XML Schema Part 2: Datatypes

Second Edition

W3C Recommendation 28 October 2004

This version:

http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

Latest version:

http://www.w3.org/TR/xmlschema-2/

Previous version:

http://www.w3.org/TR/2004/PER-xmlschema-2-20040318/

Authors and Contributors:

Paul V. Biron (Kaiser Permanente, for Health Level Seven) <Paul.V.Biron@kp.org>
Ashok Malhotra (Microsoft (formerly of IBM)) <ashokma@microsoft.com>

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 1.0, provides a superset of the capabilities found in XML 1.0 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 is a W3C Recommendation, which forms part of the Second Edition of XML Schema. This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

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 are discussed in the XML Schema Requirements document. The authors of this document are the members of the XML Schema Working Group. Different parts of this specification have different editors.

This document was produced under the 24 January 2002 Current Patent Practice (CPP) as amended by the W3C Patent Policy Transition Procedure. The Working Group maintains a public list of patent disclosures relevant to this document; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) with respect to this specification should disclose the information in accordance with section 6 of the W3C Patent Policy.

The English version of this specification is the only normative version. Information about translations of this document is available at http://www.w3.org/2001/05/xmlschema-translations.

This second edition is *not* a new version, it merely incorporates the changes dictated by the corrections to errors found in the first edition as agreed by the XML Schema Working Group, as a convenience to readers. A separate list of all such corrections is available at http://www.w3.org/2001/05/xmlschema-errata.

The errata list for this second edition is available at http://www.w3.org/2004/03/xmlschema-errata.

Please report errors in this document to www-xml-schema-comments@w3.org (archive).

☞ Ashok Malhotra's affiliation has changed since the completion of editorial work on this second edition. He is now at Oracle, and can be contacted at <ashok.malhotra@oracle.com>.

Table of Contents

Appendices

This page is intentionally left blank.

1. Introduction

1.1. Purpose

The [XML 1.0 (Second Edition)] 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.

1.2. 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).

1.3. Scope

This portion of the XML Schema Language discusses 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].

1.4. 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:

for compatibility

A feature of this specification included solely to ensure that schemas which use this feature remain compatible with [XML 1.0 (Second Edition)]

may

Conforming documents and processors are permitted to but need not behave as described.

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 it belongs to the language generated by that production.

must

Conforming documents and processors are required to behave as described; otherwise they are in error.

error

A violation of the rules of this specification; results are undefined. Conforming software may detect and report an *error* and may recover from it.

1.5. 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:

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 § 4 – Datatype components on page 35.

Schema Representation Constraint

Constraints on the representation of schema components in XML. Some but not all of these are expressed in Appendix A – Schema for Datatype Definitions (normative) on page 59 and Appendix B – DTD for Datatype Definitions (non-normative) on page 59.

Validation Rule

Constraints expressed by schema components which information items must satisfy to be schema-valid. Largely to be found in § 4 – Datatype components on page 35.

2. Type System

This section describes the conceptual framework behind the type 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 computer representations of well known abstract concepts such as *integer* and *date*. It is not the place of this specification to define these abstract concepts; many other publications provide excellent definitions.

2.1. Datatype

In this specification, a *datatype* is a 3-tuple, consisting of a) a set of distinct values, called its value space, b) a set of lexical representations, called its lexical space, and c) a set of facets that characterize properties of the value space, individual values or lexical items.

2.2. Value space

A *value space* is the set of values for a given datatype. Each value in the *value space* of a datatype is denoted by one or more literals in its lexical space.

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

- defined axiomatically from fundamental notions (intensional definition) [see primitive]
- enumerated outright (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]

value spaces have certain properties. For example, they always have the property of cardinality, some definition of *equality* and might be ordered, by which individual values within the value space can be compared to one another. The properties of value spaces that are recognized by this specification are defined in § 2.4.1 – Fundamental facets on page 5.

2.3. Lexical space

In addition to its value space, each datatype also has a lexical space.

A *lexical space* is the set of valid *literals* for a datatype.

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

☞ The literals in the lexical spaces defined in this specification have the following characteristics:

Interoperability:

The number of literals for each value has been kept small; for many datatypes there is a one-to-one mapping between literals and values. This makes it easy to exchange the values between different systems. In many cases, conversion from locale-dependent representations will be required on both the originator and the recipient side, both for computer processing and for interaction with humans.

Basic readability:

Textual, rather than binary, literals are used. This makes hand editing, debugging, and similar activities possible.

Ease of parsing and serializing:

Where possible, literals correspond to those found in common programming languages and libraries.

2.3.1. Canonical Lexical Representation

While the datatypes defined in this specification have, for the most part, a single lexical representation i.e. each value in the datatype's value space is denoted by a single literal in its lexical space, this is not always the case. The example in the previous section showed two literals for the datatype which denote the same value. Similarly, there may be several literals for one of the date or time datatypes that denote the same value using different timezone indicators.

A *canonical lexical representation* is a set of literals from among the valid set of literals for a datatype such that there is a one-to-one mapping between literals in the *canonical lexical representation* and values in the value space.

2.4. Facets

A *facet* is a single defining aspect of a value space. Generally speaking, each facet characterizes a value space along independent axes or dimensions.

The facets of a datatype serve to distinguish those aspects of one datatype which *differ* from other datatypes. Rather than being defined solely in terms of a prose description the datatypes in this specification are defined in terms of the *synthesis* of facet values which together determine the value space and properties of the datatype.

Facets are of two types: *fundamental* facets that define the datatype and *non-fundamental* or *constraining* facets that constrain the permitted values of a datatype.

2.4.1. Fundamental facets

A *fundamental facet* is an abstract property which serves to semantically characterize the values in a value space.

All *fundamental facets* are fully described in § 4.2 – Fundamental Facets on page 40.

2.4.2. Constraining or Non-fundamental facets

A *constraining facet* is an optional property that can be applied to a datatype to constrain its value space.

Constraining the value space consequently constrains the lexical space. Adding constraining facets to a base type is described in § 4.1.2.1 – Derivation by restriction on page 37.

All *constraining facets* are fully described in § 4.3 – Constraining Facets on page 44.

2.5. Datatype dichotomies

It is useful to categorize the datatypes defined in this specification along various dimensions, forming a set of characterization dichotomies.

2.5.1. Atomic vs. list vs. union datatypes

The first distinction to be made is that between atomic, list and union datatypes.

- *Atomic* datatypes are those having values which are regarded by this specification as being indivisible.
- *List* datatypes are those having values each of which consists of a finite-length (possibly empty) sequence of values of an atomic datatype.
- *Union* datatypes are those whose value spaces and lexical spaces are the union of the value spaces and lexical spaces of one or more other datatypes.

For example, a single token which matches Nmtoken from [XML 1.0 (Second Edition)] could be the value of an atomic datatype (); while a sequence of such tokens could be the value of a list datatype ().

2.5.1.1. Atomic datatypes

atomic datatypes can be either primitive or derived. The value space of an atomic datatype is a set of "atomic" values, which for the purposes of this specification, are not further decomposable. The lexical space of an atomic datatype is a set of *literals* whose internal structure is specific to the datatype in question.

2.5.1.2. List datatypes

Several type systems (such as the one described in [ISO 11404]) treat list datatypes as special cases of the more general notions of aggregate or collection datatypes.

list datatypes are always derived. The value space of a list datatype is a set of finite-length sequences of atomic values. The lexical space of a list datatype is a set of literals whose internal structure is a space-separated sequence of literals of the atomic datatype of the items in the list.

The atomic or union datatype that participates in the definition of a list datatype is known as the *itemType* of that list datatype.

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

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

A list datatype can be derived from an atomic datatype whose lexical space allows space (such as or)or a union datatype any of whose 's lexical space allows space. In such a case, regardless of the input, list items will be separated at space boundaries.

```
<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 from a list datatype, the following constraining facets apply:

- length
- maxLength
- minLength
- enumeration
- pattern
- whiteSpace

For each of length, maxLength and minLength, the *unit of 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 its itemType. Hence, any pattern specified when a new datatype is derived from a list datatype is matched against each literal of the list datatype and not against the literals of the datatype that serves as its itemType.

☞
```
<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 for the list datatype is defined as the lexical form in which each item in the list has the canonical lexical representation of its itemType.

2.5.1.3. Union datatypes

The value space and lexical space of a union datatype are the union of the value spaces and lexical spaces of its memberTypes. union datatypes are always derived. Currently, there are no built-in union datatypes.

☞ 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 (greater than 1) of atomic or list datatypes can participate in a union type.

The datatypes that participate in the definition of a union datatype are known as the *memberTypes* of that union datatype.

The order in which the memberTypes are specified in the definition (that is, the order of the <simpleType> children of the <union> element, or the order of the s in the *memberTypes* attribute) is significant. During validation, an element or attribute's value is validated against the memberTypes in the order in which they appear in the definition until a match is found. The evaluation order can be overridden with the use of xsi:type.

☞ For example, given the definition below, the first instance of the <size> element validates correctly as an § 3.3.13 – integer on page 30, the second and third as § 3.2.1 – string on page 11.

```
<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 for a union datatype is defined as the lexical form in which the values have the canonical lexical representation of the appropriate memberTypes.

☞ A datatype which is atomic in this specification need not be an "atomic" datatype in any programming language used to implement this specification. Likewise, a datatype which is a list in this specification need not be a "list" datatype in any programming language used to implement this specification. Furthermore, a datatype which is a union in this specification need not be a "union" datatype in any programming language used to implement this specification.

2.5.2. Primitive vs. derived datatypes

Next, we distinguish between primitive and derived datatypes.

- *Primitive* datatypes are those that are not defined in terms of other datatypes; they exist *ab initio*.
- *Derived* datatypes are those that are defined in terms of other datatypes.

For example, in this specification, is a well-defined mathematical concept that cannot be defined in terms of other datatypes, while a is a special case of the more general datatype .

The simple ur-type definition is a special restriction of the ur-type definition whose name is *anySimpleType* in the XML Schema namespace. *anySimpleType* can be considered as the base type of all primitive datatypes. *anySimpleType* is considered to have an unconstrained lexical space and a value space consisting of the union of the value spaces of all the primitive datatypes and the set of all lists of all members of the value spaces of all the primitive datatypes.

The datatypes defined by this specification fall into both the primitive and derived categories. It is felt that a judiciously chosen set of primitive datatypes will serve the widest possible audience by providing a set of convenient datatypes that can be used as is, as well as providing a rich enough base from which the variety of datatypes needed by schema designers can be derived.

In the example above, is derived from .

A datatype which is primitive in this specification need not be a "primitive" datatype in any programming language used to implement this specification. Likewise, a datatype which is derived in this specification need not be a "derived" datatype in any programming language used to implement this specification.

As described in more detail in § 4.1.2 – XML Representation of Simple Type Definition Schema Components on page 36, each user-derived datatype must be defined in terms of another datatype in one of three ways: 1) by assigning constraining facets which serve to *restrict* the value space of the user-derived datatype to a subset of that of the base type; 2) by creating a list datatype whose value space consists of finite-length sequences of values of its itemType; or 3) by creating a union datatype whose value space consists of the union of the value spaces of its memberTypes.

2.5.2.1. Derived by restriction

A datatype is said to be derived by *restriction* from another datatype 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 its base type.

Every datatype that is derived by *restriction* is defined in terms of an existing datatype, referred to as its *base type*. *base types* can be either primitive or derived.

2.5.2.2. Derived by list

A list datatype can be derived from another datatype (its itemType) by creating a value space that consists of a finite-length sequence of values of its itemType.

2.5.2.3. Derived by union

One datatype can be derived from one or more datatypes by unioning their value spaces and, consequently, their lexical spaces.

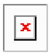
2.5.3. Built-in vs. user-derived datatypes

- *Built-in* datatypes are those which are defined in this specification, and can be either primitive or derived;
- *User-derived* datatypes are those derived datatypes that are defined by individual schema designers.

Conceptually there is no difference between the built-in derived datatypes included in this specification and the user-derived datatypes which will be created by individual schema designers. The built-in derived 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 derived datatypes in this specification serves to demonstrate the mechanics and utility of the datatype generation facilities of this specification.

A datatype which is built-in in this specification need not be a "built-in" datatype in any programming language used to implement this specification. Likewise, a datatype which is user-derived in this specification need not be a "user-derived" datatype in any programming language used to implement this specification.

3. Built-in datatypes



Each built-in datatype in this specification (both primitive and derived) 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 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`

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

This applies to both built-in primitive and built-in derived datatypes.

Each user-derived datatype is also associated with a unique namespace. However, user-derived datatypes do not come from the namespace defined by this specification; rather, they come from the namespace of the schema in which they are defined (see XML Representation of Schemas in [XML Schema Part 1: Structures]).

3.2. Primitive datatypes

The primitive datatypes defined by this specification are described below. For each datatype, the value space and lexical space are defined, constraining facets which apply to the datatype are listed and any datatypes derived from this datatype are specified.

primitive datatypes can only be added by revisions to this specification.

3.2.1. string

The *string* datatype represents character strings in XML. The value space of *string* is the set of finite-length sequences of characters (as defined in [XML 1.0 (Second Edition)]) that match the Char production from [XML 1.0 (Second Edition)]. 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 code point, which is an integer.

☞ Many human languages have writing systems that require child elements for control of aspects such as bidirectional formating 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].

☞ As noted in , the fact that this specification does not specify an order-relation for string does not preclude other applications from treating strings as being ordered.

3.2.1.1. Constraining facets

3.2.1.2. Derived datatypes

3.2.2. boolean

boolean has the value space required to support the mathematical concept of binary-valued logic: {true, false}.

3.2.2.1. Lexical representation

An instance of a datatype that is defined as boolean can have the following legal literals {true, false, 1, 0}.

3.2.2.2. Canonical representation

The canonical representation for *boolean* is the set of literals {true, false}.

3.2.2.3. Constraining facets

3.2.3. decimal

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 multiplying an integer by a non-positive power of ten, i.e., expressible as $i \times 10$^-n where i and n are integers and $n >= 0$. Precision is not reflected in this value space; the number 2.0 is not distinct from the number 2.00. The order-relation on *decimal* is the order relation on real numbers, restricted to this subset.

☞ All minimally conforming processors must support decimal numbers with a minimum of 18 decimal digits (i.e., with a totalDigits of 18). However, minimally conforming processors may set an application-defined limit on the maximum number of decimal digits they are prepared to support, in which case that application-defined maximum number must be clearly documented.

3.2.3.1. Lexical representation

decimal has a lexical representation consisting of a 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`.

3.2.3.2. Canonical representation

The canonical representation for *decimal* is defined by prohibiting certain options from the § 3.2.3.1 – Lexical representation on page 12. Specifically, the preceding optional "+" sign is prohibited. The decimal point is required. 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.

3.2.3.3. Constraining facets

3.2.3.4. Derived datatypes

3.2.4. float

float is patterned after the IEEE single-precision 32-bit floating point type [IEEE 754-1985]. The basic value space of *float* consists of the values *m* × *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 the basic value space described above, the value space of *float* also contains the following three *special values*: positive and negative infinity and not-a-number (NaN). The order-relation on *float* is: *x < y iff y - x* is positive for x and y in the value space. Positive infinity is greater than all other non-NaN values. NaN equals itself but is incomparable with (neither greater than nor less than) any other value in the value space.

☞ "Equality" in this Recommendation is defined to be "identity" (i.e., values that are identical in the value space are equal and vice versa). Identity must be used for the few operations that are defined in this Recommendation. Applications using any of the datatypes defined in this Recommendation may use different definitions of equality for computational purposes; [IEEE 754-1985]-based computation systems are examples. Nothing in this Recommendation should be construed as requiring that such applications use identity as their equality relationship when computing.

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 other *float* values are comparable with it, the result is a value space either having NaN as its only member (the inclusive cases) or that is empty (the exclusive cases). 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.

This datatype differs from that of [IEEE 754-1985] in that there is only one NaN and only one zero. This makes the equality and ordering of values in the data space differ from that of [IEEE 754-1985] only in that for schema purposes NaN = NaN.

A literal in the lexical space representing a decimal number *d* maps to the normalized value in the value space of *float* that is closest to *d* in the sense defined by [Clinger, WD (1990)]; if *d* is exactly halfway between two such values then the even value is chosen.

3.2.4.1. Lexical representation

float values have a lexical representation consisting of a mantissa followed, optionally, by the character "E" or "e", followed by an exponent. The exponent must be an . The mantissa must be a number. The representations for exponent and mantissa must follow the lexical rules for and . If the "E" or "e" and the following exponent are omitted, an exponent value of 0 is assumed.

The *special values* positive and negative infinity and not-a-number have lexical representations `INF`, `-INF` and `NaN`, respectively. Lexical representations for zero may take a positive or negative sign.

For example, `-1E4, 1267.43233E12, 12.78e-2, 12 , -0, 0` and `INF` are all legal literals for *float*.

3.2.4.2. Canonical representation

The canonical representation for *float* is defined by prohibiting certain options from the § 3.2.4.1 – Lexical representation on page 13. Specifically, the exponent must be indicated by "E". Leading zeroes and the preceding optional "+" sign are prohibited in the exponent. If the exponent is zero, it must be indicated by "E0". For the mantissa, the preceding optional "+" sign is prohibited and the decimal point is required. Leading and trailing zeroes are prohibited subject to the following: number representations must be normalized such that there is a single digit which is non-zero to the left of the decimal point and at least a single digit to the right of the decimal point unless the value being represented is zero. The canonical representation for zero is 0.0E0.

3.2.4.3. Constraining facets

3.2.5. double

The *double* datatype is patterned after the IEEE double-precision 64-bit floating point type [IEEE 754-1985]. The basic value space of *double* consists of the values $m \times 2^e$, where *m* is an integer whose absolute value is less than 2^{53}, and *e* is an integer between -1075 and 970, inclusive. In addition to the basic value space described above, the value space of *double* also contains the following three *special values*: positive and negative infinity and not-a-number (NaN). The order-relation on *double* is: $x < y$ *iff* $y - x$ is positive for x and y in the value space. Positive infinity is greater than all other non-NaN values. NaN equals itself but is incomparable with (neither greater than nor less than) any other value in the value space.

☞ "Equality" in this Recommendation is defined to be "identity" (i.e., values that are identical in the value space are equal and vice versa). Identity must be used for the few operations that are defined in this Recommendation. Applications using any of the datatypes defined in this Recommendation may use different definitions of equality for computational purposes; [IEEE 754-1985]-based computation systems are examples. Nothing in this Recommendation should be construed as requiring that such applications use identity as their equality relationship when computing.

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 other *double* values are comparable with it, the result is a value space either having NaN as its only member (the inclusive cases) or that is empty (the exclusive cases). 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.

This datatype differs from that of [IEEE 754-1985] in that there is only one NaN and only one zero. This makes the equality and ordering of values in the data space differ from that of [IEEE 754-1985] only in that for schema purposes NaN = NaN.

A literal in the lexical space representing a decimal number *d* maps to the normalized value in the value space of *double* that is closest to *d*; if *d* is exactly halfway between two such values then the even value is chosen. This is the *best approximation* of *d* ([Clinger, WD (1990)], [Gay, DM (1990)]), which is more accurate than the mapping required by [IEEE 754-1985].

3.2.5.1. Lexical representation

double values have a lexical representation consisting of a mantissa followed, optionally, by the character "E" or "e", followed by an exponent. The exponent must be an integer. The mantissa must be a decimal number. The representations for exponent and mantissa must follow the lexical rules for and . If the "E" or "e" and the following exponent are omitted, an exponent value of 0 is assumed.

The *special values* positive and negative infinity and not-a-number have lexical representations `INF`, `-INF` and `NaN`, respectively. Lexical representations for zero may take a positive or negative sign.

For example, `-1E4, 1267.43233E12, 12.78e-2, 12 , -0, 0` and `INF` are all legal literals for *double*.

3.2.5.2. Canonical representation

The canonical representation for *double* is defined by prohibiting certain options from the § 3.2.5.1 – Lexical representation on page 14. Specifically, the exponent must be indicated by "E". Leading zeroes and the preceding optional "+" sign are prohibited in the exponent. If the exponent is zero, it must be indicated by "E0". For the mantissa, the preceding optional "+" sign is prohibited and the decimal point is required. Leading and trailing zeroes are prohibited subject to the following: number representations must be normalized such that there is a single digit which is non-zero to the left of the decimal point and at least a single digit to the right of the decimal point unless the value being represented is zero. The canonical representation for zero is 0.0E0.

3.2.5.3. Constraining facets

3.2.6. duration

duration represents a duration of time. The value space of *duration* is a six-dimensional space where the coordinates designate the Gregorian year, month, day, hour, minute, and second components defined in § 5.5.3.2 of [ISO 8601], respectively. These components are ordered in their significance by their order of appearance i.e. as year, month, day, hour, minute, and second.

☞ All minimally conforming processors must support year values with a minimum of 4 digits (i.e., `YYYY`) and a minimum fractional second precision of milliseconds or three decimal digits (i.e. `s.sss`). However, minimally conforming processors may set an application-defined limit on the maximum number of digits they are prepared to support in these two cases, in which case that application-defined maximum number must be clearly documented.

3.2.6.1. Lexical representation

The lexical representation for *duration* is the [ISO 8601] extended format P*n*Y*n* M*n*DT*n*H *n*M*n*S, where *n*Y represents the number of years, *n*M the number of months, *n*D the number of days, 'T' is the date/time separator, *n*H the number of hours, *n*M the number of minutes and *n*S the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

The values of the Year, Month, Day, Hour and Minutes components are not restricted but allow an arbitrary unsigned integer, i.e., an integer that conforms to the pattern `[0-9]+`.. Similarly, the value of the Seconds component allows an arbitrary unsigned decimal. Following [ISO 8601], at least one digit must follow the decimal point if it appears. That is, the value of the Seconds component must conform to the pattern `[0-`

`9]+(\.[0-9]+)?`. Thus, the lexical representation of *duration* does not follow the alternative format of § 5.5.3.2.1 of [ISO 8601].

An optional preceding minus sign ('-') is allowed, to indicate a negative duration. If the sign is omitted a positive duration is indicated. See also Appendix D – ISO 8601 Date and Time Formats on page 59.

For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, one would write: `P1Y2M3DT10H30M`. One could also indicate a duration of minus 120 days as: `-P120D`.

Reduced precision and truncated representations of this format are allowed provided they conform to the following:

- If the number of years, months, days, hours, minutes, or seconds in any expression equals zero, the number and its corresponding designator may be omitted. However, at least one number and its designator must be present.
- The seconds part may have a decimal fraction.
- The designator 'T' must be absent if and only if all of the time items are absent. The designator 'P' must always be present.

For example, P1347Y, P1347M and P1Y2MT2H are all allowed; P0Y1347M and P0Y1347M0D are allowed. P-1347M is not allowed although -P1347M is allowed. P1Y2MT is not allowed.

3.2.6.2. Order relation on duration

In general, the order-relation on *duration* is a partial order since there is no determinate relationship between certain durations such as one month (P1M) and 30 days (P30D). The order-relation of two *duration* values x and y is $x < y$ *iff* $s+x < s+y$ for each qualified s in the list below. These values for s cause the greatest deviations in the addition of dateTimes and durations. Addition of durations to time instants is defined in Appendix E – Adding durations to dateTimes on page 61.

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

The following table shows the strongest relationship that can be determined between example durations. The symbol <> means that the order relation is indeterminate. Note that because of leap-seconds, a seconds field can vary from 59 to 60. However, because of the way that addition is defined in Appendix E – Adding durations to dateTimes on page 61, they are still totally ordered.

	Relation					
P1Y	> P364D	<> P365D			<> P366D	< P367D
P1M	> P27D	<> P28D	<> P29D	<> P30D	<> P31D	< P32D
P5M	> P149D	<> P150D	<> P151D	<> P152D	<> P153D	< P154D

Implementations are free to optimize the computation of the ordering relationship. For example, the following table can be used to compare durations of a small number of months against days.

	Months	1	2	3	4	5	6	7	8	9	10	11	12	13	...
Days	**Minimum**	28	59	89	120	150	181	212	242	273	303	334	365	393	...
	Maximum	31	62	92	123	153	184	215	245	276	306	337	366	397	...

3.2.6.3. Facet Comparison for durations

In comparing *duration* values with , , and facet values indeterminate comparisons should be considered as "false".

3.2.6.4. Totally ordered durations

Certain derived datatypes of durations can be guaranteed have a total order. For this, they must have fields from only one row in the list below and the time zone must either be required or prohibited.

- year, month
- day, hour, minute, second

For example, a datatype could be defined to correspond to the [SQL] datatype Year-Month interval that required a four digit year field and a two digit month field but required all other fields to be unspecified. This datatype could be defined as below and would have a total order.

```
<simpleType name='SQL-Year-Month-Interval'>
    <restriction base='duration'>
      <pattern value='P\p{Nd}{4}Y\p{Nd}{2}M'/>
    </restriction>
</simpleType>
```

3.2.6.5. Constraining facets

3.2.7. dateTime

dateTime values may be viewed as objects with integer-valued year, month, day, hour and minute properties, a decimal-valued second property, and a boolean timezoned property. Each such object also has one decimal-valued method or computed property, timeOnTimeline, whose value is always a decimal number; the values are dimensioned in seconds, the integer 0 is 0001-01-01T00:00:00 and the value of timeOnTimeline for other *dateTime* values is computed using the Gregorian algorithm as modified for leap-seconds. The timeOnTimeline values form two related "timelines", one for timezoned values and one for non-timezoned values. Each timeline is a copy of the value space of , with integers given units of seconds.

The value space of *dateTime* is closely related to the dates and times described in ISO 8601. For clarity, the text above specifies a particular origin point for the timeline. It should be noted, however, that schema processors need not expose the timeOnTimeline value to schema users, and there is no requirement that a timeline-based implementation use the particular origin described here in its internal representation. Other interpretations of the value space which lead to the same results (i.e., are isomorphic) are of course acceptable.

All timezoned times are Coordinated Universal Time (UTC, sometimes called "Greenwich Mean Time"). Other timezones indicated in lexical representations are converted to UTC during conversion of literals to values. "Local" or untimezoned times are presumed to be the time in the timezone of some unspecified locality as prescribed by the appropriate legal authority; currently there are no legally prescribed timezones which are durations whose magnitude is greater than 14 hours. The value of each numeric-valued property (other than timeOnTimeline) is limited to the maximum value within the interval determined by the next-higher property. For example, the day value can never be 32, and cannot even be 29 for month 02 and year 2002 (February 2002).

☞ The date and time datatypes described in this recommendation were inspired by [ISO 8601]. '0001' is the lexical representation of the year 1 of the Common Era (1 CE, sometimes written "AD 1" or "1 AD"). There is no year 0, and '0000' is not a valid lexical representation. '-0001' is the lexical representation of the year 1 Before Common Era (1 BCE, sometimes written "1 BC").

Those using this (1.0) version of this Recommendation to represent negative years should be aware that the interpretation of lexical representations beginning with a `'-'` is likely to change in subsequent versions.

[ISO 8601] makes no mention of the year 0; in [ISO 8601:1998 Draft Revision] the form '0000' was disallowed and this recommendation disallows it as well. However, [ISO 8601:2000 Second Edition], which became available just as we were completing version 1.0, allows the form '0000', representing the year 1 BCE. A number of external commentators have also suggested that '0000' be allowed, as the lexical representation for 1 BCE, which is the normal usage in astronomical contexts. It is the intention of the XML Schema Working Group to allow '0000' as a lexical representation in the *dateTime*, *date*, *gYear*, and *gYearMonth* datatypes in a subsequent version of this Recommendation. '0000' will be the lexical representation of 1 BCE (which is a leap year), '-0001' will become the lexical representation of 2 BCE (not 1 BCE as in this (1.0) version), '-0002' of 3 BCE, etc.

☞ See the conformance note in All processors support year values with a minimum of 4 digits (i.e., YYYY) and a minimum fractional second precision of milliseconds or three decimal digits (i.e. s.sss). However, processors set an application-defined limit on the maximum number of digits they are prepared to support in these two cases, in which case that application-defined maximum number be clearly documented. which applies to this datatype as well.

3.2.7.1. Lexical representation

The lexical space of *dateTime* consists of finite-length sequences of characters of the form: `'-'? yyyy '-' mm '-' dd 'T' hh ':' mm ':' ss ('.' s+)? (zzzzzz)?`, where

- '-'? *yyyy* is a four-or-more digit optionally negative-signed numeral that represents the year; if more than four digits, leading zeros are prohibited, and '0000' is prohibited (see the Note above The date and time datatypes described in this recommendation were inspired by . '0001' is the lexical representation of the year 1 of the Common Era (1 CE, sometimes written "AD 1" or "1 AD"). There is no year 0, and '0000' is not a valid lexical representation. '-0001' is the lexical representation of the year 1 Before Common Era (1 BCE, sometimes written "1 BC"). Those using this (1.0) version of this Recommendation to represent negative years should be aware that the interpretation of lexical representations beginning with a '-' is likely to change in subsequent versions. makes no mention of the year 0; in the form '0000' was disallowed and this recommendation disallows it as well. However, , which became available just as we were completing version 1.0, allows the form '0000', representing the year 1 BCE. A number of external commentators have also suggested that '0000' be allowed, as the lexical representation for 1 BCE, which is the normal usage in astronomical contexts. It is the intention of the XML Schema Working Group to allow '0000' as a lexical representation in the dateTime, date, gYear, and gYearMonth datatypes in a subsequent version of this Recommendation. '0000' will be the lexical representation of 1 BCE (which is a leap year), '-0001' will become the lexical representation of 2 BCE (not 1 BCE as in this (1.0) version), '-0002' of 3 BCE, etc. ; also note that a plus sign is not permitted);
- the remaining '-'s are separators between parts of the date portion;
- the first *mm* is a two-digit numeral that represents the month;
- *dd* is a two-digit numeral that represents the day;

- 'T' is a separator indicating that time-of-day follows;
- *hh* is a two-digit numeral that represents the hour; '24' is permitted if the minutes and seconds represented are zero, and the *dateTime* value so represented is the first instant of the following day (the hour property of a *dateTime* object in the value space cannot have a value greater than 23);
- ':' is a separator between parts of the time-of-day portion;
- the second *mm* is a two-digit numeral that represents the minute;
- *ss* is a two-integer-digit numeral that represents the whole seconds;
- '.' *s+* (if present) represents the fractional seconds;
- *zzzzzz* (if present) represents the timezone (as described below).

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 2002-10-10T17:00:00Z, five hours later than 2002-10-10T12:00:00Z.

For further guidance on arithmetic with *dateTime*s and durations, see Appendix E – Adding durations to dateTimes on page 61.

3.2.7.2. Canonical representation

Except for trailing fractional zero digits in the seconds representation, '24:00:00' time representations, and timezone (for timezoned values), the mapping from literals to values is one-to-one. Where there is more than one possible representation, the canonical representation is as follows:

- The 2-digit numeral representing the hour must not be '`24`';
- The fractional second string, if present, must not end in '`0`';
- for timezoned values, the timezone must be represented with '`Z`' (All timezoned *dateTime* values are UTC.).

3.2.7.3. Timezones

Timezones are durations with (integer-valued) hour and minute properties (with the hour magnitude limited to at most 14, and the minute magnitude limited to at most 59, except that if the hour magnitude is 14, the minute value must be 0); they may be both positive or both negative.

The lexical representation of a timezone is a string of the form: `(('+' | '-') hh ':' mm) | 'Z'`, where

- *hh* is a two-digit numeral (with leading zeros as required) that represents the hours,
- *mm* is a two-digit numeral that represents the minutes,
- '+' indicates a nonnegative duration,
- '-' indicates a nonpositive duration.

The mapping so defined is one-to-one, except that '+00:00', '-00:00', and 'Z' all represent the same zero-length duration timezone, UTC; 'Z' is its canonical representation.

When a timezone is added to a UTC *dateTime*, the result is the date and time "in that timezone". For example, 2002-10-10T12:00:00+05:00 is 2002-10-10T07:00:00Z and 2002-10-10T00:00:00+05:00 is 2002-10-09T19:00:00Z.

3.2.7.4. Order relation on dateTime

dateTime value objects on either timeline are totally ordered by their timeOnTimeline values; between the two timelines, *dateTime* value objects are ordered by their timeOnTimeline values when their timeOnTimeline values differ by more than fourteen hours, with those whose difference is a duration of 14 hours or less being incomparable.

In general, the order-relation on *dateTime* is a partial order since there is no determinate relationship between certain instants. For example, there is no determinate ordering between (a) 2000-01-20T12:00:00 and (b) 2000-01-20T12:00:00Z. Based on timezones currently in use, (c) could vary from 2000-01-20T12:00:00+12:00 to 2000-01-20T12:00:00-13:00. It is, however, possible for this range to expand or contract in the future, based on local laws. Because of this, the following definition uses a somewhat broader range of indeterminate values: +14:00..-14:00.

The following definition uses the notation S[year] to represent the year field of S, S[month] to represent the month field, and so on. The notation (Q & "-14:00") means adding the timezone -14:00 to Q, where Q did not already have a timezone. *This is a logical explanation of the process. Actual implementations are free to optimize as long as they produce the same results.*

The ordering between two *dateTime*s P and Q is defined by the following algorithm:

A.Normalize P and Q. That is, if there is a timezone present, but it is not Z, convert it to Z using the addition operation defined in Appendix E – Adding durations to dateTimes on page 61

- Thus 2000-03-04T23:00:00+03:00 normalizes to 2000-03-04T20:00:00Z

B. If P and Q either both have a time zone or both do not have a time zone, compare P and Q field by field from the year field down to the second field, and return a result as soon as it can be determined. That is:

1. For each i in {year, month, day, hour, minute, second}
    A. If P[i] and Q[i] are both not specified, continue to the next i
    B. If P[i] is not specified and Q[i] is, or vice versa, stop and return P <> Q
    C. If P[i] < Q[i], stop and return P < Q
    D. If P[i] > Q[i], stop and return P > Q
2. Stop and return P = Q

C.Otherwise, if P contains a time zone and Q does not, compare as follows:

1. P < Q if P < (Q with time zone +14:00)
2. P > Q if P > (Q with time zone -14:00)
3. P <> Q otherwise, that is, if (Q with time zone +14:00) < P < (Q with time zone -14:00)

D. Otherwise, if P does not contain a time zone and Q does, compare as follows:

1. P < Q if (P with time zone -14:00) < Q.
2. P > Q if (P with time zone +14:00) > Q.
3. P <> Q otherwise, that is, if (P with time zone +14:00) < Q < (P with time zone -14:00)

Examples:

Determinate	Indeterminate
2000-01-15T00:00:00 < 2000-02-15T00:00:00	2000-01-01T12:00:00 <> 1999-12-31T23:00:00Z
2000-01-15T12:00:00 < 2000-01-16T12:00:00Z	2000-01-16T12:00:00 <> 2000-01-16T12:00:00Z
	2000-01-16T00:00:00 <> 2000-01-16T12:00:00Z

3.2.7.5. Totally ordered dateTimes

Certain derived types from *dateTime* can be guaranteed have a total order. To do so, they must require that a specific set of fields are always specified, and that remaining fields (if any) are always unspecified. For example, the date datatype without time zone is defined to contain exactly year, month, and day. Thus dates without time zone have a total order among themselves.

3.2.7.6. Constraining facets

3.2.8. time

time represents an instant of time that recurs every day. The value space of *time* is the space of *time of day* values as defined in § 5.3 of [ISO 8601]. Specifically, it is a set of zero-duration daily time instances.

Since the lexical representation allows an optional time zone indicator, *time* values are partially ordered because it may not be able to determine the order of two values one of which has a time zone and the other does not. The order relation on *time* values is the § 3.2.7.4 – Order relation on dateTime on page 19 using an arbitrary date. See also Appendix E – Adding durations to dateTimes on page 61. Pairs of *time* values with or without time zone indicators are totally ordered.

See the conformance note in All processors support year values with a minimum of 4 digits (i.e., YYYY) and a minimum fractional second precision of milliseconds or three decimal digits (i.e. s.sss). However, processors set an application-defined limit on the maximum number of digits they are prepared to support in these two cases, in which case that application-defined maximum number be clearly documented. which applies to the seconds part of this datatype as well.

3.2.8.1. Lexical representation

The lexical representation for *time* is the left truncated lexical representation for : hh:mm:ss.sss with optional following time zone indicator. For example, to indicate 1:20 pm for Eastern Standard Time which is 5 hours behind Coordinated Universal Time (UTC), one would write: 13:20:00-05:00. See also Appendix D – ISO 8601 Date and Time Formats on page 59.

3.2.8.2. Canonical representation

The canonical representation for *time* is defined by prohibiting certain options from the § 3.2.8.1 – Lexical representation on page 20. Specifically, either the time zone must be omitted or, if present, the time zone must be Coordinated Universal Time (UTC) indicated by a "Z". Additionally, the canonical representation for midnight is 00:00:00.

3.2.8.3. Constraining facets

3.2.9. date

The value space of *date* consists of top-open intervals of exactly one day in length on the timelines of , beginning on the beginning moment of each day (in each timezone), i.e. '00:00:00', up to but not including '24:00:00' (which is identical with '00:00:00' 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.

A "date object" is an object with year, month, and day properties just like those of objects, plus an optional *timezone-valued* timezone property. (As with values of timezones are a special case of durations.) Just as a object corresponds to a point on one of the timelines, a *date* object corresponds to an interval on one of the two timelines as just described.

Timezoned *date* values track the starting moment of their day, as determined by their timezone; said timezone is generally recoverable for canonical representations. The *recoverable timezone* is that duration which is the result of subtracting the first moment (or any moment) of the timezoned *date* from the first moment (or the corresponding moment) UTC on the same *date*. recoverable timezones are always durations between '+12:00' and '-11:59'. This "timezone normalization" (which follows automatically from the definition of the *date* value space) is explained more in § 3.2.9.1 – Lexical representation on page 21.

For example: the first moment of 2002-10-10+13:00 is 2002-10-10T00:00:00+13, which is 2002-10-09T11:00:00Z, which is also the first moment of 2002-10-09-11:00. Therefore 2002-10-10+13:00 is 2002-10-09-11:00; *they are the same interval*.

☞ For most timezones, either the first moment or last moment of the day (a value, always UTC) will have a *date* portion different from that of the *date* itself! However, noon of that *date* (the midpoint of the interval) in that (normalized) timezone will always have the same *date* portion as the *date* itself, even when that noon point in time is normalized to UTC. For example, 2002-10-10-05:00 begins during 2002-10-09Z and 2002-10-10+05:00 ends during 2002-10-11Z, but noon of both 2002-10-10-05:00 and 2002-10-10+05:00 falls in the interval which is 2002-10-10Z.

☞ See the conformance note in All processors support year values with a minimum of 4 digits (i.e., YYYY) and a minimum fractional second precision of milliseconds or three decimal digits (i.e. s.sss). However, processors set an application-defined limit on the maximum number of digits they are prepared to support in these two cases, in which case that application-defined maximum number be clearly documented. which applies to the year part of this datatype as well.

3.2.9.1. Lexical representation

For the following discussion, let the "date portion" of a or *date* object be an object similar to a or *date* object, with similar year, month, and day properties, but no others, having the same value for these properties as the original or *date* object.

The lexical space of *date* consists of finite-length sequences of characters of the form: `'-'? yyyy '-' mm '-' dd zzzzzz?` where the *date* and optional timezone are represented exactly the same way as they are for . The first moment of the interval is that represented by: `'-' yyyy '-' mm '-' dd 'T00:00:00' zzzzzz?` and the least upper bound of the interval is the timeline point represented (noncanonically) by: `'-' yyyy '-' mm '-' dd 'T24:00:00' zzzzzz?`.

☞ The recoverable timezone of a *date* will always be a duration between '+12:00' and '11:59'. Timezone lexical representations, as explained for , can range from '+14:00' to '-14:00'. The result is that literals of *date*s with very large or very negative timezones will map to a "normalized" *date* value with a recoverable timezone different from that represented in the original representation, and a matching difference of +/- 1 day in the *date* itself.

3.2.9.2. Canonical representation

Given a member of the *date* value space, the *date* portion of the canonical representation (the entire representation for nontimezoned values, and all but the timezone representation for timezoned values) is always the *date* portion of the canonical representation of the interval midpoint (the representation, truncated on the right to eliminate 'T' and all following characters). For timezoned values, append the canonical representation of the recoverable timezone.

3.2.10. gYearMonth

gYearMonth represents a specific gregorian month in a specific gregorian year. The value space of *gYearMonth* is the set of Gregorian calendar months as defined in § 5.2.1 of [ISO 8601]. Specifically, it is a set of one-month long, non-periodic instances e.g. 1999-10 to represent the whole month of 1999-10, independent of how many days this month has.

Since the lexical representation allows an optional time zone indicator, *gYearMonth* values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If *gYearMonth* values are considered as periods of time, the order relation on *gYearMonth* values is the order relation on their starting instants. This is discussed in § 3.2.7.4 – Order relation on dateTime on page 19. See also Appendix E – Adding durations to dateTimes on page 61. Pairs of *gYearMonth* values with or without time zone indicators are totally ordered.

☞ 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.

☞ See the conformance note in All processors support year values with a minimum of 4 digits (i.e., YYYY) and a minimum fractional second precision of milliseconds or three decimal digits (i.e. s.sss). However, processors set an application-defined limit on the maximum number of digits they are prepared to support in these two cases, in which case that application-defined maximum number be clearly documented. which applies to the year part of this datatype as well.

3.2.10.1. Lexical representation

The lexical representation for *gYearMonth* is the reduced (right truncated) lexical representation for : CCYY-MM. No left truncation is allowed. An optional following time zone qualifier is allowed. To accommodate year values outside the range from 0001 to 9999, additional digits can be added to the left of this representation and a preceding "-" sign is allowed.

For example, to indicate the month of May 1999, one would write: 1999-05. See also Appendix D – ISO 8601 Date and Time Formats on page 59.

3.2.10.2. Constraining facets

3.2.11. gYear

gYear represents a gregorian calendar year. The value space of *gYear* is the set of Gregorian calendar years as defined in § 5.2.1 of [ISO 8601]. Specifically, it is a set of one-year long, non-periodic instances e.g. lexical 1999 to represent the whole year 1999, independent of how many months and days this year has.

Since the lexical representation allows an optional time zone indicator, *gYear* values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If *gYear* values are considered as periods of time, the order relation on *gYear* values is the order relation on their starting instants. This is discussed in § 3.2.7.4 – Order relation on dateTime on page 19. See also Appendix E – Adding durations to dateTimes on page 61. Pairs of *gYear* values with or without time zone indicators are totally ordered.

☞ 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.

☞ See the conformance note in All processors support year values with a minimum of 4 digits (i.e., YYYY) and a minimum fractional second precision of milliseconds or three decimal digits (i.e. s.sss). However, processors set an application-defined limit on the maximum number of digits they are prepared to support in these two cases, in which case that application-defined maximum number be clearly documented. which applies to the year part of this datatype as well.

3.2.11.1. Lexical representation

The lexical representation for *gYear* is the reduced (right truncated) lexical representation for : CCYY. No left truncation is allowed. An optional following time zone qualifier is allowed as for . To accommodate year values outside the range from 0001 to 9999, additional digits can be added to the left of this representation and a preceding "-" sign is allowed.

For example, to indicate 1999, one would write: 1999. See also Appendix D – ISO 8601 Date and Time Formats on page 59.

3.2.11.2. Constraining facets

3.2.12. gMonthDay

gMonthDay is a gregorian date that recurs, specifically a day of the year such as the third of May. Arbitrary recurring dates are not supported by this datatype. The value space of *gMonthDay* is the set of *calendar dates*, as defined in § 3 of [ISO 8601]. Specifically, it is a set of one-day long, annually periodic instances.

Since the lexical representation allows an optional time zone indicator, *gMonthDay* values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If *gMonthDay* values are considered as periods of time, in an arbitrary leap year, the order relation on *gMonthDay* values is the order relation on their starting instants. This is discussed in § 3.2.7.4 – Order relation on dateTime on page 19. See also Appendix E – Adding durations to dateTimes on page 61. Pairs of *gMonthDay* values with or without time zone indicators are totally ordered.

☞ 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.

3.2.12.1. Lexical representation

The lexical representation for *gMonthDay* is the left truncated lexical representation for : --MM-DD. An optional following time zone qualifier is allowed as for . No preceding sign is allowed. No other formats are allowed. See also Appendix D – ISO 8601 Date and Time Formats on page 59.

This datatype can be used to represent a specific day in a month. To say, for example, that my birthday occurs on the 14th of September ever year.

3.2.12.2. Constraining facets

3.2.13. gDay

gDay is a gregorian day that recurs, specifically a day of the month such as the 5th of the month. Arbitrary recurring days are not supported by this datatype. The value space of *gDay* is the space of a set of *calendar dates* as defined in § 3 of [ISO 8601]. Specifically, it is a set of one-day long, monthly periodic instances.

This datatype can be used to represent a specific day of the month. To say, for example, that I get my paycheck on the 15th of each month.

Since the lexical representation allows an optional time zone indicator, *gDay* values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If *gDay* values are considered as periods of time, in an arbitrary month that has 31 days, the order relation on *gDay* values is the order relation on their starting instants. This is discussed in § 3.2.7.4 – Order relation on dateTime on page 19. See also Appendix E – Adding durations to dateTimes on page 61. Pairs of *gDay* values with or without time zone indicators are totally ordered.

☞ Because days in one calendar only rarely correspond to days 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.

3.2.13.1. Lexical representation

The lexical representation for *gDay* is the left truncated lexical representation for : ---DD . An optional following time zone qualifier is allowed as for . No preceding sign is allowed. No other formats are allowed. See also Appendix D – ISO 8601 Date and Time Formats on page 59.

3.2.13.2. Constraining facets

3.2.14. gMonth

gMonth is a gregorian month that recurs every year. The value space of *gMonth* is the space of a set of *calendar months* as defined in § 3 of [ISO 8601]. Specifically, it is a set of one-month long, yearly periodic instances.

This datatype can be used to represent a specific month. To say, for example, that Thanksgiving falls in the month of November.

Since the lexical representation allows an optional time zone indicator, *gMonth* values are partially ordered because it may not be possible to unequivocally determine the order of two values one of which has a time zone and the other does not. If *gMonth* values are considered as periods of time, the order relation on *gMonth* is the order relation on their starting instants. This is discussed in § 3.2.7.4 – Order relation on dateTime on page 19. See also Appendix E – Adding durations to dateTimes on page 61. Pairs of *gMonth* values with or without time zone indicators are totally ordered.

☞ 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.

3.2.14.1. Lexical representation

The lexical representation for *gMonth* is the left and right truncated lexical representation for : --MM. An optional following time zone qualifier is allowed as for . No preceding sign is allowed. No other formats are allowed. See also Appendix D – ISO 8601 Date and Time Formats on page 59.

3.2.14.2. Constraining facets

3.2.15. hexBinary

hexBinary represents arbitrary hex-encoded binary data. The value space of *hexBinary* is the set of finite-length sequences of binary octets.

3.2.15.1. Lexical Representation

hexBinary has a lexical representation where each binary octet is encoded as a character tuple, consisting of two hexadecimal digits ([0-9a-fA-F]) representing the octet code. For example, "0FB7" is a *hex* encoding for the 16-bit integer 4023 (whose binary representation is 111110110111).

3.2.15.2. Canonical Representation

The canonical representation for *hexBinary* is defined by prohibiting certain options from the § 3.2.15.1 – Lexical Representation on page 25. Specifically, the lower case hexadecimal digits ([a-f]) are not allowed.

3.2.15.3. Constraining facets

3.2.16. base64Binary

base64Binary represents Base64-encoded arbitrary binary data. The value space of *base64Binary* is the set of finite-length sequences of binary octets. For *base64Binary* data the entire binary stream is encoded using the Base64 Alphabet in [RFC 2045].

The lexical forms of *base64Binary* values are limited to the 65 characters of the Base64 Alphabet defined in [RFC 2045], i.e., `a-z`, `A-Z`, `0-9`, the plus sign (+), the forward slash (/) and the equal sign (=), together with the characters defined in [XML 1.0 (Second Edition)] as white space. 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 mandated in the lexical forms of *base64Binary* data and must not be enforced by XML Schema processors.

The lexical space of *base64Binary* is given by the following grammar (the notation is that used in [XML 1.0 (Second Edition)]); legal lexical forms must match the *Base64Binary* production.

```
Base64Binary  ::=  ((B64S B64S B64S B64S)*                              ((B64S
B64S B64S B64) |                               (B64S B64S B16S '=') |
                      (B64S B04S '=' #x20? '=')))?B64S           ::= B64
#x20?  B16S             ::= B16 #x20? B04S               ::= B04 #x20?
```

```
B04          ::=   [AQgw]  B16          ::=   [AEIMQUYcgkosw048]
B64        ::=  [A-Za-z0-9+/]
```

Note that this grammar requires the number of non-whitespace characters in the lexical form to be a multiple of four, and for equals signs to appear only at the end of the lexical form; strings which do not meet these constraints are not legal lexical forms of *base64Binary* because they cannot successfully be decoded by base64 decoders.

☞ The above definition of the lexical space is more restrictive than that given in [RFC 2045] as regards whitespace -- this is not an issue in practice. Any string compatible with the 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 canonical lexical form of a *base64Binary* data value is the base64 encoding of the value which matches the Canonical-base64Binary production in the following grammar:

```
Canonical-base64Binary      ::=      (B64   B64   B64   B64)*
                              ((B64 B64 B16 '=') | (B64 B04 '=='))?
```

☞ For some values the canonical form defined above does not conform to [RFC 2045], which requires breaking with linefeeds at appropriate intervals.

The length of a *base64Binary* value is the number of octets it contains. This may be calculated from the lexical form 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] explicitly references US-ASCII 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 1.0 (Second Edition)]

3.2.16.1. Constraining facets

3.2.17. anyURI

anyURI represents a Uniform Resource Identifier Reference (URI). An *anyURI* value can be absolute or relative, and may have an optional fragment identifier (i.e., it may be a URI Reference). This type should be used to specify the intention that the value fulfills the role of a URI as defined by [RFC 2396], as amended by [RFC 2732].

The mapping from *anyURI* values to URIs is as defined by the URI reference escaping procedure defined in Section 5.4 Locator Attribute of [XML Linking Language] (see also Section 8 Character Encoding in URI References of [Character Model]). 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 2396], as amended by [RFC 2732], where appropriate to identify resources.

☞ Section 5.4 Locator Attribute of [XML Linking Language] requires that relative URI references be absolutized as defined in [XML Base] before use. This is an XLink-specific requirement and is not appropriate for XML Schema, since neither the lexical space nor the value space of the type are restricted to absolute URIs. Accordingly absolutization must not be performed by schema processors as part of schema validation.

☞ 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, this specification follows the lead of [RFC 2396] (as amended by [RFC 2732]) in this matter: such rules and restrictions are not part of type validity and are not checked by minimally conforming processors. Thus in practice the above definition imposes only very modest obligations on minimally conforming processors.

3.2.17.1. Lexical representation

The lexical space of *anyURI* is finite-length character sequences which, when the algorithm defined in Section 5.4 of [XML Linking Language] is applied to them, result in strings which are legal URIs according to [RFC 2396], as amended by [RFC 2732].

☞ Spaces are, in principle, allowed in the lexical space of *anyURI*, however, their use is highly discouraged (unless they are encoded by %20).

3.2.17.2. Constraining facets

3.2.18. QName

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

☞ The mapping between literals in the lexical space and values in the value space of *QName* requires a namespace declaration to be in scope for the context in which *QName* is used.

3.2.18.1. Constraining facets

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

3.2.19. NOTATION

NOTATION represents the NOTATION attribute type from [XML 1.0 (Second Edition)]. The value space of *NOTATION* is the set of s 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 s).

cos: enumeration facet value required for NOTATION

It is an error for *NOTATION* to be used directly in a schema. Only datatypes that are derived from *NOTATION* by specifying a value for enumeration can be used in a schema.

For compatibility (see § 1.4 – Terminology on page 2) *NOTATION* should be used only on attributes and should only be used in schemas with no target namespace.

3.2.19.1. Constraining facets

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

3.3. Derived datatypes

This section gives conceptual definitions for all built-in derived datatypes defined by this specification. The XML representation used to define derived datatypes (whether built-in or user-derived) is given in section § 4.1.2 – XML Representation of Simple Type Definition Schema Components on page 36 and the complete definitions of the built-in derived datatypes are provided in Appendix A Appendix A – Schema for Datatype Definitions (normative) on page 59.

3.3.1. normalizedString

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 .

3.3.1.1. Constraining facets

3.3.1.2. Derived datatypes

3.3.2. token

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 .

3.3.2.1. Constraining facets

3.3.2.2. Derived datatypes

3.3.3. language

language represents natural language identifiers as defined by by [RFC 3066] . The value space of *language* is the set of all strings that are valid language identifiers as defined [RFC 3066] . The lexical space of *language* is the set of all strings that conform to the pattern `[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*` . The base type of *language* is .

3.3.3.1. Constraining facets

3.3.4. NMTOKEN

NMTOKEN represents the NMTOKEN attribute type from [XML 1.0 (Second Edition)]. The value space of *NMTOKEN* is the set of tokens that match the Nmtoken production in [XML 1.0 (Second Edition)]. The lexical space of *NMTOKEN* is the set of strings that match the Nmtoken production in [XML 1.0 (Second Edition)]. The base type of *NMTOKEN* is .

For compatibility (see § 1.4 – Terminology on page 2) *NMTOKEN* should be used only on attributes.

3.3.4.1. Constraining facets

3.3.4.2. Derived datatypes

3.3.5. NMTOKENS

NMTOKENS represents the NMTOKENS attribute type from [XML 1.0 (Second Edition)]. The value space of *NMTOKENS* is the set of finite, non-zero-length sequences of NMTOKENs. The lexical space of *NMTOKENS* is the set of space-separated lists of tokens, of which each token is in the lexical space of . The itemType of *NMTOKENS* is .

For compatibility (see § 1.4 – Terminology on page 2) *NMTOKENS* should be used only on attributes.

3.3.5.1. Constraining facets

3.3.6. Name

Name represents XML Names. The value space of *Name* is the set of all strings which match the Name production of [XML 1.0 (Second Edition)]. The lexical space of *Name* is the set of all strings which match the Name production of [XML 1.0 (Second Edition)]. The base type of *Name* is .

3.3.6.1. Constraining facets

3.3.6.2. Derived datatypes

3.3.7. NCName

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 .

3.3.7.1. Constraining facets

3.3.7.2. Derived datatypes

3.3.8. ID

ID represents the ID attribute type from [XML 1.0 (Second Edition)]. 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 .

For compatibility (see § 1.4 – Terminology on page 2) *ID* should be used only on attributes.

3.3.8.1. Constraining facets

3.3.9. IDREF

IDREF represents the IDREF attribute type from [XML 1.0 (Second Edition)]. 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 .

For compatibility (see § 1.4 – Terminology on page 2) this datatype should be used only on attributes.

3.3.9.1. Constraining facets

3.3.9.2. Derived datatypes

3.3.10. IDREFS

IDREFS represents the IDREFS attribute type from [XML 1.0 (Second Edition)]. The value space of *IDREFS* is the set of finite, non-zero-length sequences of s. The lexical space of *IDREFS* is the set of space-separated lists of tokens, of which each token is in the lexical space of . The itemType of *IDREFS* is .

For compatibility (see § 1.4 – Terminology on page 2) *IDREFS* should be used only on attributes.

3.3.10.1. Constraining facets

3.3.11. ENTITY

ENTITY represents the ENTITY attribute type from [XML 1.0 (Second Edition)]. 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 .

☞ The value space of *ENTITY* is scoped to a specific instance document.

For compatibility (see § 1.4 – Terminology on page 2) *ENTITY* should be used only on attributes.

3.3.11.1. Constraining facets

3.3.11.2. Derived datatypes

3.3.12. ENTITIES

ENTITIES represents the ENTITIES attribute type from [XML 1.0 (Second Edition)]. The value space of *ENTITIES* is the set of finite, non-zero-length sequences of ENTITYs 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 . The itemType of *ENTITIES* is .

☞ The value space of *ENTITIES* is scoped to a specific instance document.

For compatibility (see § 1.4 – Terminology on page 2) *ENTITIES* should be used only on attributes.

3.3.12.1. Constraining facets

3.3.13. integer

integer is derived from by fixing the value of fractionDigits to be 0and 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 .

3.3.13.1. Lexical representation

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

3.3.13.2. Canonical representation

The canonical representation for *integer* is defined by prohibiting certain options from the § 3.3.13.1 – Lexical representation on page 31. Specifically, the preceding optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.13.3. Constraining facets

3.3.13.4. Derived datatypes

3.3.14. nonPositiveInteger

nonPositiveInteger is derived from 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 .

3.3.14.1. Lexical representation

nonPositiveInteger has a lexical representation consisting of an optional preceding sign followed by a 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.

3.3.14.2. Canonical representation

The canonical representation for *nonPositiveInteger* is defined by prohibiting certain options from the § 3.3.14.1 – Lexical representation on page 31. In the canonical form for zero, the sign must be omitted. Leading zeroes are prohibited.

3.3.14.3. Constraining facets

3.3.14.4. Derived datatypes

3.3.15. negativeInteger

negativeInteger is derived from 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 .

3.3.15.1. Lexical representation

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

3.3.15.2. Canonical representation

The canonical representation for *negativeInteger* is defined by prohibiting certain options from the § 3.3.15.1 – Lexical representation on page 31. Specifically, leading zeroes are prohibited.

3.3.15.3. Constraining facets

3.3.16. long

long is derived from by setting the value of maxInclusive to be 9223372036854775807 and minInclusive to be -9223372036854775808. The base type of *long* is .

3.3.16.1. Lexical representation

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

3.3.16.2. Canonical representation

The canonical representation for *long* is defined by prohibiting certain options from the § 3.3.16.1 – Lexical representation on page 32. Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.16.3. Constraining facets

3.3.16.4. Derived datatypes

3.3.17. int

int is derived from by setting the value of maxInclusive to be 2147483647 and minInclusive to be -2147483648. The base type of *int* is .

3.3.17.1. Lexical representation

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

3.3.17.2. Canonical representation

The canonical representation for *int* is defined by prohibiting certain options from the § 3.3.17.1 – Lexical representation on page 32. Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.17.3. Constraining facets

3.3.17.4. Derived datatypes

3.3.18. short

short is derived from by setting the value of maxInclusive to be 32767 and minInclusive to be -32768. The base type of *short* is .

3.3.18.1. Lexical representation

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

3.3.18.2. Canonical representation

The canonical representation for *short* is defined by prohibiting certain options from the § 3.3.18.1 – Lexical representation on page 32. Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.18.3. Constraining facets

3.3.18.4. Derived datatypes

3.3.19. byte

byte is derived from by setting the value of maxInclusive to be 127 and minInclusive to be -128. The base type of *byte* is .

3.3.19.1. Lexical representation

byte has a lexical representation consisting of an optional sign followed by a finite-length sequence of decimal digits (#x30-#x39). If the sign is omitted, "+" is assumed. For example: -1, 0, 126, +100.

3.3.19.2. Canonical representation

The canonical representation for *byte* is defined by prohibiting certain options from the § 3.3.19.1 – Lexical representation on page 33. Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.19.3. Constraining facets

3.3.20. nonNegativeInteger

nonNegativeInteger is derived from 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 .

3.3.20.1. Lexical representation

nonNegativeInteger has a lexical representation consisting of an optional sign followed by a 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.

3.3.20.2. Canonical representation

The canonical representation for *nonNegativeInteger* is defined by prohibiting certain options from the § 3.3.20.1 – Lexical representation on page 33. Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.20.3. Constraining facets

3.3.20.4. Derived datatypes

3.3.21. unsignedLong

unsignedLong is derived from by setting the value of maxInclusive to be 18446744073709551615. The base type of *unsignedLong* is .

3.3.21.1. Lexical representation

unsignedLong has a lexical representation consisting of a finite-length sequence of decimal digits (#x30-#x39). For example: 0, 12678967543233, 100000.

3.3.21.2. Canonical representation

The canonical representation for *unsignedLong* is defined by prohibiting certain options from the § 3.3.21.1 – Lexical representation on page 34. Specifically, leading zeroes are prohibited.

3.3.21.3. Constraining facets

3.3.21.4. Derived datatypes

3.3.22. unsignedInt

unsignedInt is derived from by setting the value of maxInclusive to be 4294967295. The base type of *unsignedInt* is .

3.3.22.1. Lexical representation

unsignedInt has a lexical representation consisting of a finite-length sequence of decimal digits (#x30-#x39). For example: 0, 1267896754, 100000.

3.3.22.2. Canonical representation

The canonical representation for *unsignedInt* is defined by prohibiting certain options from the § 3.3.22.1 – Lexical representation on page 34. Specifically, leading zeroes are prohibited.

3.3.22.3. Constraining facets

3.3.22.4. Derived datatypes

3.3.23. unsignedShort

unsignedShort is derived from by setting the value of maxInclusive to be 65535. The base type of *unsignedShort* is .

3.3.23.1. Lexical representation

unsignedShort has a lexical representation consisting of a finite-length sequence of decimal digits (#x30-#x39). For example: 0, 12678, 10000.

3.3.23.2. Canonical representation

The canonical representation for *unsignedShort* is defined by prohibiting certain options from the § 3.3.23.1 – Lexical representation on page 34. Specifically, the leading zeroes are prohibited.

3.3.23.3. Constraining facets

3.3.23.4. Derived datatypes

3.3.24. unsignedByte

unsignedByte is derived from by setting the value of maxInclusive to be 255. The base type of *unsignedByte* is .

3.3.24.1. Lexical representation

unsignedByte has a lexical representation consisting of a finite-length sequence of decimal digits (#x30-#x39). For example: 0, 126, 100.

3.3.24.2. Canonical representation

The canonical representation for *unsignedByte* is defined by prohibiting certain options from the § 3.3.24.1 – Lexical representation on page 35. Specifically, leading zeroes are prohibited.

3.3.24.3. Constraining facets

3.3.25. positiveInteger

positiveInteger is derived from 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 .

3.3.25.1. Lexical representation

positiveInteger has a lexical representation consisting of an optional positive sign ("+") followed by a finite-length sequence of decimal digits (#x30-#x39). For example: 1, 12678967543233, +100000.

3.3.25.2. Canonical representation

The canonical representation for *positiveInteger* is defined by prohibiting certain options from the § 3.3.25.1 – Lexical representation on page 35. Specifically, the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.25.3. Constraining facets

4. Datatype components

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 datatype (schema) components, see Schema Component Details of [XML Schema Part 1: Structures].

4.1. Simple Type Definition

Simple Type definitions provide for:

- Establishing the value space and lexical space of a datatype, through the combined set of constraining facets specified in the definition;
- Attaching a unique name (actually a) to the value space and lexical space.

4.1.1. The Simple Type Definition Schema Component

The Simple Type Definition schema component has the following properties:

Optional. An NCName as defined by [Namespaces in XML]. Either absent or a namespace name, as defined in [Namespaces in XML]. One of {atomic, list, union}. Depending on the value of , further properties are defined as follows:

atomic

A built-in primitive datatype definition).

list

An atomic or union simple type definition.

union

A non-empty sequence of simple type definitions.

A possibly empty set of § 2.4 – Facets on page 4. A set of § 2.4.1 – Fundamental facets on page 5 If the datatype has been derived by restriction then the component from which it is derived, otherwise the § 4.1.6 – Simple Type Definition for anySimpleType on page 40. A subset of *{restriction, list, union}*. Optional. An annotation.
Datatypes are identified by their and . Except for anonymous datatypes (those with no), datatype definitions must be uniquely identified within a schema.

If is atomic then the value space of the datatype defined will be a subset of the value space of (which is a subset of the value space of). If is list then the value space of the datatype defined will be the set of finite-length sequence of values from the value space of . If is union then the value space of the datatype defined will be the union of the value spaces of each datatype in .

If is atomic then the of must be atomic. If is list then the of must be either atomic or union. If is union then must be a list of datatype definitions.

The value of consists of the set of facets specified directly in the datatype definition unioned with the possibly empty set of of .

The value of consists of the set of fundamental facets and their values.

If is the empty set then the type can be used in deriving other types; the explicit values *restriction*, *list* and *union* prevent further derivations by restriction, list and union respectively.

4.1.2. XML Representation of Simple Type Definition Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

The actual value of the `name` attribute, if present, otherwise null A set corresponding to the actual value of the `final` attribute, if present, otherwise the actual value of the `finalDefault` attribute of the ancestor schema element information item, if present, otherwise the empty string, as follows:

the empty string

the empty set;

`#all`

{restriction, list, union};

otherwise

a set with members drawn from the set above, each being present or absent depending on whether the string contains an equivalently named space-delimited substring.

☞ Although the `finalDefault` attribute of schema may include values other than restriction, list or union, those values are ignored in the determination of

The actual value of the `targetNamespace` attribute of the parent `schema` element information item.
The annotation corresponding to the element information item in the children, if present, otherwise null
A derived datatype can be derived from a primitive datatype or another derived datatype by one of three means: by *restriction*, by *list* or by *union*.

4.1.2.1. Derivation by restriction

The actual value of of The union of the set of § 2.4 – Facets on page 4 components resolved to by the facet children merged with from , subject to the Facet Restriction Valid constraints specified in § 2.4 – Facets on page 4. The component resolved to by the actual value of the `base` attribute or the children, whichever is present.

☞ An electronic commerce schema might define a datatype called *Sku* (the barcode number that appears on products) from the built-in datatype 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-derived datatype, is its base type and pattern is the facet.

4.1.2.2. Derivation by list

list The component resolved to by the actual value of the `itemType` attribute or the children, whichever is present.
A list datatype must be derived from an atomic or a union datatype, known as the itemType of the list datatype. This yields a datatype whose value space is composed of finite-length sequences of values from the value space of the itemType and whose lexical space is composed of space-separated lists of literals of the itemType.

☞ 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-derived datatype, is its itemType and list is the derivation method.

As mentioned in § 2.5.1.2 – List datatypes on page 5, when a datatype is derived from a list datatype, the following constraining facets can be used:

- length
- maxLength

- minLength
- enumeration
- pattern
- whiteSpace

regardless of the constraining facets that are applicable to the atomic datatype that serves as the itemType of the list.

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

4.1.2.3. Derivation by union

union The sequence of components resolved to by the items in the actual value of the `memberTypes` attribute, if any, in order, followed by the components resolved to by the children, if any, in order. If is *union* for any components resolved to above, then the is replaced by its .
A union datatype can be derived from one or more atomic, list or other union datatypes, known as the memberTypes of that union datatype.

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 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>
```

As mentioned in § 2.5.1.3 – Union datatypes on page 7, when a datatype is derived from a union datatype, the only following constraining facets can be used:

- pattern
- enumeration

regardless of the constraining facets that are applicable to the datatypes that participate in the union

4.1.3. Constraints on XML Representation of Simple Type Definition

src: Single Facet Value

Unless otherwise specifically allowed by this specification (Multiple patterns If multiple element information items appear as children of a , the values should be combined as if they appeared in a single as separate es. and Multiple enumerations If multiple element information items appear as children of a the of the component should be the set of all such values.) any given constraining facet can only be specifed once within a single derivation step.

src: itemType attribute or simpleType child

Either the `itemType` attribute or the child of the element must be present, but not both.

src: base attribute or simpleType child

Either the `base` attribute or the `simpleType` child of the element must be present, but not both.

src: memberTypes attribute or simpleType children

Either the `memberTypes` attribute of the element must be non-empty or there must be at least one `simpleType` child.

4.1.4. Simple Type Definition Validation Rules

cvc: Facet Valid

A value in a value space is facet-valid with respect to a constraining facet component if:

1. the value is facet-valid with respect to the particular constraining facet as specified below.

cvc: Datatype Valid

A string is datatype-valid with respect to a datatype definition if:

1. it matches a literal in the lexical space of the datatype, determined as follows:
   A. if pattern is a member of , then the string must be pattern valid A literal in a is facet-valid with respect to if: 1. ;
   B. if pattern is not a member of , then
      i. if is atomic then the string must match a literal in the lexical space of
      ii. if is list then the string must be a sequence of space-separated tokens, each of which matches a literal in the lexical space of
      iii. if is union then the string must match a literal in the lexical space of at least one member of

2. the value denoted by the literal matched in the previous step is a member of the value space of the datatype, as determined by it being Facet Valid A value in a is facet-valid with respect to a component if: 1. with respect to each member of (except for pattern).

4.1.5. Constraints on Simple Type Definition Schema Components

cos: applicable facets

The constraining facets which are allowed to be members of are dependent on as specified in the following table:

cos: list of atomic

If is list, then the of must be atomic or union.

cos: no circular unions

If is union, then it is an error if and match and of any member of .

4.1.6. Simple Type Definition for anySimpleType

There is a simple type definition nearly equivalent to the simple version of the ur-type definition present in every schema by definition. It has the following properties:

anySimpleType http://www.w3.org/2001/XMLSchema the ur-type definition the empty set absent

4.2. Fundamental Facets

4.2.1. equal

Every value space supports the notion of equality, with the following rules:

- for any *a* and *b* in the value space, either *a* is equal to *b*, denoted *a = b*, or *a* is not equal to *b*, denoted *a != b*
- there is no pair *a* and *b* from the value space such that both *a = b* and *a != b*
- for all *a* in the value space, *a = a*
- for any *a* and *b* in the value space, *a = b* if and only if *b = a*
- for any *a*, *b* and *c* in the value space, if *a = b* and *b = c*, then *a = c*
- for any *a* and *b* in the value space if *a = b*, then *a* and *b* cannot be distinguished (i.e., equality is identity)
- the value spaces of all primitive datatypes are disjoint (they do not share any values)

On every datatype, the operation Equal is defined in terms of the equality property of the value space: for any values *a*, *b* drawn from the value space, *Equal(a,b)* is true if *a = b*, and false otherwise.

Note that in consequence of the above:

- given value space *A* and value space *B* where *A* and *B* are disjoint, every pair of values *a* from *A* and *b* from *B*, *a != b*

- two values which are members of the value space of the same primitive datatype may always be compared with each other
- if a datatype T is derived by union from memberTypes A, B, ... then the value space of T is the union of value spaces of its memberTypes A, B, ... Some values in the value space of T are also values in the value space of A. Other values in the value space of T will be values in the value space of B and so on. Values in the value space of T which are also in the value space of A can be compared with other values in the value space of A according to the above rules. Similarly for values of type T and B and all the other memberTypes.
- if a datatype T' is derived by restriction from an atomic datatype T then the value space of T' is a subset of the value space of T. Values in the value spaces of T and T' can be compared according to the above rules
- if datatypes T' and T'' are derived by restriction from a common atomic ancestor T then the value spaces of T' and T'' may overlap. Values in the value spaces of T' and T'' can be compared according to the above rules

There is no schema component corresponding to the *equal* fundamental facet.

4.2.2. ordered

An *order relation* on a value space is a mathematical relation that imposes a total order or a partial order on the members of the value space.

A value space, and hence a datatype, is said to be *ordered* if there exists an order-relation defined for that value space.

A *partial order* is an order-relation that is *irreflexive*, *asymmetric* and *transitive*.

A partial order has the following properties:

- for no a in the value space, $a < a$ (irreflexivity)
- for all a and b in the value space, $a < b$ implies not($b < a$) (asymmetry)
- for all a, b and c in the value space, $a < b$ and $b < c$ implies $a < c$ (transitivity)

The notation $a <> b$ is used to indicate the case when $a \mathrel{!=} b$ and neither $a < b$ nor $b < a$. For any values a and b from different primitive value spaces, $a <> b$.

When $a <> b$, a and b are *incomparable*,otherwise they are *comparable*.

A *total order* is an partial order such that for no a and b is it the case that $a <> b$.

A total order has all of the properties specified above for partial order, plus the following property:

- for all a and b in the value space, either $a < b$ or $b < a$ or $a = b$

The fact that this specification does not define an order-relation for some datatype does not mean that some other application cannot treat that datatype as being ordered by imposing its own order relation.

ordered provides for:

- indicating whether an order-relation is defined on a value space, and if so, whether that order-relation is a partial order or a total order

4.2.2.1. The ordered Schema Component

One of *{false, partial, total}*.
depends on , and in the component in which a ordered component appears as a member of .

When is atomic, is inherited from of . For all primitive types is as specified in the table in Appendix C.1 – Fundamental Facets on page 59.

When is list, is *false*.

When is union, is *partial* unless one of the following:

- If every member of is derived from a common ancestor other than the simple ur-type, then is the same as that ancestor's *ordered* facet
- If every member of has a of *false* for the *ordered* facet, then is *false*

4.2.3. bounded

A value *u* in an ordered value space *U* is said to be an *inclusive upper bound* of a value space *V* (where *V* is a subset of *U*) if for all *v* in *V*, *u* >= *v*.

A value *u* in an ordered value space *U* is said to be an *exclusive upper bound* of a value space *V* (where *V* is a subset of *U*) if for all *v* in *V*, *u* > *v*.

A value *l* in an ordered value space *L* is said to be an *inclusive lower bound* of a value space *V* (where *V* is a subset of *L*) if for all *v* in *V*, *l* <= *v*.

A value *l* in an ordered value space *L* is said to be an *exclusive lower bound* of a value space *V* (where *V* is a subset of *L*) if for all *v* in *V*, *l* < *v*.

A datatype is *bounded* if its value space has either an inclusive upper bound or an exclusive upper bound and either an inclusive lower bound or an exclusive lower bound.

bounded provides for:

- indicating whether a value space is bounded

4.2.3.1. The bounded Schema Component

A .
depends on , and in the component in which a bounded component appears as a member of .

When is atomic, if one of minInclusive or minExclusive and one of maxInclusive or maxExclusive are among , then is *true*; else is *false*.

When is list, if length or both of minLength and maxLength are among , then is *true*; else is *false*.

When is union, if is *true* for every member of and all members of share a common ancestor, then is *true*; else is *false*.

4.2.4. cardinality

Every value space has associated with it the concept of *cardinality*. Some value spaces are finite, some are countably infinite while still others could conceivably be uncountably infinite (although no value space defined by this specification is uncountable infinite). A datatype is said to have the cardinality of its value space.

It is sometimes useful to categorize value spaces (and hence, datatypes) as to their cardinality. There are two significant cases:

- value spaces that are finite
- value spaces that are countably infinite

cardinality provides for:

- indicating whether the cardinality of a value space is *finite* or *countably infinite*

4.2.4.1. The cardinality Schema Component

One of *{finite, countably infinite}*.
depends on , and in the component in which a cardinality component appears as a member of .

When is atomic and of is *finite*, then is *finite*.

When is atomic and of is *countably infinite* and either of the following conditions are true, then is *finite*; else is *countably infinite*:

1. one of length, maxLength, totalDigits is among ,
2. all of the following are true:
   A. one of minInclusive or minExclusive is among
   B. one of maxInclusive or maxExclusive is among
   C. either of the following are true:
      i. fractionDigits is among
      ii. is one of , , , , or or any type derived from them

When is list, if length or both of minLength and maxLength are among , then is *finite*; else is *countably infinite*.

When is union, if is *finite* for every member of , then is *finite*; else is *countably infinite*.

4.2.5. numeric

A datatype is said to be *numeric* if its values are conceptually quantities (in some mathematical number system).

A datatype whose values are not numeric is said to be *non-numeric*.

numeric provides for:

- indicating whether a value space is numeric

4.2.5.1. The numeric Schema Component

A
depends on , , and in the component in which a cardinality component appears as a member of .

When is atomic, is inherited from of . For all primitive types is as specified in the table in Appendix C.1 – Fundamental Facets on page 59.

When is list, is *false*.

When is union, if is *true* for every member of , then is *true*; else is *false*.

4.3. Constraining Facets

4.3.1. length

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 .

For and datatypes derived from , *length* is measured in units of characters as defined in [XML 1.0 (Second Edition)]. For , *length* is measured in units of characters (as for). For and and datatypes derived from them, *length* is measured in octets (8 bits) of binary data. For datatypes derived by list, *length* is measured in number of list items.

☞ For and datatypes derived from , *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 .

☞ The following is the definition of a user-derived 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

A . A . Optional. An annotation.
If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.1.2. XML Representation of length Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.1.3. length Validation Rules

cvc: Length Valid
A value in a value space is facet-valid with respect to length, determined as follows:

1. if the is atomic then

   A. if is or , then the length of the value, as measured in characters must be equal to ;

   B. if is or , then the length of the value, as measured in octets of the binary data, must be equal to ;

C. if is or , then any is facet-valid.

2. if the is list, then the length of the value, as measured in list items, must be equal to

The use of length on datatypes derived from and is deprecated. Future versions of this specification may remove this facet for these datatypes.

4.3.1.4. Constraints on length Schema Components

cos: length and minLength or maxLength

If is a member of then

1. It is an error for to be a member of unless
   A. the of <= the of and
   B. there is type definition from which this one is derived by one or more restriction steps in which has the same and is not specified.
2. It is an error for to be a member of unless
   A. the of <= the of and
   B. there is type definition from which this one is derived by one or more restriction steps in which has the same and is not specified.

cos: length valid restriction

It is an error if is among the members of of and is not equal to the of the parent .

4.3.2. minLength

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 .

For and datatypes derived from , *minLength* is measured in units of characters as defined in [XML 1.0 (Second Edition)]. For and and datatypes derived from them, *minLength* is measured in octets (8 bits) of binary data. For datatypes derived by list, *minLength* is measured in number of list items.

☞ For and datatypes derived from , *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 .

☞ The following is the definition of a user-derived 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

A . A . Optional. An annotation.
If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.2.2. XML Representation of minLength Schema Component

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.2.3. minLength Validation Rules

cvc: minLength Valid

A value in a value space is facet-valid with respect to minLength, determined as follows:

1. if the is atomic then
   A. if is or , then the length of the value, as measured in characters must be greater than or equal to ;
   B. if is or , then the length of the value, as measured in octets of the binary data, must be greater than or equal to ;
   C. if is or , then any is facet-valid.
2. if the is list, then the length of the value, as measured in list items, must be greater than or equal to

The use of minLength on datatypes derived from and is deprecated. Future versions of this specification may remove this facet for these datatypes.

4.3.2.4. Constraints on minLength Schema Components

cos: minLength <= maxLength

If both and are members of , then the of must be less than or equal to the of .

cos: minLength valid restriction

It is an error if is among the members of of and is less than the of the parent .

4.3.3. maxLength

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 .

For and datatypes derived from , *maxLength* is measured in units of characters as defined in [XML 1.0 (Second Edition)]. For and and datatypes derived from them, *maxLength* is measured in octets (8 bits) of binary data. For datatypes derived by list, *maxLength* is measured in number of list items.

☞ For and datatypes derived from , *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 .

☞ The following is the definition of a user-derived 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

A . A . Optional. An annotation.
If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.3.2. XML Representation of maxLength Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.3.3. maxLength Validation Rules

cvc: maxLength Valid

A value in a value space is facet-valid with respect to maxLength, determined as follows:

1. if the is atomic then
   A. if is or , then the length of the value, as measured in characters must be less than or equal to ;
   B. if is or , then the length of the value, as measured in octets of the binary data, must be less than or equal to ;
   C. if is or , then any is facet-valid.
2. if the is list, then the length of the value, as measured in list items, must be less than or equal to

The use of maxLength on datatypes derived from and is deprecated. Future versions of this specification may remove this facet for these datatypes.

4.3.3.4. Constraints on maxLength Schema Components

cos: maxLength valid restriction

It is an error if is among the members of of and is greater than the of the parent .

4.3.4. pattern

pattern is a constraint on the value space of a datatype which is achieved by constraining the lexical space to literals which match a specific pattern. The value of *pattern* must be a regular expression.

pattern provides for:

- Constraining a value space to values that are denoted by literals which match a specific regular expression.

☞ The following is the definition of a user-derived 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

A regular expression. Optional. An annotation.

4.3.4.2. XML Representation of pattern Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

must be a valid regular expression. The actual value of the `value` attribute The annotations corresponding to all the element information items in the children, if any.

4.3.4.3. Constraints on XML Representation of pattern

src: Multiple patterns

If multiple element information items appear as children of a , the values should be combined as if they appeared in a single regular expression as separate branches.

☞ It is a consequence of the schema representation constraint Multiple patterns If multiple element information items appear as children of a , the values should be combined as if they appeared in a single as separate es. and of the rules for restriction that pattern facets specified on the *same* step in a type derivation are ORed together, while pattern facets specified on *different* steps of a type derivation are ANDed together.

Thus, to impose two pattern constraints simultaneously, schema authors may either write a single pattern which expresses the intersection of the two patterns they wish to impose, or define each pattern on a separate type derivation step.

4.3.4.4. pattern Validation Rules

cvc: pattern valid

A literal in a lexical space is facet-valid with respect to pattern if:

1. the literal is among the set of character sequences denoted by the regular expression specified in .

4.3.5. enumeration

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.

The following example is a datatype definition for a user-derived datatype which limits the values of dates to the three US holidays enumerated. This datatype 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

A set of values from the value space of the . Optional. An annotation.

4.3.5.2. XML Representation of enumeration Schema Components

The XML representation for an schema component is an element information item. The correspondences between the properties of the information item and properties of the component are as follows:

must be in the value space of . The actual value of the `value` attribute The annotations corresponding to all the element information items in the children, if any.

4.3.5.3. Constraints on XML Representation of enumeration

src: Multiple enumerations

If multiple element information items appear as children of a the of the component should be the set of all such values.

4.3.5.4. enumeration Validation Rules

cvc: enumeration valid

A value in a value space is facet-valid with respect to enumeration if the value is one of the values specified in

4.3.5.5. Constraints on enumeration Schema Components

cos: enumeration valid restriction

It is an error if any member of is not in the value space of .

4.3.6. whiteSpace

whiteSpace constrains the value space of types derived from such that the various behaviors specified in Attribute Value Normalization in [XML 1.0 (Second Edition)] 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 1.0 (Second Edition)] 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 leading and trailing #x20's are removed.

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 (and types derived by restriction from it) the value of *whiteSpace* is `collapse` and cannot be changed by a schema author; for the value of *whiteSpace* is `preserve`; for any type derived by restriction from the value of *whiteSpace* can be any of the three legal values. For all datatypes derived by list the value of *whiteSpace* is `collapse` and cannot be changed by a schema author. For all datatypes derived 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 memberTypes against which the union is successfully validated.

☞ For more information on *whiteSpace*, see the discussion on white space normalization in Schema Component Details in [XML Schema Part 1: Structures].

whiteSpace provides for:

- Constraining a value space according to the white space normalization rules.

☞ The following example is the datatype definition for the built-in derived datatype.

```
<simpleType name='token'>
    <restriction base='normalizedString'>
      <whiteSpace value='collapse'/>
    </restriction>
</simpleType>
```

4.3.6.1. The whiteSpace Schema Component

One of `{preserve, replace, collapse}`. A . Optional. An annotation.
If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.6.2. XML Representation of whiteSpace Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.6.3. whiteSpace Validation Rules

☞ There are no Validation Rules associated whiteSpace. For more information, see the discussion on white space normalization in Schema Component Details in [XML Schema Part 1: Structures].

4.3.6.4. Constraints on whiteSpace Schema Components

cos: whiteSpace valid restriction

It is an error if is among the members of of and any of the following conditions is true:

1. is *replace* or *preserve* and the of the parent is *collapse*
2. is *preserve* and the of the parent is *replace*

4.3.7. maxInclusive

maxInclusive is the inclusive upper bound of the value space for a datatype with the ordered property. The value of *maxInclusive* must be in the value space of the base type.

maxInclusive provides for:

- Constraining a value space to values with a specific inclusive upper bound.

The following is the definition of a user-derived 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

A value from the value space of the . A . Optional. An annotation.
If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.7.2. XML Representation of maxInclusive Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

must be in the value space of . The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.7.3. maxInclusive Validation Rules

cvc: maxInclusive Valid

A value in an ordered value space is facet-valid with respect to maxInclusive, determined as follows:

1. if the numeric property in is *true*, then the value must be numerically less than or equal to ;
2. if the numeric property in is *false* (i.e., is one of the date and time related datatypes), then the value must be chronologically less than or equal to ;

4.3.7.4. Constraints on maxInclusive Schema Components

cos: 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.

cos: maxInclusive valid restriction

It is an error if any of the following conditions is true:

1. is among the members of of and is greater than the of the parent
2. is among the members of of and is greater than or equal to the of the parent
3. is among the members of of and is less than the of the parent
4. is among the members of of and is less than or equal to the of the parent

4.3.8. maxExclusive

maxExclusive is the exclusive upper bound of the value space for a datatype with the ordered property. The value of *maxExclusive* must be in the value space of the base type or be equal to in .

maxExclusive provides for:

- Constraining a value space to values with a specific exclusive upper bound.

The following is the definition of a user-derived 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

A value from the value space of the . A . Optional. An annotation.
If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.8.2. XML Representation of maxExclusive Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

must be in the value space of . The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.8.3. maxExclusive Validation Rules

cvc: maxExclusive Valid

A value in an ordered value space is facet-valid with respect to maxExclusive, determined as follows:

1. if the numeric property in is *true*, then the value must be numerically less than ;
2. if the numeric property in is *false* (i.e., is one of the date and time related datatypes), then the value must be chronologically less than ;

4.3.8.4. Constraints on maxExclusive Schema Components

cos: maxInclusive and maxExclusive

It is an error for both maxInclusive and maxExclusive to be specified in the same derivation step of a datatype definition.

cos: 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.

cos: maxExclusive valid restriction

It is an error if any of the following conditions is true:

1. is among the members of of and is greater than the of the parent
2. is among the members of of and is greater than the of the parent
3. is among the members of of and is less than or equal to the of the parent
4. is among the members of of and is less than or equal to the of the parent

4.3.9. minExclusive

minExclusive is the exclusive lower bound of the value space for a datatype with the ordered property. The value of *minExclusive* must be in the value space of the base type or be equal to in .

minExclusive provides for:

- Constraining a value space to values with a specific exclusive lower bound.

☞ The following is the definition of a user-derived 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 previous one (named 'one-hundred-or-more').

4.3.9.1. The minExclusive Schema Component

A value from the value space of the . A . Optional. An annotation.
If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.9.2. XML Representation of minExclusive Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

must be in the value space of . The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.9.3. minExclusive Validation Rules

cvc: minExclusive Valid

A value in an ordered value space is facet-valid with respect to minExclusive if:

1. if the numeric property in is *true*, then the value must be numerically greater than ;
2. if the numeric property in is *false* (i.e., is one of the date and time related datatypes), then the value must be chronologically greater than ;

4.3.9.4. Constraints on minExclusive Schema Components

cos: minInclusive and minExclusive

It is an error for both minInclusive and minExclusive to be specified for the same datatype.

cos: 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.

cos: minExclusive valid restriction

It is an error if any of the following conditions is true:

1. is among the members of of and is less than the of the parent
2. is among the members of of and is greater the of the parent
3. is among the members of of and is less than the of the parent
4. is among the members of of and is greater than or equal to the of the parent

4.3.10. minInclusive

minInclusive is the inclusive lower bound of the value space for a datatype with the ordered property. The value of *minInclusive* must be in the value space of the base type.

minInclusive provides for:

- Constraining a value space to values with a specific inclusive lower bound.

The following is the definition of a user-derived 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

A value from the value space of the . A . Optional. An annotation.

If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.10.2. XML Representation of minInclusive Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

must be in the value space of . The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.10.3. minInclusive Validation Rules

cvc: minInclusive Valid

A value in an ordered value space is facet-valid with respect to minInclusive if:

1. if the numeric property in is *true*, then the value must be numerically greater than or equal to ;
2. if the numeric property in is *false* (i.e., is one of the date and time related datatypes), then the value must be chronologically greater than or equal to ;

4.3.10.4. Constraints on minInclusive Schema Components

cos: 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.

cos: minInclusive valid restriction

It is an error if any of the following conditions is true:

1. is among the members of of and is less than the of the parent
2. is among the members of of and is greater the of the parent
3. is among the members of of and is less than or equal to the of the parent
4. is among the members of of and is greater than or equal to the of the parent

4.3.11. totalDigits

totalDigits controls the maximum number of values in the value space of datatypes derived from , by restricting it to numbers that are expressible as $i \times 10^{-n}$ where i and n are integers such that $|i| < 10^{totalDigits}$ and $0 <= n <= totalDigits$. The value of *totalDigits* must be a .

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. Note that it does not restrict the lexical space directly; a lexical representation that adds additional leading zero digits or trailing fractional zero digits is still permitted.

4.3.11.1. The totalDigits Schema Component

A . A . Optional. An annotation.
If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.11.2. XML Representation of totalDigits Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.11.3. totalDigits Validation Rules

cvc: totalDigits Valid

A value in a value space is facet-valid with respect to totalDigits if:

1. that value is expressible as $i \times 10$^-n where i and n are integers such that $|i| < 10$^ and $0 <= n <=$.

4.3.11.4. Constraints on totalDigits Schema Components

cos: totalDigits valid restriction

It is an error if is among the members of of and is greater than the of the parent

4.3.12. fractionDigits

fractionDigits controls the size of the minimum difference between values in the value space of datatypes derived from *decimal*, by restricting the value space to numbers that are expressible as $i \times 10$^-n where i and n are integers and $0 <= n <=$ *fractionDigits*. The value of *fractionDigits* must be a .

The term *fractionDigits* is chosen to reflect the fact that it restricts the value space to those values that can be represented lexically using at most *fractionDigits* to the right of the decimal point. Note that it does not restrict the lexical space directly; a non-canonical lexical representation that adds additional leading zero digits or trailing fractional zero digits is still permitted.

The following is the definition of a user-derived 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'>
    <totalDigits value='4'/>
    <fractionDigits value='1'/>
    <minInclusive value='36.4'/>
    <maxInclusive value='40.5'/>
  </restriction>
</simpleType>
```

4.3.12.1. The fractionDigits Schema Component

A . A . Optional. An annotation.
If is *true*, then types for which the current type is the cannot specify a value for other than .

4.3.12.2. XML Representation of fractionDigits Schema Components

The XML representation for a schema component is a element information item. The correspondences between the properties of the information item and properties of the component are as follows:

The actual value of the `value` attribute The actual value of the `fixed` attribute, if present, otherwise false The annotations corresponding to all the element information items in the children, if any.

4.3.12.3. fractionDigits Validation Rules

cvc: fractionDigits Valid

A value in a value space is facet-valid with respect to fractionDigits if:

1. that value is expressible as $i \times 10$^-n where i and n are integers and 0 <= n <= .

4.3.12.4. Constraints on fractionDigits Schema Components

cos: fractionDigits less than or equal to totalDigits

It is an error for fractionDigits to be greater than totalDigits.

cos: fractionDigits valid restriction

It is an error if fractionDigits is among the members of of and is greater than the of the parent fractionDigits.

5. Conformance

This specification describes two levels of conformance for datatype processors. The first is required of all processors. Support for the other will depend on the application environments for which the processor is intended.

Minimally conforming processors must completely and correctly implement the Constraint on Schemas and Validation Rule .

Processors which accept schemas in the form of XML documents as described in § 4.1.2 – XML Representation of Simple Type Definition Schema Components on page 36 (and other relevant portions of § 4 – Datatype components on page 35) are additionally said to provide *conformance to the XML Representation of Schemas*, and must, when processing schema documents, completely and correctly implement all Schema Representation Constraints in this specification, and must adhere exactly to the specifications in § 4.1.2 – XML Representation of Simple Type Definition Schema Components on page 36 (and other relevant portions of § 4 – Datatype components on page 35) for mapping the contents of such documents to schema components for use in validation.

☞ By separating the conformance requirements relating to the concrete syntax of XML schema documents, this specification admits processors which validate using schemas stored in optimized binary representations, dynamically created schemas represented as programming language data structures, or implementations in which particular schemas are compiled into executable code such as C or Java. Such processors can be said to be minimally conforming but not necessarily in conformance to the XML Representation of Schemas.

Appendix A. Schema for Datatype Definitions (normative)

Appendix B. DTD for Datatype Definitions (non-normative)

Appendix C. Datatypes and Facets

C.1. Fundamental Facets

The following table shows the values of the fundamental facets for each built-in datatype.

Appendix D. ISO 8601 Date and Time Formats

D.1. ISO 8601 Conventions

The primitive datatypes , , , , , , , and use lexical formats inspired by [ISO 8601]. Following [ISO 8601], the lexical forms of these datatypes can include only the characters #20 through #7F. This appendix provides more detail on the ISO formats and discusses some deviations from them for the datatypes defined in this specification.

[ISO 8601] "specifies the representation of dates in the proleptic Gregorian calendar and times and representations of periods of time". The proleptic Gregorian calendar includes dates prior to 1582 (the year it came into use as an ecclesiastical calendar). It should be pointed out that the datatypes described in this specification do not cover all the types of data covered by [ISO 8601], nor do they support all the lexical representations for those types of data.

[ISO 8601] lexical formats are described using "pictures" in which characters are used in place of decimal digits. The allowed decimal digits are (#x30-#x39). For the primitive datatypes , , , , , , and . these characters have the following meanings:

- C -- represents a digit used in the thousands and hundreds components, the "century" component, of the time element "year". Legal values are from 0 to 9.
- Y -- represents a digit used in the tens and units components of the time element "year". Legal values are from 0 to 9.
- M -- represents a digit used in the time element "month". The two digits in a MM format can have values from 1 to 12.
- D -- represents a digit used in the time element "day". The two digits in a DD format can have values from 1 to 28 if the month value equals 2, 1 to 29 if the month value equals 2 and the year is a leap year, 1 to 30 if the month value equals 4, 6, 9 or 11, and 1 to 31 if the month value equals 1, 3, 5, 7, 8, 10 or 12.
- h -- represents a digit used in the time element "hour". The two digits in a hh format can have values from 0 to 24. If the value of the hour element is 24 then the values of the minutes element and the seconds element must be 00 and 00.
- m -- represents a digit used in the time element "minute". The two digits in a mm format can have values from 0 to 59.

- s -- represents a digit used in the time element "second". The two digits in a ss format can have values from 0 to 60. In the formats described in this specification the whole number of seconds may be followed by decimal seconds to an arbitrary level of precision. This is represented in the picture by "ss.sss". A value of 60 or more is allowed only in the case of leap seconds.

  Strictly speaking, a value of 60 or more is not sensible unless the month and day could represent March 31, June 30, September 30, or December 31 *in UTC*. Because the leap second is added or subtracted as the last second of the day in UTC time, the long (or short) minute could occur at other times in local time. In cases where the leap second is used with an inappropriate month and day it, and any fractional seconds, should considered as added or subtracted from the following minute.

For all the information items indicated by the above characters, leading zeros are required where indicated.

In addition to the above, certain characters are used as designators and appear as themselves in lexical formats.

- T -- is used as time designator to indicate the start of the representation of the time of day in .
- Z -- is used as time-zone designator, immediately (without a space) following a data element expressing the time of day in Coordinated Universal Time (UTC) in , , , , , , , and .

In the lexical format for the following characters are also used as designators and appear as themselves in lexical formats:

- P -- is used as the time duration designator, preceding a data element representing a given duration of time.
- Y -- follows the number of years in a time duration.
- M -- follows the number of months or minutes in a time duration.
- D -- follows the number of days in a time duration.
- H -- follows the number of hours in a time duration.
- S -- follows the number of seconds in a time duration.

The values of the Year, Month, Day, Hour and Minutes components are not restricted but allow an arbitrary integer. Similarly, the value of the Seconds component allows an arbitrary decimal. Thus, the lexical format for and datatypes derived from it does not follow the alternative format of § 5.5.3.2.1 of [ISO 8601].

D.2. Truncated and Reduced Formats

[ISO 8601] supports a variety of "truncated" formats in which some of the characters on the left of specific formats, for example, the century, can be omitted. Truncated formats are, in general, not permitted for the datatypes defined in this specification with three exceptions. The datatype uses a truncated format for which represents an instant of time that recurs every day. Similarly, the and datatypes use left-truncated formats for . The datatype uses a right and left truncated format for .

[ISO 8601] also supports a variety of "reduced" or right-truncated formats in which some of the characters to the right of specific formats, such as the time specification, can be omitted. Right truncated formats are also, in general, not permitted for the datatypes defined in this specification with the following exceptions: right-truncated representations of are used as lexical representations for , , .

D.3. Deviations from ISO 8601 Formats

D.3.1. Sign Allowed

An optional minus sign is allowed immediately preceding, without a space, the lexical representations for , , , , .

D.3.2. No Year Zero

The year "0000" is an illegal year value.

D.3.3. More Than 9999 Years

To accommodate year values greater than 9999, more than four digits are allowed in the year representations of , , , and . This follows [ISO 8601:2000 Second Edition].

D.3.4. Time zone permitted

The lexical representations for the datatypes , , , , and permit an optional trailing time zone specificiation.

Appendix E. Adding durations to dateTimes

Given a S and a D, this appendix specifies how to compute a 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 is within a specific time period. This appendix also addresses the addition of s to the datatypes , , , and , which can be viewed as a set of s. In such cases, the addition is made to the first or starting in the set.

This is a logical explanation of the process. Actual implementations are free to optimize as long as they produce the same results. The calculation uses the notation S[year] to represent the year field of S, S[month] to represent the month field, and so on. It also depends on the following functions:

- fQuotient(a, b) = the greatest integer less than or equal to a/b
  - fQuotient(-1,3) = -1
  - fQuotient(0,3)...fQuotient(2,3) = 0
  - fQuotient(3,3) = 1
  - fQuotient(3.123,3) = 1
- modulo(a, b) = a - fQuotient(a,b)*b
  - modulo(-1,3) = 2
  - modulo(0,3)...modulo(2,3) = 0...2
  - modulo(3,3) = 0
  - modulo(3.123,3) = 0.123
- fQuotient(a, low, high) = fQuotient(a - low, high - low)
  - fQuotient(0, 1, 13) = -1
  - fQuotient(1, 1, 13) ... fQuotient(12, 1, 13) = 0

   - fQuotient(13, 1, 13) = 1
   - fQuotient(13.123, 1, 13) = 1

- modulo(a, low, high) = modulo(a - low, high - low) + low
   - modulo(0, 1, 13) = 12
   - modulo(1, 1, 13) ... modulo(12, 1, 13) = 1...12
   - modulo(13, 1, 13) = 1
   - modulo(13.123, 1, 13) = 1.123

- maximumDayInMonthFor(yearValue, monthValue) =
   - M := modulo(monthValue, 1, 13)
   - Y := yearValue + fQuotient(monthValue, 1, 13)
   - Return a value based on M and Y:

31	M = January, March, May, July, August, October, or December	
30	M = April, June, September, or November	
29	M = February AND (modulo(Y, 400) = 0 OR (modulo(Y, 100) != 0) AND modulo(Y, 4) = 0)	
28	Otherwise	

E.1. Algorithm

Essentially, this calculation is equivalent to separating D into <year,month> and <day,hour,minute,second> fields. The <year,month> is added to S. If the day is out of range, it is *pinned* to be within range. Thus April 31 turns into April 30. Then the <day,hour,minute,second> is added. This latter addition can cause the year and month to change.

Leap seconds are handled by the computation by treating them as overflows. Essentially, a value of 60 seconds in S is treated as if it were a duration of 60 seconds added to S (with a zero seconds field). All calculations thereafter 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].

The following is the precise specification. These steps must be followed in the same order. If a field in D is not specified, it is treated as if it were zero. If a field in S is not specified, it is treated in the calculation as if it were the minimum allowed value in that field, however, after the calculation is concluded, the corresponding field in E is removed (set to unspecified).

- *Months (may be modified additionally below)*
  - temp := S[month] + D[month]
  - E[month] := modulo(temp, 1, 13)
  - carry := fQuotient(temp, 1, 13)
- *Years (may be modified additionally below)*
  - E[year] := S[year] + D[year] + carry
- *Zone*
  - E[zone] := S[zone]
- *Seconds*
  - temp := S[second] + D[second]
  - E[second] := modulo(temp, 60)
  - carry := fQuotient(temp, 60)
- *Minutes*
  - temp := S[minute] + D[minute] + carry
  - E[minute] := modulo(temp, 60)
  - carry := fQuotient(temp, 60)
- *Hours*
  - temp := S[hour] + D[hour] + carry
  - E[hour] := modulo(temp, 24)
  - carry := fQuotient(temp, 24)
- *Days*
  - if S[day] > maximumDayInMonthFor(E[year], E[month])
    - tempDays := maximumDayInMonthFor(E[year], E[month])
  - else if S[day] < 1
    - tempDays := 1
  - else
    - tempDays := S[day]
  - E[day] := tempDays + D[day] + carry
  - *START LOOP*
    - *IF* E[day] < 1
      - E[day] := E[day] + maximumDayInMonthFor(E[year], E[month] - 1)

  - carry := -1

- *ELSE IF* E[day] > maximumDayInMonthFor(E[year], E[month])
  - E[day] := E[day] - maximumDayInMonthFor(E[year], E[month])
  - carry := 1
- *ELSE EXIT LOOP*
- temp := E[month] + carry
- E[month] := modulo(temp, 1, 13)
- E[year] := E[year] + fQuotient(temp, 1, 13)
- *GOTO START LOOP*

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

E.2. Commutativity and Associativity

Time durations are added by simply adding each of their fields, respectively, without overflow.

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

Appendix F. 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.

☞ 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>
```

A *regular expression* is composed from zero or more branches, separated by `|` characters.

Regular Expression

[1] regExp ::= ('|')*

For all branches *S*, and for all regular expressions *T*, valid regular expressions *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)*

A *branch* consists of zero or more pieces, concatenated together.

Branch

[2] branch ::= *

For all pieces *S*, and for all branches *T*, valid branches *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)*

A *piece* is an atom, possibly followed by a quantifier.

Piece

[3] piece ::= ?

For all atoms *S* and non-negative integers *n*, *m* such that *n* <= *m*, valid pieces *R* are:	Denoting the set of strings *L(R)* containing:
S	all strings in *L(S)*

For all atoms *S* and non-negative integers *n*, *m* such that *n* <= *m*, valid pieces *R* are:	Denoting the set of strings *L(R)* containing:
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

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.

A *quantifier* is one of `?`, `*`, `+`, `{n,m}` or `{n,}`, which have the meanings defined in the table above.

Quanitifer

[4]	quantifier	::=	[?*+] | ('{' '}')
[5]	quantity	::=	| |
[6]	quantRange	::=	','
[7]	quantMin	::=	','
[8]	QuantExact	::=	[0-9]+

An *atom* is either a normal character, a character class, or a parenthesized regular expression.

Atom

[9]	atom	::=	| | ('(' ')')

For all normal characters *c*, character classes *C*, and regular expressions *S*, valid atoms *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)*

A *metacharacter* is either `.`, `\`, `?`, `*`, `+`, `{`, `}` `(`, `)`, `[` or `]`. These characters have special meanings in regular expressions, but can be escaped to form atoms that denote the sets of strings containing only themselves, i.e., an escaped metacharacter behaves like a normal character.

A *normal character* is any XML character that is not a metacharacter. In regular expressions, a normal character is an atom that denotes the singleton set of strings containing only itself.

Normal Character

[10] Char ::= [^.\?*+()|#x5B#x5D]

Note that a normal character can be represented either as itself, or with a character reference.

F.1. Character Classes

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

[11] charClass ::= | |

A character class is either a character class escape or a character class expression.

A *character class expression* 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

[12] charClassExpr ::= '[' ']'

A *character group* is either a positive character group, a negative character group, or a character class subtraction.

Character Group

[13] charGroup ::= | |

A *positive character group* consists of one or more character ranges or character class escapes, concatenated together. A *positive character group* identifies the set of characters containing all of the characters in all of the sets identified by its constituent ranges or escapes.

Positive Character Group

[14] posCharGroup ::= (|)+

For all character ranges *R*, all character class escapes *E*, and all positive character groups *P*, valid positive character groups *G* are:	Identifying the set of characters *C(G)* containing:
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)*.

A *negative character group* is a positive character group preceded by the ^ character. For all positive character groups *P*, ^*P* is a valid *negative character group*, and *C(^P)* contains all XML characters that are *not* in *C(P)*.

Negative Character Group

[15]	negCharGroup	::=	'^'

A *character class subtraction* is a character class expression subtracted from a positive character group or negative character group, using the `-` character.

Character Class Subtraction

[16]	charClassSub	::=	(|) '-'

For any positive character group or negative character group *G*, and any character class expression *C*, *G-C* is a valid character class subtraction, identifying the set of all characters in *C(G)* that are not also in *C(C)*.

A *character range R* identifies a set of characters *C(R)* containing all XML characters with UCS code points in a specified range.

Character Range

[17]	charRange	::=	|
[18]	seRange	::=	'-'
[20]	charOrEsc	::=	|
[21]	XmlChar	::=	[^\#x2D#x5B#x5D]
[22]	XmlCharIncDash	::=	[^\#x5B#x5D]

A single XML character is a character range that identifies the set of characters containing only itself. All XML characters are valid character ranges, except as follows:

- The `[`, `]`, `-` and \ characters are not valid character ranges;
- The `^` character is only valid at the beginning of a positive character group if it is part of a negative character group
- The `-` character is a valid character range only at the beginning or end of a positive character group.

☞ The grammar for character range as given above is ambiguous, but the second and third bullets above together remove the ambiguity.

A character range may also be written in the form *s-e*, identifying the set that contains all XML characters with UCS code points greater than or equal to the code point of *s*, but not greater than the code point of *e*.

s-e is a valid character range iff:

- *s* is a single character escape, or an XML character;
- *s* is not \
- If s is the first character in a character class expression, then *s* is not `^`
- *e* is a single character escape, or an XML character;
- *e* is not \ or `[`; and

- The code point of *e* is greater than or equal to the code point of *s*;

The code point of a single character escape is the code point of the single character in the set of characters that it identifies.

F.1.1. Character Class Escapes

A *character class escape* is a short sequence of characters that identifies predefined character class. The valid character class escapes are the single character escapes, the multi-character escapes, and the category escapes (including the block escapes).

Character Class Escape

[23] charClassEsc ::= (| | |)

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

[24] SingleCharEsc ::= '\' [nrt\|.?*+(){}#x2D#x5B#x5D#x5E]

<table>
<tr><th>The valid single character escapes are:</th><th>Identifying the set of characters C(R) containing:</th></tr>
<tr><td>\n</td><td>the newline character (#xA)</td></tr>
<tr><td>\r</td><td>the return character (#xD)</td></tr>
<tr><td>\t</td><td>the tab character (#x9)</td></tr>
<tr><td>\\</td><td>\</td></tr>
<tr><td>\|</td><td>|</td></tr>
<tr><td>\.</td><td>.</td></tr>
<tr><td>\-</td><td>-</td></tr>
<tr><td>\^</td><td>^</td></tr>
<tr><td>\?</td><td>?</td></tr>
<tr><td>*</td><td>*</td></tr>
<tr><td>\+</td><td>+</td></tr>
<tr><td>\{</td><td>{</td></tr>
<tr><td>\}</td><td>}</td></tr>
<tr><td>\(</td><td>(</td></tr>
<tr><td>\)</td><td>)</td></tr>
<tr><td>\[</td><td>[</td></tr>
<tr><td>\]</td><td>]</td></tr>
</table>

[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

[25] catEsc ::= '\p{' '}'

[26] complEsc ::= '\P{' '}'

[27] charProp ::= |

[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] that is current at the time this specification became a W3C Recommendation. However, implementors are encouraged to support the character properties defined in any future version.

The following table specifies the recognized values of the "General Category" property.

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

[28]	IsCategory	::=	| | | | | |
[29]	Letters	::=	'L' [ultmo]?
[30]	Marks	::=	'M' [nce]?
[31]	Numbers	::=	'N' [dlo]?
[32]	Punctuation	::=	'P' [cdseifo]?
[33]	Separators	::=	'Z' [slp]?
[34]	Symbols	::=	'S' [mcko]?
[35]	Others	::=	'C' [cfon]?

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.

[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

[36]	IsBlock	::=	'Is' [a-zA-Z0-9#x2D]+

The following table specifies the recognized block names (for more information, see the "Blocks.txt" file in [Unicode Database]).

Start Code	End Code	Block Name		Start Code	End Code	Block Name
#x0000	#x007F	BasicLatin		#x0080	#x00FF	Latin-1Supplement
#x0100	#x017F	LatinExtended-A		#x0180	#x024F	LatinExtended-B
#x0250	#x02AF	IPAExtensions		#x02B0	#x02FF	SpacingModifierLetters
#x0300	#x036F	CombiningDiacriticalMarks		#x0370	#x03FF	Greek
#x0400	#x04FF	Cyrillic		#x0530	#x058F	Armenian
#x0590	#x05FF	Hebrew		#x0600	#x06FF	Arabic

#x0700	#x074F	Syriac		#x0780	#x07BF	Thaana
#x0900	#x097F	Devanagari		#x0980	#x09FF	Bengali
#x0A00	#x0A7F	Gurmukhi		#x0A80	#x0AFF	Gujarati
#x0B00	#x0B7F	Oriya		#x0B80	#x0BFF	Tamil
#x0C00	#x0C7F	Telugu		#x0C80	#x0CFF	Kannada
#x0D00	#x0D7F	Malayalam		#x0D80	#x0DFF	Sinhala
#x0E00	#x0E7F	Thai		#x0E80	#x0EFF	Lao
#x0F00	#x0FFF	Tibetan		#x1000	#x109F	Myanmar
#x10A0	#x10FF	Georgian		#x1100	#x11FF	HangulJamo
#x1200	#x137F	Ethiopic		#x13A0	#x13FF	Cherokee
#x1400	#x167F	UnifiedCanadianAboriginalSyllab-ics		#x1680	#x169F	Ogham
#x16A0	#x16FF	Runic		#x1780	#x17FF	Khmer
#x1800	#x18AF	Mongolian		#x1E00	#x1EFF	LatinExtendedAdditional
#x1F00	#x1FFF	GreekExtended		#x2000	#x206F	GeneralPunctuation
#x2070	#x209F	SuperscriptsandSubscripts		#x20A0	#x20CF	CurrencySymbols
#x20D0	#x20FF	CombiningMarksforSymbols		#x2100	#x214F	LetterlikeSymbols
#x2150	#x218F	NumberForms		#x2190	#x21FF	Arrows
#x2200	#x22FF	MathematicalOperators		#x2300	#x23FF	MiscellaneousTechnical
#x2400	#x243F	ControlPictures		#x2440	#x245F	OpticalCharacterRecognition
#x2460	#x24FF	EnclosedAlphanumerics		#x2500	#x257F	BoxDrawing
#x2580	#x259F	BlockElements		#x25A0	#x25FF	GeometricShapes
#x2600	#x26FF	MiscellaneousSymbols		#x2700	#x27BF	Dingbats
#x2800	#x28FF	BraillePatterns		#x2E80	#x2EFF	CJKRadicalsSupplement
#x2F00	#x2FDF	KangxiRadicals		#x2FF0	#x2FFF	IdeographicDescriptionCharac-ters
#x3000	#x303F	CJKSymbolsandPunctuation		#x3040	#x309F	Hiragana
#x30A0	#x30FF	Katakana		#x3100	#x312F	Bopomofo
#x3130	#x318F	HangulCompatibilityJamo		#x3190	#x319F	Kanbun
#x31A0	#x31BF	BopomofoExtended		#x3200	#x32FF	EnclosedCJKLettersandMonths
#x3300	#x33FF	CJKCompatibility		#x3400	#x4DB5	CJKUnifiedIdeographsExten-sionA
#x4E00	#x9FFF	CJKUnifiedIdeographs		#xA000	#xA48F	YiSyllables
#xA490	#xA4CF	YiRadicals		#xAC00	#xD7A3	HangulSyllables
				#xE000	#xF8FF	PrivateUse
#xF900	#xFAFF	CJKCompatibilityIdeographs		#xFB00	#xFB4F	AlphabeticPresentationForms
#xFB50	#xFDFF	ArabicPresentationForms-A		#xFE20	#xFE2F	CombiningHalfMarks
#xFE30	#xFE4F	CJKCompatibilityForms		#xFE50	#xFE6F	SmallFormVariants
#xFE70	#xFEFE	ArabicPresentationForms-B		#xFEFF	#xFEFF	Specials
#xFF00	#xFFEF	HalfwidthandFullwidthForms		#xFFF0	#xFFFD	Specials

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] is subject to future revision. For example, the grouping of code points into blocks might be updated. All minimally conforming processors must support the blocks defined in the version of [Unicode Database] that is current at the time this specification became a W3C Recommendation. However, implementors are encouraged to support the blocks defined in any future version of the Unicode Standard.

For example, the block escape for identifying the ASCII characters is `\p{IsBasicLatin}`.

A *multi-character escape* provides a simple way to identify a commonly used set of characters:

Multi-Character Escape

[37] MultiCharEsc ::= '\' [sSiIcCdDwW]

[38] 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 Letter | '_' | ':'
\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]

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.

Appendix G. 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.

Editor Note :

An XSL macro is used to collect definitions from throughout the spec and gather them here for easy reference.

Appendix H. References

H.1. Normative

XML Base

World Wide Web Consortium. XML Base. Available at: http://www.w3.org/TR/2001/REC-xmlbase-20010627/

IEEE 754-1985

IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. See http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html

XML Linking Language

World Wide Web Consortium. XML Linking Language (XLink). Available at: http://www.w3.org/TR/2001/REC-xlink-20010627/. Note: only the URI reference escaping procedure defined in Section 5.4 is normatively referenced.

XML 1.0 (Second Edition)

World Wide Web Consortium. *Extensible Markup Language (XML) 1.0, Second Edition*. Available at: http://www.w3.org/TR/2000/WD-xml-2e-20000814

XML Schema Part 1: Structures

XML Schema Part 1: Structures. Available at: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html

XML Schema Requirements

World Wide Web Consortium. XML Schema Requirements. Available at: http://www.w3.org/TR/1999/NOTE-xml-schema-req-19990215

Namespaces in XML

World Wide Web Consortium. *Namespaces in XML*. Available at: http://www.w3.org/TR/1999/REC-xml-names-19990114/

RFC 2396

Tim Berners-Lee, et. al. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax.*. 1998. Available at: http://www.ietf.org/rfc/rfc2396.txt

RFC 2732

RFC 2732: Format for Literal IPv6 Addresses in URL's. 1999. Available at: http://www.ietf.org/rfc/rfc2732.txt

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

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

Unicode Database

The Unicode Consortium. *The Unicode Character Database*. Available at: http://www.unicode.org/Public/3.1-Update/UnicodeCharacterDatabase-3.1.0.html

H.2. Non-normative

IETF INTERNET-DRAFT: IRIs

M. Dürst and M. Suignard . *Internationalized Resource Identifiers* 2002. Available at: http://www.w3.org/International/iri-edit/draft-duerst-iri-04.txt

Ruby

World Wide Web Consortium. Ruby Annotation. Available at: http://www.w3.org/TR/2001/WD-ruby-20010216/

HTML 4.01

World Wide Web Consortium. Hypertext Markup Language, version 4.01. Available at: http://www.w3.org/TR/1999/REC-html401-19991224/

XML Schema Language: Part 0 Primer

World Wide Web Consortium. XML Schema Language: Part 0 Primer. Available at: http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/primer.html

Unicode Regular Expression Guidelines

Mark Davis. *Unicode Regular Expression Guidelines*, 1988. Available at: http://www.unicode.org/unicode/reports/tr18/

Perl

The Perl Programming Language. See http://www.perl.com/pub/language/info/software.html

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.ch/cate/d26197.html

International Earth Rotation Service (IERS)

International Earth Rotation Service (IERS). See http://maia.usno.navy.mil

ISO 8601

ISO (International Organization for Standardization). *Representations of dates and times, 1988-06-15.*

ISO 8601:1998 Draft Revision

ISO (International Organization for Standardization). *Representations of dates and times, draft revision, 1998.*

ISO 8601:2000 Second Edition

ISO (International Organization for Standardization). *Representations of dates and times, second edition, 2000-12-15.*

ISO 11404

ISO (International Organization for Standardization). *Language-independent Datatypes.* See http://www.iso.ch/cate/d19346.html

RDF Schema

World Wide Web Consortium. *RDF Schema Specification.* Available at: http://www.w3.org/TR/2000/CR-rdf-schema-20000327/

U.S. Naval Observatory Time Service Department

Information about Leap Seconds Available at: http://tycho.usno.navy.mil/leapsec.990505.html

XSL

World Wide Web Consortium. *Extensible Stylesheet Language (XSL).* Available at: http://www.w3.org/TR/2000/CR-xsl-20001121/

Character Model

Martin J. Dürst and François Yergeau, eds. Character Model for the World Wide Web. World Wide Web Consortium Working Draft. 2001. Available at: http://www.w3.org/TR/2001/WD-charmod-20010126/

Gay, DM (1990)

David M. Gay. *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions.* AT&T Bell Laboratories Numerical Analysis Manuscript 90-10, November 1990. Available at: http://cm.bell-labs.com/cm/cs/doc/90/4-10.ps.gz

Appendix I. Acknowledgements (non-normative)

The following have contributed material to the first edition 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. From February 2001 until May 2004 it was supported by Microsoft.

The editors acknowledge the members of the XML Schema Working Group, the members of other W3C Working Groups, and industry experts in other forums who have contributed directly or indirectly to the process or content of creating this document. The Working Group is particularly grateful to Lotus Development Corp. and IBM for providing teleconferencing facilities.

At the time the first edition of this specification was published, the members of the XML Schema Working Group were:

Jim Barnette, Defense Information Systems Agency (DISA); Paul V. Biron, Health Level Seven; Don Box, DevelopMentor; Allen Brown, Microsoft; Lee Buck, TIBCO Extensibility; Charles E. Campbell, Informix; Wayne Carr, Intel; Peter Chen, Bootstrap Alliance and LSU; David Cleary, Progress Software; Dan Connolly, W3C (staff contact); Ugo Corda, Xerox; Roger L. Costello, MITRE; Haavard Danielson,

Progress Software; Josef Dietl, Mozquito Technologies; David Ezell, Hewlett-Packard Company ; Alexander Falk, Altova GmbH; David Fallside, IBM; Dan Fox, Defense Logistics Information Service (DLIS); Matthew Fuchs, Commerce One; Andrew Goodchild, Distributed Systems Technology Centre (DSTC Pty Ltd); Paul Grosso, Arbortext, Inc; Martin Gudgin, DevelopMentor; Dave Hollander, Contivo, Inc (co-chair); Mary Holstege, Invited Expert; Jane Hunter, Distributed Systems Technology Centre (DSTC Pty Ltd); Rick Jelliffe, Academia Sinica; Simon Johnston, Rational Software; Bob Lojek, Mozquito Technologies; Ashok Malhotra, Microsoft; Lisa Martin, IBM; Noah Mendelsohn, Lotus Development Corporation; Adrian Michel, Commerce One; Alex Milowski, Invited Expert; Don Mullen, TIBCO Extensibility; Dave Peterson, Graphic Communications Association; Jonathan Robie, Software AG; Eric Sedlar, Oracle Corp.; C. M. Sperberg-McQueen, W3C (co-chair); Bob Streich, Calico Commerce; William K. Stumbo, Xerox; Henry S. Thompson, University of Edinburgh; Mark Tucker, Health Level Seven; Asir S. Vedamuthu, webMethods, Inc; Priscilla Walmsley, XMLSolutions; Norm Walsh, Sun Microsystems; Aki Yoshida, SAP AG; Kongyi Zhou, Oracle Corp.
The XML Schema Working Group has benefited in its work from the participation and contributions of a number of people not currently members of the Working Group, including in particular those named below. Affiliations given are those current at the time of their work with the WG.

Paula Angerstein, Vignette Corporation; David Beech, Oracle Corp.; Gabe Beged-Dov, Rogue Wave Software; Greg Bumgardner, Rogue Wave Software; Dean Burson, Lotus Development Corporation; Mike Cokus, MITRE; Andrew Eisenberg, Progress Software; Rob Ellman, Calico Commerce; George Feinberg, Object Design; Charles Frankston, Microsoft; Ernesto Guerrieri, Inso; Michael Hyman, Microsoft; Renato Iannella, Distributed Systems Technology Centre (DSTC Pty Ltd); Dianne Kennedy, Graphic Communications Association; Janet Koenig, Sun Microsystems; Setrag Khoshafian, Technology Deployment International (TDI); Ara Kullukian, Technology Deployment International (TDI); Andrew Layman, Microsoft; Dmitry Lenkov, Hewlett-Packard Company; John McCarthy, Lawrence Berkeley National Laboratory; Murata Makoto, Xerox; Eve Maler, Sun Microsystems; Murray Maloney, Muzmo Communication, acting for Commerce One; Chris Olds, Wall Data; Frank Olken, Lawrence Berkeley National Laboratory; Shriram Revankar, Xerox; Mark Reinhold, Sun Microsystems; John C. Schneider, MITRE; Lew Shannon, NCR; William Shea, Merrill Lynch; Ralph Swick, W3C; Tony Stewart, Rivcom; Matt Timmermans, Microstar; Jim Trezzo, Oracle Corp.; Steph Tryphonas, Microstar
The lists given above pertain to the first edition. At the time work on this second edition was completed, the membership of the Working Group was:

Leonid Arbouzov, Sun Microsystems; Jim Barnette, Defense Information Systems Agency (DISA); Paul V. Biron, Health Level Seven; Allen Brown, Microsoft; Charles E. Campbell, Invited expert; Peter Chen, Invited expert; Tony Cincotta, NIST; David Ezell, National Association of Convenience Stores; Matthew Fuchs, Invited expert; Sandy Gao, IBM; Andrew Goodchild, Distributed Systems Technology Centre (DSTC Pty Ltd); Xan Gregg, Invited expert; Mary Holstege, Mark Logic; Mario Jeckle, DaimlerChrysler; Marcel Jemio, Data Interchange Standards Association; Kohsuke Kawaguchi, Sun Microsystems; Ashok Malhotra, Invited expert; Lisa Martin, IBM; Jim Melton, Oracle Corp; Noah Mendelsohn, IBM; Dave Peterson, Invited expert; Anli Shundi, TIBCO Extensibility; C. M. Sperberg-McQueen, W3C (co-chair); Hoylen Sue, Distributed Systems Technology Centre (DSTC Pty Ltd); Henry S. Thompson, University of Edinburgh; Asir S. Vedamuthu, webMethods, Inc; Priscilla Walmsley, Invited expert; Kongyi Zhou, Oracle Corp.
We note with sadness the accidental death of Mario Jeckle shortly after the completion of work on this document. In addition to those named above, several people served on the Working Group during the development of this second edition:

Oriol Carbo, University of Edinburgh; Tyng-Ruey Chuang, Academia Sinica; Joey Coyle, Health Level 7; Tim Ewald, DevelopMentor; Nelson Hung, Corel; Melanie Kudela, Uniform Code Council; Matthew

MacKenzie, XML Global; Cliff Schmidt, Microsoft; John Stanton, Defense Information Systems Agency; John Tebbutt, NIST; Ross Thompson, Contivo; Scott Vorthmann, TIBCO Extensibility