graphic character sets — Part 5: Latin/Cyrillic alphabet* |
| ISO 8859-6:1987 | *Information processing — 8-bit single byte coded graphic character sets — Part 6: Latin/Arabic alphabet* |
| ISO 8859-7:1987 | *Information processing — 8-bit single byte coded graphic character sets — Part 7: Latin/Greek alphabet* |
| ISO 8859-8:1988 | *Information processing — 8-bit single byte coded graphic character sets — Part 8: Latin/Hebrew alphabet* |
| ISO/IEC 8859-9:1989 | *Information processing — 8-bit single byte coded graphic character sets — Part 9: Latin alphabet No. 5* |

ISO/IEC 8859-10:1992 *Information technology — 8-bit single byte coded graphic character sets — Part 10: Latin alphabet No. 6*

ISO/IEC 10367:1991 *Information technology — Standardized coded graphic character sets for use in 8-bit codes*

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

ISO/IEC 6429:1992 *Information technology — Control functions for coded character sets*

ISO 6630: 1986 *Documentation — Bibliographic control characters*

ISO/IEC 10538:1991 *Information technology — Control functions for text communication*

ISO 5426:1983 *Extension of the Latin alphabet coded character set for bibliographic information interchange*

ISO 5427:1984 *Extension of the Cyrillic alphabet coded character set for bibliographic information interchange*

ISO 5428:1984 *Greek alphabet coded character set for bibliographic information interchange*

ISO 6438:1983 *Documentation — African coded character set for bibliographic information interchange*

ISO 6861: —[1] *Information and documentation — Cyrillic alphabet coded character sets for historic Slavonic languages and European non-Slavonic languages written in a Cyrillic script, for bibliographic information interchange*

ISO 6862: —[1] *Information and documentation — Mathematical coded character set for bibliographic information interchange*

ISO 8957: —[1] *Information and documentation — Hebrew alphabet coded character sets for bibliographic information interchange*

ISO 10585: —[1] *Information and documentation — Armenian alphabet coded character set for bibliographic information interchange*

ISO 10586: —[1] *Information and documentation — Georgian alphabet coded character set for bibliographic information interchange*

ISO 10754: —[1] *Information and documentation — Extension of the Cyrillic alphabet coded character set for non-Slavic languages for bibliographic information interchange*

ISO/IEC 9541-1:1991 *Information technology — Font information interchange — Part 1: Architecture*

ISO/IEC 9541-2:1991 *Information technology — Font information interchange — Part 2: Interchange Format*

ISO/IEC 9541-3:1994 *Information technology — Font information interchange — Part 3: Glyph Shape Representation*

ISO/IEC 9541-4: —[1] *Information technology — Font information interchange — Part 4: Application-specific requirements*

ISO 6093:1985 *Information processing — Representation of numeric values in character strings for information interchange*
(defines character sets and syntax for numeric strings)

ISO/IEC 8824:1990 *Information technology — Open Systems Interconnection — Abstract Syntax Notation One (ASN.1)*
(defines interchange character sets both directly and by reference to sets registered under ISO 2375)

1. To be published

The following are International Standards for character-set registration. Character sets registered under the provisions of these standards are suitable for reference by a "repertoire-identifier" in the Character and CharacterString datatypes.

| | |
|---|---|
| ISO 2375:1985 | *Data Processing — Procedure for the registration of escape sequences* |
| ISO/IEC 7350:1991 | *Information technology — Registration of repertoires of graphic characters from ISO 10367* |
| ISO/IEC 10036:1993 | *Information technology — Font information interchange — Procedure for registration of glyph and glyph collection identifiers* |

# Annex B
(informative)

# Recommended Placement of Annotations

An ***annotation*** (see 7.4) is a descriptive information unit attached to a *type-specifier*, or a component datatype, or a procedure (value), to characterize some aspect of the representations, variables, or operations associated with values of the datatype, or the component or procedure, in some particular context. This International Standard does not specify the syntax or semantics of any specific *annotations*. Common conventions for the placement of *annotations*, however, makes it easier for the reader to determine the object to which an *annotation* is intended to apply and the context in which it is intended to apply. This annex contains guidelines for placement of *annotations* in the syntax and corresponding distinctions in the scope of application of the *annotations*, as required by clause 7.4.

Use of the recommended placement conventions improves the compatibility of usages and implementations of the LI datatypes, to the extent that they involve such annotations. Use of additional or substitute conventions by other standards and implementations is consistent with this International Standard.

## B.1 Type-attributes

A **type-attribute** is an *annotation* attached to a *type-specifier*, and in particular to the *type-specifier* of a *type-definition*, which characterizes some aspect of the values or variables of the datatype specified, or the operations on those values or variables, in some particular context. Type-attributes may include, among others:

- limitations on, or identification of parameters describing, the value-space of the datatype as implemented, or as used in a particular context,
- constraints on, or specifications for, representation of the values of the datatype,
- constraints on, or specifications for, the operations which may be performed on values of the datatype,
- identification of procedures or parameters to be used for conversion of values of the datatype for a particular interchange or external medium.

Type-attributes should immediately follow the *type-specifier* for the datatype to which they are intended to apply. In particular, an *annotation* which applies to the *element-type* of an *aggregate-type* should appear **inside** the parentheses, while an *annotation* which applies to the *aggregate-type* should appear **outside** the parentheses.

## B.2 Component-attributes

A **component-attribute** is an *annotation* attached to a component of a *generated-type* which characterizes some aspect of the operations on, or representations of, values in that component of the particular generated datatype (i.e. values used in that role, as distinct from general limitations on values of the datatype of the component) in some particular context. Component-attributes may include, among others:

- any of the attribute notions given in B.1, but restricted to the component,
- specification of the ordering, representation or alignment of the component in an aggregate structure,
- limitations on access to the component.

Component-attributes should immediately precede the component *type-specifier* for the component to which they are intended to apply. That is, in a *record-type*, they should precede the *field-type*; in a *choice-type*, they should precede the *alternative-type*; and in a homogeneous *aggregate-type*, they should precede the *element-type*.

## B.3 Procedure-attributes

A **procedure-attribute** is an *annotation* attached to a *procedure-declaration* which characterizes some aspect of the invocation or use of the named procedure, in some particular context. Procedure-attributes may include, among others:

- specification of the location of its instantiations,

- specification of the procedure interface.

Procedure-attributes should precede the keyword “procedure” or follow the entire *type-specifier*. In addition, procedure-attributes should be distinguishable from type- or component- attributes by their text.

## B.4 Argument-attributes

An **argument-attribute** is an *annotation* attached to an *argument* to a *procedure-declaration* or *procedure-type* which characterizes some aspect of the operations on, or representations of, values passed through that argument of the particular procedure or procedure datatype (as distinct from general limitations on the datatype which is the *argument-type*) in some particular context. Argument-attributes may include, among others:

- any of the attribute notions given in B.1, but restricted to the use of the datatype in this argument,
- specification of the means of passing the argument.

Argument-attributes should immediately precede the *argument* or *return-argument* which they are intended to describe (in a *procedure-type*, a *procedure-declaration*, or a *termination-declaration*).

# Annex C
(informative)

# Implementation Notions of Datatypes

This annex defines a collection of datatype notions excluded from this International Standard, because they were deemed to be notions of implementation or representation of datatypes, rather than conceptual notions.

The values of the datatypes defined by this International Standard are abstract objects conforming to a set of given rules. Each computer system has its own **internal datatypes**, whose value spaces are (typically fixed-length) sequences of *n* distinguished symbols (most commonly, the two symbols "0" and "1"), and whose characterizing operations are the **instructions** built into the computer system. A **representation** of a LI datatype is a mapping from the value space of the LI datatype to a computer system value space.

In addition to *values* of datatypes, a computer system has the notion of **variable** – an object to which a value of some datatype or datatypes is dynamically associated. (In a certain sense, a variable is an implementation of a value of a pointer datatype (8.3.2).) The characterizing operations defined by this International Standard are abstract computational notions of functions applicable to the values of datatypes, used to identify the semantics of the datatypes. In a computer system, the operations on representations of those values and variables containing those representations are actually **executed**.

The characteristics of representations, variables, and the execution of operations are beyond the scope of this International Standard. Nonetheless, because these characteristics are inextricably mixed with the datatype notions in many programming languages, and because these characteristics are important to many applications of this International Standard, this International Standard provides for their inclusion in *type-specifiers* and in datatype- and procedure-declarations via *annotations* (see 7.4). An ***annotation*** is 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, or the component or procedure, in some particular context.

This annex identifies notions for which such *annotations* may be appropriate and even necessary for certain language mappings. This International Standard does not specify the syntax or semantics of any specific *annotations* to describe implementation notions. The development of standards for such *annotations* may be appropriate, but is outside the scope of this International Standard.

## C.1 StorageSize

StorageSize is a type-attribute specifying the number (and type) of storage units required or allotted to represent values of the datatype. It may also specify whether the number of storage units is constant over all values of (this instance of) the datatype, or varies according to the requirements of the particular value to be represented.

StorageSize may apply to any datatype, except procedure datatypes.

NOTE — If there is a limitation on the maximum size of representable values, it implies that there is a limitation on the value space of this datatype, which may be better documented by appropriate subtype specifications (see 8.2).

## C.2 Mode

Mode is a type-attribute which specifies the radix of representation of a numeric datatype, the representation of the digits, the representation of the decimal-point, if any, and the sign representation and placement conventions. Such notions as “two’s complement binary”, “packed decimal with trailing sign” and the numeric representation formats of ISO 6093:1985, *Information processing — Representation of numeric values in character strings for information interchange,* are examples of “modes”.

Mode applies only to numeric datatypes, principally Integer and Scaled.

## C.3 Floating-Point

Floating-point is a type-attribute which specifies that a numeric datatype has a floating-point representation and the characteristics of that representation.

Following ISO/IEC 10967-1:1994, *Information technology — Programming languages, their envrionements and system software interfaces — Language-independent arithmetic — Part 1: Integer and real arithmetic*, a floating-point representation of the value v has the form:

$v = S \bullet M \bullet R^E$

where

R is the *radix* of the representation;
E is the *exponent*;, and
S is the *sign*, i.e. either S = 1 or S = -1;
M is the *mantissa*, either zero or a value of the datatype scaled(*radix*, *precision*) range(*radix* ^ - *precision*, 1) excluding(1).

This representation can be characterized by five parameters:

*radix* and *precision*, from above;
*emin* and *emax*, with the requirement: $emin \leq E \leq emax$; and
*denorm*, with the requirement that *denorm* = "false" implies $d = R^{-1}$ and *denorm* = "true" implies $d = R^{-precision}$.

Floating-point applies only to numeric datatypes, principally Real and Complex.

## C.4 Fixed-Point

Fixed-point is a type-attribute which specifies that a numeric datatype has a fixed-point representation and the characteristics of that representation.

A fixed-point representation has the form:

$v = S \times M \times R^{-P}$

where

R is the *radix* of the representation;
S is the *sign*, i.e. either S = 1 or S = -1;
M is the *mantissa*, a value of the datatype Integer;
P is the *precision*.

This representation can be characterized by the *radix* and *precision* parameters.

Fixed-point applies only to numeric datatypes, principally Scaled.

## C.5 Tag

Tag is a type-attribute which specifies whether and how the tag-value of a value of a value of a choice datatype is represented.

Tag applies only to choice datatypes or their generators.

## C.6 Discriminant

Discriminant specifies the source of the discriminant value of a Choice datatype.

Discriminant applies only to choice datatypes or their generators.

## C.7 StorageSequence

StorageSequence attributes describe the order of presentation of the component values of a value of an aggregate datatype, such as Set or Record, whose ordering is not implied by the type properties. Their values and meaning depend on the aggregate datatype involved.

StorageSequence attributes apply only to aggregate datatypes or to their generators.

## C.8 Packed

Packed and “unpacked” or “aligned” are type-attributes which characterize the juxtaposition of all components of a value of an aggregate datatype. They distinguish between the optimization of space and the optimization of access-time.

Packed attributes apply only to aggregate datatypes or to their generators.

## C.9 Alignment

Alignment is a component-attribute that characterizes the forced alignment of the representations of values of a given component datatype on storage-unit boundaries. It implies that "padding" to achieve the necessary alignment may be inserted in the representation of the aggregate datatype which contains the annotated component.

## C.10 Form

Form is a type-attribute which specifies that one datatype has the same representation as another. In particular, *form* permits an implementation to specify that a primitive LI datatype has a visible information structure, or that a particular generated datatype has a primitive implementation.

Form may apply to any datatype.

# Annex D
(informative)

# Syntax for the Common Interface Definition Notation

The syntax used in this International Standard is a subset of the syntax prescribed for the Interface Definition Notation (IDN) in ISO/IEC 13886:1995, *Information technology — Programming languages — Language-independent procedure calling*. This annex contains the the complete IDN syntax, for reference only. A conforming IDN text is an *interface-type*, whereas a conforming LI datatype specification is a *type-specifier*. In addition, a mapping, as provided in Clause 11, may contain *declarations*.

Character-set productions: Normative text page(s)

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" . 13
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" . 13
special = "(" | ")" | "." | "," | ":" | ";" | "=" | "/" | "*" | "-" | "{" | "}" | "[" | "]" . 13
apostrophe = "'" . 13
escape = "!" . 13
quote = '"' . 13
space = " " . 13
underscore = "_" . 13
added-character = *not defined by this International Standard* . 13, 20, 49
bound-character = non-quote-character | quote . 13, 20
non-quote-character = letter | digit | underscore | special | apostrophe | space . 13, 20, 49

NOTE — Character-set productions are always subject to minor changes from implementation to implementation, in order to handle the vagaries of available character-sets.

Productions of the IDN used in this International Standard: Normative text page(s)

actual-type-parameter = value-expression | type-specifier . 44
actual-type-parameter-list = actual-type-parameter { "," actual-type-parameter } . 44
aggregate-type = record-type | set-type | sequence-type | bag-type | array-type | table-type . 36
alternative = tag-value-list [ field-identifier ] ":" alternative-type . 31
alternative-list = alternative { "," alternative } [ default-alternative ] . 31
alternative-type = type-specifier . 31
alternative-value = independent-value . 32
annotation = "[" annotation-label ":" annotation-text "]" . 14
annotation-label = objectidentifiercomponent-list . 15
annotation-text = *not defined by this International Standard* . 15
any-character = bound-character | added-character | escape-character . 14, 20
array-type = "array" "(" index-type-list ")" "of" "(" element-type ")" . 41
array-value = value-list . 42
ASN-object-identifier = “{“ objectidentifiercomponent-list “}” . 51
bag-type = "bag" "of" "(" element-type ")" . 39
bag-value = empty-value | value-list . 40
base = type-specifier . 28, 29, 30
bit-literal = "0" | "1" . 48
bitstring-literal = quote { bit-literal } quote . 48

value-declaration = "value" value-identifier ":" type-specifier "=" independent-value . 46
value-expression = independent-value | dependent-value | formal-parametric-value . 15
value-identifier = identifier . 46
value-list = "(" independent-value { "," independent-value } ")" . 37, 39, 40, 42, 43
value-reference = value-identifier . 15
void-literal = "nil" . 27
void-type = "void" . 27

Productions of the common IDN which appear in a more restricted form above:

procedure-reference = [ interface-synonym "::" ] procedure-identifier .
termination-reference = [ interface-synonym "::" ] termination-identifier .
type-reference = [ interface-synonym "::" ] type-identifier [ "(" actual-type-parameter-list ")" ] .
value-reference = [ interface-synonym "::" ] value-identifier .

Additional productions of the IDN not used in this International Standard:

interface-type = "interface" interface-reference "begin" interface-body "end" .
interface-reference = interface-synonym | [ interface-synonym ":" ] interface-identifier .
interface-identifier = object-identifier-value .
interface-synonym = identifier .
interface-body = { import } { declaration ";" } .
import = "imports" [ "(" import-symbol-list ")" ] "from" interface-reference .
import-symbol-list = import-symbol { "," import-symbol } .
import-symbol = procedure-identifier | termination-identifier | type-identifier | value-identifier .

# Annex D
(informative)

# Example Mapping to Pascal

This annex contains a draft "inward" mapping from the LI datatypes into the programming language Pascal, as defined by ISO/IEC 7185:1990, *Information technology — Programming languages — Pascal*. Where appropriate, differences in the mapping to the Extended Pascal language (ISO/IEC 10206:1991, *Information technology — Programming languages — Extended Pascal*) are noted.

The purpose of this annex is to exemplify the nature and content of an inward mapping, and possibly a mapping standard. This mapping should not be considered a definitive mapping from LI datatypes to the Pascal language.

## D.1 LI Primitive Datatypes

### D.1.1 Boolean

Boolean maps to the Pascal type Boolean. Values true and false map to the corresponding values of Pascal Boolean. All characterizing operations are preserved, using the Boolean operators of Pascal.

### D.1.2 State

A state datatype of the form state(*state-value-list*) maps to the Pascal enumeration type (state-value-list). Each state-value is mapped to the Pascal value with the corresponding identifier. All characterizing operations are preserved.

### D.1.3 Enumerated

An enumerated datatype of the form enumerated(enumerated-value-list) maps to the Pascal enumeration type (enumerated-value-list). Each enumerated-value is mapped to the Pascal value with the corresponding identifier. All characterizing operations are preserved.

### D.1.4 Character

A single character datatype of the form character or character(repertoire-list) maps to the Pascal type char. Pascal requires each implementation to define the character-set associated with the type char. The default character-set designated by the LI datatype syntax character is presumed to be that character-set, and *repertoire-list*, if present, must identify that character-set, or a subset of it. Each character-value in that character-set is mapped to the Pascal value having the same character-code. All characterizing operations are preserved.

No other character datatype is mapped into a Pascal datatype, although an implementation may specify a mapping of the character-codes into the Pascal type integer.

### D.1.5 Ordinal

The LI datatype ordinal range(1..maxint) maps to the Pascal subrange type 1..maxint. Pascal requires each implementation to define the value of maxint. The ordinal datatype with the corresponding maximum value (and any subtype thereof) is mapped as given above, with each ordinal value being mapped to the corresponding integer value under the mathematical isomorphism. All characterizing operations are preserved.

No ordinal value greater than *maxint* can be mapped, and no datatype containing such a value can be mapped into Pascal.

### D.1.6 Date-and-time

The LI datatype time(*unit, radix, factor*) range(time1..time2) is mapped to Pascal in the same way that time interval datatypes are mapped (see D.4.6), with the convention that the Pascal value represents the interval between time1 and the designated point in time, but only if the Pascal value representing the interval time2 – time1is less than the implementation-defined value maxint.

No other date-and-time types can be mapped to Pascal.

### D.1.7 Integer

The LI datatype integer range(minint..maxint) maps to the Pascal type integer, where *minint* is defined to be Negate(*maxint*). Pascal requires each implementation to define the value of maxint. The integer datatype with the corresponding minimum and maximum values (and any subtype thereof) is mapped to the Pascal type integer, with each integer value being mapped into the identical Pascal integer value. All characterizing operations are preserved.

No integer value greater than maxint can be mapped, no integer value less than minint can be mapped, and no datatype containing such a value can be mapped into Pascal.

### D.1.8 Rational

Rational maps to the Pascal type declared by

```
type rational = array [1..2] of integer;
```

with the characterizing operations defined as follows:

```
procedure Reduce(var x: rational);   (* reduces a rational value to lowest-terms *)
    var t, r, d: integer;
    begin
        d := abs(x[1]);
        r := abs(x[2]);
        while (d mod r) > 0 do begin
            t := d mod r;
            d := r;  r := t;
        end;
        x[1] := x[1] div r;
        x[2] := x[2] div r;
    end;

procedure Add(x, y: rational; var t: rational);
    begin
        if x[2] = y[2] then begin
            t[1] := x[1] + y[1];
            t[2] := x[2];
        end else begin
            t[1] := x[1] * y[2] + y[1] * x[2];
            t[2] := x[2] * y[2];
        end;
        Reduce(t);
    end;

procedure Multiply(x, y: rational; var t: rational);
    begin
        t[1] := x[1] * y[1];
        t[2] := x[2] * y[2];
        Reduce(t);
    end;

procedure Negate(x: rational; var t: rational);
    begin
        t[1] :=  - x[1];
        t[2] := x[2];
    end;

procedure Reciprocal(x: rational; var t: rational);
    begin
        t[1] := x[2];
        t[2] := x[1];
        if t[2] < 0 then begin
            t[1] := -t[1];
            t[2] := -t[2];
```

```
            end;
        end;
    function NonNegative(x: rational): Boolean;
        begin NonNegative := (x[1] >= 0)  end;
    function Equal(x, y: rational): Boolean;
        begin Equal := ((x[1] * y[2]) = (x[2] * y[1]))  end;
```

Only rational values whose numerator and denominator are both within the range [*–maxint, maxint*] are mapped into the Pascal datatype. (This cannot be stated as a range constraint on the value space of the Rational datatype.)

NOTE — The above procedures are not optimal and a good implementation would require techniques for sign management and overflow avoidance. These procedures are intended only as a demonstration that the characterizing operations can be implemented "conveniently" on the type as mapped.

### D.1.9 Scaled

The LI datatype scaled(r, f) range(minrf..maxrf) maps to the Pascal type integer, where *minrf* has the value $-maxint \cdot r^{(-f)}$ and *maxrf* has the value $maxint \cdot r^{(-f)}$. A scaled datatype with the corresponding minimum and maximum values (and any subtype thereof) is mapped to the Pascaltype integer, with each scaled value $N \cdot r^{(-f)}$ being mapped into the Pascal integer value N. In order for the characterizing operations to be preserved, scaled multiply and divide operations have to be defined, as follows:

```
type scaled = integer;
(* const rtothef = r pow f; *)
function scaledMultiply(x, y: scaled): scaled;
    var
        t: scaled;
        round: Boolean;
        negate: Boolean;
    begin
        t := x * y;
        negate := (t < 0);
        if negate then t := -t;
        round := (t mod rtothef > rtothef / 2);
        t := t div rtothef;
        if round then t := t + 1;
        if negate then t := -t;
        scaledMultiply := t;
    end;
function scaledDivide(x, y: scaled): scaled;
    var
        t: scaled;
        negate: Boolean;
    begin
        negate := (x < 0);
        if negate then x := -x;
        if y < 0 then begin
            negate := not negate;
            y := -y;
        end;
        t := ( x * rtothef ) / y;
        if (x * rtothef mod y) > rtothef / 2 then t := t + 1;
        if negate then t := -t;
        scaledDivide := t;
    end;
```

Only those values of the datatype scaled(r, f) which are within the above range are mapped and no scaled datatype containing values outside this range can be mapped into Pascal.

NOTE — A more general version of the scaled datatype can be defined using the Pascal type:

```
    type scaled =  record
```

```
                numerator: integer;
                radix: 0..maxint;
                factor: integer
        end;
```

with "characterizing operations" which generalize the arithmetic on scaled datatypes. This model can be further tailored to a fixed radix (like 10) to get improved performance. The integer model is more useful for simple exchanges of information, while the generalized model is preferable for extensive manipulation of scaled values.

### D.1.10 Real

The LI datatypes real range(rmin..rmax) and real(radix, precision) range(rmin..rmax) map to the Pascal type real, only if the given or default *radix, precision, rmin* and *rmax* parameters define a subset of the real values which is distinguishable in the subset of the mathematical real values defined by the Pascal implementation under the following mapping: Each LI Real value is mapped into the Pascal real value which is mathematically nearest it and if two values are equidistant then either may be chosen. All characterizing operations are conceptually preserved, although the implementation-defined arithmetic may affect the correctness of results.

No real value requiring more range or more precision can be mapped, and no datatype containing such a value can be mapped into Pascal.

### D.1.11 Complex

The LI datatypes complex and complex(radix, precision) are mapped into Pascal using the Pascal type:

```
type complex = record realpart,  imagpart: real end;
```

This type, however, only maps values $c$ in $C$ such that $|\mathrm{Re}(c)| < rmax$ and $|\mathrm{Im}(c)| < rmax$, where *rmax* is implementation-defined, and then only if *rmax* and the given or default *radix* and *precision* parameters define a subset of the complex values whose Cartesian representations $(x + iy)$ are distinguishable in the Cartesian product of the real values defined by the Pascal implementation. (This cannot be stated as a constraint on the value space of the LI complex datatype.) No complex datatype requiring more range or precision can be mapped.

Each LI Complex value $c$ is mapped to the Pascal value whose realpart field has the Pascal real value mathematically nearest $\mathrm{Re}(c)$ and whose imagpart field has the Pascal real value mathematically nearest $\mathrm{Im}(c)$. (Re and Im are the mathematical projections onto the real and imaginary axes, respectively.)

The definition of "characterizing operations" appropriate to the Cartesian representation of a complex-number can be defined by the following Pascal procedures, although the implementation-defined arithmetic may affect the correctness of results.

```
function Equal(x, y: complex): Boolean;
    begin Equal := (x.realpart = y.realpart) and (x.imagpart = y.imagpart)  end;

procedure Promote(x: real; var t: complex);
    begin t.realpart := x; t.imagpart := 0.0; end;

procedure Add(x, y: complex; var t: complex);
    begin
        t.realpart := x.realpart + y.realpart;
        t.imagpart := x.imagpart + y.imagpart;
    end;

procedure Multiply(x, y: complex; var t: complex);
    begin
        t.realpart := x.realpart * y.realpart - x.imagpart * y.imagpart;
        t.imagpart := x.realpart * y.imagpart + x.imagpart * y.realpart;
    end;

procedure Negate(x: complex; var t: complex);
    begin
        t.realpart := - x.realpart
        t.imagpart := - x.imagpart;
    end;

procedure Reciprocal(x: complex; var t: complex);
    var r: real;
    begin
```

```
            r := x.realpart * x.realpart + x.imagpart * x.imagpart;
            t.realpart := x.realpart / r;
            t.imagpart := - x.imagpart / r;
        end;

    procedure Squareroot(x: complex; var t: complex);
        var
            r: real;
            theta: real;
        begin
            r := sqrt(x.realpart * x.realpart + x.imagpart * x.imagpart);
            if x.realpart = 0.0 then begin
                if x.imagpart >= 0.0 then theta := 0.5 * pi;
                else theta := - 0.5 * pi;
            end else begin
                theta := arctan(x.imagpart / x.realpart);
                if x.realpart < 0.0 then theta := theta + pi;
            end;
            t.realpart := sqrt(r) * cos(0.5 * theta);
            t.imagpart := sqrt(r) * sin(0.5 * theta);
        end;
```

NOTE — In Extended Pascal , the LI datatypes complex and complex(radix, precision) can be mapped to the type complex, only if *rmax* and the given or default *radix* and *precision* parameters define a subset of the complex values which is distinguishable in the subset of the mathematical complex values defined by the Pascal implementation. All characterizing operations are conceptually preserved, although the implementation-defined arithmetic may affect the correctness of results.

### D.1.12 Void

The LI datatype void is mapped into Pascal only when it appears as an alternative of a choice datatype. In this case, it is mapped into an empty-variant "()" of a variant-record (see D.2.1).

## D.2 LI Generated Types

### D.2.1 Choice

A choice datatype of the form:

choice (*tag-type*) of (
*select-list1* : *alternative-1,*
*. . .*
*select-listN* : *alternative-N* )

is mapped into the Pascal variant-record type:

```
record case tag-variable : mapped-tag-type of
            case-constant-list1 : mapped-type1;
            . . .
            case-constant-listN : mapped-typeN
end;
```

only when the following conditions are met:

1) The *tag-type* maps to a Pascal ordinal type, as specified in this Annex. The *mapped-tag-type* is then the ordinal type which is the image of the mapping.

2) The *alternative-type* of each *alternative-i* can be mapped into a Pascal type, as specified in this Annex. If the *alternative-type* maps to a Pascal record-type, then the corresponding *mapped-type* is: ( all-fields-of-the-Pascal-record-type ). If the *alternative-type* is void, then the corresponding *mapped-type* is: (). If the *alternative-type* does not map to a Pascal record-type then the corresponding *mapped-type* is: (*mapped-field-identifier : mapped-alternative* ), where *mapped-alternative* is the image of the *alternative-type* under the mapping, and *mapped-field-identifier* is the *field-identifier* of *alternative-i*, if it is present and forms a valid Pascal field identifier, otherwise any identifier which does not conflict with any other field identifier in the Pascal record-type.

No other choice datatype can be mapped into Pascal.

The *tag-variable* is an invented identifier, used solely to implement the characterizing operations (see below), and is not otherwise required. Each *select-item* in the *select-list* which is a single value is mapped to the *case-constant* denoting the corresponding value of the *mapped-tag-type*. Each *select-item* in the *select-list* which is a *select-range* is mapped into a *case-constant-list* containing the denotations of all corresponding values of the *mapped-tag-type.* A *select-list* which is default is mapped into the *case-constant-list* containing the denotations of all corresponding values of the *mapped-tag-type*.

NOTE — In Extended Pascal, each *select-item* in the *select-list* which is a *select-range* is mapped into the analogous abbreviated-list form, and a *select-list* which is default is mapped into the *case-constant-list* otherwise.

All values of the choice datatype are mapped to the corresponding values of the *mapped-types* specified above.

The characterizing operations Tag and Cast are implemented (at least conceptually) in Pascal by referencing a particular field of the corresponding *mapped-type*, or assigning to it, respectively. The characterizing operation Discriminant is the value of the tag-variable. Equal can be implemented in Pascal by a case-statement using the tag-variable and the mapped *select-lists* given above to select field-by-field comparison for each alternative.

### D.2.2 Pointer

A pointer datatype of the form pointer to (element-type) is mapped into the Pascal type ^mapped-type, only when the *element-type* maps to a Pascal type, as specified in this Annex. The *mapped-type* is then the Pascal type which is the image of the mapping.

Only those values of the pointer datatype which refer to objects on the Pascal "heap" are mapped into the corresponding Pascal pointer-value. Other pointer-values may be supported by dereferencing them and copying the *element-value* onto the Pascal heap, thereby generating an "equivalent" Pascal pointer-value, in the sense that Dereference will work correctly, but the unspecified "assignment" operation (see Note 3 to clause 8.3.2) will not.

The Dereference operation is the Pascal *identified-variable*, i.e. pointer-value^.

### D.2.3 Procedure

A procedure datatype of the form: procedure (parameter-list)
is mapped into a Pascal "procedure parameter specification", only when it appears as the datatype of a procedure parameter, and only if all of its *parameter-types* can be mapped to Pascal types, as specified in this Annex.
A procedure datatype of the form: procedure (parameter-list) returns (return-parameter)
can be mapped into a Pascal "procedure parameter specification" or "function parameter specification", only when it appears as the datatype of a procedure parameter, and only if all of its *parameter-types*, including that of the *return-parameter*, can be mapped to Pascal types, as specified in this Annex. If the *return-parameter* maps to a simple type or a pointer type in Pascal, then the procedure datatype is mapped to a Pascal "function parameter specification"; otherwise, it is mapped to a "procedure parameter specification".

Every LI *parameter-declaration* of the form in identifier : parameter-type is mapped into a Pascal value-parameter-specification of the form identifier : mapped-type where *mapped-type* is the image of the *parameter-type* under the mapping into Pascal. Every LI *parameter-declaration* of the forms inout identifier : parameter-type or out identifier : parameter-type is mapped into a Pascal variable-parameter-specification of the form var identifier : mapped-type where *mapped-type* is the image of the *parameter-type* under the mapping into Pascal. If the procedure datatype is mapped to a functional parameter specification, the *parameter-type* of the *return-parameter* is mapped into the *result-type* of the Pascal function parameter-specification. If the procedure datatype has a *return-parameter* and is mapped to a procedure parameter specification, the *return-parameter* is mapped as if it were an additional out parameter.

Conceptually, every value of an LI procedure datatype which satisfies the above constraints could be defined as a Pascal procedure or function and could then appear as an actual parameter satisfying the corresponding formal parameter specification.

The Invoke operation is supported by the Pascal function-designator (call) within an expression or the Pascal procedure (call) statement, as appropriate to the form. Equal, in the sense defined for the LI datatype, is supported in Pascal by comparing all results of the invocations, to the extent that this is possible.

Terminations other than normal are not supported by Pascal, and no procedure datatype involving them can be mapped into Pascal.

### D.2.4 Record

A LI record datatype of the form: record (field-list) is mapped into a Pascal record-type of the form: record field-list end, only if all of its *field-types* can be mapped to Pascal types, as specified in this Annex. No other record datatype can be mapped into Pascal.

Every LI *field* of the form identifier : field-type is mapped into a Pascal field of the form identifier : mapped-type where *mapped-type* is the image of the *field-type* under the mapping into Pascal.

Every value of an LI record datatype which satisfies the above constraints is mapped to a value of the corresponding Pascal record-type by mapping the value of each field to its corresponding value, as specified in this Annex.

The FieldSelect operation is supported by the Pascal field-selection expression. The Aggregate operation is supported in Pascal by assignment of the given values to the appropriate fields of the record-variable. Equal is not directly supported by Pascal. It can be supported for each individual record-type by constructing a function which compares the corresponding field values.

### D.2.5 Set

A set datatype of the form set of (element-type) is mapped into the Pascal type set of mapped-type, only if the *element-type* maps to a Pascal ordinal-type, as specified in this Annex, and the cardinality of the ordinal-type does not exceed the implementation-defined maximum set cardinality required by Pascal. The *mapped-type* is then the Pascal ordinal-type which is the image of the mapping.

Every value of an LI set datatype which satisfies the above constraints is mapped to a value of the corresponding Pascal set-type by mapping the value of each member of the set-value to its corresponding value, as specified in this Annex.

All characterizing operations are supported by Pascal set operations.

No other set datatype can be mapped into Pascal directly. It is possible to map some other set datatypes as a variant of Sequence (see D.2.7), by defining the characterizing operations specifically for that structure.

### D.2.6 Bag

No bag datatype can be mapped into Pascal directly. Some bag datatypes can be mapped as a variant of Sequence (see D.2.7), by defining the characterizing operations on that structure.

### D.2.7 Sequence

A LI sequence datatype of the form sequence of (*element-type*) is mapped to the Pascal type: file of *mapped-type,* only if the *element-type* can be mapped to a Pascal type other than a file type, as specified in this Annex. No other sequence datatype can be mapped into Pascal directly.

Every value of a sequence datatype which satisfies the above constraints is mapped to a value of the corresponding Pascal file type by mapping the value of each element of the sequence-value to its corresponding value, as specified in this Annex.

With the declaration:

```
type sequenceof type = file of mapped-type;
```

the characterizing operations are supported by the required procedures for file types, as follows:

```
function IsEmpty(var s: sequenceof type): Boolean;
    begin IsEmpty := eof(s) end;

procedure Head(var s: sequenceof type; var t: mapped-type);
    begin reset(s);  read(s, t);  reset(s);  end;

procedure Tail(var s: sequenceof type; var t: sequenceof type);
    begin
        reset(s); rewrite(t);
        if not eof(s) then begin
            get(s);
            while not eof(s) do begin
```

```
                    t^ := s^; get(s); put(t);
                end;
            end;
            reset(s); reset(t);
        end;
    function Equal(var s, t: sequenceof type): Boolean;
        var continue: Boolean;
        begin
            reset(s);  reset(t); continue := true;
            while continue do begin
                continue := not (eof(s) or eof(t));
                if continue then begin
                    get(s);  get(t);
                    continue := mapped-typeEqual(s^, t^);
                    if not continue then Equal := false;
                end else
                    Equal := eof(s) and eof(t);
            end;
            reset(s); reset(t);
        end;
    procedure Empty(var s: sequenceof type);
        begin rewrite(s) end;
    procedure Append(var s: sequenceof type; t: mapped-type);
        begin write(s, t) end;
```

Because a Pascal *file-type*, however, cannot be the *component-type* of another *file-type*, LI datatypes of the form: sequence of (sequence (...)) or sequence of (record(...)), where the record datatype contains a sequence datatype, cannot be mapped into Pascal. Moreover, when the *component-type* of a *file-type* is, or contains, a *pointer-type*, there may be implementation-dependent limitations which defeat the purpose of the mapping.

NOTE — Values of a sequence datatype of the form sequence of (element-type), where the *element-type* maps to some Pascal type *mapped-type*, as specified in this Annex, can also be mapped into Pascal using the type:

```
type    sequenceofT = ^sequenceofTmember;
        sequenceofTmember = record
            next: sequenceofT;
            elementvalue: mapped-type
        end;
```

Each member (value of *element-type*) of a value of the sequence datatype is mapped to a heap variable of the Pascal type sequenceofTmember, by mapping its value to the corresponding value of *mapped-type*, as specified in this Annex, and placing that value in the field elementvalue. The value of the sequence datatype is then represented by a value of the type sequenceofT, which is the pointer to the heap variable representing the first member, or nil if the sequence is empty. The next field of the first member is set to point to the heap variable representing the second member, etc. The next field of the last member is set to nil. All characterizing operations can be defined on this representation.

### D.2.8 Array

An array datatype of the form array (index-list) of (element-type) is mapped into the Pascal type array [mapped-index-list] of mapped-element-type, only if the following conditions hold:

1) The *element-type* maps to some Pascal type *mapped-element-type*, as specified in this Annex.

2) Each *index-type* in the *index-list* can be mapped into some Pascal ordinal-type *mapped-index-type*, as specified in this Annex. The *mapped-index-list* is then the list of the *mapped-index-types*, in corresponding order.

No other array datatype can be mapped into Pascal.

Every value of an LI array datatype which satisfies the above constraints is mapped to a value of the corresponding Pascal array-type by mapping the value of each element of the array-value to its corresponding value, as specified in this Annex.

The Select operation is supported by Pascal indexing. The Replace operation is supported by assignment to the appropriate cell of an array variable. Equal is not directly supported by Pascal. It can be supported for each individual array-type by constructing a function which compares the corresponding array element values.

### D.2.9 Table

No table datatype can be mapped into a Pascal datatype directly.

Values of a table datatype of the form table (field-list), where each *field-type* in the *field-list* maps to some Pascal type *mapped-field-type*, as specified in this Annex, can be mapped into Pascal using the type:

```
type tableentry = record
      field1: mapped-field-type-1;
      . . .
      fieldN: mapped-field-type-N
end;
```

Each tableentry value is a Pascal record-value having the corresponding field values assigned to the fields field1, ..., fieldN. The value of the table datatype is then represented as a value of the Pascal type file of tableentry, in the same way as a sequence datatype (see D.2.7). The characterizing operations for the table datatype can be defined on that structure.

## D.3 LI Subtypes

### D.3.1 Range

LI range-subtypes map into Pascal subrange types, but only if the base type maps into a Pascal ordinal-type, as specified in this Annex.

### D.3.2 Selecting

LI selecting-subtypes do not have equivalents in Pascal. A selecting-subtype of a state type or an enumerated type is mapped as if it were the base type.

### D.3.3 Excluding

LI excluding-subtypes do not have equivalents in Pascal. An excluding-subtype of a state type or an enumerated type is mapped as if it were the base type.

### D.3.4 Size

LI size-subtypes do not map into native Pascal concepts. Size-subtypes could be supported by the sequence datatype implementation in D.2.7, and certain size-subtypes are mapped to specific Pascal types in D.4.

### D.3.5 Explicit subtypes

LI explicit-subtypes do not have equivalents in Pascal. An explicit-subtype is mapped as if it were the base type.

### D.3.6 Extended

LI extended-types cannot be mapped into Pascal, in general. In the case of enumerated datatypes, definition of an entirely new type with value isomorphisms based on ordinal position may be possible.

## D.4 LI Defined Datatypes

### D.4.1 Natural number

The LI datatype naturalnumber range(0..maxint) maps to the Pascal subrange type 0..maxint, according to the mapping for its *type-definition*. No naturalnumber value greater than maxint can be mapped, and no datatype containing such a value can be mapped into Pascal.

### D.4.2 Modulo

The LI datatype modulo(*modulus*) maps to the Pascal subrange type 0..*modulus–1*, according to the mapping for its *type-def-*

*inition*, but only if *modulus-1* is less than or equal to the implementation-defined value maxint. The characterizing operations can be derived from those of Pascal type integer (i.e. those of the subrange type) analogously to the derivation in clause 10.1.2.

No other modulo datatype can be mapped into Pascal.

### D.4.3 Bit

The bit datatype maps to the Pascal type declared by

```
type bit = 0..1;
```

0 and 1 map to the corresponding integer values. All characterizing operations are preserved, although the Add operation must be defined as:

```
function Add(x,y: bit): bit;
    begin
        if (x = y) then Add := 0 else Add := 1;
    end;
```

### D.4.4 Bit string

A bitstring datatype all of whose values are of a fixed constant length, i.e. bitstring size(k), is mapped into the Pascal type packed array [1..k] of Boolean.

NOTE — While bitstring can just as well be mapped into packed array of bit, packed array of Boolean is often much more efficiently implemented.

With the definitions:

```
type bitstringsizek = packed array [1..k] of Boolean;
    bitstringsizek1 = packed array [1.. (k-1)] of Boolean;
```

the characterizing operations Equal, Head and Tail are defined as follows:

```
function Equal(x,y: bitstringsizek): Boolean;
    var i: integer;
    begin
        Equal := true;
        for i := 1 to k do Equal := Equal and (x[i] = y[i]);
    end;

function Head(x : bitstringsizek): bit;
    begin
        if x[1] then Head := 1
        else Head := 0
    end;

procedure Tail(x : bitstringsizek, var y: bitstringsizek1);
    var i: integer;
    begin
        for i := 1 to k-1 do y[i] := x[i+1];
    end;
```

Append, Empty, IsEmpty are not meaningful operations on a bit-string of fixed size.

The bitstring datatype can be mapped according to its *type-definition*, that is, sequence of (bit) (see D.2.7), although more efficient structures for bitstring can be developed.

### D.4.5 Character string

A characterstring datatype whose underlying character datatype can be mapped to Pascal (see D.1.4) and all of whose values are of a fixed constant length, i.e. characterstring size(k), is mapped into the Pascal type packed array [1..k] of char.

With the definitions:

```
type charstringsizek = packed array [1..k] of char;
     charstringsizek1 = packed array [1.. (k-1)] of char;
```

the characterizing operations Head and Tail are defined as follows:

```
function Head(x : charstringsizek): char;
     begin Head := x[1] end;

procedure Tail(x : charstringsizek; var y: charstringsizek1);
     var i: integer;
     begin
          for i := 1 to k-1 do y[i] := x[i+1];
     end;
```

Equal is Pascal "=". Append, Empty, IsEmpty are not meaningful operations on a character-string of fixed size.

A characterstring datatype whose underlying character datatype can be mapped to Pascal (see D.1.4) can be mapped according to its *type-definition*, that is, sequence of (character) (see D.2.7), although more efficient structures for characterstring types can be developed.

### D.4.6 Time interval

Time interval datatypes are mapped according to their *type-definitions*, that is, as specified for scaled datatypes (see D.1.9). The scalarMultiply operation is mapped to

```
function scalarMultiply(x: scaled, y: timeinterval): timeinterval;
```

and the body is exactly the same as for the scaledMultiply operation defined in D.1.9, with the substitution of timeinterval for the type of the temporary result t.

### D.4.7 Octet

The octet datatype is mapped into the Pascal type:

```
type octet = 0..255;
```

All characterizing operations are preserved.

### D.4.8 Octetstring

An octetstring datatype all of whose values are of a fixed constant length, i.e. octetstring size(k), is mapped into the Pascal type packed array [1..k] of octet, where octet is defined as in D.4.7.

With the definitions:

```
type octetstringsizek = packed array [1..k] of octet;
     octetstringsizek1 = packed array [1.. (k-1)] of octet;
```

the characterizing operations Equal, Head and Tail are defined as follows:

```
function Equal(x,y: octetstringsizek): Boolean;
     var i: integer;
     begin
          Equal := true;
          for i := 1 to k do Equal := Equal and (x[i] = y[i]);
     end;

function Head(x : octetstringsizek): octet;
     begin Head := x[1]  end;

procedure Tail(x : octetstringsizek, var y: octetstringsizek1);
     var i: integer;
     begin
          for i := 1 to k-1 do y[i] := x[i+1];
     end;
```

Append, Empty, IsEmpty are not meaningful operations on an octetstring of fixed size.

The octetstring datatype can be mapped according to its *type-definition*, that is, sequence of (octet) (see D.2.7), although more efficient structures for octetstring can be developed.

### D.4.9 Private

Private is defined in Pascal essentially as it is in 10.1.9:

```
type private = packed array [1..size] of bit;
```

or:

```
type private = packed array [1..size] of Boolean;
```

In many cases, only the latter will produce the desired (contiguous bitstring) implementation, although neither is in fact required to do so.

### D.4.10 Object identifier

The objectidentifier datatype can be mapped into Pascal according to its *type-definition*, that is, sequence of (objectidentifiercomponent) (see D.2.7), where objectidentifiercomponent is mapped to the Pascal type:

```
type objectidentifiercomponent = 0..maxint;
```

In many cases, however, the component values of an objectidentifier value are not useful to the application, and it may be more useful to map the objectidentifier type into an octetstring type (see D.4.8).

## D.5 Defined Generators

### D.5.1 Stack

No stack datatype can be mapped into Pascal directly. Individual stack datatypes can be mapped into a linked structure similar to the one suggested for sequence (see the Note to D.2.7), by defining the characterizing operations on that structure.

### D.5.2 Tree

No tree datatype can be mapped into Pascal directly. Individual tree datatypes can be mapped by a linked structure similar to the one suggested for sequence (see the Note to D.2.7), but there are many possible implementation choices, depending on the intended searching strategies, i.e. the true "characterizing operations" of the type.

### D.5.3 Cyclic enumerated

LI datatypes of the form cyclic of (T) are mapped into Pascal as provided for the type T in D.1.3, because T is required to be an enumerated datatype. The chaacterizing operation Successor does not map to Pascal succ(); it must be defined as specified in 10.2.3.

### D.5.4 Optional

An LI datatype of the form optional(T) can only be mapped to Pascal if the type T can be mapped to Pascal, as specified in this Annex. The datatype optional(T) is mapped to Pascal as:

```
record case present: Boolean of
    true: (valuegiven: mappedT);
    false: ()
end;
```

where *mappedT* is the mapping of LI datatype T into Pascal. The characterizing operation IsPresent is defined by:

```
function IsPresent(t: optionalT): Boolean;
    begin IsPresent := t.present  end;
```

Unary characterizing operations on type T of the form Op(t: optional(T)):T are supported by a Pascal procedure of the form:

```
procedure op(t: optionalT, var result: mappedT);
    begin
        if IsPresent(t) then result := mappedTOp(t.valuegiven);
```

```
    end;
```

And binary operations are similarly supported.

NOTE — Alternatively, optional(T) can be mapped to ^mappedT, where mappedT is the mapping of LI datatype T into Pascal, and the object of type mappedT, when present, is allocated on the heap.

The characterizing operation IsPresent is defined by:

```
function IsPresent(t: optionalT): Boolean;
    begin IsPresent := t <> nil  end;
```

Unary characterizing operations on type T of the form Op(t: optional(T)):T are supported by a Pascal procedure of the form:

```
procedure op(t: optionalT, var result: mappedT);
    begin
        if IsPresent(t) then result := mappedTOp(t^);
    end;
```

And binary operations are similarly supported.

## D.6 Type-Declarations

In Pascal two type-specifiers refer to the same datatype only if they are both identifiers and spelled identically. Type-specifiers which are not identifiers always refer to distinct datatypes. Because of this, additional datatype definitions may be needed in a mapping Pascal to correctly support the identity of LI datatypes which do not have names.

### D.6.1 Renaming declarations

This concept is supported in Pascal only for named datatypes. That is, if a Pascal type y is denoted by an identifier, then a Pascal type definition of the form:

```
type x = y;
```

is a renaming declaration, equivalent to the LI *type-declaration*:

```
type x = y;
```

But if the Pascal type y is a syntactic designation other than an identifier, the Pascal type declaration of the form:

```
type x = y;
```

is effectively a “new” datatype declaration in all cases.

### D.6.2 Datatype declarations

An LI datatype declaration which declares a single datatype (no parameters) can be mapped to Pascal as a Pascal type-declaration in which the LI *type-definition* is mapped into Pascal, as specified in this Annex. If the *type-definition* does not have a mapping, then the datatype so declared cannot be mapped into Pascal.

An LI datatype declaration which declares a family of datatypes, using one or more parameters, cannot, in general, be mapped into Pascal. In many cases, however, each member of the family which is to be used in a given context can be mapped into a distinct Pascal type, by inventing a unique name and mapping the *type-definition* after making lexical substitutions for the parameter values.

### D.6.3 Generator declarations

An LI generator declaration cannot, in general, be mapped into Pascal. In many cases, however, each resulting datatype which is to be used in a given context can be mapped into a distinct Pascal type, by inventing a unique name and mapping the *type-definition* after making lexical substitutions for the parameter values.

NOTE — In Extended Pascal, many generators can be mapped to schemata.

# Annex E
(informative)

# Example Mapping to MUMPS

This annex contains a draft "inward" mapping from the LI datatypes into the programming language MUMPS, as defined by ISO/IEC 11756:1992, *Information technology — Programming languages — MUMPS.*

NOTE — It is anticipated that the prospective revision of ISO 11756:1992 will use the name M, instead of MUMPS, as the primary name for the language.

The purpose of this annex is to exemplify a mapping to a language whose concept of datatype is significantly different from that of strongly typed programming languages. This mapping should not be considered a definitive mapping from LI datatypes to the MUMPS language.

This annex specifies a mapping from *values* of LI datatypes into MUMPS values. In all cases, the MUMPS data being mapped to is a string and the mapping expresses the form of the resulting string values.

For inward-mappings, the values produced are in a canonic form, as defined in ISO 11756:1992, unless otherwise stated. An inward-mapping that produces values exceeding the Portability limits defined in section 2 of ISO 11756:1992 is non-portable. When the result of mapping a value as herein specified would exceed the implementation limits, the result is unspecified.

For the reverse-inward-mappings any necessary coercion from the internal format takes place. Unless otherwise stated the reverse-inward-mapping is the inverse of the inward-mapping, using the necessary coercions. If the reverse-inward-mapping would result in values which are not within the range of the LI datatype, the result is unspecified. For example, a state-value might be produced from a string which is not one of the permissible state values.

When mapping to or from a numeric format is required, the accuracy of the the conversion is the responsibility of the implementation.

A further assumption of this binding is that it is an operational one, i.e. that the conversions are handled at run-time with the implementation mapping the interface specification in an automated fashion.

NOTE — An alternative approach would be to extend or "annotate" (see 7.4) the interface specification language — the Common Interface Definition Notation (IDN) — to include mapping specifications, and then generate a mapping module which would handle the specific interface essentially external to the process.

In this specification, the MUMPS operation sequences that implement the characterizing operations on the LI datatypes are not explicitly specified. Except as noted, all characterizing operations are supported on the resulting MUMPS values. Many of these operations are provided as part of the MUMPS language; others can be implemented as additional extrinsic functions, if required.

Use of the in-built MUMPS operations, such as addition, on data which is mapped to or from certain LI datatypes may cause these values to be interpreted in ways other than specified in LI characterizing operations. Therefore the use of these within a MUMPS program for manipulation, as opposed to transfer operations, requires the programmer to perform the appropriate conversions. The LI datatypes involved are Date and Time, Rational, Scaled, Complex and all Generated and Defined Types.

## E.1 LI Primitive Datatypes

### E.1.1 Boolean

This maps to truth-value, true maps to 1 and false to 0.

### E.1.2 State

Each state-value is mapped to its string value.

### E.1.3 Enumerated

Each enumeration value maps to its index in the LI enumerated-type definition, i.e. the first value maps to 1, the second to 2 etc.

### E.1.4 Character

A character datatype maps to a MUMPS Character Set Profile definition, which has an associated encoding for the characters.

### E.1.5 Ordinal

Each ordinal value maps to the corresponding positive integer value.

### E.1.6 Date and Time

Date and time type values are mapped to the character string representation defined in ISO 8601:1988.

NOTE — An alternative is to map date and time values to a character string in $H[OROLOG] format, which has the form

D,S

where D is the numbers of days since December 31, 1840, and S is the number of seconds since midnight.
Since there are no intrinsic operations available on this format, this alternative may not be of greater value.

### E.1.7 Integer

Each value maps to its canonic form.

### E.1.8 Rational

Each value maps to the character representation of the corresponding rational-literal.

NOTE — An alternative if the denominator is greater than 0 is to map the value to numerator-value/denominator-value, i.e. the number created by *performing* the division of the two parts. This would allow normal arithmetic operations, but at a loss of precision. (See the note in D.1.9.)

### E.1.9 Scaled

Each value maps to the character representation of the corresponding scaled-literal.

NOTE — A scaled value could also be converted to a numeric value, as for Rational.

### E.1.10 Real

Real values are mapped to the nearest numeric values.

### E.1.11 Complex

Values are mapped to strings of the form

real-value%imaginary-value

where

real-value is the numeric value of the real-part of the corresponding complex-literal, and
imaginary-value is the numeric value of the imaginary-part of the corresponding complex-literal.

### E.1.12 Void

There is no mapping for this datatype, since it only appears as a formal part of an interface specification and has no values, i.e. does not represent data actually transferred across an interface.

## E.2 LI Generated Types

### E.2.1 Choice

A value of a LI Choice datatype is mapped according to the specification for the type actually instantiated. In MUMPS only a variable can actually have the behavior of a Choice datatype. The discriminant of the Choice is provided in V(0), where V is the variable name of the associated MUMPS variable.

### E.2.2 Pointer

A Pointer maps to a MUMPS variable. Access to the element value – the data pointed to – is provided by use of indirection or some implementation-specific mechanism. That is, indirection (@) is the MUMPS support for the characterizing operation Dereference.

### E.2.3 Procedure

A Procedure value maps to a label and formallist of a formalline, which defines a subroutine call. Termination parameters are mapped to additional formallist names. Inout and out parameters are mapped (at run-time) to parameters called by reference.

NOTE — The exact mechanism of the call may be subject to restrictions, such as those specified in ISO/IEC 13886:1995, *Information technology — Programming languages — Language-independent procedure calling*.

### E.2.4 Record

A Record value maps to a MUMPS array in which the subscripts are the field-identifiers, and the data is the mapping of the value of the corresponding field of the record value.

NOTES

1. If the LI value were represented in one of the record-value forms, the data would be the mapping of the independent-value. In the value-list form, the subscript is the field-identifier corresponding to this position in the record type specification.

2. A record value could also be modelled with subscripts being the field position numbers, but the Notes to clause 8.4.1 indicate that the field identifier is significant while the position is not.

### E.2.5 Set

A Set maps to a MUMPS array with the subscripts being an integer, starting at 1, denoting the position of the independent-value in the value-list.

### E.2.6 Bag

A Bag maps in exactly the same way as Set.

### E.2.7 Sequence

A Sequence maps in exactly the same way as Set.

### E.2.8 Array

An Array maps to a MUMPS array with the first level subscript being the first independent-value in the value-list, the second level subscript being the second independent-value etc.

### E.2.9 Table

A Table maps to a MUMPS array with the first level subscript being an integer, starting at 1, denoting the position of the table-entry within the table-value, the second level subscript being the field identifier associated with the independent-value. An empty value is denoted by no data.

## E.3 LI Subtypes

In general all the subtypes are treated exactly as if they were the base type.

Extended types can be mapped, provided that the values are within the permissible range.

## E.4 LI Defined Datatypes

### E.4.1 Natural number

Values of Naturalnumber are mapped as values of the base type – integer (see E.1.7).

### E.4.2 Modulo

Values of Modulo types are mapped as values of the base type – integer (see E.1.7).

### E.4.3 Bit

Bit maps to the values 0 and 1.

### E.4.4 Bit string

Bitstring maps to a string of 0s and 1s.

NOTE — This mapping may have smaller length limitations than expected because it is dependent on the maximum length of strings. (The portability minimum limit for this in ISO 11756:1992 is 255, that for the proposed revision is 510. Many implementations have larger limits.) Other possibilities are mapping to an array of Bit values or mapping to a character string whose values are made of (say) eight bit values.

### E.4.5 Character string

Characterstring maps to a MUMPS string.

### E.4.6 Time interval

Values of Time interval types are mapped as values of the base type – scaled (see E.1.9).

### E.4.7 Octet

An Octet value *x* maps to the character value $CHARACTER(*x*).

### E.4.8 Octet string

Octetstring maps to a string whose individual characters are the mappings of the equivalent Octet values.

### E.4.9 Private

Private maps to an array of strings with numeric subscripts indicating the order of data within the array.

### E.4.10 Object identifier

Objectidentifier maps into a string, with the value being the characters of the objectidentifier-value.

## E.5 Type-Declarations and Defined Datatypes

Since MUMPS has no declaration facilities the implementation of these facilities is the responsibility of the interface specification interpretation process.

# Annex F
(informative)

# Resolved Issues

This annex contains a brief discussion of technical problems encountered in the development of this International Standard and the consensus resolution thereof by the technical committee.

## F.1 Scope

**Issue 1.** Should LI Datatypes be a reference model only?

Consensus is that LI Datatypes has characteristics of a reference model, but its scope goes beyond that. An entity claiming to use this International Standard as a “reference model” is said to *comply indirectly*, but indirect compliance places requirements on the entity for formal statements of the relationships (mappings). These requirements are necessary to meet the original intent of the standard. Because of the formal syntax for the identification and definition of datatypes, *direct compliance* is also possible. Direct compliance is needed so that products such as cross-language or cross-entity utilities can reference, use, and claim conformity to, LI Datatypes, especially where no other relevant standards exist. In addition, the possibility of direct compliance may encourage future software products, including new kinds of products, to use standard LI datatypes directly rather than defining their own syntax and semantics and then performing the mapping.

**Issue 2.** What datatypes should be included in the standard?

Consensus is that the standard should include all of the datatypes needed to support ISO programming languages and the expected needs of interface specifications. If any language finds the need to distinguish two "possibly equivalent" datatypes or constructors, then the standard should distinguish them; and if it is necessary to insure that datatypes of two different languages could be mapped into different LI datatypes, then the standard should distinguish them; otherwise the standard should not.

**Issue 3.** Should the standard specify a minimal collection of common datatypes or a rich collection?

A primary purpose of the standard is to specify datatypes for various forms of interchange and interface. A rich collection of datatypes encourages interface definitions to use datatypes which may be difficult to map to many programming languages. This suggests that the set of “common” datatypes should be restricted to those that are readily mapped to most programming languages. On the other hand, a rich collection of datatypes encourages the user to specify the datatype he *means*, which may be both clearer and more efficiently mapped than some work-around based on a small set of “common” datatypes.

The consensus is that the standard should provide a rich collection of conceptually distinct datatypes. As Annex E demonstrates, most of the LI datatypes *can* be mapped to most programming languages, and the work-arounds for particular languages become a part of the language-specific mapping rather than a part of the interface specification. For example, Sequence is a native datatype in LISP, and Set is a native datatype in Pascal. Both are common in conceptual interface specifications, but they require work-arounds to be mapped to C or Fortran. The user should not be forced to characterize a Sequence as a fixed-length Array (which it is not) just to accommodate the limited type vocabulary of a programming language which may not even be relevant to the application.

For various reasons, specific applications (and the related standards, if any) may find it useful to constrain the set of LI datatypes allowed/supported in that application (see Issue 6). Whether a language mapping should provide for all datatypes in this International Standard is an unresolved issue, but out of the scope of this International Standard in any case.

**Issue 4.** Are representation concerns appropriate in the standard?

The scope of the project expressly stated that representation is *not* a part of the standard. A number of representation concerns, such as the characterization of Real as floating-point and the ordering of fields in a Record, clearly need to be addressed by any use of the LI datatypes in defining “neutral representations”. Moreover, the datatypes of programming languages often have representation properties which are important in distinguishing "internal datatypes" and are therefore necessary for mappings. Representation attributes, on the other hand, are only a fraction of the datatype annotation capabilities needed by procedure calling standards and applications. Consensus is that a common mechanism for such annotation is necessary (and provided in clause 7.4), but particular annotations should not be a normative part of this International Standard.

**Issue 5.** What is the relationship between this International Standard and ISO 13886 Language-Independent Procedure Calling?

ISO/IEC 13886:1995, *Information technology — Programming languages — Language-independent procedure calling*, provides the procedure call model, the requirements for interface specifications and the syntax of the Interface Definition Notation (IDN), and the requirements for LI procedure calling service implementations. ISO 13886 makes normative reference to this International Standard (ISO 11404) to define all the datatype-related aspects of the IDN. ISO 13886 defines in detail the dynamic notions associated with the Procedure and Pointer datatypes as they relate to the procedure calling model.

It was originally expected that ISO 13886 would provide the IDN syntax and this International Standard would provide only the fundamental definitions of datatypes. But the complexities of defining datatypes made it necessary for much of the IDN to be introduced into this International Standard. Thus, the overlap between the two standards is the common IDN.

## F.2 Conformance

**Issue 6.** Should support of certain datatypes be required of complying entities?

The nature of the standard should not be such as to *require* the support of any datatype. Rather other standards which incorporate the LI Datatypes, such as LI Procedure Calling and Remote Procedure Call, should specify what datatypes are required for the purposes of those standards.

**Issue 7.** Should implementations be required to support the characterizing operations?

The purpose of considering operations in this International Standard is solely to distinguish semantically distinct datatypes which have common or similar value spaces. Moreover, where several choices were available, the choices of characterizing operations included in the standard are arbitrary. Consequently, mappings between language datatypes and LI datatypes should not necessarily imply express support for the characterizing operations appearing in the standard. However, an internal datatype should never be mapped into a LI datatype having characterizing operations which the internal datatype *could not* support. Such a mapping violates the notion of semantic equivalence of the datatypes.

## F.6 Fundamental Notions

**Issue 8.** Should the LI datatypes provide axiomatic datatype definitions?

Much of the axiomatic definition work would be replication of well-known mathematical work. There is consensus that mathematical datatypes should be defined by appeal to standard mathematical references. There is also consensus that most "axiomatic definition" of other datatypes is nothing more than mathematical statement of closure under what is herein called "characterizing operations".

### F.6.6 Characterizing operations

**Issue 9.** Is InOrder necessary? Does the standard need to define an ordering operation?

Order is an important property of a datatype, and when the value space has multiple possible order relationships, the choice of a particular order relationship is what makes the datatype ordered. When a datatype has a universally accepted order relationship, it is appropriate to require that order in the standard. When there is no such order relationship, or when everyone disagrees on the order relationship, then not necessarily will a given implementation of the datatype support any order relationship given, and the LI datatype should not be defined to be ordered.

**Issue 10.** How many characterizing operations are enough?

There is consensus that the characterizing operations on any datatype should be limited to those which are necessary to distinguish the datatype from types with similar value spaces. It was later determined to be useful to include operations which, though redundant with respect to distinguishing the datatype, would be used in the definitions of characterizing operations on other datatypes, e.g. Boolean And and Or.

**Issue 11.** Are conversion operations between datatypes characterizing?

"Conversion operations", that is, operations which map one datatype into another, are of several kinds, each of which needs to be considered differently:

a) Operations which are part of the mathematical derivation of primitive datatypes are generally "characterizing". Specifically, the Promote operation, which maps Integer into Rational and Rational to Real, etc., is part of the mathematical characterization of the numeric datatypes.
b) Other operations which map one *primitive* datatype into another are clearly not "characterizing", if the datatype is well-defined. Specifically, the Pascal ORD operation on enumerated types is not characterizing - it has nothing to do with

the meaning of the enumerated datatype itself. Similarly, Floor, which maps Real to Integer, is useful but not characterizing for either the Real or Integer datatypes.

c) Operations which create a value of a generated type from values of the component datatypes may be characterizing for the generator. Thus Setof is characterizing for the Set generator, and Replace is characterizing for the Array generator.

d) Operations which project a value of a generated type onto any of its component datatypes may be characterizing for the generator. Thus Select (subscripting) is characterizing for Array and Dereference is characterizing for Pointer.

e) All characterizing operations on datatype generators must be one of the above, but not necessarily are all such operations characterizing. It suffices to define any set of such operations which unambiguously identifies the datatype generator.

**Issue 12.** Should characterizing operations identify exception conditions?

Consensus is no. Exceptions result from the performance of operations on datatype values, or from attempts to move or convert a value from one environment to another. Specifications for operations, exchanges and conversions are out of the scope of this International Standard, as stated in clauses 1 and 6.1. They are addressed by related standards.

## F.7 Elements of the Datatype Specification Language

**Issue 13.** Should the LI datatypes have a concrete syntax?

To allow the standard to be used to specify datatypes unambiguously, it must have a syntax, with specific production rules for each of the datatypes and generators. Moreover, this syntax must permit datatype definitions to be recursive or contain forward references, in order to permit definition of datatypes such as Tree, or the LISP-characteristic indefinite-list datatype.

The syntax chosen is a subset of the “common” Interface Definition Notation (see Issue 5).

## F.8 Datatypes

**Issue 14.** Should datatypes with “units” be included in the standard?

The concept of datatypes which express values in particular units is considered important to interface definitions, but the collection of values which might be appropriate for the “units” is open-ended and very application-dependent. For this reason, there is consensus that this version of LI Datatypes should not standardize such datatypes. There is one exception to this: Time units are standardized and supported by a number of programming languages. Therefore, Date-and-Time and TimeInterval *are* included in this version.

**Issue 15.** Should some of the datatypes in Clause 8 be in Clause 10 (derived)?

The question of whether Enumerated can be “derived from” State, or Ordinal from Integer, etc., depends on the particular taxonomy of datatypes which is chosen. Other taxonomies of datatypes are possible which might entail such changes. No claim is made that the taxonomy in Clause 8 is the best available, but it is viable, and changing taxonomies would not bring about substantive improvements in the specification. What is important is that datatypes that are similar but can be distinguished are distinguished.

### F.8.1.4 Character

**Issue 16.** Should Character types be ordered?

The problem is that the accepted ordering of characters in a standard character-set by ascending value of their integer codes is a machine-oriented view of the datatype. The “dictionary” order for the character-set may vary from nation to nation or from application to application. Thus, although everyone agrees that these datatypes are conceptually ordered, there is no agreement on what the order relationship is. Therefore, no standard InOrder function can be defined, and for that reason these types are said to be unordered. (See Issue 9.)

### F.8.1.8 Rational

**Issue 17.** Can the cardinality of the Rational datatype be supported by any language or implementation?

It is possible for a mapping of Rational to fully support the datatype, as defined in 6.3.4, if the language supports unbounded integers.

For a language/implementation which does not support unbounded integers, however, no mapping of the Rational datatype can

satisfy the requirements of clause 11.4.4.

### F.8.1.9 Scaled

**Issue 18.** How is Scaled distinct from Real? Is Scaled an implementation?

Scaled is a mathematically tractable datatype which has a number of properties which tend to be associated with representation, such as rounding. Scaled is not merely a subtype of Real, nor a poorer representation of Real values than floating-point. (In fact, Scaled is properly represented by integral values and not, in general, by floating-point.) It is the datatype of objects which are *exact* to some number of (radix) places. Scaled, with these semantics, is the most frequently occurring datatype in COBOL programs, and also appears in other standard languages, such as PL/I. Parameters radix and factor are provided for consistency with the usage in programming languages. Only a single parameter, giving the common denominator of the datatype, is semantically necessary. Since both base-two and base-ten scaling are in common usage, generalizing to an arbitrary radix seems to be appropriate. Mappings and implementations will limit this.

**Issue 19.** Is it necessary to support radices of Scaled datatypes other than 2 and 10?

Many applications use conceptually Scaled datatypes with unusual radices, notably 60 and 360, although they are represented in programs by an Integer with the scale-factor hidden in the semantic units. There is no reason not to make such datatypes expressible as LI datatypes, although there may be strong constraints on the mappings to programming languages.

### F.8.1.10 Real

**Issue 20.** What is the computational notion of datatypes Real and Complex?

The LI Datatypes Real and Complex cannot usefully be the mathematical datatypes. The computational notion of these types, regardless of representation mechanism, is one of “approximate” values. The model used is the “scientific number”, which was a widely accepted computational model in the physical sciences before the advent of computers. It is conceptually similar to the “floating point” model, but the standard floating-point models (IEC 559) are too closely tied to representation concerns.

### F.8.1.12 Void

**Issue 21.** Is Void a value of multiple types, as in SQL2 Null, or a datatype itself?

Void, or nil or null, is not a value of every type (or of many types). It has none of the properties of any datatype to which it might be assigned. Every value of type Integer, for example, can be compared with zero. Is nil < 0? Is nil = 0? Allowing such a comparison is clearly inappropriate. Nil must therefore be a value distinct from those of any other primitive type. The SQL2 null-valued column is properly described in LI datatypes as a choice datatype one of whose alternatives is the true datatype of the column and and the other is some state datatype representing the "null values". And in general, objects which “could be null”, are better modelled as having choice datatypes. “Void” was originally called “Null”, but has been renamed to avoid confusion with “null values” in SQL.

**Issue 22.** Is Undefined the same as Void?

There is consensus that Undefined is *not* a datatype. Undefined is a part of the behaviour of entities which have the concept datatype, but it is distinct from the datatype of the entity. Its meaning arises from the nature of the entity and its usage. In general, “undefined” models the case in which a value of some datatype is appropriate, but not available. Some processing entities, e.g. SQL, have more than one “undefined” value, in order to model different “situations” in which no value is available. Void, on the other hand, models the empty variant in Pascal and Ada and the Null type in ASN.1 and other places where an element datatype, or value, is syntactically or semantically required to complete a complex datatype, or value, but no (other) datatype or value is appropriate. The Void datatype should not be confused with “undefined values” in various languages, which do not have these semantics.

### F.8.2.2 Selecting

**Issue 23.** Should the base type of Selecting and Excluding be restricted to exact datatypes?

Exactness is required to ensure independence of implementation. Any implementation of an exact datatype must be able to distinguish exactly the conceptual values. This requirement does not exist for approximate datatypes — it is permissible in representing approximate datatypes to have more than the conceptual values and to be unable to distinguish values which are sufficiently close. If this is permitted for "Selecting" and “Excluding” subtypes, the same LI datatype as implemented by two machines might actually have non-isomorphic value spaces.

### F.8.3.2 Pointer

**Issue 24.** Is Pointer a conceptual datatype or solely an implementation mechanism?

Pointer is the name of an implementation mechanism, but it has a conceptual foundation. Pointer is the datatype form of the concept *relationship* in conceptual models, specifically of relationships between otherwise independent data objects which may possess multiple such relationships. Objects of pointer datatype represent single-ended relationships, i.e. from (any) to (object of element type), in which the usage of the pointer determines the other object (any) in the relationship. In this regard, pointer may be considered to be similar to the database concept *key*, which also conveys a single-ended relationship to the object which the key identifies. The related concept *handle*, meaning a manipulable representative for an otherwise inaccessible object, does not appear to be quite the same, since the notion of accessing the data object to which the handle refers is intentionally not supported, while accessing the object to which a pointer refers is a characterizing operation of Pointer.

**Issue 25.** Is Pointer a primitive datatype or an aggregate datatype?

There is consensus that Pointer is a primitive datatype in that its values are objects with the property that values of another datatype can be associated to them. These objects are not “constructed from” values of the associated datatype; rather they are distinct primitive objects drawn from a conceptually large state-value space by the process of association. This notion is similar to the mapping notion of Arrays, but unlike these explicit mappings, the values in the domain – the pointer value-space – have no other semantics.

**Issue 26.** Must there be a characterizing operation which produces values of type Pointer to (T)?

After much debate on the merits of the Allocate and Associate operations, there is consensus that no single "constructor" for datatype pointer is truly characterizing, in the sense that any implementation of the datatype Pointer would necessarily be able to support it.

**Issue 27.** Must there be a null value of every datatype Pointer to (T)?

It is acknowledged that “null” is not a useful value of a pointer datatype – the sole characterizing operation Dereference does not apply to “null”. Therefore it is possible to define “pointer” to mean “pure” pointer datatypes that do not have “null” values, and to model the commonly occurring pointer datatypes as:

choice (boolean) of ((true): pointer to x, (false): void).

On the other hand, most programming languages which support pointer datatypes support null values of such datatypes. Consensus is to make “null” a value of the LI datatype pointer to (T) for consistency with most applications. “Pure” pointer datatypes can be modelled as: pointer to (T) excluding (null).

### F.8.4.1 Record

**Issue 28.** Is the ordering of fields in a Record significant?

Conceptually, a record is a collection of related information units which are accessible by name rather than by position. Therefore, the ordering of fields in a Record is not a property of the conceptual datatype itself. Order is, however, an important consideration in mappings and representations of the datatype.

### F.8.4.2 Set

**Issue 29.** Should the element type of a Set be required to be finite?

At the conceptual level, there is no reason to require the base datatype of a Set to be finite. There may, of course, be implementation limitations.

**Issue 30.** Should the base type of Set be restricted to exact datatypes?

Exactness is required to assure independence of implementation. Any implementation of an exact datatype must be able to distinguish exactly the conceptual values. This requirement does not exist for approximate datatypes — it is permissible in representing approximate datatypes to have more than the conceptual values and to be unable to distinguish values which are sufficiently close. But the values of members of a set-value must be clearly distinguishable, in order for the uniqueness constraint and the IsIn operation to be well-defined.

### F.8.4.3 Bag

**Issue 31.** Should the base type of Bag be restricted to exact datatypes?

Exactness is required to assure independence of implementation. Like Sets, the values of members of a bag-value must be clearly

distinguishable, in order for the Delete and Insert operations to be well-defined.

### F.8.4.5 Array

**Issue 32.** Is Array a variant of Sequence?

No. The important characteristic of an Array is the mapping of the index types onto the element type, while Sequence captures the fundamental notion of *sequence*. They are only related by having similar representations. An Array can be made into a sequence by adopting a convention for mapping the index space into the ordinals. There is nothing intrinsic about this mapping: if one chooses different conventions, as Fortran and Pascal do, one gets *different* sequences which represent the *same* array value. And in general, there is *no* array datatype which can be mapped to the value space of a sequence datatype: the set of values of a given size is the image of many array datatypes, but each different size is the image of a different array datatype.

**Issue 33.** Does the syntax of the array-type properly support “Dynamic sized arrays”?

There are several “dynamic” size and shape notions applied to array types in various programming languages:

Array-types whose values have different numbers of elements (Ada [1:?n]). Such types are designated Sequence in this International Standard (clause 8.4.4) and are fully supported thereby, although the complete Ada semantics may also require use of the SIZE subtype capability (clause 8.2.4).

“Conformant” array-types -- types of procedure parameters whose subscript ranges are dependent on the values of other parameters. Such types are supported in this International Standard by Array types (clause 8.4.5) whose subscript ranges are “dependent-values” (clause 7.5.2), i.e. values of other parameters or other elements of a Record which contains the Array.

Array parameters whose “shape” is implicitly passed by the caller, possibly including array parameters with a variable number of dimensions. This is not supported directly by LI datatypes. In general, what is actually passed is either a caller-defined subscripting function or a set of parameters by which the called subprogram can reconstruct the subscripting function. In a language-independent interface, in order for the two language environments to agree on the operations on the passed array value, the “shape” function or parameters must be made explicit. Thus, this case is a special case of “conformant” arrays using “dependent-values” which are other passed parameters.

## F.9 Declarations

**Issue 34.** How will multiple and contradictory definitions of defined-datatypes be avoided?

It is expected that datatype definitions will occur in at least the following places:

a) this International Standard
b) standards containing the outward mappings of programming languages
c) standards defining service interfaces
d) the LI Procedure Calling and Remote Procedure Calling standards
e) users using the Interface Definition Notation for the LIPC/RPC.
f) other user applications

In all of cases a-d, the reference to a *standard* ensures common understanding of the name and meaning of the defined-datatype. In case e, it is expected that all users of the same procedure interface will share a common IDN description – a kind of “local standard” ensuring common understanding. In case f, if the application is private to a particular user, it is not necessary for it to be shared, and if it is not private, then one of the means a-e should be sought. Nonetheless, over time, it may be expected that multiple definitions of a common datatype will occur in cases b and c. This would certainly be grounds for modifying Clause 10 of this International Standard. On the other hand, definitions of different datatypes with the same name can be expected in cases b, c and e as well. This is unfortunate and cannot be avoided in the general case, but it does not affect the interchange of datatypes, except when conflicting standards are used in the same application. A work-around for this should be provided in the LIPC/RPC, but in general, this situation is probably grounds for a revision of the standards in question.

### F.10.1.1 Natural number

**Issue 35.** Should NaturalNumber or Unsigned be LI datatypes?

Naturalnumber is a semantic datatype, but for LI datatype purposes, it is nothing more than integer range(0..*) and is so declared. "Unsigned" is an implementation convention for the representation of certain Integer and Enumerated datatypes, including Naturalnumber.

### F.10.1.2 Modulo

**Issue 36.** Should Modulo be limited to integers?

In various drafts, Modulo has been:

a) a datatype derived from Integer,
b) a datatype generator applicable to any ordered datatype, with extremely complex characterizing operations,
c) a defined generator, applicable only to enumerated datatypes, which redefines Successor.

Characterization (a) is deemed to be the only commonly occurring instance of (b) and has properties that do not generalize, such as multiplication. Characterization (b) is at most a defined generator, because Modulo affects only the operations, not the value space, and applicability to arbitrary ordered datatypes is an unnecessarily complex generalization. Characterization (c), however, is thought to be potentially useful and is retained as “Cyclic of (enumerated datatype)”.

### F.10.1.3 Bit

**Issue 37.** What is the nature of the Bit datatype?

The LI datatypes define four two-valued datatypes, all of which are semantically different, and each of which is some expert’s definition of "Bit". Making some or all of these datatypes identical is a feature of some programming languages, while making them distinct is a feature of others. The LI datatypes must support the latter, while proper use of mapping will support the former.

In the standard, the datatype Bit is used to refer to the numeric finite field of two values — the Modulo(2) datatype derived from Integer — which is conveyed by the term "binary digit". The datatype integer range(0..1) is different, in that Add (1,1) produces different results in the two datatypes. The datatype Boolean is mathematically equivalent to Bit, in that identification of the Xor (Add) and And (Multiply) operations produces the same finite field. But semantically, Boolean is not a numeric datatype and can be characterized by other operations associated with the logic notions true and false, while Bit is a numeric datatype and is characterized by the numeric operations Add and Multiply only. Two-valued Enumerated or State datatypes are none of the above. They have neither numeric nor logical operations. Since the cardinality of all the value spaces is 2, it is obviously possible to map one into another, but it is the characterizing operations which determine the true datatype.

### F.10.1.5 Character string

**Issue 38.** Is Character-string primitive?

No. A character-string must be manipulated as a sequence of members of some character-set in order for the definition of the character-set itself to be useful. That is, the definition of any such datatype is dependent on the (International) Standard defining the character-set. Thus the character datatype whose value space is defined by the standard is the primitive datatype and the character-string datatypes are constructed from it. Some programming languages make the character-string primitive in order to define useful operations that don’t generalize to Sequences or Arrays in that language. Others, such as LISP, APL and Pascal make the single character a primitive type.

**Issue 39.** Should Character-string types be ordered?

The problem is that the collating sequence for character-strings using the same character-set varies from nation to nation and is often constrained by other application-dependent standards. Thus, although everyone agrees that these datatypes are conceptually ordered, there is no agreement on what that ordering is. Therefore, no standard InOrder function can be defined, and for that reason these types are said to be unordered. (See Issue 9.)

### F.10.2 Defined generators

**Issue 40.** Should mathematical Matrix and Tensor constructors be standard generators?

At one level, Tensor-of-degree-n is simply an array datatype with mathematical operations, e.g.

```
type tensor2 (rows: integer, columns: integer, numbers: type) = new array (1..rows, 1..columns) of (numbers);
```

But Tensor is, at another level, a legitimate mathematical datatype generator, which generates vector spaces, or linear operator spaces, over a numeric datatype. The consensus is:

a) The tensor datatype generator is adequately supported by generator-declaration, and could be added to subclause 10.2 if there were consensus on the numbering of the elements (from 0, from 1) and on the ordering of the dimension specifications (rows first, columns first, etc.). (There is no such consensus.)
b) Conceptually, Tensor should be the mathematical object, but the mathematical type generator is not really supported by any programming language. Some programming languages (e.g. BASIC, APL) support special operations on array datatypes which support the mathematical interpretation of the array representation, but these operations tend to be generalized to the array datatypes as such and only in some cases emulate the mathematical operations. Thus Tensor is

outside the scope of the LI datatypes.

**Issue 41.** Should File be a standard generator?

A file, seen as a medium or the object managed by the operating system, which has name, type, organization, state, position, etc., attributes, goes beyond the scope of this standard. The datatype, its attributes and operations, are better defined by an operating system services standard. To the extent that such file objects are integral to programming languages, it is necessary that they be defined for the specific programming language, since there does not appear to be a common model.

A file, seen as a structure of datatype values, may be adequately supported by an aggregate type generator, such as Sequence, Array or Table (see clause 8.4 and also Annex D.2.7).

## F.11 Mappings

**Issue 42.** How much of the concept "mapping onto the LI datatypes" should be standardized?

Consensus is that formal requirements for *indirect conformance* are necessary to relate language standards to language-independent specifications. The mapping is a necessary part of the concept of indirect conformance and therefore a necessary part of this standard. There is further consensus that the standard should specify exactly what a mapping, or a set of mappings, consists of. This should include specifying values of all "parameters" of the LI datatypes, and a discussion of the distinction between "logical identification of two datatypes" and "physical transformation between two datatypes". It should be left to the language standards to formalize the individual mappings, since distinguishing the language syntax constructions which equate to various LI datatypes might be quite complicated.

**Issue 43.** What support of “aggregate properties” should be required?

There was no consensus on requirements for support of aggregate properties, most notably the nature of array indexing (direct access) as against position in sequence (indirect access). Thus the consensus standard contains no requirements for support of aggregate properties.

**Issue 44.** Should the standard address implementation of a mapping?

The implementation of a mapping or binding may occur at the level of language syntax (the representation of the type itself in another language) or at the level of value representation or both. Such requirements are left to other standards which use datatypes and datatype syntax for a particular purpose. The binding for a datatype in databases and exchange files, for example, may specify a particular value representation but no operations, while requirements for support of the same datatype in a programming language specify syntax and operations but not representation.