W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes

W3C Candidate Recommendation 30 April 2009

This version:
http://www.w3.org/TR/2009/CR-xmlschema11-2-20090430/

Latest version:
http://www.w3.org/TR/xmlschema11-2/

Previous versions:
http://www.w3.org/TR/2009/WD-xmlschema11-2-20090130/
http://www.w3.org/TR/2008/WD-xmlschema11-2-20080620/
http://www.w3.org/TR/2006/WD-xmlschema11-2-20060217/
http://www.w3.org/TR/2006/WD-xmlschema11-2-20060116/
http://www.w3.org/TR/2005/WD-xmlschema11-2-20050224/
http://www.w3.org/TR/2004/WD-xmlschema11-2-20040716/

Editors (Version 1.1):
David Peterson, invited expert (SGML*Works!*) <davep@iit.edu>
Shudi (Sandy) Gao 高殊镝, IBM <sandygao@ca.ibm.com>
Ashok Malhotra, Oracle Corporation <ashokmalhotra@alum.mit.edu>
C. M. Sperberg-McQueen, Black Mesa Technologies LLC <cmsmcq@blackmesatech.com>
Henry S. Thompson, University of Edinburgh <ht@inf.ed.ac.uk>

Editors (Version 1.0):
Paul V. Biron, Kaiser Permanente, for Health Level Seven <paul@sparrow-hawk.org>
Ashok Malhotra, Oracle Corporation <ashokmalhotra@alum.mit.edu>

This document is also available in these non-normative formats: XML, XHTML with changes since version 1.0 marked, XHTML with changes since previous Working Draft marked, Independent copy of the schema for schema documents, Independent copy of the DTD for schema documents, and List of translations.

Abstract

XML Schema: Datatypes is part 2 of the specification of the XML Schema language. It defines facilities for defining datatypes to be used in XML Schemas as well as other XML specifications. The datatype language, which is itself represented in XML, provides a superset of the capabilities found in XML document type definitions (DTDs) for specifying datatypes on

elements and attributes.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This W3C Candidate Recommendation specifies W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. It is here made available for review by W3C members and the public. This version of this document was created on 30 April 2009.

For those primarily interested in the changes since version 1.0, the Changes since version 1.0 (§I) appendix, which summarizes both changes already made and also those in prospect, with links to the relevant sections of this draft, is the recommended starting point. An accompanying version of this document displays in color all changes to normative text since version 1.0; another shows changes since the previous Working Draft.

The major changes since version 1.0 include:

- Support for XML 1.1 has been added. It is now implementation defined whether datatypes dependent on definitions in [XML] and [Namespaces in XML] use the definitions as found in version 1.1 or version 1.0 of those specifications.
- A new primitive decimal type has been defined, which retains information about the precision of the value. This type is aligned with the floating-point decimal types which are included in [IEEE 754-2008].
- In order to align this specification with those being prepared by the XSL and XML Query Working Groups, a new datatype named anyAtomicType which serves as the base type definition for all primitive atomic datatypes has been introduced.
- The conceptual model of the date- and time-related types has been defined more formally.
- A more formal treatment of the fundamental facets of the primitive datatypes has been adopted.
- More formal definitions of the lexical space of most types have been provided, with detailed descriptions of the mappings from lexical representation to value and from value to ·canonical representation·.
- The validation rule Datatype Valid (§4.1.4) has been recast in more declarative form. A paraphrase of the constraint in procedural terms, which corrects some errors in the previous versions of this document, has been added as a note.
- The rules governing partial implementations of infinite datatypes have been clarified.
- Various changes have been made in order to align the relevant parts of this specification more closely with other relevant specifications, including especially the corresponding sections of [XSD 1.1 Part 1: Structures].

Changes since the previous public Working Draft include the following:

- To reduce confusion and avert a widespread misunderstanding, the normative references to various W3C specifications now state explicitly that while the reference describes the particular edition of a specification current at the time this specification is published, conforming implementations of this specification are not required to ignore later editions of the other specification but instead MAY support later editions, thus allowing users of this specification to benefit from corrections to other specifications on which this one depends.

- Schema Component Constraint enumeration facet value required for NOTATION (§3.3.20), which restricts the use of NOTATION to validate ·literals· without first enumerating a set of values, has been clarified.

- The use of the namespace whose URI is `http://www.w3.org/2001/XMLSchema-datatypes` continues to be defined. An earlier draft of this specification had introduced text deprecating that use; that text has been deleted. This change resolves issue 6522 Please un-deprecate the the namespace http://www.w3.org/2001/XMLSchema-datatypes, raised by John Cowan.

- The discussion of whitespace handling in whiteSpace (§4.3.6) makes clearer that when the value is **collapse**, ·literals· consisting solely of whitespace characters are reduced to the empty string; the earlier formulation has been misunderstood by some implementors.

- The value space of anyURI is now explicitly identified; this resolves issue 3264 xs:anyURI definition, raised by the W3C XML Query and XSL working groups.

- References to IEEE 754-1985 have been updated to refer to 754-2000 (resolves issue 6664).

- The description of the value space of precisionDecimal has been revised for better clarity; this resolves issue 3248.

- In the discussions of the built-in list datatypes, the display of facets which have a value for the datatype in question has been corrected; this resolves issue 6734 NMTOKENS IDREFS and ENTITIES should all have a "whiteSpace" facet. The wording used to introduce facets with values has also been revised to try to reduce confusion.

- The historical list of leap seconds given in earlier versions of this document has been removed (see issue 6554).

- The publication form of this document now includes a detailed prose description of the type hierarchy diagram in section Built-in Datatypes and Their Definitions (§3). We thank the W3C Web Accessibility Initiative's Protocols and Formats Working Group for their comments and assistance in this connection.

- Several other editorial corrections and improvements have been made.

Comments on this document should be made in W3C's public installation of Bugzilla, specifying "XML Schema" as the product. Instructions can be found at http://www.w3.org/XML/2006/01/public-bugzilla. If access to Bugzilla is not feasible, please send your comments to the W3C XML Schema comments mailing list, www-xml-schema-comments@w3.org (archive) and note explicitly that you have not made a Bugzilla entry for the comment. Each Bugzilla entry and email message should contain only one comment.

The Candidate Recommendation review period for this document extends until 3 August 2009; comments received after that date will be considered if time allows, but no guarantees can be offered.

Although feedback based on any aspect of this specification is welcome, there are certain aspects of the design presented herein for which the Working Group is particularly interested in feedback. These are designated 'priority feedback' aspects of the design, and identified as such in editorial notes at appropriate points in this draft. Any feature mentioned in a priority feedback note is a "feature at risk": the feature may be retained as is or dropped, depending on the feedback received from readers, schema authors, schema users, and implementors.

Publication as a Candidate Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

The W3C XML Schema Working Group intends to request advancement of this specification and publication as a Proposed Recommendation (possibly with editorial changes, and possibly removing features identified as being at risk) as soon after 3 August 2009 as the following conditions are met.

- A test suite is available which tests each required and optional feature of XSD 1.1.
- Each feature of the specification has been implemented successfully by at least two independent implementations.
- The Working Group has responded formally to all issues raised against this document during the Candidate Recommendation period.

At the time this Candidate Recommendation was published, no interoperability or implementation report had yet been prepared.

This document has been produced by the W3C XML Schema Working Group as part of the W3C XML Activity. The goals of the XML Schema language version 1.1 are discussed in the Requirements for XML Schema 1.1 document. The authors of this document are the members of the XML Schema Working Group. Different parts of this specification have different editors.

The English version of this specification is the only normative version. Information about translations of this document is available at http://www.w3.org/2003/03/Translations/byTechnology?technology=xmlschema.

Table of Contents

Appendices

---

1 Introduction

1.1 Introduction to Version 1.1

The Working Group has two main goals for this version of W3C XML Schema:

- Significant improvements in simplicity of design and clarity of exposition *without* loss of backward *or* forward compatibility;
- Provision of support for versioning of XML languages defined using the XML Schema specification, including the XML transfer syntax for schemas itself.

These goals are slightly in tension with one another -- the following summarizes the Working Group's strategic guidelines for changes between versions 1.0 and 1.1:

1. Add support for versioning (acknowledging that this *may* be slightly disruptive to the XML transfer syntax at the margins)
2. Allow bug fixes (unless in specific cases we decide that the fix is too disruptive for a point release)
3. Allow editorial changes
4. Allow design cleanup to change behavior in edge cases
5. Allow relatively non-disruptive changes to type hierarchy (to better support current and forthcoming international standards and W3C recommendations)
6. Allow design cleanup to change component structure (changes to functionality restricted to edge cases)
7. Do not allow any significant changes in functionality
8. Do not allow any changes to XML transfer syntax except those required by version control hooks and bug fixes

The overall aim as regards compatibility is that

- All schema documents conformant to version 1.0 of this specification should also conform to version 1.1, and should have the same validation behavior across 1.0 and 1.1 implementations (except possibly in edge cases and in the details of the resulting PSVI);
- The vast majority of schema documents conformant to version 1.1 of this specification should also conform to version 1.0, leaving aside any incompatibilities arising from support for versioning, and when they are conformant to version 1.0 (or are made conformant by the removal of versioning information), should have the same validation behavior across 1.0 and 1.1 implementations (again except possibly in edge cases and in the details of the resulting PSVI);

1.2 Purpose

The [XML] specification defines limited facilities for applying datatypes to document content in that documents may contain or refer to DTDs that assign types to elements and attributes. However, document authors, including authors of traditional *documents* and those transporting *data* in XML, often require a higher degree of type checking to ensure robustness in document understanding and data interchange.

The table below offers two typical examples of XML instances in which datatypes are implicit: the instance on the left represents a billing invoice, the instance on the right a memo or perhaps an email message in XML.

<table>
<tr><th>Data oriented</th><th>Document oriented</th></tr>
<tr><td><pre>
<invoice>
  <orderDate>1999-01-21</orderDate>
  <shipDate>1999-01-25</shipDate>
  <billingAddress>
   <name>Ashok Malhotra</name>
   <street>123 Microsoft Ave.</street>
   <city>Hawthorne</city>
   <state>NY</state>
   <zip>10532-0000</zip>
  </billingAddress>
  <voice>555-1234</voice>
  <fax>555-4321</fax>
</invoice>
</pre></td><td><pre>
<memo importance='high'
      date='1999-03-23'>
  <from>Paul V. Biron</from>
  <to>Ashok Malhotra</to>
  <subject>Latest draft</subject>
  <body>
    We need to discuss the latest
    draft <emph>immediately</emph>.
    Either email me at <email>
    mailto:paul.v.biron@kp.org</email>
    or call <phone>555-9876</phone>
  </body>
</memo>
</pre></td></tr>
</table>

The invoice contains several dates and telephone numbers, the postal abbreviation for a state (which comes from an enumerated list of sanctioned values), and a ZIP code (which takes a definable regular form). The memo contains many of the same types of information: a date, telephone number, email address and an "importance" value (from an enumerated list, such as "low", "medium" or "high"). Applications which process invoices and memos need to raise exceptions if something that was supposed to be a date or telephone number does not conform to the rules for valid dates or telephone numbers.

In both cases, validity constraints exist on the content of the instances that are not expressible in XML DTDs. The limited datatyping facilities in XML have prevented validating XML processors from supplying the rigorous type checking required in these situations. The result has been that individual applications writers have had to implement type checking in an ad hoc manner. This specification addresses the need of both document authors and applications writers for a robust, extensible datatype system for XML which could be incorporated into XML processors. As discussed below, these datatypes could be used in other XML-related standards as well.

1.3 Dependencies on Other Specifications

Other specifications on which this one depends are listed in References (§K).

This specification defines some datatypes which depend on definitions in [XML] and [Namespaces in XML]; those definitions, and therefore the datatypes based on them, vary between version 1.0 ([XML 1.0], [Namespaces in XML 1.0]) and version 1.1 ([XML], [Namespaces in XML]) of those specifications. In any given use of this specification, the choice of the 1.0 or the 1.1 definition of those datatypes is ·implementation-defined·.

Conforming implementations of this specification may provide either the 1.1-based datatypes or the 1.0-based datatypes, or both. If both are supported, the choice of which datatypes to use in a particular assessment episode SHOULD be under user control.

> **Note:** When this specification is used to check the datatype validity of XML input, implementations MAY provide the heuristic of using the 1.1 datatypes if the input is labeled as XML 1.1, and using the 1.0 datatypes if the input is labeled 1.0, but this heuristic SHOULD be subject to override by users, to support cases where users wish to accept XML 1.1 input but validate it using the 1.0 datatypes, or accept XML 1.0 input and validate it using the 1.1 datatypes.

This specification makes use of the EBNF notation used in the [XML] specification. Note that some constructs of the EBNF notation used here resemble the regular-expression syntax defined in this specification (Regular Expressions (§G)), but that they are not identical: there are differences. For a fuller description of the EBNF notation, see Section 6. Notation of the [XML] specification.

1.4 Requirements

The [XML Schema Requirements] document spells out concrete requirements to be fulfilled by this specification, which state that the XML Schema Language must:

1. provide for primitive data typing, including byte, date, integer, sequence, SQL and Java primitive datatypes, etc.;
2. define a type system that is adequate for import/export from database systems (e.g., relational, object, OLAP);
3. distinguish requirements relating to lexical data representation vs. those governing an underlying information set;
4. allow creation of user-defined datatypes, such as datatypes that are derived from existing datatypes and which may constrain certain of its properties (e.g., range, precision, length, format).

1.5 Scope

This specification defines datatypes that can be used in an XML Schema. These datatypes can be specified for element content that would be specified as #PCDATA and attribute values of various types in a DTD. It is the intention of this specification that it be usable outside of the context of XML Schemas for a wide range of other XML-related activities such as [XSL] and [RDF Schema].

1.6 Terminology

The terminology used to describe XML Schema Datatypes is defined in the body of this specification. The terms defined in the following list are used in building those definitions and in describing the actions of a datatype processor:

[Definition:] for compatibility

A feature of this specification included solely to ensure that schemas which use this feature remain compatible with [XML].

[Definition:] match

(Of strings or names:) Two strings or names being compared must be identical. Characters with multiple possible representations in ISO/IEC 10646 (e.g. characters with both precomposed and base+diacritic forms) match only if they have the same representation in both strings. No case folding is performed.

(Of strings and rules in the grammar:) A string matches a grammatical production if and only if it belongs to the language generated by that production.

[Definition:] MAY

Schemas, schema documents, and processors are permitted to but need not behave as described.

[Definition:] SHOULD

It is recommended that schemas, schema documents, and processors behave as described, but there can be valid reasons for them not to; it is important that the full implications be understood and carefully weighed before adopting behavior at variance with the recommendation.

[Definition:] MUST

(Of schemas and schema documents:) Schemas and documents are required to behave as described; otherwise they are in ·error·.

(Of processors:) Processors are required to behave as described.

[Definition:] MUST NOT

Schemas, schema documents and processors are forbidden to behave as described; schemas and documents which nevertheless do so are in ·error·.

[Definition:] error

A failure of a schema or schema document to conform to the rules of this specification.

Except as otherwise specified, processors MUST distinguish error-free (conforming) schemas and schema documents from those with errors; if a schema used in type-validation or a schema document used in constructing a schema is in error, processors MUST report the fact; if more than one is in error, it is ·implementation-dependent· whether more than one is reported as being in error. If more than one of the constraints given in this specification is violated, it is ·implementation-dependent· how many of the violations, and which, are reported.

Note: Failure of an XML element or attribute to be datatype-valid against a particular datatype in a particular schema is not in itself a failure to conform to this specification and thus, for purposes of this specification, not an error.

1.7 Constraints and Contributions

This specification provides three different kinds of normative statements about schema components, their representations in XML and their contribution to the schema-validation of information items:

[Definition:] Constraint on Schemas

Constraints on the schema components themselves, i.e. conditions components ·must· satisfy to be components at all. Largely to be found in Datatype components (§4).

[Definition:] Schema Representation Constraint

Constraints on the representation of schema components in XML. Some but not all of these are expressed in Schema for Schema Documents (Datatypes) (normative) (§A) and DTD for Datatype Definitions (non-normative) (§B).

[Definition:] Validation Rule

Constraints expressed by schema components which information items ·must· satisfy to be schema-valid. Largely to be found in Datatype components (§4).

2 Datatype System

This section describes the conceptual framework behind the datatype system defined in this specification. The framework has been influenced by the [ISO 11404] standard on language-independent datatypes as well as the datatypes for [SQL] and for programming languages such as Java.

The datatypes discussed in this specification are for the most part well known abstract concepts such as *integer* and *date*. It is not the place of this specification to thoroughly define these abstract concepts; many other publications provide excellent definitions. However, this specification will attempt to describe the abstract concepts well enough that they can be readily recognized and distinguished from other abstractions with which they may be confused.

Note: Only those operations and relations needed for schema processing are defined in this specification. Applications using these datatypes are generally expected to implement appropriate additional functions and/or relations to make the datatype generally useful. For example, the description herein of the float datatype does not define addition or multiplication, much less all of the operations defined for that datatype in [IEEE 754-2008] on which it is based. For some datatypes (e.g. language or anyURI) defined in part by reference to other specifications which impose constraints not part of the datatypes as defined here, applications may also wish to check that values conform to the requirements given in the current version of the relevant external specification.

2.1 Datatype

[Definition:] In this specification, a **datatype** has three properties:

- A ·value space·, which is a set of values.
- A ·lexical space·, which is a set of ·literals· used to denote the values.
- A small collection of *functions, relations, and procedures* associated with the datatype. Included are equality and (for some datatypes) order relations on the ·value space·, and a ·lexical mapping·, which is a mapping from the ·lexical space· into the ·value space·.

Note: This specification only defines the operations and relations needed for schema processing. The choice of terminology for describing/naming the datatypes is selected to guide users and implementers in how to expand the datatype to be generally useful—i.e., how to recognize the "real world" datatypes and their variants for which the datatypes defined herein are meant to be used for data interchange.

Along with the ·lexical mapping· it is often useful to have an inverse which provides a standard ·lexical representation· for each value. Such a ·canonical mapping· is not required for schema processing, but is described herein for the benefit of users of this specification, and other specifications which might find it useful to reference these descriptions normatively. For some datatypes, notably QName and NOTATION, the mapping from lexical representations to values is context-dependent; for these types, no ·canonical mapping· is defined.

Note: Where ·canonical mappings· are defined in this specification, they are defined for ·primitive· datatypes. When a datatype is derived using facets which directly constrain the ·value space·, then for each value eliminated from the ·value space·, the corresponding lexical representations are dropped from the lexical space. The ·canonical mapping· for such a datatype is a subset of the ·canonical mapping· for its ·primitive· type and provides a ·canonical representation· for each value remaining in the ·value space·.

The ·pattern· facet, on the other hand, and any other (·implementation-defined·) ·lexical· facets, restrict the ·lexical space· directly. When more than one lexical representation is provided for a given value, such facets may remove the ·canonical representation· while permitting a different lexical representation; in this case, the value remains in the ·value space· but has no ·canonical representation·. This specification provides no recourse in such situations. Applications are free to deal with it as they see fit.

Note: This specification sometimes uses the shorter form "type" where one might strictly speaking expect the longer form "datatype" (e.g. in the phrases "union type", "list type", "base type", "item type", etc. No systematic distinction is intended between the forms of these phrase with "type" and those with "datatype"; the two forms are used interchangeably.

The distinction between "datatype" and "simple type definition", by contrast, carries more information: the datatype is characterized by its ·value space·, ·lexical space·, ·lexical mapping·, etc., as just described, independently of the specific facets or other definitional mechanisms used in the simple type definition to describe that particular ·value space· or ·lexical space·. Different simple type definitions with different selections of facets can describe the same datatype.

2.2 Value space

[Definition:] The **value space** *of a datatype* is the set of values for that datatype. Associated with each value space are selected operations and relations necessary to permit proper schema processing. Each value in the value space of a ·primitive· or ·ordinary· datatype is denoted by one or more character strings in its ·lexical space·, according to ·the lexical mapping·; ·special· datatypes, by contrast, may include "ineffable" values not mapped to by any lexical representation. (If the mapping is restricted during a derivation in such a way that a value has no denotation, that value is dropped from the value space.)

The value spaces of datatypes are abstractions, and are defined in Built-in Datatypes and Their Definitions (§3) to the extent needed to clarify them for readers. For example, in defining the numerical datatypes, we assume some general numerical concepts such as number and integer are known. In many cases we provide references to other documents providing more complete definitions.

> **Note:** *The value spaces and the values therein are abstractions.* This specification does not prescribe any particular internal representations that must be used when implementing these datatypes. In some cases, there are references to other specifications which do prescribe specific internal representations; these specific internal representations must be used to comply with those other specifications, but need not be used to comply with this specification.
>
> In addition, other applications are expected to define additional appropriate operations and/or relations on these value spaces (e.g., addition and multiplication on the various numerical datatypes' value spaces), and are permitted where appropriate to even redefine the operations and relations defined within this specification, provided that *for schema processing the relations and operations used are those defined herein*.

The ·value space· of a datatype can be defined in one of the following ways:

- defined elsewhere axiomatically from fundamental notions (intensional definition) [see ·primitive·]
- enumerated outright from values of an already defined datatype (extensional definition) [see ·enumeration·]
- defined by restricting the ·value space· of an already defined datatype to a particular subset with a given set of properties [see derived]
- defined as a combination of values from one or more already defined ·value space·(s) by a specific construction procedure [see ·list· and ·union·]

The relations of *identity* and *equality* are required for each value space. An order relation is specified for some value spaces, but not all. A very few datatypes have other relations or operations prescribed for the purposes of this specification.

2.2.1 Identity

The identity relation is always defined. Every value space inherently has an identity relation. Two things are *identical* if and only if they are actually the same thing: i.e., if there is no way whatever to tell them apart.

> **Note:** This does not preclude implementing datatypes by using more than one *internal* representation for a given value, provided no mechanism inherent in the datatype implementation (i.e., other than bit-string-preserving "casting" of the datum to a different datatype) will distinguish between the two representations.

In the identity relation defined herein, values from different ·primitive· datatypes' ·value spaces· are made artificially distinct if they might otherwise be considered identical. For example, there is a number *two* in the decimal datatype and a number *two* in the float datatype. In the identity relation defined herein, these two values are considered distinct. Other applications making use of these datatypes may choose to consider values such as these identical, but for the view of ·primitive· datatypes' ·value spaces· used herein, they are distinct.

WARNING: Care must be taken when identifying values across distinct primitive datatypes. The ·literals· '`0.1`' and '`0.10000000009`' map to the same value in float (neither 0.1 nor 0.10000000009 is in the value space, and each literal is mapped to the nearest value, namely 0.100000001490116119384765625), but map to distinct values in decimal.

> **Note:** Datatypes ·constructed· by ·facet-based restriction· do not create new values; they define subsets of some ·primitive· datatype's ·value space·. A consequence of this fact is that the ·literals· '`+2`', treated as a decimal, '`+2`', treated as an integer, and '`+2`', treated as a byte, all denote the same value. They are not only equal but identical.

Given a list ***A*** and a list ***B***, ***A*** and ***B*** are the same list if they are the same sequence of atomic values. The necessary and sufficient conditions for this identity are that ***A*** and ***B*** have the same length and that the items of ***A*** are pairwise identical to the items of ***B***.

> **Note:** It is a consequence of the rule just given for list identity that there is only one empty list. An empty list declared as having ·item type· decimal and an empty list declared as having ·item type· string are not only equal but identical.

2.2.2 Equality

Each ·primitive· datatype has prescribed an equality relation for its value space. The equality relation for most datatypes is the identity relation. In the few cases where it is not, equality has been carefully defined so that for most operations of interest to the datatype, if two values are equal and one is substituted for the other as an argument to any of the operations, the results will always also be equal.

On the other hand, equality need not cover the entire value space of the datatype (though it usually does). In particular, NaN is not equal to itself in the precisionDecimal, float, and double datatypes.

This equality relation is used when making ·facet-based restrictions· by *enumeration*, when checking identity constraints (in the context of [XSD 1.1 Part 1: Structures]), when checking value constraints, and in conjunction with order when making ·facet-based restrictions· involving order, with the following exception: When processing XPath expressions as part of XML schema-validity assessment or otherwise testing membership in the ·value space· of a datatype whose derivation involves ·assertions·, equality (like all other relations) within those expressions is interpreted using the rules of XPath ([XPath 2.0]). All comparisons for "sameness" prescribed by this specification test for equality, not for identity.

> **Note:** In the prior version of this specification (1.0), equality was always identity. This has been changed to permit the datatypes defined herein to more closely match the "real world" datatypes for which they are intended to be used as transmission formats.
>
> For example, the float datatype has an equality which is not the identity (−0 = +0 , but they are not identical—although they *were* identical in the 1.0 version of this specification), and whose domain excludes one value, NaN, so that NaN ≠ NaN .
>
> For another example, the dateTime datatype previously lost any time-zone offset information in the ·lexical representation· as the value was converted to ·UTC·; now the time zone offset is retained and two values representing the same "moment in time" but with different remembered time zone offsets are now *equal* but not *identical*.

In the equality relation defined herein, values from different primitive data spaces are made artificially unequal even if they might otherwise be considered equal. For example, there is a number *two* in the decimal datatype and a number *two* in the float datatype. In the equality relation defined herein, these two values are considered unequal. Other applications making use of these datatypes may choose to consider values such as these equal; nonetheless, in the equality relation defined herein, they are unequal.

Two lists ***A*** and ***B*** are equal if and only if they have the same length and their items are pairwise equal. A list of length one containing a value ***V1*** and an atomic value ***V2*** are equal if and only if ***V1*** is equal to ***V2***.

For the purposes of this specification, there is one equality relation for all values of all datatypes (the union of the various datatype's individual equalities, if one consider relations to be sets of ordered pairs). The *equality* relation is denoted by '=' and its negation by '≠', each used as a binary infix predicate: $x = y$ and $x \neq y$. On the other hand, *identity* relationships are always described in words.

2.2.3 Order

For some datatypes, an order relation is prescribed for use in checking upper and lower bounds of the ·value space·. This order may be a *partial* order, which means that there may be values in the ·value space· which are neither equal, less-than, nor greater-than. Such value pairs are *incomparable*. In many cases, no order is prescribed; each pair of values is either equal or ·incomparable·. [Definition:] Two values that are neither equal, less-than, nor greater-than are **incomparable**. Two values that are not ·incomparable· are **comparable**.

The order relation is used in conjunction with equality when making ·facet-based restrictions· involving order. This is the only use of this order relation for schema processing. Of course, when processing XPath expressions as part of XML schema-validity assessment or otherwise testing membership in the ·value space· of a datatype whose derivation involves ·assertions·, order (like all other relations) within those expressions is interpreted using the rules of XPath ([XPath 2.0]).

In this specification, this less-than order relation is denoted by '<' (and its inverse by '>'), the weak order by '≤' (and its inverse by '≥'), and the resulting ·incomparable· relation by '<>', each used as a binary infix predicate: $x < y$, $x \leq y$, $x > y$, $x \geq y$, and $x <> y$.

Note: The weak order "less-than-or-equal" means "less-than" or "equal" *and one can tell which*. For example, the duration P1M (one month) is *not* less-than-or-equal P31D (thirty-one days) because P1M is not less than P31D, nor is P1M equal to P31D. Instead, P1M is ·incomparable· with P31D.) The formal definition of order for duration (duration (§3.3.7)) ensures that this is true.

For purposes of this specification, the value spaces of primitive datatypes are disjoint, even in cases where the abstractions they represent might be thought of as having values in common. In the order relations defined in this specification, values from different value spaces are ·incomparable·. For example, the numbers two and three are values in both the decimal datatype and the float datatype. In the order relation defined here, the two in the decimal datatype is not less than the three in the float datatype; the two values are incomparable. Other applications making use of these datatypes may choose to consider values such as these comparable.

Note: Comparison of values from different ·primitive· datatypes can sometimes be an error and sometimes not, depending on context.

When made for purposes of checking an enumeration constraint, such a comparison is not in itself an error, but since no two values from different ·primitive· ·value spaces· are equal, any comparison of ·incomparable· values will invariably be false.

Specifying an upper or lower bound which is of the wrong primitive datatype (and therefore ·incomparable· with the values of the datatype it is supposed to restrict) is, by contrast, always an error. It is a consequence of the rules for ·facet-based restriction· that in conforming simple type definitions, the values of upper and lower bounds, and enumerated values, MUST be drawn from the value space of the ·base type·, which necessarily means from the same ·primitive· datatype.

Comparison of ·incomparable· values in the context of an XPath expression (e.g. in an assertion or in the rules for conditional type assignment) can raise a dynamic error in the evaluation of the XPath expression; see [XQuery 1.0 and XPath 2.0 Functions and Operators] for details.

2.3 The Lexical Space and Lexical Mapping

[Definition:] The **lexical mapping** for a datatype is a prescribed relation which maps from the ·lexical space· of the datatype into its ·value space·.

[Definition:] The **lexical space** of a datatype is the prescribed set of strings which ·the lexical mapping· for that datatype maps to values of that datatype.

[Definition:] The members of the ·lexical space· are **lexical representations** of the values to which they are mapped.

Note: For the ·special· datatypes, the ·lexical mappings· defined here map from the ·lexical space· into, but not onto, the ·value space·. The ·value spaces· of the ·special· datatypes include "ineffable" values for which the ·lexical mappings· defined in this specification provide no lexical representation.

For the ·primitive· and ·ordinary· atomic datatypes, the ·lexical mapping· is a (total) function on the entire ·lexical space· *onto* (not merely *into*) the ·value space·: every member of the ·lexical space· maps into the ·value space·, and every value is mapped to by some member of the ·lexical space·.

For ·union· datatypes, the ·lexical mapping· is not necessarily a function, since the same ·literal· may map to different values in different member types. For ·list· datatypes, the ·lexical mapping· is a function if and only if the ·lexical mapping· of the list's ·item type· is a function.

[Definition:] A sequence of zero or more characters in the Universal Character Set (UCS) which may or may not prove upon inspection to be a member of the ·lexical space· of a given datatype and thus a ·lexical representation· of a given value in that datatype's ·value space·, is referred to as a **literal**. The term is used indifferently both for character sequences which are members of a particular ·lexical space· and for those which are not.

Note: One should be aware that in the context of XML schema-validity assessment, there are ·pre-lexical· transformations of the input character string (controlled by the whiteSpace facet and any implementation-defined ·pre-lexical· facets) which result in the intended ·literal·. Other systems utilizing this specification may or may not implement these transformations. If they do not, then input character strings that would have been transformed into correct lexical representations, when taken "raw", may not be correct ·lexical representations·.

Should a derivation be made using a derivation mechanism that removes ·lexical representations· from the·lexical space· to the extent that one or more values cease to have any ·lexical representation·, then those values are dropped from the ·value space·.

Note: This could happen by means of a pattern or other ·lexical· facet.

Conversely, should a derivation remove values then their ·lexical representations· are dropped from the ·lexical space· unless there is a facet value whose impact is defined to cause the otherwise-dropped ·lexical representation· to be mapped to another value instead.

Note: There are currently no facets with such an impact. There may be in the future.

For example, '100' and '1.0E2' are two different ·lexical representations· from the float datatype which both denote the same value. The datatype system defined in this specification provides mechanisms for schema designers to control the ·value space· and the corresponding set of acceptable ·lexical representations· of those values for a datatype.

2.3.1 Canonical Mapping

While the datatypes defined in this specification often have a single ·lexical representation· for each value (i.e., each value in the datatype's ·value space· is denoted by a single ·representation· in its ·lexical space·), this is not always the case. The example in the previous section shows two ·lexical representations· from the float datatype which denote the same value.

[Definition:] The **canonical mapping** is a prescribed subset of the inverse of a ·lexical mapping· which is one-to-one and whose domain (where possible) is the entire range of the ·lexical mapping· (the ·value space·). Thus a ·canonical mapping· selects one ·lexical representation· for each value in the ·value space·.

[Definition:] The **canonical representation** of a value in the ·value space· of a datatype is the ·lexical representation· associated with that value by the datatype's ·canonical mapping·.

·Canonical mappings· are not available for datatypes whose ·lexical mappings· are context dependent (i.e., mappings for which the value of a ·lexical representation· depends on the context in which it occurs, or for which a character string may or may not be a valid ·lexical representation· similarly depending on its context)

> **Note:** ·Canonical representations· are provided where feasible for the use of other applications; they are not required for schema processing itself. *A conforming schema processor implementation is not required to implement* ·canonical mappings·*.*

2.4 Datatype Distinctions

It is useful to categorize the datatypes defined in this specification along various dimensions, defining terms which can be used to characterize datatypes and the Simple Type Definitions which define them.

2.4.1 Atomic vs. List vs. Union Datatypes

First, we distinguish ·atomic·, ·list·, and ·union· datatypes.

[Definition:] An **atomic value** is an elementary value, not constructed from simpler values by any user-accessible means defined by this specification.

- [Definition:] **Atomic** datatypes are those whose ·value spaces· contain only ·atomic values·. **Atomic** datatypes are anyAtomicType and all datatypes ·derived· from it.
- [Definition:] **List** datatypes are those having values each of which consists of a finite-length (possibly empty) sequence of ·atomic values·. The values in a list are drawn from some ·atomic· datatype (or from a ·union· of ·atomic· datatypes), which is the ·item type· of the **list**.

  > **Note:** It is a consequence of constraints normatively specified elsewhere in this document that the ·item type· of a list MAY be any ·atomic· datatype, or any ·union· datatype whose ·transitive membership· consists solely of ·atomic· datatypes (so a ·list· of a ·union· of ·atomic· datatypes is possible, but not a ·list· of a ·union· of ·lists·). The ·item type· of a list MUST NOT itself be a list datatype.

- [Definition:] **Union** datatypes are (a) those whose ·value spaces·, ·lexical spaces·, and ·lexical mappings· are the union of the ·value spaces·, ·lexical spaces·, and ·lexical mappings· of one or more other datatypes, which are the ·member types· of the union, or (b) those derived by ·facet-based restriction· of another union datatype.

Note: It is a consequence of constraints normatively specified elsewhere in this document that any ·primitive· or ·ordinary· datatype MAY occur among the ·member types· of a ·union·. (In particular, ·union· datatypes may themselves be members of ·unions·, as may ·lists·.) The only prohibition is that no ·special· datatype may be a member of a ·union·.

For example, a single token which ·matches· Nmtoken from [XML] is in the value space of the ·atomic· datatype NMTOKEN, while a sequence of such tokens is in the value space of the ·list· datatype NMTOKENS.

2.4.1.1 Atomic Datatypes

An ·atomic· datatype has a ·value space· consisting of a set of "atomic" or elementary values.

Note: Atomic values are sometimes regarded, and described, as "not decomposable", but in fact the values in several datatypes defined here are described with internal structure, which is appealed to in checking whether particular values satisfy various constraints (e.g. upper and lower bounds on a datatype). Other specifications which use the datatypes defined here may define operations which attribute internal structure to values and expose or act upon that structure.

The ·lexical space· of an ·atomic· datatype is a set of ·literals· whose internal structure is specific to the datatype in question.

There is one ·special· ·atomic· datatype (anyAtomicType), and a number of ·primitive· ·atomic· datatypes which have anyAtomicType as their ·base type·. All other ·atomic· datatypes are derived either from one of the ·primitive· ·atomic· datatypes or from another ·ordinary· ·atomic· datatype. No ·user-defined· datatype MAY have anyAtomicType as its ·base type·.

2.4.1.2 List Datatypes

·List· datatypes are always ·constructed· from some other type; they are never ·primitive·. The ·value space· of a ·list· datatype is the set of finite-length sequences of zero or more ·atomic· values where each ·atomic· value is drawn from the ·value space· of the lists's ·item type· and has a ·lexical representation· containing no whitespace. The ·lexical space· of a ·list· datatype is a set of ·literals· each of which is a space-separated sequence of ·literals· of the ·item type·.

[Definition:] The ·atomic· or ·union· datatype that participates in the definition of a ·list· datatype is the **item type** of that ·list· datatype. If the ·item type· is a ·union·, each of its ·basic members· MUST be ·atomic·.

Example

```
<simpleType name='sizes'>
  <list itemType='decimal'/>
</simpleType>
```

```
<cerealSizes xsi:type='sizes'> 8 10.5 12 </cerealSizes>
```

A ·list· datatype can be ·constructed· from an ordinary or ·primitive· ·atomic· datatype whose ·lexical space· allows whitespace (such as string or anyURI) or a ·union· datatype any of whose {member type definitions}'s ·lexical space· allows space. Since ·list· items are separated at whitespace before the ·lexical representations· of the items are mapped to values, no whitespace will ever occur in the ·lexical representation· of a ·list· item, even when the item type would in principle allow it. For the same reason, when every possible ·lexical representation· of a given value in the ·value space· of the ·item type· includes whitespace, that value can never occur as an item in any value of the ·list· datatype.

Example

```
<simpleType name='listOfString'>
  <list itemType='string'/>
</simpleType>
```

```
<someElement xsi:type='listOfString'>
this is not list item 1
this is not list item 2
this is not list item 3
</someElement>
```

In the above example, the value of the *someElement* element is not a ·list· of ·length· 3; rather, it is a ·list· of ·length· 18.

When a datatype is derived by ·restricting· a ·list· datatype, the following ·constraining facets· apply:

- ·length·
- ·maxLength·
- ·minLength·
- ·enumeration·
- ·pattern·
- ·whiteSpace·
- ·assertions·

For each of ·length·, ·maxLength· and ·minLength·, the *length* is measured in number of list items. The value of ·whiteSpace· is fixed to the value ***collapse***.

For ·list· datatypes the ·lexical space· is composed of space-separated ·literals· of the ·item type·. Any ·pattern· specified when a new datatype is derived from a ·list· datatype applies to the members of the ·list· datatype's ·lexical space·, not to the members of the ·lexical space· of the ·item type·. Similarly, enumerated values are compared to the entire ·list·, not to individual list items, and assertions apply to the entire ·list· too. Lists are identical if and only if they have the same length and their items are pairwise identical; they are equal if and only if they have the same length and their items are pairwise equal. And a list of length one whose item is an atomic value ***V1*** is equal to an atomic value ***V2*** if and only if ***V1*** is equal to ***V2***.

Example

```
<xs:simpleType name='myList'>
        <xs:list itemType='xs:integer'/>
</xs:simpleType>
<xs:simpleType name='myRestrictedList'>
        <xs:restriction base='myList'>
                <xs:pattern value='123 (\d+\s)*456'/>
        </xs:restriction>
</xs:simpleType>
<someElement xsi:type='myRestrictedList'>123 456</someElement>
<someElement xsi:type='myRestrictedList'>123 987 456</someElement>
<someElement xsi:type='myRestrictedList'>123 987 567 456</someElement>
```

The ·canonical mapping· of a ·list· datatype maps each value onto the space-separated concatenation of the ·canonical representations· of all the items in the value (in order), using the ·canonical mapping· of the ·item type·.

2.4.1.3 Union datatypes

Union types may be defined in either of two ways. When a union type is ·constructed· by ·union·, its ·value space·, ·lexical space·, and ·lexical mapping· are the "ordered unions" of the ·value spaces·, ·lexical spaces·, and ·lexical mappings· of its ·member types·.

It will be observed that the ·lexical mapping· of a union, so defined, is not necessarily a function: a given ·literal· may map to one value or to several values of different ·primitive· datatypes, and it may be indeterminate which value is to be preferred in a particular context. When the datatypes defined here are used in the context of [XSD 1.1 Part 1: Structures], the `xsi:type` attribute defined by that specification in section xsi:type can be used to indicate which value a ·literal· which is the content of an element should map to. In other contexts, other rules (such as type coercion rules) may be employed to determine which value is to be used.

When a union type is defined by ·restricting· another ·union·, its ·value space·, ·lexical space·, and ·lexical mapping· are subsets of the ·value spaces·, ·lexical spaces·, and ·lexical mappings· of its ·base type·.

·Union· datatypes are always ·constructed· from other datatypes; they are never ·primitive·. Currently, there are no ·built-in· ·union· datatypes.

Example

A prototypical example of a ·union· type is the maxOccurs attribute on the element element in XML Schema itself: it is a union of nonNegativeInteger and an enumeration with the single member, the string "unbounded", as shown below.

```
    <attributeGroup name="occurs">
      <attribute name="minOccurs" type="nonNegativeInteger"
          use="optional" default="1"/>
      <attribute name="maxOccurs"use="optional" default="1">
        <simpleType>
          <union>
            <simpleType>
              <restriction base='nonNegativeInteger'/>
            </simpleType>
            <simpleType>
```

```
          <restriction base='string'>
            <enumeration value='unbounded'/>
          </restriction>
        </simpleType>
      </union>
    </simpleType>
  </attribute>
</attributeGroup>
```

Any number (zero or more) of ordinary or ·primitive· ·datatypes· can participate in a ·union· type.

[Definition:] The datatypes that participate in the definition of a ·union· datatype are known as the **member types** of that ·union· datatype.

[Definition:] The **transitive membership** of a ·union· is the set of its own ·member types·, and the ·member types· of its members, and so on. More formally, if ***U*** is a ·union·, then (a) its ·member types· are in the transitive membership of ***U***, and (b) for any datatypes ***T1*** and ***T2***, if ***T1*** is in the transitive membership of ***U*** and ***T2*** is one of the ·member types· of ***T1***, then ***T2*** is also in the transitive membership of ***U***.

The ·transitive membership· of a ·union· MUST NOT contain the ·union· itself, nor any datatype derived or ·constructed· from the ·union·.

[Definition:] Those members of the ·transitive membership· of a ·union· datatype ***U*** which are themselves not ·union· datatypes are the **basic members** of ***U***.

[Definition:] If a datatype ***M*** is in the ·transitive membership· of a ·union· datatype ***U***, but not one of ***U***'s ·member types·, then a sequence of one or more ·union· datatypes necessarily exists, such that the first is one of the ·member types· of ***U***, each is one of the ·member types· of its predecessor in the sequence, and ***M*** is one of the ·member types· of the last in the sequence. The ·union· datatypes in this sequence are said to **intervene** between ***M*** and ***U***. When ***U*** and ***M*** are given by the context, the datatypes in the sequence are referred to as the **intervening unions**. When ***M*** is one of the ·member types· of ***U***, the set of **intervening unions** is the empty set.

[Definition:] In a valid instance of any ·union·, the first of its members in order which accepts the instance as valid is the **active member type**. [Definition:] If the ·active member type· is itself a ·union·, one of *its* members will be *its* ·active member type·, and so on, until finally a ·basic (non-union) member· is reached. That ·basic member· is the **active basic member** of the union.

The order in which the ·member types· are specified in the definition (that is, in the case of datatypes defined in a schema document, the order of the <simpleType> children of the <union> element, or the order of the QNames in the `memberTypes` attribute) is significant. During validation, an element or attribute's value is validated against the ·member types· in the order in which they appear in the definition until a match is found. As noted above, the evaluation order can be overridden with the use of xsi:type.

Example

For example, given the definition below, the first instance of the <size> element validates correctly as an integer (§3.4.13), the second and third as string (§3.3.1).

```
<xsd:element name='size'>
  <xsd:simpleType>
    <xsd:union>
      <xsd:simpleType>
        <xsd:restriction base='integer'/>
      </xsd:simpleType>
      <xsd:simpleType>
        <xsd:restriction base='string'/>
      </xsd:simpleType>
    </xsd:union>
  </xsd:simpleType>
</xsd:element>
```

```
<size>1</size>
<size>large</size>
<size xsi:type='xsd:string'>1</size>
```

The ·canonical mapping· of a ·union· datatype maps each value onto the ·canonical representation· of that value obtained using the ·canonical mapping· of the first ·member type· in whose value space it lies.

When a datatype is derived by ·restricting· a ·union· datatype, the following ·constraining facets· apply:

- ·enumeration·
- ·pattern·
- ·assertions·

2.4.2 Special vs. Primitive vs. Ordinary Datatypes

Next, we distinguish ·special·, ·primitive·, and ·ordinary· (or ·constructed·) datatypes. Each datatype defined by or in accordance with this specification falls into exactly one of these categories.

- [Definition:] The **special** datatypes are anySimpleType and anyAtomicType. They are special by virtue of their position in the type hierarchy.
- [Definition:] **Primitive** datatypes are those datatypes that are not ·special· and are not defined in terms of other datatypes; they exist *ab initio*. All ·primitive· datatypes have anyAtomicType as their ·base type·, but their ·value· and ·lexical spaces· must be given in prose; they cannot be described as ·restrictions· of anyAtomicType by the application of particular ·constraining facets·.

  Note: As normatively specified elsewhere, conforming processors MUST support all the primitive datatypes defined in this specification; it is ·implementation-defined· whether other primitive datatypes are supported.
- [Definition:] **Ordinary** datatypes are all datatypes other than the ·special· and ·primitive·

datatypes. ·Ordinary· datatypes can be understood fully in terms of their Simple Type Definition and the properties of the datatypes from which they are ·constructed·.

For example, in this specification, float is a ·primitive· datatype based on a well-defined mathematical concept and not defined in terms of other datatypes, while integer is ·constructed· from the more general datatype decimal.

2.4.2.1 Facet-based Restriction

[Definition:] A datatype is defined by **facet-based restriction** of another datatype (its ·base type·), when values for zero or more ·constraining facets· are specified that serve to constrain its ·value space· and/or its ·lexical space· to a subset of those of the ·base type·. The ·base type· of a ·facet-based restriction· MUST be a ·primitive· or ·ordinary· datatype.

2.4.2.2 Construction by List

A ·list· datatype can be ·constructed· from another datatype (its ·item type·) by creating a ·value space· that consists of finite-length sequences of zero or more values of its ·item type·. Datatypes so ·constructed· have anySimpleType as their ·base type·. Note that since the ·value space· and ·lexical space· of any ·list· datatype are necessarily subsets of the ·value space· and ·lexical space· of anySimpleType, any datatype ·constructed· as a ·list· is a ·restriction· of its base type.

2.4.2.3 Construction by Union

One datatype can be ·constructed· from one or more datatypes by unioning their ·lexical mappings· and, consequently, their ·value spaces· and ·lexical spaces·. Datatypes so ·constructed· also have anySimpleType as their ·base type·. Note that since the ·value space· and ·lexical space· of any ·union· datatype are necessarily subsets of the ·value space· and ·lexical space· of anySimpleType, any datatype ·constructed· as a ·union· is a ·restriction· of its base type.

2.4.3 Definition, Derivation, Restriction, and Construction

Definition, derivation, restriction, and construction are conceptually distinct, although in practice they are frequently performed by the same mechanisms.

By 'definition' is meant the explicit identification of the relevant properties of a datatype, in particular its ·value space·, ·lexical space·, and ·lexical mapping·.

The properties of the ·special· and the standard ·primitive· datatypes are defined by this specification. A Simple Type Definition is present for each of these datatypes in every valid schema; it serves as a representation of the datatype, but by itself it does not capture all the relevant information and does not suffice (without knowledge of this specification) to *define* the datatype.

> **Note:** The properties of any ·implementation-defined· ·primitive· datatypes are given not here but in the documentation for the implementation in question.

For all other datatypes, a Simple Type Definition does suffice. The properties of an ·ordinary· datatype can be inferred from the datatype's Simple Type Definition and the properties of the ·base type·, ·item type· if any, and ·member types· if any. All ·ordinary· datatypes can be defined in this way.

By 'derivation' is meant the relation of a datatype to its ·base type·, or to the ·base type· of its ·base type·, and so on.

[Definition:] Every datatype other than anySimpleType is associated with another datatype, its **base type**. **Base types** can be ·special·, ·primitive·, or ·ordinary·.

[Definition:] A datatype ***T*** is **immediately derived** from another datatype ***X*** if and only if ***X*** is the ·base type· of ***T***.

> **Note:** The above does not preclude the Simple Type Definition for anySimpleType from having a value for its {base type definition}. (It does, and its value is anyType.)

More generally,
[Definition:] A datatype ***R*** is **derived** from another datatype ***B*** if and only if one of the following is true:

- ***B*** is the ·base type· of ***R***.
- There is some datatype ***X*** such that ***X*** is the ·base type· of ***R***, and ***X*** is derived from ***B***.

A datatype MUST NOT be derived from itself. That is, the base type relation must be acyclic.

It is a consequence of the above that every datatype other than anySimpleType is derived from anySimpleType.

Since each datatype has exactly one ·base type·, and every datatype other than anySimpleType is derived directly or indirectly from anySimpleType, it follows that the ·base type· relation arranges all simple types into a tree structure, which is conventionally referred to as the *derivation hierarchy*.

By 'restriction' is meant the definition of a datatype whose ·value space· and ·lexical space· are subsets of those of its ·base type·.

Formally, [Definition:] A datatype ***R*** is a **restriction** of another datatype ***B*** when

- the ·value space· of ***R*** is a subset of the ·value space· of ***B***, and
- the ·lexical space· of ***R*** is a subset of the ·lexical space· of ***B***.

Note that all three forms of datatype ·construction· produce ·restrictions· of the ·base type·: ·facet-based restriction· does so by means of ·constraining facets·, while ·construction· by ·list· or ·union· does so because those ·constructions· take anySimpleType as the ·base type·. It follows that all datatypes are ·restrictions· of anySimpleType. This specification provides no means by which a datatype may be defined so as to have a larger ·lexical space· or ·value space· than its ·base type·.

By 'construction' is meant the creation of a datatype by defining it in terms of another.

[Definition:] All ·ordinary· datatypes are defined in terms of, or **constructed** from, other datatypes, either by ·restricting· the ·value space· or ·lexical space· of a ·base type· using zero or more ·constraining facets· or by specifying the new datatype as a ·list· of items of some ·item type·, or by defining it as a ·union· of some specified sequence of ·member types·. These three forms of ·construction· are often called "·facet-based restriction·", "·construction· by ·list·", and "·construction· by ·union·", respectively. Datatypes so constructed may be understood fully (for purposes of a type system) in terms of (a) the properties of the datatype(s) from which they are constructed, and (b) their Simple Type Definition. This distinguishes ·ordinary· datatypes from the ·special· and ·primitive· datatypes, which can be understood only in the light of documentation (namely, their descriptions elsewhere in this specification, or, for ·implementation-defined· ·primitives·, in the appropriate implementation-specific documentation). All ·ordinary· datatypes are ·constructed·, and all ·constructed· datatypes are ·ordinary·.

2.4.4 Built-in vs. User-Defined Datatypes

- [Definition:] **Built-in** datatypes are those which are defined in this specification; they can be ·special·, ·primitive·, or ·ordinary· datatypes .
- [Definition:] **User-defined** datatypes are those datatypes that are defined by individual schema designers.

The ·built-in· datatypes are intended to be available automatically whenever this specification is implemented or used, whether by itself or embedded in a host language. In the language defined by [XSD 1.1 Part 1: Structures], the ·built-in· datatypes are automatically included in every valid schema. Other host languages SHOULD specify that all of the datatypes decribed here as built-ins are automatically available; they MAY specify that additional datatypes are also made available automatically.

> **Note:** ·Implementation-defined· datatypes, whether ·primitive· or ·ordinary·, may sometimes be included automatically in any schemas processed by that implementation; nevertheless, they are not built in to *every* schema, and are thus not included in the term 'built-in', as that term is used in this specification.

The mechanism for making ·user-defined· datatypes available for use is not defined in this specification; if ·user-defined· datatypes are to be available, some such mechanism MUST be specified by the host language.

[Definition:] A datatype which is not available for use is said to be **unknown**.

> **Note:** From the schema author's perspective, a reference to a datatype which proves to be ·unknown· might reflect any of the following causes, or others:
> 1 An error has been made in giving the name of the datatype.
> 2 The datatype is a ·user-defined· datatype which has not been made available using the means defined by the host language (e.g. because the appropriate schema document has not been consulted).
> 3 The datatype is an ·implementation-defined· ·primitive· datatype not supported by the implementation being used.
> 4 The datatype is an ·implementation-defined· ·ordinary· datatype which is made automatically available by some implementations, but not by the implementation being used.
> 5 The datatype is a ·user-defined· ·ordinary· datatype whose base type is ·unknown·
> From the point of view of the implementation, these cases are likely to be

indistinguishable.

Note: In the terminology of [XSD 1.1 Part 1: Structures], the datatypes here called ·unknown· are referred to as absent.

Conceptually there is no difference between the ·ordinary· ·built-in· datatypes included in this specification and the ·user-defined· datatypes which will be created by individual schema designers. The ·built-in· ·constructed· datatypes are those which are believed to be so common that if they were not defined in this specification many schema designers would end up reinventing them. Furthermore, including these ·constructed· datatypes in this specification serves to demonstrate the mechanics and utility of the datatype generation facilities of this specification.

3 Built-in Datatypes and Their Definitions

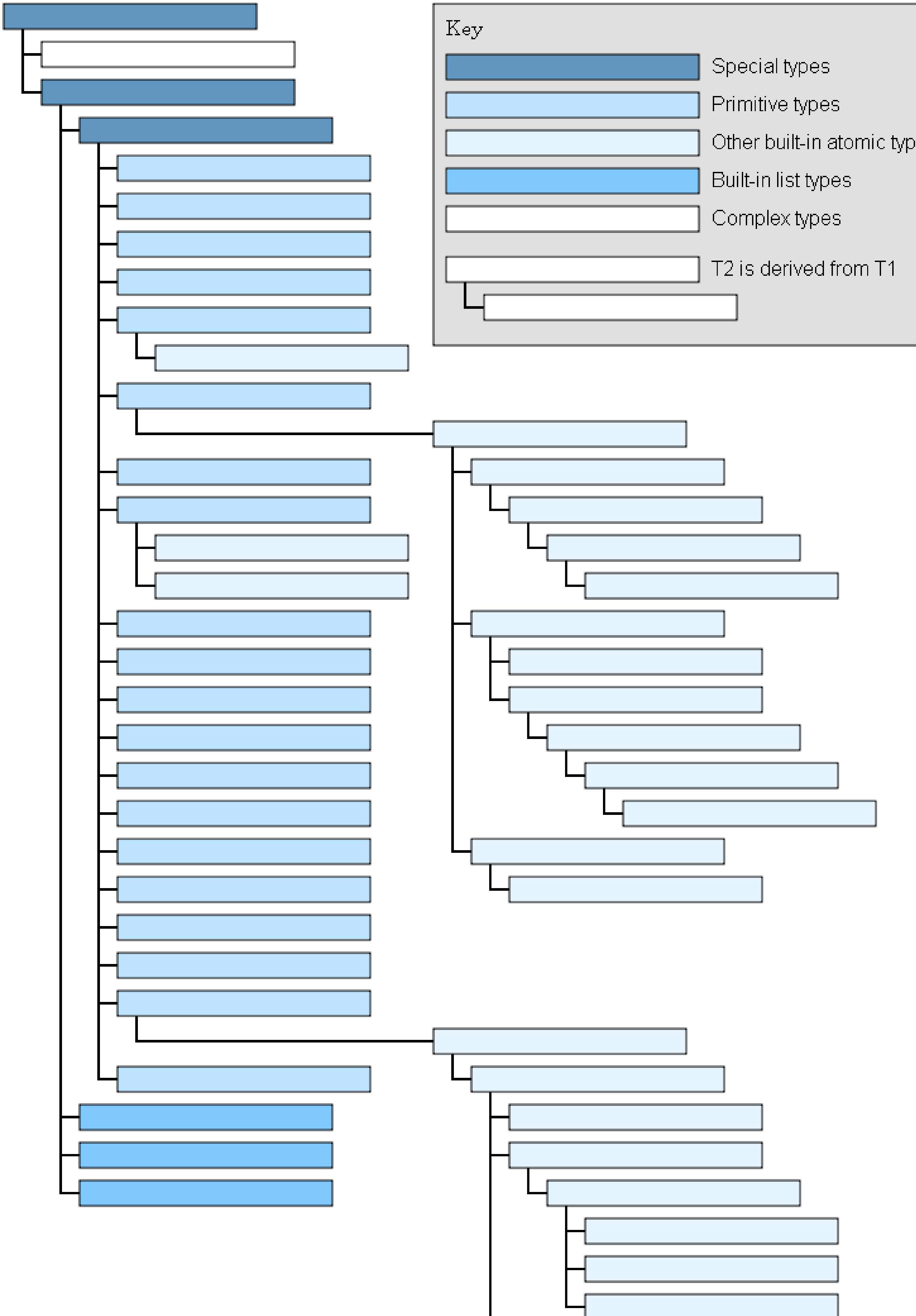
Key
Special types
Primitive types
Other built-in atomic typ
Built-in list types
Complex types
T2 is derived from T1

Diagram showing the derivation relations in the built-in type hierarchy. (A long description of the diagram is available separately.)

Each built-in datatype defined in this specification can be uniquely addressed via a URI Reference constructed as follows:

1. the base URI is the URI of the XML Schema namespace
2. the fragment identifier is the name of the datatype

For example, to address the int datatype, the URI is:

- `http://www.w3.org/2001/XMLSchema#int`

Additionally, each facet definition element can be uniquely addressed via a URI constructed as follows:

1. the base URI is the URI of the XML Schema namespace
2. the fragment identifier is the name of the facet

For example, to address the maxInclusive facet, the URI is:

- `http://www.w3.org/2001/XMLSchema#maxInclusive`

Additionally, each facet usage in a built-in Simple Type Definition can be uniquely addressed via a URI constructed as follows:

1. the base URI is the URI of the XML Schema namespace
2. the fragment identifier is the name of the Simple Type Definition, followed by a period ('.') followed by the name of the facet

For example, to address the usage of the maxInclusive facet in the definition of int, the URI is:

- `http://www.w3.org/2001/XMLSchema#int.maxInclusive`

3.1 Namespace considerations

The ·built-in· datatypes defined by this specification are designed to be used with the XML Schema definition language as well as other XML specifications. To facilitate usage within the XML Schema definition language, the ·built-in· datatypes in this specification have the namespace name:

- http://www.w3.org/2001/XMLSchema

To facilitate usage in specifications other than the XML Schema definition language, such as those that do not want to know anything about aspects of the XML Schema definition language other than the datatypes, each ·built-in· datatype is also defined in the namespace whose URI is:

- http://www.w3.org/2001/XMLSchema-datatypes

Each ·user-defined· datatype may also be associated with a target namespace. If it is constructed from a schema document, then its namespace is typically the target namespace of that schema document. (See XML Representation of Schemas in [XSD 1.1 Part 1: Structures].)

3.2 Special Built-in Datatypes

The two datatypes at the root of the hierarchy of simple types are anySimpleType and anyAtomicType.

3.2.1 anySimpleType

[Definition:] The definition of anySimpleType is a special ·restriction· of ***anyType***. Its ·lexical space· is the set of all sequences of Unicode characters, and its ·value space· includes all ·atomic values· and all finite-length lists of zero or more ·atomic values·.

For further details of anySimpleType and its representation as a Simple Type Definition, see Built-in Simple Type Definitions (§4.1.6).

3.2.1.1 Value space

The ·value space· of anySimpleType is the set of all ·atomic values· and of all finite-length lists of zero or more ·atomic values·.

> **Note:** It is a consequence of this definition, together with the definition of the ·lexical mapping· in the next section, that some values of this datatype have no ·lexical representation· using the ·lexical mappings· defined by this specification. That is, the "potential" ·value space· and the "effable" or "nameable" ·value space· diverge for this datatype. As far as this specification is concerned, there is no operational difference between the potential and effable ·value spaces· and the distinction is of mostly formal interest. Since some host languages for the type system defined here may allow means of construction values other than mapping from a ·lexical representation·, the difference may have practical importance in some contexts. In those contexts, the term ·value space· should unless otherwise qualified be taken to mean the potential ·value space·.

3.2.1.2 Lexical mapping

The ·lexical space· of anySimpleType is the set of all finite-length sequences of zero or more characters (as defined in [XML]) that ·match· the Char production from [XML]. This is equivalent to the union of the ·lexical spaces· of all ·primitive· and all possible ·ordinary· datatypes.

It is ·implementation-defined· whether an implementation of this specification supports the Char production from [XML], or that from [XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

The ·lexical mapping· of anySimpleType is the union of the ·lexical mappings· of all ·primitive· datatypes and all list datatypes. It will be noted that this mapping is not a function: a given ·literal· may map to one value or to several values of different ·primitive· datatypes, and it may be indeterminate which value is to be preferred in a particular context. When the datatypes defined here are used in the context of [XSD 1.1 Part 1: Structures], the `xsi:type` attribute defined by that specification in section xsi:type can be used to indicate which value a ·literal· which is the content of an element should map to. In other contexts, other rules (such as type coercion rules) may be employed to determine which value is to be used.

3.2.1.3 Facets

When a new datatype is defined by ·facet-based restriction·, anySimpleType MUST NOT be used as the ·base type·. So no ·constraining facets· are directly applicable to anySimpleType.

3.2.2 anyAtomicType

[Definition:] anyAtomicType is a special ·restriction· of anySimpleType. The ·value· and ·lexical spaces· of **anyAtomicType** are the unions of the ·value· and ·lexical spaces· of all the ·primitive· datatypes, and **anyAtomicType** is their ·base type·.

For further details of anyAtomicType and its representation as a Simple Type Definition, see Built-in Simple Type Definitions (§4.1.6).

3.2.2.1 Value space

The ·value space· of anyAtomicType is the union of the ·value spaces· of all the ·primitive· datatypes defined here or supplied as ·implementation-defined· ·primitives·.

3.2.2.2 Lexical mapping

The ·lexical space· of anyAtomicType is the set of all finite-length sequences of zero or more characters (as defined in [XML]) that ·match· the Char production from [XML]. This is equivalent to the union of the ·lexical spaces· of all ·primitive· datatypes.

It is ·implementation-defined· whether an implementation of this specification supports the Char production from [XML], or that from [XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

The ·lexical mapping· of anyAtomicType is the union of the ·lexical mappings· of all ·primitive· datatypes. It will be noted that this mapping is not a function: a given ·literal· may map to one value or to several values of different ·primitive· datatypes, and it may be indeterminate which value is to be preferred in a particular context. When the datatypes defined here are used in the context of [XSD 1.1 Part 1: Structures], the `xsi:type` attribute defined by that specification in section xsi:type can be used to indicate which value a ·literal· which is the content of an element should map to. In other contexts, other rules (such as type coercion rules) may be employed to determine which value is to be used.

3.2.2.3 Facets

When a new datatype is defined by ·facet-based restriction·, anyAtomicType MUST NOT be used as the ·base type·. So no ·constraining facets· are directly applicable to anyAtomicType.

3.3 Primitive Datatypes

The ·primitive· datatypes defined by this specification are described below. For each datatype, the ·value space· is described; the ·lexical space· is defined using an extended Backus Naur Format grammar (and in most cases also a regular expression using the regular expression language of Regular Expressions (§G)); ·constraining facets· which apply to the datatype are listed; and any datatypes ·constructed· from this datatype are specified.

Conforming processors MUST support the ·primitive· datatypes defined in this specification; it is ·implementation-defined· whether they support others. ·Primitive· datatypes may be added by revisions to this specification.

3.3.1 string

[Definition:] The **string** datatype represents character strings in XML.

> **Note:** Many human languages have writing systems that require child elements for

control of aspects such as bidirectional formatting or ruby annotation (see [Ruby] and Section 8.2.4 Overriding the bidirectional algorithm: the BDO element of [HTML 4.01]). Thus, string, as a simple type that can contain only characters but not child elements, is often not suitable for representing text. In such situations, a complex type that allows mixed content should be considered. For more information, see Section 5.5 Any Element, Any Attribute of [XML Schema Language: Part 0 Primer].

3.3.1.1 Value Space

The ·value space· of string is the set of finite-length sequences of zero or more characters (as defined in [XML]) that ·match· the Char production from [XML]. A character is an atomic unit of communication; it is not further specified except to note that every character has a corresponding Universal Character Set (UCS) code point, which is an integer.

It is ·implementation-defined· whether an implementation of this specification supports the Char production from [XML], or that from [XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

Equality for string is identity. No order is prescribed.

> **Note:** As noted in ordered, the fact that this specification does not specify an order relation for ·string· does not preclude other applications from treating strings as being ordered.

3.3.1.2 Lexical Mapping

The ·lexical space· of string is the set of finite-length sequences of zero or more characters (as defined in [XML]) that ·match· the Char production from [XML].

Lexical Space
stringRep ::= Char* /* (as defined in [XML]) */

It is ·implementation-defined· whether an implementation of this specification supports the Char production from [XML], or that from [XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

The ·lexical mapping· for string is *·stringLexicalMap·*, and the ·canonical mapping· is *·stringCanonicalMap·*; each is a subset of the identity function.

3.3.1.3 Facets

string has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- whiteSpace = `preserve`

Datatypes derived by restriction from string MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- pattern
- enumeration
- assertions

string has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.1.4 Derived datatypes

The following ·built-in· datatype is ·derived· from string

- normalizedString

3.3.2 boolean

[Definition:] **boolean** represents the values of two-valued logic.

3.3.2.1 Value Space

boolean has the ·value space· of two-valued logic: {***true***, ***false***}.

3.3.2.2 Lexical Mapping

boolean's lexical space is a set of four ·literals·:

Lexical Space
booleanRep ::= '`true`' | '`false`' | '`1`' | '`0`'

The ·lexical mapping· for boolean is *·booleanLexicalMap·*; the ·canonical mapping· is *·booleanCanonicalMap·*.

3.3.2.3 Facets

boolean and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from boolean MAY also specify values for the following ·constraining facets·:

- pattern
- assertions

boolean has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***finite***
- numeric = ***false***

3.3.3 decimal

[Definition:] **decimal** represents a subset of the real numbers, which can be represented by decimal numerals. The ·value space· of **decimal** is the set of numbers that can be obtained by dividing an integer by a non-negative power of ten, i.e., expressible as i / 10^n where i and n are integers and $n \geq 0$. Precision is not reflected in this value space; the number 2.0 is not distinct from the number 2.00. (The datatype precisionDecimal may be used for values in which precision is significant.) The order relation on **decimal** is the order relation on real numbers, restricted to this subset.

3.3.3.1 Lexical Mapping

decimal has a lexical representation consisting of a non-empty finite-length sequence of decimal digits (#x30–#x39) separated by a period as a decimal indicator. An optional leading sign is allowed. If the sign is omitted, "+" is assumed. Leading and trailing zeroes are optional. If the fractional part is zero, the period and following zero(es) can be omitted. For example: '`-1.23`', '`12678967.543233`', '`+100000.00`', '`210`'.

The decimal Lexical Representation
decimalLexicalRep ::= *decimalPtNumeral* | *noDecimalPtNumeral*

The lexical space of decimal is the set of lexical representations which match the grammar given above, or (equivalently) the regular expression

```
(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)
```

The mapping from lexical representations to values is the usual one for decimal numerals; it is given formally in *·decimalLexicalMap·*.

The definition of the ·canonical representation· has the effect of prohibiting certain options from the Lexical Mapping (§3.3.3.1). Specifically, for integers, the decimal point and fractional part are prohibited. For other values, the preceding optional "+" sign is prohibited. The decimal point is required. In all cases, leading and trailing zeroes are prohibited subject to the following: there must be at least one digit to the right and to the left of the decimal point which may be a zero.

The mapping from values to ·canonical representations· is given formally in *·decimalCanonicalMap·*.

3.3.3.2 Facets

decimal and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from decimal MAY also specify values for the following ·constraining facets·:

- totalDigits
- fractionDigits
- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

decimal has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***true***

3.3.3.3 Datatypes based on decimal

The following ·built-in· datatype is ·derived· from decimal

- integer

3.3.4 precisionDecimal

[Definition:] The **precisionDecimal** datatype represents the numeric value and (arithmetic) precision of decimal numbers which retain precision; it also includes values for positive and negative infinity and for "not a number", and it differentiates between "positive zero" and "negative zero". This datatype is introduced to provide a variant of decimal that closely corresponds to the floating-point decimal datatypes described by [IEEE 754-2008]. Precision of values is retained and values are included for two zeroes, two infinities, and not-a-number.

> **Note:** Users wishing to implement useful operations for this datatype (beyond the equality and order specified herein) are urged to consult [IEEE 754-2008].

Informally, the precision of a value is denoted by the number of decimal digits after the decimal point in its ·lexical representations·. The numbers 2 and 2.00, although numerically equal, have different precision (0 and 2 respectively). The precision of a value is derived from the ·lexical representation· using rules described in Lexical Mapping (§3.3.4.2). Values retain their precision, but that precision plays no part in any operations defined in this specification other than identity: specifically, it does not affect equality or ordering comparisons. Precision may play a role in arithmetic operations, but that is outside the scope of this specification.

Note: See the conformance note in Partial Implementation of Infinite Datatypes (§5.4), which applies to this datatype.

Editorial Note: Priority Feedback Request

The precisionDecimal datatype is a 'feature at risk'. It may be retained or dropped at the end of the Candidate Recommendation period. The determination of whether to retain it or remove it will depend both on the existence of two independent implementations in the context of this specification and on the degree of uptake of [IEEE 754-2008] in the industry. Possible outcomes include retention of precisionDecimal, dropping precisionDecimal, rearrangement of the type hierarchy in this area, and merger with the existing decimal datatype.

3.3.4.1 Value Space

Properties of precisionDecimal Values
·numericalValue·
a decimal number, ***positiveInfinity***, ***negativeInfinity*** or ***notANumber***
·arithmeticPrecision·
an integer or ***absent***; ***absent*** if and only if *·numericalValue·* is a ·special value·.
·sign·
positive, ***negative***, or ***absent***; must be ***positive*** if *·numericalValue·* is positive or ***positiveInfinity***, must be ***negative*** if *·numericalValue·* is negative or ***negativeInfinity***, must be ***absent*** if and only if *·numericalValue·* is ***notANumber***

Note: The *·sign·* property is redundant except when *·numericalValue·* is zero; in other cases, the *·sign·* value is fully determined by the *·numericalValue·* value.

Note: As explained below, '`NaN`' is the lexical representation of the precisionDecimal value whose *·numericalValue·* property has the ·special value· ***notANumber***. Accordingly, in English text we use 'NaN' to refer to that value. Similarly we use 'INF' and '−INF' to refer to the two values whose *·numericalValue·* properties have the ·special values· ***positiveInfinity*** and ***negativeInfinity***. These three precisionDecimal values are also informally called "not-a-number", "positive infinity", and "negative infinity". The latter two together are called "the infinities".

Equality and order for precisionDecimal are defined as follows:

- Two numerical precisionDecimal values are ordered (or equal) as their *·numericalValue·* values are ordered (or equal). (This means that two zeroes with different *·sign·*s are *equal*; negative zeroes are *not* ordered less than positive zeroes.)
- INF is equal only to itself, and is greater than −INF and all numerical precisionDecimal values.
- −INF is equal only to itself, and is less than INF and all numerical precisionDecimal values.
- NaN is incomparable with all values, *including itself*.

Note: Any value ·incomparable· with the value used for the four bounding facets (·minInclusive·, ·maxInclusive·, ·minExclusive·, and ·maxExclusive·) will be excluded from the resulting restricted ·value space·. In particular, when NaN is used as a facet value for a bounding facet, since no float values are ·comparable· with it, the result is a ·value space· that is empty. If any other value is used for a bounding facet, NaN will be excluded from the resulting restricted ·value space·; to add NaN back in requires union with the NaN-only space (which may be derived using a pattern).

Note: As specified elsewhere, enumerations test values for equality with one of the enumerated values. Because NaN ≠ NaN, including NaN in an enumeration does not have the effect of accepting NaNs as instances of the enumerated type; a union with a NaN-only datatype (which may be derived using the pattern "`NaN`") can be used instead.

3.3.4.2 Lexical Mapping

precisionDecimal's lexical space is the set of all decimal numerals with or without a decimal point, numerals in scientific (exponential) notation, and the character strings '`INF`', '`+INF`', '`-INF`', and '`NaN`'.

Lexical Space
pDecimalRep ::= *noDecimalPtNumeral* | *decimalPtNumeral* | *scientificNotationNumeral* | *numericalSpecialRep*

The *pDecimalRep* production is equivalent (after whitespace is removed) to the following regular expression:

```
(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)([Ee](\+|-)?[0-9]+)?
|(\+|-)?INF|NaN
```

The ·lexical mapping· for precisionDecimal is *·precisionDecimalLexicalMap·*. The ·canonical mapping· is *·precisionDecimalCanonicalMap·*.

For example, each of the ·lexical representations· shown below is followed by its corresponding value triple (*·numericalValue·*, *·arithmeticPrecision·*, and *·sign·*) and ·canonical representation·:

- '`3`' (3 , 0 , ***positive***) '`3`'

- '`3.00`' (3 , 2 , ***positive***) '`3.00`'
- '`03.00`' (3 , 2 , ***positive***) '`3.00`'
- '`300`' (300 , 0 , ***positive***) '`300`'
- '`3.00e2`' (300 , 0 , ***positive***) '`300`'
- '`3.0e2`' (300 , −1 , ***positive***) '`3.0E2`'
- '`30e1`' (300 , −1 , ***positive***) '`3.0E2`'
- '`.30e3`' (300 , −1 , ***positive***) '`3.0E2`'

Note that the last three examples not only show different ·lexical representations· for the same value, but are of particular interest because values with negative precision can *only* have ·lexical representations· in scientific notation.

3.3.4.3 Facets

precisionDecimal and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from precisionDecimal MAY also specify values for the following ·constraining facets·:

- totalDigits
- maxScale
- minScale
- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

precisionDecimal has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***true***

3.3.5 float

[Definition:] The **float** datatype is patterned after the IEEE single-precision 32-bit floating point datatype [IEEE 754-2008]. Its value space is a subset of the rational numbers. Floating point numbers are often used to approximate arbitrary real numbers.

3.3.5.1 Value Space

The ·value space· of float contains the non-zero numbers $m \times 2^{e}$, where m is an integer whose absolute value is less than 2^{24}, and e is an integer between −149 and 104, inclusive. In addition to these values, the ·value space· of float also contains the following ·special values·: ***positiveZero***, ***negativeZero***, ***positiveInfinity***, ***negativeInfinity***, and ***notANumber***.

> **Note:** As explained below, the ·lexical representation· of the float value ***notANumber*** is '`NaN`'. Accordingly, in English text we generally use 'NaN' to refer to that value. Similarly, we use 'INF' and '−INF' to refer to the two values ***positiveInfinity*** and ***negativeInfinity***, and '0' and '−0' to refer to ***positiveZero*** and ***negativeZero***.

Equality and order for float are defined as follows:

- Equality is identity, except that 0 = −0 (although they are not identical) and NaN ≠ NaN (although NaN is of course identical to itself).

  0 and −0 are thus equivalent for purposes of enumerations and identity constraints, as well as for minimum and maximum values.

- For the basic values, the order relation on float is the order relation for rational numbers. INF is greater than all other non-NaN values; −INF is less than all other non-NaN values. NaN is ·incomparable· with any value in the ·value space· including itself. 0 and −0 are greater than all the negative numbers and less than all the positive numbers.

> **Note:** Any value ·incomparable· with the value used for the four bounding facets (·minInclusive·, ·maxInclusive·, ·minExclusive·, and ·maxExclusive·) will be excluded from the resulting restricted ·value space·. In particular, when NaN is used as a facet value for a bounding facet, since no float values are ·comparable· with it, the result is a ·value space· that is empty. If any other value is used for a bounding facet, NaN will be excluded from the resulting restricted ·value space·; to add NaN back in requires union with the NaN-only space (which may be derived using a pattern).

> **Note:** The Schema 1.0 version of this datatype did not differentiate between 0 and −0 and NaN was equal to itself. The changes were made to make the datatype more closely mirror [IEEE 754-2008].

> **Note:** As specified elsewhere, enumerations test values for equality with one of the enumerated values. Because NaN ≠ NaN, including NaN in an enumeration does not have the effect of accepting NaNs as instances of the enumerated type; a union with a NaN-only datatype (which may be derived using the pattern "`NaN`") can be used instead.

3.3.5.2 Lexical Mapping

The ·lexical space· of float is the set of all decimal numerals with or without a decimal point, numerals in scientific (exponential) notation, and the ·literals· '`INF`', '`+INF`', '`-INF`', and '`NaN`'

Lexical Space

floatRep ::= *noDecimalPtNumeral* | *decimalPtNumeral* | *scientificNotationNumeral* | *numericalSpecialRep*

The *floatRep* production is equivalent to this regular expression (after whitespace is removed from the regular expression):

```
(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)([Ee](\+|-)?[0-9]+)?
|(\+|-)?INF|NaN
```

The float datatype is designed to implement for schema processing the single-precision floating-point datatype of [IEEE 754-2008]. That specification does not specify specific ·lexical representations·, but does prescribe requirements on any ·lexical mapping· used. Any ·lexical mapping· that maps the ·lexical space· just described onto the ·value space·, is a function, satisfies the requirements of [IEEE 754-2008], and correctly handles the mapping of the literals '`INF`', '`NaN`', etc., to the ·special values·, satisfies the conformance requirements of this specification.

Since IEEE allows some variation in rounding of values, processors conforming to this specification may exhibit some variation in their ·lexical mappings·.

The ·lexical mapping· *·floatLexicalMap·* is provided as an example of a simple algorithm that yields a conformant mapping, and that provides the most accurate rounding possible—and is thus useful for insuring inter-implementation reproducibility and inter-implementation round-tripping. The simple rounding algorithm used in *·floatLexicalMap·* may be more efficiently implemented using the algorithms of [Clinger, WD (1990)].

> **Note:** The Schema 1.0 version of this datatype did not permit rounding algorithms whose results differed from [Clinger, WD (1990)].

The ·canonical mapping· *·floatCanonicalMap·* is provided as an example of a mapping that does not produce unnecessarily long ·canonical representations·. Other algorithms which do not yield identical results for mapping from float values to character strings are permitted by [IEEE 754-2008].

3.3.5.3 Facets

float and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from float MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

float has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.3.6 double

[Definition:] The **double** datatype is patterned after the IEEE double-precision 64-bit floating point datatype [IEEE 754-2008]. Each floating point datatype has a value space that is a subset of the rational numbers. Floating point numbers are often used to approximate arbitrary real numbers.

> **Note:** The only significant differences between float and double are the three defining constants 53 (vs 24), −1074 (vs −149), and 971 (vs 104).

3.3.6.1 Value Space

The ·value space· of double contains the non-zero numbers $m \times 2^{e}$, where ***m*** is an integer whose absolute value is less than 2^{53}, and ***e*** is an integer between −1074 and 971, inclusive. In addition to these values, the ·value space· of double also contains the following ·special values·: ***positiveZero***, ***negativeZero***, ***positiveInfinity***, ***negativeInfinity***, and ***notANumber***.

> **Note:** As explained below, the ·lexical representation· of the double value ***notANumber*** is '`NaN`'. Accordingly, in English text we generally use 'NaN' to refer to that value. Similarly, we use 'INF' and '−INF' to refer to the two values ***positiveInfinity*** and ***negativeInfinity***, and '0' and '−0' to refer to ***positiveZero*** and ***negativeZero***.

Equality and order for double are defined as follows:

- Equality is identity, except that 0 = −0 (although they are not identical) and NaN ≠ NaN (although NaN is of course identical to itself).

  0 and −0 are thus equivalent for purposes of enumerations, identity constraints, and minimum and maximum values.

- For the basic values, the order relation on double is the order relation for rational numbers. INF is greater than all other non-NaN values; −INF is less than all other non-NaN values. NaN is ·incomparable· with any value in the ·value space· including itself. 0 and −0 are greater than all the negative numbers and less than all the positive numbers.

> **Note:** Any value ·incomparable· with the value used for the four bounding facets (·minInclusive·, ·maxInclusive·, ·minExclusive·, and ·maxExclusive·) will be excluded from the resulting restricted ·value space·. In particular, when NaN is used as a facet value for a bounding facet, since no double values are ·comparable· with it, the result is a ·value space· that is empty. If any other value is used for a bounding facet, NaN will be excluded from the resulting restricted ·value space·; to add NaN back in requires union with the NaN-only space (which may be derived with a pattern).

> **Note:** The Schema 1.0 version of this datatype did not differentiate between 0 and −0 and NaN was equal to itself. The changes were made to make the datatype more closely mirror [IEEE 754-2008].

> **Note:** As specified elsewhere, enumerations test values for equality with one of the enumerated values. Because NaN ≠ NaN, including NaN in an enumeration does not have the effect of accepting NaNs as instances of the enumerated type; a union with a NaN-only datatype (which may be derived using the pattern "`NaN`") can be used instead.

3.3.6.2 Lexical Mapping

The ·lexical space· of double is the set of all decimal numerals with or without a decimal point, numerals in scientific (exponential) notation, and the ·literals· '`INF`', '`+INF`', '`-INF`', and '`NaN`'

Lexical Space
doubleRep ::= *noDecimalPtNumeral* | *decimalPtNumeral* | *scientificNotationNumeral* | *numericalSpecialRep*

The *doubleRep* production is equivalent to this regular expression (after whitespace is eliminated from the expression):

```
(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)([Ee](\+|-)?[0-9]+)? |(\+|-)?INF|NaN
```

The double datatype is designed to implement for schema processing the double-precision floating-point datatype of [IEEE 754-2008]. That specification does not specify specific ·lexical representations·, but does prescribe requirements on any ·lexical mapping· used. Any ·lexical mapping· that maps the ·lexical space· just described onto the ·value space·, is a function, satisfies the requirements of [IEEE 754-2008], and correctly handles the mapping of the literals '`INF`', '`NaN`', etc., to the ·special values·, satisfies the conformance requirements of this specification.

Since IEEE allows some variation in rounding of values, processors conforming to this specification may exhibit some variation in their ·lexical mappings·.

The ·lexical mapping· *·doubleLexicalMap·* is provided as an example of a simple algorithm that yields a conformant mapping, and that provides the most accurate rounding possible—and is thus useful for insuring inter-implementation reproducibility and inter-implementation round-tripping. The simple rounding algorithm used in *·doubleLexicalMap·* may be more efficiently implemented using the algorithms of [Clinger, WD (1990)].

> **Note:** The Schema 1.0 version of this datatype did not permit rounding algorithms whose results differed from [Clinger, WD (1990)].

The ·canonical mapping· *·doubleCanonicalMap·* is provided as an example of a mapping that does not produce unnecessarily long ·canonical representations·. Other algorithms which do not yield identical results for mapping from float values to character strings are permitted by [IEEE 754-2008].

3.3.6.3 Facets

double and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from double MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

double has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.3.7 duration

[Definition:] **duration** is a datatype that represents durations of time. The concept of duration being captured is drawn from those of [ISO 8601], specifically *durations without fixed endpoints*. For example, "15 days" (whose most common lexical representation in duration is "'`P15D`'") is a duration value; "15 days beginning 12 July 1995" and "15 days ending 12 July 1995" are not duration values. duration can provide addition and subtraction operations between duration values and between duration/dateTime value pairs, and can be the result of subtracting dateTime values. However, only addition to dateTime is required for XML Schema processing and is defined in the function *·dateTimePlusDuration·*.

3.3.7.1 Value Space

Duration values can be modelled as two-property tuples. Each value consists of an integer number of months and a decimal number of seconds. The *·seconds·* value MUST NOT be negative if the *·months·* value is positive and MUST NOT be positive if the *·months·* is negative.

Properties of duration Values
·months·
integer
·seconds·
a decimal value; MUST NOT be negative if *·months·* is positive, and MUST NOT be positive if *·months·* is negative.

duration is partially ordered. Equality of duration is defined in terms of equality of dateTime;

order for duration is defined in terms of the order of dateTime. Specifically, the equality or order of two duration values is determined by adding each duration in the pair to each of the following four dateTime values:

- 1696-09-01T00:00:00Z
- 1697-02-01T00:00:00Z
- 1903-03-01T00:00:00Z
- 1903-07-01T00:00:00Z

If all four resulting dateTime value pairs are ordered the same way (less than, equal, or greater than), then the original pair of duration values is ordered the same way; otherwise the original pair is ·incomparable·.

> **Note:** These four values are chosen so as to maximize the possible differences in results that could occur, such as the difference when adding P1M and P30D: 1697-02-01T00:00:00Z + P1M < 1697-02-01T00:00:00Z + P30D , but 1903-03-01T00:00:00Z + P1M > 1903-03-01T00:00:00Z + P30D , so that P1M <> P30D . If two duration values are ordered the same way when added to each of these four dateTime values, they will retain the same order when added to *any* other dateTime values. Therefore, two duration values are incomparable if and only if they can *ever* result in different orders when added to *any* dateTime value.

Under the definition just given, two duration values are equal if and only if they are identical.

> **Note:** Two totally ordered datatypes (yearMonthDuration and dayTimeDuration) are derived from duration in Other Built-in Datatypes (§3.4).

> **Note:** There are many ways to implement duration, some of which do not base the implementation on the two-component model. This specification does not prescribe any particular implementation, as long as the visible results are isomorphic to those described herein.

> **Note:** See the conformance notes in Partial Implementation of Infinite Datatypes (§5.4), which apply to this datatype.

3.3.7.2 Lexical Mapping

The ·lexical representations· of duration are more or less based on the pattern:

```
PnYnMnDTnHnMnS
```

More precisely, the ·lexical space· of duration is the set of character strings that satisfy *durationLexicalRep* as defined by the following productions:

Lexical Representation Fragments

duYearFrag ::= *unsignedNoDecimalPtNumeral* 'Y'

duMonthFrag ::= *unsignedNoDecimalPtNumeral* 'M' *duDayFrag* ::= *unsignedNoDecimalPtNumeral* 'D'

duHourFrag ::= *unsignedNoDecimalPtNumeral* 'H'

duMinuteFrag ::= *unsignedNoDecimalPtNumeral* 'M'

duSecondFrag ::= (*unsignedNoDecimalPtNumeral* | *unsignedDecimalPtNumeral*) 'S'

duYearMonthFrag ::= (*duYearFrag* *duMonthFrag*?) | *duMonthFrag*

duTimeFrag ::= 'T' ((*duHourFrag* *duMinuteFrag*? *duSecondFrag*?) |
(*duMinuteFrag* *duSecondFrag*?) | *duSecondFrag*)

duDayTimeFrag ::= (*duDayFrag* *duTimeFrag*?) | *duTimeFrag*

Lexical Representation

durationLexicalRep ::= '-'? 'P' ((*duYearMonthFrag* *duDayTimeFrag*?) | *duDayTimeFrag*)

Thus, a *durationLexicalRep* consists of one or more of a *duYearFrag*, *duMonthFrag*, *duDayFrag*, *duHourFrag*, *duMinuteFrag*, and/or *duSecondFrag*, in order, with letters 'P' and 'T' (and perhaps a '-') where appropriate.

The language accepted by the *durationLexicalRep* production is the set of strings which satisfy all of the following three regular expressions:

- The expression

  ```
  -?P([0-9]+Y)?([0-9]+M)?([0-9]+D)?(T([0-9]+H)?([0-9]+M)?([0-9]+(\.[0-9]+)?S)?)?
  ```

  matches only strings in which the fields occur in the proper order.
- The expression '`.*[YMDHS].*`' matches only strings in which at least one field occurs.
- The expression '`.*[^T]`' matches only strings in which 'T' is not the final character, so that if 'T' appears, something follows it. The first rule ensures that what follows 'T' will be an hour, minute, or second field.

The intersection of these three regular expressions is equivalent to the following (after removal of the white space inserted here for legibility):

```
-?P(([0-9]+Y)([0-9]+M)?([0-9]+D)?(T(([0-9]+H)([0-9]+M)?([0-9]+(\.[0-9]+)?S)?|([0-9]+
   |([0-9]+M)([0-9]+D)?(T(([0-9]+H)([0-9]+M)?([0-9]+(\.[0-9]+)?S)?|([0-9]+M)?([0-9]+
   |([0-9]+D)?(T(([0-9]+H)([0-9]+M)?([0-9]+(\.[0-9]+)?S)?|([0-9]+M)?([0-9]+(\.[0-9]+
   |T(([0-9]+H)([0-9]+M)?([0-9]+(\.[0-9]+)?S)?
     |([0-9]+M)?([0-9]+(\.[0-9]+)?S)?
     |([0-9]+(\.[0-9]+)?S)))
```

The ·lexical mapping· for duration is *·durationMap·*.

·The canonical mapping· for duration is *·durationCanonicalMap·*.

3.3.7.3 Facets

duration and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from duration MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

duration has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.7.4 Related Datatypes

The following ·built-in· datatypes are ·derived· from duration

- yearMonthDuration
- dayTimeDuration

3.3.8 dateTime

dateTime represents instants of time, optionally marked with a particular time zone offset. Values representing the same instant but having different time zone offsets are equal but not identical.

3.3.8.1 Value Space

dateTime uses the date/timeSevenPropertyModel, with no properties except *·timezoneOffset·* permitted to be ***absent***. The *·timezoneOffset·* property remains ·optional·.

> **Note:** In version 1.0 of this specification, the *·year·* property was not permitted to have the value zero. The year before the year 1 in the proleptic Gregorian calendar, traditionally referred to as 1 BC or as 1 BCE, was represented by a *·year·* value of −1, 2 BCE by −2, and so forth. Of course, many, perhaps most, references to 1 BCE (or 1 BC) actually refer not to a year in the proleptic Gregorian calendar but to a year in the Julian or "old style" calendar; the two correspond approximately but not exactly to each other.
>
> In this version of this specification, two changes are made in order to agree with existing usage. First, *·year·* is permitted to have the value zero. Second, the interpretation of *·year·* values is changed accordingly: a *·year·* value of zero represents 1 BCE, −1

represents 2 BCE, etc. This representation simplifies interval arithmetic and leap-year calculation for dates before the common era (which may be why astronomers and others interested in such calculations with the proleptic Gregorian calendar have adopted it), and is consistent with the current edition of [ISO 8601].

Note that 1 BCE, 5 BCE, and so on (years 0000, -0004, etc. in the lexical representation defined here) are leap years in the proleptic Gregorian calendar used for the date/time datatypes defined here. Version 1.0 of this specification was unclear about the treatment of leap years before the common era. If existing schemas or data specify dates of 29 February for any years before the common era, then some values giving a date of 29 February which were valid under a plausible interpretation of XSD 1.0 will be invalid under this specification, and some which were invalid will be valid. With that possible exception, schemas and data valid under the old interpretation remain valid under the new.

Constraint: Day-of-month Values

The *·day·* value MUST be no more than 30 if *·month·* is one of 4, 6, 9, or 11; no more than 28 if *·month·* is 2 and *·year·* is not divisible 4, or is divisible by 100 but not by 400; and no more than 29 if *·month·* is 2 and *·year·* is divisible by 400, or by 4 but not by 100.

Note: See the conformance note in Partial Implementation of Infinite Datatypes (§5.4) which applies to the *·year·* and *·second·* values of this datatype.

Equality and order are as prescribed in The Seven-property Model (§D.2.1). dateTime values are ordered by their *·timeOnTimeline·* value.

Note: Since the order of a dateTime value having a *·timezoneOffset·* relative to another value whose *·timezoneOffset·* is ***absent*** is determined by imputing time zone offsets of both +14:00 and −14:00 to the value with no time zone offset, many such combinations will be ·incomparable· because the two imputed time zone offsets yield different orders.

Although dateTime and other types related to dates and times have only a partial order, it is possible for datatypes derived from dateTime to have total orders, if they are restricted (e.g. using the pattern facet) to the subset of values with, or the subset of values without, time zone offsets. Similar restrictions on other date- and time-related types will similarly produce totally ordered subtypes. Note, however, that such restrictions do not affect the value shown, for a given Simple Type Definition, in the ordered facet.

Note: Order and equality are essentially the same for dateTime in this version of this specification as they were in version 1.0. However, since values now distinguish time zone offsets, equal values with different *·timezoneOffset·*s are not *identical*, and values with extreme *·timezoneOffset·*s may no longer be equal to any value with a smaller *·timezoneOffset·*.

3.3.8.2 Lexical Mapping

The lexical representations for dateTime are as follows:

Lexical Space

dateTimeLexicalRep ::=

yearFrag '-' monthFrag '-' dayFrag 'T' ((hourFrag ':' minuteFrag ':' secondFrag) | endOfDayFrag) timezoneFrag? **Constraint:** Day-of-month Representations

Constraint: Day-of-month Representations

Within a *dateTimeLexicalRep*, a *dayFrag* MUST NOT begin with the digit '3' or be '29' unless the value to which it would map would satisfy the value constraint on *·day·* values ("Constraint: Day-of-month Values") given above.

In such representations:

- *yearFrag* is a numeral consisting of at least four decimal digits, optionally preceded by a minus sign; leading '0' digits are prohibited except to bring the digit count up to four. It represents the *·year·* value.
- Subsequent '-', 'T', and ':', separate the various numerals.
- *monthFrag*, *dayFrag*, *hourFrag*, and *minuteFrag* are numerals consisting of exactly two decimal digits. They represent the *·month·*, *·day·*, *·hour·*, and *·minute·* values respectively.
- *secondFrag* is a numeral consisting of exactly two decimal digits, or two decimal digits, a decimal point, and one or more trailing digits. It represents the *·second·* value.
- Alternatively, *endOfDayFrag* combines the *hourFrag*, *minuteFrag*, *minuteFrag*, and their separators to represent midnight of the day, which is the first moment of the next day.
- *timezoneFrag*, if present, specifies an offset between UTC and local time. Time zone offsets are a count of minutes (expressed in *timezoneFrag* as a count of hours and minutes) that are added or subtracted from UTC time to get the "local" time. 'Z' is an alternative representation of the time zone offset '00:00', which is, of course, zero minutes from UTC.

  For example, 2002-10-10T12:00:00−05:00 (noon on 10 October 2002, Central Daylight Savings Time as well as Eastern Standard Time in the U.S.) is equal to 2002-10-10T17:00:00Z, five hours later than 2002-10-10T12:00:00Z.

  > **Note:** For the most part, this specification adopts the distinction between 'timezone' and 'timezone offset' laid out in [Timezones]. Version 1.0 of this specification did not make this distinction, but used the term 'timezone' for the time zone offset information associated with date- and time-related datatypes. Some traces of the earlier usage remain visible in this and other specifications. The names *timezoneFrag* and explicitTimezone are such traces ; others will be found in the names of functions defined in [XQuery 1.0 and XPath 2.0 Functions and Operators], or in references in this specification to "timezoned" and "untimezoned" values.

The *dateTimeLexicalRep* production is equivalent to this regular expression once whitespace is removed.

```
-?([1-9][0-9]{3,}|0[0-9]{3})
-(0[1-9]|1[0-2])
-(0[1-9]|[12][0-9]|3[01])
T(([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9](\.[0-9]+)?|(24:00:00(\.0+)?))
```

```
(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?
```

Note that neither the *dateTimeLexicalRep* production nor this regular expression alone enforce the constraint on *dateTimeLexicalRep* given above.

The ·lexical mapping· for dateTime is *·dateTimeLexicalMap·*. The ·canonical mapping· is *·dateTimeCanonicalMap·*.

3.3.8.3 Facets

dateTime and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

dateTime has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- explicitTimezone = `optional`

Datatypes derived by restriction from dateTime MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

dateTime has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.8.4 Related Datatypes

The following ·built-in· datatype is ·derived· from dateTime

- dateTimeStamp

3.3.9 time

time represents instants of time that recur at the same point in each calendar day, or that occur in some arbitrary calendar day.

3.3.9.1 Value Space

time uses the date/timeSevenPropertyModel, with *·year·*, *·month·*, and *·day·* required to be ***absent***. *·timezoneOffset·* remains ·optional·.

> **Note:** See the conformance note in Partial Implementation of Infinite Datatypes (§5.4) which applies to the *·second·* value of this datatype.

Equality and order are as prescribed in The Seven-property Model (§D.2.1). time values (points in time in an "arbitrary" day) are ordered taking into account their *·timezoneOffset·*.

A calendar (or "local time") day with a larger positive time zone offset begins earlier than the same calendar day with a smaller (or negative) time zone offset. Since the time zone offsets allowed spread over 28 hours, it is possible for the period denoted by a given calendar day with one time zone offset to be completely disjoint from the period denoted by the same calendar day with a different offset — the earlier day ends before the later one starts. The moments in time represented by a single calendar day are spread over a 52-hour interval, from the beginning of the day in the +14:00 time zone offset to the end of that day in the −14:00 time zone offset.

> **Note:** The relative order of two time values, one of which has a *·timezoneOffset·* of ***absent*** is determined by imputing time zone offsets of both +14:00 and −14:00 to the value without an offset. Many such combinations will be ·incomparable· because the two imputed time zone offsets yield different orders. However, for a given untimezoned value, there will always be timezoned values at one or both ends of the 52-hour interval that are ·comparable· (because the interval of ·incomparability· is only 24 hours wide).
>
> Date values with different time zone offsets that were identical in the 1.0 version of this specification, such as 2000-12-12+13:00 and 2000-12-11−11:00, are in this version of this specification equal (because they begin at the same moment on the time line) but are not identical (because they have and retain different time zone offsets).

3.3.9.2 Lexical Mappings

The lexical representations for time are "projections" of those of dateTime, as follows:

Lexical Space
timeLexicalRep ::= ((*hourFrag* ':' *minuteFrag* ':' *secondFrag*) | *endOfDayFrag*) *timezoneFrag*?

The *timeLexicalRep* production is equivalent to this regular expression, once whitespace is removed:

```
(([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9](\.[0-9]+)?|(24:00:00(\.0+)?))(Z|(\+
```

Note that neither the *timeLexicalRep* production nor this regular expression alone enforce the constraint on *timeLexicalRep* given above.

The ·lexical mapping· for time is *·timeLexicalMap·*; the ·canonical mapping· is *·timeCanonicalMap·*.

> **Note:** The ·lexical mapping· maps '`00:00:00`' and '`24:00:00`' to the same value, namely

midnight (·*hour*· = 0 , ·*minute*· = 0 , ·*second*· = 0).

3.3.9.3 Facets

time and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

time has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- explicitTimezone = `optional`

Datatypes derived by restriction from time MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

time has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.10 date

[Definition:] **date** represents top-open intervals of exactly one day in length on the timelines of dateTime, beginning on the beginning moment of each day, up to but not including the beginning moment of the next day). For nontimezoned values, the top-open intervals disjointly cover the nontimezoned timeline, one per day. For timezoned values, the intervals begin at every minute and therefore overlap.

3.3.10.1 Value Space

date uses the date/timeSevenPropertyModel, with ·*hour*·, ·*minute*·, and ·*second*· required to be ***absent***. ·*timezoneOffset*· remains ·optional·.

Constraint: Day-of-month Values

The ·*day*· value MUST be no more than 30 if ·*month*· is one of 4, 6, 9, or 11, no more than 28 if ·*month*· is 2 and ·*year*· is not divisble 4, or is divisible by 100 but not by 400, and no more than 29 if ·*month*· is 2 and ·*year*· is divisible by 400, or by 4 but not by 100.

Note: See the conformance note in Partial Implementation of Infinite Datatypes (§5.4) which applies to the *·year·* value of this datatype.

Equality and order are as prescribed in The Seven-property Model (§D.2.1).

Note: In version 1.0 of this specification, date values did not retain a time zone offset explicitly, but for offsets not too far from zero their time zone offset could be recovered based on their value's first moment on the timeline. The date/timeSevenPropertyModel retains all time zone offsets.

Examples that show the difference from version 1.0 (see Lexical Mapping (§3.3.10.2) for the notations):

- A day is a calendar (or "local time") day offset from ·UTC· by the appropriate interval; this is now true for all *·day·* values, including those with time zone offsets outside the range +12:00 through -11:59 inclusive:

  2000-12-12+13:00 < 2000-12-12+11:00 (just as 2000-12-12+12:00 has always been less than 2000-12-12+11:00, but in version 1.0 2000-12-12+13:00 > 2000-12-12+11:00 , since 2000-12-12+13:00's "recoverable time zone offset" was −11:00)

- Similarly:

  2000-12-12+13:00 = 2000-12-13−11:00 (whereas under 1.0, as just stated, 2000-12-12+13:00 = 2000-12-12−11:00)

3.3.10.2 Lexical Mapping

The lexical representations for date are "projections" of those of dateTime, as follows:

Lexical Space
dateLexicalRep ::= *yearFrag* '`-`' *monthFrag* '`-`' *dayFrag* *timezoneFrag*? **Constraint:** Day-of-month Representations

Constraint: Day-of-month Representations

Within a *dateLexicalRep*, a *dayFrag* MUST NOT begin with the digit '`3`' or be '`29`' unless the value to which it would map would satisfy the value constraint on *·day·* values ("Constraint: Day-of-month Values") given above.

The *dateLexicalRep* production is equivalent to this regular expression:

```
-?([1-9][0-9]{3,}|0[0-9]{3})-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9
```

Note that neither the *dateLexicalRep* production nor this regular expression alone enforce the constraint on *dateLexicalRep* given above.

The ·lexical mapping· for date is *·dateLexicalMap·*. The ·canonical mapping· is *·dateCanonicalMap·*.

3.3.10.3 Facets

date and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

date has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- explicitTimezone = `optional`

Datatypes derived by restriction from date MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

date has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.11 gYearMonth

gYearMonth represents specific whole Gregorian months in specific Gregorian years.

> **Note:** Because month/year combinations in one calendar only rarely correspond to month/year combinations in other calendars, values of this type are not, in general, convertible to simple values corresponding to month/year combinations in other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

3.3.11.1 Value Space

gYearMonth uses the date/timeSevenPropertyModel, with *·day·*, *·hour·*, *·minute·*, and *·second·* required to be ***absent***. *·timezoneOffset·* remains ·optional·.

> **Note:** See the conformance note in Partial Implementation of Infinite Datatypes (§5.4) which applies to the *·year·* value of this datatype.

Equality and order are as prescribed in The Seven-property Model (§D.2.1).

3.3.11.2 Lexical Mapping

The lexical representations for gYearMonth are "projections" of those of dateTime, as follows:

Lexical Space

gYearMonthLexicalRep ::= *yearFrag* '-' *monthFrag* *timezoneFrag*?

The *gYearMonthLexicalRep* is equivalent to this regular expression:

```
-?([1-9][0-9]{3,}|0[0-9]{3})-(0[1-9]|1[0-2])(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?
```

The ·lexical mapping· for gYearMonth is *·gYearMonthLexicalMap·*. The ·canonical mapping· is *·gYearMonthCanonicalMap·*.

3.3.11.3 Facets

gYearMonth and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

gYearMonth has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- explicitTimezone = `optional`

Datatypes derived by restriction from gYearMonth MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

gYearMonth has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.12 gYear

gYear represents Gregorian calendar years.

> **Note:** Because years in one calendar only rarely correspond to years in other calendars,

values of this type are not, in general, convertible to simple values corresponding to years in other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

3.3.12.1 Value Space

gYear uses the date/timeSevenPropertyModel, with *·month·*, *·day·*, *·hour·*, *·minute·*, and *·second·* required to be ***absent***. *·timezoneOffset·* remains ·optional·.

> **Note:** See the conformance note in Partial Implementation of Infinite Datatypes (§5.4) which applies to the *·year·* value of this datatype.

Equality and order are as prescribed in The Seven-property Model (§D.2.1).

3.3.12.2 Lexical Mapping

The lexical representations for gYear are "projections" of those of dateTime, as follows:

Lexical Space
gYearLexicalRep ::= *yearFrag* *timezoneFrag*?

The *gYearLexicalRep* is equivalent to this regular expression:

```
-?([1-9][0-9]{3,}|0[0-9]{3})(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?
```

The ·lexical mapping· for gYear is *·gYearLexicalMap·*. The ·canonical mapping· is *·gYearCanonicalMap·*.

3.3.12.3 Facets

gYear and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

gYear has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- explicitTimezone = `optional`

Datatypes derived by restriction from gYear MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive

- minExclusive
- assertions

gYear has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.13 gMonthDay

gMonthDay represents whole calendar days that recur at the same point in each calendar year, or that occur in some arbitrary calendar year. (Obviously, days beyond 28 cannot occur in all Februaries; 29 is nonetheless permitted.)

This datatype can be used, for example, to record birthdays; an instance of the datatype could be used to say that someone's birthday occurs on the 14th of September every year.

> **Note:** Because day/month combinations in one calendar only rarely correspond to day/month combinations in other calendars, values of this type do not, in general, have any straightforward or intuitive representation in terms of most other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

3.3.13.1 Value Space

gMonthDay uses the date/timeSevenPropertyModel, with *·year·*, *·hour·*, *·minute·*, and *·second·* required to be ***absent***. *·timezoneOffset·* remains ·optional·.

Constraint: Day-of-month Values

The *·day·* value MUST be no more than 30 if *·month·* is one of 4, 6, 9, or 11, and no more than 29 if *·month·* is 2.

Equality and order are as prescribed in The Seven-property Model (§D.2.1).

> **Note:** In version 1.0 of this specification, gMonthDay values did not retain a time zone offset explicitly, but for time zone offsets not too far from ·UTC· their time zone offset could be recovered based on their value's first moment on the timeline. The date/timeSevenPropertyModel retains all time zone offsets.
>
> An example that shows the difference from version 1.0 (see Lexical Mapping (§3.3.13.2) for the notations):
>
> - A day is a calendar (or "local time") day offset from ·UTC· by the appropriate interval; this is now true for all *·day·* values, including those with time zone offsets outside the range +12:00 through -11:59 inclusive:
>
>   --12-12+13:00 < --12-12+11:00 (just as --12-12+12:00 has always been less than --12-12+11:00, but in version 1.0 --12-12+13:00 > --12-12+11:00 , since --12-12+13:00's "recoverable time zone offset" was −11:00)

3.3.13.2 Lexical Mapping

The lexical representations for gMonthDay are "projections" of those of dateTime, as follows:

Lexical Space
gMonthDayLexicalRep ::= '--' *monthFrag* '-' *dayFrag* *timezoneFrag*? **Constraint:** Day-of-month Representations

Constraint: Day-of-month Representations

Within a *gMonthDayLexicalRep*, a *dayFrag* MUST NOT begin with the digit '3' or be '29' unless the value to which it would map would satisfy the value constraint on *·day·* values ("Constraint: Day-of-month Values") given above.

The *gMonthDayLexicalRep* is equivalent to this regular expression:

```
--(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?
```

Note that neither the *gMonthDayLexicalRep* production nor this regular expression alone enforce the constraint on *gMonthDayLexicalRep* given above.

The ·lexical mapping· for gMonthDay is *·gMonthDayLexicalMap·*. The ·canonical mapping· is *·gMonthDayCanonicalMap·*.

3.3.13.3 Facets

gMonthDay and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

gMonthDay has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- explicitTimezone = `optional`

Datatypes derived by restriction from gMonthDay MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

gMonthDay has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.14 gDay

[Definition:] **gDay** represents whole days within an arbitrary month—days that recur at the same point in each (Gregorian) month. This datatype is used to represent a specific day of the month. To indicate, for example, that an employee gets a paycheck on the 15th of each month. (Obviously, days beyond 28 cannot occur in *all* months; they are nonetheless permitted, up to 31.)

> **Note:** Because days in one calendar only rarely correspond to days in other calendars, gDay values do not, in general, have any straightforward or intuitive representation in terms of most non-Gregorian calendars. gDay should therefore be used with caution in contexts where conversion to other calendars is desired.

3.3.14.1 Value Space

gDay uses the date/timeSevenPropertyModel, with *·year·*, *·month·*, *·hour·*, *·minute·*, and *·second·* required to be ***absent***. *·timezoneOffset·* remains ·optional· and *·day·* must be between 1 and 31 inclusive.

Equality and order are as prescribed in The Seven-property Model (§D.2.1). Since gDay values (days) are ordered by their first moments, it is possible for apparent anomalies to appear in the order when *·timezoneOffset·* values differ by at least 24 hours. (It is possible for *·timezoneOffset·* values to differ by up to 28 hours.)

Examples that may appear anomalous (see Lexical Mapping (§3.3.14.2) for the notations):

- ---15 < ---16 , but ---15−13:00 > ---16+13:00
- ---15−11:00 = ---16+13:00
- ---15−13:00 <> ---16 , because ---15−13:00 > ---16+14:00 and ---15−13:00 < 16−14:00

> **Note:** Time zone offsets do not cause wrap-around at the end of the month: the last day of a given month with a time zone offset of −13:00 may start after the first day of the *next* month with offset +13:00, as measured on the global timeline, but nonetheless ---01+13:00 < ---31−13:00 .

3.3.14.2 Lexical Mapping

The lexical representations for gDay are "projections" of those of dateTime, as follows:

Lexical Space
gDayLexicalRep ::= '`---`' *dayFrag* *timezoneFrag*?

The [gDayLexicalRep](#) is equivalent to this regular expression:

```
---(0[1-9]|[12][0-9]|3[01])(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?
```

The ·lexical mapping· for gDay is ·*gDayLexicalMap*·. The ·canonical mapping· is ·*gDayCanonicalMap*·.

3.3.14.3 Facets

gDay and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

gDay has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- explicitTimezone = `optional`

Datatypes derived by restriction from gDay MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

gDay has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.15 gMonth

gMonth represents whole (Gregorian) months within an arbitrary year—months that recur at the same point in each year. It might be used, for example, to say what month annual Thanksgiving celebrations fall in different countries (--11 in the United States, --10 in Canada, and possibly other months in other countries).

> **Note:** Because months in one calendar only rarely correspond to months in other calendars, values of this type do not, in general, have any straightforward or intuitive representation in terms of most other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

3.3.15.1 Value Space

gMonth uses the date/timeSevenPropertyModel, with ·*year*·, ·*day*·, ·*hour*·, ·*minute*·, and ·*second*· required to be ***absent***. ·*timezoneOffset*· remains ·optional·.

Equality and order are as prescribed in The Seven-property Model (§D.2.1).

3.3.15.2 Lexical Mapping

The lexical representations for gMonth are "projections" of those of dateTime, as follows:

Lexical Space
gMonthLexicalRep ::= '`--`' *monthFrag* *timezoneFrag*?

The *gMonthLexicalRep* is equivalent to this regular expression:

```
--(0[1-9]|1[0-2])(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?
```

The ·lexical mapping· for gMonth is ·*gMonthLexicalMap*·. The ·canonical mapping· is ·*gMonthCanonicalMap*·.

3.3.15.3 Facets

gMonth and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

gMonth has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- explicitTimezone = `optional`

Datatypes derived by restriction from gMonth MAY also specify values for the following ·constraining facets·:

- pattern
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

gMonth has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.16 hexBinary

[Definition:] **hexBinary** represents arbitrary hex-encoded binary data.

3.3.16.1 Value Space

The ·value space· of hexBinary is the set of finite-length sequences of zero or more binary octets. The length of a value is the number of octets.

3.3.16.2 Lexical Mapping

hexBinary's ·lexical space· consists of strings of hex (hexadecimal) digits, two consecutive digits representing each octet in the corresponding value (treating the octet as the binary representation of a number between 0 and 255). For example, '`0FB7`' is a ·lexical representation· of the two-octet value 00001111 10110111.

More formally, the ·lexical space· of hexBinary is the set of literals matching the *hexBinary* production.

Lexical space of hexBinary

hexDigit ::= [`0-9a-fA-F`]

hexOctet ::= *hexDigit hexDigit*

hexBinary ::= *hexOctet**

The set recognized by *hexBinary* is the same as that recognized by the regular expression '`([0-9a-fA-F]{2})*`'.

The ·lexical mapping· of hexBinary is *·hexBinaryMap·*.

The ·canonical mapping· of hexBinary is given formally in *·hexBinaryCanonical·*.

3.3.16.3 Facets

hexBinary and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from hexBinary MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- pattern

- enumeration
- assertions

hexBinary has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.17 base64Binary

[Definition:] **base64Binary** represents arbitrary Base64-encoded binary data. For **base64Binary** data the entire binary stream is encoded using the Base64 Encoding defined in [RFC 3548], which is derived from the encoding described in [RFC 2045].

3.3.17.1 Value Space

The ·value space· of base64Binary is the set of finite-length sequences of zero or more binary octets. The length of a value is the number of octets.

3.3.17.2 Lexical Mapping

The ·lexical representations· of base64Binary values are limited to the 65 characters of the Base64 Alphabet defined in [RFC 3548], i.e., `a-z`, `A-Z`, `0-9`, the plus sign (+), the forward slash (/) and the equal sign (=), together with the space character (#x20). No other characters are allowed.

For compatibility with older mail gateways, [RFC 2045] suggests that Base64 data should have lines limited to at most 76 characters in length. This line-length limitation is not required by [RFC 3548] and is not mandated in the ·lexical representations· of base64Binary data. It MUST NOT be enforced by XML Schema processors.

The ·lexical space· of base64Binary is the set of literals which ·match· the base64Binaryproduction.

Lexical space of base64Binary

Base64Binary ::= (*B64quad** *B64final*)?

B64quad ::= (*B64 B64 B64 B64*)
    /* *B64quad represents three octets of binary data.* */

B64final ::= *B64finalquad* | *Padded16* | *Padded8*

B64finalquad ::= (*B64 B64 B64 B64char*)
    /* *B64finalquad represents three octets of binary data without trailing space.* */

Padded16 ::= *B64 B64 B16* '='

/* *Padded16* represents a two-octet at the end of the data. */

Padded8 ::= *B64* *B04* '=' #x20? '='
/* *Padded8* represents a single octet at the end of the data. */

B64 ::= *B64char* #x20?

B64char ::= [A-Za-z0-9+/]

B16 ::= *B16char* #x20?

B16char ::= [AEIMQUYcgkosw048]
/* Base64 characters whose bit-string value ends in '00' */

B04 ::= *B04char* #x20?

B04char ::= [AQgw]
/* Base64 characters whose bit-string value ends in '0000' */

The *Base64Binary* production is equivalent to the following regular expression.

```
((([A-Za-z0-9+/] ?){4})*(([A-Za-z0-9+/] ?){3}[A-Za-z0-9+/]|([A-Za-z0-9+/]
?){2}[AEIMQUYcgkosw048] ?=|[A-Za-z0-9+/] ?[AQgw] ?= ?=))?
```

Note that each '?' except the last is preceded by a single space character.

Note that this grammar requires the number of non-whitespace characters in the ·lexical representation· to be a multiple of four, and for equals signs to appear only at the end of the ·lexical representation·; literals which do not meet these constraints are not legal ·lexical representations· of base64Binary.

The ·lexical mapping· for base64Binary is as given in [RFC 2045] and [RFC 3548].

> **Note:** The above definition of the ·lexical space· is more restrictive than that given in [RFC 2045] as regards whitespace — and less restrictive than [RFC 3548]. This is not an issue in practice. Any string compatible with either RFC can occur in an element or attribute validated by this type, because the ·whiteSpace· facet of this type is fixed to ***collapse***, which means that all leading and trailing whitespace will be stripped, and all internal whitespace collapsed to single space characters, *before* the above grammar is enforced. The possibility of ignoring whitespace in Base64 data is foreseen in clause 2.3 of [RFC 3548], but for the reasons given there this specification does not allow implementations to ignore non-whitespace characters which are not in the Base64 Alphabet.

The canonical ·lexical representation· of a base64Binary data value is the Base64 encoding of the value which matches the Canonical-base64Binary production in the following grammar:

Canonical representation of base64Binary

Canonical-base64Binary ::= *CanonicalQuad** *CanonicalPadded*?

CanonicalQuad ::= *B64char* *B64char* *B64char* *B64char*

CanonicalPadded ::= *B64char* *B64char* *B16char* '=' | *B64char* *B04char* '=='

That is, the ·canonical representation· of a base64Binary value is the ·lexical representation· which maps to that value and contains no whitespace. The ·canonical mapping· for base64Binary is thus the encoding algorithm for Base64 data given in [RFC 2045] and [RFC 3548], with the proviso that no characters except those in the Base64 Alphabet are to be written out.

> **Note:** For some values the ·canonical representation· defined above does not conform to [RFC 2045], which requires breaking with linefeeds at appropriate intervals. It does conform with [RFC 3548].

The length of a base64Binary value may be calculated from the ·lexical representation· by removing whitespace and padding characters and performing the calculation shown in the pseudo-code below:

```
lex2   := killwhitespace(lexform)    -- remove whitespace characters
lex3   := strip_equals(lex2)         -- strip padding characters at end
length := floor (length(lex3) * 3 / 4)         -- calculate length
```

Note on encoding: [RFC 2045] and [RFC 3548] explicitly reference US-utf-8 encoding. However, decoding of **base64Binary** data in an XML entity is to be performed on the Unicode characters obtained after character encoding processing as specified by [XML].

3.3.17.3 Facets

base64Binary and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from base64Binary MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- pattern
- enumeration
- assertions

base64Binary has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***

- cardinality = ***countably infinite***
- numeric = ***false***

3.3.18 anyURI

[Definition:] **anyURI** represents an Internationalized Resource Identifier Reference (IRI). An **anyURI** value can be absolute or relative, and may have an optional fragment identifier (i.e., it may be an IRI Reference). This type should be used when the value fulfills the role of an IRI, as defined in [RFC 3987] or its successor(s) in the IETF Standards Track.

> **Note:** IRIs may be used to locate resources or simply to identify them. In the case where they are used to locate resources using a URI, applications should use the mapping from anyURI values to URIs given by the reference escaping procedure defined in [LEIRI] and in Section 3.1 Mapping of IRIs to URIs of [RFC 3987] or its successor(s) in the IETF Standards Track. This means that a wide range of internationalized resource identifiers can be specified when an anyURI is called for, and still be understood as URIs per [RFC 3986] and its successor(s).

3.3.18.1 Value Space

The value space of anyURI is the set of finite-length sequences of zero or more characters (as defined in [XML]) that ·match· the Char production from [XML].

3.3.18.2 Lexical Mapping

The ·lexical space· of anyURI is the set of finite-length sequences of zero or more characters (as defined in [XML]) that ·match· the Char production from [XML].

> **Note:** For an anyURI value to be usable in practice as an IRI, the result of applying to it the algorithm defined in Section 3.1 of [RFC 3987] should be a string which is a legal URI according to [RFC 3986]. (This is true at the time this document is published; if in the future [RFC 3987] and [RFC 3986] are replaced by other specifications in the IETF Standards Track, the relevant constraints will be those imposed by those successor specifications.)
>
> Each URI scheme imposes specialized syntax rules for URIs in that scheme, including restrictions on the syntax of allowed fragment identifiers. Because it is impractical for processors to check that a value is a context-appropriate URI reference, neither the syntactic constraints defined by the definitions of individual schemes nor the generic syntactic constraints defined by [RFC 3987] and [RFC 3986] and their successors are part of this datatype as defined here. Applications which depend on anyURI values being legal according to the rules of the relevant specifications should make arrangements to check values against the appropriate definitions of IRI, URI, and specific schemes.

> **Note:** Spaces are, in principle, allowed in the ·lexical space· of anyURI, however, their use is highly discouraged (unless they are encoded by '`%20`').

The ·lexical mapping· for anyURI is the identity mapping.

> **Note:** The definitions of URI in the current IETF specifications define certain URIs as equivalent to each other. Those equivalences are not part of this datatype as defined

here: if two "equivalent" URIs or IRIs are different character sequences, they map to different values in this datatype.

3.3.18.3 Facets

anyURI and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from anyURI MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- pattern
- enumeration
- assertions

anyURI has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.19 QName

[Definition:] **QName** represents XML qualified names. The ·value space· of **QName** is the set of tuples {namespace name, local part}, where namespace name is an anyURI and local part is an NCName. The ·lexical space· of **QName** is the set of strings that ·match· the QName production of [Namespaces in XML].

It is ·implementation-defined· whether an implementation of this specification supports the QName production from [Namespaces in XML], or that from [Namespaces in XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

The mapping from lexical space to value space for a particular QName ·literal· depends on the namespace bindings in scope where the literal occurs.

When QNames appear in an XML context, the bindings to be used in the ·lexical mapping· are those in the [in-scope namespaces] property of the relevant element. When this datatype is used in a non-XML host language, the host language MUST specify what namespace bindings are to be used.

The host language, whether XML-based or otherwise, MAY specify whether unqualified names are bound to the default namespace (if any) or not; the host language may also place this under user control. If the host language does not specify otherwise, unqualified names are bound to the default namespace.

Note: The default treatment of unqualified names parallels that specified in [Namespaces in XML] for element names (as opposed to that specified for attribute names).

> **Note:** The mapping between ·literals· in the ·lexical space· and values in the ·value space· of QName depends on the set of namespace declarations in scope for the context in which QName is used.

Because the lexical representations available for any value of type QName vary with context, no ·canonical representation· is defined for QName in this specification.

3.3.19.1 Facets

QName and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from QName MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- pattern
- enumeration
- assertions

QName has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.3.20 NOTATION

[Definition:] **NOTATION** represents the NOTATION attribute type from [XML]. The ·value space· of **NOTATION** is the set of QNames of notations declared in the current schema. The ·lexical space· of **NOTATION** is the set of all names of notations declared in the current schema (in the form of QNames).

> **Note:** Because its ·value space· depends on the notion of a "current schema", as instantiated for example by [XSD 1.1 Part 1: Structures], the NOTATION datatype is unsuitable for use in other contexts which lack the notion of a current schema.

The lexical mapping rules for NOTATION are as given for QName in QName (§3.3.19).

Schema Component Constraint: enumeration facet value required for NOTATION

It is (with one exception) an ·error· for NOTATION to be used directly to validate a literal as described in Datatype Valid (§4.1.4): only datatypes derived from NOTATION by specifying a value for ·enumeration· can be used to validate literals.

The exception is that in the derivation of a new type the ·literals· used to enumerate the allowed values MAY be (and in the context of [XSD 1.1 Part 1: Structures] will be) validated directly against NOTATION; this amounts to verifying that the value is a QName and that the QName is the name of a **NOTATION** declared in the current schema.

For compatibility (see Terminology (§1.6)) NOTATION should be used only on attributes and should only be used in schemas with no target namespace.

> **Note:** Because the lexical representations available for any given value of NOTATION vary with context, this specification defines no ·canonical representation· for NOTATION values.

3.3.20.1 Facets

NOTATION and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

Datatypes derived by restriction from NOTATION MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- pattern
- enumeration
- assertions

NOTATION has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

The use of ·length·, ·minLength· and ·maxLength· on NOTATION or datatypes derived from NOTATION is deprecated. Future versions of this specification may remove these facets for this datatype.

3.4 Other Built-in Datatypes

This section gives conceptual definitions for all ·built-in· ·ordinary· datatypes defined by this specification. The XML representation used to define ·ordinary· datatypes (whether ·built-in· or ·user-defined·) is given in XML Representation of Simple Type Definition Schema Components (§4.1.2) and the complete definitions of the ·built-in· ·ordinary· datatypes are provided in the appendix Schema for Schema Documents (Datatypes) (normative) (§A).

3.4.1 normalizedString

[Definition:] **normalizedString** represents white space normalized strings. The ·value space· of **normalizedString** is the set of strings that do not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters. The ·lexical space· of **normalizedString** is the set of strings that do not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters. The ·base type· of **normalizedString** is string.

3.4.1.1 Facets

normalizedString has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- whiteSpace = `replace`

Datatypes derived by restriction from normalizedString MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- pattern
- enumeration
- assertions

normalizedString has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.1.2 Derived datatypes

The following ·built-in· datatype is ·derived· from normalizedString

- token

3.4.2 token

[Definition:]   **token** represents tokenized strings. The ·value space· of **token** is the set of strings that do not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces. The ·lexical space· of **token** is the set of strings that do not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters, that have no leading or trailing spaces (#x20) and that have no internal sequences of two or more spaces. The ·base type· of **token** is normalizedString.

3.4.2.1 Facets

token has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- whiteSpace = `collapse`

Datatypes derived by restriction from token MAY also specify values for the following

·constraining facets·:

- length
- minLength
- maxLength
- pattern
- enumeration
- assertions

token has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.2.2 Derived datatypes

The following ·built-in· datatypes are ·derived· from token

- language
- NMTOKEN
- Name

3.4.3 language

[Definition:] **language** represents formal natural language identifiers, as defined by [BCP 47] (currently represented by [RFC 4646] and [RFC 4647]) or its successor(s). The ·value space· and ·lexical space· of language are the set of all strings that conform to the pattern

```
[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*
```

This is the set of strings accepted by the grammar given in [RFC 3066], which is now obsolete; the current specification of language codes is more restrictive. The ·base type· of language is token.

> **Note:** The regular expression above provides the only normative constraint on the lexical and value spaces of this type. The additional constraints imposed on language identifiers by [BCP 47] and its successor(s), and in particular their requirement that language codes be registered with IANA or ISO if not given in ISO 639, are not part of this datatype as defined here.

> **Note:** [BCP 47] specifies that language codes "are to be treated as case insensitive; there exist conventions for capitalization of some of the subtags, but these MUST NOT be taken to carry meaning." Since the language datatype is derived from string, it inherits from string a one-to-one mapping from lexical representations to values. The literals '`MN`' and '`mn`' (for Mongolian) therefore correspond to distinct values and have distinct canonical forms. Users of this specification should be aware of this fact, the consequence of which is that the case-insensitive treatment of language values prescribed by [BCP 47] does not follow from the definition of this datatype given here; applications which require case-insensitivity should make appropriate adjustments.

Note: The empty string is not a member of the ·value space· of language. Some constructs which normally take language codes as their values, however, also allow the empty string. The attribute `xml:lang` defined by [XML] is one example; there, the empty string overrides a value which would otherwise be inherited, but without specifying a new value.

One way to define the desired set of possible values is illustrated by the schema document for the XML namespace at http://www.w3.org/2001/xml.xsd, which defines the attribute `xml:lang` as having a type which is a union of language and an anonymous type whose only value is the empty string:

```
<xs:attribute name="lang">
  <xs:annotation>
    <xs:documentation>
      See RFC 3066 at http://www.ietf.org/rfc/rfc3066.txt
      and the IANA registry at
      http://www.iana.org/assignments/lang-tag-apps.htm for
      further information.

      The union allows for the 'un-declaration' of xml:lang with
      the empty string.
    </xs:documentation>
  </xs:annotation>
  <xs:simpleType>
    <xs:union memberTypes="xs:language">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value=""/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
</xs:attribute>
```

3.4.3.1 Facets

language has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*`
- whiteSpace = `collapse`

Datatypes derived by restriction from language MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- enumeration
- assertions

language has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.4 NMTOKEN

[Definition:] **NMTOKEN** represents the NMTOKEN attribute type from [XML]. The ·value space· of **NMTOKEN** is the set of tokens that ·match· the Nmtoken production in [XML]. The ·lexical space· of **NMTOKEN** is the set of strings that ·match· the Nmtoken production in [XML]. The ·base type· of **NMTOKEN** is token.

It is ·implementation-defined· whether an implementation of this specification supports the NMTOKEN production from [XML], or that from [XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

For compatibility (see Terminology (§1.6) NMTOKEN should be used only on attributes.

3.4.4.1 Facets

NMTOKEN has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `\c+`
- whiteSpace = `collapse`

Datatypes derived by restriction from NMTOKEN MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- enumeration
- assertions

NMTOKEN has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.4.2 Related datatypes

The following ·built-in· datatype is ·constructed· from NMTOKEN

- NMTOKENS

3.4.5 NMTOKENS

[Definition:] **NMTOKENS** represents the NMTOKENS attribute type from [XML]. The ·value space· of **NMTOKENS** is the set of finite, non-zero-length sequences of ·NMTOKEN·s. The ·lexical space· of **NMTOKENS** is the set of space-separated lists of tokens, of which each token is in the ·lexical space· of NMTOKEN. The ·item type· of **NMTOKENS** is NMTOKEN.

For compatibility (see Terminology (§1.6)) NMTOKENS should be used only on attributes.

3.4.5.1 Facets

NMTOKENS has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- minLength = `1`
- whiteSpace = `collapse`

Datatypes derived by restriction from NMTOKENS MAY also specify values for the following ·constraining facets·:

- length
- maxLength
- enumeration
- pattern
- assertions

NMTOKENS has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.6 Name

[Definition:]   **Name** represents XML Names. The ·value space· of **Name** is the set of all strings which ·match· the Name production of [XML]. The ·lexical space· of **Name** is the set of all strings which ·match· the Name production of [XML]. The ·base type· of **Name** is token.

It is ·implementation-defined· whether an implementation of this specification supports the Name production from [XML], or that from [XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

3.4.6.1 Facets

Name has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `\i\c*`
- whiteSpace = `collapse`

Datatypes derived by restriction from Name MAY also specify values for the following ·constraining facets·:

- length
- minLength

- maxLength
- enumeration
- assertions

Name has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.6.2 Derived datatypes

The following ·built-in· datatype is ·derived· from Name

- NCName

3.4.7 NCName

[Definition:] **NCName** represents XML "non-colonized" Names. The ·value space· of **NCName** is the set of all strings which ·match· the NCName production of [Namespaces in XML]. The ·lexical space· of **NCName** is the set of all strings which ·match· the NCName production of [Namespaces in XML]. The ·base type· of **NCName** is Name.

It is ·implementation-defined· whether an implementation of this specification supports the NCName production from [Namespaces in XML], or that from [Namespaces in XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

3.4.7.1 Facets

NCName has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `\i\c* ∩ [\i-[:]][\c-[:]]*`
- whiteSpace = `collapse`

Datatypes derived by restriction from NCName MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- enumeration
- assertions

NCName has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***

- numeric = ***false***

3.4.7.2 Derived datatypes

The following ·built-in· datatypes are ·derived· from NCName

- ID
- IDREF
- ENTITY

3.4.8 ID

[Definition:] **ID** represents the ID attribute type from [XML]. The ·value space· of **ID** is the set of all strings that ·match· the NCName production in [Namespaces in XML]. The ·lexical space· of **ID** is the set of all strings that ·match· the NCName production in [Namespaces in XML]. The ·base type· of **ID** is NCName.

> **Note:** It is ·implementation-defined· whether an implementation of this specification supports the NCName production from [Namespaces in XML], or that from [Namespaces in XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

For compatibility (see Terminology (§1.6)), ID should be used only on attributes.

> **Note:** Uniqueness of items validated as ID is not part of this datatype as defined here. When this specification is used in conjunction with [XSD 1.1 Part 1: Structures], uniqueness is enforced at a different level, not as part of datatype validity; see Validation Rule: Validation Root Valid (ID/IDREF) in [XSD 1.1 Part 1: Structures].

3.4.8.1 Facets

ID has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `\i\c* ∩ [\i-[:]][\c-[:]]*`
- whiteSpace = `collapse`

Datatypes derived by restriction from ID MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- enumeration
- assertions

ID has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.9 IDREF

[Definition:] **IDREF** represents the IDREF attribute type from [XML]. The ·value space· of **IDREF** is the set of all strings that ·match· the NCName production in [Namespaces in XML]. The ·lexical space· of **IDREF** is the set of strings that ·match· the NCName production in [Namespaces in XML]. The ·base type· of **IDREF** is NCName.

> **Note:** It is ·implementation-defined· whether an implementation of this specification supports the NCName production from [Namespaces in XML], or that from [Namespaces in XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

For compatibility (see Terminology (§1.6)) this datatype should be used only on attributes.

> **Note:** Existence of referents for items validated as IDREF is not part of this datatype as defined here. When this specification is used in conjunction with [XSD 1.1 Part 1: Structures], referential integrity is enforced at a different level, not as part of datatype validity; see Validation Rule: Validation Root Valid (ID/IDREF) in [XSD 1.1 Part 1: Structures].

3.4.9.1 Facets

IDREF has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `\i\c* ∩ [\i-[:]][\c-[:]]*`
- whiteSpace = `collapse`

Datatypes derived by restriction from IDREF MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- enumeration
- assertions

IDREF has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.9.2 Related datatypes

The following ·built-in· datatype is ·constructed· from IDREF

- IDREFS

3.4.10 IDREFS

[Definition:]   **IDREFS** represents the IDREFS attribute type from [XML]. The ·value space· of **IDREFS** is the set of finite, non-zero-length sequences of IDREFs. The ·lexical space· of **IDREFS** is the set of space-separated lists of tokens, of which each token is in the ·lexical space· of IDREF. The ·item type· of **IDREFS** is IDREF.

For compatibility (see Terminology (§1.6)) IDREFS should be used only on attributes.

> **Note:** Existence of referents for items validated as IDREFS is not part of this datatype as defined here. When this specification is used in conjunction with [XSD 1.1 Part 1: Structures], referential integrity is enforced at a different level, not as part of datatype validity; see Validation Rule: Validation Root Valid (ID/IDREF) in [XSD 1.1 Part 1: Structures].

3.4.10.1 Facets

IDREFS has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- minLength = `1`
- whiteSpace = `collapse`

Datatypes derived by restriction from IDREFS MAY also specify values for the following ·constraining facets·:

- length
- maxLength
- enumeration
- pattern
- assertions

IDREFS has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.11 ENTITY

[Definition:]   **ENTITY** represents the ENTITY attribute type from [XML]. The ·value space· of **ENTITY** is the set of all strings that ·match· the NCName production in [Namespaces in XML] and have been declared as an unparsed entity in a document type definition. The ·lexical space· of **ENTITY** is the set of all strings that ·match· the NCName production in [Namespaces in XML]. The ·base type· of **ENTITY** is NCName.

> **Note:** It is ·implementation-defined· whether an implementation of this specification supports the NCName production from [Namespaces in XML], or that from [Namespaces in XML 1.0], or both. See Dependencies on Other Specifications (§1.3).

> **Note:** The ·value space· of ENTITY is scoped to a specific instance document.

For compatibility (see Terminology (§1.6)) ENTITY should be used only on attributes.

3.4.11.1 Facets

ENTITY has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `\i\c* ∩ [\i-[:]][\c-[:]]*`
- whiteSpace = `collapse`

Datatypes derived by restriction from ENTITY MAY also specify values for the following ·constraining facets·:

- length
- minLength
- maxLength
- enumeration
- assertions

ENTITY has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.11.2 Related datatypes

The following ·built-in· datatype is ·constructed· from ENTITY

- ENTITIES

3.4.12 ENTITIES

[Definition:] **ENTITIES** represents the ENTITIES attribute type from [XML]. The ·value space· of **ENTITIES** is the set of finite, non-zero-length sequences of ·ENTITY· valuess that have been declared as unparsed entities in a document type definition. The ·lexical space· of **ENTITIES** is the set of space-separated lists of tokens, of which each token is in the ·lexical space· of ENTITY. The ·item type· of **ENTITIES** is ENTITY.

> **Note:** The ·value space· of ENTITIES is scoped to a specific instance document.

For compatibility (see Terminology (§1.6)) ENTITIES should be used only on attributes.

3.4.12.1 Facets

ENTITIES has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- minLength = `1`
- whiteSpace = `collapse`

Datatypes derived by restriction from ENTITIES MAY also specify values for the following ·constraining facets·:

- length
- maxLength
- enumeration
- pattern
- assertions

ENTITIES has the following values for its ·fundamental facets·:

- ordered = ***false***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

3.4.13 integer

[Definition:] **integer** is derived from decimal by fixing the value of ·fractionDigits· to be 0 and disallowing the trailing decimal point. This results in the standard mathematical concept of the integer numbers. The ·value space· of **integer** is the infinite set {...,-2,-1,0,1,2,...}. The ·base type· of **integer** is decimal.

3.4.13.1 Lexical representation

integer has a lexical representation consisting of a finite-length sequence of one or more decimal digits (#x30-#x39) with an optional leading sign. If the sign is omitted, "+" is assumed. For example: -1, 0, 12678967543233, +100000.

3.4.13.2 Canonical representation

The ·canonical representation· for integer is defined by prohibiting certain options from the Lexical representation (§3.4.13.1). Specifically, the preceding optional "+" sign is prohibited and leading zeroes are prohibited.

3.4.13.3 Facets

integer and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

integer has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`

Datatypes derived by restriction from integer MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

integer has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***true***

3.4.13.4 Derived datatypes

The following ·built-in· datatypes are ·derived· from integer

- nonPositiveInteger
- long
- nonNegativeInteger

3.4.14 nonPositiveInteger

[Definition:] **nonPositiveInteger** is derived from integer by setting the value of ·maxInclusive· to be 0. This results in the standard mathematical concept of the non-positive integers. The ·value space· of **nonPositiveInteger** is the infinite set {...,-2,-1,0}. The ·base type· of **nonPositiveInteger** is integer.

3.4.14.1 Lexical representation

nonPositiveInteger has a lexical representation consisting of an optional preceding sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). The sign may be "+" or may be omitted only for lexical forms denoting zero; in all other lexical forms, the negative sign ('-') must be present. For example: -1, 0, -12678967543233, -100000.

3.4.14.2 Canonical representation

The ·canonical representation· for nonPositiveInteger is defined by prohibiting certain options from the Lexical representation (§3.4.14.1). In the canonical form for zero, the sign must be omitted. Leading zeroes are prohibited.

3.4.14.3 Facets

nonPositiveInteger and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

nonPositiveInteger has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `0`

Datatypes derived by restriction from nonPositiveInteger MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxExclusive
- minInclusive
- minExclusive
- assertions

nonPositiveInteger has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***true***

3.4.14.4 Derived datatypes

The following ·built-in· datatype is ·derived· from nonPositiveInteger

- negativeInteger

3.4.15 negativeInteger

[Definition:] **negativeInteger** is derived from nonPositiveInteger by setting the value of ·maxInclusive· to be -1. This results in the standard mathematical concept of the negative integers. The ·value space· of **negativeInteger** is the infinite set {...,-2,-1}. The ·base type· of **negativeInteger** is nonPositiveInteger.

3.4.15.1 Lexical representation

negativeInteger has a lexical representation consisting of a negative sign ('-') followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). For example: -1, -12678967543233, -100000.

3.4.15.2 Canonical representation

The ·canonical representation· for negativeInteger is defined by prohibiting certain options from the Lexical representation (§3.4.15.1). Specifically, leading zeroes are prohibited.

3.4.15.3 Facets

negativeInteger and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

negativeInteger has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `-1`

Datatypes derived by restriction from negativeInteger MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxExclusive
- minInclusive
- minExclusive
- assertions

negativeInteger has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***true***

3.4.16 long

[Definition:] **long** is derived from integer by setting the value of ·maxInclusive· to be 9223372036854775807 and ·minInclusive· to be -9223372036854775808. The ·base type· of **long** is integer.

3.4.16.1 Lexical Representation

long has a lexical representation consisting of an optional sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, "+" is assumed. For example: -1, 0, 12678967543233, +100000.

3.4.16.2 Canonical Representation

The ·canonical representation· for long is defined by prohibiting certain options from the Lexical Representation (§3.4.16.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.4.16.3 Facets

long and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

long has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `9223372036854775807`
- minInclusive = `-9223372036854775808`

Datatypes derived by restriction from long MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxExclusive
- minExclusive
- assertions

long has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.4.16.4 Derived datatypes

The following ·built-in· datatype is ·derived· from long

- int

3.4.17 int

[Definition:] **int** is derived from long by setting the value of ·maxInclusive· to be 2147483647 and ·minInclusive· to be -2147483648. The ·base type· of **int** is long.

3.4.17.1 Lexical Representation

int has a lexical representation consisting of an optional sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, "+" is assumed. For example: -1, 0, 126789675, +100000.

3.4.17.2 Canonical representation

The ·canonical representation· for int is defined by prohibiting certain options from the Lexical Representation (§3.4.17.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.4.17.3 Facets

int and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

int has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `2147483647`
- minInclusive = `-2147483648`

Datatypes derived by restriction from int MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxExclusive
- minExclusive
- assertions

int has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.4.17.4 Derived datatypes

The following ·built-in· datatype is ·derived· from int

- short

3.4.18 short

[Definition:] **short** is derived from int by setting the value of ·maxInclusive· to be 32767 and ·minInclusive· to be -32768. The ·base type· of **short** is int.

3.4.18.1 Lexical representation

short has a lexical representation consisting of an optional sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, "+" is assumed. For example: -1, 0, 12678, +10000.

3.4.18.2 Canonical representation

The ·canonical representation· for short is defined by prohibiting certain options from the Lexical representation (§3.4.18.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.4.18.3 Facets

short and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

short has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `32767`
- minInclusive = `-32768`

Datatypes derived by restriction from short MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxExclusive
- minExclusive
- assertions

short has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.4.18.4 Derived datatypes

The following ·built-in· datatype is ·derived· from short

- byte

3.4.19 byte

[Definition:] **byte** is derived from short by setting the value of ·maxInclusive· to be 127 and ·minInclusive· to be -128. The ·base type· of **byte** is short.

3.4.19.1 Lexical representation

byte has a lexical representation consisting of an optional sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, "+" is assumed. For example: -1, 0, 126, +100.

3.4.19.2 Canonical representation

The ·canonical representation· for byte is defined by prohibiting certain options from the Lexical representation (§3.4.19.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.4.19.3 Facets

byte and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

byte has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `127`
- minInclusive = `-128`

Datatypes derived by restriction from byte MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxExclusive
- minExclusive
- assertions

byte has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.4.20 nonNegativeInteger

[Definition:] **nonNegativeInteger** is derived from integer by setting the value of ·minInclusive· to be 0. This results in the standard mathematical concept of the non-negative integers. The ·value space· of **nonNegativeInteger** is the infinite set {0,1,2,...}. The ·base type· of **nonNegativeInteger** is integer.

3.4.20.1 Lexical representation

nonNegativeInteger has a lexical representation consisting of an optional sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, the positive sign ('+') is assumed. If the sign is present, it must be "+" except for lexical forms denoting zero, which may be preceded by a positive ('+') or a negative ('-') sign. For example: 1, 0, 12678967543233, +100000.

3.4.20.2 Canonical representation

The ·canonical representation· for nonNegativeInteger is defined by prohibiting certain options from the Lexical representation (§3.4.20.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.4.20.3 Facets

nonNegativeInteger and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

nonNegativeInteger has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- minInclusive = `0`

Datatypes derived by restriction from nonNegativeInteger MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxInclusive
- maxExclusive
- minExclusive
- assertions

nonNegativeInteger has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***false***
- cardinality = ***countably infinite***

- numeric = ***true***

3.4.20.4 Derived datatypes

The following ·built-in· datatypes are ·derived· from nonNegativeInteger

- unsignedLong
- positiveInteger

3.4.21 unsignedLong

[Definition:] **unsignedLong** is derived from nonNegativeInteger by setting the value of ·maxInclusive· to be 18446744073709551615. The ·base type· of **unsignedLong** is nonNegativeInteger.

3.4.21.1 Lexical representation

unsignedLong has a lexical representation consisting of an optional sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, the positive sign ('+') is assumed. If the sign is present, it must be '+' except for lexical forms denoting zero, which may be preceded by a positive ('+') or a negative ('-') sign. For example: 0, 12678967543233, 100000.

3.4.21.2 Canonical representation

The ·canonical representation· for unsignedLong is defined by prohibiting certain options from the Lexical representation (§3.4.21.1). Specifically, leading zeroes are prohibited.

3.4.21.3 Facets

unsignedLong and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

unsignedLong has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `18446744073709551615`
- minInclusive = `0`

Datatypes derived by restriction from unsignedLong MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration

- maxExclusive
- minExclusive
- assertions

unsignedLong has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.4.21.4 Derived datatypes

The following ·built-in· datatype is ·derived· from unsignedLong

- unsignedInt

3.4.22 unsignedInt

[Definition:] **unsignedInt** is derived from unsignedLong by setting the value of ·maxInclusive· to be 4294967295. The ·base type· of **unsignedInt** is unsignedLong.

3.4.22.1 Lexical representation

unsignedInt has a lexical representation consisting of an optional sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, the positive sign ('`+`') is assumed. If the sign is present, it must be '`+`' except for lexical forms denoting zero, which may be preceded by a positive ('`+`') or a negative ('`-`') sign. For example: 0, 1267896754, 100000.

3.4.22.2 Canonical representation

The ·canonical representation· for unsignedInt is defined by prohibiting certain options from the Lexical representation (§3.4.22.1). Specifically, leading zeroes are prohibited.

3.4.22.3 Facets

unsignedInt and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

unsignedInt has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `4294967295`

- minInclusive = 0

Datatypes derived by restriction from unsignedInt MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxExclusive
- minExclusive
- assertions

unsignedInt has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.4.22.4 Derived datatypes

The following ·built-in· datatype is ·derived· from unsignedInt

- unsignedShort

3.4.23 unsignedShort

[Definition:] **unsignedShort** is derived from unsignedInt by setting the value of ·maxInclusive· to be 65535. The ·base type· of **unsignedShort** is unsignedInt.

3.4.23.1 Lexical representation

unsignedShort has a lexical representation consisting of an optional sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, the positive sign ('+') is assumed. If the sign is present, it must be '+' except for lexical forms denoting zero, which may be preceded by a positive ('+') or a negative ('-') sign. For example: 0, 12678, 10000.

3.4.23.2 Canonical representation

The ·canonical representation· for unsignedShort is defined by prohibiting certain options from the Lexical representation (§3.4.23.1). Specifically, the leading zeroes are prohibited.

3.4.23.3 Facets

unsignedShort and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = 0 (fixed)
- whiteSpace = `collapse` (fixed)

unsignedShort has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `65535`
- minInclusive = `0`

Datatypes derived by restriction from unsignedShort MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxExclusive
- minExclusive
- assertions

unsignedShort has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.4.23.4 Derived datatypes

The following ·built-in· datatype is ·derived· from unsignedShort

- unsignedByte

3.4.24 unsignedByte

[Definition:] **unsignedByte** is derived from unsignedShort by setting the value of ·maxInclusive· to be 255. The ·base type· of **unsignedByte** is unsignedShort.

3.4.24.1 Lexical representation

unsignedByte has a lexical representation consisting of an optional sign followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, the positive sign ('+') is assumed. If the sign is present, it must be '+' except for lexical forms denoting zero, which may be preceded by a positive ('+') or a negative ('-') sign. For example: 0, 126, 100.

3.4.24.2 Canonical representation

The ·canonical representation· for unsignedByte is defined by prohibiting certain options from the Lexical representation (§3.4.24.1). Specifically, leading zeroes are prohibited.

3.4.24.3 Facets

unsignedByte and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

unsignedByte has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- maxInclusive = `255`
- minInclusive = `0`

Datatypes derived by restriction from unsignedByte MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxExclusive
- minExclusive
- assertions

unsignedByte has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***true***
- cardinality = ***finite***
- numeric = ***true***

3.4.25 positiveInteger

[Definition:] **positiveInteger** is derived from nonNegativeInteger by setting the value of ·minInclusive· to be 1. This results in the standard mathematical concept of the positive integer numbers. The ·value space· of **positiveInteger** is the infinite set {1,2,...}. The ·base type· of **positiveInteger** is nonNegativeInteger.

3.4.25.1 Lexical representation

positiveInteger has a lexical representation consisting of an optional positive sign ('+') followed by a non-empty finite-length sequence of decimal digits (#x30-#x39). For example: 1, 12678967543233, +100000.

3.4.25.2 Canonical representation

The ·canonical representation· for positiveInteger is defined by prohibiting certain options from the Lexical representation (§3.4.25.1). Specifically, the optional "+" sign is prohibited and leading zeroes are prohibited.

3.4.25.3 Facets

positiveInteger and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- fractionDigits = `0` (fixed)
- whiteSpace = `collapse` (fixed)

positiveInteger has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[\-+]?[0-9]+`
- minInclusive = `1`

Datatypes derived by restriction from positiveInteger MAY also specify values for the following ·constraining facets·:

- totalDigits
- enumeration
- maxInclusive
- maxExclusive
- minExclusive
- assertions

positiveInteger has the following values for its ·fundamental facets·:

- ordered = ***total***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***true***

3.4.26 yearMonthDuration

[Definition:] **yearMonthDuration** is a datatype derived from duration by restricting its ·lexical representations· to instances of *yearMonthDurationLexicalRep*. The ·value space· of **yearMonthDuration** is therefore that of duration restricted to those whose *·seconds·* property is 0. This results in a duration datatype which is totally ordered.

> **Note:** The always-zero *·seconds·* is formally retained in order that yearMonthDuration's (abstract) value space truly be a subset of that of duration An obvious implementation optimization is to ignore the zero and implement yearMonthDuration values simply as integer values.

3.4.26.1 The yearMonthDuration Lexical Mapping

The lexical space is reduced from that of duration by disallowing *duDayFrag* and *duTimeFrag* fragments in the ·lexical representations·.

The yearMonthDuration Lexical Representation
yearMonthDurationLexicalRep ::= '`-`'? '`P`' *duYearMonthFrag*

The lexical space of yearMonthDuration consists of strings which match the regular expression '-?P((([0-9]+Y)([0-9]+M)?)|([0-9]+M))' or the expression '-?P[0-9]+(Y([0-9]+M)?|M)', but the formal definition of yearMonthDuration uses a simpler regular expression in its ·pattern· facet: '[^DT]*'. This pattern matches only strings of characters which contain no 'D' and no 'T', thus restricting the ·lexical space· of duration to strings with no day, hour, minute, or seconds fields.

The ·canonical mapping· is that of duration restricted in its range to the ·lexical space· (which reduces its domain to omit any values not in the yearMonthDuration value space).

> **Note:** The yearMonthDuration value whose *·months·* and *·seconds·* are both zero has no ·canonical representation· in this datatype since its ·canonical representation· in duration ('PT0S') is not in the ·lexical space· of yearMonthDuration.

3.4.26.2 Facets

yearMonthDuration and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = collapse (fixed)

yearMonthDuration has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = [^DT]*

Datatypes derived by restriction from yearMonthDuration MAY also specify values for the following ·constraining facets·:

- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

yearMonthDuration has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

> **Note:** The ordered facet has the value ***partial*** even though the datatype is in fact totally ordered, because (as explained in ordered (§4.2.1)), the value of that facet is unchanged by derivation.

3.4.27 dayTimeDuration

[Definition:] **dayTimeDuration** is a datatype derived from duration by restricting its ·lexical representations· to instances of *dayTimeDurationLexicalRep*. The ·value space· of **dayTimeDuration** is therefore that of duration restricted to those whose *·months·* property is 0. This results in a duration datatype which is totally ordered.

3.4.27.1 The dayTimeDuration Lexical Space

The lexical space is reduced from that of duration by disallowing *duYearFrag* and *duMonthFrag* fragments in the ·lexical representations·.

The dayTimeDuration Lexical Representation
dayTimeDurationLexicalRep ::= '`-`'? '`P`' *duDayTimeFrag*

The lexical space of dayTimeDuration consists of strings in the ·lexical space· of duration which match the regular expression '`[^YM]*[DT].*`'; this pattern eliminates all durations with year or month fields, leaving only those with day, hour, minutes, and/or seconds fields.

The ·canonical mapping· is that of duration restricted in its range to the ·lexical space· (which reduces its domain to omit any values not in the dayTimeDuration value space).

3.4.27.2 Facets

dayTimeDuration and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)

dayTimeDuration has the following ·constraining facets· with the values shown; these facets MAY be specified in the derivation of new types, if the value given is at least as restrictive as the one shown:

- pattern = `[^YM]*(T.*)?`

Datatypes derived by restriction from dayTimeDuration MAY also specify values for the following ·constraining facets·:

- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

dayTimeDuration has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***

- numeric = ***false***

Note: The ordered facet has the value ***partial*** even though the datatype is in fact totally ordered, because (as explained in ordered (§4.2.1)), the value of that facet is unchanged by derivation.

3.4.28 dateTimeStamp

[Definition:] The **dateTimeStamp** datatype is derived from dateTime by giving the value ***required*** to its explicitTimezone facet. The result is that all values of dateTimeStamp are required to have explicit time zone offsets and the datatype is totally ordered.

3.4.28.1 The dateTimeStamp Lexical Space

As a consequence of requiring an explicit time zone offset, the lexical space of dateTimeStamp is reduced from that of dateTime by requiring a *timezoneFrag* fragment in the ·lexical representations·.

The dateTimeStamp Lexical Representation

dateTimeStampLexicalRep ::=
yearFrag '`-`' *monthFrag* '`-`' *dayFrag* '`T`' ((*hourFrag* '`:`' *minuteFrag* '`:`' *secondFrag*) | *endOfDayFrag*) *timezoneFrag* **Constraint:** Day-of-month Representations

Note: For details of the Day-of-month Representations (§3.3.8.2) constraint, see dateTime, from which the constraint is inherited.

In other words, the lexical space of dateTimeStamp consists of strings which are in the ·lexical space· of dateTime and which also match the regular expression '`.*(Z|(\+|-)[0-9][0-9]:[0-9][0-9])`'.

The ·lexical mapping· is that of dateTime restricted to the dateTimeStamp lexical space.

The ·canonical mapping· is that of dateTime restricted to the dateTimeStamp value space.

3.4.28.2 Facets

dateTimeStamp and all datatypes derived from it by restriction have the following ·constraining facets· with ***fixed*** values; these facets MUST NOT be changed from the values shown:

- whiteSpace = `collapse` (fixed)
- explicitTimezone = `required` (fixed)

Datatypes derived by restriction from dateTimeStamp MAY also specify values for the following ·constraining facets·:

- pattern

- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- assertions

dateTimeStamp has the following values for its ·fundamental facets·:

- ordered = ***partial***
- bounded = ***false***
- cardinality = ***countably infinite***
- numeric = ***false***

> **Note:** The ordered facet has the value ***partial*** even though the datatype is in fact totally ordered, because (as explained in ordered (§4.2.1)), the value of that facet is unchanged by derivation.

4 Datatype components

The preceding sections of this specification have described datatypes in a way largely independent of their use in the particular context of schema-aware processing as defined in [XSD 1.1 Part 1: Structures].

This section presents the mechanisms necessary to integrate datatypes into the context of [XSD 1.1 Part 1: Structures], mostly in terms of the schema component abstraction introduced there. The account of datatypes given in this specification is also intended to be useful in other contexts. Any specification or other formal system intending to use datatypes as defined above, particularly if definition of new datatypes via facet-based restriction is envisaged, will need to provide analogous mechanisms for some, but not necessarily all, of what follows below. For example, the {target namespace} and {final} properties are required because of particular aspects of [XSD 1.1 Part 1: Structures] which are not in principle necessary for the use of datatypes as defined here.

The following sections provide full details on the properties and significance of each kind of schema component involved in datatype definitions. For each property, the kinds of values it is allowed to have is specified. Any property not identified as optional is required to be present; optional properties which are not present have absent as their value. Any property identified as a having a set, subset or ·list· value may have an empty value unless this is explicitly ruled out: this is not the same as absent. Any property value identified as a superset or a subset of some set may be equal to that set, unless a proper superset or subset is explicitly called for.

For more information on the notion of schema components, see Schema Component Details of [XSD 1.1 Part 1: Structures].

[Definition:] A component may be referred to as the **owner** of its properties, and of the values of those properties.

4.1 Simple Type Definition

Simple Type Definitions provide for:

- In the case of ·primitive· datatypes, identifying a datatype with its definition in this specification.
- In the case of ·constructed· datatypes, defining the datatype in terms of other datatypes.
- Attaching a QName to the datatype.

4.1.1 The Simple Type Definition Schema Component

The Simple Type Definition schema component has the following properties:

Schema Component: Simple Type Definition

`{annotations}`
: A sequence of Annotation components.

`{name}`
: An xs:NCName value. Optional.

`{target namespace}`
: An xs:anyURI value. Optional.

`{final}`
: A subset of {***restriction***, ***extension***, ***list***, ***union***}

`{context}`
: Required if {name} is ***absent***, otherwise MUST be ***absent***
Either an Attribute Declaration, an Element Declaration, a Complex Type Definition or a Simple Type Definition.

`{base type definition}`
: A Type Definition component. Required.
With one exception, the {base type definition} of any Simple Type Definition is a Simple Type Definition. The exception is ·anySimpleType·, which has anyType, a Complex Type Definition, as its {base type definition}.

`{facets}`
: A set of Constraining Facet components.

`{fundamental facets}`
: A set of Fundamental Facet components.

`{variety}`
: One of {***atomic***, ***list***, ***union***}. Required for all Simple Type Definitions except ·anySimpleType·, in which it is ***absent***.

`{primitive type definition}`
: A Simple Type Definition component. With one exception, required if {variety} is ***atomic***, otherwise MUST be ***absent***. The exception is ·anyAtomicType·, whose {primitive type definition} is ***absent***.
If not ***absent***, MUST be a ·primitive· built-in definition.

`{item type definition}`
: A Simple Type Definition component. Required if {variety} is ***list***, otherwise MUST be ***absent***.

`{member type definitions}`
: A sequence of Simple Type Definition components.

MUST be present (but MAY be empty) if {variety} is ***union***, otherwise MUST be ***absent***.

Simple type definitions are identified by their {name} and {target namespace}. Except for anonymous Simple Type Definitions (those with no {name}), Simple Type Definitions MUST be uniquely identified within a schema. Within a valid schema, each Simple Type Definition uniquely determines one datatype. The ·value space·, ·lexical space·, ·lexical mapping·, etc., of a Simple Type Definition are the ·value space·, ·lexical space·, etc., of the datatype uniquely determined (or "defined") by that Simple Type Definition.

If {variety} is ·atomic· then the ·value space· of the datatype defined will be a subset of the ·value space· of {base type definition} (which is a subset of the ·value space· of {primitive type definition}). If {variety} is ·list· then the ·value space· of the datatype defined will be the set of (possibly empty) finite-length sequences of values from the ·value space· of {item type definition}. If {variety} is ·union· then the ·value space· of the datatype defined will be a subset (possibly an improper subset) of the union of the ·value spaces· of each Simple Type Definition in {member type definitions}.

If {variety} is ·atomic· then the {variety} of {base type definition} must be ·atomic·, unless the {base type definition} is anySimpleType. If {variety} is ·list· then the {variety} of {item type definition} must be either ·atomic· or ·union·, and if {item type definition} is ·union· then all its ·basic members· MUST be ·atomic·. If {variety} is ·union· then {member type definitions} must be a list of Simple Type Definitions.

The {facets} property determines the ·value space· and ·lexical space· of the datatype being defined by imposing constraints which must be satisfied by values and ·lexical representations·.

The {fundamental facets} property provides some basic information about the datatype being defined: its cardinality, whether an ordering is defined for it by this specification, whether it has upper and lower bounds, and whether it is numeric.

If {final} is the empty set then the type can be used in deriving other types; the explicit values ***restriction***, ***list*** and ***union*** prevent further derivations of Simple Type Definitions by ·facet-based restriction·, ·list· and ·union· respectively; the explicit value ***extension*** prevents any derivation of Complex Type Definitions by extension.

The {context} property is only relevant for anonymous type definitions, for which its value is the component in which this type definition appears as the value of a property, e.g. {item type definition} or {base type definition}.

4.1.2 XML Representation of Simple Type Definition Schema Components

The XML representation for a Simple Type Definition schema component is a <simpleType> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `simpleType` Element Information Item et al.

```
<simpleType
  final = (#all | List of (list | union | restriction | extension))
  id = ID
```

```
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | list | union))
</simpleType>
```

```
<restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType?, (minExclusive | minInclusive |
maxExclusive | maxInclusive | totalDigits | fractionDigits | maxScale | minScale
| length | minLength | maxLength | enumeration | whiteSpace | pattern |
assertion | explicitTimezone | {any with namespace: ##other})*))
</restriction>
```

```
<list
  id = ID
  itemType = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, simpleType?)
</list>
```

```
<union
  id = ID
  memberTypes = List of QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, simpleType*)
</union>
```

Simple Type Definition **Schema Component**

Property	**Representation**
{name}	The actual value of the `name` [attribute], if present on the <simpleType> element, otherwise absent
{target namespace}	The actual value of the `targetNamespace` [attribute] of the parent `schema` element information item, if present, otherwise absent.
{base type definition}	The appropriate **case** among the following: 1 **If** the <restriction> alternative is chosen, **then** the type definition resolved to by the actual value of the `base` [attribute] of <restriction>, if present, otherwise the type definition corresponding to the <simpleType> among the [children] of <restriction>. 2 **If** the <list> or <union> alternative is chosen, **then** ·anySimpleType·.
{final}	A subset of {***restriction***, ***extension***, ***list***, ***union***}, determined as follows. [Definition:] Let **FS** be the actual value of the `final` [attribute], if present, otherwise the actual value of the `finalDefault` [attribute] of the ancestor `schema` element, if present, otherwise the empty string. Then the property value is the appropriate **case** among the following: 1 **If** ·FS· is the empty string, **then** the empty set; 2 **If** ·FS· is '`#all`', **then** {***restriction***, ***extension***, ***list***, ***union***}; 3 **otherwise** Consider ·FS· as a space-separated list, and include ***restriction*** if '`restriction`' is in that list, and similarly for ***extension***, ***list*** and ***union***.
{context}	The appropriate **case** among the following: 1 **If** the `name` [attribute] is present, **then** absent 2 **otherwise** the appropriate **case** among the following: 2.1 **If** the parent element information item is <attribute>, **then** the

	corresponding Attribute Declaration 2.2 **If** the parent element information item is <element>, **then** the corresponding Element Declaration 2.3 **If** the parent element information item is <list> or <union>, **then** the Simple Type Definition corresponding to the grandparent <simpleType> element information item 2.4 **otherwise** (the parent element information item is <restriction>), the appropriate **case** among the following: 2.4.1 **If** the grandparent element information item is <simpleType>, **then** the Simple Type Definition corresponding to the grandparent 2.4.2 **otherwise** (the grandparent element information item is <simpleContent>), the Simple Type Definition which is the {content type} of the Complex Type Definition corresponding to the great-grandparent <complexType> element information item.
{variety}	If the <list> alternative is chosen, then ***list***, otherwise if the <union> alternative is chosen, then ***union***, otherwise (the <restriction> alternative is chosen) the {variety} of the {base type definition}.
{facets}	The appropriate **case** among the following: 1 **If** the <restriction> alternative is chosen, **then** a set of Constraining Facet components constituting a restriction of the {facets} of the {base type definition} with respect to a set of Constraining Facet components corresponding to the appropriate element information items among the [children] of <restriction> (i.e. those which specify facets, if any), as defined in Schema Component Constraint: Simple Type Restriction (Facets) . 2 **If** the <list> alternative is chosen, **then** a set with one member, a whiteSpace facet with {value} = ***collapse*** and {fixed} = ***true***. 3 **otherwise** the empty set
{fundamental facets}	Based on {variety}, {facets}, {base type definition} and {member type definitions}, a set of Fundamental Facet components, one each as specified in The ordered Schema Component (§4.2.1.1), The bounded Schema Component (§4.2.2.1), The cardinality Schema Component (§4.2.3.1) and The numeric Schema Component (§4.2.4.1).
{annotations}	The annotation mapping of the set of elements containing the <simpleType>, and the <restriction>, the <list>, or the <union> [child], whichever is present, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

[Definition:] The **ancestors** of a type definition are its {base type definition} and the ·ancestors· of its {base type definition}. (The ancestors of a Simple Type Definition ***T*** in the type hierarchy are themselves type definitions; they are distinct from the XML elements which may be ancestors, in the XML document hierarchy, of the <simpleType> element which declares ***T***.)

If the {variety} is ***atomic***, the following additional property mapping also applies:

Atomic Simple Type Definition **Schema Component**

Property	Representation
{primitive	From among the ·ancestors· of this Simple Type Definition, that Simple

type definition}	Type Definition which corresponds to a ·primitive· datatype.

Example

An electronic commerce schema might define a datatype called 'SKU' (the barcode number that appears on products) from the ·built-in· datatype string by supplying a value for the ·pattern· facet.

```
<simpleType name='SKU'>
    <restriction base='string'>
      <pattern value='\d{3}-[A-Z]{2}'/>
    </restriction>
</simpleType>
```

In this case, 'SKU' is the name of the new ·user-defined· datatype, string is its ·base type· and ·pattern· is the facet.

If the {variety} is ***list***, the following additional property mappings also apply:

List Simple Type Definition **Schema Component**

Property	Representation
{item type definition}	The appropriate **case** among the following: 1 **If** the {base type definition} is ·anySimpleType·, **then** the Simple Type Definition (a) resolved to by the actual value of the itemType [attribute] of <list>, or (b) corresponding to the <simpleType> among the [children] of <list>, whichever is present. **Note:** In this case, a <list> element will invariably be present; it will invariably have either an itemType [attribute] or a <simpleType> [child], but not both. 2 **otherwise** (that is, the {base type definition} is not ·anySimpleType·), the {item type definition} of the {base type definition}. **Note:** In this case, a <restriction> element will invariably be present.

Example

A system might want to store lists of floating point values.

```
<simpleType name='listOfFloat'>
  <list itemType='float'/>
</simpleType>
```

In this case, *listOfFloat* is the name of the new ·user-defined· datatype, float is its ·item type· and ·list· is the derivation method.

If the {variety} is ***union***, the following additional property mappings also apply:

Union Simple Type Definition **Schema Component**

Property	Representation
{member type definitions}	The appropriate **case** among the following: 1 **If** the {base type definition} is ·anySimpleType·, **then** the sequence of (a) the Simple Type Definitions (a) resolved to by the items in the actual value of the `memberTypes` [attribute] of <union>, if any, and (b) those corresponding to the <simpleType>s among the [children] of <union>, if any, in order. **Note:** In this case, a <union> element will invariably be present; it will invariably have either a `memberTypes` [attribute] or one or more <simpleType> [children], or both. 2 **otherwise** (that is, the {base type definition} is not ·anySimpleType·), the {member type definitions} of the {base type definition}. **Note:** In this case, a <restriction> element will invariably be present.

Example

As an example, taken from a typical display oriented text markup language, one might want to express font sizes as an integer between 8 and 72, or with one of the tokens "small", "medium" or "large". The ·union· Simple Type Definition below would accomplish that.

```
<xsd:attribute name="size">
  <xsd:simpleType>
    <xsd:union>
      <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
          <xsd:minInclusive value="8"/>
          <xsd:maxInclusive value="72"/>
        </xsd:restriction>
      </xsd:simpleType>
      <xsd:simpleType>
        <xsd:restriction base="xsd:NMTOKEN">
          <xsd:enumeration value="small"/>
          <xsd:enumeration value="medium"/>
          <xsd:enumeration value="large"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:union>
  </xsd:simpleType>
</xsd:attribute>
```

```
<p>
<font size='large'>A header</font>
</p>
<p>
<font size='12'>this is a test</font>
</p>
```

A datatype can be ·constructed· from a ·primitive· datatype or an ·ordinary· datatype by one of three means: by ·facet-based restriction·, by ·list· or by ·union·.

4.1.3 Constraints on XML Representation of Simple Type Definition

Schema Representation Constraint: itemType attribute or simpleType child

Either the `itemType` [attribute] or the <simpleType> [child] of the <list> element must be present, but not both.

Schema Representation Constraint: base attribute or simpleType child

Either the `base` [attribute] or the `simpleType` [child] of the <restriction> element must be present, but not both.

Schema Representation Constraint: memberTypes attribute or simpleType children

Either the `memberTypes` [attribute] of the <union> element must be non-empty or there must be at least one `simpleType` [child].

4.1.4 Simple Type Definition Validation Rules

Validation Rule: Facet Valid

A value in a ·value space· is facet-valid with respect to a ·constraining facet· component if and only if:

1 the value is facet-valid with respect to the particular ·constraining facet· as specified below.

Validation Rule: Datatype Valid

A ·literal· is datatype-valid with respect to a Simple Type Definition if and only if it is a member of the ·lexical space· of the corresponding datatype.

Note: Since every value in the ·value space· is denoted by some ·literal·, and every ·literal· in the ·lexical space· maps to some value, the requirement that the ·literal· be in the ·lexical space· entails the requirement that the value it maps to should fulfill all of the constraints imposed by the {facets} of the datatype. If the datatype is a ·list·, the Datatype Valid constraint also entails that each whitespace-delimited token in the list be datatype-valid against the ·item type· of the list. If the datatype is a ·union·, the Datatype Valid constraint entails that the ·literal· be datatype-valid against at least one of the ·member types·.

That is, the constraints on Simple Type Definitions and on datatype derivation defined in this specification have as a consequence that a ·literal· ***L*** is datatype-valid with respect to a Simple Type Definition ***T*** if and only if either ***T*** corresponds to a ·special· datatype or **all** of the following are true:

1 If there is a pattern in {facets}, then ***L*** is pattern valid (§4.3.4.4) with respect to the pattern. If there are other ·lexical· facets in {facets}, then ***L*** is facet-valid with respect to them.

2 The appropriate case among the following is true:

2.1 If the {variety} of ***T*** is ·atomic·, then ***L*** is in the ·lexical space· of the {primitive type definition} of ***T***, as defined in the appropriate documentation. Let ***V*** be the member of the ·value space· of the {primitive type definition} of ***T*** mapped to by ***L***, as defined in the appropriate documentation.

Note: For ·built-in· ·primitives·, the "appropriate documentation" is the relevant section of this specification. For ·implementation-defined· ·primitives·, it is the normative specification of the ·primitive·, which will typically be included in, or referred to from, the implementation's documentation.

2.2 If the {variety} of ***T*** is ·list·, then each space-delimited substring of ***L*** is Datatype Valid with respect to the {item type definition} of ***T***. Let ***V*** be the sequence consisting of the values identified by Datatype Valid for each of those substrings, in order.
2.3 If the {variety} of ***T*** is ·union·, then ***L*** is Datatype Valid with respect to at least one member of the {member type definitions} of ***T***. Let ***B*** be the ·active basic member· of ***T*** for ***L***. Let ***V*** be the value identified by Datatype Valid for ***L*** with respect to ***B***.

3 ***V***, as determined by the appropriate sub-clause of clause 2 above, is Facet Valid (§4.1.4) with respect to each member of the {facets} of ***T*** which is a ·value-based· (and not a ·pre-lexical· or ·lexical·) facet.

Note that whiteSpace facets and other ·pre-lexical· facets do not take part in checking Datatype Valid. In cases where this specification is used in conjunction with schema-validation of XML documents, such facets are used to normalize infoset values *before* the normalized results are checked for datatype validity. In the case of unions the ·pre-lexical· facets to use are those associated with ***B*** in clause 2.3 above. When more than one ·pre-lexical· facet applies, the whiteSpace facet is applied first; the order in which ·implementation-defined· facets are applied is ·implementation-defined·.

4.1.5 Constraints on Simple Type Definition Schema Components

Schema Component Constraint: Applicable Facets

The ·constraining facets· which are allowed to be members of {facets} depend on the {variety} and {primitive type definition} of the type, as follows:

If {variety} is ***absent***, then no facets are applicable. (This is true for anySimpleType.)

If {variety} is list, then the applicable facets are assertions, length, minLength, maxLength, pattern, enumeration, and whiteSpace.

If {variety} is union, then the applicable facets are pattern, enumeration, and assertions.

If {variety} is atomic, and {primitive type definition} is ***absent*** then no facets are applicable. (This is true for anyAtomicType.)

In all other cases ({variety} is atomic and {primitive type definition} is not ***absent***), then the applicable facets are shown in the table below.

{primitive type definition}	applicable {facets}
string	length, minLength, maxLength, pattern, enumeration, whiteSpace, assertions
boolean	pattern, whiteSpace, assertions
float	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions
double	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions

decimal	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions
precisionDecimal	totalDigits, maxScale, minScale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions
duration	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions
dateTime	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions, explicitTimezone
time	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions, explicitTimezone
date	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions, explicitTimezone
gYearMonth	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions, explicitTimezone
gYear	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions, explicitTimezone
gMonthDay	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions, explicitTimezone
gDay	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions, explicitTimezone
gMonth	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive, assertions, explicitTimezone
hexBinary	length, minLength, maxLength, pattern, enumeration, whiteSpace, assertions
base64Binary	length, minLength, maxLength, pattern, enumeration, whiteSpace, assertions
anyURI	length, minLength, maxLength, pattern, enumeration, whiteSpace, assertions
QName	length, minLength, maxLength, pattern, enumeration, whiteSpace, assertions
NOTATION	length, minLength, maxLength, pattern, enumeration, whiteSpace, assertions

Note: For any ·implementation-defined· primitive types, it is ·implementation-defined· which constraining facets are applicable to them.

Similarly, for any ·implementation-defined· constraining facets, it is ·implementation-defined· which ·primitives· they apply to.

4.1.6 Built-in Simple Type Definitions

The Simple Type Definition of anySimpleType is present in every schema. It has the following properties:

Simple type definition of `anySimpleType`

Property	Value
{name}	'anySimpleType'
{target namespace}	'http://www.w3.org/2001/XMLSchema'
{final}	The empty set
{context}	***absent***
{base type definition}	anyType
{facets}	The empty set
{fundamental facets}	The empty set
{variety}	***absent***
{primitive type definition}	***absent***
{item type definition}	***absent***
{member type definitions}	***absent***
{annotations}	The empty sequence

The definition of anySimpleType is the root of the Simple Type Definition hierarchy; as such it mediates between the other simple type definitions, which all eventually trace back to it via their {base type definition} properties, and the definition of ***anyType***, which is *its* {base type definition}.

The Simple Type Definition of anyAtomicType is present in every schema. It has the following properties:

Simple type definition of anyAtomicType	
Property	**Value**
{name}	'anyAtomicType'
{target namespace}	'http://www.w3.org/2001/XMLSchema'
{final}	The empty set
{context}	***absent***
{base type definition}	anySimpleType
{facets}	The empty set
{fundamental	The empty set

facets}	
{variety}	***atomic***
{primitive type definition}	***absent***
{item type definition}	***absent***
{member type definitions}	***absent***
{annotations}	The empty sequence

Simple type definitions for all the built-in primitive datatypes, namely string, boolean, float, double, decimal, precisionDecimal, dateTime, duration, time, date, gMonth, gMonthDay, gDay, gYear, gYearMonth, hexBinary, base64Binary, anyURI are present by definition in every schema. All have a very similar structure, with only the {name}, the {primitive type definition} (which is self-referential), the {fundamental facets}, and in one case the {facets} varying from one to the next:

Simple Type Definition corresponding to the built-in primitive datatypes

Property	**Value**
{name}	[as appropriate]
{target namespace}	'`http://www.w3.org/2001/XMLSchema`'
{base type definition}	anyAtomicType Definition
{final}	The empty set
{variety}	***atomic***
{primitive type definition}	[this Simple Type Definition itself]
{facets}	{a whiteSpace facet with {value} = ***collapse*** and {fixed} = ***true*** in all cases except string, which has {value} = ***preserve*** and {fixed} = ***false***}
{fundamental facets}	[as appropriate]
{context}	***absent***
{item type definition}	***absent***
{member type definitions}	***absent***
{annotations}	The empty sequence

·Implementation-defined· ·primitives· MUST have a Simple Type Definition with the values shown above, with the following exceptions.

1. The {facets} property MUST contain a whiteSpace facet, the value of which is ·implementation-defined·. It MAY contain other facets, whether defined in this specification or ·implementation-defined·.

2. The value of {fundamental facets} is ·implementation-defined·.

3. The value of {annotations} MAY be empty, but need not be.

Note: It is a consequence of the rule just given that each ·implementation-defined· ·primitive· will have an expanded name by which it can be referred to.

Note: ·Implementation-defined· datatypes will normally have a value other than '`http://www.w3.org/2001/XMLSchema`' for the {target namespace} property. That namespace is controlled by the W3C and datatypes will be added to it only by W3C or its designees.

Similarly, Simple Type Definitions for all the built-in ·ordinary· datatypes are present by definition in every schema, with properties as specified in Other Built-in Datatypes (§3.4) and as represented in XML in Illustrative XML representations for the built-in ordinary type definitions (§C.2).

Simple Type Definition corresponding to the built-in ordinary datatypes

Property	**Value**
{name}	[as appropriate]
{target namespace}	'`http://www.w3.org/2001/XMLSchema`'
{base type definition}	[as specified in the appropriate sub-section of Other Built-in Datatypes (§3.4)]
{final}	The empty set
{variety}	[***atomic*** or ***list***, as specified in the appropriate sub-section of Other Built-in Datatypes (§3.4)]
{primitive type definition}	[if {variety} is ***atomic***, then the {primitive type definition} of the {base type definition}, otherwise ***absent***]
{facets}	[as specified in the appropriate sub-section of Other Built-in Datatypes (§3.4)]
{fundamental facets}	[as specified in the appropriate sub-section of Other Built-in Datatypes (§3.4)]
{context}	***absent***
{item type definition}	if {variety} is ***atomic***, then ***absent***, otherwise as specified in the appropriate sub-section of Other Built-in Datatypes (§3.4)]
{member type definitions}	***absent***
{annotations}	As shown in the XML representations of the ordinary built-in datatypes in Illustrative XML representations for the built-in ordinary

type definitions (§C.2)

4.2 Fundamental Facets

[Definition:] Each **fundamental facet** is a schema component that provides a limited piece of information about some aspect of each datatype. All ·fundamental facet· components are defined in this section. For example, cardinality is a ·fundamental facet·. Most ·fundamental facets· are given a value fixed with each primitive datatype's definition, and this value is not changed by subsequent ·derivations· (even when it would perhaps be reasonable to expect an application to give a more accurate value based on the constraining facets used to define the ·derivation·). The cardinality and bounded facets are exceptions to this rule; their values may change as a result of certain ·derivations·.

> **Note:** Schema components are identified by kind. "Fundamental" is not a kind of component. Each kind of ·fundamental facet· ("ordered", "bounded", etc.) is a separate kind of schema component.

A ·fundamental facet· can occur only in the {fundamental facets} of a Simple Type Definition, and this is the only place where ·fundamental facet· components occur. Each kind of ·fundamental facet· component occurs (once) in each Simple Type Definition's {fundamental facets} set.

> **Note:** The value of any ·fundamental facet· component can always be calculated from other properties of its ·owner·. Fundamental facets are not required for schema processing, but some applications use them.

4.2.1 ordered

For some datatypes, this document specifies an order relation for their value spaces (see Order (§2.2.3)); the *ordered* facet reflects this. It takes the values ***total***, ***partial***, and ***false***, with the meanings described below. For the ·primitive· datatypes, the value of the *ordered* facet is specified in Fundamental Facets (§F.1). For ·ordinary· datatypes, the value is inherited without change from the ·base type·. For a ·list·, the value is always ***false***; for a ·union·, the value is computed as described below.

A ***false*** value means no order is prescribed; a ***total*** value assures that the prescribed order is a total order; a ***partial*** value means that the prescribed order is a partial order, but not (for the primitive type in question) a total order.

> **Note:** The value ***false*** in the *ordered* facet does not mean no partial or total ordering *exists* for the value space, only that none is specified by this document for use in checking upper and lower bounds. Mathematically, any set of values possesses least one

trivial partial ordering, in which every value pair that is not equal is incomparable.

> **Note:** When new datatypes are derived from datatypes with partial orders, the constraints imposed can sometimes result in a value space for which the ordering is total, or trivial. The value of the ordered facet is not, however, changed to reflect this. The value ***partial*** should therefore be interpreted with appropriate caution.

[Definition:] A ·value space·, and hence a datatype, is said to be **ordered** if some members of the ·value space· are drawn from a ·primitive· datatype for which the table in Fundamental Facets (§F.1) specifies the value ***total*** or ***partial*** for the *ordered* facet.

> **Note:** Some of the "real-world" datatypes which are the basis for those defined herein are ordered in some applications, even though no order is prescribed for schema-processing purposes. For example, boolean is sometimes ordered, and string and ·list· datatypes ·constructed· from ordered ·atomic· datatypes are sometimes given "lexical" orderings. They are *not* ordered for schema-processing purposes.

4.2.1.1 The ordered Schema Component

Schema Component: ordered, a kind of Fundamental Facet
`{value}` One of {***false***, ***partial***, ***total***}. Required.

{value} depends on the ·owner's· {variety}, {facets}, and {member type definitions}. The appropriate **case** among the following MUST be true:

1 **If** the ·owner's· {variety} is ***atomic***, **then** the appropriate **case** among the following MUST be true:
  1.1 **If** the ·owner· is ·primitive·, **then** {value} is as specified in the table in Fundamental Facets (§F.1).
  1.2 **otherwise** {value} is the ·owner's· {base type definition}'s ordered {value}.
2 **If** the ·owner's· {variety} is ***list***, **then** {value} is ***false***.
3 **otherwise** the ·owner's· {variety} is ***union***; the appropriate **case** among the following MUST be true:
  3.1 **If** every ·basic member· of the ·owner· has {variety} atomic and has the same {primitive type definition}, **then** {value} is the same as the ordered component's {value} in that primitive type definition's {fundamental facets}.
  3.2 **If** each member of the ·owner's· {member type definitions} has an ordered component in its {fundamental facets} whose {value} is ***false***, **then** {value} is ***false***.
  3.3 **otherwise** {value} is ***partial***.

4.2.2 bounded

Some ordered datatypes have the property that there is one value greater than or equal to every other value, and another that is less than or equal to every other value. (In the case of ·ordinary· datatypes, these two values are not necessarily in the value space of the derived datatype, but they must be in the value space of the primitive datatype from which they have been derived.) The *bounded* facet value is boolean and is generally ***true*** for such *bounded* datatypes. However, it will remain ***false*** when the mechanism for imposing such a bound is difficult to detect, as, for example, when the boundedness occurs because of derivation using a pattern component.

4.2.2.1 The bounded Schema Component

Schema Component: bounded, a kind of Fundamental Facet
`{value}` An xs:boolean value. Required.

{value} depends on the ·owner's· {variety}, {facets} and {member type definitions}.

When the ·owner· is ·primitive·, {value} is as specified in the table in Fundamental Facets (§F.1). Otherwise, when the ·owner's· {variety} is ***atomic***, if one of minInclusive or minExclusive and one of maxInclusive or maxExclusive are members of the ·owner's· {facets} set, then {value} is ***true***; otherwise {value} is ***false***.

When the ·owner's· {variety} is ***list***, {value} is ***false***.

When the ·owner's· {variety} is ***union***, if {value} is ***true*** for every member of the ·owner's· {member type definitions} set and all of the ·owner's· ·basic members· have the same {primitive type definition}, then {value} is ***true***; otherwise {value} is ***false***.

4.2.3 cardinality

Every value space has a specific number of members. This number can be characterized as *finite* or *infinite*. (Currently there are no datatypes with infinite value spaces larger than *countable*.) The *cardinality* facet value is either ***finite*** or ***countably infinite*** and is generally ***finite*** for datatypes with finite value spaces. However, it will remain ***countably infinite*** when the mechanism for causing finiteness is difficult to detect, as, for example, when finiteness occurs because of a derivation using a pattern component.

4.2.3.1 The cardinality Schema Component

Schema Component: cardinality, a kind of Fundamental Facet
`{value}` One of {***finite***, ***countably infinite***}. Required.

{value} depends on the ·owner's· {variety}, {facets}, and {member type definitions}.

When the ·owner· is ·primitive·, {value} is as specified in the table in Fundamental Facets (§F.1). Otherwise, when the ·owner's· {variety} is ***atomic***, {value} is ***countably infinite*** unless **any** of the following conditions are true, in which case {value} is ***finite***:

1. the ·owner's· {base type definition}'s cardinality {value} is ***finite***,
2. at least one of length, maxLength, or totalDigits is a member of the ·owner's· {facets} set,

3. **all** of the following are true:

    a. one of minInclusive or minExclusive is a member of the ·owner's· {facets} set

    b. one of maxInclusive or maxExclusive is a member of the ·owner's· {facets} set

    c. **either** of the following are true:

        i. fractionDigits is a member of the ·owner's· {facets} set

        ii. {primitive type definition} is one of date, gYearMonth, gYear, gMonthDay, gDay or gMonth

When the ·owner's· {variety} is ***list***, if length or both minLength and maxLength are members of the ·owner's· {facets} set and the ·owner's· {item type definition}'s cardinality {value} is ***finite*** then {value} is ***finite***; otherwise {value} is ***countably infinite***.

When the ·owner's· {variety} is ***union***, if cardinality's {value} is *finite* for every member of the ·owner's· {member type definitions} set then {value} is ***finite***, otherwise {value} is ***countably infinite***.

4.2.4 numeric

Some value spaces are made up of things that are conceptually *numeric*, others are not. The *numeric* facet value indicates which are considered numeric.

4.2.4.1 The numeric Schema Component

Schema Component: numeric, a kind of Fundamental Facet
`{value}` An xs:boolean value. Required.

{value} depends on the ·owner's· {variety}, {facets}, {base type definition} and {member type definitions}.

When the ·owner· is ·primitive·, {value} is as specified in the table in Fundamental Facets (§F.1). Otherwise, when the ·owner's· {variety} is ***atomic***, {value} is inherited from the ·owner's· {base type definition}'s numeric{value}.

When the ·owner's· {variety} is ***list***, {value} is ***false***.

When the ·owner's· {variety} is ***union***, if numeric's {value} is ***true*** for every member of the ·owner's· {member type definitions} set then {value} is ***true***, otherwise {value} is ***false***.

4.3 Constraining Facets

[Definition:] **Constraining facets** are schema components whose values may be set or changed during ·derivation· (subject to facet-specific controls) to control various aspects of the derived datatype. All ·constraining facet· components defined by this specification are defined in this section. For example, whiteSpace is a ·constraining facet·. ·Constraining Facets· are given a value as part of the ·derivation· when an ·ordinary· datatype is defined by ·restricting· a ·primitive· or ·ordinary· datatype; a few ·constraining facets· have default values that are also provided for ·primitive· datatypes.

> **Note:** Schema components are identified by kind. "Constraining" is not a kind of component. Each kind of ·constraining facet· ("whiteSpace", "length", etc.) is a separate kind of schema component.

This specification distinguishes three kinds of constraining facets:

- [Definition:] A constraining facet which is used to normalize an initial ·literal· before checking to see whether the resulting character sequence is a member of a datatype's ·lexical space· is a **pre-lexical** facet.

  This specification defines just one ·pre-lexical· facet: whiteSpace.

- [Definition:] A constraining facet which directly restricts the ·lexical space· of a datatype is a **lexical** facet.

  This specification defines just one ·lexical· facet: pattern.

  > **Note:** As specified normatively elsewhere, ·lexical· facets can have an indirect effect on the ·value space·: if every lexical representation of a value is removed from the ·lexical space·, the value itself is removed from the ·value space·.

- [Definition:] A constraining facet which directly restricts the ·value space· of a datatype is a **value-based** facet.

  Most of the constraining facets defined by this specification are ·value-based· facets.

  > **Note:** As specified normatively elsewhere, ·value-based· facets can have an indirect effect on the ·lexical space·: if a value is removed from the ·value space·, its lexical representations are removed from the ·lexical space·.

Conforming processors MUST support all the facets defined in this section. It is ·implementation-defined· whether a processor supports other constraining facets. [Definition:] An ·constraining facet· which is not supported by the processor in use is **unknown**.

> **Note:** A reference to an ·unknown· facet might be a reference to an ·implementation-defined· facet supported by some other processor, or might be the result of a typographic error, or might have some other explanation.

The descriptions of individual facets given below include both constraints on Simple Type Definition components and rules for checking the datatype validity of a given literal against a given datatype. The validation rules typically depend upon having a full knowledge of the datatype; full knowledge of the datatype, in turn, depends on having a fully instantiated Simple Type Definition. A full instantiation of the Simple Type Definition, and the checking of the component constraints, require knowledge of the ·base type·. It follows that if a datatype's ·base type· is ·unknown·, the Simple Type Definition defining the datatype will be incompletely instantiated, and the datatype itself will be ·unknown·. Similarly, any datatype defined using an ·unknown· ·constraining facet· will be ·unknown·. It is not possible to perform datatype validation as defined here using ·unknown· datatypes.

> **Note:** The preceding paragraph does not forbid implementations from attempting to make use of such partial information as they have about ·unknown· datatypes. But the exploitation of such partial knowledge is not datatype validity checking as defined here and is to be distinguished from it in the implementation's documentation and interface.

4.3.1 length

[Definition:] **length** is the number of *units of length*, where *units of length* varies depending on the type that is being derived from. The value of **length** ·must· be a nonNegativeInteger.

For string and datatypes derived from string, **length** is measured in units of characters as defined in [XML]. For anyURI, **length** is measured in units of characters (as for string). For hexBinary and base64Binary and datatypes derived from them, **length** is measured in octets (8 bits) of binary data. For datatypes ·constructed· by ·list·, **length** is measured in number of list items.

> **Note:** For string and datatypes derived from string, **length** will not always coincide with "string length" as perceived by some users or with the number of storage units in some digital representation. Therefore, care should be taken when specifying a value for **length** and in attempting to infer storage requirements from a given value for **length**.

·length· provides for:

- Constraining a ·value space· to values with a specific number of *units of length*, where *units of length* varies depending on {base type definition}.

Example

The following is the definition of a ·user-defined· datatype to represent product codes which must be exactly 8 characters in length. By fixing the value of the **length** facet we ensure that types derived from productCode can change or set the values of other facets, such as **pattern**, but cannot change the length.

```
<simpleType name='productCode'>
   <restriction base='string'>
     <length value='8' fixed='true'/>
   </restriction>
</simpleType>
```

4.3.1.1 The length Schema Component

Schema Component: length, a kind of Constraining Facet

{annotations}
    A sequence of Annotation components.
{value}
    An xs:nonNegativeInteger value. Required.
{fixed}
    An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} cannot specify a value for length other than {value}.

> **Note:** The {fixed} property is defined for parallelism with other facets and for compatiblity with version 1.0 of this specification. But it is a consequence of length valid restriction (§4.3.1.4) that the value of the length facet cannot be changed, regardless of whether {fixed} is *true* or *false*.

4.3.1.2 XML Representation of length Schema Components

The XML representation for a length schema component is a <length> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `length` Element Information Item

```
<length
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</length>
```

length Schema Component	
Property	**Representation**
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <length> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.1.3 length Validation Rules

Validation Rule: Length Valid

A value in a ·value space· is facet-valid with respect to ·length· if and only if:

1 if the {variety} is ·atomic· then
  1.1 if {primitive type definition} is string or anyURI, then the length of the value, as measured in characters ·must· be equal to {value};
  1.2 if {primitive type definition} is hexBinary or base64Binary, then the length of the value, as measured in octets of the binary data, ·must· be equal to {value};
  1.3 if {primitive type definition} is QName or NOTATION, then any {value} is facet-valid.
2 if the {variety} is ·list·, then the length of the value, as measured in list items, ·must· be equal to {value}

The use of ·length· on QName, NOTATION, and datatypes derived from them is deprecated. Future versions of this specification may remove this facet for these datatypes.

4.3.1.4 Constraints on length Schema Components

Schema Component Constraint: length and minLength or maxLength

If length is a member of {facets} then
1 It is an error for minLength to be a member of {facets} unless
  1.1 the {value} of minLength <= the {value} of length and
  1.2 there is some type definition from which this one is derived by one or more ·restriction· steps in which minLength has the same {value} and length is not specified.
2 It is an error for maxLength to be a member of {facets} unless
  2.1 the {value} of length <= the {value} of maxLength and
  2.2 there is some type definition from which this one is derived by one or more restriction steps in which maxLength has the same {value} and length is not specified.

Schema Component Constraint: length valid restriction

It is an ·error· if length is among the members of {facets} of {base type definition} and {value} is not equal to the {value} of the parent length.

4.3.2 minLength

[Definition:] **minLength** is the minimum number of *units of length*, where *units of length* varies depending on the type that is being derived from. The value of **minLength** ·must· be a nonNegativeInteger.

For string and datatypes derived from string, **minLength** is measured in units of characters as defined in [XML]. For hexBinary and base64Binary and datatypes derived from them, **minLength** is measured in octets (8 bits) of binary data. For datatypes ·constructed· by ·list·, **minLength** is measured in number of list items.

> **Note:** For string and datatypes derived from string, **minLength** will not always coincide with "string length" as perceived by some users or with the number of storage units in some digital representation. Therefore, care should be taken when specifying a value for **minLength** and in attempting to infer storage requirements from a given value for **minLength**.

·minLength· provides for:

- Constraining a ·value space· to values with at least a specific number of *units of length*, where *units of length* varies depending on {base type definition}.

Example

The following is the definition of a ·user-defined· datatype which requires strings to have at least one character (i.e., the empty string is not in the ·value space· of this datatype).

```
<simpleType name='non-empty-string'>
  <restriction base='string'>
    <minLength value='1'/>
  </restriction>
</simpleType>
```

4.3.2.1 The minLength Schema Component

Schema Component: minLength, a kind of Constraining Facet

`{annotations}`
A sequence of Annotation components.

`{value}`
An xs:nonNegativeInteger value. Required.

`{fixed}`
An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} cannot specify a value for minLength other than {value}.

4.3.2.2 XML Representation of minLength Schema Component

The XML representation for a minLength schema component is a <minLength> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `minLength` Element Information Item

```
<minLength
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</minLength>
```

minLength Schema Component

Property	**Representation**
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <minLength> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.2.3 minLength Validation Rules

Validation Rule: minLength Valid

A value in a ·value space· is facet-valid with respect to ·minLength·, determined as follows:

1 if the {variety} is ·atomic· then
  1.1 if {primitive type definition} is string or anyURI, then the length of the value, as measured in characters ·must· be greater than or equal to {value};
  1.2 if {primitive type definition} is hexBinary or base64Binary, then the length of the value, as measured in octets of the binary data, ·must· be greater than or equal to {value};
  1.3 if {primitive type definition} is QName or NOTATION, then any {value} is facet-valid.
2 if the {variety} is ·list·, then the length of the value, as measured in list items, ·must· be greater than or equal to {value}

The use of ·minLength· on QName, NOTATION, and datatypes derived from them is deprecated. Future versions of this specification may remove this facet for these datatypes.

4.3.2.4 Constraints on minLength Schema Components

Schema Component Constraint: minLength <= maxLength

If both minLength and maxLength are members of {facets}, then the {value} of minLength ·must· be less than or equal to the {value} of maxLength.

Schema Component Constraint: minLength valid restriction

It is an ·error· if minLength is among the members of {facets} of {base type definition} and {value} is less than the {value} of the parent minLength.

4.3.3 maxLength

[Definition:] **maxLength** is the maximum number of *units of length*, where *units of length* varies depending on the type that is being derived from. The value of **maxLength** ·must· be a nonNegativeInteger.

For string and datatypes derived from string, **maxLength** is measured in units of characters as defined in [XML]. For hexBinary and base64Binary and datatypes derived from them, **maxLength** is measured in octets (8 bits) of binary data. For datatypes ·constructed· by ·list·, **maxLength** is measured in number of list items.

> **Note:** For string and datatypes derived from string, **maxLength** will not always coincide with "string length" as perceived by some users or with the number of storage units in some digital representation. Therefore, care should be taken when specifying a value for **maxLength** and in attempting to infer storage requirements from a given value for **maxLength**.

·maxLength· provides for:

- Constraining a ·value space· to values with at most a specific number of *units of length*, where *units of length* varies depending on {base type definition}.

Example

The following is the definition of a ·user-defined· datatype which might be used to accept form input with an upper limit to the number of characters that are acceptable.

```
<simpleType name='form-input'>
  <restriction base='string'>
    <maxLength value='50'/>
  </restriction>
</simpleType>
```

4.3.3.1 The maxLength Schema Component

Schema Component: maxLength, a kind of Constraining Facet

`{annotations}`
: A sequence of Annotation components.

`{value}`
: An xs:nonNegativeInteger value. Required.

`{fixed}`
: An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} cannot specify a value for maxLength other than {value}.

4.3.3.2 XML Representation of maxLength Schema Components

The XML representation for a maxLength schema component is a <maxLength> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `maxLength` Element Information Item

```
<maxLength
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</maxLength>
```

maxLength Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <maxLength> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.3.3 maxLength Validation Rules

Validation Rule: maxLength Valid

A value in a ·value space· is facet-valid with respect to ·maxLength·, determined as follows:

1 if the {variety} is ·atomic· then
  1.1 if {primitive type definition} is string or anyURI, then the length of the value, as measured in characters ·must· be less than or equal to {value};
  1.2 if {primitive type definition} is hexBinary or base64Binary, then the length of the value, as measured in octets of the binary data, ·must· be less than or equal to {value};
  1.3 if {primitive type definition} is QName or NOTATION, then any {value} is facet-valid.
2 if the {variety} is ·list·, then the length of the value, as measured in list items, ·must· be less than or equal to {value}

The use of ·maxLength· on QName, NOTATION, and datatypes derived from them is deprecated. Future versions of this specification may remove this facet for these datatypes.

4.3.3.4 Constraints on maxLength Schema Components

Schema Component Constraint: maxLength valid restriction

It is an ·error· if maxLength is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent maxLength.

4.3.4 pattern

[Definition:] **pattern** is a constraint on the ·value space· of a datatype which is achieved by constraining the ·lexical space· to ·literals· which match each member of a set of patterns. The value of **pattern** MUST be a set of ·regular expressions·.

·pattern· provides for:

- Constraining a ·value space· to values that are denoted by ·literals· which match each of a set of ·regular expressions·.

Example

The following is the definition of a ·user-defined· datatype which is a better representation of postal codes in the United States, by limiting strings to those which are matched by a specific ·regular expression·.

```
<simpleType name='better-us-zipcode'>
  <restriction base='string'>
    <pattern value='[0-9]{5}(-[0-9]{4})?'/>
  </restriction>
</simpleType>
```

4.3.4.1 The pattern Schema Component

Schema Component: pattern, a kind of Constraining Facet

{annotations}
A sequence of Annotation components.
{value}
A non-empty set of ·regular expressions·.

4.3.4.2 XML Representation of pattern Schema Components

The XML representation for a pattern schema component is one or more <pattern> element information items. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `pattern` Element Information Item

```
<pattern
  id = ID
  value = string
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</pattern>
```

pattern Schema Component

Property	Representation
{value}	[Definition:] Let **R** be a regular expression given by the appropriate **case** among the following: 1 **If** there is only one <pattern> among the [children] of a <restriction>, **then** the actual value of its `value` [attribute] 2 **otherwise** the concatenation of the actual values of all the <pattern> [children]'s `value` [attributes], in order, separated by '`|`', so forming a single regular expression with multiple ·branches·. The value is then given by the appropriate **case** among the following:

	1 **If** the {base type definition} of the ·owner· has a pattern facet among its {facets}, **then** the union of that pattern facet's {value} and {·R·} 2 **otherwise** just {·R·}
{annotations}	The annotation mapping of the set containing all of the <pattern> elements among the [children] of the <restriction> element information item, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

Note: The {value} property will only have more than one member when ·facet-based restriction· involves a pattern facet at more than one step in a type derivation. During validation, lexical forms will be checked against every member of the resulting {value}, effectively creating a conjunction of patterns.

In summary, ·pattern· facets specified on the *same* step in a type derivation are **OR**ed together, while ·pattern· facets specified on *different* steps of a type derivation are **AND**ed together.

Thus, to impose two ·pattern· constraints simultaneously, schema authors may either write a single ·pattern· which expresses the intersection of the two ·pattern·s they wish to impose, or define each ·pattern· on a separate type derivation step.

4.3.4.3 Constraints on XML Representation of pattern

Schema Representation Constraint: Pattern value

The actual value of the `value` [attribute] must be a ·regular expression· as defined in Regular Expressions (§G).

4.3.4.4 pattern Validation Rules

Validation Rule: pattern valid

A ·literal· in a ·lexical space· is pattern-valid (or: facet-valid with respect to ·pattern·) if and only if for each ·regular expression· in its {value}, the ·literal· is among the set of character sequences denoted by the ·regular expression·.

Note: As noted in Datatype (§2.1), certain uses of the ·pattern· facet may eliminate from the lexical space the canonical forms of some values in the value space; this can be inconvenient for applications which write out the canonical form of a value and rely on being able to read it in again as a legal lexical form. This specification provides no recourse in such situations; applications are free to deal with it as they see fit. Caution is advised.

4.3.4.5 Constraints on pattern Schema Components

Schema Component Constraint: Valid restriction of pattern

It is an ·error· if there is any member of the {value} of the pattern facet on the {base type definition} which is not also a member of the {value}.

Note: For components constructed from XML representations in schema documents, the satisfaction of this constraint is a consequence of the XML mapping rules: any pattern imposed by a simple type definition ***S*** will always also be imposed by any type derived from ***S*** by ·facet-based restriction·. This constraint ensures that components constructed by other means (so-called "born-binary" components) similarly preserve pattern facets across ·facet-based restriction·.

4.3.5 enumeration

[Definition:] **enumeration** constrains the ·value space· to a specified set of values.

enumeration does not impose an order relation on the ·value space· it creates; the value of the ·ordered· property of the derived datatype remains that of the datatype from which it is derived.

·enumeration· provides for:

- Constraining a ·value space· to a specified set of values.

Example

The following example is a Simple Type Definition for a ·user-defined· datatype which limits the values of dates to the three US holidays enumerated. This Simple Type Definition would appear in a schema authored by an "end-user" and shows how to define a datatype by enumerating the values in its ·value space·. The enumerated values must be type-valid ·literals· for the ·base type·.

```
<simpleType name='holidays'>
    <annotation>
        <documentation>some US holidays</documentation>
    </annotation>
    <restriction base='gMonthDay'>
      <enumeration value='--01-01'>
        <annotation>
            <documentation>New Year's day</documentation>
        </annotation>
      </enumeration>
      <enumeration value='--07-04'>
        <annotation>
            <documentation>4th of July</documentation>
        </annotation>
      </enumeration>
      <enumeration value='--12-25'>
        <annotation>
            <documentation>Christmas</documentation>
        </annotation>
      </enumeration>
    </restriction>
</simpleType>
```

4.3.5.1 The enumeration Schema Component

Schema Component: enumeration, a kind of Constraining Facet

{annotations}
: A sequence of Annotation components.

{value}
: A set of values from the ·value space· of the {base type definition}.

4.3.5.2 XML Representation of enumeration Schema Components

The XML representation for an enumeration schema component is one or more <enumeration> element information items. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `enumeration` Element Information Item

```
<enumeration
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</enumeration>
```

enumeration Schema Component

Property	Representation
{value}	The appropriate **case** among the following: 1 **If** there is only one <enumeration> among the [children] of a <restriction>, **then** a set with one member, the actual value of its `value` [attribute], interpreted as an instance of the {base type definition}. 2 **otherwise** a set of the actual values of all the <enumeration> [children]'s `value` [attributes], interpreted as instances of the {base type definition}. **Note:** The `value` [attribute] is declared as having type ·`anySimpleType`·, but the {value} property of the enumeration facet MUST be a member of the {base type definition}. So in mapping from the XML representation to the enumeration component, the actual value is identified by using the ·lexical mapping· of the {base type definition}.
{annotations}	A (possibly empty) sequence of Annotation components, one for each <annotation> among the [children] of the <enumeration>s among the [children] of a <restriction>, in order.

4.3.5.3 Constraints on XML Representation of enumeration

Schema Representation Constraint: Enumeration value

The normalized value of the `value` [attribute] must be Datatype Valid (§4.1.4) with respect to the {base type definition} of the Simple Type Definition corresponding to the nearest <simpleType> ancestor element.

4.3.5.4 enumeration Validation Rules

Validation Rule: enumeration valid

A value in a ·value space· is facet-valid with respect to ·enumeration· if and only if the value is equal to one of the values specified in {value}.

Note: As specified normatively elsewhere, for purposes of checking enumerations, no distinction is made between an atomic value ***V*** and a list of length one containing ***V*** as its only item.

In this question, the behavior of this specification is thus the same as the behavior specified by [XQuery 1.0 and XPath 2.0 Functions and Operators] and related specifications.

4.3.5.5 Constraints on enumeration Schema Components

Schema Component Constraint: enumeration valid restriction

It is an ·error· if any member of {value} is not in the ·value space· of {base type definition}.

4.3.6 whiteSpace

[Definition:] **whiteSpace** constrains the ·value space· of types derived from string such that the various behaviors specified in Attribute Value Normalization in [XML] are realized. The value of **whiteSpace** must be one of {preserve, replace, collapse}.

preserve

No normalization is done, the value is not changed (this is the behavior required by [XML] for element content)

replace

All occurrences of #x9 (tab), #xA (line feed) and #xD (carriage return) are replaced with #x20 (space)

collapse

After the processing implied by **replace**, contiguous sequences of #x20's are collapsed to a single #x20, and any #x20 at the start or end of the string is then removed.

Note: The notation #xA used here (and elsewhere in this specification) represents the Universal Character Set (UCS) code point `hexadecimal A` (line feed), which is denoted by U+000A. This notation is to be distinguished from `
`, which is the XML character reference to that same UCS code point.

whiteSpace is applicable to all ·atomic· and ·list· datatypes. For all ·atomic· datatypes other than string (and types derived by ·facet-based restriction· from it) the value of **whiteSpace** is `collapse` and cannot be changed by a schema author; for string the value of **whiteSpace** is `preserve`; for any type derived by ·facet-based restriction· from string the value of **whiteSpace** can be any of the three legal values. For all datatypes ·constructed· by ·list· the value of

whiteSpace is `collapse` and cannot be changed by a schema author. For all datatypes ·constructed· by ·union· **whiteSpace** does not apply directly; however, the normalization behavior of ·union· types is controlled by the value of **whiteSpace** on that one of the ·basic members· against which the ·union· is successfully validated.

> **Note:** For more information on **whiteSpace**, see the discussion on white space normalization in [Schema Component Details] in [XSD 1.1 Part 1: Structures].

·whiteSpace· provides for:

- Constraining a ·value space· according to the white space normalization rules.

Example

The following example is the Simple Type Definition for the ·built-in· token datatype.

```
<simpleType name='token'>
    <restriction base='normalizedString'>
      <whiteSpace value='collapse'/>
    </restriction>
</simpleType>
```

> **Note:** The values "`replace`" and "`collapse`" may appear to provide a convenient way to "unwrap" text (i.e. undo the effects of pretty-printing and word-wrapping). In some cases, especially highly constrained data consisting of lists of artificial tokens such as part numbers or other identifiers, this appearance is correct. For natural-language data, however, the whitespace processing prescribed for these values is not only unreliable but will systematically remove the information needed to perform unwrapping correctly. For Asian scripts, for example, a correct unwrapping process will replace line boundaries not with blanks but with zero-width separators or nothing. In consequence, it is normally unwise to use these values for natural-language data, or for any data other than lists of highly constrained tokens.

4.3.6.1 The whiteSpace Schema Component

Schema Component: whiteSpace, a kind of Constraining Facet

{annotations}
: A sequence of Annotation components.

{value}
: One of {***preserve***, ***replace***, ***collapse***}. Required.

{fixed}
: An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} cannot specify a value for whiteSpace other than {value}.

4.3.6.2 XML Representation of whiteSpace Schema Components

The XML representation for a whiteSpace schema component is a <whiteSpace> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `whiteSpace` **Element Information Item**

```
<whiteSpace
  fixed = boolean : false
  id = ID
  value = (collapse | preserve | replace)
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</whiteSpace>
```

whiteSpace Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <whiteSpace> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.6.3 whiteSpace Validation Rules

> **Note:** There are no ·Validation Rule·s associated ·whiteSpace·. For more information, see the discussion on white space normalization in Schema Component Details in [XSD 1.1 Part 1: Structures].

4.3.6.4 Constraints on whiteSpace Schema Components

Schema Component Constraint: whiteSpace valid restriction

It is an ·error· if whiteSpace is among the members of {facets} of {base type definition} and any of the following conditions is true:

1 {value} is *replace* or *preserve* and the {value} of the parent whiteSpace is *collapse*
2 {value} is *preserve* and the {value} of the parent whiteSpace is *replace*

4.3.7 maxInclusive

[Definition:] maxInclusive is the inclusive upper bound of the ·value space· for a datatype with the ·ordered· property. The value of **maxInclusive** ·must· be equal to some value in the ·value space· of the ·base type·.

·maxInclusive· provides for:

- Constraining a ·value space· to values with a specific inclusive upper bound.

Example

The following is the definition of a ·user-defined· datatype which limits values to integers less than or equal to 100, using ·maxInclusive·.

```
<simpleType name='one-hundred-or-less'>
  <restriction base='integer'>
    <maxInclusive value='100'/>
  </restriction>
</simpleType>
```

4.3.7.1 The maxInclusive Schema Component

Schema Component: maxInclusive, a kind of Constraining Facet

{annotations}
A sequence of Annotation components.
{value}
Required.
A value from the ·value space· of the {base type definition}.
{fixed}
An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} cannot specify a value for maxInclusive other than {value}.

4.3.7.2 XML Representation of maxInclusive Schema Components

The XML representation for a maxInclusive schema component is a <maxInclusive> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `maxInclusive` Element Information Item

```
<maxInclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</maxInclusive>
```

{value} ·must· be equal to some value in the ·value space· of {base type definition}.

maxInclusive Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <maxInclusive> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.7.3 maxInclusive Validation Rules

Validation Rule: maxInclusive Valid

A value in an ·ordered· ·value space· is facet-valid with respect to ·maxInclusive· if and only if the value is less than or equal to {value}, according to the datatype's order relation.

4.3.7.4 Constraints on maxInclusive Schema Components

Schema Component Constraint: minInclusive <= maxInclusive

It is an ·error· for the value specified for ·minInclusive· to be greater than the value specified for ·maxInclusive· for the same datatype.

Schema Component Constraint: maxInclusive valid restriction

It is an ·error· if any of the following conditions is true:

1 maxInclusive is among the members of {facets} of {base type definition} and {value} is greater than the {value} of that maxInclusive.
2 maxExclusive is among the members of {facets} of {base type definition} and {value} is greater than or equal to the {value} of that maxExclusive.
3 minInclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of that minInclusive.
4 minExclusive is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of that minExclusive.

4.3.8 maxExclusive

[Definition:] **maxExclusive** is the exclusive upper bound of the ·value space· for a datatype with the ·ordered· property. The value of **maxExclusive** ·must· be equal to some value in the ·value space· of the ·base type· or be equal to {value} in {base type definition}.

·maxExclusive· provides for:

- Constraining a ·value space· to values with a specific exclusive upper bound.

Example

The following is the definition of a ·user-defined· datatype which limits values to integers less than or equal to 100, using ·maxExclusive·.

```
<simpleType name='less-than-one-hundred-and-one'>
  <restriction base='integer'>
    <maxExclusive value='101'/>
  </restriction>
</simpleType>
```

Note that the ·value space· of this datatype is identical to the previous one (named 'one-hundred-or-less').

4.3.8.1 The maxExclusive Schema Component

Schema Component: maxExclusive, a kind of Constraining Facet

{annotations}
A sequence of Annotation components.
{value}
Required.
A value from the ·value space· of the {base type definition}.
{fixed}
An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} cannot specify a value for maxExclusive other than {value}.

4.3.8.2 XML Representation of maxExclusive Schema Components

The XML representation for a maxExclusive schema component is a <maxExclusive> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: maxExclusive Element Information Item

```
<maxExclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</maxExclusive>
```

{value} ·must· be equal to some value in the ·value space· of {base type definition}.

maxExclusive Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <maxExclusive> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.8.3 maxExclusive Validation Rules

Validation Rule: maxExclusive Valid

A value in an ·ordered· ·value space· is facet-valid with respect to ·maxExclusive· if and only if the value is less than {value}, according to the datatype's order relation.

4.3.8.4 Constraints on maxExclusive Schema Components

Schema Component Constraint: maxInclusive and maxExclusive

It is an ·error· for both ·maxInclusive· and ·maxExclusive· to be specified in the same derivation step of a Simple Type Definition.

Schema Component Constraint: minExclusive <= maxExclusive

It is an ·error· for the value specified for ·minExclusive· to be greater than the value specified for ·maxExclusive· for the same datatype.

Schema Component Constraint: maxExclusive valid restriction

It is an ·error· if any of the following conditions is true:

1 maxExclusive is among the members of {facets} of {base type definition} and {value} is greater than the {value} of that maxExclusive.
2 maxInclusive is among the members of {facets} of {base type definition} and {value} is greater than the {value} of that maxInclusive.
3 minInclusive is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of that minInclusive.
4 minExclusive is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of that minExclusive.

4.3.9 minExclusive

[Definition:] **minExclusive** is the exclusive lower bound of the ·value space· for a datatype with the ·ordered· property. The value of **minExclusive** ·must· be equal to some value in the ·value space· of the ·base type· or be equal to {value} in {base type definition}.

·minExclusive· provides for:

- Constraining a ·value space· to values with a specific exclusive lower bound.

Example

The following is the definition of a ·user-defined· datatype which limits values to integers greater than or equal to 100, using ·minExclusive·.

```
<simpleType name='more-than-ninety-nine'>
  <restriction base='integer'>
    <minExclusive value='99'/>
  </restriction>
</simpleType>
```

Note that the ·value space· of this datatype is identical to the following one (named 'one-hundred-or-more').

4.3.9.1 The minExclusive Schema Component

Schema Component: minExclusive, a kind of Constraining Facet

`{annotations}`
: A sequence of Annotation components.

`{value}`
: Required.
: A value from the ·value space· of the {base type definition}.

`{fixed}`
: An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} cannot specify a value for minExclusive other than {value}.

4.3.9.2 XML Representation of minExclusive Schema Components

The XML representation for a minExclusive schema component is a <minExclusive> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `minExclusive` Element Information Item

```
<minExclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</minExclusive>
```

{value} ·must· be equal to some value in the ·value space· of {base type definition}.

minExclusive Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <minExclusive> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.9.3 minExclusive Validation Rules

Validation Rule: minExclusive Valid

A value in an ·ordered· ·value space· is facet-valid with respect to ·minExclusive· if and only if the value is greater than {value}, according to the datatype's order relation.

4.3.9.4 Constraints on minExclusive Schema Components

Schema Component Constraint: minInclusive and minExclusive

It is an ·error· for both ·minInclusive· and ·minExclusive· to be specified in the same derivation step of a Simple Type Definition.

Schema Component Constraint: minExclusive < maxInclusive

It is an ·error· for the value specified for ·minExclusive· to be greater than or equal to the value specified for ·maxInclusive· for the same datatype.

Schema Component Constraint: minExclusive valid restriction

It is an ·error· if any of the following conditions is true:

1 minExclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of that minExclusive.
2 minInclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of that minInclusive.
3 maxInclusive is among the members of {facets} of {base type definition} and {value} is greater than or equal to the {value} of that maxInclusive.
4 maxExclusive is among the members of {facets} of {base type definition} and {value} is greater than or equal to the {value} of that maxExclusive.

4.3.10 minInclusive

[Definition:] **minInclusive** is the inclusive lower bound of the ·value space· for a datatype with the ·ordered· property. The value of **minInclusive** ·must· be equal to some value in the ·value space· of the ·base type·.

·minInclusive· provides for:

- Constraining a ·value space· to values with a specific inclusive lower bound.

Example

The following is the definition of a ·user-defined· datatype which limits values to integers greater than or equal to 100, using ·minInclusive·.

```
<simpleType name='one-hundred-or-more'>
  <restriction base='integer'>
    <minInclusive value='100'/>
  </restriction>
</simpleType>
```

4.3.10.1 The minInclusive Schema Component

Schema Component: minInclusive, a kind of Constraining Facet

{annotations}
A sequence of Annotation components.
{value}
Required.
A value from the ·value space· of the {base type definition}.

{fixed}
An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} cannot specify a value for minInclusive other than {value}.

4.3.10.2 XML Representation of minInclusive Schema Components

The XML representation for a minInclusive schema component is a <minInclusive> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `minInclusive` Element Information Item

```
<minInclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</minInclusive>
```

{value} ·must· be equal to some value in the ·value space· of {base type definition}.

minInclusive Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <minInclusive> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.10.3 minInclusive Validation Rules

Validation Rule: minInclusive Valid

A value in an ·ordered· ·value space· is facet-valid with respect to ·minInclusive· if and only if the value is greater than or equal to {value}, according to the datatype's order relation.

4.3.10.4 Constraints on minInclusive Schema Components

Schema Component Constraint: minInclusive < maxExclusive

It is an ·error· for the value specified for ·minInclusive· to be greater than or equal to the value specified for ·maxExclusive· for the same datatype.

Schema Component Constraint: minInclusive valid restriction

It is an ·error· if any of the following conditions is true:

1 minInclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of that minInclusive.
2 maxInclusive is among the members of {facets} of {base type definition} and {value} is greater the {value} of that maxInclusive.
3 minExclusive is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of that minExclusive.
4 maxExclusive is among the members of {facets} of {base type definition} and {value} is greater than or equal to the {value} of that maxExclusive.

4.3.11 totalDigits

[Definition:] **totalDigits** restricts the magnitude and arithmetic precision of values in the ·value spaces· of precisionDecimal and decimal and datatypes derived from them. The effect must be described separately for the two primitive types.

For decimal, if the {value} of totalDigits is ***t***, the effect is to require that values be equal to $i / 10^{n}$, for some integers ***i*** and ***n***, with $| i | < 10^{t}$ and $0 \le n \le t$. This has as a consequence that the values are expressible using at most ***t*** digits in decimal notation.

For precisionDecimal, values with *·numericalValue·* of ***nV*** and *·arithmeticPrecision·* of ***aP***, if the {value} of totalDigits is ***t***, the effect is to require that (aP + 1 + $\log_{10}(| nV |)$ ·div· 1) ≤ ***t***, for values other than zero, NaN, and the infinities. This means in effect that values are expressible in scientific notation using at most ***t*** digits for the coefficient.

The {value} of totalDigits MUST be a positiveInteger.

The term 'totalDigits' is chosen to reflect the fact that it restricts the ·value space· to those values that can be represented lexically using at most *totalDigits* digits in decimal notation, or at most *totalDigits* digits for the coefficient, in scientific notation. Note that it does not restrict the ·lexical space· directly; a lexical representation that adds non-significant leading or trailing zero digits is still permitted. It also has no effect on the values NaN, INF, and -INF.

4.3.11.1 The totalDigits Schema Component

Schema Component: totalDigits, a kind of Constraining Facet

`{annotations}`
A sequence of Annotation components.
`{value}`
An xs:positiveInteger value. Required.
`{fixed}`
An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} MUST not specify a value for totalDigits other than {value}.

4.3.11.2 XML Representation of totalDigits Schema Components

The XML representation for a totalDigits schema component is a <totalDigits> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `totalDigits` Element Information Item

```
<totalDigits
  fixed = boolean : false
  id = ID
  value = positiveInteger
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</totalDigits>
```

totalDigits Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <totalDigits> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.11.3 totalDigits Validation Rules

Validation Rule: totalDigits Valid

A value ***v*** is facet-valid with respect to a totalDigits facet with a {value} of ***t*** if and only if one of the following is true:

1 ***v*** is a precisionDecimal value with *·numericalValue·* of ***positiveInfinity***, ***negativeInfinity***, ***notANumber***, or zero.
2 ***v*** is a precisionDecimal value with *·numericalValue·* of ***nV*** and *·arithmeticPrecision·* of ***aP***, and ***v*** is not NaN, INF, -INF, or zero, and (aP + 1 + $\log_{10}$(| nV |) ·div· 1) ≤ ***t***.
3 ***v*** is a decimal value equal to $i / 10^n$, for some integers ***i*** and ***n***, with | ***i*** | < 10^t and 0 ≤ ***n*** ≤ ***t***.

4.3.11.4 Constraints on totalDigits Schema Components

Schema Component Constraint: totalDigits valid restriction

It is an ·error· if the ·owner·'s {base type definition} has a totalDigits facet among its {facets} and {value} is greater than the {value} of that totalDigits facet.

4.3.12 fractionDigits

[Definition:] **fractionDigits** places an upper limit on the arithmetic precision of decimal values: if the {value} of **fractionDigits** = ***f***, then the value space is restricted to values equal to $i / 10^n$ for some integers ***i*** and ***n*** and 0 ≤ ***n*** ≤ ***f***. The value of **fractionDigits** ·must· be a nonNegativeInteger

The term **fractionDigits** is chosen to reflect the fact that it restricts the ·value space· to those values that can be represented lexically in decimal notation using at most *fractionDigits* to the right of the decimal point. Note that it does not restrict the ·lexical space· directly; a lexical representation that adds non-significant leading or trailing zero digits is still permitted.

Example

The following is the definition of a ·user-defined· datatype which could be used to represent the magnitude of a person's body temperature on the Celsius scale. This definition would appear in a schema authored by an "end-user" and shows how to define a datatype by specifying facet values which constrain the range of the ·base type·.

```
<simpleType name='celsiusBodyTemp'>
  <restriction base='decimal'>
    <fractionDigits value='1'/>
    <minInclusive value='32'/>
         <maxInclusive value='41.7'/>
  </restriction>
</simpleType>
```

4.3.12.1 The fractionDigits Schema Component

Schema Component: fractionDigits, a kind of Constraining Facet

{annotations}
A sequence of Annotation components.
{value}
An xs:nonNegativeInteger value. Required.
{fixed}
An xs:boolean value. Required.

If {fixed} is *true*, then types for which the current type is the {base type definition} MUST not specify a value for fractionDigits other than {value}.

4.3.12.2 XML Representation of fractionDigits Schema Components

The XML representation for a fractionDigits schema component is a <fractionDigits> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `fractionDigits` Element Information Item

```
<fractionDigits
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</fractionDigits>
```

fractionDigits Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <fractionDigits> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.12.3 fractionDigits Validation Rules

Validation Rule: fractionDigits Valid

A value is facet-valid with respect to ·fractionDigits· if and only if that value is equal to $i / 10^n$ for integer i and n, with $0 \le n \le$ {value}.

4.3.12.4 Constraints on fractionDigits Schema Components

Schema Component Constraint: fractionDigits less than or equal to totalDigits

It is an ·error· for the {value} of fractionDigits to be greater than that of totalDigits.

Schema Component Constraint: fractionDigits valid restriction

It is an ·error· if ·fractionDigits· is among the members of {facets} of {base type definition} and {value} is greater than the {value} of that ·fractionDigits·.

4.3.13 maxScale

[Definition:] **maxScale** places an upper limit on the *·arithmeticPrecision·* of precisionDecimal values: if the {value} of **maxScale** = m, then only values with *·arithmeticPrecision·* $\le m$ are retained in the ·value space·. As a consequence, every value in the value space will have *·numericalValue·* equal to $i / 10^n$ for some integers i and n, with $n \le m$. The {value} of maxScale must be an integer. If it is negative, the numeric values of the datatype are restricted to multiples of 10 (or 100, or …).

The term 'maxScale' is chosen to reflect the fact that it restricts the ·value space· to those values that can be represented lexically in scientific notation using an integer coefficient and a scale (or negative exponent) no greater than maxScale. (It has nothing to do with the use of the term 'scale' to denote the radix or base of a notation.) Note that maxScale does not restrict the ·lexical space· directly; a lexical representation that adds non-significant leading or trailing zero digits, or that uses a lower exponent with a non-integer coefficient is still permitted.

Example

The following is the definition of a user-defined datatype which could be used to represent a floating-point decimal datatype which allows seven decimal digits for the coefficient and exponents between −95 and 96. Note that the scale is −1 times the exponent.

```
<simpleType name='decimal32'>
  <restriction base='precisionDecimal'>
    <totalDigits value='7'/>
    <maxScale value='95'/>
    <minScale value='-96'/>
  </restriction>
</simpleType>
```

4.3.13.1 The maxScale Schema Component

Schema Component: maxScale, a kind of Constraining Facet

{annotations}
A sequence of Annotation components.
{value}
An xs:integer value. Required.
{fixed}
An xs:boolean value. Required.

If {fixed} is ***true***, then types for which the current type is the {base type definition} MUST not specify a value for maxScale other than {value}.

4.3.13.2 XML Representation of maxScale Schema Components

The XML representation for a maxScale schema component is a <maxScale> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `maxScale` Element Information Item

```
<maxScale
  fixed = boolean : false
  id = ID
  value = integer
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</maxScale>
```

maxScale Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <maxScale> element, as defined in section XML Representation of Annotation Schema Components of

[XSD 1.1 Part 1: Structures].

4.3.13.3 maxScale Validation Rules

Validation Rule: maxScale Valid

A precisionDecimal value ***v*** is facet-valid with respect to maxScale if and only if one of the following is true:

1 ***v*** has *·arithmeticPrecision·* less than or equal to the {value} of maxScale.
2 The *·arithmeticPrecision·* of ***v*** is ***absent***.

4.3.13.4 Constraints on maxScale Schema Components

Schema Component Constraint: maxScale valid restriction

It is an ·error· if maxScale is among the members of {facets} of {base type definition} and {value} is greater than the {value} of that maxScale.

4.3.14 minScale

[Definition:] **minScale** places a lower limit on the *·arithmeticPrecision·* of precisionDecimal values. If the {value} of **minScale** is ***m***, then the value space is restricted to values with *·arithmeticPrecision·* ≥ ***m***. As a consequence, every value in the value space will have *·numericalValue·* equal to i / 10^n for some integers ***i*** and ***n***, with ***n*** ≥ ***m***.

The term **minScale** is chosen to reflect the fact that it restricts the ·value space· to those values that can be represented lexically in exponential form using an integer coefficient and a scale (negative exponent) at least as large as *minScale*. Note that it does not restrict the ·lexical space· directly; a lexical representation that adds additional leading zero digits, or that uses a larger exponent (and a correspondingly smaller coefficient) is still permitted.

Example

The following is the definition of a user-defined datatype which could be used to represent amounts in a decimal currency; it corresponds to a SQL column definition of `DECIMAL(8,2)`. The effect is to allow values between -999,999.99 and 999,999.99, with a fixed interval of 0.01 between values.

```
<simpleType name='price'>
  <restriction base='precisionDecimal'>
    <totalDigits value='8'/>
    <minScale value='2'/>
    <maxScale value='2'/>
  </restriction>
</simpleType>
```

4.3.14.1 The minScale Schema Component

Schema Component: minScale, a kind of Constraining Facet

`{annotations}`
: A sequence of Annotation components.

`{value}`
: An xs:integer value. Required.

`{fixed}`
: An xs:boolean value. Required.

If {fixed} is ***true***, then types for which the current type is the {base type definition} MUST not specify a value for minScale other than {value}.

4.3.14.2 XML Representation of minScale Schema Components

The XML representation for a minScale schema component is a <minScale> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `minScale` Element Information Item

```
<minScale
  fixed = boolean : false
  id = ID
  value = integer
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</minScale>
```

minScale Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <minScale> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.14.3 minScale Validation Rules

Validation Rule: minScale Valid

A precisionDecimal value ***v*** is facet-valid with respect to minScale if and only if one of the following is true:

1 ***v*** has *·arithmeticPrecision·* greater than or equal to the {value} of minScale.
2 The *·arithmeticPrecision·* of ***v*** is ***absent***.

4.3.14.4 Constraints on minScale Schema Components

Schema Component Constraint: minScale less than or equal to maxScale

It is an ·error· for minScale to be greater than maxScale.

Note that it is *not* an error for minScale to be greater than totalDigits.

Schema Component Constraint: minScale valid restriction

It is an ·error· if minScale is among the members of {facets} of {base type definition} and {value} is less than the {value} of that minScale.

4.3.15 Assertions

[Definition:] **Assertions** constrain the ·value space· by requiring the values to satisfy specified XPath ([XPath 2.0]) expressions. The value of the assertions facet is a sequence of Assertion components as defined in [XSD 1.1 Part 1: Structures].

The following is the definition of a ·user-defined· datatype which allows all integers but 0 by using an assertion to disallow the value 0.

```
<simpleType name='nonZeroInteger'>
  <restriction base='integer'>
    <assertion test='$value ne 0'/>
  </restriction>
</simpleType>
```

The following example defines the datatype "triple", whose ·value space· is the set of integers evenly divisible by three.

```
<simpleType name='triple'>
  <restriction base='integer'>
    <assertion test='$value mod 3 eq 0'/>
  </restriction>
</simpleType>
```

The same datatype can be defined without the use of assertions, but the pattern necessary to represent the set of triples is long and error-prone:

```
<simpleType name='triple'>
  <restriction base='integer'>
    <pattern value=
    "([0369]|[147][0369]*[258]|(([258]|[147][0369]*[147])([0369]|[258][0369]*[147]))
  </restriction>
</simpleType>
```

The assertion used in the first version of "triple" is likely to be clearer for many readers of the schema document.

4.3.15.1 The assertions Schema Component

Schema Component: assertions, a kind of Constraining Facet

{annotations}
: A sequence of Annotation components.

{value}
: A sequence of Assertion components.

4.3.15.2 XML Representation of assertions Schema Components

The XML representation for an assertions schema component is one or more <assertion> element information items. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `assertion` Element Information Item

```
<assertion
  id = ID
  test = an XPath expression
  xpathDefaultNamespace = (anyURI | (##defaultNamespace | ##targetNamespace |
##local))
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</assertion>
```

assertions Schema Component

Property	Representation
{value}	A sequence whose members are Assertions drawn from the following sources, in order: 1 If the {base type definition} of the ·owner· has an assertions facet among its {facets}, then the Assertions which appear in the {value} of that assertions facet. 2 Assertions corresponding to the <assertion> element information items among the [children] of <restriction>, if any, in document order. For details of the construction of the Assertion components, see section 3.13.2 of [XSD 1.1 Part 1: Structures].
{annotations}	The empty sequence. **Note:** Annotations specified within an <assertion> element are captured by the individual Assertion component to which it maps.

4.3.15.3 Assertions Validation Rules

The following rule refers to "the nearest built-in" datatype and to the "XDM representation" of a value under a datatype. [Definition:] For any datatype ***T***, the **nearest built-in datatype** to ***T*** is the first ·built-in· datatype encountered in following the chain of links connecting each datatype to its ·base type·. If ***T*** is a ·built-in· datatype, then the nearest built-in datatype of ***T*** is ***T*** itself; otherwise, it is the nearest built-in datatype of ***T***'s ·base type·.

[Definition:] For any value ***V*** and any datatype ***T***, the **XDM representation of *V* under *T*** is defined recursively as follows. Call the XDM representation ***X***. Then

1 If ***T*** = ·`xs:anySimpleType`· or ·`xs:anyAtomicType`· then ***X*** is ***V***, and the dynamic type of ***X*** is

xs:untypedAtomic.

2 If ***T*** . {variety} = ***atomic***, then let ***T2*** be the ·nearest built-in datatype· to ***T***. If ***V*** is a member of the ·value space· of ***T2***, then ***X*** is ***V*** and the dynamic type of ***X*** is ***T2***. Otherwise (i.e. if ***V*** is not a member of the ·value space· of ***T2***), ***X*** is the ·XDM representation· of ***V*** under ***T2*** . {base type definition}.

3 If ***T*** . {variety} = ***list***, then ***X*** is a sequence of atomic values, each atomic value being the ·XDM representation· of the corresponding item in the list ***V*** under ***T*** . {item type definition}.

4 If ***T*** . {variety} = ***union***, then ***X*** is the ·XDM representation· of ***V*** under the ·active basic member· of ***V*** when validated against ***T***. If there is no ·active basic member·, then ***V*** has no ·XDM representation· under ***T***.

Note: If the {item type definition} of a ·list· is a ·union·, or the ·active basic member· is a ·list·, then several steps may be necessary before the ·atomic· datatype which serves as the dynamic type of ***X*** is found.

Because the {item type definition} of a ·list· is required to be an ·atomic· or ·union· datatype, and the ·active basic member· of a ·union· which accepts the value ***V*** is by definition not a ·union·, the recursive rule given above is guaranteed to terminate in a sequence of one or more ·atomic· values, each belonging to an ·atomic· datatype.

Validation Rule: Assertions Valid

A value ***V*** is facet-valid with respect to an assertions facet belonging to a simple type ***T*** if and only if the {test} property of each Assertion in its {value} evaluates to `true` under the conditions laid out below, without raising any dynamic error or type error.

Evaluation of {test} is performed as defined in [XPath 2.0], with the following conditions:

1 The XPath expression {test} is evaluated, following the rules given in XPath Evaluation of [XSD 1.1 Part 1: Structures], with the following modifications.

1.1 The in-scope variables in the static context is a set with a single member. The `expanded QName` of that member has no namespace URI and has '`value`' as the local name. The (static) `type` of the member is `anyAtomicType*`.

Note: The XDM type label `anyAtomicType*` simply says that for static typing purposes the variable `$value` will have a value consisting of a sequence of zero or more atomic values.

1.2 There is no context item for the evaluation of the XPath expression.

Note: In the terminology of [XPath 2.0], the context item is "undefined".

Note: As a consequence the expression '.', or any implicit or explicit reference to the context item, will raise a dynamic error, which will cause the assertion to be treated as false. If an error is detected statically, then the assertion violates the schema component constraint XPath Valid and causes an error to be flagged in the schema.

The variable "`$value`" can be used to refer to the value being checked.

1.3 There is likewise no value for the context size and the context position in the dynamic context used for evaluation of the assertion.

1.4 The variable values in the dynamic context is a set with a single member. The expanded QName of that member has no namespace URI and 'value' as the local name. The value of the member is the ·XDM representation· of ***V*** under ***T***.

1.5 If ***V*** has no ·XDM representation· under ***T***, then the XPath expression cannot usefully be evaluated, and ***V*** is not facet-valid against the assertions facet of ***T***.

2 The evaluation result is converted to either true or false as if by a call to the XPath fn:boolean function.

4.3.15.4 Constraints on assertions Schema Components

Schema Component Constraint: Valid restriction of assertions

The {value} of the assertions facet on the {base type definition} MUST be a prefix of the {value}.

> **Note:** For components constructed from XML representations in schema documents, the satisfaction of this constraint is a consequence of the XML mapping rules: any assertion imposed by a simple type definition ***S*** will always also be imposed by any type derived from ***S*** by ·facet-based restriction·. This constraint ensures that components constructed by other means (so-called "born-binary" components) similarly preserve assertions facets across ·facet-based restriction·.

4.3.16 explicitTimezone

[Definition:] **explicitTimezone** is a three-valued facet which can can be used to require or prohibit the time zone offset in date/time datatypes.

Example

The following ·user-defined· datatype accepts only date values without a time zone offset, using the explicitTimezone facet.

```
<simpleType name='bare-date'>
  <restriction base='date'>
    <explicitTimezone value='prohibited'/>
  </restriction>
</simpleType>
```

The same effect could also be achieved using the pattern facet, as shown below, but it is somewhat less clear what is going on in this derivation, and it is better practice to use the more straightforward explicitTimezone for this purpose.

```
<simpleType name='bare-date'>
  <restriction base='date'>
    <pattern value='[^:Z]*'/>
  </restriction>
</simpleType>
```

4.3.16.1 The explicitTimezone Schema Component

Schema Component: explicitTimezone, a kind of Constraining Facet

`{annotations}`
A sequence of Annotation components.
`{value}`
One of {***required***, ***prohibited***, ***optional***}. Required.
`{fixed}`
An xs:boolean value. Required.

If {fixed} is *true*, then datatypes for which the current type is the {base type definition} cannot specify a value for explicitTimezone other than {value}.

> **Note:** It is a consequence of timezone valid restriction (§4.3.16.4) that the value of the explicitTimezone facet cannot be changed unless that value is ***optional***, regardless of whether {fixed} is ***true*** or ***false***. Accordingly, {fixed} is relevant only when {value} is ***optional***.

4.3.16.2 XML Representation of explicitTimezone Schema Components

The XML representation for an explicitTimezone schema component is an <explicitTimezone> element information item. The correspondences between the properties of the information item and properties of the component are as follows:

XML Representation Summary: `explicitTimezone` Element Information Item

```
<explicitTimezone
  fixed = boolean : false
  id = ID
  value = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</explicitTimezone>
```

explicitTimezone Schema Component

Property	Representation
{value}	The actual value of the `value` [attribute]
{fixed}	The actual value of the `fixed` [attribute], if present, otherwise ***false***
{annotations}	The annotation mapping of the <explicitTimezone> element, as defined in section XML Representation of Annotation Schema Components of [XSD 1.1 Part 1: Structures].

4.3.16.3 explicitTimezone Validation Rules

Validation Rule: explicitOffset Valid

A dateTime value ***V*** is facet-valid with respect to ·explicitTimezone· if and only if **one** of the following is true

1 The {value} of the facet is ***required*** and ***V*** has a (non-***absent***) value for the *·timezoneOffset·* property.

2 The {value} of the facet is ***prohibited*** and the value for the [·*timezoneOffset*·] property in ***V*** is ***absent***.
3 The {value} of the facet is ***optional***.

4.3.16.4 Constraints on explicitTimezone Schema Components

Schema Component Constraint: timezone valid restriction

If the explicitTimezone facet on the {base type definition} has a {value} other than ***optional***, then the {value} of the facet on the ·restriction· MUST be equal to the {value} on the {base type definition}; otherwise it is an ·error·.

> **Note:** The effect of this rule is to allow datatypes with a explicitTimezone value of ***optional*** to be restricted by specifying a value of ***required*** or ***prohibited***, and to forbid any other derivations using this facet.

5 Conformance

XSD 1.1: Datatypes is intended to be usable in a variety of contexts.

In the usual case, it will embedded in a **host language** such as [XSD 1.1 Part 1: Structures], which refers to this specification normatively to define some part of the host language. In some cases, *XSD 1.1: Datatypes* may be implemented independently of any host language.

Certain aspects of the behavior of conforming processors are described in this specification as ·implementation-defined· or ·implementation-dependent·.

- [Definition:] Something which MAY vary among conforming implementations, but which MUST be specified by the implementor for each particular implementation, is **implementation-defined**.
- [Definition:] Something which MAY vary among conforming implementations, is not specified by this or any W3C specification, and is not required to be specified by the implementor for any particular implementation, is **implementation-dependent**.

Anything described in this specification as ·implementation-defined· or ·implementation-dependent· MAY be further constrained by the specifications of a host language in which the datatypes and other material specified here are used. A list of implementation-defined and implementation-dependent features can be found in Implementation-defined and implementation-dependent features (normative) (§H)

5.1 Host Languages

When *XSD 1.1: Datatypes* is embedded in a host language, the definition of conformance is specified by the host language, not by this specification. That is, when this specification is implemented in the context of an implementation of a host language, the question of conformance to this specification (separate from the host language) does not arise.

This specification imposes certain constraints on the embedding of *XSD 1.1: Datatypes* by a host language; these are indicated in the normative text by the use of the verbs 'MUST', etc., with the phrase "host language" as the subject of the verb.

Note: For convenience, the most important of these constraints are noted here:

- Host languages SHOULD specify that all of the datatypes decribed here as built-ins are automatically available.
- Host languages MAY specify that additional datatypes are also made available automatically.
- If user-defined datatypes are to be supported in the host language, then the host language MUST specify how user-defined datatypes are defined and made available for use.

In addition, host languages MUST require conforming implementations of the host language to obey all of the constraints and rules specified here.

5.2 Independent implementations

[Definition:] Implementations claiming **minimal conformance** to this specification independent of any host language MUST do **all** of the following:

1 Support all the ·built-in· datatypes defined in this specification.

2 Completely and correctly implement all of the ·constraints on schemas· defined in this specification.

3 Completely and correctly implement all of the ·Validation Rules· defined in this specification, when checking the datatype validity of literals against datatypes.

Implementations claiming **schema-document-aware conformance** to this specification, independent of any host language MUST be minimally conforming. In addition, they must do **all** of the following:

1 Accept simple type definitions in the form specified in Datatype components (§4).

2 Completely and correctly implement all of rules governing the XML representation of simple type definitions specified in Datatype components (§4).

3 Map the XML representations of simple type definitions to simple type definition components as specified in the mapping rules given in Datatype components (§4).

Note: The term **schema-document aware** is used here for parallelism with the corresponding term in [XSD 1.1 Part 1: Structures]. The reference to schema documents may be taken as referring to the fact that schema-document-aware implementations accept the XML representation of simple type definitions found in XSD schema documents. It does *not* mean that the simple type definitions must themselves be free-standing XML documents, nor that they typically will be.

5.3 Conformance of data

Abstract representations of simple type definitions conform to this specification if and only if they obey all of the ·constraints on schemas· defined in this specification.

XML representations of simple type definitions conform to this specification if they obey all of the applicable rules defined in this specification.

Note: Because the conformance of the resulting simple type definition component depends not only on the XML representation of a given simple type definition, but on the properties of its ·base type·, the conformance of an XML representation of a simple type definition does not guarantee that, in the context of other schema components, it will map to a conforming component.

5.4 Partial Implementation of Infinite Datatypes

Some ·primitive· datatypes defined in this specification have infinite ·value spaces·; no finite implementation can completely handle all their possible values. For some such datatypes, minimum implementation limits are specified below. For other infinite types such as string, hexBinary, and base64Binary, no minimum implementation limits are specified.

When this specification is used in the context of other languages (as it is, for example, by [XSD 1.1 Part 1: Structures]), the host language may specify other minimum implementation limits.

When presented with a literal or value exceeding the capacity of its partial implementation of a datatype, a minimally conforming implementation of this specification will sometimes be unable to determine with certainty whether the value is datatype-valid or not. Sometimes it will be unable to represent the value correctly through its interface to any downsteam application.

When either of these is so, a conforming processor MUST indicate to the user and/or downstream application that it cannot process the input data with assured correctness (much as it would indicate if it ran out of memory). When the datatype validity of a value or literal is uncertain because it exceeds the capacity of a partial implementation, the literal or value MUST NOT be treated as invalid, and the unsupported value MUST NOT be quietly changed to a supported value.

This specification does not constrain the method used to indicate that a literal or value in the input data has exceeded the capacity of the implementation, or the form such indications take.

·Minimally conforming· processors which set an application- or ·implementation-defined· limit on the size of the values supported MUST clearly document that limit.

These are the partial-implementation ·minimal conformance· requirements:

- All ·minimally conforming· processors MUST support decimal values whose absolute value can be expressed as $i / 10^k$, where i and k are nonnegative integers such that $i < 10^{16}$ and $k \le 16$ (i.e., those expressible with sixteen total digits).
- All ·minimally conforming· processors MUST support nonnegative *·year·* values less than 10000 (i.e., those expressible with four digits) in all datatypes which use the seven-property model defined in The Seven-property Model (§D.2.1) and have a non-·absent· value for *·year·* (i.e. dateTime, dateTimeStamp, date, gYearMonth, and gYear). .
- All ·minimally conforming· processors MUST support *·second·* values to milliseconds (i.e. those expressible with three fraction digits) in all datatypes which use the seven-property model defined in The Seven-property Model (§D.2.1) and have a non-·absent· value for *·second·* (i.e. dateTime, dateTimeStamp, and time). .
- All ·minimally conforming· processors MUST support fractional-second duration values to milliseconds (i.e. those expressible with three fraction digits).

- All ·minimally conforming· processors MUST support duration values with ·*months*· values in the range -119999 to 119999 months (9999 years and 11 months) and ·*seconds*· values in the range -31622400 to 31622400 seconds (one leap-year).

- All ·minimally conforming· processors MUST support all precisionDecimal values in the ·value space· of the otherwise unconstrained derived datatype for which totalDigits is set to sixteen, maxScale to 369, and minScale to −398.

  Note: The conformance limits given in the text correspond to those of the decimal64 type defined in [IEEE 754-2008], which can be stored in a 64-bit field. The XML Schema Working Group recommends that implementors support limits corresponding to those of the decimal128 type. This entails supporting the values in the value space of the otherwise unconstrained datatype for which totalDigits is set to 34, maxScale to 6176, and minScale to −6111.

A Schema for Schema Documents (Datatypes) (normative)

The XML representation of the datatypes-relevant part of the schema for schema documents is presented here as a normative part of the specification.

Like any other XML document, schema documents may carry XML and document type declarations. An XML declaration and a document type declaration are provided here for convenience. Since this schema document describes the XML Schema language, the `targetNamespace` attribute on the `schema` element refers to the XML Schema namespace itself.

Schema documents conforming to this specification may be in XML 1.0 or XML 1.1. Conforming implementations may accept input in XML 1.0 or XML 1.1 or both. See Dependencies on Other Specifications (§1.3).

Schema for Schema Documents (Datatypes)

```
<?xml version='1.0'?>
<!DOCTYPE xs:schema PUBLIC "-//W3C//DTD XMLSCHEMA 200102//EN" "XMLSchema.dtd" [

<!--
        Make sure that processors that do not read the external
        subset will know about the various IDs we declare
  -->
        <!ATTLIST xs:simpleType id ID #IMPLIED>
        <!ATTLIST xs:maxExclusive id ID #IMPLIED>
        <!ATTLIST xs:minExclusive id ID #IMPLIED>
        <!ATTLIST xs:maxInclusive id ID #IMPLIED>
        <!ATTLIST xs:minInclusive id ID #IMPLIED>
        <!ATTLIST xs:totalDigits id ID #IMPLIED>
        <!ATTLIST xs:fractionDigits id ID #IMPLIED>
        <!ATTLIST xs:maxScale id ID #IMPLIED>
        <!ATTLIST xs:minScale id ID #IMPLIED>
        <!ATTLIST xs:length id ID #IMPLIED>
        <!ATTLIST xs:minLength id ID #IMPLIED>
        <!ATTLIST xs:maxLength id ID #IMPLIED>
        <!ATTLIST xs:enumeration id ID #IMPLIED>
        <!ATTLIST xs:pattern id ID #IMPLIED>
        <!ATTLIST xs:assertion id ID #IMPLIED>
        <!ATTLIST xs:explicitTimezone id ID #IMPLIED>
        <!ATTLIST xs:appinfo id ID #IMPLIED>
        <!ATTLIST xs:documentation id ID #IMPLIED>
        <!ATTLIST xs:list id ID #IMPLIED>
        <!ATTLIST xs:union id ID #IMPLIED>
```

```
        ]>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           xml:lang="en"
           targetNamespace="http://www.w3.org/2001/XMLSchema"
           version="datatypes.xsd (cr-20090430)">
  <xs:annotation>
    <xs:documentation source="../datatypes/datatypes.html">
      The schema corresponding to this document is normative,
      with respect to the syntactic constraints it expresses in the
      XML Schema language.  The documentation (within <documentation>
      elements) below, is not normative, but rather highlights important
      aspects of the W3C Recommendation of which this is a part.
    </xs:documentation>
  </xs:annotation>


  <xs:simpleType name="derivationControl">
    <xs:annotation>
      <xs:documentation>
   A utility type, not for public use</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="substitution"/>
      <xs:enumeration value="extension"/>
      <xs:enumeration value="restriction"/>
      <xs:enumeration value="list"/>
      <xs:enumeration value="union"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:group name="simpleDerivation">
    <xs:choice>
      <xs:element ref="xs:restriction"/>
      <xs:element ref="xs:list"/>
      <xs:element ref="xs:union"/>
    </xs:choice>
  </xs:group>
  <xs:simpleType name="simpleDerivationSet">
    <xs:annotation>
      <xs:documentation>
   #all or (possibly empty) subset of {restriction, extension, union, list}
   </xs:documentation>
      <xs:documentation>
   A utility type, not for public use</xs:documentation>
    </xs:annotation>
    <xs:union>
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="#all"/>
        </xs:restriction>
      </xs:simpleType>
      <xs:simpleType>
        <xs:list>
          <xs:simpleType>
            <xs:restriction base="xs:derivationControl">
              <xs:enumeration value="list"/>
              <xs:enumeration value="union"/>
              <xs:enumeration value="restriction"/>
              <xs:enumeration value="extension"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:list>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
  <xs:complexType name="simpleType" abstract="true">
    <xs:complexContent>
      <xs:extension base="xs:annotated">
```

```
        <xs:group ref="xs:simpleDerivation"/>
        <xs:attribute name="final" type="xs:simpleDerivationSet"/>
        <xs:attribute name="name" type="xs:NCName">
          <xs:annotation>
            <xs:documentation>
              Can be restricted to required or forbidden
            </xs:documentation>
          </xs:annotation>
        </xs:attribute>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="topLevelSimpleType">
    <xs:complexContent>
      <xs:restriction base="xs:simpleType">
        <xs:sequence>
          <xs:element ref="xs:annotation" minOccurs="0"/>
          <xs:group ref="xs:simpleDerivation"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:NCName" use="required">
          <xs:annotation>
            <xs:documentation>
              Required at the top level
            </xs:documentation>
          </xs:annotation>
        </xs:attribute>
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="localSimpleType">
    <xs:complexContent>
      <xs:restriction base="xs:simpleType">
        <xs:sequence>
          <xs:element ref="xs:annotation" minOccurs="0"/>
          <xs:group ref="xs:simpleDerivation"/>
        </xs:sequence>
        <xs:attribute name="name" use="prohibited">
          <xs:annotation>
            <xs:documentation>
              Forbidden when nested
            </xs:documentation>
          </xs:annotation>
        </xs:attribute>
        <xs:attribute name="final" use="prohibited"/>
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="simpleType" type="xs:topLevelSimpleType" id="simpleType">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-simpleType"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="facet" abstract="true">
    <xs:annotation>
      <xs:documentation>
        An abstract element, representing facets in general.
        The facets defined by this spec are substitutable for
        this element, and implementation-defined facets should
        also name this as a substitution-group head.
      </xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:group name="simpleRestrictionModel">
    <xs:sequence>
      <xs:element name="simpleType" type="xs:localSimpleType" minOccurs="0"/>
      <xs:choice minOccurs="0"
```

```
          maxOccurs="unbounded">
        <xs:element ref="xs:facet"/>
        <xs:any processContents="lax"
            namespace="##other"/>
      </xs:choice>
    </xs:sequence>
  </xs:group>
  <xs:element name="restriction" id="restriction">
    <xs:complexType>
      <xs:annotation>
        <xs:documentation
             source="http://www.w3.org/TR/xmlschema11-2/#element-restriction">
          base attribute and simpleType child are mutually
          exclusive, but one or other is required
        </xs:documentation>
      </xs:annotation>
      <xs:complexContent>
        <xs:extension base="xs:annotated">
          <xs:group ref="xs:simpleRestrictionModel"/>
          <xs:attribute name="base" type="xs:QName" use="optional"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="list" id="list">
    <xs:complexType>
      <xs:annotation>
        <xs:documentation
             source="http://www.w3.org/TR/xmlschema11-2/#element-list">
          itemType attribute and simpleType child are mutually
          exclusive, but one or other is required
        </xs:documentation>
      </xs:annotation>
      <xs:complexContent>
        <xs:extension base="xs:annotated">
          <xs:sequence>
            <xs:element name="simpleType" type="xs:localSimpleType"
                        minOccurs="0"/>
          </xs:sequence>
          <xs:attribute name="itemType" type="xs:QName" use="optional"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="union" id="union">
    <xs:complexType>
      <xs:annotation>
        <xs:documentation
             source="http://www.w3.org/TR/xmlschema11-2/#element-union">
          memberTypes attribute must be non-empty or there must be
          at least one simpleType child
        </xs:documentation>
      </xs:annotation>
      <xs:complexContent>
        <xs:extension base="xs:annotated">
          <xs:sequence>
            <xs:element name="simpleType" type="xs:localSimpleType"
                        minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="memberTypes" use="optional">
            <xs:simpleType>
              <xs:list itemType="xs:QName"/>
            </xs:simpleType>
          </xs:attribute>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="facet">
```

```
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:attribute name="value" use="required"/>
        <xs:attribute name="fixed" type="xs:boolean" default="false"
                      use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="noFixedFacet">
    <xs:complexContent>
      <xs:restriction base="xs:facet">
        <xs:sequence>
          <xs:element ref="xs:annotation" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="fixed" use="prohibited"/>
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="minExclusive" type="xs:facet"
    id="minExclusive"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-minExclusive"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="minInclusive" type="xs:facet"
    id="minInclusive"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-minInclusive"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="maxExclusive" type="xs:facet"
    id="maxExclusive"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-maxExclusive"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="maxInclusive" type="xs:facet"
    id="maxInclusive"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-maxInclusive"/>
    </xs:annotation>
  </xs:element>
  <xs:complexType name="numFacet">
    <xs:complexContent>
      <xs:restriction base="xs:facet">
        <xs:sequence>
          <xs:element ref="xs:annotation" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="value"
            type="xs:nonNegativeInteger" use="required"/>
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="intFacet">
    <xs:complexContent>
      <xs:restriction base="xs:facet">
        <xs:sequence>
          <xs:element ref="xs:annotation" minOccurs="0"/>
```

```
        </xs:sequence>
        <xs:attribute name="value" type="xs:integer" use="required"/>
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="totalDigits" id="totalDigits"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-totalDigits"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:restriction base="xs:numFacet">
          <xs:sequence>
            <xs:element ref="xs:annotation" minOccurs="0"/>
          </xs:sequence>
          <xs:attribute name="value" type="xs:positiveInteger" use="required"/>
          <xs:anyAttribute namespace="##other" processContents="lax"/>
        </xs:restriction>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="fractionDigits" type="xs:numFacet"
    id="fractionDigits"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-fractionDigits"/>
    </xs:annotation>
  </xs:element>

  <xs:element name="maxScale" type="xs:intFacet"
    id="maxScale"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-maxScale"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="minScale" type="xs:intFacet"
    id="minScale"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-minScale"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="length" type="xs:numFacet" id="length"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-length"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="minLength" type="xs:numFacet"
    id="minLength"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-minLength"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="maxLength" type="xs:numFacet"
    id="maxLength"
    substitutionGroup="xs:facet">
    <xs:annotation>
```

```
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-maxLength"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="enumeration" type="xs:noFixedFacet"
    id="enumeration"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-enumeration"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="whiteSpace" id="whiteSpace"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-whiteSpace"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:restriction base="xs:facet">
          <xs:sequence>
            <xs:element ref="xs:annotation" minOccurs="0"/>
          </xs:sequence>
          <xs:attribute name="value" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:NMTOKEN">
                <xs:enumeration value="preserve"/>
                <xs:enumeration value="replace"/>
                <xs:enumeration value="collapse"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
          <xs:anyAttribute namespace="##other" processContents="lax"/>
        </xs:restriction>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="pattern" id="pattern"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-pattern"/>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:restriction base="xs:noFixedFacet">
          <xs:sequence>
            <xs:element ref="xs:annotation" minOccurs="0"/>
          </xs:sequence>
          <xs:attribute name="value" type="xs:string"
              use="required"/>
          <xs:anyAttribute namespace="##other"
              processContents="lax"/>
        </xs:restriction>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="assertion" type="xs:assertion"
              id="assertion" substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
           source="http://www.w3.org/TR/xmlschema11-2/#element-assertion"/>
    </xs:annotation>
  </xs:element>
  <xs:element name="explicitTimezone" id="explicitTimezone"
    substitutionGroup="xs:facet">
    <xs:annotation>
      <xs:documentation
```

```
            source="http://www.w3.org/TR/xmlschema11-2/#element-explicitTimezone"/
      </xs:annotation>
      <xs:complexType>
        <xs:complexContent>
          <xs:restriction base="xs:facet">
            <xs:sequence>
              <xs:element ref="xs:annotation" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="value" use="required">
              <xs:simpleType>
                <xs:restriction base="xs:NMTOKEN">
                  <xs:enumeration value="optional"/>
                  <xs:enumeration value="required"/>
                  <xs:enumeration value="prohibited"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
            <xs:anyAttribute namespace="##other" processContents="lax"/>
          </xs:restriction>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

B DTD for Datatype Definitions (non-normative)

The DTD for the datatypes-specific aspects of schema documents is given below. Note there is *no* implication here that `schema` MUST be the root element of a document.

DTD for datatype definitions

```
<!--
        DTD for XML Schemas: Part 2: Datatypes

        Id: datatypes.dtd,v 1.1.2.4 2005/01/31 18:40:42 cmsmcq Exp
        Note this DTD is NOT normative, or even definitive.
  -->

<!--
        This DTD cannot be used on its own, it is intended
        only for incorporation in XMLSchema.dtd, q.v.
  -->

<!-- Define all the element names, with optional prefix -->
<!ENTITY % simpleType "%p;simpleType">
<!ENTITY % restriction "%p;restriction">
<!ENTITY % list "%p;list">
<!ENTITY % union "%p;union">
<!ENTITY % maxExclusive "%p;maxExclusive">
<!ENTITY % minExclusive "%p;minExclusive">
<!ENTITY % maxInclusive "%p;maxInclusive">
<!ENTITY % minInclusive "%p;minInclusive">
<!ENTITY % totalDigits "%p;totalDigits">
<!ENTITY % fractionDigits "%p;fractionDigits">

<!ENTITY % maxScale "%p;maxScale">
<!ENTITY % minScale "%p;minScale">
<!ENTITY % length "%p;length">
<!ENTITY % minLength "%p;minLength">
<!ENTITY % maxLength "%p;maxLength">
<!ENTITY % enumeration "%p;enumeration">
<!ENTITY % whiteSpace "%p;whiteSpace">
<!ENTITY % pattern "%p;pattern">
```

```
<!ENTITY % assertion "%p;assertion">

<!ENTITY % explicitTimezone "%p;explicitTimezone">


<!--
        Customization entities for the ATTLIST of each element
        type. Define one of these if your schema takes advantage
        of the anyAttribute='##other' in the schema for schemas
  -->

<!ENTITY % simpleTypeAttrs "">
<!ENTITY % restrictionAttrs "">
<!ENTITY % listAttrs "">
<!ENTITY % unionAttrs "">
<!ENTITY % maxExclusiveAttrs "">
<!ENTITY % minExclusiveAttrs "">
<!ENTITY % maxInclusiveAttrs "">
<!ENTITY % minInclusiveAttrs "">
<!ENTITY % totalDigitsAttrs "">
<!ENTITY % fractionDigitsAttrs "">
<!ENTITY % lengthAttrs "">
<!ENTITY % minLengthAttrs "">
<!ENTITY % maxLengthAttrs "">

<!ENTITY % maxScaleAttrs "">
<!ENTITY % minScaleAttrs "">

<!ENTITY % enumerationAttrs "">
<!ENTITY % whiteSpaceAttrs "">
<!ENTITY % patternAttrs "">
<!ENTITY % assertionAttrs "">
<!ENTITY % explicitTimezoneAttrs "">

<!-- Define some entities for informative use as attribute
        types -->
<!ENTITY % URIref "CDATA">
<!ENTITY % XPathExpr "CDATA">
<!ENTITY % QName "NMTOKEN">
<!ENTITY % QNames "NMTOKENS">
<!ENTITY % NCName "NMTOKEN">
<!ENTITY % nonNegativeInteger "NMTOKEN">
<!ENTITY % boolean "(true|false)">
<!ENTITY % simpleDerivationSet "CDATA">
<!--
        #all or space-separated list drawn from derivationChoice
  -->

<!--
        Note that the use of 'facet' below is less restrictive
        than is really intended:  There should in fact be no
        more than one of each of minInclusive, minExclusive,
        maxInclusive, maxExclusive, totalDigits, fractionDigits,
        length, maxLength, minLength within datatype,
        and the min- and max- variants of Inclusive and Exclusive
        are mutually exclusive. On the other hand,  pattern and
        enumeration and assertion may repeat.
  -->
<!ENTITY % minBound "(%minInclusive; | %minExclusive;)">
<!ENTITY % maxBound "(%maxInclusive; | %maxExclusive;)">
<!ENTITY % bounds "%minBound; | %maxBound;">
<!ENTITY % numeric "%totalDigits; | %fractionDigits; |
%minScale; | %maxScale;">
<!ENTITY % ordered "%bounds; | %numeric;">
<!ENTITY % unordered
   "%pattern; | %enumeration; | %whiteSpace; | %length; |
   %maxLength; | %minLength; | %assertion;
   | %explicitTimezone;">
```

```
<!ENTITY % implementation-defined-facets "">
<!ENTITY % facet "%ordered; | %unordered; %implementation-defined-facets;">
<!ENTITY % facetAttr
        "value CDATA #REQUIRED
        id ID #IMPLIED">
<!ENTITY % fixedAttr "fixed %boolean; #IMPLIED">
<!ENTITY % facetModel "(%annotation;)?">
<!ELEMENT %simpleType;
        ((%annotation;)?, (%restriction; | %list; | %union;))>
<!ATTLIST %simpleType;
    name      %NCName; #IMPLIED
    final     %simpleDerivationSet; #IMPLIED
    id        ID       #IMPLIED
    %simpleTypeAttrs;>
<!-- name is required at top level -->
<!ELEMENT %restriction; ((%annotation;)?,
                         (%restriction1; |
                          ((%simpleType;)?,(%facet;)*)),
                         (%attrDecls;))>
<!ATTLIST %restriction;
    base      %QName;                  #IMPLIED
    id        ID       #IMPLIED
    %restrictionAttrs;>
<!--
        base and simpleType child are mutually exclusive,
        one is required.

        restriction is shared between simpleType and
        simpleContent and complexContent (in XMLSchema.xsd).
        restriction1 is for the latter cases, when this
        is restricting a complex type, as is attrDecls.
  -->
<!ELEMENT %list; ((%annotation;)?,(%simpleType;)?)>
<!ATTLIST %list;
    itemType      %QName;             #IMPLIED
    id        ID       #IMPLIED
    %listAttrs;>
<!--
        itemType and simpleType child are mutually exclusive,
        one is required
  -->
<!ELEMENT %union; ((%annotation;)?,(%simpleType;)*)>
<!ATTLIST %union;
    id            ID       #IMPLIED
    memberTypes   %QNames;            #IMPLIED
    %unionAttrs;>
<!--
        At least one item in memberTypes or one simpleType
        child is required
  -->

<!ELEMENT %maxExclusive; %facetModel;>
<!ATTLIST %maxExclusive;
        %facetAttr;
        %fixedAttr;
        %maxExclusiveAttrs;>
<!ELEMENT %minExclusive; %facetModel;>
<!ATTLIST %minExclusive;
        %facetAttr;
        %fixedAttr;
        %minExclusiveAttrs;>

<!ELEMENT %maxInclusive; %facetModel;>
<!ATTLIST %maxInclusive;
        %facetAttr;
        %fixedAttr;
        %maxInclusiveAttrs;>
<!ELEMENT %minInclusive; %facetModel;>
<!ATTLIST %minInclusive;
```

```
        %facetAttr;
        %fixedAttr;
        %minInclusiveAttrs;>

<!ELEMENT %totalDigits; %facetModel;>
<!ATTLIST %totalDigits;
        %facetAttr;
        %fixedAttr;
        %totalDigitsAttrs;>
<!ELEMENT %fractionDigits; %facetModel;>
<!ATTLIST %fractionDigits;
        %facetAttr;
        %fixedAttr;
        %fractionDigitsAttrs;>
<!ELEMENT %maxScale; %facetModel;>
<!ATTLIST %maxScale;
        %facetAttr;
        %fixedAttr;
        %maxScaleAttrs;>
<!ELEMENT %minScale; %facetModel;>
<!ATTLIST %minScale;
        %facetAttr;
        %fixedAttr;
        %minScaleAttrs;>

<!ELEMENT %length; %facetModel;>
<!ATTLIST %length;
        %facetAttr;
        %fixedAttr;
        %lengthAttrs;>
<!ELEMENT %minLength; %facetModel;>
<!ATTLIST %minLength;
        %facetAttr;
        %fixedAttr;
        %minLengthAttrs;>
<!ELEMENT %maxLength; %facetModel;>
<!ATTLIST %maxLength;
        %facetAttr;
        %fixedAttr;
        %maxLengthAttrs;>

<!-- This one can be repeated -->
<!ELEMENT %enumeration; %facetModel;>
<!ATTLIST %enumeration;
        %facetAttr;
        %enumerationAttrs;>

<!ELEMENT %whiteSpace; %facetModel;>
<!ATTLIST %whiteSpace;
        %facetAttr;
        %fixedAttr;
        %whiteSpaceAttrs;>

<!-- This one can be repeated -->
<!ELEMENT %pattern; %facetModel;>
<!ATTLIST %pattern;
        %facetAttr;
        %patternAttrs;>

<!ELEMENT %assertion; %facetModel;>
<!ATTLIST %assertion;
        %facetAttr;
        %assertionAttrs;>

<!ELEMENT %explicitTimezone; %facetModel;>
<!ATTLIST %explicitTimezone;
        %facetAttr;
        %explicitTimezoneAttrs;>
```

C Illustrative XML representations for the built-in simple type definitions

C.1 Illustrative XML representations for the built-in primitive type definitions

The following, although in the form of a schema document, does not conform to the rules for schema documents defined in this specification. It contains explicit XML representations of the primitive datatypes which need not be declared in a schema document, since they are automatically included in every schema, and indeed must not be declared in a schema document, since it is forbidden to try to derive types with anyAtomicType as the base type definition. It is included here as a form of documentation.

The (not a) schema document for primitive built-in type definitions

```
<?xml version='1.0'?>
<!DOCTYPE xs:schema SYSTEM "../namespace/XMLSchema.dtd" [

<!--
     keep this schema XML1.0 DTD valid
  -->
        <!ENTITY % schemaAttrs 'xmlns:hfp CDATA #IMPLIED'>

        <!ELEMENT hfp:hasFacet EMPTY>
        <!ATTLIST hfp:hasFacet
                name NMTOKEN #REQUIRED>

        <!ELEMENT hfp:hasProperty EMPTY>
        <!ATTLIST hfp:hasProperty
                name NMTOKEN #REQUIRED
                value CDATA #REQUIRED>
]>
<xs:schema
  xmlns:hfp="http://www.w3.org/2001/XMLSchema-hasFacetAndProperty"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  xml:lang="en"
  targetNamespace="http://www.w3.org/2001/XMLSchema">

  <xs:annotation>
    <xs:documentation>
      This document contains XML elements which look like
      definitions for the primitive datatypes.  These definitions are for
      information only; the real built-in definitions are magic.
    </xs:documentation>
    <xs:documentation>
      For each built-in datatype in this schema (both primitive and
      derived) can be uniquely addressed via a URI constructed
      as follows:
        1) the base URI is the URI of the XML Schema namespace
        2) the fragment identifier is the name of the datatype

      For example, to address the int datatype, the URI is:

        http://www.w3.org/2001/XMLSchema#int

      Additionally, each facet definition element can be uniquely
      addressed via a URI constructed as follows:
        1) the base URI is the URI of the XML Schema namespace
        2) the fragment identifier is the name of the facet

      For example, to address the maxInclusive facet, the URI is:
```

```
        http://www.w3.org/2001/XMLSchema#maxInclusive

      Additionally, each facet usage in a built-in datatype definition
      can be uniquely addressed via a URI constructed as follows:
        1) the base URI is the URI of the XML Schema namespace
        2) the fragment identifier is the name of the datatype, followed
           by a period (".") followed by the name of the facet

      For example, to address the usage of the maxInclusive facet in
      the definition of int, the URI is:

        http://www.w3.org/2001/XMLSchema#int.maxInclusive

    </xs:documentation>
  </xs:annotation>
  <xs:simpleType name="string" id="string">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="length"/>
        <hfp:hasFacet name="minLength"/>
        <hfp:hasFacet name="maxLength"/>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="false"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#string"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace value="preserve" id="string.whiteSpace"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="boolean" id="boolean">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="false"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="finite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#boolean"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="boolean.whiteSpace"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="float" id="float">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="true"/>
        <hfp:hasProperty name="cardinality" value="finite"/>
        <hfp:hasProperty name="numeric" value="true"/>
      </xs:appinfo>
```

```
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#float"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="float.whiteSpace"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="double" id="double">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="true"/>
        <hfp:hasProperty name="cardinality" value="finite"/>
        <hfp:hasProperty name="numeric" value="true"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#double"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="double.whiteSpace"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="decimal" id="decimal">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="totalDigits"/>
        <hfp:hasFacet name="fractionDigits"/>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="total"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="true"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#decimal"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="decimal.whiteSpace"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="precisionDecimal" id="precisionDecimal">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="totalDigits"/>
        <hfp:hasFacet name="maxScale"/>
        <hfp:hasFacet name="minScale"/>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
```

```
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="true"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#precisionDeci
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="precisionDecimal.whiteSpac
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="duration" id="duration">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#duration"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="duration.whiteSpace"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="dateTime" id="dateTime">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasFacet name="explicitTimezone"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#dateTime"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="dateTime.whiteSpace"/>
      <xs:explicitTimezone value="optional" id="dateTime.explicitTimezone"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="time" id="time">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasFacet name="explicitTimezone"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
```

```
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#time"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="time.whiteSpace"/>
      <xs:explicitTimezone value="optional" id="time.explicitTimezone"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="date" id="date">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasFacet name="explicitTimezone"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#date"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="date.whiteSpace"/>
      <xs:explicitTimezone value="optional" id="date.explicitTimezone"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="gYearMonth" id="gYearMonth">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasFacet name="explicitTimezone"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#gYearMonth"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="gYearMonth.whiteSpace"/>
      <xs:explicitTimezone value="optional" id="gYearMonth.explicitTimezone"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="gYear" id="gYear">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
```

```
        <hfp:hasFacet name="explicitTimezone"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#gYear"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="gYear.whiteSpace"/>
      <xs:explicitTimezone value="optional" id="gYear.explicitTimezone"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="gMonthDay" id="gMonthDay">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasFacet name="explicitTimezone"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#gMonthDay"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="gMonthDay.whiteSpace"/>
      <xs:explicitTimezone value="optional" id="gMonthDay.explicitTimezone"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="gDay" id="gDay">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasFacet name="explicitTimezone"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#gDay"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="gDay.whiteSpace"/>
      <xs:explicitTimezone value="optional" id="gDay.explicitTimezone"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="gMonth" id="gMonth">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="maxInclusive"/>
        <hfp:hasFacet name="maxExclusive"/>
```

```
        <hfp:hasFacet name="minInclusive"/>
        <hfp:hasFacet name="minExclusive"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasFacet name="explicitTimezone"/>
        <hfp:hasProperty name="ordered" value="partial"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#gMonth"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="gMonth.whiteSpace"/>
      <xs:explicitTimezone value="optional" id="gMonth.explicitTimezone"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="hexBinary" id="hexBinary">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="length"/>
        <hfp:hasFacet name="minLength"/>
        <hfp:hasFacet name="maxLength"/>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="false"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#hexBinary"/>
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="hexBinary.whiteSpace"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="base64Binary" id="base64Binary">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="length"/>
        <hfp:hasFacet name="minLength"/>
        <hfp:hasFacet name="maxLength"/>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="false"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#base64Binary'
    </xs:annotation>
    <xs:restriction base="xs:anyAtomicType">
      <xs:whiteSpace fixed="true" value="collapse" id="base64Binary.whiteSpace"/:
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="anyURI" id="anyURI">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="length"/>
        <hfp:hasFacet name="minLength"/>
        <hfp:hasFacet name="maxLength"/>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="false"/>
```

```
          <hfp:hasProperty name="bounded" value="false"/>
          <hfp:hasProperty name="cardinality" value="countably infinite"/>
          <hfp:hasProperty name="numeric" value="false"/>
        </xs:appinfo>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#anyURI"/>
      </xs:annotation>
      <xs:restriction base="xs:anyAtomicType">
        <xs:whiteSpace fixed="true" value="collapse" id="anyURI.whiteSpace"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="QName" id="QName">
      <xs:annotation>
        <xs:appinfo>
          <hfp:hasFacet name="length"/>
          <hfp:hasFacet name="minLength"/>
          <hfp:hasFacet name="maxLength"/>
          <hfp:hasFacet name="pattern"/>
          <hfp:hasFacet name="enumeration"/>
          <hfp:hasFacet name="whiteSpace"/>
          <hfp:hasFacet name="assertions"/>
          <hfp:hasProperty name="ordered" value="false"/>
          <hfp:hasProperty name="bounded" value="false"/>
          <hfp:hasProperty name="cardinality" value="countably infinite"/>
          <hfp:hasProperty name="numeric" value="false"/>
        </xs:appinfo>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#QName"/>
      </xs:annotation>
      <xs:restriction base="xs:anyAtomicType">
        <xs:whiteSpace fixed="true" value="collapse" id="QName.whiteSpace"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="NOTATION" id="NOTATION">
      <xs:annotation>
        <xs:appinfo>
          <hfp:hasFacet name="length"/>
          <hfp:hasFacet name="minLength"/>
          <hfp:hasFacet name="maxLength"/>
          <hfp:hasFacet name="pattern"/>
          <hfp:hasFacet name="enumeration"/>
          <hfp:hasFacet name="whiteSpace"/>
          <hfp:hasFacet name="assertions"/>
          <hfp:hasProperty name="ordered" value="false"/>
          <hfp:hasProperty name="bounded" value="false"/>
          <hfp:hasProperty name="cardinality" value="countably infinite"/>
          <hfp:hasProperty name="numeric" value="false"/>
        </xs:appinfo>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#NOTATION"/>
        <xs:documentation>
          NOTATION cannot be used directly in a schema; rather a type
          must be derived from it by specifying at least one enumeration
          facet whose value is the name of a NOTATION declared in the
          schema.
        </xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:anyAtomicType">
        <xs:whiteSpace fixed="true" value="collapse" id="NOTATION.whiteSpace"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>
```

C.2 Illustrative XML representations for the built-in ordinary type definitions

The following, although in the form of a schema document, contains XML representations of components already present in all schemas by definition. It is included here as a form of documentation.

Note: These datatypes do not need to be declared in a schema document, since they are automatically included in every schema.

Issue (B-1933):

It is an open question whether this and similar XML documents should be accepted or rejected by software conforming to this specification. The XML Schema Working Group expects to resolve this question in connection with its work on issues relating to schema composition.

In the meantime, some existing schema processors will accept declarations for them; other existing processors will reject such declarations as duplicates.

Illustrative schema document for derived built-in type definitions

```
<?xml version='1.0'?>
<!DOCTYPE xs:schema SYSTEM "../namespace/XMLSchema.dtd" [

<!--
     keep this schema XML1.0 DTD valid
  -->
        <!ENTITY % schemaAttrs 'xmlns:hfp CDATA #IMPLIED'>

        <!ELEMENT hfp:hasFacet EMPTY>
        <!ATTLIST hfp:hasFacet
                name NMTOKEN #REQUIRED>

        <!ELEMENT hfp:hasProperty EMPTY>
        <!ATTLIST hfp:hasProperty
                name NMTOKEN #REQUIRED
                value CDATA #REQUIRED>

]>
<xs:schema
  xmlns:hfp="http://www.w3.org/2001/XMLSchema-hasFacetAndProperty"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  xml:lang="en"
  targetNamespace="http://www.w3.org/2001/XMLSchema">
 <xs:annotation>
    <xs:documentation>
      This document contains XML representations for the
     ordinary non-primitive built-in datatypes
    </xs:documentation>
  </xs:annotation>
  <xs:simpleType name="normalizedString" id="normalizedString">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#normalizedSt
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="replace" id="normalizedString.whiteSpace"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="token" id="token">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#token"/>
    </xs:annotation>
    <xs:restriction base="xs:normalizedString">
      <xs:whiteSpace value="collapse" id="token.whiteSpace"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="language" id="language">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#language"/>
```

```
    </xs:annotation>
    <xs:restriction base="xs:token">
      <xs:pattern value="[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*" id="language.pattern'
        <xs:annotation>
          <xs:documentation source="http://www.ietf.org/rfc/bcp/bcp47.txt">
            pattern specifies the content of section 2.12 of XML 1.0e2
            and RFC 3066 (Revised version of RFC 1766).  N.B. RFC 3066 is now
            obsolete; the grammar of RFC4646 is more restrictive.  So strict
            conformance to the rules for language codes requires extra checking
            beyond validation against this type.
          </xs:documentation>
        </xs:annotation>
      </xs:pattern>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="IDREFS" id="IDREFS">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="length"/>
        <hfp:hasFacet name="minLength"/>
        <hfp:hasFacet name="maxLength"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="false"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#IDREFS"/>
    </xs:annotation>
    <xs:restriction>
      <xs:simpleType>
        <xs:list itemType="xs:IDREF"/>
      </xs:simpleType>
      <xs:minLength value="1" id="IDREFS.minLength"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="ENTITIES" id="ENTITIES">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasFacet name="length"/>
        <hfp:hasFacet name="minLength"/>
        <hfp:hasFacet name="maxLength"/>
        <hfp:hasFacet name="enumeration"/>
        <hfp:hasFacet name="whiteSpace"/>
        <hfp:hasFacet name="pattern"/>
        <hfp:hasFacet name="assertions"/>
        <hfp:hasProperty name="ordered" value="false"/>
        <hfp:hasProperty name="bounded" value="false"/>
        <hfp:hasProperty name="cardinality" value="countably infinite"/>
        <hfp:hasProperty name="numeric" value="false"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#ENTITIES"/>
    </xs:annotation>
    <xs:restriction>
      <xs:simpleType>
        <xs:list itemType="xs:ENTITY"/>
      </xs:simpleType>
      <xs:minLength value="1" id="ENTITIES.minLength"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="NMTOKEN" id="NMTOKEN">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#NMTOKEN"/>
    </xs:annotation>
    <xs:restriction base="xs:token">
      <xs:pattern value="\c+" id="NMTOKEN.pattern">
```

```
          <xs:annotation>
            <xs:documentation source="http://www.w3.org/TR/REC-xml#NT-Nmtoken">
              pattern matches production 7 from the XML spec
            </xs:documentation>
          </xs:annotation>
        </xs:pattern>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="NMTOKENS" id="NMTOKENS">
      <xs:annotation>
        <xs:appinfo>
          <hfp:hasFacet name="length"/>
          <hfp:hasFacet name="minLength"/>
          <hfp:hasFacet name="maxLength"/>
          <hfp:hasFacet name="enumeration"/>
          <hfp:hasFacet name="whiteSpace"/>
          <hfp:hasFacet name="pattern"/>
          <hfp:hasFacet name="assertions"/>
          <hfp:hasProperty name="ordered" value="false"/>
          <hfp:hasProperty name="bounded" value="false"/>
          <hfp:hasProperty name="cardinality" value="countably infinite"/>
          <hfp:hasProperty name="numeric" value="false"/>
        </xs:appinfo>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#NMTOKENS"/>
      </xs:annotation>
      <xs:restriction>
        <xs:simpleType>
          <xs:list itemType="xs:NMTOKEN"/>
        </xs:simpleType>
        <xs:minLength value="1" id="NMTOKENS.minLength"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="Name" id="Name">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#Name"/>
      </xs:annotation>
      <xs:restriction base="xs:token">
        <xs:pattern value="\i\c*" id="Name.pattern">
          <xs:annotation>
            <xs:documentation source="http://www.w3.org/TR/REC-xml#NT-Name">
              pattern matches production 5 from the XML spec
            </xs:documentation>
          </xs:annotation>
        </xs:pattern>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="NCName" id="NCName">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#NCName"/>
      </xs:annotation>
      <xs:restriction base="xs:Name">
        <xs:pattern value="[\i-[:]][\c-[:]]*" id="NCName.pattern">
          <xs:annotation>
            <xs:documentation source="http://www.w3.org/TR/REC-xml-names/#NT-NCName
              pattern matches production 4 from the Namespaces in XML spec
            </xs:documentation>
          </xs:annotation>
        </xs:pattern>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="ID" id="ID">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#ID"/>
      </xs:annotation>
      <xs:restriction base="xs:NCName"/>
    </xs:simpleType>
    <xs:simpleType name="IDREF" id="IDREF">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#IDREF"/>
```

```
    </xs:annotation>
    <xs:restriction base="xs:NCName"/>
  </xs:simpleType>
  <xs:simpleType name="ENTITY" id="ENTITY">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#ENTITY"/>
    </xs:annotation>
    <xs:restriction base="xs:NCName"/>
  </xs:simpleType>
  <xs:simpleType name="integer" id="integer">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#integer"/>
    </xs:annotation>
    <xs:restriction base="xs:decimal">
      <xs:fractionDigits fixed="true" value="0" id="integer.fractionDigits"/>
      <xs:pattern value="[\-+]?[0-9]+" id="integer.pattern"/>

    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="nonPositiveInteger" id="nonPositiveInteger">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#nonPositiveIn
    </xs:annotation>
    <xs:restriction base="xs:integer">
      <xs:maxInclusive value="0" id="nonPositiveInteger.maxInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="negativeInteger" id="negativeInteger">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#negativeInteg
    </xs:annotation>
    <xs:restriction base="xs:nonPositiveInteger">
      <xs:maxInclusive value="-1" id="negativeInteger.maxInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="long" id="long">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasProperty name="bounded" value="true"/>
        <hfp:hasProperty name="cardinality" value="finite"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#long"/>
    </xs:annotation>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="-9223372036854775808" id="long.minInclusive"/>
      <xs:maxInclusive value="9223372036854775807" id="long.maxInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="int" id="int">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#int"/>
    </xs:annotation>
    <xs:restriction base="xs:long">
      <xs:minInclusive value="-2147483648" id="int.minInclusive"/>
      <xs:maxInclusive value="2147483647" id="int.maxInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="short" id="short">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#short"/>
    </xs:annotation>
    <xs:restriction base="xs:int">
      <xs:minInclusive value="-32768" id="short.minInclusive"/>
      <xs:maxInclusive value="32767" id="short.maxInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="byte" id="byte">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#byte"/>
```

```
    </xs:annotation>
    <xs:restriction base="xs:short">
      <xs:minInclusive value="-128" id="byte.minInclusive"/>
      <xs:maxInclusive value="127" id="byte.maxInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="nonNegativeInteger" id="nonNegativeInteger">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#nonNegativeIn
    </xs:annotation>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0" id="nonNegativeInteger.minInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="unsignedLong" id="unsignedLong">
    <xs:annotation>
      <xs:appinfo>
        <hfp:hasProperty name="bounded" value="true"/>
        <hfp:hasProperty name="cardinality" value="finite"/>
      </xs:appinfo>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#unsignedLong"
    </xs:annotation>
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:maxInclusive value="18446744073709551615" id="unsignedLong.maxInclusive
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="unsignedInt" id="unsignedInt">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#unsignedInt"/
    </xs:annotation>
    <xs:restriction base="xs:unsignedLong">
      <xs:maxInclusive value="4294967295" id="unsignedInt.maxInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="unsignedShort" id="unsignedShort">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#unsignedShort
    </xs:annotation>
    <xs:restriction base="xs:unsignedInt">
      <xs:maxInclusive value="65535" id="unsignedShort.maxInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="unsignedByte" id="unsignedByte">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#unsignedByte"
    </xs:annotation>
    <xs:restriction base="xs:unsignedShort">
      <xs:maxInclusive value="255" id="unsignedByte.maxInclusive"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="positiveInteger" id="positiveInteger">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#positiveInteg
    </xs:annotation>
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:minInclusive value="1" id="positiveInteger.minInclusive"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="yearMonthDuration">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#yearMonthDura
        This type includes just those durations expressed in years and months.
        Since the pattern given excludes days, hours, minutes, and seconds,
        the values of this type have a seconds property of zero.  They are
        totally ordered.
      </xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:duration">
```

```
        <xs:pattern id="yearMonthDuration.pattern" value="[^DT]*"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="dayTimeDuration">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#dayTimeDurati
          This type includes just those durations expressed in days, hours, minutes
          The pattern given excludes years and months, so the values of this type
          have a months property of zero.  They are totally ordered.
        </xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:duration">
        <xs:pattern id="dayTimeDuration.pattern" value="[^YM]*(T.*)?"/>
       </xs:restriction>
    </xs:simpleType>
      <xs:simpleType name="dateTimeStamp" id="dateTimeStamp">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/xmlschema11-2/#dateTimeStamp
          This datatype includes just those dateTime values Whose explicitTimezone
          is present.  They are totally ordered.
        </xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:dateTime">
        <xs:explicitTimezone fixed="true"
          id="dateTimeStamp.explicitTimezone" value="required"/>
       </xs:restriction>
    </xs:simpleType>

  </xs:schema>
```

D Built-up Value Spaces

Some datatypes, such as integer, describe well-known mathematically abstract systems. Others, such as the date/time datatypes, describe "real-life", "applied" systems. Certain of the systems described by datatypes, both abstract and applied, have values in their value spaces most easily described as things having several *properties*, which in turn have values which are in some sense "primitive" or are from the value spaces of simpler datatypes.

In this document, the arguments to functions are assumed to be "call by value" unless explicitly noted to the contrary, meaning that if the argument is modified during the processing of the algorithm, that modification is *not* reflected in the "outside world". On the other hand, the arguments to procedures are assumed to be "call by location", meaning that modifications *are* so reflected, since that is the only way the processing of the algorithm can have any effect.

Properties always have values. [Definition:] An **optional** property is *permitted* but not *required* to have the distinguished value ***absent***.

[Definition:] Throughout this specification, the value ***absent*** is used as a distinguished value to indicate that a given instance of a property "has no value" or "is absent". This should not be interpreted as constraining implementations, as for instance between using a ***null*** value for such properties or not representing them at all.

Those values that are more primitive, and are used (among other things) herein to construct object value spaces but which we do not explicitly define are described here:

- A **number (without precision)** is an ordinary mathematical number; 1, 1.0, and 1.000000000000 are the same number. The decimal numbers and integers generally used in the algorithms of appendix Function Definitions (§E) are such ordinary numbers, not carrying precision.

- [Definition:] A **special value** is an object whose only relevant properties for purposes of this specification are that it is distinct from, and unequal to, any other values (special or otherwise). A few special values in different value spaces (e.g. ***positiveInfinity***, ***negativeInfinity***, and ***notANumber*** in float and double) share names. Thus, special values can be distinguished from each other in the general case by considering both the name and the primitive datatype of the value; in some cases, of course, the name alone suffices to identify the value uniquely.

  Note: In the case of float and double, the ·special values· are members of the datatype's ·value space·. In the case of precisionDecimal, the term ·special value· denotes not members of the precisionDecimal ·value space· but some possible values of the *·numericalValue·* property they possess: ***positiveInfinity***, ***negativeInfinity***, and ***notANumber***. (Less formally, the term may sometimes also be used for the precisionDecimal values whose *·numericalValue·* property has such a ·special value·. These precisionDecimal values do not quite fit the definition of ·special value·, since they do have relevant properties other than distinctness from other values.)

D.1 Numerical Values

The following standard operators are defined here in case the reader is unsure of their definition:

- [Definition:] If ***m*** and ***n*** are numbers, then ***m*** **div** ***n*** is the greatest integer less than or equal to ***m*** / ***n*** .
- [Definition:] If ***m*** and ***n*** are numbers, then ***m*** **mod** ***n*** is ***m*** − ***n*** × (***m*** ·div· ***n***) .

Note: ***n*** ·div· 1 is a convenient and short way of expressing "the greatest integer less than or equal to ***n***".

D.1.1 Exact Lexical Mappings

Numerals and Fragments Thereof
digit ::= [0-9] *unsignedNoDecimalPtNumeral* ::= *digit*+ *noDecimalPtNumeral* ::= ('+' | '-')? *unsignedNoDecimalPtNumeral* *fracFrag* ::= *digit*+ *unsignedDecimalPtNumeral* ::= (*unsignedNoDecimalPtNumeral* '.' *fracFrag*?) | ('.' *fracFrag*) *unsignedFullDecimalPtNumeral* ::= *unsignedNoDecimalPtNumeral* '.' *fracFrag*

decimalPtNumeral ::= ('+' | '-')? *unsignedDecimalPtNumeral*

unsignedScientificNotationNumeral ::=
(*unsignedNoDecimalPtNumeral* | *unsignedDecimalPtNumeral*) ('e' | 'E') *noDecimalPtNumeral*

scientificNotationNumeral ::= ('+' | '-')? *unsignedScientificNotationNumeral*

Generic Numeral-to-Number Lexical Mappings

·unsignedNoDecimalMap· (***N***) → integer
Maps an *unsignedNoDecimalPtNumeral* to its numerical value.

·noDecimalMap· (***N***) → integer
Maps an *noDecimalPtNumeral* to its numerical value.

·unsignedDecimalPtMap· (***D***) → decimal number
Maps an *unsignedDecimalPtNumeral* to its numerical value.

·decimalPtMap· (***N***) → decimal number
Maps a *decimalPtNumeral* to its numerical value.

·scientificMap· (***N***) → decimal number
Maps a *scientificNotationNumeral* to its numerical value.

Generic Number to Numeral Canonical Mappings

·unsignedNoDecimalPtCanonicalMap· (***i***) → *unsignedNoDecimalPtNumeral*
Maps a nonnegative integer to a *unsignedNoDecimalPtNumeral*, its ·canonical representation·.

·noDecimalPtCanonicalMap· (***i***) → *noDecimalPtNumeral*
Maps an integer to a *noDecimalPtNumeral*, its ·canonical representation·.

·unsignedDecimalPtCanonicalMap· (***n***) → *unsignedDecimalPtNumeral*
Maps a nonnegative decimal number to a *unsignedDecimalPtNumeral*, its ·canonical representation·.

·decimalPtCanonicalMap· (***n***) → *decimalPtNumeral*
Maps a decimal number to a *decimalPtNumeral*, its ·canonical representation·.

·unsignedScientificCanonicalMap· (***n***) → *unsignedScientificNotationNumeral*

Maps a nonnegative decimal number to a *unsignedScientificNotationNumeral*, its ·canonical representation·.

·scientificCanonicalMap· (***n***) → *scientificNotationNumeral*

Maps a decimal number to a *scientificNotationNumeral*, its ·canonical representation·.

Some numerical datatypes include some or all of three non-numerical ·special values·: ***positiveInfinity***, ***negativeInfinity***, and ***notANumber***. Their lexical spaces include non-numeral lexical representations for these non-numeric values:

Special Non-numerical Lexical Representations Used With Numerical Datatypes

minimalNumericalSpecialRep ::= '`INF`' | '`-INF`' | '`NaN`'

numericalSpecialRep ::= '`+INF`' | *minimalNumericalSpecialRep*

Lexical Mapping for Non-numerical ·Special Values· Used With Numerical Datatypes

·specialRepValue· (***S***) → a ·special value·

Maps the ·lexical representations· of ·special values· used with some numerical datatypes to those ·special values·.

Canonical Mapping for Non-numerical ·Special Values· Used with Numerical Datatypes

·specialRepCanonicalMap· (***c***) → *numericalSpecialRep*

Maps the ·special values· used with some numerical datatypes to their ·canonical representations·.

D.2 Date/time Values

There are several different primitive but related datatypes defined in the specification which pertain to various combinations of dates and times, and parts thereof. They all use related value-space models, which are described in detail in this section. It is not difficult for a casual reader of the descriptions of the individual datatypes elsewhere in this specification to misunderstand some of the details of just what the datatypes are intended to represent, so more detail is presented here in this section.

All of the value spaces for dates and times described here represent moments or periods of time in Universal Coordinated Time (UTC). [Definition:] **Universal Coordinated Time** (**UTC**) is an adaptation of TAI which closely approximates UT1 by adding ·leap-seconds· to selected ·UTC· days.

[Definition:] A **leap-second** is an additional second added to the last day of December, June, October, or March, when such an adjustment is deemed necessary by the International Earth Rotation and Reference Systems Service in order to keep ·UTC· within 0.9 seconds of observed astronomical time. When leap seconds are introduced, the last minute in the day has more than sixty seconds. In theory leap seconds can also be removed from a day, but this has not yet occurred. (See [International Earth Rotation Service (IERS)], [ITU-R TF.460-6].) Leap seconds are *not* supported by the types defined here.

Because the dateTime type and other date- and time-related types defined in this specification do not support leap seconds, there are portions of the ·UTC· timeline which cannot be represented by values of these types. Users whose applications require that leap seconds be represented and that date/time arithmetic take historically occurring leap seconds into account will wish to make appropriate adjustments at the application level, or to use other types.

D.2.1 The Seven-property Model

There are two distinct ways to model moments in time: either by tracking their year, month, day, hour, minute and second (with fractional seconds as needed), or by tracking their time (measured generally in seconds or days) from some starting moment. Each has its advantages. The two are isomorphic. For definiteness, we choose to model the first using five integer and one decimal number properties. We superimpose the second by providing one decimal number-valued function which gives the corresponding count of seconds from zero (the "time on the time line").

There is also a seventh integer property which specifies the time zone offset as the number of minutes of offset from UTC. Values for the six primary properties are always stored in their "local" values (the values shown in the lexical representations), rather than converted to ·UTC·.

Properties of Date/time Seven-property Models

·year·
an integer

·month·
an integer between 1 and 12 inclusive

·day·
an integer between 1 and 31 inclusive, possibly restricted further depending on *·month·* and *·year·*

·hour·
an integer between 0 and 23 inclusive

·minute·
an integer between 0 and 59 inclusive

·second·

a decimal number greater than or equal to 0 and less than 60.

·timezoneOffset·

an ·optional· integer between −840 and 840 inclusive

Non-negative values of the properties map to the years, months, days of month, etc. of the Gregorian calendar in the obvious way. Values less than 1582 in the *·year·* property represent years in the "proleptic Gregorian calendar". A value of zero in the *·year·* property represents the year 1 BCE; a value of −1 represents the year 2 BCE, −2 is 3 BCE, etc.

> **Note:** In version 1.0 of this specification, the *·year·* property was not permitted to have the value zero. The year before the year 1 in the proleptic Gregorian calendar, traditionally referred to as 1 BC or as 1 BCE, was represented by a *·year·* value of −1, 2 BCE by −2, and so forth. Of course, many, perhaps most, references to 1 BCE (or 1 BC) actually refer not to a year in the proleptic Gregorian calendar but to a year in the Julian or "old style" calendar; the two correspond approximately but not exactly to each other.
>
> In this version of this specification, two changes are made in order to agree with existing usage. First, *·year·* is permitted to have the value zero. Second, the interpretation of *·year·* values is changed accordingly: a *·year·* value of zero represents 1 BCE, −1 represents 2 BCE, etc. This representation simplifies interval arithmetic and leap-year calculation for dates before the common era (which may be why astronomers and others interested in such calculations with the proleptic Gregorian calendar have adopted it), and is consistent with the current edition of [ISO 8601].
>
> Note that 1 BCE, 5 BCE, and so on (years 0000, -0004, etc. in the lexical representation defined here) are leap years in the proleptic Gregorian calendar used for the date/time datatypes defined here. Version 1.0 of this specification was unclear about the treatment of leap years before the common era. If existing schemas or data specify dates of 29 February for any years before the common era, then some values giving a date of 29 February which were valid under a plausible interpretation of XSD 1.0 will be invalid under this specification, and some which were invalid will be valid. With that possible exception, schemas and data valid under the old interpretation remain valid under the new.

The model just described is called herein the "seven-property" model for date/time datatypes. It is used "as is" for dateTime; all other date/time datatypes except duration use the same model except that some of the six primary properties are *required* to have the value ***absent***, instead of being required to have a numerical value. (An ·optional· property, like *·timezoneOffset·*, is always *permitted* to have the value ***absent***.)

·timezoneOffset· values are limited to 14 hours, which is 840 (= 60 × 14) minutes.

> **Note:** Leap-seconds are not permitted

Readers interested in when leap-seconds have been introduced should consult [USNO Historical List], which includes a list of times when the difference between TAI and ·UTC· has changed. Because the simple types defined here do not support leap seconds, they cannot be used to represent the final second, in ·UTC·, of any of the days containing one. If it is important, at the application level, to track the occurrence of leap seconds, then users will need to make special arrangements for special handling of them and of time intervals crossing them.

While calculating, property values from the dateTime 1972-12-31T00:00:00 are used to fill in for those that are ***absent***, except that if *·day·* is ***absent*** but *·month·* is not, the largest permitted day for that month is used.

Time on Timeline for Date/time Seven-property Model Datatypes
·timeOnTimeline· (***dt***) → decimal number
Maps a date/timeSevenPropertyModel value to the decimal number representing its position on the "time line".

Values from any one date/time datatype using the seven-component model (all except duration) are ordered the same as their *·timeOnTimeline·* values, except that if one value's *·timezoneOffset·* is ***absent*** and the other's is not, and using maximum and minimum *·timezoneOffset·* values for the one whose *·timezoneOffset·* is actually ***absent*** changes the resulting (strict) inequality, the original two values are incomparable.

D.2.2 Lexical Mappings

[Definition:] Each lexical representation is made up of certain **date/time fragments**, each of which corresponds to a particular property of the datatype value. They are defined by the following productions.

Date/time Lexical Representation Fragments

yearFrag ::= '-'? (([1-9] digit digit digit+)) | ('0' digit digit digit))

monthFrag ::= ('0' [1-9]) | ('1' [0-2])

dayFrag ::= ('0' [1-9]) | ([12] digit) | ('3' [01])

hourFrag ::= ([01] digit) | ('2' [0-3])

minuteFrag ::= [0-5] digit

secondFrag ::= ([0-5] digit) ('.' digit+)?

endOfDayFrag ::= '24:00:00' ('.' '0'+)?

timezoneFrag ::= 'Z' | ('+' | '-') (('0' digit | '1' [0-3]) ':' minuteFrag | '14:00')

Each fragment other than *timezoneFrag* defines a subset of the ·lexical space· of decimal; the corresponding ·lexical mapping· is the decimal ·lexical mapping· restricted to that subset. These fragment ·lexical mappings· are combined separately for each date/time datatype (other than duration) to make up ·the complete lexical mapping· for that datatype. The *·yearFragValue·* mapping is used to obtain the value of the *·year·* property, the *·monthFragValue·* mapping is used to obtain the value of the *·month·* property, etc. Each datatype which specifies some properties to be mandatorily ***absent*** also does not permit the corresponding lexical fragments in its lexical representations.

Partial Date/time Lexical Mappings

·yearFragValue· (***YR***) → integer
: Maps a *yearFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·year·* property of a date/timeSevenPropertyModel value.

·monthFragValue· (***MO***) → integer
: Maps a *monthFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·month·* property of a date/timeSevenPropertyModel value.

·dayFragValue· (***DA***) → integer
: Maps a *dayFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·day·* property of a date/timeSevenPropertyModel value.

·hourFragValue· (***HR***) → integer
: Maps a *hourFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·hour·* property of a date/timeSevenPropertyModel value.

·minuteFragValue· (***MI***) → integer
: Maps a *minuteFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·minute·* property of a date/timeSevenPropertyModel value.

·secondFragValue· (***SE***) → decimal number
: Maps a *secondFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto a decimal number, presumably the *·second·* property of a date/timeSevenPropertyModel value.

·timezoneFragValue· (***TZ***) → integer
: Maps a *timezoneFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·timezoneOffset·* property of a date/timeSevenPropertyModel value.

(The redundancy between '`Z`', '`+00:00`', and '`-00:00`', and the possibility of trailing fractional '`0`' digits for *secondFrag*, are the only redundancies preventing these mappings from being one-to-one.)

The following fragment ·canonical mappings· for each value-object property are combined as appropriate to make the ·canonical mapping· for each date/time datatype (other than duration):

Partial Date/time Canonical Mappings

·yearCanonicalFragmentMap· (***y***) → *yearFrag*

Maps an integer, presumably the *·year·* property of a date/timeSevenPropertyModel value, onto a *yearFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

·monthCanonicalFragmentMap· (***m***) → *monthFrag*

Maps an integer, presumably the *·month·* property of a date/timeSevenPropertyModel value, onto a *monthFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

·dayCanonicalFragmentMap· (***d***) → *dayFrag*

Maps an integer, presumably the *·day·* property of a date/timeSevenPropertyModel value, onto a *dayFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

·hourCanonicalFragmentMap· (***h***) → *hourFrag*

Maps an integer, presumably the *·hour·* property of a date/timeSevenPropertyModel value, onto a *hourFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

·minuteCanonicalFragmentMap· (***m***) → *minuteFrag*

Maps an integer, presumably the *·minute·* property of a date/timeSevenPropertyModel value, onto a *minuteFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

·secondCanonicalFragmentMap· (***s***) → *secondFrag*

Maps a decimal number, presumably the *·second·* property of a date/timeSevenPropertyModel value, onto a *secondFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

·timezoneCanonicalFragmentMap· (***t***) → *timezoneFrag*

Maps an integer, presumably the *·timezoneOffset·* property of a date/timeSevenPropertyModel value, onto a *timezoneFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

E Function Definitions

The more important functions and procedures defined here are summarized in the text When there is a text summary, the name of the function in each is a "hot-link" to the same name in the other. All other links to these functions link to the complete definition in this section.

E.1 Generic Number-related Functions

The following functions are used with various numeric and date/time datatypes.

Auxiliary Functions for Operating on Numeral Fragments

·digitValue· (***d***) → integer

Maps each digit to its numerical value.

Arguments:

d : matches *digit*

Result:

a nonnegative integer less than ten

Algorithm:

Return

- 0 when ***d*** = '`0`' ,
- 1 when ***d*** = '`1`' ,
- 2 when ***d*** = '`2`' ,
- *etc.*

·digitSequenceValue· (***S***) → integer

Maps a sequence of digits to the position-weighted sum of the terms numerical values.

Arguments:

S : a finite sequence of ·literals·, each term matching *digit*.

Result:

a nonnegative integer

Algorithm:

Return the sum of *·digitValue·*(S_i) × $10^{\text{length}(S)-i}$ where ***i*** runs over the domain of ***S***.

·fractionDigitSequenceValue· (***S***) → integer

Maps a sequence of digits to the position-weighted sum of the terms numerical values, weighted appropriately for fractional digits.

Arguments:

S : a finite sequence of ·literals·, each term matching *digit*.

Result:

a nonnegative integer

Algorithm:

Return the sum of *·digitValue·*(S_i) − 10^{-i} where ***i*** runs over the domain of ***S***.

·fractionFragValue· (***N***) → decimal number

Maps a *fracFrag* to the appropriate fractional decimal number.

Arguments:

N : matches *fracFrag*

Result:

a nonnegative decimal number

Algorithm:

N is necessarily the left-to-right concatenation of a finite sequence ***S*** of ·literals·, each term matching *digit*.
Return *·fractionDigitSequenceValue·*(***S***).

Generic Numeral-to-Number Lexical Mappings

·unsignedNoDecimalMap· (***N***) → integer

Maps an *unsignedNoDecimalPtNumeral* to its numerical value.

Arguments:

N : matches *unsignedNoDecimalPtNumeral*

Result:

a nonnegative integer

Algorithm:

N is the left-to-right concatenation of a finite sequence ***S*** of ·literals·, each term matching *digit*.

Return *·digitSequenceValue·*(***S***).

·noDecimalMap· (***N***) → integer

Maps an *noDecimalPtNumeral* to its numerical value.

Arguments:

N : matches *noDecimalPtNumeral*

Result:

an integer

Algorithm:

N necessarily consists of an optional sign('+' or '-') and then a ·literal· ***U*** that matches *unsignedNoDecimalPtNumeral*.

Return

- −1 × *·unsignedNoDecimalMap·*(***U***) when '-' is present, and
- *·unsignedNoDecimalMap·*(***U***) otherwise.

·unsignedDecimalPtMap· (***D***) → decimal number

Maps an *unsignedDecimalPtNumeral* to its numerical value.

Arguments:

D : matches *unsignedDecimalPtNumeral*

Result:

a nonnegative decimal number

Algorithm:

D necessarily consists of an optional ·literal· ***N*** matching *unsignedNoDecimalPtNumeral*, a decimal point, and then an optional ·literal· ***F*** matching *fracFrag*.

Return

- *·unsignedNoDecimalMap·*(***N***) when ***F*** is not present,
- *·fractionFragValue·*(***F***) when ***N*** is not present, and
- *·unsignedNoDecimalMap·*(***N***) + *·fractionFragValue·*(***F***) otherwise.

·decimalPtMap· (***N***) → decimal number

Maps a *decimalPtNumeral* to its numerical value.

Arguments:

N : matches *decimalPtNumeral*

Result:

a decimal number

Algorithm:

N necessarily consists of an optional sign('+' or '-') and then an instance ***U*** of *unsignedDecimalPtNumeral*.

Return

- −*·unsignedDecimalPtMap·*(***U***)   when '-' is present, and
- *·unsignedDecimalPtMap·*(***U***)   otherwise.

·scientificMap· (***N***) → decimal number

Maps a *scientificNotationNumeral* to its numerical value.

Arguments:

N : matches *scientificNotationNumeral*

Result:

a decimal number

Algorithm:

N necessarily consists of an instance ***C*** of either *noDecimalPtNumeral* or *decimalPtNumeral*, either an 'e' or an 'E', and then an instance ***E*** of *noDecimalPtNumeral*.

Return

- *·decimalPtMap·*(***C***) − 10 ^ *·unsignedDecimalPtMap·*(***E***)   when a '.' is present in ***N***, and
- *·noDecimalMap·*(***C***) − 10 ^ *·unsignedDecimalPtMap·*(***E***)   otherwise.

Auxiliary Functions for Producing Numeral Fragments

·digit· (***i***) → *digit*

Maps each integer between 0 and 9 to the corresponding *digit*.

Arguments:

i : between 0 and 9 inclusive

Result:

matches *digit*

Algorithm:

Return

- '0'   when ***i*** = 0 ,
- '1'   when ***i*** = 1 ,
- '2'   when ***i*** = 2 ,
- etc.

·digitRemainderSeq· (***i***) → sequence of integers

Maps each nonnegative integer to a sequence of integers used by *·digitSeq·* to ultimately create an *unsignedNoDecimalPtNumeral*.

Arguments:

i : a nonnegative integer

Result:
sequence of nonnegative integers
Algorithm:
Return that sequence ***s*** for which

- $s_0 = i$ and
- $s_{j+1} = s_j$ ·div· 10 .

·digitSeq· (***i***) → sequence of integers
Maps each nonnegative integer to a sequence of integers used by *·unsignedNoDecimalPtCanonicalMap·* to create an *unsignedNoDecimalPtNumeral*.

Arguments:
i : a nonnegative integer
Result:
sequence of integers where each term is between 0 and 9 inclusive
Algorithm:
Return that sequence ***s*** for which $s_j =$ *·digitRemainderSeq·*$(i)_j$ ·mod· 10 .

·lastSignificantDigit· (***s***) → integer
Maps a sequence of nonnegative integers to the index of the first zero term.

Arguments:
s : a sequence of nonnegative integers
Result:
a nonnegative integer
Algorithm:
Return the smallest nonnegative integer ***j*** such that $s(i)_{j+1}$ is 0.

·FractionDigitRemainderSeq· (***f***) → sequence of decimal numbers
Maps each nonnegative decimal number less than 1 to a sequence of decimal numbers used by *·fractionDigitSeq·* to ultimately create an *unsignedNoDecimalPtNumeral*.

Arguments:
f : nonnegative and less than 1
Result:
a sequence of nonnegative decimal numbers
Algorithm:
Return that sequence ***s*** for which

- $s_0 = f - 10$, and
- $s_{j+1} = (s_j$ ·mod· $1) - 10$.

·fractionDigitSeq· (***f***) → sequence of integers
Maps each nonnegative decimal number less than 1 to a sequence of integers used by *·fractionDigitsCanonicalFragmentMap·* to ultimately create an *unsignedNoDecimalPtNumeral*.

Arguments:

f : nonnegative and less than 1

Result:

a sequence of integer;s where each term is between 0 and 9 inclusive

Algorithm:

Return that sequence ***s*** for which s_j = *·FractionDigitRemainderSeq·*(***f***)$_j$ ·div· 1 .

·fractionDigitsCanonicalFragmentMap· (***f***) → *fracFrag*

Maps each nonnegative decimal number less than 1 to a ·literal· used by *·unsignedDecimalPtCanonicalMap·* to create an *unsignedDecimalPtNumeral*.

Arguments:

f : nonnegative and less than 1

Result:

matches *fracFrag*

Algorithm:

Return *·digit·*(*·fractionDigitSeq·*(***f***)$_0$) & . . . & *·digit·*(*·fractionDigitSeq·*(***f***)$_{·lastSignificantDigit·(·FractionDigitRemainderSeq·(f))}$) .

Generic Number to Numeral Canonical Mappings

·unsignedNoDecimalPtCanonicalMap· (***i***) → *unsignedNoDecimalPtNumeral*

Maps a nonnegative integer to a *unsignedNoDecimalPtNumeral*, its ·canonical representation·.

Arguments:

i : a nonnegative integer

Result:

matches *unsignedNoDecimalPtNumeral*

Algorithm:

Return *·digit·*(*·digitSeq·*(***i***)$_{·lastSignificantDigit·(·digitRemainderSeq·(i))}$) & . . . & *·digit·*(*·digitSeq·*(***i***)$_0$) . (Note that the concatenation is in reverse order.)

·noDecimalPtCanonicalMap· (***i***) → *noDecimalPtNumeral*

Maps an integer to a *noDecimalPtNumeral*, its ·canonical representation·.

Arguments:

i : an integer

Result:

matches *noDecimalPtNumeral*

Algorithm:

Return

- '-' & *·unsignedNoDecimalPtCanonicalMap·*(−***i***) when ***i*** is negative,
- *·unsignedNoDecimalPtCanonicalMap·*(***i***) otherwise.

·unsignedDecimalPtCanonicalMap· (***n***) → *unsignedDecimalPtNumeral*

Maps a nonnegative decimal number to a *unsignedDecimalPtNumeral*, its ·canonical representation·.

Arguments:

n : a nonnegative decimal number

Result:
matches *unsignedDecimalPtNumeral*

Algorithm:
Return ·*unsignedNoDecimalPtCanonicalMap*·(*n*·div·1) & '.' & ·*fractionDigitsCanonicalFragmentMap*·(*n*·mod·1) .

·*decimalPtCanonicalMap*· (*n*) → *decimalPtNumeral*
Maps a decimal number to a *decimalPtNumeral*, its ·canonical representation·.

Arguments:
n : a decimal number

Result:
matches *decimalPtNumeral*

Algorithm:
Return

- '-' & ·*unsignedDecimalPtCanonicalMap*·(−*i*) when *i* is negative,
- ·*unsignedDecimalPtCanonicalMap*·(*i*) otherwise.

·*unsignedScientificCanonicalMap*· (*n*) → *unsignedScientificNotationNumeral*
Maps a nonnegative decimal number to a *unsignedScientificNotationNumeral*, its ·canonical representation·.

Arguments:
n : a nonnegative decimal number

Result:
matches *unsignedScientificNotationNumeral*

Algorithm:
Return ·*unsignedDecimalPtCanonicalMap*·(n / $10^{\log(n) \cdot \mathrm{div} \cdot 1}$) & 'E' & ·*noDecimalPtCanonicalMap*·(log(*n*) ·div· 1)

·*scientificCanonicalMap*· (*n*) → *scientificNotationNumeral*
Maps a decimal number to a *scientificNotationNumeral*, its ·canonical representation·.

Arguments:
n : a decimal number

Result:
matches *scientificNotationNumeral*

Algorithm:
Return

- '-' & ·*unsignedScientificCanonicalMap*·(−*n*) when *n* is negative,
- ·*unsignedScientificCanonicalMap*·(*i*) otherwise.

For example:

- 123.4567 ·mod· 1 = 0.4567 and 123.4567 ·div· 1 = 123 .

- ·digitRemainderSeq·(123) is 123 , 12 , 1 , 0 , 0 , ...
- ·digitSeq·(123) is 3 , 2 , 1 , 0 , 0 , ...
- ·lastSignificantDigit·(·digitRemainderSeq·(123)) = 2 (Sequences count from 0.)
- ·unsignedNoDecimalPtCanonicalMap·(123) = '123'
- ·FractionDigitRemainderSeq·(0.4567) is 4.567 , 5.67 , 6.7 , 7 , 0 , 0 , ...
- ·fractionDigitSeq·(0.4567) is 4 , 5 , 6 , 7 , 0 , 0 , ...
- ·lastSignificantDigit·(·FractionDigitRemainderSeq·(0.4567)) = 3
- ·fractionDigitsCanonicalFragmentMap·(0.4567) = '4567'
- ·unsignedDecimalPtCanonicalMap·(123.4567) = '123.4567'

Lexical Mapping for Non-numerical ·Special Values· Used With Numerical Datatypes

·specialRepValue· (***S***) → a ·special value·
Maps the ·lexical representations· of ·special values· used with some numerical datatypes to those ·special values·.

Arguments:
S : matches *numericalSpecialRep*

Result:
one of ***positiveInfinity***, ***negativeInfinity***, or ***notANumber***.

Algorithm:
Return

- ***positiveInfinity*** when ***S*** is 'INF' or '+INF',
- ***negativeInfinity*** when ***S*** is '-INF', and
- ***notANumber*** when ***S*** is 'NaN'

Canonical Mapping for Non-numerical ·Special Values· Used with Numerical Datatypes

·specialRepCanonicalMap· (***c***) → *numericalSpecialRep*
Maps the ·special values· used with some numerical datatypes to their ·canonical representations·.

Arguments:
c : one of ***positiveInfinity***, ***negativeInfinity***, and ***notANumber***

Result:
matches *numericalSpecialRep*

Algorithm:
Return

- 'INF' when ***c*** is ***positiveInfinity***
- '-INF' when ***c*** is ***negativeInfinity***
- 'NaN' when ***c*** is ***notANumber***

Auxiliary Functions for Reading Instances of *pDecimalRep*

·decimalPtPrecision· (***LEX***) → integer
Maps a *decimalPtNumeral* onto an integer; used in calculating the *·arithmeticPrecision·* of a precisionDecimal value.

Arguments:
LEX : matches *decimalPtNumeral*

Result:
an integer

Algorithm:
LEX necessarily contains a decimal point ('.') and may optionally contain a following *fracFrag* ***F*** consisting of some number ***n*** of *digit*s.
Return

- ***n*** when ***F*** is present, and
- 0 otherwise.

·scientificPrecision· (***LEX***) → integer
Maps a *scientificNotationNumeral* onto an integer; used in calculating the *·arithmeticPrecision·* of a precisionDecimal value.

Arguments:
LEX : matches *scientificNotationNumeral*

Result:
an integer

Algorithm:
LEX necessarily contains a *noDecimalPtNumeral* or *decimalPtNumeral* ***C*** preceding an exponent indicator ('E' or 'e', and a following *noDecimalPtNumeral* ***E***.
Return

- −1 × *·noDecimalMap·*(***E***) when ***C*** is a *noDecimalPtNumeral*, and
- *·decimalPtPrecision·*(***C***) − *·noDecimalMap·*(***E***) otherwise.

Lexical Mapping

·precisionDecimalLexicalMap· (***LEX***) → precisionDecimal
Maps a *pDecimalRep* onto a complete precisionDecimal value.

Arguments:
LEX : matches *pDecimalRep*

Result:
a precisionDecimal value

Algorithm:

Let ***pD*** be a complete precisionDecimal value.

1. Set ***pD***'s *·numericalValue·* to
   - *·noDecimalMap·*(***LEX***) when ***LEX*** is an instance of *noDecimalPtNumeral*,
   - *·decimalPtMap·*(***LEX***) when ***LEX*** is an instance of *decimalPtNumeral*,
   - *·scientificMap·*(***LEX***) when ***LEX*** is an instance of *scientificNotationNumeral* and
   - *·specialRepValue·*(***LEX***) otherwise.
2. Set ***pD***'s *·arithmeticPrecision·* to
   - 0 when ***LEX*** is a *noDecimalPtNumeral*,
   - *·decimalPtPrecision·*(***LEX***) when ***LEX*** is a *decimalPtNumeral*,
   - *·scientificPrecision·*(***LEX***) when ***LEX*** is a *scientificNotationNumeral*, and
   - ***absent*** otherwise
3. Set ***pD***'s *·sign·* to
   - ***absent*** when ***LEX*** is '`NaN`'
   - ***negative*** when the first character of ***LEX*** is '`-`', and
   - ***positive*** otherwise.
4. Return ***pD***.

Lexical Mapping

·decimalLexicalMap· (***LEX***) → decimal

Maps a *decimalLexicalRep* onto a decimal value.

Arguments:

LEX : matches *decimalLexicalRep*

Result:

a decimal value

Algorithm:

Let ***d*** be a decimal value.

1. Set ***d*** to
   - *·noDecimalMap·*(***LEX***) when ***LEX*** is an instance of *noDecimalPtNumeral*, and
   - *·decimalPtMap·*(***LEX***) when ***LEX*** is an instance of *decimalPtNumeral*,

2. Return ***d***.

Canonical Mapping

·decimalCanonicalMap· (***d***) → *decimalLexicalRep*
Maps a decimal to its ·canonical representation·, a *decimalLexicalRep*.

Arguments:

d : a decimal value

Result:

a ·literal· matching *decimalLexicalRep*

Algorithm:

1. If ***d*** is an integer, then return *·noDecimalPtCanonicalMap·*(***d***).
2. Otherwise, return *·decimalPtCanonicalMap·*(***d***).

Auxiliary Functions for Binary Floating-point Lexical/Canonical Mappings

·floatingPointRound· (***nV***, ***cWidth***, ***eMin***, ***eMax***) → decimal number or ·special value·
Rounds a non-zero decimal number to the nearest floating-point value.

Arguments:

nV : an initially non-zero decimal number *(may be set to zero during calculations)*
cWidth : a positive integer
eMin : an integer
eMax : an integer greater than ***eMin***

Result:

a decimal number or ·special value· *(**INF** or −**INF**)*

Algorithm:

Let

- ***s*** be an integer intially 1,
- ***c*** be a nonnegative integer, and
- ***e*** be an integer.

1. Set ***s*** to −1 when $nV < 0$.
2. So select ***e*** that $2^{cWidth} \times 2^{(e-1)} \leq |nV| < 2^{cWidth} \times 2^{e}$.
3. So select ***c*** that $(c - 1) \times 2^{e} \leq |nV| < c \times 2^{e}$.
4.
   - when $eMax < e$ *(overflow)* return:
     - ***positiveInfinity*** when ***s*** is positive, and
     - ***negativeInfinity*** otherwise.

- otherwise:

  a. When $e < eMin$ *(underflow):*

     - Set $e = eMin$
     - So select c that $(c - 1) \times 2^{e} \leq |nV| < c \times 2^{e}$.

  b. Set nV to

     - $c \times 2^{e}$ when $|nV| > c \times 2^{e} - 2^{(e-1)}$;
     - $(c - 1) \times 2^{e}$ when $|nV| < c \times 2^{e} - 2^{(e-1)}$;
     - $c \times 2^{e}$ or $(c - 1) \times 2^{e}$ according to whether c is even or $c - 1$ is even, otherwise (i.e., $|nV| = c \times 2^{e} - 2^{(e-1)}$, the midpoint between the two values).

  c. Return

     - $s \times nV$ when $nV < 2^{cWidth} \times 2^{eMax}$,
     - ***positiveInfinity*** when s is positive, and
     - ***negativeInfinity*** otherwise.

Note: Implementers will find the algorithms of [Clinger, WD (1990)] more efficient in memory than the simple abstract algorithm employed above.

·round· (***n***, ***k***) → decimal number
Maps a decimal number to that value rounded by some power of 10.

Arguments:

n	:	a decimal number
k	:	a nonnegative integer

Result:
a decimal number

Algorithm:
Return $((n / 10^{k} + 0.5) \cdot\text{div}\cdot 1) \times 10^{k}$.

·floatApprox· (***c***, ***e***, ***j***) → decimal number
Maps a decimal number ($c \times 10^{e}$) to successive approximations.

Arguments:

c	:	a nonnegative integer
e	:	an integer
j	:	a nonnegative integer

Result:
a decimal number

Algorithm:

Return ·*round*·(*c*, *j*) × 10^{e}

Lexical Mapping

·*floatLexicalMap*· (***LEX***) → float
Maps a *floatRep* onto a float value.

Arguments:

LEX : matches *floatRep*

Result:

a float value

Algorithm:

Let ***nV*** be a decimal number or ·special value· (INF or −INF).

- Return ·*specialRepValue*·(***LEX***) when ***LEX*** is an instance of *numericalSpecialRep*;
- otherwise (***LEX*** is a numeral):
  1. Set ***nV*** to
     - ·*noDecimalMap*·(***LEX***) when ***LEX*** is an instance of *noDecimalPtNumeral*,
     - ·*decimalPtMap*·(***LEX***) when ***LEX*** is an instance of *decimalPtNumeral*, and
     - ·*scientificMap*·(***LEX***) otherwise (***LEX*** is an instance of *scientificNotationNumeral*).
  2. Set ***nV*** to ·*floatingPointRound*·(***nV***, 24, −149, 104) when ***nV*** is not zero. *(·floatingPointRound· may nonetheless return zero, or INF or −INF.)*
  3. Return:
     - When ***nV*** is zero:
       - ***negativeZero*** when the first character of ***LEX*** is '-', and
       - ***positiveZero*** otherwise.
     - ***nV*** otherwise.

Note: This specification permits the substitution of any other rounding algorithm which conforms to the requirements of [IEEE 754-2008].

Lexical Mapping

·*doubleLexicalMap*· (***LEX***) → double
Maps a *doubleRep* onto a double value.

Arguments:

LEX : matches *doubleRep*

Result:
a double value
Algorithm:
Let *nV* be a decimal number or ·special value· (INF or −INF).

- Return *·specialRepValue·*(*LEX*) when *LEX* is an instance of *numericalSpecialRep*;
- otherwise (*LEX* is a numeral):
  1. Set *nV* to
     - *·noDecimalMap·*(*LEX*) when *LEX* is an instance of *noDecimalPtNumeral*,
     - *·decimalPtMap·*(*LEX*) when *LEX* is an instance of *decimalPtNumeral*, and
     - *·scientificMap·*(*LEX*) otherwise (*LEX* is an instance of *scientificNotationNumeral*).
  2. Set *nV* to *·floatingPointRound·*(*nV*, 53, −1074, 971) when *nV* is not zero. (*·floatingPointRound· may nonetheless return zero, or INF or −INF.)*
  3. Return:
     - When *nV* is zero:
       - ***negativeZero*** when the first character of *LEX* is '-', and
       - ***positiveZero*** otherwise.
     - ***nV*** otherwise.

Note: This specification permits the substitution of any other rounding algorithm which conforms to the requirements of [IEEE 754-2008].

Canonical Mapping

·floatCanonicalMap· (***f***) → *floatRep*
Maps a float to its ·canonical representation·, a *floatRep*.
Arguments:
f : a float value
Result:
a ·literal· matching *floatRep*
Algorithm:
Let

- *l* be a nonnegative integer
- *s* be an integer intially 1,
- *c* be a positive integer, and

- ***e*** be an integer.

- Return [·*specialRepCanonicalMap*·](***f***) when ***f*** is one of ***positiveInfinity***, ***negativeInfinity***, or ***notANumber***;
- return '`0.0E0`' when ***f*** is ***positiveZero***;
- return '`-0.0E0`' when ***f*** is ***negativeZero***;
- otherwise (***f*** is numeric and non-zero):
  1. Set ***s*** to −1 when ***f*** < 0 .
  2. Let ***c*** be the smallest integer for which there exists an integer ***e*** for which $|f| = c \times 10^{e}$.
  3. Let ***e*** be $\log_{10}(|f| / c)$ (so that $|f| = c \times 10^{e}$).
  4. Let ***l*** be the largest nonnegative integer for which $c \times 10^{e}$ = [·*floatingPointRound*·](·*floatApprox*·(***c***, ***e***, ***l***), 24, −149, 104)
  5. Return [·*scientificCanonicalMap*·](***s*** × [·*floatApprox*·](***c***, ***e***, ***l***)) .

Canonical Mapping

·*doubleCanonicalMap*· (***f***) → [*doubleRep*]

Maps a [double] to its ·canonical representation·, a [*doubleRep*].

Arguments:

f : a [double] value

Result:

a ·literal· matching [*doubleRep*]

Algorithm:

Let

- ***l*** be a nonnegative integer
- ***s*** be an integer intially 1,
- ***c*** be a positive integer, and
- ***e*** be an integer.

- Return [·*specialRepCanonicalMap*·](***f***) when ***f*** is one of ***positiveInfinity***, ***negativeInfinity***, or ***notANumber***;
- return '`0.0E0`' when ***f*** is ***positiveZero***;
- return '`-0.0E0`' when ***f*** is ***negativeZero***;

- otherwise (***f*** is numeric and non-zero):
  1. Set ***s*** to −1 when ***f*** < 0 .
  2. Let ***c*** be the smallest integer for which there exists an integer ***e*** for which $|f| = c \times 10^{e}$.
  3. Let ***e*** be $\log_{10}(|f| / c)$ (so that $|f| = c \times 10^{e}$).
  4. Let ***l*** be the largest nonnegative integer for which $c \times 10^{e}$ = *·floatingPointRound·*(*·floatAprox·*(***c***, ***e***, ***l***), 53, −1074, 971)
  5. Return *·scientificCanonicalMap·*(***s*** × *·floatApprox·*(***c***, ***e***, ***l***)) .

Canonical Mapping

·precisionDecimalCanonicalMap· (***pD***) → *pDecimalRep*
Maps a precisionDecimal to its ·canonical representation·, a *pDecimalRep*.

Arguments:

pD : a precisionDecimal value

Result:

a ·literal· matching *pDecimalRep*

Algorithm:

1. Let ***nV*** be the *·numericalValue·* of ***pD***.

   Let ***aP*** be the *·arithmeticPrecision·* of ***pD***.
2. If ***pD*** is one of NaN, INF, or -INF, then return *·specialRepCanonicalMap·*(***nV***).
3. Otherwise, if ***nV*** is an integer and ***aP*** is zero and 1E-6 ≤ ***nV*** ≤ 1E6, then return *·noDecimalPtCanonicalMap·*(***nV***).
4. Otherwise, if ***aP*** is greater than zero and 1E-6 ≤ ***nV*** ≤ 1E6, then let ***s*** be *·decimalPtCanonicalMap·*(***nV***). Let ***f*** be the number of fractional digits in ***s***; ***f*** will invariably be less than or equal to ***aP***. Return the concatenation of ***s*** with ***aP*** − ***f*** occurrences of the digit '`0`'.
5. Otherwise, it will be the case that ***nV*** is less than 1E−6 or greater than 1E6, or that ***aP*** is less than zero. Let
   - ***s*** be *·scientificCanonicalMap·*(***nV***).
   - ***m*** be the part of ***s*** which precedes the "E".
   - ***n*** be the part of ***s*** which follows the "E".
   - ***p*** be the integer denoted by ***n***.
   - ***f*** be the number of fractional digits in ***m***; note that ***f*** will invariably be less than or equal to ***aP*** + ***p***.

- *t* be a string consisting of *aP* + *p* − *f* occurrences of the digit '0', preceded by a decimal point if and only if *m* contains no decimal point and *aP* + *p* − *f* is greater than zero.

Return the concatenation *m* & *t* & 'E' & *n*.

E.2 Duration-related Definitions

The following functions are primarily used with the duration datatype and its derivatives.

Auxiliary duration-related Functions Operating on Representation Fragments

·duYearFragmentMap· (***Y***) → integer
Maps a *duYearFrag* to an integer, intended as part of the value of the *·months·* property of a duration value.

Arguments:
Y : matches *duYearFrag*

Result:
a nonnegative integer

Algorithm:
Y is necessarily the letter 'Y' followed by a numeral ***N***:
Return *·noDecimalMap·*(***N***).

·duMonthFragmentMap· (***M***) → integer
Maps a *duMonthFrag* to an integer, intended as part of the value of the *·months·* property of a duration value.

Arguments:
M : matches *duYearFrag*

Result:
a nonnegative integer

Algorithm:
M is necessarily the letter 'M' followed by a numeral ***N***:
Return *·noDecimalMap·*(***N***).

·duDayFragmentMap· (***D***) → integer
Maps a *duDayFrag* to an integer, intended as part of the value of the *·seconds·* property of a duration value.

Arguments:
D : matches *duDayFrag*

Result:
a nonnegative integer

Algorithm:
D is necessarily the letter 'D' followed by a numeral ***N***:
Return *·noDecimalMap·*(***N***).

·duHourFragmentMap· (***H***) → integer
Maps a *duHourFrag* to an integer, intended as part of the value of the *·seconds·* property

of a duration value.

Arguments:

H : matches *duHourFrag*

Result:

a nonnegative integer

Algorithm:

D is necessarily the letter 'D' followed by a numeral ***N***:
Return *·noDecimalMap·*(***N***).

·duMinuteFragmentMap· (***M***) → integer

Maps a *duMinuteFrag* to an integer, intended as part of the value of the *·seconds·* property of a duration value.

Arguments:

M : matches *duMinuteFrag*

Result:

a nonnegative integer

Algorithm:

M is necessarily the letter 'M' followed by a numeral ***N***:
Return *·noDecimalMap·*(***N***).

·duSecondFragmentMap· (***S***) → decimal number

Maps a *duSecondFrag* to a decimal number, intended as part of the value of the *·seconds·* property of a duration value.

Arguments:

S : matches *duSecondFrag*

Result:

a nonnegative decimal number

Algorithm:

S is necessarily 'S' followed by a numeral ***N***:
Return

- *·decimalPtMap·*(***N***) when '.' occurs in ***N***, and
- *·noDecimalMap·*(***N***) otherwise.

·duYearMonthFragmentMap· (***YM***) → integer

Maps a *duYearMonthFrag* into an integer, intended as part of the *·months·* property of a duration value.

Arguments:

YM : matches *duYearMonthFrag*

Result:

a nonnegative integer

Algorithm:

YM necessarily consists of an instance ***Y*** of *duYearFrag* and/or an instance ***M*** of *duMonthFrag*:

Let

- ***y*** be *·duYearFragmentMap·*(***Y***) (or 0 if ***Y*** is not present) and

- ***m*** be *·duMonthFragmentMap·*(***M***) (or 0 if ***M*** is not present).

Return 12 × ***y*** + ***m*** .

·duTimeFragmentMap· (***T***) → decimal number
Maps a *duTimeFrag* into a decimal number, intended as part of the *·seconds·* property of a duration value.

Arguments:
T : matches *duTimeFrag*

Result:
a nonnegative decimal number

Algorithm:
T necessarily consists of an instance ***H*** of *duHourFrag*, and/or an instance ***M*** of *duMinuteFrag*, and/or an instance ***S*** of *duSecondFrag*.
Let

- ***h*** be *·duDayFragmentMap·*(***H***) (or 0 if ***H*** is not present),
- ***m*** be *·duMinuteFragmentMap·*(***M***) (or 0 if ***M*** is not present), and
- ***s*** be *·duSecondFragmentMap·*(***S***) (or 0 if ***S*** is not present).

Return 3600 × ***h*** + 60 × ***m*** + s .

·duDayTimeFragmentMap· (***DT***) → decimal number
Maps a *duDayTimeFrag* into a decimal number, which is the potential value of the *·seconds·* property of a duration value.

Arguments:
DT : matches *duDayTimeFrag*

Result:
a nonnegative decimal number

Algorithm:
DT necesarily consists of an instance ***D*** of *duDayFrag* and/or an instance ***T*** of *duTimeFrag*.
Let

- ***d*** be *·duDayFragmentMap·*(***D***) (or 0 if ***D*** is not present) and
- ***t*** be *·duTimeFragmentMap·*(***T***) (or 0 if ***T*** is not present).

Return 86400 × ***d*** + ***t*** .

The duration Lexical Mapping

·durationMap· (***DUR***) → duration
Separates the *durationLexicalRep* into the month part and the seconds part, then maps them into the *·months·* and *·seconds·* of the duration value.

Arguments:
DUR : matches *durationLexicalRep*

Result:
a complete duration value

Algorithm:

DUR consists of possibly a leading '-', followed by 'P' and then an instance ***Y*** of *duYearMonthFrag* and/or an instance ***D*** of *duDayTimeFrag*:
Return a duration whose

- *·months·* value is
  - 0 if ***Y*** is not present,
  - −*·duYearMonthFragmentMap·*(***Y***) if both '-' and ***Y*** are present, and
  - *·duYearMonthFragmentMap·*(***Y***) otherwise.

and whose

- *·seconds·* value is
  - 0 if ***D*** is not present,
  - −*·duDayTimeFragmentMap·*(***D***) if both '-' and ***D*** are present, and
  - *·duDayTimeFragmentMap·*(***D***) otherwise.

The yearMonthDuration Lexical Mapping

·yearMonthDurationMap· (***YM***) → yearMonthDuration
Maps the lexical representation into the *·months·* of a yearMonthDuration value. (A yearMonthDuration's *·seconds·* is always zero.) *·yearMonthDurationMap·* is a restriction of *·durationMap·*.

Arguments:

YM : matches *yearMonthDurationLexicalRep*

Result:

a complete yearMonthDuration value

Algorithm:

YM necessarily consists of an optional leading '-', followed by 'P' and then an instance ***Y*** of *duYearMonthFrag*:
Return a yearMonthDuration whose

- *·months·* value is
  - −*·duYearMonthFragmentMap·*(***Y***) if '-' is present in ***YM*** and
  - *·duYearMonthFragmentMap·*(***Y***) otherwise, and
- *·seconds·* value is (necessarily) 0.

The dayTimeDuration Lexical Mapping

·dayTimeDurationMap· (***DT***) → dayTimeDuration
Maps the lexical representation into the *·seconds·* of a dayTimeDuration value. (A

dayTimeDuration's *·months·* is always zero.) *·dayTimeDurationMap·* is a restriction of *·durationMap·*.

Arguments:

DT : a dayTimeDuration value

Result:

a complete dayTimeDuration value

Algorithm:

DT necessarily consists of possibly a leading '-', followed by 'P' and then an instance ***D*** of *duDayTimeFrag*:

Return a dayTimeDuration whose

- *·months·* value is (necessarily) 0, and
- *·seconds·* value is
  - −*·duDayTimeFragmentMap·*(***D***) if '-' is present in ***DT*** and
  - *·duDayTimeFragmentMap·*(***D***) otherwise.

Auxiliary duration-related Functions Producing Representation Fragments

·duYearMonthCanonicalFragmentMap· (***ym***) → *duYearMonthFrag*

Maps a nonnegative integer, presumably the absolute value of the *·months·* of a duration value, to a *duYearMonthFrag*, a fragment of a duration ·lexical representation·.

Arguments:

ym : a nonnegative integer

Result:

a ·literal· matching *duYearMonthFrag*

Algorithm:

Let

- ***y*** be ***ym*** ·div· 12 , and
- ***m*** be ***ym*** ·mod· 12 ,

Return

- *·unsignedNoDecimalPtCanonicalMap·*(***y***) & 'Y' & *·unsignedNoDecimalPtCanonicalMap·*(***m***) & 'M' when neither ***y*** nor ***m*** is zero,
- *·unsignedNoDecimalPtCanonicalMap·*(***y***) & 'Y' when ***y*** is not zero but ***m*** is, and
- *·unsignedNoDecimalPtCanonicalMap·*(***m***) & 'M' when ***y*** is zero.

·duDayCanonicalFragmentMap· (***d***) → *duDayFrag*

Maps a nonnegative integer, presumably the day normalized value from the *·seconds·* of a duration value, to a *duDayFrag*, a fragment of a duration ·lexical representation·.

Arguments:

d : a nonnegative integer

Result:

a ·literal· matching *duDayFrag*

Algorithm:

Return

- *·unsignedNoDecimalPtCanonicalMap·*(***d***) & 'D' when ***d*** is not zero, and
- the empty string ('') when ***d*** is zero.

·duHourCanonicalFragmentMap· (***h***) → *duHourFrag*

Maps a nonnegative integer, presumably the hour normalized value from the *·seconds·* of a duration value, to a *duHourFrag*, a fragment of a duration ·lexical representation·.

Arguments:

h : a nonnegative integer

Result:

a ·literal· matching *duHourFrag*

Algorithm:

Return

- *·unsignedNoDecimalPtCanonicalMap·*(***h***) & 'H' when ***h*** is not zero, and
- the empty string ('') when ***h*** is zero.

·duMinuteCanonicalFragmentMap· (***m***) → *duMinuteFrag*

Maps a nonnegative integer, presumably the minute normalized value from the *·seconds·* of a duration value, to a *duMinuteFrag*, a fragment of a duration ·lexical representation·.

Arguments:

m : a nonnegative integer

Result:

a ·literal· matching *duMinuteFrag*

Algorithm:

Return

- *·unsignedNoDecimalPtCanonicalMap·*(***m***) & 'M' when ***m*** is not zero, and
- the empty string ('') when ***m*** is zero.

·duSecondCanonicalFragmentMap· (***s***) → *duSecondFrag*

Maps a nonnegative decimal number, presumably the second normalized value from the *·seconds·* of a duration value, to a *duSecondFrag*, a fragment of a duration ·lexical representation·.

Arguments:

s : a nonnegative decimal number

Result:

matches *duSecondFrag*

Algorithm:

Return

- *·unsignedNoDecimalPtCanonicalMap·*(***s***) & 'S' when ***s*** is a non-zero integer,

- ·unsignedDecimalPtCanonicalMap·(***s***) & 'S' when ***s*** is not an integer, and
- the empty string ('') when ***s*** is zero.

·duTimeCanonicalFragmentMap· (***h***, ***m***, ***s***) → *duTimeFrag*
Maps three nonnegative numbers, presumably the hour, minute, and second normalized values from a duration's *·seconds·*, to a *duTimeFrag*, a fragment of a duration ·lexical representation·.

Arguments:

h	:	a nonnegative integer
m	:	a nonnegative integer
s	:	a nonnegative decimal number

Result:
a ·literal· matching *duTimeFrag*

Algorithm:
Return

- 'T' & *·duHourCanonicalFragmentMap·*(***h***) & *·duMinuteCanonicalFragmentMap·*(***m***) & *·duSecondCanonicalFragmentMap·*(***s***) when ***h***, ***m***, and ***s*** are not all zero, and
- the empty string ('') when all arguments are zero.

·duDayTimeCanonicalFragmentMap· (***ss***) → *duDayTimeFrag*
Maps a nonnegative decimal number, presumably the absolute value of the *·seconds·* of a duration value, to a *duDayTimeFrag*, a fragment of a duration ·lexical representation·.

Arguments:

ss	:	a nonnegative decimal number

Result:
matches *duDayTimeFrag*

Algorithm:
Let

- ***d*** is ***ss*** ·div· 86400 ,
- ***h*** is (***ss*** ·mod· 86400) ·div· 3600 ,
- ***m*** is (***ss*** ·mod· 3600) ·div· 60 , and
- ***s*** is ***ss*** ·mod· 60 ,

Return

- *·duDayCanonicalFragmentMap·*(***d***) & *·duTimeCanonicalFragmentMap·*(***h***, ***m***, ***s***) when ***ss*** is not zero and
- 'T0S' when ***ss*** is zero.

The duration Canonical Mapping

·durationCanonicalMap· (***v***) → *durationLexicalRep*

Maps a duration's property values to *durationLexicalRep* fragments and combines the fragments into a complete *durationLexicalRep*.

Arguments:

v : a complete duration value

Result:

matches *durationLexicalRep*

Algorithm:

Let

- ***m*** be ***v***'s *·months·*,
- ***s*** be ***v***'s *·seconds·*, and
- ***sgn*** be '-' if ***m*** or ***s*** is negative and the empty string ('') otherwise.

Return

- ***sgn*** & 'P' & *·duYearMonthCanonicalFragmentMap·*(| ***m*** |) & *·duDayTimeCanonicalFragmentMap·*(| ***s*** |) when neither ***m*** nor ***s*** is zero,
- ***sgn*** & 'P' & *·duYearMonthCanonicalFragmentMap·*(| ***m*** |) when ***m*** is not zero but ***s*** is, and
- ***sgn*** & 'P' & *·duDayTimeCanonicalFragmentMap·*(| ***s*** |) when ***m*** is zero.

The yearMonthDuration Canonical Mapping

·yearMonthDurationCanonicalMap· (***ym***) → *yearMonthDurationLexicalRep*

Maps a yearMonthDuration's *·months·* value to a *yearMonthDurationLexicalRep*. (The *·seconds·* value is necessarily zero and is ignored.) *·yearMonthDurationCanonicalMap·* is a restriction of *·durationCanonicalMap·*.

Arguments:

ym : a complete yearMonthDuration value

Result:

matches *yearMonthDurationLexicalRep*

Algorithm:

Let

- ***m*** be ***ym***'s *·months·* and
- ***sgn*** be '-' if ***m*** is negative and the empty string ('') otherwise.

Return ***sgn*** & 'P' & *·duYearMonthCanonicalFragmentMap·*(| ***m*** |) .

The dayTimeDuration Canonical Mapping

·dayTimeDurationCanonicalMap· (***dt***) → *dayTimeDurationLexicalRep*

Maps a dayTimeDuration's *·seconds·* value to a *dayTimeDurationLexicalRep*. (The *·months·* value is necessarily zero and is ignored.) *·dayTimeDurationCanonicalMap·* is a restriction of *·durationCanonicalMap·*.

Arguments:

dt : a complete dayTimeDuration value

Result:

matches *dayTimeDurationLexicalRep*

Algorithm:

Let

- ***s*** be ***dt***'s *·months·* and
- ***sgn*** be '-' if ***s*** is negative and the empty string ('') otherwise.

Return ***sgn*** & 'P' & *·duYearMonthCanonicalFragmentMap·*(| ***s*** |) .

E.3 Date/time-related Definitions

E.3.1 Normalization of property values

When adding and subtracting numbers from date/time properties, the immediate results may not conform to the limits specified. Accordingly, the following procedures are used to "normalize" potential property values to corresponding values that do conform to the appropriate limits. Normalization is required when dealing with time zone offset changes (as when converting to ·UTC· from "local" values) and when adding duration values to or subtracting them from dateTime values.

Date/time Datatype Normalizing Procedures

·normalizeMonth· (***yr***, ***mo***)

If month (***mo***) is out of range, adjust month and year (***yr***) accordingly; otherwise, make no change.

Arguments:

yr : an integer

mo : an integer

Algorithm:

1. Add (***mo*** − 1) ·div· 12 to ***yr***.
2. Set ***mo*** to (***mo*** − 1) ·mod· 12 + 1 .

·normalizeDay· (***yr***, ***mo***, ***da***)

If month (***mo***) is out of range, or day (***da***) is out of range for the appropriate month, then adjust values accordingly, otherwise make no change.

Arguments:

yr : an integer

mo : an integer

da	:	an integer

Algorithm:

1. *·normalizeMonth·*(***yr***, ***mo***)
2. Repeat until ***da*** is positive and not greater than *·daysInMonth·*(***yr***, ***mo***):
   a. If ***da*** exceeds the upper limit from the table then:
      i. Subtract that limit from ***da***.
      ii. Add 1 to ***mo***.
      iii. *·normalizeMonth·*(***yr***, ***mo***)
   b. If ***da*** is not positive then:
      i. Subtract 1 from ***mo***.
      ii. *·normalizeMonth·*(***yr***, ***mo***)
      iii. Add the new upper limit from the table to ***da***.

·normalizeMinute· (***yr***, ***mo***, ***da***, ***hr***, ***mi***)
Normalizes minute, hour, month, and year values to values that obey the appropriate constraints.

Arguments:

yr	:	an integer
mo	:	an integer
da	:	an integer
hr	:	an integer
mi	:	an integer

Algorithm:

1. Add ***mi*** ·div· 60 to ***hr***.
2. Set ***mi*** to ***mi*** ·mod· 60 .
3. Add ***hr*** ·div· 24 to ***da***.
4. Set ***hr*** to ***hr*** ·mod· 24 .
5. *·normalizeDay·*(***yr***, ***mo***, ***da***).

·normalizeSecond· (***yr***, ***mo***, ***da***, ***hr***, ***mi***, ***se***)
Normalizes second, minute, hour, month, and year values to values that obey the appropriate constraints. (This algorithm ignores leap seconds.)

Arguments:

yr	:	an integer
mo	:	an integer

da	:	an integer
hr	:	an integer
mi	:	an integer
se	:	a decimal number

Algorithm:

1. Add ***se*** ·div· 60 to ***mi***.
2. Set ***se*** to ***se*** ·mod· 60 .
3. *·normalizeMinute·*(***yr***, ***mo***, ***da***, ***hr***, ***mi***).

E.3.2 Auxiliary Functions

Date/time Auxiliary Functions

·daysInMonth· (***y***, ***m***) → integer
Returns the number of the last day of the month for any combination of year and month.

Arguments:

y	:	an ·optional· integer
m	:	an integer between 1 and 12

Result:
between 28 and 31 inclusive

Algorithm:
Return:

- 28 when ***m*** is 2 and ***y*** is not evenly divisible by 4, or is evenly divisible by 100 but not by 400, or is ***absent***,
- 29 when ***m*** is 2 and ***y*** is evenly divisible by 400, or is evenly divisible by 4 but not by 100,
- 30 when ***m*** is 4, 6, 9, or 11,
- 31 otherwise (***m*** is 1, 3, 5, 7, 8, 10, or 12)

·newDateTime· (***Yr***, ***Mo***, ***Da***, ***Hr***, ***Mi***, ***Se***, ***Tz***) → an instance of the date/timeSevenPropertyModel
Returns an instance of the date/timeSevenPropertyModel with property values as specified in the arguments. If an argument is omitted, the corresponding property is set to ***absent***.

Arguments:

Yr	:	an ·optional· integer
Mo	:	an ·optional· integer between 1 and 12 inclusive
Da	:	an ·optional· integer between 1 and 31 inclusive
Hr	:	an ·optional· integer between 0 and 24 inclusive

Mi : an ·optional· integer between 0 and 59 inclusive
Se : an ·optional· decimal number greater than or equal to 0 and less than 60
Tz : an ·optional· integer between −840 and 840 inclusive.

Result:

Algorithm:

Let

- ***dt*** be an instance of the date/timeSevenPropertyModel
- ***yr*** be ***Yr*** when ***Yr*** is not ***absent***, otherwise 1
- ***mo*** be ***Mo***when ***Mo*** is not ***absent***, otherwise 1
- ***da*** be ***Da***when ***Da*** is not ***absent***, otherwise 1
- ***hr*** be ***Hr***when ***Hr*** is not ***absent***, otherwise 0
- ***mi*** be ***Mi***when ***Mi*** is not ***absent***, otherwise 0
- ***se*** be ***Se***when ***Se*** is not ***absent***, otherwise 0

1. *·normalizeSecond·*(***yr***, ***mo***, ***da***, ***hr***, ***mi***, ***se***)
2. Set the *·year·* property of ***dt*** to ***absent*** when ***Yr*** is ***absent***, otherwise ***yr***.
3. Set the *·month·* property of ***dt*** to ***absent*** when ***Mo*** is ***absent***, otherwise ***mo***.
4. Set the *·day·* property of ***dt*** to ***absent*** when ***Da*** is ***absent***, otherwise ***da***.
5. Set the *·hour·* property of ***dt*** to ***absent*** when ***Hr*** is ***absent***, otherwise ***hr***.
6. Set the *·minute·* property of ***dt*** to ***absent*** when ***Mi*** is ***absent***, otherwise ***mi***.
7. Set the *·second·* property of ***dt*** to ***absent*** when ***Se*** is ***absent***, otherwise ***se***.
8. Set the *·timezoneOffset·* property of ***dt*** to ***Tz***
9. Return ***dt***.

E.3.3 Adding durations to dateTimes

Given a dateTime ***S*** and a duration ***D***, function *·dateTimePlusDuration·* specifies how to compute a dateTime ***E***, where ***E*** is the end of the time period with start ***S*** and duration ***D*** i.e. ***E*** = ***S*** + ***D***. Such computations are used, for example, to determine whether a dateTime is within a specific time period. This algorithm can also be applied, when applications need the operation, to the addition of durations to the datatypes date, gYearMonth, gYear, gDay and gMonth, each of which can be viewed as denoting a set of dateTimes. In such cases, the addition is made to the first or starting dateTime in the set. Note that the extension of this algorithm to types other than dateTime is not needed for schema-validity assessment.

Essentially, this calculation adds the *·months·* and *·seconds·* properties of the duration value separately to the dateTime value. The *·months·* value is added to the starting dateTime value first. If the day is out of range for the new month value, it is *pinned* to be within range. Thus April 31 turns into April 30. Then the *·seconds·* value is added. This latter addition can cause the year, month, day, hour, and minute to change.

Leap seconds are ignored by the computation. All calculations use 60 seconds per minute.

Thus the addition of either PT1M or PT60S to any dateTime will always produce the same result. This is a special definition of addition which is designed to match common practice, and—most importantly—be stable over time.

A definition that attempted to take leap-seconds into account would need to be constantly updated, and could not predict the results of future implementation's additions. The decision to introduce a leap second in ·UTC· is the responsibility of the [International Earth Rotation Service (IERS)]. They make periodic announcements as to when leap seconds are to be added, but this is not known more than a year in advance. For more information on leap seconds, see [U.S. Naval Observatory Time Service Department].

Adding duration to dateTime

·dateTimePlusDuration· (***du***, ***dt***) → dateTime
Adds a duration to a dateTime value, producing another dateTime value.

Arguments:

du	:	a duration value
dt	:	a dateTime value

Result:
a dateTime value

Algorithm:
Let

- ***yr*** be ***dt***'s *·year·*,
- ***mo*** be ***dt***'s *·month·*,
- ***da*** be ***dt***'s *·day·*,
- ***hr*** be ***dt***'s *·hour·*,
- ***mi*** be ***dt***'s *·minute·*, and
- ***se*** be ***dt***'s *·second·*.
- ***tz*** be ***dt***'s *·timezoneOffset·*.

1. Add ***du***'s *·months·* to ***mo***.
2. *·normalizeMonth·*(***yr***, ***mo***). (I.e., carry any over- or underflow, adjust month.)
3. Set ***da*** to min(***da***, *·daysInMonth·*(***yr***, ***mo***)). (I.e., *pin* the value if necessary.)
4. Add ***du***'s *·seconds·* to ***se***.

5. *·normalizeSecond·*(***yr***, ***mo***, ***da***, ***hr***, ***mi***, ***se***). (I.e., carry over- or underflow of seconds up to minutes, hours, etc.)
6. Return *·newDateTime·*(***yr***, ***mo***, ***da***, ***hr***, ***mi***, ***se***, ***tz***)

This algorithm may be applied to date/time types other than dateTime, by

1. For each ***absent*** property, supply the minimum legal value for that property (1 for years, months, days, 0 for hours, minutes, seconds).
2. Call the function.
3. For each property ***absent*** in the initial value, set the corresponding property in the result value to ***absent***.

Examples:

dateTime	duration	result
2000-01-12T12:13:14Z	P1Y3M5DT7H10M3.3S	2001-04-17T19:23:17.3Z
2000-01	-P3M	1999-10
2000-01-12	PT33H	2000-01-13

Note that the addition defined by *·dateTimePlusDuration·* differs from addition on integers or real numbers in not being commutative. The order of addition of durations to instants *is* significant. For example, there are cases where:

> ((dateTime + duration1) + duration2) != ((dateTime + duration2) + duration1)

Example:

- (2000-03-30 + P1D) + P1M = 2000-03-31 + P1M = 2000-**04-30**
- (2000-03-30 + P1M) + P1D = 2000-04-30 + P1D = 2000-**05-01**

E.3.4 Time on timeline

Time on Timeline for Date/time Seven-property Model Datatypes

·timeOnTimeline· (***dt***) → decimal number
Maps a date/timeSevenPropertyModel value to the decimal number representing its position on the "time line".

Arguments:

dt : a date/timeSevenPropertyModel value

Result:

a decimal number

Algorithm:

Let

- ***yr*** be 1971 when ***dt***'s *·year·* is ***absent***, and ***dt***'s *·year·* − 1 otherwise,
- ***mo*** be 12 or ***dt***'s *·month·* , similarly,
- ***da*** be *·daysInMonth·*(***yr***+1, ***mo***) − 1 or (***dt***'s *·day·*) − 1 , similarly,
- ***hr*** be 0 or ***dt***'s *·hour·* , similarly, and
- ***mi*** be 0 or ***dt***'s *·minute·* , similarly.

1. Subtract *·timezoneOffset·* from ***mi*** when *·timezoneOffset·* is not ***absent***.
2. (*·year·*)
   a. Set ***ToTI*** to 31536000 × ***yr*** .
3. (Leap-year Days, *·month·*, and *·day·*)
   a. Add 86400 × (***yr*** ·div· 400 − ***yr*** ·div· 100 + ***yr*** ·div· 4) to ***ToTI***.
   b. Add 86400 × $\text{Sum}_{m < mo}$ *·daysInMonth·*(***yr*** + 1, ***m***) to ***ToTI***
   c. Add 86400 × ***da*** to ***ToTI***.
4. (*·hour·*, *·minute·*, and *·second·*)
   a. Add 3600 × ***hr*** + 60 × ***mi*** + ***se*** to ***ToTI***.
5. Return ***ToTI***.

E.3.5 Lexical mappings

Partial Date/time Lexical Mappings

·yearFragValue· (***YR***) → integer
Maps a *yearFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·year·* property of a date/timeSevenPropertyModel value.

Arguments:
YR : matches *yearFrag*
Result:
an integer
Algorithm:
Return *·noDecimalMap·*(***YR***)

·monthFragValue· (***MO***) → integer
Maps a *monthFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·month·* property of a date/timeSevenPropertyModel value.

Arguments:
MO : matches *monthFrag*
Result:
an integer
Algorithm:
Return *·unsignedNoDecimalMap·*(***MO***)

·dayFragValue· (***DA***) → integer
Maps a *dayFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto

an integer, presumably the *·day·* property of a date/timeSevenPropertyModel value.

Arguments:

DA : matches *dayFrag*

Result:

an integer

Algorithm:

Return *·unsignedNoDecimalMap·*(***DA***)

·hourFragValue· (***HR***) → integer

Maps a *hourFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·hour·* property of a date/timeSevenPropertyModel value.

Arguments:

HR : matches *hourFrag*

Result:

an integer

Algorithm:

Return *·unsignedNoDecimalMap·*(***HR***)

·minuteFragValue· (***MI***) → integer

Maps a *minuteFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·minute·* property of a date/timeSevenPropertyModel value.

Arguments:

MI : matches *minuteFrag*

Result:

an integer

Algorithm:

Return *·unsignedNoDecimalMap·*(***MI***)

·secondFragValue· (***SE***) → decimal number

Maps a *secondFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto a decimal number, presumably the *·second·* property of a date/timeSevenPropertyModel value.

Arguments:

SE : matches *secondFrag*

Result:

a decimal number

Algorithm:

Return

- *·unsignedNoDecimalMap·*(***SE***) when no decimal point occurs in ***SE***, and
- *·unsignedDecimalPtMap·*(***SE***) otherwise.

·timezoneFragValue· (***TZ***) → integer

Maps a *timezoneFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·, onto an integer, presumably the *·timezoneOffset·* property of a date/timeSevenPropertyModel value.

Arguments:

TZ : matches *timezoneFrag*

Result:

an integer

Algorithm:

TZ necessarily consists of either just 'Z', or a sign ('+' or '-') followed by an instance ***H*** of *hourFrag*, a colon, and an instance ***M*** of *minuteFrag*

Return

- 0 when ***TZ*** is 'Z',
- −(*·unsignedDecimalPtMap·*(***H***) × 60 + *·unsignedDecimalPtMap·*(***M***)) when the sign is '-', and
- *·unsignedDecimalPtMap·*(***H***) × 60 + *·unsignedDecimalPtMap·*(***M***) otherwise.

Lexical Mapping

·dateTimeLexicalMap· (***LEX***) → dateTime

Maps a *dateTimeLexicalRep* to a dateTime value.

Arguments:

LEX : matches *dateTimeLexicalRep*

Result:

a complete dateTime value

Algorithm:

LEX necessarily includes an instance ***Y*** of *yearFrag*, an instance ***MO*** of *monthFrag*, and an instance ***D*** of *dayFrag* hyphen-separated, an instance ***H*** of *hourFrag*, an instance ***MI*** of *minuteFrag*, and an instance ***S*** of *secondFrag*, colon-separated and optionally followed by an instance ***T*** of *timezoneFrag*.

Let ***tz*** be *·timezoneFragValue·*(***T***) when ***T*** is present, otherwise ***absent***.

1. Return *·newDateTime·*(*·yearFragValue·*(***Y***), *·monthFragValue·*(***MO***), *·dayFragValue·*(***D***), *·hourFragValue·*(***H***), *·minuteFragValue·*(***MI***), *·secondFragValue·*(***S***), ***tz***)

Lexical Mapping

·timeLexicalMap· (***LEX***) → time

Maps a *timeLexicalRep* to a time value.

Arguments:

LEX : matches *timeLexicalRep*

Result:

a complete time value

Algorithm:

LEX necessarily includes an instance ***H*** of *hourFrag*, an instance ***M*** of *minuteFrag*, and an instance ***S*** of *secondFrag*, colon-separated and optionally followed by an instance ***T*** of *timezoneFrag*.

Let ***tz*** be *[·timezoneFragValue·](#)*(***T***) when ***T*** is present, otherwise ***absent***

1. Return *·newDateTime·*(***absent***, ***absent***, ***absent***, *·hourFragValue·*(***H***), *·minuteFragValue·*(***M***), *·secondFragValue·*(***S***), ***tz***).

Lexical Mapping

·dateLexicalMap· (***LEX***) → date
Maps a *dateLexicalRep* to a date value.

Arguments:

LEX : matches *dateLexicalRep*

Result:

a complete date value

Algorithm:

LEX necessarily includes an instance ***Y*** of *yearFrag*, an instance ***M*** of *monthFrag*, and an instance ***D*** of *dayFrag*, hyphen-separated and optionally followed by an instance ***T*** of *timezoneFrag*.

Let ***tz*** be *·timezoneFragValue·*(***T***) when ***T*** is present, otherwise ***absent***

1. Return *·newDateTime·*(*·yearFragValue·*(***Y***), *·monthFragValue·*(***M***), *·dayFragValue·*(***D***), ***absent***, ***absent***, ***absent***, ***tz***).

Lexical Mapping

·gYearMonthLexicalMap· (***LEX***) → gYearMonth
Maps a *gYearMonthLexicalRep* to a gYearMonth value.

Arguments:

LEX : matches *gYearMonthLexicalRep*

Result:

a complete gYearMonth value

Algorithm:

LEX necessarily includes an instance ***Y*** of *yearFrag* and an instance ***M*** of *monthFrag*, hyphen-separated and optionally followed by an instance ***T*** of *timezoneFrag*.

Let ***tz*** be *·timezoneFragValue·*(***T***) when ***T*** is present, otherwise ***absent***.

1. Return *·newDateTime·*(*·yearFragValue·*(***Y***), *·monthFragValue·*(***M***), ***absent***, ***absent***, ***absent***, ***absent***, ***tz***)

Lexical Mapping

·gYearLexicalMap· (***LEX***) → gYear
Maps a *gYearLexicalRep* to a gYear value.

Arguments:

LEX : matches *gYearLexicalRep*

Result:

a complete gYear value

Algorithm:

LEX necessarily includes an instance ***Y*** of *dayFrag*, optionally followed by an instance ***T*** of *timezoneFrag*.

Let ***tz*** be *·timezoneFragValue·*(***T***) when ***T*** is present, otherwise ***absent***.

1. Return *·newDateTime·*(*·dayFragValue·*(***Y***), ***absent***, ***absent***, ***absent***, ***absent***, ***absent***, ***tz***).

Lexical Mapping

·gMonthDayLexicalMap· (***LEX***) → gMonthDay
Maps a *gMonthDayLexicalRep* to a gMonthDay value.

Arguments:

LEX : matches *gMonthDayLexicalRep*

Result:

a complete gMonthDay value

Algorithm:

LEX necessarily includes an instance ***M*** of *monthFrag* and an instance ***D*** of *dayFrag*, hyphen-separated and optionally followed by an instance ***T*** of *timezoneFrag*.

Let ***tz*** be *·timezoneFragValue·*(***T***) when ***T*** is present, otherwise ***absent***.

1. Return *·newDateTime·*(***gMD***, ***absent***, *·monthFragValue·*(***Y***), *·dayFragValue·*(***M***), ***absent***, ***absent***, ***absent***, ***tz***)

Lexical Mapping

·gDayLexicalMap· (***LEX***) → gDay
Maps a *gDayLexicalRep* to a gDay value.

Arguments:

LEX : matches *gDayLexicalRep*

Result:

a complete gDay value

Algorithm:

LEX necessarily includes an instance ***D*** of *dayFrag*, optionally followed by an instance ***T*** of *timezoneFrag*.

Let ***tz*** be *·timezoneFragValue·*(***T***) when ***T*** is present, otherwise ***absent***.

1. Return *·newDateTime·*(***gD***, ***absent***, ***absent***, *·dayFragValue·*(***D***), ***absent***, ***absent***, ***absent***, ***tz***).

Lexical Mapping

·gMonthLexicalMap· (***LEX***) → gMonth
Maps a *gMonthLexicalRep* to a gMonth value.

Arguments:

LEX : matches *gMonthLexicalRep*

Result:
a complete gMonth value
Algorithm:
LEX necessarily includes an instance ***M*** of *monthFrag*, optionally followed by an instance ***T*** of *timezoneFrag*.

Let ***tz*** be *·timezoneFragValue·*(***T***) when ***T*** is present, otherwise ***absent***.

1. Return *·newDateTime·*(***gM***, ***absent***, *·monthFragValue·*(***M***), ***absent***, ***absent***, ***absent***, ***absent***, ***tz***)

E.3.6 Canonical Mappings

Auxiliary Functions for Date/time Canonical Mappings

·unsTwoDigitCanonicalFragmentMap· (***i***) → *unsignedNoDecimalPtNumeral*
Maps a nonnegative integer less than 100 onto an unsigned always-two-digit numeral.
Arguments:
i : a nonnegative integer less than 100
Result:
matches *unsignedNoDecimalPtNumeral*
Algorithm:
Return *·digit·*(***i*** ·div· 10) & *·digit·*(***i*** ·mod· 10)

·fourDigitCanonicalFragmentMap· (***i***) → *noDecimalPtNumeral*
Maps an integer between -10000 and 10000 onto an always-four-digit numeral.
Arguments:
i : an integer whose absolute value is less than 10000
Result:
matches *noDecimalPtNumeral*
Algorithm:
Return

- '-' & *·unsTwoDigitCanonicalFragmentMap·*(−***i*** ·div· 100) & *·unsTwoDigitCanonicalFragmentMap·*(−***i*** ·mod· 100) when ***i*** is negative,
- *·unsTwoDigitCanonicalFragmentMap·*(***i*** ·div· 100) & *·unsTwoDigitCanonicalFragmentMap·*(***i*** ·mod· 100) otherwise.

Partial Date/time Canonical Mappings

·yearCanonicalFragmentMap· (***y***) → *yearFrag*
Maps an integer, presumably the *·year·* property of a date/timeSevenPropertyModel value, onto a *yearFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.
Arguments:
y : an integer
Result:

matches *yearFrag*

Algorithm:

Return

- *·noDecimalPtCanonicalMap·*(***y***) when |***y***| > 9999 .
- *·fourDigitCanonicalFragmentMap·*(***y***) otherwise.

·monthCanonicalFragmentMap· (***m***) → *monthFrag*

Maps an integer, presumably the *·month·* property of a date/timeSevenPropertyModel value, onto a *monthFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

Arguments:

m : an integer between 1 and 12 inclusive

Result:

matches *monthFrag*

Algorithm:

Return *·unsTwoDigitCanonicalFragmentMap·*(***m***)

·dayCanonicalFragmentMap· (***d***) → *dayFrag*

Maps an integer, presumably the *·day·* property of a date/timeSevenPropertyModel value, onto a *dayFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

Arguments:

d : an integer between 1 and 31 inclusive (may be limited further depending on associated *·year·* and *·month·*)

Result:

matches *dayFrag*

Algorithm:

Return *·unsTwoDigitCanonicalFragmentMap·*(***d***)

·hourCanonicalFragmentMap· (***h***) → *hourFrag*

Maps an integer, presumably the *·hour·* property of a date/timeSevenPropertyModel value, onto a *hourFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

Arguments:

h : an integer between 0 and 23 inclusive.

Result:

matches *hourFrag*

Algorithm:

Return *·unsTwoDigitCanonicalFragmentMap·*(***h***)

·minuteCanonicalFragmentMap· (***m***) → *minuteFrag*

Maps an integer, presumably the *·minute·* property of a date/timeSevenPropertyModel value, onto a *minuteFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

Arguments:

m : an integer between 0 and 59 inclusive.

Result:

matches *minuteFrag*
Algorithm:
Return *·unsTwoDigitCanonicalFragmentMap·*(***m***)

·secondCanonicalFragmentMap· (***s***) → *secondFrag*
Maps a decimal number, presumably the *·second·* property of a date/timeSevenPropertyModel value, onto a *secondFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

Arguments:
s : a nonnegative decimal number less than 70

Result:
matches *secondFrag*

Algorithm:
Return

- *·unsTwoDigitCanonicalFragmentMap·*(***s***) when ***s*** is an integer, and
- *·unsTwoDigitCanonicalFragmentMap·*(***s***·div·1) & '.' & *·fractionDigitsCanonicalFragmentMap·*(***s***·mod·1) otherwise.

·timezoneCanonicalFragmentMap· (***t***) → *timezoneFrag*
Maps an integer, presumably the *·timezoneOffset·* property of a date/timeSevenPropertyModel value, onto a *timezoneFrag*, part of a date/timeSevenPropertyModel's ·lexical representation·.

Arguments:
t : an integer between −840 and 840 inclusive

Result:
matches *timezoneFrag*

Algorithm:
Return

- 'Z' when ***t*** is zero,
- '-' & *·unsTwoDigitCanonicalFragmentMap·*(−***t*** ·div· 60) & ':' & *·unsTwoDigitCanonicalFragmentMap·*(−***t*** ·mod· 60) when ***t*** is negative, and
- '+' & *·unsTwoDigitCanonicalFragmentMap·*(***t*** ·div· 60) & ':' & *·unsTwoDigitCanonicalFragmentMap·*(***t*** ·mod· 60) otherwise.

Canonical Mapping

·dateTimeCanonicalMap· (***dt***) → *dateLexicalRep*
Maps a dateTime value to a *dateTimeLexicalRep*.

Arguments:
dt : a complete dateTime value

Result:
matches *dateLexicalRep*

Algorithm:

Let ***DT*** be ·*yearCanonicalFragmentMap*·(***dt***'s ·*year*·) & '-' &
·*monthCanonicalFragmentMap*·(***dt***'s ·*month*·) & '-' &
·*dayCanonicalFragmentMap*·(***dt***'s ·*day*·) & 'T' &
·*hourCanonicalFragmentMap*·(***dt***'s ·*hour*·) & ':' &
·*minuteCanonicalFragmentMap*·(***dt***'s ·*minute*·) & ':' &
·*secondCanonicalFragmentMap*·(***dt***'s ·*second*·) .

Return

- ***DT*** when ***dt***'s *·timezoneOffset·* is ***absent***, and
- ***DT*** & *·timezoneCanonicalFragmentMap·*(***dt***'s *·timezoneOffset·*) otherwise.

Canonical Mapping

·timeCanonicalMap· (***ti***) → *timeLexicalRep*
Maps a time value to a *timeLexicalRep*.

Arguments:
ti : a complete time value

Result:
matches *timeLexicalRep*

Algorithm:
Let ***T*** be *·hourCanonicalFragmentMap·*(***ti***'s *·hour·*) & ':' & *·minuteCanonicalFragmentMap·*(***ti***'s *·minute·*) & ':' & *·secondCanonicalFragmentMap·*(***ti***'s *·second·*) .

Return

- ***T*** when ***ti***'s *·timezoneOffset·* is ***absent***, and
- ***T*** & *·timezoneCanonicalFragmentMap·*(***ti***'s *·timezoneOffset·*) otherwise.

Canonical Mapping

·dateCanonicalMap· (***da***) → *dateLexicalRep*
Maps a date value to a *dateLexicalRep*.

Arguments:
da : a complete date value

Result:
matches *dateLexicalRep*

Algorithm:
Let ***D*** be *·yearCanonicalFragmentMap·*(***da***'s *·year·*) & '-' & *·monthCanonicalFragmentMap·*(***da***'s *·month·*) & '-' & *·dayCanonicalFragmentMap·*(***da***'s *·day·*) .

Return

- ***D*** when ***da***'s *·timezoneOffset·* is ***absent***, and
- ***D*** & *·timezoneCanonicalFragmentMap·*(***da***'s *·timezoneOffset·*) otherwise.

Canonical Mapping

·gYearMonthCanonicalMap· (***ym***) → *gYearMonthLexicalRep*
Maps a gYearMonth value to a *gYearMonthLexicalRep*.

Arguments:

ym : a complete gYearMonth value

Result:

matches *gYearMonthLexicalRep*

Algorithm:

Let ***YM*** be *·yearCanonicalFragmentMap·*(***ym***'s *·year·*) & '-' & *·monthCanonicalFragmentMap·*(***ym***'s *·month·*) .

Return

- ***YM*** when ***ym***'s *·timezoneOffset·* is ***absent***, and
- ***YM*** & *·timezoneCanonicalFragmentMap·*(***ym***'s *·timezoneOffset·*) otherwise.

Canonical Mapping

·gYearCanonicalMap· (***gY***) → *gYearLexicalRep*

Maps a gYear value to a *gYearLexicalRep*.

Arguments:

gY : a complete gYear value

Result:

matches *gYearLexicalRep*

Algorithm:

Return

- *·yearCanonicalFragmentMap·*(***gY***'s *·year·*) when ***gY***'s *·timezoneOffset·* is ***absent***, and
- *·yearCanonicalFragmentMap·*(***gY***'s *·year·*) & *·timezoneCanonicalFragmentMap·*(***gY***'s *·timezoneOffset·*) otherwise.

Canonical Mapping

·gMonthDayCanonicalMap· (***md***) → *gMonthDayLexicalRep*

Maps a gMonthDay value to a *gMonthDayLexicalRep*.

Arguments:

md : a complete gMonthDay value

Result:

matches *gMonthDayLexicalRep*

Algorithm:

Let ***MD*** be '--' & *·monthCanonicalFragmentMap·*(***md***'s *·month·*) & '-' & *·dayCanonicalFragmentMap·*(***md***'s *·day·*) .

Return

- ***MD*** when ***md***'s *·timezoneOffset·* is ***absent***, and
- ***MD*** & *·timezoneCanonicalFragmentMap·*(***md***'s *·timezoneOffset·*) otherwise.

Canonical Mapping

·*gDayCanonicalMap*· (***gD***) → *gDayLexicalRep*
Maps a gDay value to a *gDayLexicalRep*.

Arguments:
gD : a complete gDay value

Result:
matches *gDayLexicalRep*

Algorithm:
Return

- '---' & *·dayCanonicalFragmentMap·*(***gD***'s *·day·*) when ***gD***'s *·timezoneOffset·* is ***absent***, and
- '---' & *·dayCanonicalFragmentMap·*(***gD***'s *·day·*) & *·timezoneCanonicalFragmentMap·*(***gD***'s *·timezoneOffset·*) otherwise.

Canonical Mapping

·*gMonthCanonicalMap*· (***gM***) → *gMonthLexicalRep*
Maps a gMonth value to a *gMonthLexicalRep*.

Arguments:
gM : a complete gMonth value

Result:
matches *gMonthLexicalRep*

Algorithm:
Return

- '--' & *·monthCanonicalFragmentMap·*(***gM***'s *·day·*) when ***gM***'s *·timezoneOffset·* is ***absent***, and
- '--' & *·monthCanonicalFragmentMap·*(***gM***'s *·day·*) & *·timezoneCanonicalFragmentMap·*(***gM***'s *·timezoneOffset·*) otherwise.

E.4 Lexical and Canonical Mappings for Other Datatypes

The following functions are used with various datatypes neither numeric nor date/time related.

Lexical Mapping

·*stringLexicalMap*· (***LEX***) → string
Maps a ·literal· matching the *stringRep* production to a string value.

Arguments:
LEX : a ·literal· matching *stringRep*

Result:
A string value

Algorithm:

Return ***LEX***. (The function is the identity function on the domain.)

Lexical Mapping

·***booleanLexicalMap***· (***LEX***) → boolean
Maps a ·literal· matching the *booleanRep* production to a boolean value.

Arguments:
LEX : a ·literal· matching *booleanRep*

Result:
A boolean value

Algorithm:
Return

- ***true*** when ***LEX*** is '`true`' or '`1`' , and
- ***false*** otherwise (***LEX*** is '`false`' or '`0`').

Canonical Mapping

·***stringCanonicalMap***· (***s***) → *stringRep*
Maps a string value to a *stringRep*.

Arguments:
s : a string value

Result:
matches *stringRep*

Algorithm:
Return ***s***. (The function is the identity function on the domain.)

Canonical Mapping

·***booleanCanonicalMap***· (***b***) → *booleanRep*
Maps a boolean value to a *booleanRep*.

Arguments:
b : a boolean value

Result:
matches *booleanRep*

Algorithm:
Return

- '`true`' when ***b*** is ***true***, and
- '`false`' otherwise (***b*** is ***false***).

E.4.1 Lexical and canonical mappings for hexBinary

The ·lexical mapping· for hexBinary maps each pair of hexadecimal digits to an octet, in the conventional way:

Lexical Mapping for hexBinary

·*hexBinaryMap*· (***LEX***) → hexBinary
Maps a ·literal· matching the *hexBinary* production to a sequence of octets in the form of a hexBinary value.

Arguments:

LEX : a ·literal· matching *hexBinary*

Result:

A sequence of binary octets in the form of a hexBinary value

Algorithm:

LEX necessarily includes a sequence of zero or more substrings matching the *hexOctet* production.

Let ***o*** be the sequence of octets formed by applying *·hexOctetMap·* to each *hexOctet* in ***LEX***, in order, and concatenating the results.

Return ***o***.

The auxiliary functions *·hexOctetMap·* and *·hexDigitMap·* are used by *·hexBinaryMap·*.

Mappings for hexadecimal digits

·*hexOctetMap*· (***LEX***) → octet
Maps a ·literal· matching the *hexOctet* production to a single octet.

Arguments:

LEX : a ·literal· matching *hexOctet*

Result:

A single binary octet

Algorithm:

LEX necessarily includes exactly two hexadecimal digits.

Let ***d0*** be the first hexadecimal digit in ***LEX***. Let ***d1*** be the second hexadecimal digit in ***LEX***.

Return the octet whose four high-order bits are *·hexDigitMap·*(***d0***) and whose four low-order bits are *·hexDigitMap·*(***d1***).

·*hexDigitMap*· (***d***) → a bit-sequence of length four
Maps a hexadecimal digit (a character matching the *hexDigit* production) to a sequence of four binary digits.

Arguments:

d : a hexadecimal digit

Result:

a sequence of four binary digits

Algorithm:

Return

- 0000 when ***d*** = '0',
- 0001 when ***d*** = '1',
- 0010 when ***d*** = '2',

- 0011 when ***d*** = '3',
- ...
- 1110 when ***d*** = 'E' or 'e',
- 1111 when ***d*** = 'F' or 'f'.

The ·canonical mapping· for hexBinary uses only the uppercase forms of A-F.

Canonical Mapping for hexBinary

·hexBinaryCanonical· (***o***) → *hexBinary*
Maps a hexBinary value to a literal matching the *hexBinary* production.

Arguments:
o : a hexBinary value

Result:
matches *hexBinary*

Algorithm:
Let ***h*** be the sequence of literals formed by applying *·hexOctetCanonical·* to each octet in ***o***, in order, and concatenating the results.
Return ***h***.

Auxiliary procedures for canonical mapping of hexBinary

·hexOctetCanonical· (***o***) → *hexOctet*
Maps a binary octet to a literal matching the *hexOctet* production.

Arguments:
o : a binary octet

Result:
matches *hexOctet*

Algorithm:
Let ***lo*** be the four low-order bits of ***o***, and ***hi*** be the four high-order bits.
Return *·hexDigitCanonical·*(***hi***) & *·hexDigitCanonical·*(***lo***).

·hexDigitCanonical· (***b***) → *hexDigit*
Maps a four-bit sequence to a hexadecimal digit (a literal matching the *hexDigit* production).

Arguments:
b : a sequence of four binary digits

Result:
matches *hexDigit*

Algorithm:
Return

- '0' when ***d*** = 0000,
- '1' when ***d*** = 0001,

- '2' when ***d*** = 0010,
- '3' when ***d*** = 0011,
- ...
- 'E' when ***d*** = 1110,
- 'F' when ***d*** = 1111.

F Datatypes and Facets

F.1 Fundamental Facets

The following table shows the values of the fundamental facets for each ·built-in· datatype.

	Datatype	ordered	bounded	cardinality	numeric

	token	false	false	countably infinite	false
	language	false	false	countably infinite	false
	IDREFS	false	false	countably infinite	false
	ENTITIES	false	false	countably infinite	false
	NMTOKEN	false	false	countably infinite	false
	NMTOKENS	false	false	countably infinite	false
	Name	false	false	countably infinite	false
	NCName	false	false	countably infinite	false
	ID	false	false	countably infinite	false
	IDREF	false	false	countably infinite	false
	ENTITY	false	false	countably infinite	false
	integer	total	false	countably infinite	true
	nonPositiveInteger	total	false	countably infinite	true
	negativeInteger	total	false	countably infinite	true
	long	total	true	finite	true
	int	total	true	finite	true
	short	total	true	finite	true
	byte	total	true	finite	true
	nonNegativeInteger	total	false	countably infinite	true
	unsignedLong	total	true	finite	true
	unsignedInt	total	true	finite	true
	unsignedShort	total	true	finite	true
	unsignedByte	total	true	finite	true
	positiveInteger	total	false	countably infinite	true
	yearMonthDuration	partial	false	countably infinite	false
	dayTimeDuration	partial	false	countably infinite	false
	dateTimeStamp	partial	false	countably infinite	false

G Regular Expressions

A ·regular expression· ***R*** is a sequence of characters that denote a **set of strings** *L(R)*. When used to constrain a ·lexical space·, a **regular expression** ***R*** asserts that only strings in *L(R)* are valid ·literals· for values of that type.

> **Note:** Unlike some popular regular expression languages (including those defined by Perl and standard Unix utilities), the regular expression language defined here implicitly anchors all regular expressions at the head and tail, as the most common use of regular expressions in ·pattern· is to match entire ·literals·. For example, a datatype derived from string such that all values must begin with the character `A` (#x41) and end with the character `Z` (#x5a) would be defined as follows:

```
<simpleType name='myString'>
 <restriction base='string'>
  <pattern value='A.*Z'/>
```

```
  </restriction>
</simpleType>
```

In regular expression languages that are not implicitly anchored at the head and tail, it is customary to write the equivalent regular expression as:

```
^A.*Z$
```

where "^" anchors the pattern at the head and "$" anchors at the tail.

In those rare cases where an unanchored match is desired, including `.*` at the beginning and ending of the regular expression will achieve the desired results. For example, a datatype derived from string such that all values must contain at least 3 consecutive `A` (`#x41`) characters somewhere within the value could be defined as follows:

```
<simpleType name='myString'>
 <restriction base='string'>
  <pattern value='.*AAA.*'/>
 </restriction>
</simpleType>
```

[Definition:] A **regular expression** is composed from zero or more ·branch·es, separated by `|` characters.

Regular Expression			
[65]	`regExp`	::=	`branch ( '|' branch )*`

For all ·branch·es *S*, and for all ·regular expression·s *T*, valid ·regular expression·s *R* are:	Denoting the set of strings *L(R)* containing:
(empty string)	the set containing just the empty string
S	all strings in *L(S)*
S*|*T	all strings in *L(S)* and all strings in *L(T)*

[Definition:] A **branch** consists of zero or more ·piece·s, concatenated together.

Branch			
[66]	`branch`	::=	`piece*`

For all ·piece·s *S*, and for all ·branch·es *T*, valid ·branch·es *R* are:	Denoting the set of strings *L(R)* containing:
S	all strings in *L(S)*
ST	all strings *st* with ***s*** in *L(S)* and *t* in *L(T)*

[Definition:] A **piece** is an ·atom·, possibly followed by a ·quantifier·.

Piece			
[67]	piece	::=	*atom* *quantifier*?

For all ·atom·s ***S*** and non-negative integers ***n***, ***m*** such that ***n*** **<= *m***, valid ·piece·s ***R*** are:	**Denoting the set of strings *L(R)* containing:**
S	all strings in *L(S)*
S?	the empty string, and all strings in *L(S)*.
S*	All strings in *L(S?)* and all strings *st* with ***s*** in *L(S*)* and *t* in *L(S)*. *(all concatenations of zero or more strings from L(S))*
S+	All strings *st* with ***s*** in *L(S)* and *t* in *L(S*)*. *(all concatenations of one or more strings from L(S))*
S{n,m}	All strings *st* with ***s*** in *L(S)* and *t* in *L(S{n-1,m-1})*. *(All sequences of at least n, and at most m, strings from L(S))*
S{n}	All strings in *L(S{n,n})*. *(All sequences of exactly n strings from L(S))*
S{n,}	All strings in L(S{n}S*) *(All sequences of at least n, strings from L(S))*
S{0,m}	All strings *st* with ***s*** in *L(S?)* and *t* in *L(S{0,m-1})*. *(All sequences of at most m, strings from L(S))*
S{0,0}	The set containing only the empty string

Note: The regular expression language in the Perl Programming Language [Perl] does not include a quantifier of the form `S{,m}`, since it is logically equivalent to `S{0,m}`. We have, therefore, left this logical possibility out of the regular expression language defined by this specification.

[Definition:] A **quantifier** is one of `?`, `*`, `+`, `{n,m}` or `{n,}`, which have the meanings defined in the table above.

Quantifier			
[68]	quantifier	::=	`[?*+] | ( '{' ` *quantity* ` '}' )`
[69]	quantity	::=	*quantRange* | *quantMin* | *QuantExact*
[70]	quantRange	::=	*QuantExact* `','` *QuantExact*
[71]	quantMin	::=	*QuantExact* `','`
[72]	QuantExact	::=	`[0-9]+`

[Definition:] An **atom** is either a ·normal character·, a ·character class·, or a parenthesized

·regular expression·.

Atom			
[73]	atom	::=	NormalChar | charClass | ('(' regExp ')')

For all ·normal character·s *c*, ·character class·es ***C***, and ·regular expression·s ***S***, valid ·atom·s ***R*** are:	Denoting the set of strings *L(R)* containing:
c	the single string consisting only of *c*
C	all strings in *L(C)*
(***S***)	all strings in *L(S)*

[Definition:] A **metacharacter** is either ., \, ?, *, +, {, } (,), |, [, or]. These characters have special meanings in ·regular expression·s, but can be escaped to form ·atom·s that denote the sets of strings containing only themselves, i.e., an escaped ·metacharacter· behaves like a ·normal character·.

[Definition:] A **normal character** is any XML character that is not a metacharacter. In ·regular expression·s, a normal character is an atom that denotes the singleton set of strings containing only itself.

Normal Character				
[74]	NormalChar	::=	`[^.\?*+{}()|#x5B#x5D]`	*/* n.b. #x5B = [, #x5D =] */*

G.1 Character Classes

[Definition:] A **character class** is an ·atom· ***R*** that identifies a **set of characters *C*(*R*)**. The set of strings ***L*(*R*)** denoted by a character class ***R*** contains one single-character string "***c***" for each character ***c*** in ***C*(*R*)**.

Character Class			
[75]	charClass	::=	charClassEsc | charClassExpr | WildcardEsc

A character class is either a ·character class escape· or a ·character class expression· or a ·wildcard character·.

> **Note:** The rules for which characters must be escaped and which can represent themselves are different when inside a ·character class expression·; some ·normal characters· must be escaped and some ·metacharacters· need not be.

[Definition:] A **character class expression** (charClassExpr) is a ·character group· surrounded by [and] characters. For all character groups ***G***, [***G***] is a valid **character class**

expression, identifying the set of characters ***C***([***G***]) = ***C***(***G***).

Character Class Expression			
[76]	`charClassExpr`	::=	`'[' charGroup ']'`

[Definition:] A **character group** (*charGroup*) starts with either a ·positive character group· or a ·negative character group·, and is optionally followed by a subtraction operator '-' and a further ·character class expression·. [Definition:] A ·character group· that contains a subtraction operator is referred to as a **character class subtraction**.

Character Group			
[77]	`charGroup`	::=	`'(' posCharGroup | negCharGroup ')' ( '-' charClassExpr )?`

If the first character in a *charGroup* is '^', this is taken as indicating that the *charGroup* starts with a *negCharGroup*. A *posCharGroup* can itself start with '^' but only when it appears within a *negCharGroup*, that is, when the '^' is preceded by another '^'.

> **Note:** For example, the string "`[^X]`" is ambiguous according the grammar rules, denoting either a character class consisting of a negative character group with "`X`" as a member, or a positive character class with "`X`" and "`^`" as members. The normative prose rule just given requires that the first interpretation be taken.
>
> The string "`[^]`" is unambiguous: the grammar recognizes it as a positive character group containing the character "`^`". But the grammatical derivation of the string violates the rule just given, so the string "`[^]`" MUST NOT be accepted as a regular expression.

A '`-`' character is recognized as a subtraction operator (and hence, as terminating the *posCharGroup* or *negCharGroup*) if it is immediately followed by a '`[`' character.

For any ·positive character group· or ·negative character group· ***G***, and any ·character class expression· ***C***, ***G***-C is a valid ·character group·, identifying the set of all characters in ***C(G)*** that are not in ***C(C)***.

[Definition:] A **positive character group** consists of one or more ·character group parts·, concatenated together. The set of characters identified by a **positive character group** is the union of all of the sets identified by its constituent ·character group parts·.

Positive Character Group			
[78]	`posCharGroup`	::=	`( charGroupPart )+`

For all ·character ranges· ***R***, all ·character class escapes· ***E***, and all ·positive character groups· ***P***, valid ·positive	Identifying the set of characters ***C(G)*** containing:

charater groups· *G* are:	
R	all characters in *C(R)*.
E	all characters in *C(E)*.
RP	all characters in *C(R)* and all characters in *C(P)*.
EP	all characters in *C(E)* and all characters in *C(P)*.

[Definition:] A **negative character group** (*negCharGroup*) consists of a ^ character followed by a ·positive character group·. The set of characters identified by a negative character group ***C***(^***P***) is the set of all characters that are *not* in ***C***(***P***).

Negative Character Group			
[79]	`negCharGroup`	::=	`'^' posCharGroup`

[Definition:] A **character group part** (*charGroupPart*) is any of: a single unescaped character (SingleCharNoEsc), a single escaped character (SingleCharEsc), a character class escape (charClassEsc), or a character range (charRange).

Character Group Part			
[80]	`charGroupPart`	::=	`singleChar | charRange | charClassExpr`
[81]	`singleChar`	::=	`SingleCharEsc | SingleCharNoEsc`

If a *charGroupPart* starts with a *singleChar* and this is immediately followed by a hyphen, and if the hyphen is part of the character group (that is, it is not being treated as a subtraction operator because it is followed by '`[`'), then the hyphen MUST be followed by another *singleChar*, and the sequence (*singleChar*, hyphen, *singleChar*) is treated as a *charRange*. It is an error if either of the two *singleChar*s in a *charRange* is a *SingleCharNoEsc* comprising an unescaped hyphen.

> **Note:** The rule just given resolves what would otherwise be the ambiguous interpretion of some strings, e.g. "`[a-k-z]`"; it also constrains regular expressions in ways not expressed in the grammar. For example, the rule (not the grammar) excludes the string "`[--z]`" from the regular expression language defined here.

[Definition:] A **character range** ***R*** identifies a set of characters ***C***(***R***) with UCS code points in a specified range.

Character Range			
[82]	`charRange`	::=	`singleChar '-' singleChar`

A ·character range· in the form *s*-*e*identifies the set that characters with UCS code points greater than or equal to the code point of ***s***, but not greater than the code point of *e*.

Single Unescaped Character				
[83]	`SingleCharNoEsc`	::=	`[^\#x5B#x5D]`	*/* N.B. #x5B = '[', #x5D = ']' */*

A single unescaped character (*SingleCharNoEsc*) is any character except '`[`' or '`]`'. There are special rules, described earlier, that constraint the use of the characters '`-`' and '`^`' in order to disambiguate the syntax.

A single unescaped character identifies the singleton set of characters containing that character alone.

A single escaped character (*SingleCharEsc*), when used within a character group, identifies the singleton set of characters containing the character denoted by the escape (see Character Class Escapes (§G.1.1)).

G.1.1 Character Class Escapes

[Definition:] A **character class escape** is a short sequence of characters that identifies predefined character class. The valid character class escapes are the ·single character escape·s, the ·multi-character escape·s, and the ·category escape·s (including the ·block escape·s).

Character Class Escape			
[84]	`charClassEsc`	::=	`(` *`SingleCharEsc`* `|` *`MultiCharEsc`* `|` *`catEsc`* `|` *`complEsc`* `)`

[Definition:] A **single character escape** identifies a set containing a only one character -- usually because that character is difficult or impossible to write directly into a ·regular expression·.

Single Character Escape				
[85]	`SingleCharEsc`	::=	`'\' [nrt|.?*+(){}#x2D#x5B#x5D#x5E]`	*/* N.B. #x2D = '-', #x5B = '[', #x5D = ']', #x5E = '^' */*

The valid ·single character escape·s are:	**Identifying the set of characters *C(R)* containing:**
`\n`	the newline character (#xA)
`\r`	the return character (#xD)
`\t`	the tab character (#x9)
`\\`	\
`|`	|

The valid ·single character escape·s are:	Identifying the set of characters *C(R)* containing:
`\.`	.
`\-`	-
`\^`	^
`\?`	?
`\*`	*
`\+`	+
`\{`	{
`\}`	}
`\(`	(
`\)`	)
`\[`	[
`\]`	]

[Definition:] [Unicode Database] specifies a number of possible values for the "General Category" property and provides mappings from code points to specific character properties. The set containing all characters that have property `X`, can be identified with a **category escape** `\p{X}`. The complement of this set is specified with the **category escape** `\P{X}`. (`[\P{X}]` = `[^\p{X}]`).

Category Escape

[86]	catEsc	::=	`'\p{'` *charProp* `'}'`
[87]	complEsc	::=	`'\P{'` *charProp* `'}'`
[88]	charProp	::=	*IsCategory* | *IsBlock*

[Unicode Database] is subject to future revision. For example, the mapping from code points to character properties might be updated. All ·minimally conforming· processors ·must· support the character properties defined in the version of [Unicode Database] cited in the normative references (Normative (§K.1)). However, implementors are encouraged to support the character properties defined in any later versions. When the implementation supports multiple versions of the Unicode database, and they differ in salient respects (e.g. different properties are assigned to the same character in different versions of the database), then it is ·implementation-defined· which set of property definitions is used for any given assessment episode.

> **Note:** In order to benefit from continuing work on the Unicode database, a conforming implementation might by default use the latest supported version of the character properties. In order to maximize consistency with other implementations of this specification, however, an implementation might choose to provide user options to specify the use of the version of the database cited in the normative references. The `PropertyAliases.txt` and `PropertyValueAliases.txt` files of the Unicode database may be helpful to implementors in this connection.

For convenience, the following table lists the values of the "General Category" property in the version of [Unicode Database] cited in the normative references (Normative (§K.1)).

Category	Property	Meaning
Letters	L	All Letters
	Lu	uppercase
	Ll	lowercase
	Lt	titlecase
	Lm	modifier
	Lo	other
Marks	M	All Marks
	Mn	nonspacing
	Mc	spacing combining
	Me	enclosing
Numbers	N	All Numbers
	Nd	decimal digit
	Nl	letter
	No	other
Punctuation	P	All Punctuation
	Pc	connector
	Pd	dash
	Ps	open
	Pe	close
	Pi	initial quote (may behave like Ps or Pe depending on usage)
	Pf	final quote (may behave like Ps or Pe depending on usage)
	Po	other
Separators	Z	All Separators
	Zs	space
	Zl	line
	Zp	paragraph
Symbols	S	All Symbols
	Sm	math
	Sc	currency
	Sk	modifier
	So	other

Other	C	All Others
	Cc	control
	Cf	format
	Co	private use
	Cn	not assigned

Categories

[89]	`IsCategory`	::=	*Letters* | *Marks* | *Numbers* | *Punctuation* | *Separators* | *Symbols* | *Others*
[90]	`Letters`	::=	`'L' [ultmo]?`
[91]	`Marks`	::=	`'M' [nce]?`
[92]	`Numbers`	::=	`'N' [dlo]?`
[93]	`Punctuation`	::=	`'P' [cdseifo]?`
[94]	`Separators`	::=	`'Z' [slp]?`
[95]	`Symbols`	::=	`'S' [mcko]?`
[96]	`Others`	::=	`'C' [cfon]?`

Note: The properties mentioned above exclude the `Cs` property. The `Cs` property identifies "surrogate" characters, which do not occur at the level of the "character abstraction" that XML instance documents operate on.

[Definition:] [Unicode Database] groups code points into a number of blocks such as Basic Latin (i.e., ASCII), Latin-1 Supplement, Hangul Jamo, CJK Compatibility, etc. The set containing all characters that have block name `X` (with all white space stripped out), can be identified with a **block escape** `\p{IsX}`. The complement of this set is specified with the **block escape** `\P{IsX}`. (`[\P{IsX}]` = `[^\p{IsX}]`).

Block Escape

[97]	`IsBlock`	::=	`'Is' [a-zA-Z0-9#x2D]+`	*/* N.B. #x2D = '-' */*

For convenience, the following table lists the recognized block names given in the version of [Unicode Database] cited in the normative references (Normative (§K.1)); the normative authority for any given version of [Unicode Database] is the Unicode database itself; see the "Blocks.txt" file.

Start Code	End Code	Block Name		Start Code	End Code	
#x0000	#x007F	BasicLatin		#x0080	#x00FF	Latin-1
#x0100	#x017F	LatinExtended-A		#x0180	#x024F	LatinE
#x0250	#x02AF	IPAExtensions		#x02B0	#x02FF	Spacir

#x0300	#x036F	CombiningDiacriticalMarks		#x0370	#x03FF	Greek
#x0400	#x04FF	Cyrillic		#x0500	#x052F	Cyrilli
#x0530	#x058F	Armenian		#x0590	#x05FF	Hebre
#x0600	#x06FF	Arabic		#x0700	#x074F	Syriac
#x0750	#x077F	ArabicSupplement		#x0780	#x07BF	Thaan
#x07C0	#x07FF	NKo				
#x0900	#x097F	Devanagari		#x0980	#x09FF	Benga
#x0A00	#x0A7F	Gurmukhi		#x0A80	#x0AFF	Gujara
#x0B00	#x0B7F	Oriya		#x0B80	#x0BFF	Tamil
#x0C00	#x0C7F	Telugu		#x0C80	#x0CFF	Kanna
#x0D00	#x0D7F	Malayalam		#x0D80	#x0DFF	Sinhal
#x0E00	#x0E7F	Thai		#x0E80	#x0EFF	Lao
#x0F00	#x0FFF	Tibetan		#x1000	#x109F	Myanr
#x10A0	#x10FF	Georgian		#x1100	#x11FF	Hangu
#x1200	#x137F	Ethiopic		#x1380	#x139F	Ethiop
#x13A0	#x13FF	Cherokee		#x1400	#x167F	Unifie
#x1680	#x169F	Ogham		#x16A0	#x16FF	Runic
#x1700	#x171F	Tagalog		#x1720	#x173F	Hanur
#x1740	#x175F	Buhid		#x1760	#x177F	Tagba
#x1780	#x17FF	Khmer		#x1800	#x18AF	Mongo
#x1900	#x194F	Limbu		#x1950	#x197F	TaiLe
#x1980	#x19DF	NewTaiLue		#x19E0	#x19FF	Khme
#x1A00	#x1A1F	Buginese		#x1B00	#x1B7F	Baline
#x1B80	#x1BBF	Sundanese		#x1C00	#x1C4F	Lepch
#x1C50	#x1C7F	OlChiki		#x1D00	#x1D7F	Phone
#x1D80	#x1DBF	PhoneticExtensionsSupplement		#x1DC0	#x1DFF	Comb
#x1E00	#x1EFF	LatinExtendedAdditional		#x1F00	#x1FFF	Greek
#x2000	#x206F	GeneralPunctuation		#x2070	#x209F	Super
#x20A0	#x20CF	CurrencySymbols		#x20D0	#x20FF	Comb

#x2100	#x214F	LetterlikeSymbols		#x2150	#x218F	Numb
#x2190	#x21FF	Arrows		#x2200	#x22FF	Mathe
#x2300	#x23FF	MiscellaneousTechnical		#x2400	#x243F	Contr
#x2440	#x245F	OpticalCharacterRecognition		#x2460	#x24FF	Enclos
#x2500	#x257F	BoxDrawing		#x2580	#x259F	BlockE
#x25A0	#x25FF	GeometricShapes		#x2600	#x26FF	Miscel
#x2700	#x27BF	Dingbats		#x27C0	#x27EF	Miscel
#x27F0	#x27FF	SupplementalArrows-A		#x2800	#x28FF	Braille
#x2900	#x297F	SupplementalArrows-B		#x2980	#x29FF	Miscel
#x2A00	#x2AFF	SupplementalMathematicalOperators		#x2B00	#x2BFF	Miscel
#x2C00	#x2C5F	Glagolitic		#x2C60	#x2C7F	LatinE
#x2C80	#x2CFF	Coptic				
#x2D00	#x2D2F	GeorgianSupplement		#x2D30	#x2D7F	Tifinag
#x2D80	#x2DDF	EthiopicExtended		#x2DE0	#x2DFF	Cyrillic
#x2E00	#x2E7F	SupplementalPunctuation				
#x2E80	#x2EFF	CJKRadicalsSupplement		#x2F00	#x2FDF	Kangx
#x2FF0	#x2FFF	IdeographicDescriptionCharacters		#x3000	#x303F	CJKSy
#x3040	#x309F	Hiragana		#x30A0	#x30FF	Kataka
#x3100	#x312F	Bopomofo		#x3130	#x318F	Hangu
#x3190	#x319F	Kanbun		#x31A0	#x31BF	Bopon
#x31C0	#x31EF	CJKStrokes		#x31F0	#x31FF	Kataka
#x3200	#x32FF	EnclosedCJKLettersandMonths		#x3300	#x33FF	CJKC
#x3400	#x4DBF	CJKUnifiedIdeographsExtensionA		#x4DC0	#x4DFF	YijingH
#x4E00	#x9FFF	CJKUnifiedIdeographs		#xA000	#xA48F	YiSylla
#xA490	#xA4CF	YiRadicals		#xA500	#xA63F	Vai
#xA640	#xA69F	CyrillicExtended-B		#xA700	#xA71F	Modifi
#xA720	#xA7FF	LatinExtended-D		#xA800	#xA82F	SylotiN
#xA840	#xA87F	Phags-pa		#xA880	#xA8DF	Saura
#xA900	#xA92F	KayahLi		#xA930	#xA95F	Rejang

#xAA00	#xAA5F	Cham		#xAC00	#xD7AF	Hangu
		[See note following this table.]				[See r
		[See note following this table.]		#xE000	#xF8FF	Private
#xF900	#xFAFF	CJKCompatibilityIdeographs		#xFB00	#xFB4F	Alphal
#xFB50	#xFDFF	ArabicPresentationForms-A		#xFE00	#xFE0F	Variati
#xFE10	#xFE1F	VerticalForms		#xFE20	#xFE2F	Comb
#xFE30	#xFE4F	CJKCompatibilityForms		#xFE50	#xFE6F	SmallF
#xFE70	#xFEFF	ArabicPresentationForms-B				
#xFF00	#xFFEF	HalfwidthandFullwidthForms		#xFFF0	#xFFFF	Specia
#x10000	#x1007F	LinearBSyllabary		#x10080	#x100FF	Linear
#x10100	#x1013F	AegeanNumbers		#x10140	#x1018F	Ancier
#x10190	#x101CF	AncientSymbols		#x101D0	#x101FF	Phaist
#x10280	#x1029F	Lycian		#x102A0	#x102DF	Cariar
#x10300	#x1032F	OldItalic		#x10330	#x1034F	Gothic
#x10380	#x1039F	Ugaritic		#x103A0	#x103DF	OldPe
#x10400	#x1044F	Deseret		#x10450	#x1047F	Shavia
#x10480	#x104AF	Osmanya		#x10800	#x1083F	Cypric
#x10900	#x1091F	Phoenician		#x10920	#x1093F	Lydiar
#x10A00	#x10A5F	Kharoshthi		#x12000	#x123FF	Cuneif
#x12400	#x1247F	CuneiformNumbersandPunctuation		#x1D000	#x1D0FF	Byzan
#x1D100	#x1D1FF	MusicalSymbols		#x1D200	#x1D24F	Ancier
#x1D300	#x1D35F	TaiXuanJingSymbols		#x1D360	#x1D37F	Count
				#x1D400	#x1D7FF	Mathe
#x1F000	#x1F02F	MahjongTiles		#x1F030	#x1F09F	Domir
#x20000	#x2A6DF	CJKUnifiedIdeographsExtensionB		#x2F800	#x2FA1F	CJKC
#xE0000	#xE007F	Tags		#xE0100	#xE01EF	Variati
#xF0000	#xFFFFF	SupplementaryPrivateUseArea-A		#x100000	#x10FFFF	Supple

Note: The blocks mentioned above exclude the `HighSurrogates`, `LowSurrogates` and `HighPrivateUseSurrogates` blocks. These blocks identify "surrogate" characters, which do not occur at the level of the "character abstraction" that XML instance documents operate on.

[Unicode Database] has been revised since XSD 1.0 was published, and is subject to future revision. In particular, the grouping of code points into blocks has changed, and may change again. All ·minimally conforming· processors MUST support the blocks defined in the version of [Unicode Database] cited in the normative references (Normative (§K.1)). However, implementors are encouraged to support the blocks defined in earlier and/or later versions of the Unicode Standard. When the implementation supports multiple versions of the Unicode database, and they differ in salient respects (e.g. different characters are assigned to a given block in different versions of the database), then it is ·implementation-defined· which set of block definitions is used for any given assessment episode.

In particular, the version of [Unicode Database] referenced in XSD 1.0 (namely, Unicode 3.1) contained the following blocks which have been renamed in the version cited in this specification. Since these block names may appear in regular expressions within XSD 1.0 schemas, implementors are encouraged to support the superseded block names in XSD 1.1 processors for compatibility, either by default or at user option:

- #x0370 - #x03FF: Greek
- #x20D0 - #x20FF: CombiningMarksforSymbols
- #xE000 - #xF8FF: PrivateUse
- #xF0000 - #xFFFFD: PrivateUse
- #x100000 - #x10FFFD: PrivateUse

For example, the ·block escape· for identifying the ASCII characters is `\p{IsBasicLatin}`.

[Definition:] A **multi-character escape** provides a simple way to identify any of a commonly used set of characters: [Definition:] The **wildcard character** is a metacharacter which matches almost any single character:

Multi-Character Escape			
[98]	`MultiCharEsc`	::=	`'\' [sSiIcCdDwW]`
[99]	`WildcardEsc`	::=	`'.'`

Character sequence	Equivalent ·character class·
.	[^\n\r]
\s	[#x20\t\n\r]
\S	[^\s]
\i	the set of initial name characters, those ·matched· by NameStartChar in [XML] or by Letter | '_' | ':' in [XML

Character sequence	Equivalent ·character class·
	1.0]
\I	[^\i]
\c	the set of name characters, those ·matched· by NameChar
\C	[^\c]
\d	\p{Nd}
\D	[^\d]
\w	[#x0000-#x10FFFF]-[\p{P}\p{Z}\p{C}] (*all characters except the set of "punctuation", "separator" and "other" characters*)
\W	[^\w]

> **Note:** The ·regular expression· language defined here does not attempt to provide a general solution to "regular expressions" over UCS character sequences. In particular, it does not easily provide for matching sequences of base characters and combining marks. The language is targeted at support of "Level 1" features as defined in [Unicode Regular Expression Guidelines]. It is hoped that future versions of this specification will provide support for "Level 2" features.

H Implementation-defined and implementation-dependent features (normative)

H.1 Implementation-defined features

The following features in this specification are ·implementation-defined·. Any software which claims to conform to this specification (or to the specification of any host language which embeds *XSD 1.1: Datatypes*) MUST describe how these choices have been exercised, in documentation which accompanies any conformance claim.

1. For the datatypes which depend on [XML] or [Namespaces in XML], it is ·implementation-defined· whether a conforming processor takes the relevant definitions from [XML] and [Namespaces in XML], or from [XML 1.0] and [Namespaces in XML 1.0]. Implementations MAY support either the form of these datatypes based on version 1.0 of those specifications, or the form based on version 1.1, or both.

2. For the datatypes with infinite ·value spaces·, it is ·implementation-defined· whether conforming processors set a limit on the size of the values supported. If such limits are set, they MUST be documented, and the limits MUST be equal to, or exceed, the minimal limits specified in Partial Implementation of Infinite Datatypes (§5.4). .

3. It is ·implementation-defined· whether ·primitive· datatypes other than those defined in this specification are supported.

For each ·implementation-defined· datatype, a Simple Type Definition MUST be specified which conforms to the rules given in Built-in Simple Type Definitions (§4.1.6).

In addition, the following information MUST be provided:

a. The nature of the datatype's ·lexical space·, ·value space·, and ·lexical mapping·.

b. The nature of the equality relation; in particular, how to determine whether two values which are not identical are equal.

   Note: There is no requirement that equality be distinct from identity, but it MAY be.

c. The values of the ·fundamental facets·.

d. Which of the ·constraining facets· defined in this specification are applicable to the datatype (and MAY thus be used in ·facet-based restriction· from it), and what they mean when applied to it.

e. If ·implementation-defined· ·constraining facets· are supported, which of those ·constraining facets· are applicable to the datatype, and what they mean when applied to it.

f. What URI reference (more precisely, what anyURI value) is to be used to refer to the datatype, analogous to those provided for the datatypes defined here in section Built-in Datatypes and Their Definitions (§3).

   Note: It is convenient if the URI for a datatype and the expanded name of its simple type definition are related by a simple mapping, like the URIs given for the ·built-in· datatypes in Built-in Datatypes and Their Definitions (§3). However, this is not a requirement.

g. For each ·constraining facet· given a value for the new ·primitive·, what URI reference (more precisely, what anyURI value) is to be used to refer to the usage of that facet on the datatype, analogous to those provided, for the ·built-in· datatypes, in section Built-in Datatypes and Their Definitions (§3).

   Note: As specified normatively elsewhere, the set of facets given values will at the very least include the whiteSpace facet.

The ·value space· of the ·primitive· datatype MUST be disjoint from those of the other ·primitive· datatypes.

The ·lexical mapping· defined for an ·implementation-defined· primitive MUST be a total function from the ·lexical space· onto the ·value space·. That is, (1) each ·literal· in the ·lexical space· MUST map to exactly one value, and (2) each value MUST be the image of at least one member of the ·lexical space·, and MAY be the image of more than one.

For consistency with the ·constraining facets· defined here, implementors who define new ·primitive· datatypes SHOULD allow the ·pattern· and ·enumeration· facets to apply.

The implementor SHOULD specify a ·canonical mapping· for the datatype if practicable.

4. It is ·implementation-defined· whether ·constraining facets· other than those defined in

this specification are supported.

For each ·implementation-defined· facet, the following information MUST be provided:

a. What properties the facet has, viewed as a schema component.

   Note: For most ·implementation-defined· facets, the structural pattern used for most ·constraining facets· defined in this specification is expected to be satisfactory, but other structures MAY be specified.

b. Whether the facet is a ·pre-lexical·, ·lexical·, or ·value-based· facet.

c. Whether restriction of the facet takes the form of replacing a less restrictive facet value with a more restrictive value (as in the ·minInclusive· and most other ·constraining facets· defined in this specification) or of adding new values to a set of facet values (as for the ·pattern· facet). In the former case, the information provided MUST also specify how to determine which of two given values is more restrictive (and thus can be used to restrict the other).

   When an ·implementation-defined· facet is used in ·facet-based restriction·, the new value MUST be at least as restrictive as the existing value, if any.

   Note: The effect of the preceding paragraph is to ensure that a type derived by ·facet-based restriction· using an ·implementation-defined· facet does not allow, or appear to allow, values not present in the ·base type·.

d. What ·primitive· datatypes the new ·constraining facet· applies to, and what it means when applied to them.

   For a ·pre-lexical· facet, how to compute the result of applying the facet value to any given ·literal·.

   For a ·lexical· facet, how to tell whether any given ·literal· is facet-valid with respect to it.

   For a ·value-based· facet, how to tell whether any given value in the relevant ·primitive· datatypes is facet-valid with respect to it.

   Note: The host language MAY choose to specify that ·implementation-defined· ·constraining facets· are applicable to ·built-in· ·primitive· datatypes; this information is necessary to make the ·implementation-defined· facet usable in such host languages.

e. What URI reference (more precisely, what anyURI value) is to be used to refer to the facet, analogous to those provided for the datatypes defined here in section Built-in Datatypes and Their Definitions (§3).

f. What element is to be used in XSD schema documents to apply the facet in the course of ·facet-based restriction·. A schema document MUST be provided with an element declaration for each ·implementation-defined· facet; the element declarations SHOULD specify `xs:facet` as their substitution-group head.

   Note: The elements' expanded names are used by the condition-inclusion mechanism of [XSD 1.1 Part 1: Structures] to allow schema authors to test whether a particular facet is supported and adjust the schema document's contents accordingly.

·Implementation-defined· ·pre-lexical· facets MUST NOT, when applied to ·literals· which have been whitespace-normalized by the whiteSpace facet, produce ·literals· which are no longer whitespace-normalized.

Note: It follows from the above that each ·implementation-defined· ·primitive· datatype and each ·implementation-defined· constraining facet has an expanded name. These expanded names are used by the condition-inclusion mechanism of [XSD 1.1 Part 1: Structures] to allow schema authors to test whether a particular datatype or facet is supported and adjust the schema document's contents accordingly.

H.2 Implementation-dependent features

The following features in this specification are ·implementation-dependent·. Software which claims to conform to this specification (or to the specification of any host language which embeds *XSD 1.1: Datatypes*) MAY describe how these choices have been exercised, in documentation which accompanies any conformance claim.

1. When multiple errors are encountered in type definitions or elsewhere, it is ·implementation-dependent· how many of the errors are reported (as long as at least one error is reported), and which, what form the report of errors takes, and how much detail is included.

I Changes since version 1.0

I.1 Datatypes and Facets

In order to align this specification with those being prepared by the XSL and XML Query Working Groups, a new datatype named anyAtomicType has been introduced; it serves as the base type definition for all ·primitive· ·atomic· datatypes has been introduced.

The treatment of datatypes has been made more precise and explicit; most of these changes affect the section on Datatype System (§2). Definitions have been revised thoroughly and technical terms are used more consistently.

The (numeric) equality of values is now distinguished from the identity of the values themselves; this allows float and double to treat positive and negative zero as distinct values, but nevertheless to treat them as equal for purposes of bounds checking. This allows a better alignment with the expectations of users working with IEEE floating-point binary numbers.

The {value} of the bounded component for ***list*** datatypes is now always ***false***, reflecting the fact that no ordering is prescribed for ·list· datatypes, and so they cannot be bounded using the facets defined by this specification.

Units of length have been specified for all datatypes that are permitted the length constraining facet.

The use of the namespace `http://www.w3.org/2001/XMLSchema-datatypes` has been deprecated. The definition of a namespace separate from the main namespace defined by this specification proved not to be necessary or helpful in facilitating the use, by other specifications, of the datatypes defined here, and its use raises a number of difficult unsolved practical questions.

An assertions facet has been added, to allow schema authors to associated assertions with simple type definitions, analogous to those allowed by [XSD 1.1 Part 1: Structures] for complex type definitions.

Conforming implementations MAY now support ·primitive· datatypes and facets in addition to those defined here.

I.2 Numerical Datatypes

The precisionDecimal datatype has been added. It is intended to support the floating-point decimal datatypes defined in [IEEE 754-2008]. The precisionDecimal datatype differs from decimal in that values carry not only a numeric value but also an (arithmetic) precision.

As noted above, positive and negative zero, float and double are now treated as distinct but arithmetically equal values.

The description of the lexical spaces of unsignedLong, unsignedInt, unsignedShort, and unsignedByte has been revised to agree with the schema for schemas by allowing for the possibility of a leading sign.

The float and double datatypes now follow IEEE 754 implementation practice more closely; in particular, negative and positive zero are now distinct values, although arithmetically equal. Conversely, NaN is identical but not arithmetically equal to itself.

The minimum requirements for implementation support of the precisionDecimal datatype have been clarified.

The character sequence '`+INF`' has been added to the lexical spaces of float and double.

I.3 Date/time Datatypes

The treatment of dateTime and related datatypes has been changed to provide a more explicit account of the value space in terms of seven numeric properties. The most important substantive change is that values now explicitly retain information about the time zone offset indicated in the lexical form; this allows better alignment with the treatment of such values in [XQuery 1.0 and XPath 2.0 Functions and Operators].

At the suggestion of the W3C OWL Working Group , a explicitTimezone facet has been added to allow date/time datatypes to be restricted by requiring or forbidding an explicit time zone offset from UTC, instead of making it optional. The dateTimeStamp datatype has been defined using this facet.

The treatment of the date/time datatype includes a carefully revised definition of order that ensures that for repeating datatypes (time, gDay, etc.), timezoned values will be compared as though they are on the same "calendar day" ("local" property values) so that in any given timezone, the days start at the local midnight and end just before local midnight. Days do not run from 00:00:00Z to 24:00:00Z in timezones other than Z.

The lexical representation '`0000`' for years is recognized and maps to the year 1 BCE; '`-0001`' maps to 2 BCE, etc. This is a change from version 1.0 of this specification, in order to align with established practice (the so-called "astronomical year numbering") and [ISO 8601].

Algorithms for arithmetic involving dateTime and duration values have been provided, and corrections made to the *·timeOnTimeline·* function.

The treatment of leap seconds is no longer ·implementation-defined·: the date/time types described here do not include leap-second values.

At the suggestion of the W3C Internationalization Core Working Group, most references to "time zone" have been replaced with references to "time zone offset"; this resolves issue 4642 Terminology: zone offset versus time zone.

A number of syntactic and semantic errors in some of the regular expressions given to describe the lexical spaces of the ·primitive· datatypes (most notably the date/time datatypes) have been corrected.

I.4 Other changes

Support has been added for [XML] version 1.1 and [Namespaces in XML] version 1.1. The datatypes which depend on [XML] and [Namespaces in XML] may now be used with the definitions provided by the 1.1 versions of those specifications, as well as with the definitions in the 1.0 versions. It is ·implementation-defined· whether software conforming to this specification supports the definitions given in version 1.0, or in version 1.1, of [XML] and [Namespaces in XML].

The reference to the Unicode Database [Unicode Database] has been updated from version 4.1.0 to version 5.1.0, at the suggestion of the W3C Internationalization Core Working Group

References to various other specifications have also been updated.

The account of the value space of duration has been changed to specify that values consist only of two numbers (the number of months and the number of seconds) rather than six (years, months, days, hours, minutes, seconds). This allows clearly equivalent durations like P2Y and P24M to have the same value.

Two new totally ordered restrictions of duration have been defined: yearMonthDuration, defined in yearMonthDuration (§3.4.26), and dayTimeDuration, defined in dayTimeDuration (§3.4.27). This allows better alignment with the treatment of durations in [XQuery 1.0 and XPath 2.0 Functions and Operators].

The XML representations of the ·primitive· and ·ordinary· built-in datatypes have been moved out of the schema document for schema documents in Schema for Schema Documents (Datatypes) (normative) (§A) and into a different appendix (Illustrative XML representations for the built-in simple type definitions (§C)).

Numerous minor corrections have been made in response to comments on earlier working drafts.

The treatment of topics handled both in this specification and in [XSD 1.1 Part 1: Structures] has been revised to align the two specifications more closely.

Several references to other specifications have been updated to refer to current versions of those specifications, including [XML], [Namespaces in XML], [RFC 3986], [RFC 3987], and [RFC 3548].

Requirements for the datatype-validity of values of type language have been clarified.

Explicit definitions have been provided for the lexical and ·canonical mappings· of most of the primitive datatypes.

Some errors in the definition of regular-expression metacharacters have been corrected.

The descriptions of the pattern and enumeration facets have been revised to make clearer how values from different derivation steps are combined.

A warning against using the whitespace facet for tokenizing natural-language data has been added on the request of the W3C Internationalization Working Group.

In order to correct an error in version 1 of this specification and of [XSD 1.1 Part 1: Structures], ·unions· are no longer forbidden to be members of other ·unions·. Descriptions of ·union· types have also been changed to reflect the fact that ·unions· can be derived by restricting other ·unions·. The concepts of ·transitive membership· (the members of all members, recursively) and ·basic member· (those datatypes in the transitive membership which are not ·unions·) have been introduced and are used.

The requirements of conformance have been clarified in various ways. A distinction is now made between ·implementation-defined· and ·implementation-dependent· features, and a list of such features is provided in Implementation-defined and implementation-dependent features (normative) (§H). Requirements imposed on host languages which use or incorporate the datatypes defined by this specification are defined.

The definitions of MUST, MUST NOT, and ·error· have been changed to specify that processors MUST detect and report errors in schemas and schema documents (although the quality and level of detail in the error report is not constrained).

The lexical mapping of the QName datatype, in particular its dependence on the namespace bindings in scope at the place where the ·literal· appears, has been clarified.

The characterization of ·lexical mappings· has been revised to say more clearly when they are functions and when they are not, and when (in the ·special· datatypes) there are values in the ·value space· not mapped to by any members of the ·lexical space·.

The nature of equality and identity of lists has been clarified.

Enumerations, identity constraints, and value constraints now use equality-based comparisons, not identity-based comparisons, in cases where there is a difference between identity and equality.

The mutual relations of lists and unions have been clarified, in particular the restrictions on what kinds of datatypes MAY appear as the ·item type· of a list or among the ·member types· of a union.

Unions with no member types (and thus with empty ·value space· and ·lexical space·) are now explicitly allowed.

Cycles in the definitions of ·unions· and in the derivation of simple types are now explicitly forbidden.

A number of minor errors and obscurities have been fixed.

J Glossary (non-normative)

The listing below is for the benefit of readers of a printed version of this document: it collects together all the definitions which appear in the document above.

absent
: Throughout this specification, the value ***absent*** is used as a distinguished value to indicate that a given instance of a property "has no value" or "is absent".

active basic member
: If the ·active member type· is itself a ·union·, one of *its* members will be *its* ·active member type·, and so on, until finally a ·basic (non-union) member· is reached. That ·basic member· is the **active basic member** of the union.

active member type
: In a valid instance of any ·union·, the first of its members in order which accepts the instance as valid is the **active member type**.

ancestor
: The **ancestors** of a type definition are its {base type definition} and the ·ancestors· of its {base type definition}.

atomic
: **Atomic** datatypes are those whose ·value spaces· contain only ·atomic values·. **Atomic** datatypes are anyAtomicType and all datatypes ·derived· from it.

atomic value
: An **atomic value** is an elementary value, not constructed from simpler values by any user-accessible means defined by this specification.

base type
: Every datatype other than anySimpleType is associated with another datatype, its **base type**. **Base types** can be ·special·, ·primitive·, or ·ordinary·.

basic member
: Those members of the ·transitive membership· of a ·union· datatype ***U*** which are themselves not ·union· datatypes are the **basic members** of ***U***.

built-in
: **Built-in** datatypes are those which are defined in this specification; they can be ·special·, ·primitive·, or ·ordinary· datatypes .

canonical mapping
: The **canonical mapping** is a prescribed subset of the inverse of a ·lexical mapping· which is one-to-one and whose domain (where possible) is the entire range of the ·lexical mapping· (the ·value space·).

canonical representation
: The **canonical representation** of a value in the ·value space· of a datatype is the ·lexical representation· associated with that value by the datatype's ·canonical mapping·

character class subtraction
: A ·character group· that contains a subtraction operator is referred to as a **character class subtraction**.

character group part
: A **character group part** (*charGroupPart*) is any of: a single unescaped character (SingleCharNoEsc), a single escaped character (SingleCharEsc), a character class escape (charClassEsc), or a character range (charRange).

constraining facet
: **Constraining facets** are schema components whose values may be set or changed during ·derivation· (subject to facet-specific controls) to control various aspects of the derived datatype.

Constraint on Schemas
: **Constraint on Schemas**

constructed
: All ·ordinary· datatypes are defined in terms of, or **constructed** from, other datatypes, either by ·restricting· the ·value space· or ·lexical space· of a ·base type· using zero or more ·constraining facets· or by specifying the new datatype as a ·list· of items of some ·item type·, or by defining it as a ·union· of some specified sequence of ·member types·.

datatype
: In this specification, a **datatype** has three properties:

  - A ·value space·, which is a set of values.
  - A ·lexical space·, which is a set of ·literals· used to denote the values.
  - A small collection of *functions, relations, and procedures* associated with the datatype. Included are equality and (for some datatypes) order relations on the ·value space·, and a ·lexical mapping·, which is a mapping from the ·lexical space· into the ·value space·.

derived
: A datatype ***T*** is **immediately derived** from another datatype ***X*** if and only if ***X*** is the ·base type· of ***T***.

derived
: A datatype ***R*** is **derived** from another datatype ***B*** if and only if one of the following is true:

  - ***B*** is the ·base type· of ***R***.
  - There is some datatype ***X*** such that ***X*** is the ·base type· of ***R***, and ***X*** is derived from ***B***.

div
: If ***m*** and ***n*** are numbers, then ***m*** **div** ***n*** is the greatest integer less than or equal to ***m*** / ***n*** .

error
: **error**

facet-based restriction
: A datatype is defined by **facet-based restriction** of another datatype (its ·base type·), when values for zero or more ·constraining facets· are specified that serve to constrain its ·value space· and/or its ·lexical space· to a subset of those of the ·base type·.

for compatibility
: for compatibility

fundamental facet
: Each **fundamental facet** is a schema component that provides a limited piece of information about some aspect of each datatype.

implementation-defined
: Something which MAY vary among conforming implementations, but which MUST be specified by the implementor for each particular implementation, is **implementation-defined**.

implementation-dependent
: Something which MAY vary among conforming implementations, is not specified by this or any W3C specification, and is not required to be specified by the implementor for any

particular implementation, is **implementation-dependent**.

incomparable
: Two values that are neither equal, less-than, nor greater-than are **incomparable**. Two values that are not ·incomparable· are **comparable**.

intervening union
: If a datatype ***M*** is in the ·transitive membership· of a ·union· datatype ***U***, but not one of ***U***'s ·member types·, then a sequence of one or more ·union· datatypes necessarily exists, such that the first is one of the ·member types· of ***U***, each is one of the ·member types· of its predecessor in the sequence, and ***M*** is one of the ·member types· of the last in the sequence. The ·union· datatypes in this sequence are said to **intervene** between ***M*** and ***U***. When ***U*** and ***M*** are given by the context, the datatypes in the sequence are referred to as the **intervening unions**. When ***M*** is one of the ·member types· of ***U***, the set of **intervening unions** is the empty set.

item type
: The ·atomic· or ·union· datatype that participates in the definition of a ·list· datatype is the **item type** of that ·list· datatype.

leap-second
: A **leap-second** is an additional second added to the last day of December, June, October, or March, when such an adjustment is deemed necessary by the International Earth Rotation and Reference Systems Service in order to keep ·UTC· within 0.9 seconds of observed astronomical time. When leap seconds are introduced, the last minute in the day has more than sixty seconds. In theory leap seconds can also be removed from a day, but this has not yet occurred. (See [International Earth Rotation Service (IERS)], [ITU-R TF.460-6].) Leap seconds are *not* supported by the types defined here.

lexical
: A constraining facet which directly restricts the ·lexical space· of a datatype is a **lexical** facet.

lexical mapping
: The **lexical mapping** for a datatype is a prescribed relation which maps from the ·lexical space· of the datatype into its ·value space·.

lexical representation
: The members of the ·lexical space· are **lexical representations** of the values to which they are mapped.

lexical space
: The **lexical space** of a datatype is the prescribed set of strings which ·the lexical mapping· for that datatype maps to values of that datatype.

list
: **List** datatypes are those having values each of which consists of a finite-length (possibly empty) sequence of ·atomic values·. The values in a list are drawn from some ·atomic· datatype (or from a ·union· of ·atomic· datatypes), which is the ·item type· of the **list**.

literal
: A sequence of zero or more characters in the Universal Character Set (UCS) which may or may not prove upon inspection to be a member of the ·lexical space· of a given datatype and thus a ·lexical representation· of a given value in that datatype's ·value space·, is referred to as a **literal**.

match
: **match**

may
: MAY

member types

The datatypes that participate in the definition of a ·union· datatype are known as the **member types** of that ·union· datatype.

minimally conforming

Implementations claiming **minimal conformance** to this specification independent of any host language MUST do **all** of the following:

1 Support all the ·built-in· datatypes defined in this specification.

2 Completely and correctly implement all of the ·constraints on schemas· defined in this specification.

3 Completely and correctly implement all of the ·Validation Rules· defined in this specification, when checking the datatype validity of literals against datatypes.

mod

If ***m*** and ***n*** are numbers, then ***m*** **mod** ***n*** is ***m*** − ***n*** × (***m*** ·div· ***n***) .

must

MUST

must not

MUST NOT

nearest built-in datatype

For any datatype ***T***, the **nearest built-in datatype** to ***T*** is the first ·built-in· datatype encountered in following the chain of links connecting each datatype to its ·base type·. If ***T*** is a ·built-in· datatype, then the nearest built-in datatype of ***T*** is ***T*** itself; otherwise, it is the nearest built-in datatype of ***T***'s ·base type·.

optional

An **optional** property is *permitted* but not *required* to have the distinguished value ***absent***.

ordered

A ·value space·, and hence a datatype, is said to be **ordered** if some members of the ·value space· are drawn from a ·primitive· datatype for which the table in Fundamental Facets (§F.1) specifies the value ***total*** or ***partial*** for the *ordered* facet.

ordinary

Ordinary datatypes are all datatypes other than the ·special· and ·primitive· datatypes.

owner

A component may be referred to as the **owner** of its properties, and of the values of those properties.

pre-lexical

A constraining facet which is used to normalize an initial ·literal· before checking to see whether the resulting character sequence is a member of a datatype's ·lexical space· is a **pre-lexical** facet.

precisionDecimal

The **precisionDecimal** datatype represents the numeric value and (arithmetic) precision of decimal numbers which retain precision; it also includes values for positive and negative infinity and for "not a number", and it differentiates between "positive zero" and "negative zero".

primitive

Primitive datatypes are those datatypes that are not ·special· and are not defined in terms of other datatypes; they exist *ab initio*.

regular expression

A **regular expression** is composed from zero or more ·branch·es, separated by | characters.

restriction

A datatype ***R*** is a **restriction** of another datatype ***B*** when

Schema Representation Constraint

Schema Representation Constraint

should

SHOULD

special

The **special** datatypes are anySimpleType and anyAtomicType.

special value

A **special value** is an object whose only relevant properties for purposes of this specification are that it is distinct from, and unequal to, any other values (special or otherwise).

transitive membership

The **transitive membership** of a ·union· is the set of its own ·member types·, and the ·member types· of its members, and so on. More formally, if ***U*** is a ·union·, then (a) its ·member types· are in the transitive membership of ***U***, and (b) for any datatypes ***T1*** and ***T2***, if ***T1*** is in the transitive membership of ***U*** and ***T2*** is one of the ·member types· of ***T1***, then ***T2*** is also in the transitive membership of ***U***.

union

Union datatypes are (a) those whose ·value spaces·, ·lexical spaces·, and ·lexical mappings· are the union of the ·value spaces·, ·lexical spaces·, and ·lexical mappings· of one or more other datatypes, which are the ·member types· of the union, or (b) those derived by ·facet-based restriction· of another union datatype.

unknown

A datatype which is not available for use is said to be **unknown**.

unknown

An ·constraining facet· which is not supported by the processor in use is **unknown**.

user-defined

User-defined datatypes are those datatypes that are defined by individual schema designers.

UTC

Universal Coordinated Time (**UTC**) is an adaptation of TAI which closely approximates UT1 by adding ·leap-seconds· to selected ·UTC· days.

Validation Rule

Validation Rule

value space

The **value space** *of a datatype* is the set of values for that datatype.

value-based

A constraining facet which directly restricts the ·value space· of a datatype is a **value-based** facet.

wildcard character

The **wildcard character** is a metacharacter which matches almost any single character:

XDM representation

For any value ***V*** and any datatype ***T***, the **XDM representation of *V* under *T*** is defined recursively as follows. Call the XDM representation ***X***. Then

1 If ***T*** = ·`xs:anySimpleType`· or ·`xs:anyAtomicType`· then ***X*** is ***V***, and the dynamic type of ***X*** is `xs:untypedAtomic`.

2 If ***T*** . {variety} = ***atomic***, then let ***T2*** be the ·nearest built-in datatype· to ***T***. If ***V*** is a member of the ·value space· of ***T2***, then ***X*** is ***V*** and the dynamic type of ***X*** is ***T2***. Otherwise (i.e. if ***V*** is not a member of the ·value space· of ***T2***), ***X*** is the ·XDM representation· of ***V*** under ***T2*** . {base type definition}.

3 If ***T*** . {variety} = ***list***, then ***X*** is a sequence of atomic values, each atomic value being the ·XDM representation· of the corresponding item in the list ***V*** under ***T*** . {item type definition}.

4 If ***T*** . {variety} = ***union***, then ***X*** is the ·XDM representation· of ***V*** under the ·active basic member· of ***V*** when validated against ***T***. If there is no ·active basic member·, then ***V*** has no ·XDM representation· under ***T***.

K References

K.1 Normative

IEEE 754-2008
IEEE. *IEEE Standard for Floating-Point Arithmetic*. 29 August 2008. http://ieeexplore.ieee.org/ISOL/standardstoc.jsp?punumber=4610933

Namespaces in XML
World Wide Web Consortium. *Namespaces in XML 1.1 (Second Edition)*, ed. Tim Bray et al. W3C Recommendation 16 August 2006. Available at: http://www.w3.org/TR/xml-names11/ The edition cited is the one current at the date of publication of this specification. Implementations MAY follow the edition cited and/or any later edition(s); it is implementation-defined which. For details of the dependency of this specification on Namespaces in XML 1.1, see Dependencies on Other Specifications (§1.3).

Namespaces in XML 1.0
World Wide Web Consortium. *Namespaces in XML 1.0 (Second Edition)*, ed. Tim Bray et al. W3C Recommendation 16 August 2006. Available at: http://www.w3.org/TR/REC-xml-names/ The edition cited is the one current at the date of publication of this specification. Implementations MAY follow the edition cited and/or any later edition(s); it is implementation-defined which. For details of the dependency of this specification on Namespaces in XML 1.0, see Dependencies on Other Specifications (§1.3).

RFC 3548
S. Josefsson, ed. *RFC 3548: The Base16, Base32, and Base64 Data Encodings*. July 2003. Available at: http://www.ietf.org/rfc/rfc3548.txt

Unicode Database
The Unicode Consortium. *Unicode Character Database*. Revision 5.1.0, by Mark Davis and Ken Whistler, 2008-03-18. Available at: http://www.unicode.org/Public/5.1.0/ucd/UCD.html

XDM
World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*, ed. Mary Fernández et al. W3C Recommendation 23 January 2007. Available at: http://www.w3.org/TR/xpath-datamodel/.

XML
World Wide Web Consortium. *Extensible Markup Language (XML) 1.1 (Second Edition)*, ed. Tim Bray et al. W3C Recommendation 16 August 2006, edited in place 29 September 2006. Available at http://www.w3.org/TR/xml11/ The edition cited is the one current at the date of publication of this specification. Implementations MAY follow the edition cited and/or any later edition(s); it is implementation-defined which. For details of the dependency of this specification on XML 1.1, see Dependencies on Other Specifications (§1.3).

XML 1.0
World Wide Web Consortium. *Extensible Markup Language (XML) 1.0(Fifth Edition)*, ed. Tim Bray et al. W3C Recommendation 26 November 2008. Available at http://www.w3.org/TR/REC-xml/. The edition cited is the one current at the date of publication of this specification. Implementations MAY follow the edition cited and/or any

later edition(s); it is implementation-defined which. For details of the dependency of this specification on XML, see Dependencies on Other Specifications (§1.3).

XPath 2.0
World Wide Web Consortium. *XML Path Language 2.0*, ed. Anders Berglund et al. W3C Recommendation 23 January 2007. Available at: http://www.w3.org/TR/xpath20/.

XQuery 1.0 and XPath 2.0 Functions and Operators
World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Functions and Operators*, ed. Ashok Malhotra, Jim Melton, and Norman Walsh. W3C Recommendation 23 January 2007 Available at: http://www.w3.org/TR/xpath-functions/.

XSD 1.1 Part 1: Structures
World Wide Web Consortium. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*, ed. Shudi (Sandy) Gao 高殊镝, C. M. Sperberg-McQueen, and Henry S. Thompson. W3C Candidate Recommendation 30 April 2009. Available at: http://www.w3.org/TR/2009/CR-xmlschema11-1-20090430/structures.html The edition cited is the one current at the date of publication of this specification. Implementations MAY follow the edition cited and/or any later edition(s); it is implementation-defined which.

K.2 Non-normative

BCP 47
Internet Engineering Task Force (IETF). Best Current Practices 47. 2006. Available at: http://tools.ietf.org/rfc/bcp/bcp47. Concatenation of *RFC 4646: Tags for Identifying Languages*, ed. A. Phillips and M. Davis, September 2006, http://www.ietf.org/rfc/bcp/bcp47.txt, and *RFC 4647: Matching of Language Tags*, ed. A Phillips and M. Davis, September 2006, http://www.rfc-editor.org/rfc/bcp/bcp47.txt.

Clinger, WD (1990)
William D Clinger. *How to Read Floating Point Numbers Accurately.* In *Proceedings of Conference on Programming Language Design and Implementation*, pages 92-101. Available at: ftp://ftp.ccs.neu.edu/pub/people/will/howtoread.ps

HTML 4.01
World Wide Web Consortium. *HTML 4.01 Specification*, ed. Dave Raggett, Arnaud Le Hors, and Ian Jacobs. W3C Recommendation 24 December 1999. Available at: http://www.w3.org/TR/html401/

International Earth Rotation Service (IERS)
International Earth Rotation Service (IERS). See http://maia.usno.navy.mil

ISO 11404
ISO (International Organization for Standardization). *Language-independent Datatypes.* ISO/IEC 11404:2007. See http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39479

ISO 8601
ISO (International Organization for Standardization). *Representations of dates and times, 1988-06-15.*

ISO 8601:2000 Second Edition
ISO (International Organization for Standardization). *Representations of dates and times, second edition, 2000-12-15.*

ITU-R TF.460-6
International Telecommunication Union (ITU). *Recommendation ITU-R TF.460-6: Standard-frequency and time-signal emissions*. [Geneva: ITU, February 2002.]

LEIRI
Legacy extended IRIs for XML resource identification, ed. Henry S. Thompson, Richard

Tobin, and Norman Walsh. W3C Working Group Note 3 November 2008. See http://www.w3.org/TR/leiri/

Perl

The Perl Programming Language. See http://www.perl.com/pub/language/info/software.html

RDF Schema

World Wide Web Consortium. *RDF Vocabulary Description Language 1.0: RDF Schema*, ed. Dan Brickley and R. V. Guha. W3C Recommendation 10 February 2004. Available at: http://www.w3.org/TR/rdf-schema/

RFC 2045

N. Freed and N. Borenstein. *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. 1996. Available at: http://www.ietf.org/rfc/rfc2045.txt

RFC 3066

H. Alvestrand, ed. *RFC 3066: Tags for the Identification of Languages* 1995. Available at: http://www.ietf.org/rfc/rfc3066.txt

RFC 3986

T. Berners-Lee, R. Fielding, and L. Masinter, *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*. January 2005. Available at: http://www.ietf.org/rfc/rfc3986.txt

RFC 3987

M. Duerst and M. Suignard. *RFC 3987: Internationalized Resource Identifiers (IRIs)* . January 2005. Available at: http://www.ietf.org/rfc/rfc3987.txt

RFC 4646

A. Phillips and M. Davis, ed. *RFC 4646: Tags for Identifying Languages* 2006. Available at: http://www.ietf.org/rfc/rfc4646.txt

RFC 4647

A. Phillips and M. Davis, ed. *RFC 4647: Matching of Language Tags* 2006. Available at: http://www.ietf.org/rfc/rfc4647.txt

Ruby

World Wide Web Consortium. *Ruby Annotation*, ed. Marcin Sawicki et al. W3C Recommendation 31 May 2001 (Markup errors corrected 25 June 2008). Available at: http://www.w3.org/TR/ruby/

SQL

ISO (International Organization for Standardization). *ISO/IEC 9075-2:1999, Information technology --- Database languages --- SQL --- Part 2: Foundation (SQL/Foundation)*. [Geneva]: International Organization for Standardization, 1999. See http://www.iso.org/iso/home.htm

Timezones

World Wide Web Consortium. *Working with Time Zones*, ed. Addison Phillips et al. W3C Working Group Note 13 October 2005. Available at http://www.w3.org/TR/timezone/

U.S. Naval Observatory Time Service Department

Information about Leap Seconds Available at: http://tycho.usno.navy.mil/leapsec.html

Unicode Regular Expression Guidelines

Mark Davis. *Unicode Regular Expression Guidelines*, 1988. Available at: http://www.unicode.org/unicode/reports/tr18/

USNO Historical List

U.S. Naval Observatory Time Service Department, *Historical list of leap seconds* Available at: ftp://maia.usno.navy.mil/ser7/tai-utc.dat

XML Schema Language: Part 0 Primer

World Wide Web Consortium. XML Schema Language: Part 0 Primer Second Edition, ed. David C. Fallside and Priscilla Walmsley. W3C Recommendation 28 October 2004.

Available at: http://www.w3.org/TR/xmlschema-0/
XML Schema Requirements
XML Schema Requirements , ed. Ashok Malhotra and Murray Maloney. W3C Note 15 February 1999. Available at: http://www.w3.org/TR/NOTE-xml-schema-req
XSL
World Wide Web Consortium. *Extensible Stylesheet Language (XSL)*, ed. Anders Berglund. W3C Recommendation 05 December 2006. Available at: http://www.w3.org/TR/xsl11/

L Acknowledgements (non-normative)

Along with the editors thereof, the following contributed material to the first version of this specification:

Asir S. Vedamuthu, webMethods, Inc
Mark Davis, IBM

Co-editor Ashok Malhotra's work on this specification from March 1999 until February 2001 was supported by IBM, and from then until May 2004 by Microsoft. Since July 2004 his work on this specification has been supported by Oracle Corporation.

The work of Dave Peterson as a co-editor of this specification was supported by IDEAlliance (formerly GCA) through March 2004, and beginning in April 2004 by SGML*Works!*.

The work of C. M. Sperberg-McQueen as a co-editor of this specification was supported by the World Wide Web Consortium through January 2009, and beginning in February 2009 by Black Mesa Technologies LLC.

The XML Schema Working Group acknowledges with thanks the members of other W3C Working Groups and industry experts in other forums who have contributed directly or indirectly to the creation of this document and its predecessor.

At the time this Working Draft is published, the members in good standing of the XML Schema Working Group are:

- Paul V. Biron, Invited expert
- David Ezell, National Association of Convenience Stores (NACS) (*chair*)
- Shudi (Sandy) Gao 高殊镝, IBM
- Mary Holstege, Mark Logic
- Michael Kay, Invited expert
- Paolo Marinelli, University of Bologna
- Noah Mendelsohn, IBM
- Dave Peterson, Invited expert
- C. M. Sperberg-McQueen, invited expert
- Henry S. Thompson, University of Edinburgh and W3C (*staff contact*)
- Scott Tsao, The Boeing Company
- Fabio Vitali, University of Bologna
- Stefano Zacchiroli, University of Bologna

The XML Schema Working Group has benefited in its work from the participation and contributions of a number of people who are no longer members of the Working Group in good standing at the time of publication of this Working Draft. Their names are given below. In particular we note with sadness the accidental death of Mario Jeckle shortly before publication

of the first Working Draft of XML Schema 1.1. Affiliations given are (among) those current at the time of the individuals' work with the WG.

- Paula Angerstein, Vignette Corporation
- Leonid Arbouzov, Sun Microsystems
- Jim Barnette, Defense Information Systems Agency (DISA)
- David Beech, Oracle Corp.
- Gabe Beged-Dov, Rogue Wave Software
- Laila Benhlima, Ecole Mohammadia d'Ingenieurs Rabat (EMI)
- Doris Bernardini, Defense Information Systems Agency (DISA)
- Don Box, DevelopMentor
- Allen Brown, Microsoft
- Lee Buck, TIBCO Extensibility
- Greg Bumgardner, Rogue Wave Software
- Dean Burson, Lotus Development Corporation
- Charles E. Campbell, Invited expert
- Oriol Carbo, University of Edinburgh
- Wayne Carr, Intel
- Peter Chen, Bootstrap Alliance and LSU
- Tyng-Ruey Chuang, Academia Sinica
- Tony Cincotta, NIST
- David Cleary, Progress Software
- Mike Cokus, MITRE
- Dan Connolly, W3C (*staff contact*)
- Ugo Corda, Xerox
- Roger L. Costello, MITRE
- Joey Coyle, Health Level Seven
- Haavard Danielson, Progress Software
- Josef Dietl, Mozquito Technologies
- Kenneth Dolson, Defense Information Systems Agency (DISA)
- Andrew Eisenberg, Progress Software
- Rob Ellman, Calico Commerce
- Tim Ewald, Developmentor
- Alexander Falk, Altova GmbH
- David Fallside, IBM
- George Feinberg, Object Design
- Dan Fox, Defense Logistics Information Service (DLIS)
- Charles Frankston, Microsoft
- Matthew Fuchs, Commerce One
- Andrew Goodchild, Distributed Systems Technology Centre (DSTC Pty Ltd)
- Xan Gregg, TIBCO Extensibility
- Paul Grosso, Arbortext, Inc
- Martin Gudgin, DevelopMentor
- Ernesto Guerrieri, Inso
- Dave Hollander, Hewlett-Packard Company (*co-chair*)
- Nelson Hung, Corel
- Jane Hunter, Distributed Systems Technology Centre (DSTC Pty Ltd)
- Michael Hyman, Microsoft
- Renato Iannella, Distributed Systems Technology Centre (DSTC Pty Ltd)
- Mario Jeckle, DaimlerChrysler
- Rick Jelliffe, Academia Sinica
- Marcel Jemio, Data Interchange Standards Association
- Simon Johnston, Rational Software

- Kohsuke Kawaguchi, Sun Microsystems
- Dianne Kennedy, Graphic Communications Association
- Janet Koenig, Sun Microsystems
- Setrag Khoshafian, Technology Deployment International (TDI)
- Melanie Kudela, Uniform Code Council
- Ara Kullukian, Technology Deployment International (TDI)
- Andrew Layman, Microsoft
- Dmitry Lenkov, Hewlett-Packard Company
- Bob Lojek, Mozquito Technologies
- John McCarthy, Lawrence Berkeley National Laboratory
- Matthew MacKenzie, XML Global
- Eve Maler, Sun Microsystems
- Ashok Malhotra, IBM, Microsoft, Oracle
- Murray Maloney, Muzmo Communication, acting for Commerce One
- Lisa Martin, IBM
- Jim Melton, Oracle Corp
- Adrian Michel, Commerce One
- Alex Milowski, Invited expert
- Don Mullen, TIBCO Extensibility
- Murata Makoto, Xerox
- Ravi Murthy, Oracle
- Chris Olds, Wall Data
- Frank Olken, Lawrence Berkeley National Laboratory
- David Orchard, BEA Systems, Inc.
- Paul Pedersen, Mark Logic Corporation
- Shriram Revankar, Xerox
- Mark Reinhold, Sun Microsystems
- Jonathan Robie, Software AG
- Cliff Schmidt, Microsoft
- John C. Schneider, MITRE
- Eric Sedlar, Oracle Corp.
- Lew Shannon, NCR
- Anli Shundi, TIBCO Extensibility
- William Shea, Merrill Lynch
- Jerry L. Smith, Defense Information Systems Agency (DISA)
- John Stanton, Defense Information Systems Agency (DISA)
- Tony Stewart, Rivcom
- Bob Streich, Calico Commerce
- William K. Stumbo, Xerox
- Hoylen Sue, Distributed Systems Technology Centre (DSTC Pty Ltd)
- Ralph Swick, W3C
- John Tebbutt, NIST
- Ross Thompson, Contivo
- Matt Timmermans, Microstar
- Jim Trezzo, Oracle Corp.
- Steph Tryphonas, Microstar
- Mark Tucker, Health Level Seven
- Asir S. Vedamuthu, webMethods, Inc
- Scott Vorthmann, TIBCO Extensibility
- Priscilla Walmsley, XMLSolutions
- Norm Walsh, Sun Microsystems
- Cherry Washington, Defense Information Systems Agency (DISA)
- Aki Yoshida, SAP AG

- Kongyi Zhou, Oracle