(2nd) CD 11404

JTC1/SC22/WG11 N345

ISO

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION

ISO/IEC JTC 1/SC 22

LANGUAGES

Secretariat: Canada

PROJECT: JTC1.22.17

TITLE: Information Technology —
Language-Independent Datatypes

STATUS: Second Committee Draft
(Working Draft #7)

DATE: 14 December 1992

SOURCE: ISO/IEC JTC1/SC22/WG11
LI Datatypes Editor
Edward J. Barkmeyer, USA

This document is currently being balloted by national body members of the ISO. Readers of this document are encouraged to forward their comments, both technical and editorial, to the organization responsible for casting the ballot for their national body. Technical changes to the document can now only result from issues raised by ISO member bodies or other standards development bodies. Assistance in contacting the national body organization can be obtained from:

ISO/IEC JTC1/SC22 Secretariat
Mr. Joseph L. Coté
Treasury Board of Canada
140 O'Connor Street 10th Floor
Ottawa, Ontario
CANADA K1A OR5

Telephone: +1 (613) 957-2496
FAX: +1 (613) 957-8700
Email: tbsitm@nrcvm01.bitnet

Informal and purely editorial comments may also be sent to:

Willem Wakker
Convenor, ISO/IEC JTC1/SC22/WG11
ACE Associated Computer Experts BV
Van Eeghenstraat 100
1071 GL Amsterdam
NETHERLANDS

Telephone: +31 20 6646416
FAX: +31 20 6750389
Email: willemw@ace.nl (Internet)

or to:

Edward J. Barkmeyer
Editor, ISO/IEC Project JTC1.22.17
National Institute of Standards and Technology
Bldg 220, Room A127
Gaithersburg, MD 20899-0001
USA

Telephone: +1 (301) 975-3528
FAX: +1 (301) 258-9749
Email: edbark@cme.nist.gov (Internet)

Informal technical comments may only be considered during the resolution of related formal technical comments. All editorial comments will be considered.

Foreword

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 referred to as the "<language> binding of <service>", e.g. the "Fortran binding of PHIGS (ISO 9593)".

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/WG11 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 draft 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 all common datatypes in standard programming languages and service packages can be mapped onto some datatype in the set.

The purpose of this draft 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 draft 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 draft 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.

This document was prepared by ISO/IEC JTC1, Information Technology, and approved as an International Standard on the 1st day of January 199x. This is a new standard.

This document contains the following Annexes:

- Annex A. Character-Set Standards – informative.
- Annex B. Recommended Placement of Annotations – informative.
- Annex C. Implementation Notions of Datatypes – informative.
- Annex D. Syntax for the Common Interface Definition Notation – informative.
- Annex E. Example Mapping – informative
- Annex F. Resolved Issues – informative.

This document contains many paragraphs designated "Notes", which explain relationships of the normative text to common mathematical or programming concepts, or distinguish differences in the understanding of datatypes or concepts. These Notes are not a part of the normative text.

Table of Contents

1. Scope 1

2. Normative References 1

3. Definitions 1

4. Conventions Used in this Standard 4

4.1 Formal syntax 4

4.2 Text conventions 5

5. Compliance 5

5.1 Direct compliance 5

5.2 Indirect compliance 6

5.3 Compliance of a Mapping Standard 6

6. Fundamental Notions 7

6.1 Datatype 7

6.2 Value space 7

6.3 Datatype Properties 7

6.3.1 Equality 7

6.3.2 Ordering 8

6.3.3 Bound 8

6.3.4 Cardinality 8

6.3.5 Exact and Approximate 9

6.3.6 Numeric 9

6.4 Primitive and Non-Primitive datatypes 9

6.5 Datatype generator 10

6.6 Characterizing operations 10

6.7 Datatype families 11

6.8 Aggregate Datatypes 11

6.8.1 Homogeneity 11

6.8.2 Size 12

6.8.3 Uniqueness 12

6.8.4 (Aggregate-imposed) Ordering 12

6.8.5 Access Method 12

6.8.6 Recursive structure 13

7. Elements of the Datatype Specification Language 13

7.1 Character-set 13

7.2 Whitespace 14

7.3 Lexical Objects 14

7.3.1 Identifiers 14

7.3.2 Digit-String 14

7.3.3 Character or String Literal 14

7.4 Annotations 15

7.5 Values 15

7.5.1 Independent values 15

7.5.2 Dependent values 16

8. Datatypes 17
8.1 Primitive Datatypes 18
8.1.1 Boolean 19
8.1.2 State 19
8.1.3 Enumerated 20
8.1.4 Character 20
8.1.5 Ordinal 21
8.1.6 Date-and-Time 22
8.1.7 Bit 23
8.1.8 Integer 23
8.1.9 Rational 24
8.1.10 Scaled 25
8.1.11 Real 26
8.1.12 Complex 27
8.1.13 Void 29
8.2 Subtypes 29
8.2.1 Range 30
8.2.2 Selecting 30
8.2.3 Excluding 31
8.2.4 Extended 31
8.2.5 Size 32
8.2.6 Explicit subtypes 32
8.3 Generated datatypes 32
8.3.1 Choice 33
8.3.2 Pointer 35
8.3.3 Procedure 36
8.4 Aggregate Datatypes 39
8.4.1 Record 40
8.4.2 Set 41
8.4.3 Bag 42
8.4.4 Sequence 43
8.4.5 Array 44
8.4.6 Table 46
8.5 Defined Datatypes 48

9. Declarations 48
9.1 Type Declarations 48
9.1.1 Renaming declarations 49
9.1.2 New datatype declarations 49
9.1.3 New generator declarations 49
9.2 Value Declarations 50
9.3 Termination Declarations 50

10. Derived Datatypes and Generators 50
10.1 Defined datatypes 50
10.1.1 Switch 51
10.1.2 Cardinal 51

10.1.3 Bit string 52
10.1.4 Character string 52
10.1.5 Modulo 53
10.1.6 Currency 53
10.1.7 Interval 53
10.1.8 Octet 54
10.1.9 Private 54
10.1.10 Object-Identifier 54
10.1.11 Distinguished-Name 55
10.2 Defined Generators 56
10.2.1 Stack 57
10.2.2 Tree 57
10.2.3 Cyclic-Enumerated 57

11. Support of Datatypes 58
11.1 Support of equality 58
11.2 Support of ordering and bounds 58
11.3 Support of cardinality 58
11.4 Support for the exact or approximate property 58
11.5 Support for the numeric property 59

12. Mappings 59
12.1 Outward Mappings 60
12.2 Inward Mappings 60
12.3 Reverse Inward Mapping 61

Annex A. Character-Set Standards 63

Annex B. Recommended Placement of Annotations 64
B.1 Type-attributes 64
B.2 Component-attributes 64
B.3 Procedure-attributes 64
B.4 Argument-attributes 65

Annex C. Implementation Notions of Datatypes 66
C.1 Size 66
C.2 Mode 66
C.3 Floating-Point 66
C.4 Fixed-Point 67
C.5 Tag 67
C.6 Discriminant 67
C.7 Sequence 67
C.8 Packed 67
C.9 Alignment 68
C.10 Form 68

Annex D. Syntax for the Common Interface Definition Notation 69

Annex E. Example Mapping 74
E.1 LI Primitive Datatypes 74

E.1.1 Boolean 74
E.1.2 State 74
E.1.3 Enumerated 74
E.1.4 Character 74
E.1.5 Ordinal 74
E.1.6 Time 74
E.1.7 Bit 74
E.1.8 Integer 75
E.1.9 Rational 75
E.1.10 Scaled 76
E.1.11 Real 77
E.1.12 Complex 77
E.1.13 Void 77
E.2 LI Generated Types 78
E.2.1 Choice 78
E.2.2 Pointer 78
E.2.3 Procedure 79
E.2.4 Record 79
E.2.5 Set 79
E.2.6 Bag 80
E.2.7 Sequence 80
E.2.8 Array 80
E.2.9 Table 80
E.3 LI Subtypes 81
E.3.1 Range 81
E.3.2 Selecting 81
E.3.3 Excluding 81
E.3.4 Extended 81
E.3.5 Size 81
E.3.6 Explicit 81
E.4 LI Defined Datatypes 81
E.4.1 Bit-String 81
E.4.2 Character-String 82
E.4.3 Octet 82
E.4.4 Private 82
E.4.5 Object-identifier 83
E.4.6 Distinguished-Name 83
E.5 Type-Declarations and Defined Datatypes 83
E.5.1 Renaming declarations 83
E.5.2 Datatype declarations 83
E.5.3 Generator declarations 83
Annex F. Resolved Issues 84

1. Scope

This draft 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 (Common) 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 draft 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 draft 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 draft International Standard defines the abstract notions of many commonly used primitive and non-primitive datatypes which possess the structural notion of atomicity. This draft International Standard does not define all atomic datatypes; it defines only those which are common in programming languages and software interfaces. This draft 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 draft 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 draft International Standard, which is concerned solely with the identification and distinction of datatypes.

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

2. Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this draft 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 draft 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 8601:1988 Representation of dates and times

ISO 8824:1989 Abstract Syntax Notation One

3. Definitions

For the purposes of this draft International Standard, the following definitions apply:

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

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

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

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

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

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

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

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

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

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

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

3.11 datatype: a set of distinct values, a collection of relationships among those values and a collection of characterizing operations on those values.

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

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

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

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

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

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

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

3.19 generated internal datatype: a datatype defined by the application of a datatype generator defined in a partic-

ular programming language to one or more previously-defined internal datatypes.

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

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

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

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

3.24 LI datatype: a datatype which is either:
(a) defined by this draft International Standard, or
(b) defined by the means of datatype definition provided by this draft International Standard.

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

3.26 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 draft International Standard;
(of values): a corresponding specification of the relationships between values of the internal datatypes and values of the LI datatypes.

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

3.28 ordering: a mathematical relationship among values (see 6.3.2).

3.29 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.

3.30 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 datatype resulting from the generator.

3.31 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 datatype in the family or resulting from the generator.

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

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

3.34 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.

3.35 subtype: a datatype derived from another datatype by restricting the value space to a subset whilst maintaining

all characterizing operations.

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

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

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

4. Conventions Used in this Standard

4.1 Formal syntax

This draft 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.

A **terminal symbol** is a sequence of characters drawn from the character set defined in Table 4-1, delimited by two occurrences of the quotation-mark (") or two occurrences of the apostrophe-mark ('), of which the first occurrence precedes the first character in the terminal symbol, and the second occurrence follows the last character in the terminal symbol. A terminal symbol represents the occurrence of that sequence of characters.

A **non-terminal symbol** is a sequence of marks, each of which is either a letter or the hyphen mark (-), terminated by the first mark which is neither a letter nor a hyphen. 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 an open-brace mark ({) and a close-brace mark (}). A repeated sequence represents any number of consecutive occurrences of the sequence of symbols so enclosed, including no occurrence.

Table 4-1. Character Set

Type	Characters					
letter	a b c d e f g h i j k l m n o p q r s t u v w x y z					
digit	0 1 2 3 4 5 6 7 8 9					
special	( ) (parentheses)	{ } (braces)	< > (angle-brackets)	. (full stop)	, (comma)	: (colon)
	; (semicolon)	= (equals)	/ (solidus)	* (asterisk)	- (minus)	
hyphen	_					
apostrophe	'					
quote	"					
escape	!					
space						

An **optional sequence** is a sequence of terminal and/or non-terminal symbols enclosed between an open-bracket mark ([) and a close-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 the vertical stroke (|) mark and followed by either a vertical stroke 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 stroke.

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 equal-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.

4.2 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.

5. Compliance

An information processing product, system, element or other entity may comply with this draft International Standard either directly, by utilizing datatypes specified in this draft International Standard in a conforming manner (ref. 5.1), or indirectly, by means of mappings between internal datatypes used by the entity and the datatypes specified in this draft International Standard (ref. 5.2).

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 compliance with this draft 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.

5.1 Direct compliance

An information processing entity which **complies directly** with this draft International Standard shall:

i) define and refer to datatypes within the entity using the syntax prescribed by clauses 7 through 10 of this draft International Standard; and

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

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

iv) 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.

NOTES

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

requirement *(i)* 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 draft International Standard to designate any other datatype. They make no 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 *(iii)* 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 draft 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 comply 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.

5.2 Indirect compliance

An information processing entity which **complies indirectly** with this draft International Standard shall:

i) provide mappings between its internal datatypes and the LI datatypes conforming to the specifications of Clause 12 of this draft International Standard; and

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

NOTE – Examples of entities which could comply 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.

5.3 Compliance of a Mapping Standard

In order to comply with this draft International Standard, a standard for a mapping shall include in its compliance requirements the requirement to comply with this draft International Standard.

NOTES

1. It is envisaged that this draft 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 draft International Standard.

2. Such mapping standards may define "generic" mappings, in the sense that for a given internal datatype the standard specifies a parametrized LI datatype in which the parameter values are not derived from parameters 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".

6. Fundamental Notions

6.1 Datatype

A **datatype** is a collection of distinct values, a collection of properties of those values and a collection of characterizing operations on those values.

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

A value space is said to be **ordered** if there exists for the value space an **ordering** 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 ordering 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 ordering. The chosen ordering 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 ordering 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". And thus C is *not* an exact model of M.

In this draft International Standard, all approximate datatypes have computational models which specify, via parameters, 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 (8.1.10). 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 (8.1.11) and Complex (8.1.12) 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 draft 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 draft 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 draft 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 draft 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 draft 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 **component 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 component 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 component datatype, into values of the given datatype or a component datatype.

4. In general, there is no unique collection of characterizing operations for a given datatype. This draft 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 (ref. 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 draft International Standard, the symbol set for an individual member of a datatype family is specified by one or more values, called the **parameters** 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 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 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 aggregate datatype with a single ordered index type 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 satis-

fying 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 draft 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 draft International Standard defines a datatype specification language, in order to formalize the identification and declaration of datatypes conforming to this draft International Standard. The language is a subset of the Interface Definition Notation defined in ISO ??? Language-Independent Procedure Calling, which is completely specified in Annex D. This clause defines the basic syntactic objects used in that language.

7.1 Character-set

```
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 =         "(" | ")" | "." | "," | ":" | ";" | "=" | "/" | "*" | "-" | "{" | "}" .
hyphen =          "_" .
apostrophe =      "'" .
quote =           '"' .
escape =          "!" .
space =           " " .
non-quote-character = letter | digit | hyphen | special | apostrophe | space .
bound-character = non-quote-character | quote .
added-character = <not defined by this draft International Standard> .
```

Rules for the interpretation of the character-set productions:

i) These productions are derived from Table 4-1 and are nominal. Lexical productions are always subject to minor changes from implementation to implementation, in order to handle the vagaries of available character-sets.

ii) The character *space* is required to be bound to the "space" member of ISO 10646: 1992, but it only has meaning within character-literals and string-literals.

iii) A *bound-character* is required to be associated with the member having the corresponding symbol in any character-set derived from ISO 10646:1992, except that no significance is attached to the "case" of letters.

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

v) 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 10646 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 draft 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 draft International Standard, any annotation (see 7.4) is considered whitespace.

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

7.3 Lexical Objects

The lexical objects are the terminal symbols 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 | hyphen .
```

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:

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

ii) 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.

The term **keyword** refers to any terminal symbol which is not lexically distinguishable from an identifier (is not composed of special characters). No keyword shall be interpreted as an identifier. Any two consecutive keywords or identifiers, or a keyword preceded or followed by an identifier, shall be separated by whitespace.

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 or 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), bit-string datatypes (10.1.3), and character-string datatypes (10.1.4), 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.

7.4 Annotations

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

annotation = "<" annotation-label ":" annotation-text ">" .
annotation-label = object-identifier-component-list .
annotation-text = <not defined by this draft 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 *object-identifier-component-list* shall have the structure and meaning prescribed by clause 10.1.10.

NOTE – Of the several forms of *object-identifier-component* specified in 10.1.10, the *nameform* is the most convenient for labelling annotations. Following ISO 8824, every value of the object-identifier 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 : object-identifier = { iso standard 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 11578:199x 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 draft International Standard provides a notation for values of every datatype defined herein, or which 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 .

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.

7.5.1 Independent values

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

independent-value = explicit-value | qualified-value | value-identifier | parametric-value .
qualified-value = type-specifier "." explicit-value .
explicit-value = value-literal | composite-value | derived-value .
value-literal = boolean-literal | state-literal | enumerated-literal | character-literal
| ordinal-literal | time-literal | bit-literal | integer-literal | rational-literal
| scaled-literal | real-literal | complex-literal | void-literal
| extended-literal | pointer-literal .

```
composite-value = choice-value | record-value | set-value | sequence-value
                | bag-value | array-value | table-value .
derived-value = string-literal | object-identifier-value .
parametric-value = formal-parameter-name .
value-identifier = identifier .
```

An *explicit-value* uses an explicit syntax for values of the datatype, as defined in clauses 8 and 10. The interpretation of *value-literals* is specified in subclauses of 8.1; the interpretation of *composite-values* is specified in subclauses of 8.4; and the interpretation of *derived-values* is specified in subclauses of 10.1. A *qualified-value* denotes that value of the datatype designated by the *type-specifier* which is designated by the *explicit-value*. A *qualified-value* should be used when the *explicit-value* would otherwise be ambiguous.

A *value-identifier* designates the value associated with that identifier by a *value-declaration*, as provided in 9.2. A *parametric-value* refers to the value of a *formal-parameter* in a *type-declaration*, as provided in 9.1.

NOTES

1. Two syntactically different explicit-values may designate the same value, such as rational-literals 3/4 and 6/8, or set of (integer) values (1,3,4) and (4,3,1). Conversely, the same explicit-value may designate values of two different datatypes when it appears in a qualified value.

2. In general, the syntax requires that the intended datatype of a *value-expression* can be determined (from context) when the *value-expression* is encountered. Thus *qualified-values* should rarely be required, because there are few instances in which an *"explicit-value* would otherwise be ambiguous."

7.5.2 Dependent values

When a parameterized datatype appears within a procedure argument (see 8.3.3) or a record datatype (see 8.4.1), it is possible to specify that the parameter value is always identical to the value of another argument 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 | argument-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 an *argument-name* and shall identify an argument of the subject datatype. If the *direction* of the argument (component) which contains the data-dependent type is "in" or "inout", then the *direction* of the argument designated by the *primary-dependency* shall also be "in" or "inout". If the argument which contains the data-dependent type is the *return-argument* or has *direction* "out", then the *primary-dependency* may designate any argument in the *argument-list*. If the argument which contains the data-dependent type is a *termination* argument, then the *primary-dependency* shall designate another argument in the same *termination-argument-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:

i) 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.

ii) 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.

iii) 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*.

iv) 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 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 y, 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:

1) explicit specification of **primitive** datatypes, which have universal well-defined abstract notions, each independent of any other datatype.
2) 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.
3) 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.
4) 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 | defined-type | parametric-type .

A *type-specifier* shall not be a *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 any LI datatype. 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 draft 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 draft International Standard.

primitive-type = boolean-type | state-type | enumerated-type | character-type
| ordinal-type | time-type | bit-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.
Parameters:	identification of any parameters 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, – ordered or unordered, – approximate or exact, – if ordered, bounded or unbounded.
Operations:	definitions of characterizing operations.

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

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

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

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

argument-name : argument-datatype, or

argument-name$_1$, argument-name$_2$: argument-datatype.

The *operation-name* is an identifier unique only within the datatype being defined. The *argument-names* are formal identifiers appearing in the *formal-* or *prose-definition*. Each is understood to represent an arbitrary value of the datatype designated by *argument-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 argument 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(arguments)*, 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" .

Parameters: 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 with no characterizing operations,except Equal.

Syntax:

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

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

state-literal = identifier .

Parameters: Each *state-literal* identifier shall be distinct from all other *state-literal* identifiers of the same *state-type*. If it is necessary to disambiguate a reference to a value of a state datatype, the *qualified-value* form may be used (see 7.5.1).

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. If it is necessary to disambiguate a reference to a value of a state datatype, the *qualified-value* form may be used (see 7.5.1).

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.

8.1.3 Enumerated

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

Syntax:

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

Parameters: 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 ordering 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. If it is necessary to disambiguate a reference to a value of an enumerated datatype, the *qualified-value* form may be used (see 7.5.1).

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 $y \leq z$.

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 .

Parameters: The *value-expression* for a *repertoire-identifier* shall designate a value of the object-identifier datatype (see 10.1.10), and that value shall refer to a character-set. A *repertoire-identifier* shall not be a *parametric-value*, except in some cases in declarations (see 9.1). All *repertoire-identifiers* in the *repertoire-list* shall designate subsets of a single 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 draft 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 contains only the distinct members.

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 | hyphen | special | apostrophe | space .
added-character = <not defined by this draft 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 draft 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 draft 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 draft International Standard. A partial list of the international standards defining such character-sets is included, for informative purposes only, in Annex A.

3. While ordering is important in many applications of character datatypes, there is no standard ordering for any of the International Standard character sets, and many applications require the ordering of the datatype to conform to rules which are particular to the application itself or its language environment. There will also be applications in which the ordering is unimportant. Since no standard ordering of character-sets can be defined by this draft International Standard, character datatypes are said to be "unordered", meaning, in this case, that the ordering 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 10646, but there should be analogous notions in any character set referenceable by a *repertoire-identifier*.

EXAMPLE

character({ iso standard 8859 1 })

denotes a character datatype whose values are the members of the character-set defined by ISO 8859-1 (the Latin alphabet). 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 : object-identifier = { iso standard 8859 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" .

Parameters: 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"
| parametric-value .

radix = value-expression .

factor = value-expression .

Parameters: *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 *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 = digit-string ["." digit-string] .

A *time-literal* denotes a date-and-time value. It shall be a string of digits and possibly a single decimal-point, conforming to ISO 8801:1988 and interpreted as specified therein.

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

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

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

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

8.1.7 Bit

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

Syntax:

bit-type = "bit" .

Parameters: none.

Values: 0, 1

Value-syntax:

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

Properties: ordered, exact, numeric, bounded.

Operations: Equal, InOrder, Add, Multiply

Equal(x,y: bit): boolean and InOrder(x,y: bit): boolean are defined by tabulation:

x	y	Equal(x,y)	InOrder(x,y)
1	1	true	true
1	0	false	false
0	1	false	true
0	0	true	true

Add(x,y: bit): bit and Multiply(x,y: bit):bit are defined by tabulation:

x	y	Add(x,y)	Multiply(x,y)
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

8.1.8 Integer

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

Syntax:

integer-type = "integer" .

Parameters: none.

Values: Mathematically, the infinite ring produced from the additive identity (0) and the multiplicative identity (1) by requiring Add(x,1) ≠ y for any y ≤ 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, Promote

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

Promote(x:bit): integer is defined by tabulation:

x	Promote(x)
0	0
1	1

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 $0 \leq x$ and $0 < y$, =
Add(x, Negate(Multiply(y, Quotient(x,y))));

8.1.9 Rational

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

Syntax:

rational-type = "rational" .

Parameters: 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.

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 $x \neq 0$, 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)))

8.1.10 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 .

Parameters: *Radix* shall have an integer value greater than 1, and *factor* shall have an integer value. *Radix* and *factor* shall not be *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)))

EXAMPLE

Currency values for many nations can be represented by the datatype: scaled(10, 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 draft International Standard, but could be defined consistent with the Divide operation for scaled(*r*, 0). It is recommended that the datatype scaled(*r*, 0) not be used explicitly.

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

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

4. The values in any scaled value space are taken from the value space of the Rational datatype, and for that reason Scaled may appear to be a "subtype" of both Rational and Real (ref. 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 draft 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.

8.1.11 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 .

Parameters: *Radix* shall have an integer value greater than 1, and *factor* shall have an integer value. *Radix* and *factor* shall not be *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 draft 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 parameters, *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*), i.e. the collection of values which must be distinguishable by the application. Then there shall exist an *approximation function* M, which maps $\Re$ into V with the following properties:

- 0 is a member of V, and $M(0) = 0$;
- for all v in $\Re$ such that $|v| \geq radix^{(-factor)}$,
  $|M(v) - v| \leq |v| \cdot radix^{(-factor)}$;
- for any two values v_1 and v_2 in $\Re$ such that $|v_1| < radix^{(-factor)}$ and $|v_2| < radix^{(-factor)}$, if $|v_1 - v_2| \geq radix^{(-2 \cdot factor)}$, then $M(v_1) \neq M(v_2)$.

The value space of the computational datatype real(*radix*, *factor*) is any subset of the mathematical real type for which such an approximation function exists. Detailed requirements for the approximation function and its relationship to the characterizing operations are outside the scope of this draft International Standard.

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 the approximation function defined above:

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

InOrder(x,y: real(*radix, factor*)): boolean is true if $x \leq y$, where $\leq$ designates the ordering relationship on the mathematical reals, and false otherwise.

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.

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 dIS 10967 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.

8.1.12 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 .
```

Parameters: *Radix* shall have an integer value greater than 1, and *factor* shall have an integer value. *Radix* and *factor* shall not be *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 draft 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 parameters, *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*), i.e. the collection of values which must be distinguishable by the application. Then there shall exist an *approximation function* M, which maps C into V with the following properties:

- 0 is a member of V, and $M(0) = 0$;
- for all v in C such that $|v| \geq radix^{(-factor)}$,
  $|M(v) - v| \leq |v| \bullet radix^{(-factor)}$.
- for any two values v_1 and v_2 in C such that $|v_1| < radix^{(-factor)}$ and $|v_2| < radix^{(-factor)}$, if $|v_1 - v_2| \geq radix^{(-2 \bullet factor)}$, then $M(v_1) \neq M(v_2)$.

The value space of the computational datatype complex(*radix*, *factor*) is any subset of the mathematical complex type for which such an approximation function exists. Detailed requirements for the approximation function and its relationship to the characterizing operations are outside the scope of this draft International Standard.

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: $M(real\text{-}part + (imaginary\text{-}part \bullet i))$, 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 the approximation function defined above:

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 10967 Language-Independent

Arithmetic.

8.1.13 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" .

Parameters: 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 argument 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 draft 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

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
| extended-type | size-subtype | explicit-subtype .

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 parameters which are necessary for the complete identification of a distinct subtype.
Components:	constraints on the base datatype and other parameters.
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 *parametric-values*, except in some occurrences in declarations (see 9.1).

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

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

8.2.2 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 *parametric-value*, except in some occurrences in declarations (see 9.1).

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

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

8.2.3 Excluding

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

Syntax:

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

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

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

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

8.2.4 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 | 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.

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

8.2.5 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 be not be *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.

8.2.6 Explicit subtypes

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

Syntax:

```
explicit-subtype = base ":" "subtype" "(" subtype-definition ")" .
base = type-specifier .
subtype-definition = type-specifier .
```

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

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

Properties: exactly those of the *subtype-definition* datatype.

NOTES

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

2. It is redundant, but syntactically acceptable, for the *subtype-definition* to be an occurrence of a subtype-generator, e.g. integer: subtype (integer: selecting(0..5)).

8.3 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 **component datatypes**. The generated datatype is semantically dependent on the component datatypes, but has its own characterizing operations. An important characteristic of all datatype generators is that the

component datatypes are parametric, that is, that the generator can be applied to many different component datatypes. The Pointer and Procedure generators generate datatypes whose values are atomic, while Choice and the generators of aggregate datatypes gemerate datatypes whose values admit of decomposition. A *generated-type* designates a generated datatype.

generated-type = pointer-type | procedure-type | choice-type | aggregate-type .

This draft International Standard defines common datatype generators by which an application of this draft 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 component datatypes which are necessary for the complete identification of a distinct datatype.
Components:	number of and constraints on the component datatypes and other parameters 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, – if ordered, bounded or unbounded.
Subtypes:	generators, subtype-generators and parameters 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" "(" tag-type ")" "of" "(" alternative-list ")" .
tag-type = type-specifier .
alternative-list = alternative { "," alternative } [default-alternative] .
alternative = tag-value-list ":" alternative-type .
default-alternative = "default" ":" alternative-type .
alternative-type = type-specifier .
tag-value-list = select-list .
select-list = select-item { "," select-item } .
select-item = value-expression | select-range .
select-range = lowerbound ".." upperbound .
lowerbound = value-expression | "*" .
upperbound = value-expression | "*" .

Components: Each *alternative-type* in the *alternative-list* may be any datatype. The *tag-type* shall be an exact

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

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

Every value of the tag datatype shall appear (explicitly or implicitly) in the *alternative-list*, and no value of the tag datatype shall appear in the *tag-value-list* of more than one *alternative*.

Values: all values having the conceptual form (*tag-value*, *alternative-value*), where *tag-value* is a value of the tag datatype, which is uniquely mapped to an alternative datatype according to the *alternative-list*, 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, IsType.

IsType.*type*(x: choice (*tag-type*) of (*alternative-list*)): boolean, where *type* is an alternative datatype in *alternative-list*, is true if the tag-value of the value x maps to *type*, and false otherwise.

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 IsType.*type*(x), 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 there exists an alternative datatype *type* in *alternative-list* such that
And(IsType.*type*(x), IsType.*type*(y)), then Equal.*type*(Cast.*type*(x), Cast.*type*(y)),
where Equal.*type* 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 form: *discriminant*: *tag-type*, which occurs in some programming languages, is a means for specifying the source of the tag-value for a given instance of a choice datatype. If such a mechanism is required, as for marshalling arguments to a procedure call, it should be described by annotations (see 7.4).

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.

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. Conceptually, a pointer datatype expresses a relationship between two objects or values. There are two circumstances which require a pointer datatype:

a) when the associated value or object may change, or

b) when more than one object may possess a relationship to the same object.

Pointers used for the latter purpose are said to be **aliased**, as distinguished from pointers used only for the former purpose, which are said to be **unaliased**. Other usages of pointer datatypes are typically implementation mechanisms for aggregate datatypes or procedure argument passing mechanisms, rather than conceptual pointers.

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

Parameters: An *argument-type* may designate any datatype. The *argument-names* of *arguments* in the *argument-list* shall be distinct from each other and from the *argument-name* of the *return-argument*, 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. An *argument* in the *argument-list* is said to be an **input argument** if its *argument-declaration* contains the direction "in" or "inout". The input space is the cross-product of the value spaces of the datatypes designated by the *argument-types* of all the input arguments. An argument is said to be a **result argument** if it is the *return-argument* or it appears in the *argument-list* and its *argument-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 *argument-types* of all the result arguments, 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:

$$p\colon I_1 \times I_2 \times \ldots \times I_n \rightarrow R_P \times R_1 \times R_2 \times \ldots \times R_m$$

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

When a *termination-list* is present, each *termination-reference* is 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 *argument-types* of the *arguments* in the *termination-argument-list*. Let A^j be the alternative result space of the *j*th termination. Then:

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

where E_k^j is the value space of the argument datatype of the *k*th argument in the *termination-argument-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:

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

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

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

Any of the input space, the normal result space and the alternative result space corresponding to a given *termination-identifier* may be empty. An empty space can be modelled mathematically by substituting for the empty space the value space of the datatype Void (see 8.1.13).

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 argument datatypes and yield a collection of values whose datatypes correspond to the argument datatypes of the normal result space or the appropriate alternative result space. The term **corresponding** in this regard means that to each argument 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 "(" [ argument-list ] ")"
                        [ "returns" "(" return-argument ")" ]
                        [ "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 draft 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 *argument-lists* are of the same length, the *direction* of each *argument* in the *argument-list* of *P* is the same as the corresponding *argument* in the *argument-list* of *Q*, both have a *return-argument* 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 argument of *Q* , the argument datatype of the corresponding argument of *P* is a (not necessarily proper) subtype of the argument datatype of the argument of *Q*, then *P* is said to be a **result-subtype** of *Q*. If the return argument datatype and all of the argument datatypes in the *argument-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 argument of *Q* , the argument datatype of the corresponding argument of *P* is a (not necessarily proper) subtype of the argument datatype of the argu-

ment of *Q*, then *Q* is said to be an **input-subtype** of *P*. If all of the input argument datatypes in the *argument-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 argument is a value of the procedure datatype, and whose remaining input arguments, 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(*argument-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(*argument-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 argument-list, etc. The true definition of Invoke is beyond the scope of this draft International Standard and forms a principal part of DIS ?: 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 argument 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 value of integer, 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 argument-names, in addition to argument datatypes, for the arguments. These identifiers provide a means of distinguishing the functionality of the individual argument 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 arguments a and b in the two procedure values is unrelated.

5. In describing procedures in programming languages, it is common to distinguish arguments 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 arguments 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 draft 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: *althernative-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 valuesof 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 characterising. However, the nature of such operations, together with other operations on the aggregate as a whole, is characterising.

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 character-string (10.1.4) as an aggregate, in order to support languages whose fundamental datatype is (single) Character. But Basic, for example, sees the character-string as the primitive type, and defines operations on it which yield other character-strings, wherein 1-character strings are not even a special case. This difference in viewpoint does not prevent a meaningful mapping between the character-string 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 draft 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 draft 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; 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.

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 draft 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 if and only if the element datatype is exact; homogeneous, variable size,

uniqueness, no ordering, access indirect (by value).

Subtypes:

i) 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
ii) any datatype derived from a base set datatype conforming to *(i)* by use of the Size subtype-generator (ref. 8.2.5).

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 if and only if the element datatype is exact; homogeneous, variable size, no uniqueness, no ordering, access indirect.

Subtypes:

i) 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
ii) any datatype derived from a base bag datatype conforming to *(i)* by use of the Size subtype-generator (ref. 8.2.5).

Operations: IsEmpty, Select, Delete, Equal, Empty, 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;

Select(x: bag of (*element-type*)): *element-type*, where Not(IsEmpty(x)), is some one value *e* of the element datatype such that x(*e*) > 0;

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; homogeneous, variable size, no uniqueness, imposed ordering, access indirect (by position).

Subtypes:

i) 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
ii) any datatype derived from a base sequence datatype conforming to *(i)* by use of the Size subtype-generator

(ref. 8.2.5).

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 an *argument-type*, or in a component of an *argument-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 *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; 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 ordering 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. 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.

5. 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 *lower-bound*$_i$ and *upperbound*$_i$ be the lower and upper bounds on its value space. Define the operation Map$_i$ to map the index datatype *index*$_i$ into a range of integer by:

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

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

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

And define the constant: *size*$_i$ = Map$_i$(*upperbound*$_i$) - Map$_i$(*lowerbound*$_i$) + 1. Then Ord(x_1: *index*$_1$, ..., x_n: *index*$_n$): ordinal is the ordinal value corresponding to the integer value:

$$1 + \sum_{i=1}^{n} \text{Map}_i(x_i) \cdot \left(\prod_{j=i}^{n} size_{j+1} \right),$$

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, however, requires a mapping function which gives a different sequence representation from that given in Note 5.

8.4.6 Table

Description: Table generates a datatype, called a **table datatype**, whose values are sets of associations between values in the product space of one or more *key* datatypes and values of the *element* datatype, such that any value in the product space of the key datatypes associates to at most one value of the element datatype. Although the key datatypes may be infinite, any given value of a table datatype contains a finite number of associations.

Syntax:

table-type = "table" "(" key-list ")" "of" "(" element-type ")" .
key-list = key-type { "," key-type } .
key-type = type-specifier .
element-type = type-specifier .

Components: The *element-type* shall designate any datatype, called the **element datatype**. Each *key-type* shall designate any exact datatype, called a **key datatype.**

Values: all finite sets of associations, represented by pairs of values, in which each pair comprises one value from the cross-product of the value spaces of all the key datatypes in the *key-list*, designated the *key product space*, and one value from the value space of the element datatype, with the restriction that no value in the key product space can appear in more than one pair.

Value-syntax:

table-value = empty-value | "(" table-entry { "," table-entry } ")" .
table-entry = key-value-list ":" element-value .
key-value-list = independent-value { "," independent-value } .
element-value = independent-value .

A *table-value* denotes a value of a table datatype, namely the set comprising exactly the *table-entrys* appearing in the *table-value*. Each *independent-value* in the *key-value-list* of a *table-entry* shall designate a value of the key datatype in the corresponding position in the *key-list* of the *table-type*, and the *key-value-list* shall denote that value from the key product space designated by that tuple of *independent-values*. The *element-value* of a *table-entry* shall designate a value of the element datatype of the key datatype. A *table-entry* shall denote the single association between the value of the key product space designated by the *key-value-list* and

the value of the element datatype designated by the *element-value*.

Properties: non-numeric, unordered, exact if and only if the element datatype is exact; homogeneous, variable size, no uniqueness, no ordering, access is keyed, dimensionality is equal to the number of *key-types* in the *key-list*.

Subtypes:

i) any table datatype in which:

• the element datatype of the subtype is the same as, or a subtype of, the element datatype of the base table datatype, and

• the number of *key-types* in the *key-list* of the subtype is the same as the number of *key-types* in the *key-list* of the base table datatype, and

• each key datatype of the subtype is the same as, or a subtype of, the key datatype in the corresponding position in the *key-list* of the base table datatype; or

ii) any table datatype derived from a base table datatype conforming to *(i)* by use of the Size subtype-generator (ref. 8.2.5).

Operations: Equal, Empty, IsPresent, Select, Insert, Delete.

IsPresent(x: table (key_1, ..., key_n) of (*element-type*), y_1: key_1, ..., y_n: key_n): boolean is true if there is a pair in the set x whose key member is the value (y_1, ..., y_n), else false.

Select(x: table (key_1, ..., key_n) of (*element-type*), y_1: key_1, ..., y_n: key_n): *element-type*, where IsPresent(x, y_1, ..., y_n), is that value of the element datatype which x associates with the value (y_1, ..., y_n) in the key product space.

Equal(x, y: table (key_1, ..., key_n) of (*element-type*)): boolean is true if for every value (v_1, ..., v_n) in the key product space such that IsPresent(x, v_1, ..., v_n), Select(x, v_1, ..., v_n) = Select(y, v_1, ..., v_n), else false;

Empty(): table (key_1, ..., key_n) of (*element-type*) is true if for every value (v_1, ..., v_n) in the key product space Not(IsPresent(x, v_1, ..., v_n)), else false; i.e. the set containing no associations.

Insert(x: table (key_1, ..., key_n) of (*element-type*), y_1: key_1, ..., y_n: key_n, z:*element-type*): table (key_1, ..., key_n) of (*element-type*), where Not(IsPresent(x, y)), is the set formed by inserting the pair ((y_1, ..., y_n), z) in the set x.

Delete(x: table (key_1, ..., key_n) of (*element-type*), y_1: key_1, ..., y_n: key_n): table (key_1, ..., key_n) of (*element-type*), where IsPresent(x, y_1, ..., y_n), is the set formed by deleting from the set x the pair whose key member has value (y_1, ..., y_n).

NOTES

1. A table datatype is a generalization of the Array notion to arbitrary "index" (key) types. A table datatype with multiple key datatypes does not have multiple "independent" keys, but rather a single "joint" key. Without loss of generality, a table datatype can be modelled as having a single key datatype of the form record(f_1: key_1, ..., f_n: key_n) where key_1, ..., key_n are the *key-types* in the *key-list*. This is the meaning of the "key product space" model. The syntax given above was intended to facilitate mappings to languages in which the notion of association to a "record-value" is foreign.

2. Following Note 1, the Table generator could be defined (see 9.1.3) by:

type table (key_type: type, element_type: type) = new set of (record(key: key_type, element: element_type)),

but only the characterizing operations Equal and Empty would be derived; all the others are proper to the table datatype. Changing "set" to "bag" or "sequence" does not change the situation. The table datatype uses the heterogeneous character of the "record" to distinguish the roles of "key" and "element", but it defines the role of the key, and that definition is the distinguishing character of the table datatype.

3. Unlike array datatypes, in a value of a table datatype it is not required that an element value be present for every possible value of the key datatype. This gives rise to subtle issues, such as: Can one model a Table as a complete mapping from the key product space into choice (state(present, absent)) of (present: *element-type*, absent: void)? This would mean that Not(IsPresent(T, k)) implies Select(T,k) = nil. Is "not present" different from "present with value nil"? This detail is unimportant to understanding the datatype. Neither answer conflicts with the characterizing operations given above, which don't define Select(T,k) when IsPresent(T,k) = false.

8.5 Defined Datatypes

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

```
defined-type = type-identifier [ "(" actual-parameter-list ")" ] .
type-identifier = identifier .
actual-parameter-list = actual-parameter { "," actual-parameter } .
actual-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-parameters*, if any, shall correspond in number and in type to the *formal-parameters* of the *type-declaration*. That is, each *actual-parameter* corresponds to the *formal-parameter-name* in the corresponding position in the *formal-parameter-list*. If the *formal-parameter-type* is a *type-specifier*, then the *actual-parameter* shall be a *value-expression* designating a value of the datatype specified by the *type-specifier*. If the *formal-parameter-type* is "type", then the *actual-parameter* shall be a *datatype* and shall have the properties required of that parametric datatype in the generator-declaration.

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

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

9. Declarations

This draft 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.

NOTE – This clause provides the mechanisms by which the facilities of this draft 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 defintion 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-parameter-list ")" ]
                   "=" [ "new" ] type-definition .
type-identifier = identifier .
```

```
formal-parameter-list = formal-parameter { "," formal-parameter } .
formal-parameter = formal-parameter-name ":" formal-parameter-type .
formal-parameter-name = identifier .
formal-parameter-type = type-specifier | "type" .
type-definition = type-specifier .
parametric-value = formal-parameter-name .
parametric-type = formal-parameter-name .
```

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

The *type-identifier* declared in a *type-declaration* may be referred in a subsequent use of a *defined-type* (see 8.5). The *formal-parameter-list* declares the number and required nature of the *actual-parameters* which must appear in a *defined-type* which references this *type-identifier*. A *defined-type* 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 *defined-type*.

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 *defined-type* referencing the *type-identifier* refers to the LI datatype identified by the *type-definition*, after substitution of the actual parameters for the corresponding formal 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-parameter-list* is not present, then the *type-identifier* is declared to identify a single LI datatype. If the *formal-parameter-list* is present, then the *type-identifier* is declared to identify a family of datatypes parametrized by the *formal-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* and the ordering, if any, is maintained. 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-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-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-parameters* whose *formal-parameter-type* is "type" are said to be **component datatypes**. A generator declaration shall be accompanied by a statement of the constraints on the component datatypes and on the values of the other *formal-parameters*, if any.

9.2 Value Declarations

A value-declaration declares an identifier to refer to a specific value of a specific datatype. The syntax of a value-declaration is:

```
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 specified LI datatype, as required by Clause 8.

No *independent-value* appearing in a *value-declaration* shall be a *parametric-value* and no *type-specifier* appearing in a *value-declaration* shall be a *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 arguments returned by that termination.

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

The *argument-names* of the *arguments* in a *termination-argument-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. Derived 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 draft 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.
Parameters:	when the defined datatype is a family of datatypes, identification of and constraints on the parameters 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, – ordered or unordered, – approximate or exact, – if ordered, bounded or unbounded.
Operations:	characterizing operations for the datatype.

The notation for values of a defined datatype may be of three kinds, depending on the *type-declaration*:

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 defined without the keyword *new*, then the syntax for *explicit-values* of the datatype identified by the *type-definition* is a valid notation for values of the defined datatype. That is, the *value-literal* or *composite-value* appropriate to the equivalent datatype may be used.
3) In all cases, a *qualified-value* is a valid notation for values of the defined datatype (see 7.5.1).

10.1.1 Switch

Description: Switch is a family of state datatypes, each of which comprises two distinguished but unordered values with the characteristic operation Invert.

Declaration:

type switch (*on-value*, *off-value*) = new state (*on-value*, *off-value*);

Parameters: *on-value*, *off-value* are distinct value-identifiers.

Values: Each instance of a Switch datatype has two named values, each of which is designated by a unique *value-identifier* and is distinct from the other value of the datatype.

Properties: unordered, exact, non-numeric.

Operations: Equal from State; Invert

Invert(x: switch (*on-value*, *off-value*)): switch (*on-value*, *off-value*) is defined by:

x	Invert(x)
on-value	*off-value*
off-value	*on-value*

10.1.2 Cardinal

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

Declaration:

type cardinal = range (0..) of integer;

Parameters: none.

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

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

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

10.1.3 Bit string

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

Declaration:

type bitstring = new sequence of (bit);

Parameters: none.

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

Value-syntax:

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

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

The bit-string-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-literals in the bit-string-literal, then the value denoted is the sequence of length zero.

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

Operations: (Head, Tail, Append, Equal, Empty, IsEmpty) from Sequence (8.4.4),
InOrder is application-defined.

NOTES

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

2. That bitstring is ordered is presumed to be a useful property, and is therefore specified, even though no standard for the In-Order function is appropriate.

10.1.4 Character string

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

Declaration:

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

Parameters: *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 | hyphen | special | apostrophe | space .

added-character = <not defined by this draft 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: ordered, exact, non-numeric, bounded below, unbounded above.

Operations: (Head, Tail, Append, Equal, Empty, IsEmpty) from Sequence (8.4.4),
InOrder is application-defined.

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 ordering 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.

10.1.5 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*);

Parameters: *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 $v \leq$ *modulus*.

Properties: unordered, exact, numeric.

Operations: Equal, Add, Multiply, Negate.

Equal(x, y: modulo(*modulus*)): boolean = integer.Equal(
integer.Remainder(x, *modulus*), integer.Remainder(y, *modulus*));

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*).

10.1.6 Currency

Description: Currency is a datatype representing monetary values exact to two decimal places. It is a generated datatype derived from a scaled datatype by limiting the operations.

Declaration:

type currency = new scaled (10, 2);

Parameters: none.

Values: all rational values which are integral multiples of 0.01.

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(10, *factor*), y:currency): currency, where $0 \leq$ *factor*, = scaled.Multiply(x,y).

10.1.7 Interval

Description: Interval 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 interval(*radix*: integer, *factor*: integer) = new scaled (*radix*, *factor*);

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

Values: all values which are integral multiples of $radix^{(-factpr)}$.

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: interval(*r,f*)): interval(*r,f*) = scaled.Multiply(x,y).

EXAMPLE – interval(10, 3) is the datatype of elapsed time in milliseconds.

10.1.8 Octet

Description: Octet is the datatype of arrays of exactly 8 binary digits, as used for private encodings.

Declaration:

type octet = array (1..8) of (bit);

Parameters: none.

Values: Each value of datatype Octet is an indexable sequence of 8 bit-values, i.e. 0s and 1s. The value-space comprises all 256 such values.

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

Operations: (Equal, Select, Replace) from Array.

NOTE – Octet is a common datatype in communications protocols.

10.1.9 Private

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

Declaration:

type private(*size*: cardinal) = new array (1..*size*) of (bit);

Parameters: *Size* 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: Object-identifier 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 8824:1989).

Declaration:

type object_identifier = new sequence of (object_identifier_component): size(1..*);

type object_identifier_component = new integer: range(0..*);

Parameters: none.

Values: The value space of datatype object_identifier_component is isomorphic to the cardinal numbers (10.1.2), but the meaning of each value is determined by its position in an object-identifier value.

The value-space of datatype object-identifier comprises all non-empty finite sequences of object-identifier-component values. The meaning of each object_identifier_component value within the object-identifier value is determined by the sequence of values preceding it, as provided by ISO 8824:1989 (Abstract Syntax Notation One). The sequence constituting a single value of datatype object-identifier uniquely identifies an object.

Value syntax:

object-identifier-value = "{" object-identifier-component-list "}" .
object-identifier-component-list = object-identifier-component-value
{ object-identifier-component-value } .
object-identifier-component-value = nameform | numberform | nameandnumberform .
nameform = identifier .
numberform = number .
nameandnumberform = identifier "(" numberform ")" .

An *object-identifier-value* denotes a value of datatype object-identifier. An *object-identifier-component-value* denotes a value of datatype object-identifier-component. A value-identifier appearing in the numberform shall refer to a non-negative integer value. In all cases, the value denoted is that prescribed by ISO 8824:1989 Abstract Syntax Notation One.

Properties: unordered, exact, non-numeric.

Operations on object-identifier-component: Equal from Integer;

Operations on object-identifier: Append from Sequence;
Equal, Length, Detach, Last.

Length(x: object-identifier): integer is the number of object-identifier-component values in the sequence x;

Detach(x: object-identifier): object-identifier, where Length(x) > 1, is the object-identifier formed by removing the last object-identifier-component from the sequence x;

Last(x: object-identifier): object-identifier-component is the the relative object-identifier-component value which is the last element of the sequence x;

Equal(x,y: object-identifier): boolean =
if Not(Length(x) = Length(y)) then false,
else if Not(object-identifier-component.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 object-identifier. Therefore, Length and Equal are defined here, although they could be derived by using the Sequence operations.

2. Object-Identifier 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 (object-identifier-component).

10.1.11 Distinguished-Name

Description: Distinguished Name is the datatype of the external names of objects in ISO ??? Directory Services for OSI.

Declaration:

type distinguished-name-component = new characterstring(VisibleString);

type distinguished-name = sequence of (distinguished-name-component);
value ROOT: distinguished-name = ();

Parameters: none.

Values: The value space of datatype distinguished-name-component comprises all characterstrings which meet the syntactic requirements for component names of Distinguished-Names in ISO ??? Directory Services for OSI.

The value space of datatype distinguished-name comprises all non-empty finite sequences of distinguished-name-components, in which each distinguished-name-component has a distinct meaning and that meaning depends entirely on the sequence of distinguished-name-components which precedes it. The meaning of the first distinguished-name-component in the sequence is dependent on the context in which it appears. In the most global context, the meaning is defined by ISO ???. The value space is properly restricted at any given time to those sequences which actually identify an object in the available OSI directories.

Properties: unordered, exact, non-numeric.

Operations on distinguished-name-component: Equal from CharacterString.

Operations on distinguished-name: Append from Sequence, IsRoot, Equal, Detach, Last.

Detach(x: distinguished-name): distinguished-name is:
if there is only one distinguished-name-component in x, then the distinguished-name value ROOT; else the value formed by removing the last distinguished-name-component from the sequence x;

IsRoot(x: distinguished-name): boolean = sequence.IsEmpty(x);

Last(x: distinguished-name): distinguished-name-component, where Not(IsRoot(x)), is the value which is the last element of the sequence x;

Equal(x,y: distinguished-name): boolean =
if And(IsRoot(x), IsRoot(y)), then true;
else if Or(IsRoot(x), IsRoot(y)), then false;
else if distinguished-name-component.Equal(Last(x), Last(y)) then
Equal(Detach(x), Detach(y));
else false.

NOTE – Distinguished-Name 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 (distinguished-name-component).

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 component-datatypes and other parameters 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:

- ordered or unordered,
- numeric or non-numeric,
- approximate or exact,
- if ordered, bounded or unbounded.

When the generator generates an aggregate datatype, the aggregate properties described in clause 6.8 are also specified.

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

10.2.1 Stack

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

Declaration:

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

Components: *element* may be any datatype.

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

Properties: non-numeric, unordered, exact if and only if the *element* datatype is exact; aggregate properties from Sequence.

Operations: (IsEmpty, Equal, Empty) from List; 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;
the aggregate properties are those of Sequence.

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).

11. Support of Datatypes

An information processing entity is said to *support* a LI datatype if its mapping of that datatype onto some internal datatype (see Clause 12) preserves the properties of that datatype as herein defined.

11.1 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.

11.2 Support of ordering and bounds

For a mapping to preserve the ordering property, the ordering defined on the internal datatype shall be consistent with the ordering 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, $a' \leq b'$.

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.

11.3 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 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, the following shall hold:

Let *C* be the LI datatype and *D* be the corresponding internal datatype. Let M be the inward mapping from the value space of *C* into the value space of *D*, and let $\overline{M}$ be the reverse inward mapping (see 12.3) from *D* into *C*. Then:

i) For any two values $v_1 \neq v_2$ in *C*, $M(v_1) \neq M(v_2)$, i.e. every value which is distinguishable in *C* must be distinguishable in *D*.

ii) If, for any two values $v_1 \neq v_2$ in *D*, $\overline{M}(v_1) = \overline{M}(v_2)$, then for all values x in *D* such that $| v_1 - x | < | v_1 - v_2 |$, $\overline{M}(x) = \overline{M}(v_1)$.

iii) If, for any two values v_1 and v_2 in *D*, $\overline{M}(v_1) \neq \overline{M}(v_2)$, then for all values x in *D* such that $| v_1 - v_2 | \leq | v_1 - x |$, $\overline{M}(x) \neq \overline{M}(v_1)$.

NOTE – the above rules permit the internal datatype to have *more values* than the LI datatype, i.e. a finer degree of approximation, as long as the mapping maintains consistency in the approximation.

For an outward mapping to preserve the approximate property, the mapping shall be 1-to-1 and onto.

11.5 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 draft International Standard.

12. 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 draft 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 draft 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 draft International Standard, but may be a part of standards which refer to this draft International Standard.

12.1 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 draft 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 draft 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 component datatypes will be implicitly mapped to the LI datatype generated by application of the mapped datatype generator to the mapped component 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.

12.2 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 parameters 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 component datatypes of any LI datatype generator which exclude corresponding classes of generated datatypes from the mapping. The same LI datatype generator with different component 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, 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 components 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 component 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 must 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.

12.3 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.

Annex A. [Informative]

Character-Set Standards

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

ISO 646:	ISO seven-bit coded character set for information interchange
ISO 2375:	Procedures for registration of escape sequences
ISO 4873:	ISO eight-bit code for information interchange - structure and rules for implementation
ISO 6093:	Representation of numeric values in character strings (defines character sets for numeric strings)
DIS 6862:	Mathematical coded character set for bibliographic information interchange
ISO 6937:	Coded character sets for text communication
ISO 8824:1989	Abstract Syntax Notation One (defines interchange character sets as explicit subsets of ISO 646)
ISO 8859:	Eight-bit single-byte coded graphic characters
DIS 10646:	Multiple-octet coded character set

(It is presumed that the identifiers for specific subsets of ISO 10646 which are defined in ISO 10646 are effectively "registered" thereby for purposes of reference within the Character and Character-string datatypes until a more formal character-set registration process is undertaken. That is,

{ iso standard 10646 latin },

for example, is a valid reference to the Latin character-set defined in ISO 10646.)

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 draft 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 draft 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 draft 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 draft 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 draft 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 draft 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 draft International Standard, this draft 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 draft 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 draft International Standard.

C.1 Size

Size is a type-attribute specifying the number (and type) of storage units required or alloted 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.

Size 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 6056 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 DIS 10967-1, a floating-point representation of the value v has the form:

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

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(*d*,1).

This representation can be characterized by five parameters:

radix and *precision*, from above;
emin and *emax*, with the requirement: $emin \le E \le emax$; 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

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

Discriminant applies only to choice datatypes or their generators.

C.7 Sequence

Sequence attributes describe the order of presentation of the component values of a value of an aggregate datatype. Their values and meaning depend on the aggregate datatype involved.

Sequence 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 draft International Standard is a subset of the syntax prescribed for the Interface Definition Notation (IDN) in the Language-Independent Procedure Calling standard. This annex contains the the complete IDN syntax, for reference only. A conforming IDN text is an *interface*, whereas a conforming LI datatype specification is a *type-specifier*. In addition, a mapping, as provided in Clause 12 may contain *declarations*.

In each production below, the numbers to the right of the production identify the page numbers on which the syntax rule appears in this draft International Standard.

Character-set productions:

```
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
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .                      13
special = "(" | ")" | "." | "," | ":" | ";" | "=" | "/" | "*" | "-" | "{" | "}" .       13
hyphen = "_" .                                                                           13
quote = "'" .                                                                            13
apostrophe = "'" .                                                                       13
space = " " .                                                                            13
escape = "!" .                                                                           13
added-character = <not defined by this draft International Standard> .          13, 21, 52
```

NOTE - Lexical 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 draft International Standard:

```
actual-parameter = value-expression | type-specifier .                                   48
actual-parameter-list = actual-parameter { "," actual-parameter } .                      48
aggregate-type = record-type | set-type | sequence-type | bag-type | array-type
        | table-type .                                                                   39
alternative = tag-value-list ":" alternative-type .                                      33
alternative-list = alternative { "," alternative } [ default-alternative ] .             33
alternative-type = type-specifier .                                                      33
alternative-value = independent-value .                                                  34
annotation = "<" annotation-label ":" annotation-text ">" .                              15
annotation-label = object-identifier-component-list .                                    15
annotation-text = <not defined by this draft International Standard> .                   15
any-character = bound-character | added-character | escape-character .            14, 21
argument = argument-name ":" argument-type .                                     36, 50
argument-declaration = direction argument .                                              36
argument-list = argument-declaration { "," argument-declaration } .                      36
argument-name = identifier .                                                     36,50
argument-type = type-specifier .                                                 36, 50
array-type = "array" "(" index-type-list ")" "of" "(" element-type ")" .                 44
array-value = value-list .                                                               45
bag-type = "bag" "of" "(" element-type ")" .                                             42
```

bag-value = empty-value | value-list . 42
base = type-specifier . 30, 31, 32
bit-literal = "0" | "1" . 23, 52
bit-string-literal = quote { bit-literal } quote . 52
bit-type = "bit" . 23
boolean-literal = "true" | "false" . 19
boolean-type = "boolean" . 19
bound-character = non-quote-character | quote . 13, 21
character-literal = "'" any-character "'" . 14, 21
character-name = identifier { identifier } . 14, 21, 52
character-type = "character" ["(" repertoire-list ")"] . 20
choice-type = "choice" "(" tag-type ")" "of" "(" alternative-list ")" . 33
choice-value = "(" tag-value ":" alternative-value ")" . 34
complex-literal = "(" real-part "," imaginary-part ")" . 28
complex-type = "complex" ["(" radix "," factor ")"] . 27
component-reference = field-identifier | "*" . 16
composite-value = choice-value | record-value | set-value | sequence-value
| bag-value | array-value | table-value . 16
default-alternative = "default" ":" alternative-type . 33
defined-type = type-identifier ["(" actual-parameter-list ")"] . 48
dependent-value = primary-dependency { "." component-reference } . 16
derived-value = string-literal | object-identifier-value . 16
digit-string = digit { digit } . 14
direction = "in" | "out" | "inout" . 36
element-type = type-specifier . 35, 41, 42, 43, 44, 46
element-value = independent-value . 46
empty-value = "(" ")" . 41, 42, 43
enumerated-literal = identifier . 20
enumerated-type = "enumerated" "(" enumerated-value-list ")" . 20
enumerated-value-list = enumerated-literal { "," enumerated-literal } . 20
escape-character = escape character-name escape . 14, 21, 52
excluding-subtype = base ":" "excluding" "(" select-list ")" . 31
explicit-subtype = base ":" "subtype" "(" subtype-definition ")" . 32
explicit-value = value-literal | composite-value | derived-value . 15
extended-literal = identifier . 31
extended-type = base ":" "plus" "(" extended-value-list ")" . 31
extended-value = extended-literal | parametric-value . 31
extended-value-list = extended-value { "," extended-value } . 31
factor = value-expression . 22, 25, 26, 27
field = field-identifier ":" field-type . 40
field-identifier = identifier . 40
field-list = field { "," field } . 40
field-type = type-specifier . 40
field-value = field-identifier ":" independent-value . 40
field-value-list = "(" field-value { "," field-value } ")" . 40
formal-parameter = formal-parameter-name ":" formal-parameter-type . 49

formal-parameter-list = formal-parameter { "," formal-parameter } . 49
formal-parameter-name = identifier . 49
formal-parameter-type = type-specifier | "type" . 49
generated-type = pointer-type | procedure-type | choice-type | aggregate-type . 33
identifier = letter { pseudo-letter } . 14
imaginary-part = real-literal . 28
independent-value = explicit-value | qualified-value | value-identifier | parametric-value . 15
index-lowerbound = value-expression . 44
index-type = type-specifier | index-lowerbound ".." index-upperbound . 44
index-type-list = index-type { "," index-type } . 44
index-upperbound = value-expression . 44
integer-literal = signed-number . 23
integer-type = "integer" . 23
key-list = key-type { "," key-type } . 46
key-type = type-specifier . 46
key-value-list = independent-value { "," independent-value } . 46
lowerbound = value-expression | "*" . 30, 31, 33
maximum-size = value-expression | "*" . 32
minimum-size = value-expression . 32
nameandnumberform = identifier "(" numberform ")" . 55
nameform = identifier . 55
non-quote-character = letter | digit | hyphen | special | apostrophe | space . 13, 21, 52
number = digit-string . 22, 23
numberform = number . 55
object-identifier-component-list = object-identifier-component-value
{ object-identifier-component-value } . 55
object-identifier-component-value = nameform | numberform | nameandnumberform . 55
object-identifier-value = "{" object-identifier-component-list "}" . 55
ordinal-literal = number . 22
ordinal-type = "ordinal" . 21
parametric-type = formal-parameter-name . 49
parametric-value = formal-parameter-name . 16, 49
pointer-literal = "null" . 35
pointer-type = "pointer" "to" "(" element-type ")" . 35
primary-dependency = field-identifier | argument-name . 16
primitive-type = boolean-type | state-type | enumerated-type | character-type
| ordinal-type | time-type | bit-type | integer-type | rational-type
| scaled-type | real-type | complex-type | void-type . 18
procedure-declaration = "procedure" procedure-identifier "(" [argument-list] ")"
["returns" "(" return-argument ")"]
["raises" "(" termination-list ")"] . 37
procedure-identifier = identifier . 37
procedure-type = "procedure" "(" [, argument-list] ")"
["returns" "(" return-argument ")"]
["raises" "(" termination-list ")"] . 36
pseudo-letter = letter | digit | hyphen . 14
qualified-value = type-specifier "." explicit-value . 15

radix = value-expression . 22, 25, 26, 27
range-subtype = base ":" "range" "(" select-range ")" . 30
rational-literal = signed-number ["/" number] . 24
rational-type = "rational" . 24
real-literal = integer-literal ["*" scale-factor] . 26
real-part = real-literal . 28
real-type = "real" ["(" radix "," factor ")"] . 26
record-type = "record" "(" field-list ")" . 40
record-value = field-value-list | value-list . 40
repertoire-identifier = value-expression . 20
repertoire-list = repertoire-identifier { "," repertoire-identifier } . 20
return-argument = [argument-name ":"] argument-type . 36
scaled-literal = integer-literal ["*" scale-factor] . 25
scaled-type = "scaled" "(" radix "," factor ")" . 25
scale-factor = number "^" signed-number . 25, 26
selecting-subtype = base ":" "selecting" "(" select-list ")" . 30
select-item = value-expression | select-range . 30, 31, 33
select-list = select-item { "," select-item } . 30, 31, 33
select-range = lowerbound ".." upperbound . 30, 31, 33
sequence-type = "sequence" "of" "(" element-type ")" . 43
sequence-value = empty-value | value-list . 43
set-type = "set" "of" "(" element-type ")" . 41
set-value = empty-value | value-list . 41
signed-number = ["-"] number . 23
size-subtype = base ":" "size" "(" minimum-size [".." maximum-size] ")" . 32
state-literal = identifier . 19
state-type = "state" "(" state-value-list ")" . 19
state-value-list = state-literal { "," state-literal } . 19
string-character = non-quote-character | added-character | escape-character . 15, 52
string-literal = quote { string-character } quote . 15, 52
subtype = range-subtype | selecting-subtype | excluding-subtype
| extended-type | size-subtype | explicit-subtype . 29
subtype-definition = type-specifier . 32
table-entry = key-value-list ":" element-value . 46
table-type = "table" "(" key-list ")" "of" "(" element-type ")" . 46
table-value = empty-value | "(" table-entry { "," table-entry } ")" . 46
tag-type = type-specifier . 33
tag-value = independent-value . 34
tag-value-list = select-list . 33
termination-argument-list = argument { "," argument } . 50
termination-declaration = "termination" termination-identifier
["(" termination-argument-list ")"] . 50
termination-identifier = identifier . 50
termination-list = termination-reference { "," termination-reference } . 36
termination-reference = identifier . 36
time-literal = digit-string ["." digit-string] . 22

```
time-type = "time" "(" time-unit [ "," radix "," factor ] ")" .                    22
time-unit = "year" | "month" | "day" | "hour" | "minute" | "second"
        | parametric-value .                                                       22
type-declaration = "type" type-identifier [ "(" formal-parameter-list ")" ]
        "=" [ "new" ] type-definition .                                            48
type-definition = type-specifier .                                                 49
type-identifier = identifier .                                                     48
type-specifier = primitive-type | subtype | generated-type | defined-type | parametric-type .  17
upperbound = value-expression | "*" .                                  30, 31, 33
value-declaration = "value" value-identifier ":" type-specifier "=" independent-value .  50
value-expression = independent-value | dependent-value .                           15
value-identifier = identifier .                                                    16, 50
value-list = "(" independent-value { "," independent-value } ")" .   40, 41, 42, 43, 45
value-literal = boolean-literal | state-literal | enumerated-literal | character-literal
        | ordinal-literal | time-literal | bit-literal | integer-literal | rational-literal
        | scaled-literal | real-literal | complex-literal | void-literal
        | extended-literal | pointer-literal .                                     15
void-literal = "nil" .                                                             29
void-type = "void" .                                                               29
```

Additional Productions of the IDN not used in this draft International Standard:

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

Annex E. [Informative]

Example Mapping

This annex contains a draft "inward" mapping from the LI datatypes into the programming language Pascal. These mappings are not definitive and may not be quite correct. The purpose of this annex is to exemplify the nature and content of a mapping.

E.1 LI Primitive Datatypes

E.1.1 Boolean

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

E.1.2 State

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

E.1.3 Enumerated

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

E.1.4 Character

A single character datatype of the form "character" or "character(*repertoire*)" maps to the Pascal character datatype. Pascal requires each implementation to define the character-set associated with the character datatype. The default character-set identified by the LI datatype syntax "character" is presumed to be that character-set, and *repertoire*, if present, must identify that character-set. 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 datatype "integer".

E.1.5 Ordinal

The LI datatype "ordinal:range(1..*maxint)* maps to the Pascal integer 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.

E.1.6 Time

The Time types are not mapped into Pascal.

E.1.7 Bit

Bit maps to the Pascal Integer 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:

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

E.1.8 Integer

The LI datatype "integer: range(*minint..maxint*)" maps to the Pascal integer type. Pascal requires each implementation to define the values of "minint" and "maxint". The integer datatype with the corresponding minimum and maximum values (and any subtype thereof) is mapped to the Pascal integer type, 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.

E.1.9 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(x: rational):rational
    var t:rational;
    begin
    end;

procedure Add(x,y: rational):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;
        Add := Reduce(t);
    end;

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

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

```
            end;
            Negate := t;
        end;
    procedure Reciprocal(x: rational):rational
        var t:rational;
        begin
            t[1] := x[2];
            t[2] := x[1];
            end;
            Reciprocal := t;
        end;
    procedure NonNegative(x:rational): boolean := (x[1] >= 0);
    procedure Equal(x, y: rational): boolean := ((x[1] = y[1]) and (x[2] = y[2]));
```

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

E.1.10 Scaled

The LI datatype Scaled(*r*, *f*): range(*minrf*..*maxrf*) maps to the Pascal type "integer", where *minrf* has the value *minint* • $r^{(-f)}$ and *maxrf* has the value *maxint* • $r^{(-f)}$. A scaled datatype with the corresponding minimum and maximum values (and any subtype thereof) is mapped to the Pascal integer type, with each scaled value M • $r^{(-f)}$ being mapped into the Pascal integer value M. 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 ** f; *)
procedure 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 := (mod(t, rtothef) > rtothef / 2);
        t := t / rtothef;
        if (round) then t := t + 1;
        if (negate) then t := -t;
        scaledMultiply := t;
    end;
procedure scaledDivide(x, y: scaled): scaled
    var
        t: scaled;
        round: boolean;
        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 (mod(x * rtothef, 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 can be 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 datatype:

```
type scaled = record (
    value: integer;
    radix: (0..maxint);
    factor: integer);
```

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.

E.1.11 Real

The LI datatypes "real: range(*rmin..rmax*)" and "real(*radix, precision*): range(*rmin..rmax*)" map to the Pascal real type, 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.

E.1.12 Complex

The LI datatypes "complex" and "complex(*radix, precision*)" map to the Extended Pascal complex type, only if 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 under the following mapping: Each LI Complex value is mapped into the Pascal complex value which is mathematically nearest it and if several values are equidistant then any may be chosen. All characterizing operations are conceptually preserved, although the implementation-defined arithmetic may affect the correctness of results.

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

NOTE — A complex datatype can be mapped into basic Pascal (ISO 7185) using the Pascal datatype:

```
type complex = record (
    realpart: real;
    imagpart: real);
```

and the definition of "characterizing operations" appropriate to the x + *i*y representation of a complex-number. This model defines a representable subset of the complex numbers, but its relationship to the *radix* and *precision* parameters of the LI datatypes is difficult to specify.

E.1.13 Void

The LI datatype "void" can be 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 E.2.1).

E.2 LI Generated Types

E.2.1 Choice

A choice datatype of the form:

choice (*tag-type*) of (
select-list1 : *alternative-type1*,
. . .
select-listN : *alternative-typeN*)

can be 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*)

only when the following conditions are met:

i) The *tag-type* maps to a Pascal ordinal type, as specified herein. The *mapped-tag-type* is then the ordinal type which is the image of the mapping.

ii) Each *alternative-type* can be mapped into a Pascal type, as specified herein. 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: "(*invented-field-identifier : mapped-alternative*)", where *mapped-alternative* is the image of the *alternative-type* under the mapping, and *invented-field-identifier* is any identifier which does not conflict with any other *field-name* 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* (or into the analogous abbreviated-list form in Extended Pascal). A *select-list* which is "default" is mapped into the *case-constant-list* "otherwise" in Extended Pascal, or into the *case-constant-list* containing the denotations of all corresponding values of the *mapped-tag-type* in basic Pascal.

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 operations IsType and Equal can be implemented by appropriate case-statements using the tag-variable as discriminant.

E.2.2 Pointer

A pointer datatype of the form "pointer to (*element-type*)" can be mapped into the Pascal type "pointer to *mapped-type*", only when the *element-type* maps to a Pascal type, as specified herein. 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" can be 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* "^".

E.2.3 Procedure

A procedure datatype of the form: "procedure (*arguments*)" can be mapped into a Pascal "procedure parameter specification", only when it appears as the datatype of a parameter (argument) to a procedure, and only if all of its *argument-types* can be mapped to Pascal types as provided herein. A procedure datatype of the form: "procedure (*arguments*) returns (*return-argument*)" can be mapped into a Pascal "function parameter specification", only when it appears as the datatype of a parameter (argument) to a procedure, and only if all of its *argument-types*, including that of the *return-argument*, can be mapped to Pascal types as provided herein.

The *argument-type* of the LI *return-argument* is mapped into the *result-type* of the Pascal functionparameter-specification. Every LI *argument* of the form "in *identifier* : *argument-type*" is mapped into a Pascal value-parameter-specification of the form "*identifier* : *mapped-type*" where *mapped-type* is the image of the *argument-type* under the mapping into Pascal. Every LI *argument* of the forms "inout *identifier* : *argument-type*" or "out *identifier* : *argument-type*" is mapped into a Pascal variable-parameter-specification of the form "var *identifier* : *mapped-type*" where *mapped-type* is the image of the *argument-type* under the mapping into Pascal.

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.

E.2.4 Record

A LI record datatype of the form: "record (*field-list*)" can be mapped into a Pascal record-type of the form: "record (*field-list*), only if all of its *field-types* can be mapped to Pascal types as provided herein. 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 herein.

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 supported in Pascal by the relation "=".

E.2.5 Set

A set datatype of the form "set of (*element-type*)" can be mapped into the Pascal type "set of *mapped-type*", only if the *element-type* maps to a Pascal ordinal-type, as specified herein, 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 herein.

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 into a linked

structure as a variant of Sequence (see E.2.7), by defining the characterizing operations specifically for that structure.

E.2.6 Bag

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

E.2.7 Sequence

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

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 herein, can be mapped into Pascal using the type:

```
type sequenceof type = record (
    next: pointer to sequenceof type;
    elementvalue: mapped-type);
```

Each member (value of *element-type*) of a value of the sequence datatype is mapped to a heap variable of the Pascal type sequenceof*type*, by mapping its value to the corresponding value of *mapped-type*, as specified herein, and placing that value in the field "elementvalue". The value of sequence datatype is then represented by a value of the type "pointer to sequenceof*type*", which is the pointer to the heap variable representing the first member, or "null" 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 "null".

All characterizing operations can be defined on this representation.

E.2.8 Array

An array datatype of the form "array (*index-list*) of (*element-type*)" can be 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 herein.
2) Each *index-type* in the *index-list* can be mapped into some Pascal ordinal-type *mapped-index-type*, as specified herein. 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 herein.

The Select operation is supported by Pascal indexing. The Replace operation is supported by assignment to the appropriate cell of an array variable. The Equal operation is the Pascal operation "=".

E.2.9 Table

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

Values of a table datatype of the form "table (*key-list*) of (*element-type*)", where the *element-type* maps to some Pascal type *mapped-element-type* and each *key-type* in the *key-list* maps to some Pascal type *mapped-key-type*, as specified herein, can be mapped into Pascal using the type:

```
type tablevalue = record (
    key1: mapped-key-type-1;
    . . .
    keyN: mapped-key-type-N;
    element: mapped-element-type);
```

and the structuring mechanism described for sequence datatypes in E.2.7. Each value of the table datatype has the corresponding key values assigned to the fields "key1", ..., "keyN", and the element value assigned to the "element" field. The value of the table datatype is then represented as a value of the type "sequence of (tablevalue)", by defining the characterizing operations on that structure.

E.3 LI Subtypes

E.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 herein.

E.3.2 Selecting

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

E.3.3 Excluding

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

E.3.4 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.

E.3.5 Size

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

E.3.6 Explicit

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

E.4 LI Defined Datatypes

Most of the defined datatypes in Clause 10 can be mapped into Pascal by simply mapping their *type-definitions*. The exceptions are described in this clause.

E.4.1 Bit-String

A bit-string datatype all of whose values are of a fixed constant length, i.e. bitstring : size(k), can be mapped into the Pascal datatype "packed array [1..k] of bit", where "bit" is defined as in E.1.7.

The characterizing operations Head and Tail are defined as follows:

```
procedure Head(x : packed array [1..k] of bit):bit;
    begin Head := x[1] end;

procedure Tail(x : packed array [1..k] of bit) : packed array [1.. (k-1)] of bit;
var
    i: integer;
    y: packed array [1.. (k-1)] of bit
begin
    for i := 1 to k-1 do
        y[i] := x[i+1];
    Tail := y;
```

```
end;
```

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

No other bitstring datatype can be mapped into a Pascal datatype directly, although it is possible to support values of the bitstring datatype by a "package" utilizing a complex data structure as in E.2.7, although more efficient structures for bit-string can be developed.

E.4.2 Character-String

A character-string datatype all of whose values are of a fixed constant length, i.e. characterstring : size(*k*), can be mapped into the Pascal datatype "packed array [1..*k*] of char".

The characterizing operations Head and Tail are defined as follows:

```
procedure Head(x : packed array [1..k] of char):char;
    begin Head := x[1] end;
procedure Tail(x : packed array [1..k] of char) : packed array [1.. (k-1)] of char;
var
    i: integer;
    y: packed array [1.. (k-1)] of char;
begin
    for i := 1 to k-1 do
        y[i] := x[i+1];
    Tail := y;
end;
```

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

No other characterstring datatype can be mapped into a Pascal datatype directly, although it is possible to support values of the characterstring datatype by a "package" utilizing a complex data structure as in E.2.7, although more efficient structures for character-string can be developed.

E.4.3 Octet

Octet can be mapped into any of:

i) the analogous Pascal datatype: type octet = packed array [1..8] of bit;

ii) the Pascal datatype: type octet = packed array [1..8] of boolean;

iii) the Pascal integer-type: type octet = 0..255;
in which the 8 binary digits are interpreted as an 8-place binary value.

The choice is largely a matter of intended usage and the nature of implementations. Mappings (i) and (ii) support the characterizing operations directly, except that in (ii) the bit results have to be derived from the boolean values via the Pascal function "ord()". Some implementations of (i) may be much less efficient than those of (ii), which is why (ii) is proposed. Either (i) or (ii) will generally be less efficient than (iii), unless Select and Replace are actually going to be used in the context; whereas (iii) will generally have a more efficient implementation for the manipulation of 8-bit information units, although the definitions of Select and Replace will be complex.

E.4.4 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..

E.4.5 Object-identifier

The object-identifier datatype cannot be mapped directly into Pascal. The datatype must be mapped into Pascal as follows:

The object-identifier-component type must be mapped into Pascal as:

```
type object_identifier_component = 0..maxint;
```

Object-identifier values must be mapped into Pascal using a data structure similar to that proposed for the sequence datatypes in E.2.7, wherein the *element-type* is object-identifier-component.

E.4.6 Distinguished-Name

The distinguished-name datatype cannot be mapped directly into Pascal. If necessary, this datatype can be mapped as follows:

The distinguished-name-component datatype can be mapped into Pascal by adopting a convention for the maximum length of a name-component and assuming trailing spaces are not significant. This gives:

```
type distinguished_name_component = packed array [1..name_component_max] of char;
```

where it is assumed that the implementation of the char-type contains the { iso standard 8824 type VisibleString } character-set. Since this is a subset of the ISO 646 character set, most implementations will support it, but there is no Pascal requirement for such support.

Values of the type distinguished-name must be mapped into Pascal using a data structure similar to that proposed for the sequence datatypes in E.2.7, wherein the *element-type* is distinguished_name_component.

E.5 Type-Declarations and Defined Datatypes

In Pascal two type-specifiers refer to the same datatype **only if** they are both identifiers and spelled identically. Type-specifiers which are not (simply) identifiers **always** refer to distinct datatypes, although those datatypes may be "compatible" in many cases. 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.

E.5.1 Renaming declarations

This concept is not supported in Pascal. A datatype declaration in Pascal is effectively a "new" datatype declaration in all cases.

E.5.2 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 herein. 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 datatype, by inventing a unique name and mapping the *type-definition* after making lexical substitutions for the parameter values.

E.5.3 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 datatype, by inventing a unique name and mapping the *type-definition* after making lexical substitutions for the parameter values.

Annex F. [Informative]

Resolved Issues

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

Issue 1. 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.

Issue 2. Should the LI datatypes provide axiomatic datatype definitions?

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

Issue 3. How many characterizing operations are enough?

There was 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 4. 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 Bit into Integer and Integer into Rational, 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 5. Should implementations be required to support the characterizing operations?

The purpose of considering operations in this draft 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.

Issue 6. Is InOrder necessary? Does the standard need to define an ordering operation?

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

Issue 7. How much of the concept "mapping onto the LI datatypes" should be standardized?

Consensus is that formal requirements for *indirect compliance* are necessary to relate language standards to language-independent specifications. The mapping is a necessary part of the concept of indirect compliance and therefore a necessary part of this standard. There was 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 8. 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, as this seems most clearly to be the fundamental numeric datatype - the Integer Modulo 2 datatype which is conveyed by the term "binary digit". The datatype Range (0,1) of Integer is different. Add (1,1) produces different results in the two datatypes. Invert is defined and meaningful on Bit; it is not defined or meaningful on Integer and consequently not on Range (x,y) of Integer. The datatype Boolean is mathematically equivalent to Bit, in that astute identification of the Xor and And operations produces the same finite field. But semantically, Boolean is not a numeric datatype and has only operations associated with the logic notions true and false, while Bit is a numeric datatype and has numeric operations. The datatype Switch is none of the above. It is a State datatype, which has no native operations, on which the Invert operation, but neither numeric nor logical operations, are defined.

Issue 9. 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 10. Is Null a value of multiple types, as in SQL2, or a datatype itself?

Null is not a value of every type (or of many types). Every value of type Integer, for example, can be compared with zero. Is Null < 0? Is Null = 0? Allowing such a comparison is clearly inappropriate. Null 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".

Issue 11. Is Undefined the same as Null or Void?

There was consensus that Undefined is *not* a datatype, at least not one that has any useful properties. Null, on the other hand, is needed to model the empty variant in Pascal and Ada (and possibly the Null of ASN.1) and certain other places where a datatype is syntactically or semantically required but no (other) datatype is appropriate. This datatype is re-

tained and has been renamed Void, to avoid confusion with "null values" in SQL and the null pointer value in other languages, which do not have these semantics.

Issue 12. 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.

Issue 13. What is the proper model of Pointer datatypes?

i) 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 (undefined) to (object of element type), in which the usage of the pointer determines the other object 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.

ii) Is Pointer a primitive datatype or a generated datatype?

There was 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 and Tables, but unlike these explicit mappings, the values in the domain - the pointer value-space - have no other semantics.

Issue 14. Must there be a characterizing operation which produces values of type Pointer to x?

After much debate on the merits of the Allocate and Associate operations, there was 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 15. 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 16. Should the base types of Set and Selecting 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. If this is permitted for "Selecting" datatypes, the same LI datatype as implemented by two machines might actually have non-isomorphic value spaces. Similarly, the values of members of a set-value must be clearly distinguishable, in order for the uniqueness constraint to be well-defined.

Issue 17. Should Cardinal or Unsigned be LI datatypes?

Cardinal 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 Cardinal.

Issue 18. What should be done about Modulo?

In various drafts, Modulo has been:

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

Issue 19. 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, columns) of (numbers) = 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 was:

a. The tensor datatype generator is adequately supported by generator-declaration, and could be added to Clause 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 was 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 20. Is the notion *file* adequately supported by the datatype-generators?

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.

The accepted notion of access mechanisms for common file organizations, however, map exactly onto the characterizing operations for the generators List, Array and Table.

For example, *sequential* organization maps onto List:

Open for Input is obtaining an object of datatype List of X in the first place;
Open for Output is Empty();
Read is Head();
Write is Append(); and
Close is releasing the object.

A sequential file which is simply a BitString or CharacterString is adequately modelled by those datatypes, which are themselves Lists. A sequential file which contains a single record type is List of Record(field1, field2, ..., fieldN); and a sequential file with multiple record datatypes is List of Choice(recordtype1, recordtype2, ..., recordtypeM). In a similar way, the FORTRAN/COBOL *relative* organization maps onto Array and the *indexed sequential* organization maps onto Table.

The notion which is not supported by the List generator is that a file can simultaneously be a sequence of some record-type(s) *and* a string of bits or bytes. Such a notion requires an appeal to the representation of the datatype, which is outside the scope of this standard.

Issue 21. Are Pragmata/Attributes appropriate in the standard?

The Scope of the project expressly says that representation will *not* be a part of the standard, but 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. Moreover, the datatypes of programming languages often have representation properties which are important in distinguishing "internal datatypes" and are therefore necessary for mappings. Moreover, representation attributes are only a fraction of the datatype annotation capabilities needed by procedure calling standards and applications. Some common mechanism is necessary, but it is consensus that it should not be a normative part of this draft International Standard.

Issue 22. Relationship of the LI Datatypes syntax to the LI Procedure Calls syntax.

There is consensus that both the LI Procedure Calling standard and the Remote Procedure Calling standard will require an "Interface Definition Notation" (IDN), and that this syntactic notation will necessarily include provision for datatyping of procedure parameters and other objects. There is further consensus that the syntactic notation of the LI Datatypes standard should be identical to a subset of the IDN.

Issue 23. Must there be a null value of every datatype Pointer to x?

While it may be possible to treat Pointer datatypes as choice(pointer to x, null), every programming language which supports the pointer datatypes supports null values of such a datatype. For consistency with all expected applications, "null" is made a value of pointer datatypes.

Issue 24. Is Array a variant of List?

No. The important characteristic of an Array is the mapping of the index types onto the element type, while List 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 list 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 25. 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 Lists or Arrays in that language. Others, such as LISP, APL and Pascal make the single character a primitive type.

Issue 26. Should Character and Character-string types be ordered?

The problem is that the accepted ordering of characters in a standard character-set may vary from nation to nation or from application to application, and the collating sequence for character-strings is clearly application-dependent. 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 6.)

Issue 27. 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 28. 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 in the CLID, although there may be strong constraints on the mappings to programming languages.

Issue 29. 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.

Issue 30. 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 draft 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 draft 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.

Issue 31. 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 32. Should some of the datatypes in Clause 8 be in Clause 10?

The question of whether Enumerated can be "derived from" State, or Bit from Boolean, 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.

Issue 33. 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 draft 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 CLID, especially where no other relevant standards exist. Inaddition, the possibility of direct compliance may encourage future software products, including new kinds of products, to use standard CLID datatypes directly rather than defining their own syntax and semantics and then performing the mapping.

SECOND

VOTE ON COMMITTEE DRAFT 10514	
Date of circulation 1993-01-05	Reference number ISO/TC1 /SC22 N1305
Closing date for voting 1993-04-16	

ISO/TC 1 /SC 22

Title

Programming Languages, their environments and system software interfaces

Secretariat Canada, SCC

Circulated to P-members of the committee for voting on registration of the draft as a DIS, in accordance with 2.4.3 of part 1 of the IEC/ISO Directives

Please send this form, duly completed, to the secretariat indicated above.

CD 11404

Title

Information Technology - Programming Languages, their environments and system software interfaces - Language-Independent Datatypes

☐ **We agree to the circulation of the draft as a DIS in accordance with 2.5.1 of part 1 of the IEC/ISO Directives**

☐ **We do not agree to the circulation of the draft as a DIS**
The reasons for our disagreement are the following (use a separate page as annex, if necessary)

P-member voting

Date Ter kennisneming Signature Ter kennisneming

FORM 8 (ISO)

590 E 89 11