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. The latest status of this document series is maintained at the W3C.

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 XML Schema WG members. Different parts of this specification have different editors.

This version of this document incorporates some editorial changes from earlier versions.

Please report errors in this document to www-xml-schema-comments@w3.org (archive). The list of known errors in this specification is available at http://www.w3.org/2001/05/xmlschema-errata.

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.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR/.
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.

```
Data oriented
<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>
</invoice>

Document oriented
<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 immediately.
    Either email me at <email>mailto:paul.v.biron@kp.org</email>.
  </body>
</memo>
```
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:

[Definition:] 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)]

[Definition:] may
Conforming documents and processors are permitted to but need not behave as described.

[Definition:] match
(Of strings or names:) Two strings or names being compared must be identical. Characters with multiple possible representations in ISO/IEC 10646 (e.g. characters with both precomposed and base+diascritic 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.

[Definition:] must
Conforming documents and processors are required to behave as described; otherwise they are in error.

[Definition:] 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:
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

[Definition:] 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 facet s that characterize properties of the value space, individual values or lexical items.

2.2 Value space

[Definition:] 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 space s 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 space s that are recognized by this specification are defined in Fundamental facets (§2.4.1).

2.3 Lexical space

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

[Definition:] 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 float 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.

NOTE: The literals in the lexical space s 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
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 float 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.

[Definition:] 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
2.4.1 Fundamental facets
2.4.2 Constraining or Non-fundamental facets

[Definition:] 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

[Definition:] 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 Fundamental Facets (§4.2).

2.4.2 Constraining or Non-fundamental facets

[Definition:] 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 facet s to a base type is described in Derivation by restriction (§4.1.2.1).

All constraining facets are fully described in Constraining Facets (§4.3).

2.5 Datatype dichotomies
2.5.1 Atomic vs. list vs. union datatypes
2.5.2 Primitive vs. derived datatypes
2.5.3 Built-in vs. user-derived datatypes

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.

- [Definition:] Atomic datatypes are those having values which are regarded by this specification as being indivisible.
• [Definition:] List datatypes are those having values each of which consists of a finite-length (possibly empty) sequence of values of an atomic datatype.
• [Definition:] Union datatypes are those whose value space and lexical space are the union of the value space and lexical space 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 (NMTOKEN); while a sequence of such tokens could be the value of a list datatype (NMTOKENS).

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 white space separated sequence of literals of the atomic datatype of the items in the list (where whitespace matches S in [XML 1.0 (Second Edition)]).

[Definition:] The atomic datatype that participates in the definition of a list datatype is known as the itemType of that list datatype.

Example

```xml
<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 whitespace (such as string or anyURI). In such a case, regardless of the input, list items will be separated at whitespace boundaries.

Example

```xml
<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 facet s 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.

The canonical-lexical-representation 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 space s and lexical space s of its memberTypes. union datatypes are always derived. Currently, there are no built-in union datatypes.

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

```
<attributeGroup name="occurs">
  <attribute name="minOccurs" type="nonNegativeInteger" default="1"/>
  <attribute name="maxOccurs">
    <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 datatype s can participate in a union type.

[Definition:] 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 QNames 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.

Example
For example, given the definition below, the first instance of the <size> element validates correctly as an integer (§3.3.13), the second and third as string (§3.2.1).

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

<size>1</size>
<size>large</size>
The canonical-lexical-representation for a union datatype is defined as the lexical form in which the values have the canonical lexical representation of the appropriate memberTypes.

NOTE: 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.

- [Definition:] Primitive datatypes are those that are not defined in terms of other datatypes; they exist ab initio.
- [Definition:] Derived datatypes are those that are defined in terms of other datatypes.

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

[Definition:] There exists a conceptual datatype, whose name is anySimpleType, that is the simple version of the ur-type definition from [XML Schema Part 1: Structures]. anySimpleType can be considered as the base type of all primitive types. The value space of anySimpleType can be considered to be the union of the value space s of all 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, integer is derived from decimal.

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

As described in more detail in XML Representation of Simple Type Definition Schema Components (§4.1.2), each user-derived datatype must be defined in terms of another datatype in one of three ways: 1) by assigning constraining facet s 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 space its memberTypes.

2.5.2.1 Derived by restriction

[Definition:] A datatype is said to be derived by restriction from another datatype when values for zero or more constraining facet s are specified that serve to constrain its value space and/or its lexical space to a subset of those of its base type.

[Definition:] 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 union ing their value space s and, consequently, their lexical space s.
2.5.3 Built-in vs. user-derived datatypes

- [Definition:] Built-in datatypes are those which are defined in this specification, and can be either primitive or derived.
- [Definition:] 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.

**NOTE:** 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 int datatype, the URI is:

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

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

1. the base URI is the URI of the XML Schema namespace
2. the fragment identifier is the name of the facet
For example, to address the maxInclusive facet, the URI is:

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

Additionally, each facet usage in a built-in datatype definition can be uniquely addressed via a URI constructed as follows:

1. the base URI is the URI of the XML Schema namespace
2. the fragment identifier is the name of the datatype, followed by a period ("."), followed by the name of the facet

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

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

### 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

#### 3.2.1 string

#### 3.2.2 boolean

#### 3.2.3 decimal

#### 3.2.4 float

#### 3.2.5 double

#### 3.2.6 duration

#### 3.2.7 dateTime

#### 3.2.8 time

#### 3.2.9 date

#### 3.2.10 gYearMonth

#### 3.2.11 gYear

#### 3.2.12 gMonthDay

#### 3.2.13 gDay

#### 3.2.14 gMonth

#### 3.2.15 hexBinary

#### 3.2.16 base64Binary

#### 3.2.17 anyURI

#### 3.2.18 QName

#### 3.2.19 NOTATION

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

[Definition:] 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.

**NOTE:** Many human languages have writing systems that require child elements for control of aspects such as bidirectional formatting or ruby annotation (see [Ruby] and Section 8.2.4 Overiding 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 2 Primer].

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

3.2.1.1 Constraining facets

**string** has the following constraining facets:

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

3.2.1.2 Derived datatypes

The following built-in datatypes are derived from **string**:

- **normalizedString**

3.2.2 boolean

[Definition:] **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

**boolean** has the following constraining facets:

- **pattern**
- **whiteSpace**

3.2.3 decimal

[Definition:] **decimal** represents arbitrary precision decimal numbers. The value space of **decimal** is the set of values \(i \times 10^{-n}\), where \(i\) and \(n\) are integers such that \(n \geq 0\). The order-relation on **decimal** is: \(x < y\) if \(y - x\) is positive.

[Definition:] The value space of types derived from **decimal** with a value for **totalDigits** of \(p\) is the set of values \(i \times 10^{-n}\), where
\(n\) and \(i\) are integers such that \(p \geq n \geq 0\) and the number of significant decimal digits in \(i\) is less than or equal to \(p\).

[Definition:] The value space of types derived from \texttt{decimal} with a value for fractionDigits of \(s\) is the set of values \(i \times 10^{-n}\), where \(i\) and \(n\) are integers such that \(0 \leq n \leq s\).

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

3.2.3.1 Lexical representation

\texttt{decimal} has a lexical representation consisting of a finite-length sequence of decimal digits (\#x30-\#x39) separated by a period as a decimal indicator. If totalDigits is specified, the number of digits must be less than or equal to totalDigits. If fractionDigits is specified, the number of digits following the decimal point must be less than or equal to the fractionDigits. An optional leading sign 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 \texttt{decimal} is defined by prohibiting certain options from the Lexical representation (§3.2.3.1). 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

\texttt{decimal} has the following constraining facets:

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

3.2.3.4 Derived datatypes

The following built-in datatypes are derived from \texttt{decimal}:

- \texttt{integer}

3.2.4 float

[Definition:] float corresponds to the IEEE single-precision 32-bit floating point type [IEEE 754-1985]. The basic value space of \texttt{float} consists of the values \(m \times 2^e\), where \(m\) is an integer whose absolute value is less than 2^{24}, and \(e\) is an integer between -149 and 104, inclusive. In addition to the basic value space described above, the value space of \texttt{float} also contains the following special values: positive and negative zero, positive and negative infinity and not-a-number. The order relation on float is: \(x < y\) iff \(y - x\) is positive. Positive zero is greater than negative zero. Not-a-number equals itself and is greater than all float values values including positive infinity.

A literal in the lexical space representing a decimal number \(d\) maps to the normalized value in the value space of \texttt{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

\texttt{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 integer. The mantissa must be a decimal number. The representations for exponent and mantissa must follow the lexical rules for integer and decimal. If the "E" or "e" and the following exponent are omitted, an exponent value of 0 is assumed.

The special values positive and negative zero, positive and negative infinity and not-a-number have lexical representations 0, -0, 1NF, -1NF and NaN, respectively.

For example, -1E4, 1267.43233E12, 12.78e-2, 12 and 1NF 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 Lexical representation (§3.2.4.1). Specifically, the exponent must be indicated by "E". Leading zeroes and the preceding optional "+" sign are prohibited in the exponent. For the mantissa, the preceding optional "+" sign is prohibited and the decimal point is required. For the exponent, the preceding optional "+" sign is prohibited. Leading and trailing zeroes are prohibited subject to the following: number representations must be normalized such that there is a single digit to the left of the decimal point and at least a single digit to the right of the decimal point.

### 3.2.4.3 Constraining facets

float has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

### 3.2.5 double

[Definition:] The double datatype corresponds to 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 special values: positive and negative zero, positive and negative infinity and not-a-number. The order relation on double is: \( x < y \) iff \( y - x \) is positive. Positive zero is greater than negative zero. Not-a-number equals itself and is greater greater than all double values including positive infinity.

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 integer and decimal. If the "E" or "e" and the following exponent are omitted, an exponent value of 0 is assumed.

The special values positive and negative zero, positive and negative infinity and not-a-number have lexical representations 0, -0, 1NF, -1NF and NaN, respectively.

For example, -1E4, 1267.43233E12, 12.78e-2, 12 and 1NF 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 Lexical representation (§3.2.5.1). Specifically, the exponent must be indicated by "E". Leading zeroes and the preceding optional "+" sign are prohibited in the
exponent. For the mantissa, the preceding optional "+" sign is prohibited and the decimal point is required. For the exponent, the preceding optional "+" sign is prohibited. Leading and trailing zeroes are prohibited subject to the following: number representations must be normalized such that there is a single digit to the left of the decimal point and at least a single digit to the right of the decimal point.

3.2.5.3 Constraining facets

double has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.6 duration

[Definition:] 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.

3.2.6.1 Lexical representation

The lexical representation for duration is the [ISO 8601] extended format PnYnMnDTnH nMnS, where nY represents the number of years, nM the number of months, nD the number of days, 'T' is the date/time separator, nH the number of hours, nM the number of minutes and nS 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 integer. Similarly, the value of the Seconds component allows an arbitrary decimal. 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 ISO 8601 Date and Time Formats (SD).

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' shall be absent 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 dateTime 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 Adding durations to dateTimes (§E).

- 1696-09-01T00:00:00Z
- 1697-02-01T00:00:00Z
- 1903-03-01T00:00:00Z
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 Adding durations to dateTimes (§E), they are still totally ordered.

<table>
<thead>
<tr>
<th>Relation</th>
<th>P1Y</th>
<th>P364D</th>
<th>&lt;&gt;</th>
<th>P365D</th>
<th>&lt;&gt;</th>
<th>P366D</th>
<th>&lt;&gt;</th>
<th>P367D</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1M</td>
<td>&gt;</td>
<td>P27D</td>
<td>&lt;&gt;</td>
<td>P28D</td>
<td>&lt;</td>
<td>P29D</td>
<td>&lt;</td>
<td>P30D</td>
</tr>
<tr>
<td>P5M</td>
<td>&gt;</td>
<td>P149D</td>
<td>&lt;&gt;</td>
<td>P150D</td>
<td>&lt;&gt;</td>
<td>P151D</td>
<td>&lt;&gt;</td>
<td>P152D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Days</th>
<th>Months 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>...</th>
</tr>
</thead>
</table>
|      | 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 minInclusive, minExclusive, maxInclusive and maxExclusive 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.

```xml
<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

duration has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.7 dateTime

[Definition:] dateTime represents a specific instant of time. The value space of dateTime is the space of Combinations of date
and time of day values as defined in § 5.4 of [ISO 8601].

3.2.7.1 Lexical representation

A single lexical representation, which is a subset of the lexical representations allowed by [ISO 8601], is allowed for dateTime. This lexical representation is the [ISO 8601] extended format CCYY-MM-DDThh:mm:ss where "CC" represents the century, "YY" the year, "MM" the month and "DD" the day, preceded by an optional leading "-" sign to indicate a negative number. If the sign is omitted, "+" is assumed. The letter "T" is the date/time separator and "hh", "mm", "ss" represent hour, minute and second respectively. Additional digits can be used to increase the precision of fractional seconds if desired i.e. the format ss.ss... with any number of digits after the decimal point is supported. The fractional seconds part is optional; other parts of the lexical form are not optional. To accommodate year values greater than 9999 additional digits can be added to the left of this representation. Leading zeros are required if the year would otherwise have fewer than four digits; otherwise they are forbidden. The year 0000 is prohibited.

The CCYY field must have at least four digits, the MM, DD, SS, hh, mm and ss fields exactly two digits each (not counting fractional seconds); leading zeroes must be used if the field would otherwise have too few digits.

This representation may be immediately followed by a "Z" to indicate Coordinated Universal Time (UTC) or, to indicate the time zone, i.e. the difference between the local time and Coordinated Universal Time, immediately followed by a sign, + or -, followed by the difference from UTC represented as hh:mm (note: the minutes part is required). See ISO 8601 Date and Time Formats (SD) for details about legal values in the various fields. If the time zone is included, both hours and minutes must be present.

For example, to indicate 1:20 pm on May the 31st, 1999 for Eastern Standard Time which is 5 hours behind Coordinated Universal Time (UTC), one would write: 1999-05-31T13:20:00-05:00.

3.2.7.2 Canonical representation

The canonical representation for dateTime is defined by prohibiting certain options from the Lexical representation (§3.2.7.1). Specifically, either the time zone must be omitted or, if present, the time zone must be Coordinated Universal Time (UTC) indicated by a "Z".

3.2.7.3 Order relation on dateTime

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 dateTimees 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 Adding durations to dates and times (§4).

- 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 in (year, month, day, hour, minute, second)
   1. If P[i] and Q[i] are both not specified, continue to the next i
   2. If P[i] is not specified and Q[i] is, or vice versa, stop and return P <> Q
   3. If P[i] < Q[i], stop and return P < Q
   4. 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 $P < (Q \text{ with time zone} +14:00)$
2. $P > Q$ if $P > (Q \text{ with time zone} -14:00)$
3. $P <> Q$ otherwise, that is, if $(Q \text{ with time zone} +14:00) < P < (Q \text{ with time zone} -14:00)$

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

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

Examples:

<table>
<thead>
<tr>
<th>Determinate</th>
<th>Indeterminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-15T00:00:00 &lt; 2000-02-15T00:00:00</td>
<td>2000-01-01T12:00:00 &lt;&gt; 1999-12-31T23:00:00Z</td>
</tr>
<tr>
<td>2000-01-15T12:00:00 &lt; 2000-01-16T12:00:00Z</td>
<td>2000-01-01T12:00:00 &lt;&gt; 2000-01-16T12:00:00Z</td>
</tr>
<tr>
<td>2000-01-16T00:00:00 &lt;&gt; 2000-01-16T12:00:00Z</td>
<td>2000-01-01T12:00:00 &lt;&gt; 2000-01-16T12:00:00Z</td>
</tr>
</tbody>
</table>

3.2.7.4 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.

3.2.7.5 Constraining facets

dateTime has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.8 time

[Definition:] 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 Order relation on dateTime (§3.2.7.3) using an arbitrary date. See also Adding durations to dateTimes (§E). Pairs of time values with or without time zone indicators are totally ordered.

3.2.8.1 Lexical representation

The lexical representation for time is the left truncated lexical representation for dateTime: 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 ISO 8601 Date and Time Formats (SD).

3.2.8.2 Canonical representation

The canonical representation for time is defined by prohibiting certain options from the Lexical representation (§3.2.8.1). 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

time has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.9 date

[Definition:] date represents a calendar date. The value space of date is the set of Gregorian calendar dates as defined in § 5.2.1 of [ISO 8601]. Specifically, it is a set of one-day long, non-periodic instances e.g. lexical 1999-10-26 to represent the calendar date 1999-10-26, independent of how many hours this day has.

Since the lexical representation allows an optional time zone indicator, date 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 date values are considered as periods of time, the order relation on date values is the order relation on their starting instants. This is discussed in Order relation on dateTime (§3.2.7.3). See also Adding durations to dateTimes (§E). Pairs of date values with or without time zone indicators are totally ordered.

3.2.9.1 Lexical representation

The lexical representation for date is the reduced (right truncated) lexical representation for dateTime: CCYY-MM-DD. No left truncation is allowed. An optional following time zone qualifier is allowed as for dateTime. 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 May the 31st, 1999, one would write: 1999-05-31. See also ISO 8601 Date and Time Formats (§D).

3.2.9.2 Constraining facets

date has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.10 gYearMonth

[Definition:] 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 Order relation on dateTime (§3.2.7.3). See also Adding durations to dateTimes (§E). Pairs of gYearMonth values with or without time zone indicators are totally ordered.

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

3.2.10.1 Lexical representation

The lexical representation for **gYearMonth** is the reduced (right truncated) lexical representation for **dateTime**: 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 ISO 8601 Date and Time Formats (§D).

3.2.10.2 Constraining facets

**gYearMonth** has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.11 gYear

[Definition:] **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 Order relation on **dateTime** (§3.2.7.3). See also Adding durations to **dateTime** (§E). Pairs of **gYear** values with or without time zone indicators are totally ordered.

NOTE: Because years in one calendar only rarely correspond to years in other calendars, values of this type are not, in general, convertible to simple values corresponding to years in other calendars. This type should therefore be used with caution in contexts where conversion to other calendars is desired.

3.2.11.1 Lexical representation

The lexical representation for **gYear** is the reduced (right truncated) lexical representation for **dateTime**: CCYY. No left truncation is allowed. An optional following time zone qualifier is allowed as for **dateTime**. 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 ISO 8601 Date and Time Formats (§D).

3.2.11.2 Constraining facets

**gYear** has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.12 gMonthDay
[Definition:] **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, the order relation on **gMonthDay** values is the order relation on their starting instants. This is discussed in **Order relation on dateTime** (§3.2.7.3). See also **Adding durations to dateTimes** (§E). Pairs of **gMonthDay** values with or without time zone indicators are totally ordered.

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

### 3.2.12.1 Lexical representation

The lexical representation for **gMonthDay** is the left truncated lexical representation for **date**: --MM-DD. An optional following time zone qualifier is allowed as for **date**. No preceding sign is allowed. No other formats are allowed. See also [ISO 8601 Date and Time Formats (§D)].

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 every year.

### 3.2.12.2 Constraining facets

**gMonthDay** has the following constraining facets:

- **pattern**
- **enumeration**
- **whiteSpace**
- **maxInclusive**
- **maxExclusive**
- **minInclusive**
- **minExclusive**

### 3.2.13 gDay

[Definition:] **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, the order relation on **gDay** values is the order relation on their starting instants. This is discussed in **Order relation on dateTime** (§3.2.7.3). See also **Adding durations to dateTimes** (§E). Pairs of **gDay** values with or without time zone indicators are totally ordered.

**NOTE:** 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 **date**: ---DD. An optional following time zone qualifier is allowed as for **date**. No preceding sign is allowed. No other formats are allowed. See also [ISO 8601 Date and Time Formats (§D)].
3.2.13.2 Constraining facets

gDay has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.14 gMonth

[Definition:] 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 Order relation on dateTime (§3.2.7.3). See also Adding durations to dateTimes (§E). Pairs of gMonth values with or without time zone indicators are totally ordered.

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

3.2.14.1 Lexical representation

The lexical representation for gMonth is the left and right truncated lexical representation for date: --MM--. An optional following time zone qualifier is allowed as for date. No preceding sign is allowed. No other formats are allowed. See also ISO 8601 Date and Time Formats (§D).

3.2.14.2 Constraining facets

gMonth has the following constraining facets:

- pattern
- enumeration
- whiteSpace
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

3.2.15 hexBinary

[Definition:] 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 [Lexical Representation (§3.2.15.1)](https://www.example.com). Specifically, the lower case hexadecimal digits ([a-f]) are not allowed.

### 3.2.15.3 Constraining facets

**hexBinary** has the following constraining facets:

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

### 3.2.16 base64Binary

[Definition:] **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 Content-Transfer-Encoding defined in Section 6.8 of [RFC 2045].

### 3.2.16.1 Constraining facets

**base64Binary** has the following constraining facets:

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

### 3.2.17 anyURI

[Definition:] **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 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.

**NOTE:** 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].

**NOTE:** 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

**anyURI** has the following constraining facets:
3.2.18 QName

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

**NOTE:** 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

QName has the following constraining facets:

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

3.2.19 NOTATION

[Definition:] NOTATION represents the NOTATION attribute type from [XML 1.0 (Second Edition)]. The value space of NOTATION is the set of QName. The lexical space of NOTATION is the set of all names of notations declared in the current schema.

**Schema Component Constraint: 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 Terminology (§1.4)) NOTATION should be used only on attributes.

3.2.19.1 Constraining facets

NOTATION has the following constraining facets:

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

3.3 Derived datatypes

3.3.1 normalizedString
3.3.2 token
3.3.3 language
3.3.4 NMTOKEN
3.3.5 NMTOKENS
3.3.6 Name
3.3.7 NCName
3.3.8 ID
3.3.9 IDREF
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 XML Representation of Simple Type Definition Schema Components (§4.1.2) and the complete definitions of the built-in derived datatypes are provided in Appendix A Schema for Datatype Definitions (normative) (§A).

3.3.1 normalizedString

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

3.3.1.1 Constraining facets

normalizedString has the following constraining facets:

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

3.3.1.2 Derived datatypes

The following built-in datatypes are derived from normalizedString:

- token

3.3.2 token

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

3.3.2.1 Constraining facets

token has the following constraining facets:

- length
- minLength
3.3.2.2 Derived datatypes

The following built-in datatypes are derived from token:

- language
- NMTOKEN
- Name

3.3.3 language

[Definition:] language represents natural language identifiers as defined by [RFC 1766]. The value space of language is the set of all strings that are valid language identifiers as defined in the language identification section of [XML 1.0 (Second Edition)]. The lexical space of language is the set of all strings that are valid language identifiers as defined in the language identification section of [XML 1.0 (Second Edition)]. The base type of language is token.

3.3.3.1 Constraining facets

language has the following constraining facets:

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

3.3.4 NMTOKEN

[Definition:] 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 token.

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

3.3.4.1 Constraining facets

NMTOKEN has the following constraining facets:

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

3.3.4.2 Derived datatypes

The following built-in datatypes are derived from NMTOKEN:

- NMTOKENS

3.3.5 NMTOKENS
[Definition:] **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 white space separated lists of tokens, of which each token is in the lexical space of NMTOKEN. The itemType of NMTOKENS is NMTOKEN.

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

### 3.3.5.1 Constraining facets

**NMTOKENS** has the following constraining facets:

- length
- minLength
- maxLength
- enumeration
- whiteSpace

### 3.3.6 Name

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

### 3.3.6.1 Constraining facets

**Name** has the following constraining facets:

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

### 3.3.6.2 Derived datatypes

The following built-in datatypes are derived from **Name**:

- NCName

### 3.3.7 NCName

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

### 3.3.7.1 Constraining facets

**NCName** has the following constraining facets:

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

### 3.3.7.2 Derived datatypes
The following built-in datatypes are derived from NCName:

- ID
- IDREF
- ENTITY

3.3.8 ID

[Definition:] 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 NCName.

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

3.3.8.1 Constraining facets

ID has the following constraining facets:

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

3.3.9 IDREF

[Definition:] 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 NCName.

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

3.3.9.1 Constraining facets

IDREF has the following constraining facets:

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

3.3.9.2 Derived datatypes

The following built-in datatypes are derived from IDREF:

- IDREFS

3.3.10 IDREFS

[Definition:] 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 IDREFs. The lexical space of IDREFS is the set of white space separated lists of tokens, of which each token is in the lexical space of IDREF. The itemType of IDREFS is IDREF.

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

3.3.10.1 Constraining facets
IDREFS has the following constraining facets:

- length
- minLength
- maxLength
- enumeration
- whiteSpace

3.3.11 ENTITY

[Definition:] 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 NCName.

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

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

3.3.11.1 Constraining facets

ENTITY has the following constraining facets:

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

3.3.11.2 Derived datatypes

The following built-in datatypes are derived from ENTITY:

- ENTITIES

3.3.12 ENTITIES

[Definition:] 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 white space separated lists of tokens, of which each token is in the lexical space of ENTITY. The itemType of ENTITIES is ENTITY.

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

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

3.3.12.1 Constraining facets

ENTITIES has the following constraining facets:

- length
- minLength
- maxLength
- enumeration
- whiteSpace

3.3.13 integer
integer is derived from decimal by fixing the value of fractionDigits to be 0. This results in the standard mathematical concept of the integer numbers. The value space of integer is the infinite set \{..., -2, -1, 0, 1, 2, ...\}. The base type of integer is decimal.

3.3.13.1 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 Lexical representation (§3.3.13.1). Specifically, the preceding optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.13.3 Constraining facets

integer has the following constraining facets:

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

3.3.13.4 Derived datatypes

The following built-in datatypes are derived from integer:

- nonPositiveInteger
- long
- nonNegativeInteger

3.3.14 nonPositiveInteger

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

3.3.14.1 Lexical representation

nonPositiveInteger has a lexical representation consisting of a negative sign ("-") followed by a finite-length sequence of decimal digits (#x30-#x39). If the sequence of digits consists of all zeros then the sign is optional. 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 Lexical representation (§3.3.14.1). Specifically, the negative sign ("-") is required with the token "0" and leading zeroes are prohibited.

3.3.14.3 Constraining facets

nonPositiveInteger has the following constraining facets:

- totalDigits
- fractionDigits
3.3.14.4 Derived datatypes

The following built-in datatypes are derived from nonPositiveInteger:

- negativeInteger

3.3.15 negativeInteger

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

3.3.15.1 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 Lexical representation (§3.3.15.1). Specifically, leading zeroes are prohibited.

3.3.15.3 Constraining facets

negativeInteger has the following constraining facets:

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

3.3.16 long

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

3.3.16.1 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 Lexical representation (§3.3.16.1). Specifically, the the optional “+” sign is prohibited and leading zeroes are prohibited.
3.3.16.3 Constraining facets

**long** has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- minInclusive
- minExclusive

3.3.16.4 Derived datatypes

The following built-in datatypes are derived from **long**:

- int

3.3.17 int

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

3.3.17.1 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 Lexical representation (§3.3.17.1). Specifically, the optional “+” sign is prohibited and leading zeroes are prohibited.

3.3.17.3 Constraining facets

**int** has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- minInclusive
- minExclusive

3.3.17.4 Derived datatypes

The following built-in datatypes are derived from **int**:

- short

3.3.18 short

[Definition:] **short** is derived from **int** by setting the value of maxInclusive to be 32767 and minInclusive to be -32768. The base type of **short** is **int**.
3.3.18.1 *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 *Lexical representation* (§3.3.18.1). Specifically, the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.18.3 *Constraining facets*

*short* has the following constraining facets:

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

3.3.18.4 *Derived datatypes*

The following built-in datatypes are derived from *short*:

- byte

3.3.19 *byte*

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

3.3.19.1 *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 *Lexical representation* (§3.3.19.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.19.3 *Constraining facets*

*byte* has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
- whiteSpace
- enumeration
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
3.3.20 nonNegativeInteger

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

3.3.20.1 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, "+" is assumed. 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 Lexical representation (§3.3.20.1). Specifically, the the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.20.3 Constraining facets

nonNegativeInteger has the following constraining facets:

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

3.3.20.4 Derived datatypes

The following built-in datatypes are derived from nonNegativeInteger:

- unsignedLong
- positiveInteger

3.3.21 unsignedLong

[Definition:] unsignedLong is derived from nonNegativeInteger by setting the value of maxInclusive to be 18446744073709551615. The base type of unsignedLong is nonNegativeInteger.

3.3.21.1 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 Lexical representation (§3.3.21.1). Specifically, leading zeroes are prohibited.

3.3.21.3 Constraining facets

unsignedLong has the following constraining facets:

- totalDigits
- fractionDigits
- pattern
3.3.21.4 Derived datatypes

The following built-in datatypes are derived from `<unsignedLong>`:

- `<unsignedInt>`

3.3.22 `<unsignedInt>`

[Definition:] `<unsignedInt>` is derived from `<unsignedLong>` by setting the value of `maxInclusive` to be 4294967295. The base type of `<unsignedInt>` is `<unsignedLong>`.

3.3.22.1 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 Lexical representation (§3.3.22.1). Specifically, leading zeroes are prohibited.

3.3.22.3 Constraining facets

`<unsignedInt>` has the following constraining facets:

- `totalDigits`
- `fractionDigits`
- `pattern`
- `whiteSpace`
- `enumeration`
- `maxInclusive`
- `maxExclusive`
- `minInclusive`
- `minExclusive`

3.3.22.4 Derived datatypes

The following built-in datatypes are derived from `<unsignedInt>`:

- `<unsignedShort>`

3.3.23 `<unsignedShort>`

[Definition:] `<unsignedShort>` is derived from `<unsignedInt>` by setting the value of `maxInclusive` to be 65535. The base type of `<unsignedShort>` is `<unsignedInt>`.

3.3.23.1 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 Lexical representation ($3.3.23.1$. Specifically, the leading zeroes are prohibited.

3.3.23.3 Constraining facets

unsignedShort has the following constraining facets:

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

3.3.23.4 Derived datatypes

The following built-in datatypes are derived from unsignedShort:

- unsignedByte

3.3.24 unsignedByte

[Definition:] unsignedByte is derived from unsignedShort by setting the value of maxInclusive to be 255. The base type of unsignedByte is unsignedShort.

3.3.24.1 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 Lexical representation ($3.3.24.1$. Specifically, leading zeroes are prohibited.

3.3.24.3 Constraining facets

unsignedByte has the following constraining facets:

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

3.3.25 positiveInteger

[Definition:] positiveInteger is derived from nonNegativeInteger by setting the value of minInclusive to be 1. This results in the standard mathematical concept of the positive integer numbers. The value space of positiveInteger is the infinite set {1,2,...). The base type of positiveInteger is nonNegativeInteger.

3.3.25.1 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 Lexical representation (§3.3.25.1). Specifically, the optional "+" sign is prohibited and leading zeroes are prohibited.

3.3.25.3 Constraining facets

positiveInteger has the following constraining facets:

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

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

4.1.1 The Simple Type Definition Schema Component

4.1.2 XML Representation of Simple Type Definition Schema Components

4.1.3 Constraints on XML Representation of Simple Type Definition

4.1.4 Simple Type Definition Validation Rules

4.1.5 Constraints on Simple Type Definition Schema Components

4.1.6 Simple Type Definition for anySimpleType

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 QName) 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:

<table>
<thead>
<tr>
<th>Schema Component: Simple Type Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{name}</code></td>
</tr>
<tr>
<td>Optional. An NCName as defined by [Namespaces in XML].</td>
</tr>
<tr>
<td><code>{target namespace}</code></td>
</tr>
<tr>
<td>Either absent or a namespace name, as defined in [Namespaces in XML].</td>
</tr>
<tr>
<td><code>{variety}</code></td>
</tr>
</tbody>
</table>
One of {atomic, list, union}. Depending on the value of {variety}, further properties are defined as follows:

- **atomic**

  ```
  {primitive type definition}
  ```

  A built-in primitive datatype definition (or the simple ur-type definition).

- **list**

  ```
  {item type definition}
  ```

  An atomic or union simple type definition.

- **union**

  ```
  {member type definitions}
  ```

  A non-empty sequence of simple type definitions.

- **{facets}**

  A possibly empty set of Facets (§2.4).

- **{fundamental facets}**

  A set of Fundamental facets (§2.4.1).

- **{base type definition}**

  If the datatype has been derived by restriction then the Simple Type Definition component from which it is derived, otherwise the Simple Type Definition for anySimpleType (§4.1.6).

- **{final}**

  A subset of {restriction, list, union}.

- **{annotation}**

  Optional. An annotation.

Datatypes are identified by their {name} and {target namespace}. Except for anonymous datatypes (those with no {name}), datatype definitions must be uniquely identified within a schema.

If {variety} is atomic then the value space of the datatype defined will be a subset of the value space of {base type definition} (which is a subset of the value space of {primitive type definition}). If {variety} is list then the value space of the datatype defined will be the set of finite-length sequence of values from the value space of {item type definition}. If {variety} is union then the value space of the datatype defined will be the union of the value space s of each datatype in {member type definitions}.

If {variety} is atomic then the {variety} of {base type definition} must be atomic. If {variety} is list then the {variety} of {item type definition} must be either atomic or union. If {variety} is union then {member type definitions} must be a list of datatype definitions.

The value of {facets} consists of the set of facet s specified directly in the datatype definition unioned with the possibly empty set of {facets} of {base type definition}.

The value of {fundamental facets} consists of the set of fundamental facet s and their values.

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

**4.1.2 XML Representation of Simple Type Definition Schema Components**

The XML representation for a Simple Type Definition schema component is a `<simpleType>` element information item. The correspondences between the properties of the information item and properties of the component are as follows:

```xml
<simpleType
 final = (#all | (list | union | restriction))
 id = ID
 name = NCName
 {any attributes with non-schema namespace . . .}>
Content: (annotation?, (restriction | list | union))
</simpleType>
```

**Datatype Definition Schema Component**
Property Representation
{name} The actual value of the name [attribute], if present, otherwise null
{final} A set corresponding to the actual value of the final [attribute], if present, otherwise of the actual value of the finalDefault [attribute] 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.

NOTE: Although the finalDefault [attribute] of schema may include values other than restriction, list or union, those values are ignored in the determination of {final}
{target namespace} The actual value of the targetNamespace [attribute] of the parent schema element information item.
{annotation} The annotation corresponding to the <annotation> 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

XML Representation Summary: restriction Element Information Item

```
<restriction
   base = QName
   id = ID
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive | totalDigits | fractionDigits | length | minLength | maxLength | enumeration | whiteSpace | pattern) *))
</restriction>
```

Simple Type Definition Schema Component

Property Representation
{variety} The actual value of {variety} of {base type definition}
{facets} The union of the set of Facets components resolved to by the facet [children] merged with {facets} from {base type definition}, subject to the Facet Restriction Valid constraints specified in Facets (§2.4).
{base type definition} The Simple Type Definition component resolved to by the actual value of the base [attribute] or the <simpleType> [children], whichever is present.

Example
An electronic commerce schema might define a datatype called Sku (the barcode number that appears on products) from the built-in datatype string by supplying a value for the pattern facet.

```
<simpleType name='Sku'>
   <restriction base='string'>
      <pattern value='\d{3}-[A-Z]{2}' />
   </restriction>
</simpleType>
```
In this case, **Sku** is the name of the new user-derived datatype, **string** is its base type and **pattern** is the facet.

### 4.1.2.2 Derivation by list

**XML Representation Summary: list Element Information Item**

```xml
<list
   id = ID
   itemType = QName
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (simpleType?))
</list>
```

**Simple Type Definition Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{variety}</td>
<td>list</td>
</tr>
<tr>
<td>{item type definition}</td>
<td>The Simple Type Definition component resolved to by the actual value of the itemType [attribute] or the &lt;simpleType&gt; [children], whichever is present.</td>
</tr>
</tbody>
</table>

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 white space separated lists of literals of the itemType.

**Example**

A system might want to store lists of floating point values.

```xml
<simpleType name='listOfFloat'>
   <list itemType='float'/>
</simpleType>
```

In this case, **listOfFloat** is the name of the new user-derived datatype, **float** is its itemType and **list** is the derivation method.

As mentioned in List datatypes (§2.5.1.2), 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

**XML Representation Summary: union Element Information Item**

```xml
<union
   id = ID
   {any attributes with non-schema namespace . . .}>
   Content: (annotation?, (simpleType?))
</union>
```
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.

```xml
<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>
```

As mentioned in Union datatypes (§2.5.1.3), 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

Schema Representation Constraint: Single Facet Value
Unless otherwise specifically allowed by this specification (Multiple patterns (§4.3.4.3) and Multiple enumerations (§4.3.5.3)) any given constraining facet can only be specified once within a single derivation step.

Schema Representation Constraint: itemType attribute or simpleType child

Either the itemType [attribute] or the <simpleType> [child] of the <list> element must be present, but not both.

Schema Representation Constraint: base attribute or simpleType child

Either the base [attribute] or the simpleType [child] of the <restriction> element must be present, but not both.

Schema Representation Constraint: memberTypes attribute or simpleType children

Either the memberTypes [attribute] of the <union> element must be non-empty or there must be at least one simpleType [child].

4.1.4 Simple Type Definition Validation Rules

Validation Rule: Facet Valid

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

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

Validation Rule: 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:
   1.1 if pattern is a member of {facets}, then the string must be pattern valid (§4.3.4.4);
   1.2 if pattern is not a member of {facets}, then
      1.2.1 if {variety} is atomic then the string must match a literal in the lexical space of {base type definition};
      1.2.2 if {variety} is list then the string must be a sequence of white space separated tokens, each of which matches a literal in the lexical space of {item type definition};
      1.2.3 if {variety} is union then the string must match a literal in the lexical space of at least one member of {member type definitions};
   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 (§4.1.4) with respect to each member of {facets} (except for pattern).

4.1.5 Constraints on Simple Type Definition Schema Components

Schema Component Constraint: applicable facets

The constraining facets which are allowed to be members of {facets} are dependent on {base type definition} as specified in the following table:

<table>
<thead>
<tr>
<th>{base type definition}</th>
<th>applicable {facets}</th>
</tr>
</thead>
<tbody>
<tr>
<td>If {variety} is list, then</td>
<td></td>
</tr>
<tr>
<td>[all datatypes]</td>
<td>length, minLength, maxLength, pattern, enumeration, whiteSpace</td>
</tr>
<tr>
<td>If {variety} is union, then</td>
<td></td>
</tr>
<tr>
<td>[all datatypes]</td>
<td>pattern, enumeration</td>
</tr>
<tr>
<td>else if {variety} is atomic, then</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>pattern, minLength, maxLength, pattern, enumeration, whiteSpace</td>
</tr>
<tr>
<td>boolean</td>
<td>pattern, whiteSpace</td>
</tr>
<tr>
<td>float</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>double</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>decimal</td>
<td>totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>duration</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>dateTime</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>time</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>date</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
<tr>
<td>gYear</td>
<td>pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive</td>
</tr>
</tbody>
</table>
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:

Schema Component: anySimpleType

{name}
  anySimpleType
{target namespace}
  http://www.w3.org/2001/XMLSchema
{basetype definition}
  the ur-type definition
{final}
  the empty set
{variety}
  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 \( a \neq b \)
- there is no pair a and b from the value space such that both \( a = b \) and \( a \neq b \)
- 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)

Note that a consequence of the above is that, given value space A and value space B where A and B are not related by restriction or union, for every pair of values a from A and b from B, \( a \neq b \).

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

4.2.2 ordered

[Definition:] 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.

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

[Definition:] 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 != b and neither a < b nor b < a

[Definition:] A total order is a 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

NOTE: 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

<table>
<thead>
<tr>
<th>Schema Component: ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
</tr>
<tr>
<td>One of {false, partial, total}.</td>
</tr>
</tbody>
</table>

{value} depends on {variety}, {facets} and {member type definitions} in the Simple Type Definition component in which a ordered component appears as a member of {fundamental facets}.

When {variety} is atomic, {value} is inherited from {value} of {base type definition}. For all primitive types {value} is as specified in the table in Fundamental Facets (§C.1).

When {variety} is list, {value} is false.

When {variety} is union, if {value} is true for every member of {member type definitions} and all members of {member type definitions} share a common ancestor, then {value} is true; else {value} is false.

4.2.3 bounded

[Definition:] 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 \( V \), \( l \leq v \).

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 and an exclusive lower bound.

**bounded** provides for:

- indicating whether a value space is bounded

### 4.2.3.1 The bounded Schema Component

<table>
<thead>
<tr>
<th>Schema Component: bounded</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{value}</code></td>
</tr>
<tr>
<td>A boolean</td>
</tr>
</tbody>
</table>

`{value}` depends on `{variety}`, `{facets}` and `{member type definitions}` in the **Simple Type Definition** component in which a bounded component appears as a member of `{fundamental facets}`.

When `{variety}` is `atomic`, if one of `minInclusive` or `minExclusive` and one of `maxInclusive` or `maxExclusive` are among `{facets}`, then `{value}` is `true`; else `{value}` is `false`.

When `{variety}` is `list`, if `length` or both of `minLength` and `maxLength` are among `{facets}`, then `{value}` is `true`; else `{value}` is `false`.

When `{variety}` is `union`, if `{value}` is `true` for every member of `{member type definitions}` and all members of `{member type definitions}` share a common ancestor, then `{value}` is `true`; else `{value}` is `false`.

### 4.2.4 cardinality

**Definition:** 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 space s that are finite
- value space s 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

<table>
<thead>
<tr>
<th>Schema Component: cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{value}</code></td>
</tr>
<tr>
<td>One of <code>{finite, countably infinite}</code></td>
</tr>
</tbody>
</table>
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

```latex
\textbf{Schema Component: numeric}

\{\text{value}\}

A boolean
```

4.3 Constraining Facets

4.3.1 length
4.3.2 minLength
4.3.3 maxLength
4.3.4 pattern
4.3.5 enumeration
4.3.6 whiteSpace
4.3.7 maxInclusive
4.3.8 `maxExclusive`
4.3.9 `minExclusive`
4.3.10 `minInclusive`
4.3.11 `totalDigits`
4.3.12 `fractionDigits`

4.3.1 `length`

[Definition:] `length` is the number of units of length, where units of length varies depending on the type that is being derived from. The value of `length` must be a nonNegativeInteger.

For `string` and datatypes derived from `string`, `length` is measured in units of `characters` as defined in [XML 1.0 (Second Edition)]. For `anyURI`, `length` is measured in units of characters (as for `string`). For `hexBinary` and `base64Binary` and datatypes derived from them, `length` is measured in octets (8 bits) of binary data. For datatypes derived by `list`, `length` is measured in number of list items.

**NOTE:** For `string` and datatypes derived from `string`, `length` will not always coincide with "string length" as perceived by some users or with the number of storage units in some digital representation. Therefore, care should be taken when specifying a value for `length` and in attempting to infer storage requirements from a given value for `length`.

`length` provides for:

- Constraining a value space to values with a specific number of units of length, where units of length varies depending on {base type definition}.

**Example**
The following is the definition of a user-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.

```xml
<simpleType name='productCode'>
  <restriction base='string'>
    <length value='8' fixed='true'/>
  </restriction>
</simpleType>
```

4.3.1.1 The `length` Schema Component

**Schema Component:** `length`

- `{value}`
  - A `nonNegativeInteger`.
- `{fixed}`
  - A `boolean`.
- `{annotation}`
  - Optional. An `annotation`.

If `{fixed}` is true, then types for which the current type is the `{base type definition}` cannot specify a value for `length` other than `{value}`.

4.3.1.2 XML Representation of `length` Schema Components

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

**XML Representation Summary:** `length` Element Information Item
### 4.3.1.3 length Validation Rules

**Validation Rule: Length Valid**

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

1. If the (variety) is atomic then
   1.1 If the (primitive type definition) is string, then the length of the value, as measured in characters, must be equal to (value);
   1.2 If the (primitive type definition) is hexBinary or base64Binary, then the length of the value, as measured in octets of the binary data, must be equal to (value);

2. If the (variety) is list, then the length of the value, as measured in list items, must be equal to (value).

### 4.3.1.4 Constraints on length Schema Components

**Schema Component Constraint: length and minLength or maxLength**

It is an error for both length and either of minLength or maxLength to be members of {facets}.

**Schema Component Constraint: length valid restriction**

It is an error if length is among the members of {facets} of {base type definition} and (value) is not equal to the (value) of the parent length.

### 4.3.2 minLength

[Definition:] `minLength` is the minimum number of units of length, where units of length varies depending on the type that is being derived from. The value of `minLength` must be a nonNegativeInteger.

For string and datatypes derived from string, minLength is measured in units of characters as defined in [XML 1.0 (Second Edition)]. For hexBinary and base64Binary 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.

**NOTE:** For string and datatypes derived from string, minLength will not always coincide with "string length" as perceived by some users or with the number of storage units in some digital representation. Therefore, care should be taken when specifying a value for minLength and in attempting to infer storage requirements from a given value for minLength.

minLength provides for:

- Constraining a value space to values with at least a specific number of units of length, where units of length varies depending on (base type definition).

**Example**

The following is the definition of a user-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).
4.3.2.1 The minLength Schema Component

<table>
<thead>
<tr>
<th>Schema Component: minLength</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
</tr>
<tr>
<td>{fixed}</td>
</tr>
<tr>
<td>{annotation}</td>
</tr>
</tbody>
</table>

If (fixed) is true, then types for which the current type is the (base type definition) cannot specify a value for minLength other than (value).

4.3.2.2 XML Representation of minLength Schema Component

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

<table>
<thead>
<tr>
<th>XML Representation Summary: minLength Element Information Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;minLength fixed = boolean : false id = ID value = nonNegativeInteger {any attributes with non-schema namespace . . .}&gt;</td>
</tr>
<tr>
<td>Content: (annotation?)</td>
</tr>
<tr>
<td>&lt;/minLength&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minLength Schema Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
</tr>
<tr>
<td>{value}</td>
</tr>
<tr>
<td>{fixed}</td>
</tr>
<tr>
<td>{annotation}</td>
</tr>
</tbody>
</table>

4.3.2.3 minLength Validation Rules

**Validation Rule: minLength Valid**

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

1. if the (variety) is atomic then
   1.1 if (primitive type definition) is string, then the length of the value, as measured in characters, must be greater than or equal to (value);
   1.2 if (primitive type definition) is hexBinary or base64Binary, then the length of the value, as measured in octets of the binary data, must be greater than or equal to (value);

2. if the (variety) is list, then the length of the value, as measured in list items, must be greater than or equal to (value)

4.3.2.4 Constraints on minLength Schema Components

**Schema Component Constraint: minLength <= maxLength**
If both `minLength` and `maxLength` are members of {facets}, then the {value} of `minLength` must be less than or equal to the {value} of `maxLength`.

**Schema Component Constraint: minLength valid restriction**

It is an error if `minLength` is among the members of {facets} of {base type definition} and {value} is less than the {value} of the parent `minLength`.

### 4.3.3 maxLength

[Definition:] **maxLength** is the maximum number of units of length, where units of length varies depending on the type that is being derived from. The value of **maxLength** must be a **nonNegativeInteger**.

For `string` and datatypes derived from `string`, `maxLength` is measured in units of characters as defined in [XML 1.0 (Second Edition)]. For `hexBinary` and `base64Binary` 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.

**NOTE:** For `string` and datatypes derived from `string`, `maxLength` will not always coincide with "string length" as perceived by some users or with the number of storage units in some digital representation. Therefore, care should be taken when specifying a value for `maxLength` and in attempting to infer storage requirements from a given value for `maxLength`.

**maxLength** provides for:

- Constraining a value space to values with at most a specific number of units of length, where units of length varies depending on {base type definition}.

**Example**
The following is the definition of a user-derived datatype which might be used to accept form input with an upper limit to the number of characters that are acceptable.

```xml
<simpleType name='form-input'>
  <restriction base='string'>
    <maxLength value='50'/>
  </restriction>
</simpleType>
```

### 4.3.3.1 The maxLength Schema Component

**Schema Component:** **maxLength**

- `{value}`
  - A `nonNegativeInteger`.
- `{fixed}`
  - A `boolean`.
- `{annotation}`
  - Optional. An `annotation`.

If `{fixed}` is true, then types for which the current type is the {base type definition} cannot specify a value for `maxLength` other than `{value}`.

### 4.3.3.2 XML Representation of maxLength Schema Components

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

**XML Representation Summary:** **maxLength** Element Information Item
4.3.3.3 maxLength Validation Rules

Validation Rule: maxLength Valid

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

1. if the {variety} is atomic then
   1.1 if {primitive type definition} is string, then the length of the value, as measured in characters, must be less than or equal to {value};
   1.2 if {primitive type definition} is hexBinary or base64Binary, then the length of the value, as measured in octets of the binary data, must be less than or equal to {value};
2. if the {variety} is list, then the length of the value, as measured in list items, must be less than or equal to {value}.

4.3.3.4 Constraints on maxLength Schema Components

Schema Component Constraint: maxLength valid restriction

It is an error if maxLength is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent maxLength.

4.3.4 pattern

[Definition:] pattern is a constraint on the value space of a datatype which is achieved by constraining the lexical space to literals which match 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.

Example

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

Schema Component: pattern

{value}
4.3.4.2 XML Representation of pattern Schema Components

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value [attribute]</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

4.3.4.3 Constraints on XML Representation of pattern

Schema Representation Constraint: Multiple patterns

If multiple <pattern> element information items appear as [children] of a <simpleType>, the [value]s should be combined as if they appeared in a single regular expression as separate branches.

NOTE: It is a consequence of the schema representation constraint Multiple patterns (§ 4.3.4.3) 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 pattern s they wish to impose, or define each pattern on a separate type derivation step.

4.3.4.4 pattern Validation Rules

Validation Rule: 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 {value}.

4.3.5 enumeration

[Definition:] enumeration constrains the value space to a specified set of values.

enumeration does not impose an order relation on the value space it creates; the value of the ordered property of the derived datatype remains that of the datatype from which it is derived.

enumeration provides for:

- Constraining a value space to a specified set of values.
Example
The following example is a 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.

```xml
<simpleType name='holidays'>
  <annotation>
    <documentation>some US holidays</documentation>
  </annotation>
  <restriction base='gMonthDay'>
    <enumeration value='--01-01'>
      <annotation>
        <documentation>New Year's day</documentation>
      </annotation>
    </enumeration>
    <enumeration value='--07-04'>
      <annotation>
        <documentation>4th of July</documentation>
      </annotation>
    </enumeration>
    <enumeration value='--12-25'>
      <annotation>
        <documentation>Christmas</documentation>
      </annotation>
    </enumeration>
  </restriction>
</simpleType>
```

4.3.5.1 The enumeration Schema Component

**Schema Component: enumeration**

{**value**}
A set of values from the value space of the (base type definition).

{**annotation**}
Optional. An annotation.

4.3.5.2 XML Representation of enumeration Schema Components

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

**XML Representation Summary: enumeration Element Information Item**

```
<enumeration
  id='ID'
  value='anySimpleType'>
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</enumeration>
```

(value) must be in the value space of (base type definition).
4.3.5.3 Constraints on XML Representation of enumeration

Schema Representation Constraint: Multiple enumerations
If multiple <enumeration> element information items appear as [children] of a <simpleType> the {value} of the enumeration component should be the set of all such {value}s.

4.3.5.4 enumeration Validation Rules

Validation Rule: enumeration valid
A value in a \{value\} space is facet-valid with respect to enumeration if the value is one of the values specified in {value}.

4.3.5.5 Constraints on enumeration Schema Components

Schema Component Constraint: enumeration valid restriction
It is an error if any member of \{value\} is not in the \{value\} space of {base type definition}.

4.3.6 whiteSpace

[Definition:] whiteSpace constrains the \{value\} space of types derived from string such that the various behaviors specified in Attribute Value Normalization in [XML 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.

NOTE: The notation #xA used here (and elsewhere in this specification) represents the Universal Character Set (UCS) code point hexadecimal A (line feed), which is denoted by U+000A. This notation is to be distinguished from \&xAmp; , which is the XML character reference to that same UCS code point.

whiteSpace is applicable to all atomic and list datatypes. For all atomic datatypes other than string (and types derived by restriction from it) the value of whiteSpace is collapse and cannot be changed by a schema author; for string the value of whiteSpace is preserve; for any type derived by restriction from string 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.

NOTE: 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.

Example
The following example is the datatype definition for the token built-in derived datatype.
4.3.6.1 The `whiteSpace` Schema Component

**Schema Component: whiteSpace**

```xml
<whiteSpace
    fixed = boolean : false
    id = ID
    value = (collapse | preserve | replace)
    {any attributes with non-schema namespace . . .}>
    Content: {annotation}?
</whiteSpace>
```

If `fixed` is true, then types for which the current type is the (`base type definition`) cannot specify a value for `whiteSpace` other than `value`.

4.3.6.2 XML Representation of `whiteSpace` Schema Components

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

**XML Representation Summary:** `whiteSpace` Element Information Item

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{value}</code></td>
<td>The actual value of the <code>value</code> attribute</td>
</tr>
<tr>
<td><code>{fixed}</code></td>
<td>The actual value of the <code>fixed</code> attribute, if present, otherwise false</td>
</tr>
<tr>
<td><code>{annotation}</code></td>
<td>The annotations corresponding to all the <code>&lt;annotation&gt;</code> element information items in the <code>[children]</code>, if any.</td>
</tr>
</tbody>
</table>

4.3.6.3 `whiteSpace` Validation Rules

**NOTE:** There are no Validation Rules associated with `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

**Schema Component Constraint: whiteSpace valid restriction**

It is an error if `whiteSpace` is among the members of (facets) of (`base type definition`) and any of the following conditions is true:

1. `value` is `replace` or `preserve` and the `value` of the parent `whiteSpace` is `collapse`
2. `value` is `preserve` and the `value` of the parent `whiteSpace` is `replace`

4.3.7 `maxInclusive`
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.

### Example

The following is the definition of a user-derived datatype which limits values to integers less than or equal to 100, using maxInclusive.

```xml
<simpleType name='one-hundred-or-less'>
  <restriction base='integer'>
    <maxInclusive value='100'/>
  </restriction>
</simpleType>
```

### 4.3.7.1 The maxInclusive Schema Component

**Schema Component: maxInclusive**

- `{value}` A value from the value space of the {base type definition}.
- `{fixed}` A boolean.
- `{annotation}` Optional. An annotation.

If `{fixed}` is true, then types for which the current type is the {base type definition} cannot specify a value for maxInclusive other than `{value}`.

### 4.3.7.2 XML Representation of maxInclusive Schema Components

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

- `<maxInclusive>` Element Information Item

```
<maxInclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</maxInclusive>
```

{value} must be in the value space of {base type definition}.

### maxInclusive Schema Component

- **Property** `{value}` Representation
  - The actual value of the value [attribute]
- **Property** `{fixed}` Representation
  - The actual value of the fixed [attribute], if present, otherwise false, if present, otherwise false
- **Property** `{annotation}` Representation
  - The annotations corresponding to all the <annotation> element information items in the [children], if any.
4.3.7.3 maxInclusive Validation Rules

Validation Rule: maxInclusive Valid
A value in an ordered value space is facet-valid with respect to maxInclusive, determined as follows:
1 if the numeric property in {fundamental facets} is true, then the value must be numerically less than or equal to {value};
2 if the numeric property in {fundamental facets} is false (i.e., {base type definition} is one of the date and time related datatypes), then the value must be chronologically less than or equal to {value};

4.3.7.4 Constraints on maxInclusive Schema Components

Schema Component Constraint: minInclusive <= maxInclusive
It is an error for the value specified for minInclusive to be greater than the value specified for maxInclusive for the same datatype.

Schema Component Constraint: maxInclusive valid restriction
It is an error if any of the following conditions is true:
1 maxInclusive is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent maxInclusive
2 maxExclusive is among the members of {facets} of {base type definition} and {value} is greater than or equal to the {value} of the parent maxExclusive
3 minInclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of the parent minInclusive
4 minExclusive is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of the parent minExclusive

4.3.8 maxExclusive

[Definition:] maxExclusive is the exclusive upper bound of the value space for a datatype with the ordered property. The value of maxExclusive must be in the value space of the base type.

maxExclusive provides for:

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

Example
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

<table>
<thead>
<tr>
<th>Schema Component: maxExclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{value}</code></td>
</tr>
<tr>
<td>A value from the value space of the {base type definition}.</td>
</tr>
<tr>
<td><code>{fixed}</code></td>
</tr>
<tr>
<td>A boolean.</td>
</tr>
<tr>
<td><code>{annotation}</code></td>
</tr>
<tr>
<td>Optional. An annotation.</td>
</tr>
</tbody>
</table>

If {fixed} is true, then types for which the current type is the {base type definition} cannot specify a value for maxExclusive other than {value}.
The XML representation for a `maxExclusive` schema component is a `<maxExclusive>` element information item. The correspondences between the properties of the information item and properties of the component are as follows:

**XML Representation Summary: maxExclusive Element Information Item**

```
<maxExclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
  {any attributes with non-schema namespace ...}>
  Content: (annotation?)
</maxExclusive>
```

{value} must be in the value space of (base type definition).

### maxExclusive Schema Component

**Property**

- **{value}** The actual value of the `value` attribute
- **{fixed}** The actual value of the `fixed` attribute, if present, otherwise false
- **{annotation}** The annotations corresponding to all the `<annotation>` element information items in the [children], if any.

#### 4.3.8.3 maxExclusive Validation Rules

**Validation Rule: maxExclusive Valid**

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

1. If the numeric property in {fundamental facets} is true, then the value must be numerically less than {value};
2. If the numeric property in {fundamental facets} is false (i.e., {base type definition} is one of the date and time related datatypes), then the value must be chronologically less than {value};

#### 4.3.8.4 Constraints on maxExclusive Schema Components

**Schema Component Constraint: maxInclusive and maxExclusive**

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

**Schema Component Constraint: minExclusive <= maxExclusive**

It is an error for the value specified for `minExclusive` to be greater than the value specified for `maxExclusive` for the same datatype.

**Schema Component Constraint: maxExclusive valid restriction**

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

1. `maxExclusive` is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent `maxExclusive`
2. `maxInclusive` is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent `maxInclusive`
3. `minInclusive` is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of the parent `minInclusive`
4. `minExclusive` is among the members of {facets} of {base type definition} and {value} is less than or equal to the {value} of the parent `minExclusive`

#### 4.3.9 minExclusive

**[Definition:]** `minExclusive` is the exclusive lower bound of the value space for a datatype with the ordered property. The value of `minExclusive` must be in the value space of the base type.

`minExclusive` provides for:
Constraining a value space to values with a specific exclusive lower bound.

Example
The following is the definition of a user-derived datatype which limits values to integers greater than or equal to 100, using minExclusive.

```xml
<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

```
{value}
A value from the value space of the {base type definition}.

{fixed}
A boolean.

{annotation}
Optional. An annotation.
```

If {fixed} is true, then types for which the current type is the {base type definition} cannot specify a value for minExclusive other than {value}.

4.3.9.2 XML Representation of minExclusive Schema Components

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

```
<minExclusive
  fixed = boolean : false
  id = ID
  value = anySimpleType
{any attributes with non-schema namespace . . . }>
  {annotation}
</minExclusive>
```

{value} must be in the value space of {base type definition}.

4.3.9.3 minExclusive Validation Rules

Validation Rule: minExclusive Valid
A value in an ordered value space is facet-valid with respect to minExclusive if:
1 if the numeric property in {fundamental facets} is true, then the value must be numerically greater than {value};
2 if the numeric property in {fundamental facets} is false (i.e., {base type definition} is one of the date and time related datatypes), then the value must be chronologically greater than {value};

4.3.9.4 Constraints on minExclusive Schema Components

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

Schema Component Constraint: minExclusive < maxInclusive
It is an error for the value specified for minExclusive to be greater than or equal to the value specified for maxInclusive for the same datatype.

Schema Component Constraint: minExclusive valid restriction
It is an error if any of the following conditions is true:
1 minExclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of the parent minExclusive
2 maxInclusive is among the members of {facets} of {base type definition} and {value} is greater the {value} of the parent maxInclusive
3 minInclusive is among the members of {facets} of {base type definition} and {value} is less than the {value} of the parent minInclusive
4 maxExclusive is among the members of {facets} of {base type definition} and {value} is greater than or equal to the {value} of the parent maxExclusive

4.3.10 minInclusive

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

minInclusive provides for:

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

Example
The following is the definition of a user-derived datatype which limits values to integers greater than or equal to 100, using minInclusive.

```xml
<simpleType name='one-hundred-or-more'>
  <restriction base='integer'>
    <minInclusive value='100'/>
  </restriction>
</simpleType>
```

4.3.10.1 The minInclusive Schema Component

<table>
<thead>
<tr>
<th>Schema Component: minInclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
</tr>
<tr>
<td>A value from the value space of the {base type definition}.</td>
</tr>
<tr>
<td>{fixed}</td>
</tr>
<tr>
<td>A boolean.</td>
</tr>
<tr>
<td>{annotation}</td>
</tr>
<tr>
<td>Optional. An annotation.</td>
</tr>
</tbody>
</table>

If {fixed} is true, then types for which the current type is the {base type definition} cannot specify a value for minInclusive other than {value}.

4.3.10.2 XML Representation of minInclusive Schema Components
The XML representation for a minInclusive schema component is a `<minInclusive>` element information item. The correspondences between the properties of the information item and properties of the component are as follows:

```
XMLElement Information Item
<minInclusive
    fixed = boolean
    id = ID
    value = anySimpleType
    {any attributes with non-schema namespace . . .}>
    {value}
    {minInclusive} Schema Component
    Content: (annotation?)
</minInclusive>
```

{value} must be in the value space of {base type definition}.

**minInclusive Schema Component**

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>{value}</td>
<td>The actual value of the value [attribute]</td>
</tr>
<tr>
<td>{fixed}</td>
<td>The actual value of the fixed [attribute], if present, otherwise false</td>
</tr>
<tr>
<td>{annotation}</td>
<td>The annotations corresponding to all the &lt;annotation&gt; element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

### 4.3.10.3 minInclusive Validation Rules

**Validation Rule: minInclusive Valid**

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

1. If the numeric property in (fundamental facets) is true, then the value must be numerically greater than or equal to (value);
2. If the numeric property in (fundamental facets) is false (i.e., {base type definition} is one of the date and time related datatypes), then the value must be chronologically greater than or equal to (value);

### 4.3.10.4 Constraints on minInclusive Schema Components

**Schema Component Constraint: minInclusive < maxExclusive**

It is an error for the value specified for minInclusive to be greater than or equal to the value specified for maxExclusive for the same datatype.

**Schema Component Constraint: minInclusive valid restriction**

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

1. minInclusive is among the members of {facets} of {base type definition} and (value) is less than the (value) of the parent minInclusive.
2. maxInclusive is among the members of {facets} of {base type definition} and (value) is greater than the (value) of the parent maxInclusive.
3. minExclusive is among the members of {facets} of {base type definition} and (value) is less than or equal to the (value) of the parent minExclusive.
4. maxExclusive is among the members of {facets} of {base type definition} and (value) is greater than or equal to the (value) of the parent maxExclusive.

### 4.3.11 totalDigits

**Definition:** totalDigits is the maximum number of digits in values of datatypes derived from decimal. The value of totalDigits must be a positiveInteger.

totalDigits provides for:

- Constraining a value space to values with a specific maximum number of decimal digits (#x30-#x39).

**Example**
The following is the definition of a user-derived datatype which could be used to represent monetary amounts, such as in a financial management application which does not have figures of $1M or more and only allows whole cents. 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 in a manner specific to the base type (different than specifying max/min values as before).

```xml
<simpleType name='amount'>
  <restriction base='decimal'>
    <totalDigits value='8'/>
    <fractionDigits value='2' fixed='true'/>
  </restriction>
</simpleType>
```

4.3.11.1 The totalDigits Schema Component

<table>
<thead>
<tr>
<th>Schema Component: totalDigits</th>
</tr>
</thead>
</table>

- `{value}` A positiveInteger.
- `{fixed}` A boolean.
- `{annotation}` Optional. An annotation.

If `{fixed}` is true, then types for which the current type is the {base type definition} cannot specify a value for totalDigits other than `{value}`.

4.3.11.2 XML Representation of totalDigits Schema Components

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

XML Representation Summary: totalDigits Element Information Item

- `fixed` = boolean : false
- `id` = ID
- `value` = positiveInteger

{any attributes with non-schema namespace . . .}>
Content: `{annotation}`
</totalDigits>

totalDigits Schema Component

<table>
<thead>
<tr>
<th>Property</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{value}</code></td>
<td>The actual value of the <code>value</code> [attribute]</td>
</tr>
<tr>
<td><code>{fixed}</code></td>
<td>The actual value of the <code>fixed</code> [attribute], if present, otherwise false</td>
</tr>
<tr>
<td><code>{annotation}</code></td>
<td>The annotations corresponding to all the <code>&lt;annotation&gt;</code> element information items in the [children], if any.</td>
</tr>
</tbody>
</table>

4.3.11.3 totalDigits Validation Rules

Validation Rule: totalDigits Valid
A value in a value space is facet-valid with respect to totalDigits if:
1 the number of decimal digits in the value is less than or equal to `{value}`;

4.3.11.4 Constraints on totalDigits Schema Components
Schema Component Constraint: totalDigits valid restriction

It is an error if totalDigits is among the members of {facets} of {base type definition} and {value} is greater than the {value} of the parent totalDigits.

4.3.12 fractionDigits

[Definition:] fractionDigits is the maximum number of digits in the fractional part of values of datatypes derived from decimal. The value of fractionDigits must be a nonNegativeInteger.

fractionDigits provides for:

- Constraining a value space to values with a specific maximum number of decimal digits in the fractional part.

Example

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.

```xml
<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

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>A nonNegativeInteger.</td>
</tr>
<tr>
<td>fixed</td>
<td>A boolean.</td>
</tr>
<tr>
<td>annotation</td>
<td>Optional. An annotation.</td>
</tr>
</tbody>
</table>

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

4.3.12.2 XML Representation of fractionDigits Schema Components

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

```xml
<fractionDigits
  fixed = boolean : false
  id = ID
  value = nonNegativeInteger
  {any attributes with non-schema namespace . . .}> 
  Content: (annotation?)
</fractionDigits>
```

fractionDigits Schema Component
4.3.12.3 fractionDigits Validation Rules

Validation Rule: fractionDigits Valid

A value in a value space is facet-valid with respect to fractionDigits if:
1. the number of decimal digits in the fractional part of the value is less than or equal to \{value\};

4.3.12.4 Constraints on fractionDigits Schema Components

Schema Component Constraint: fractionDigits less than or equal to totalDigits

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

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.

[Definition:] Minimally conforming processors must completely and correctly implement the Constraint on Schemas and Validation Rule.

[Definition:] Processors which accept schemas in the form of XML documents as described in XML Representation of Simple Type Definition Schema Components (§4.1.2) (and other relevant portions of Datatype components (§4)) 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 XML Representation of Simple Type Definition Schema Components (§4.1.2) (and other relevant portions of Datatype components (§4)) for mapping the contents of such documents to schema components for use in validation.

NOTE: 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.

A Schema for Datatype Definitions (normative)

```xml
<xs:element name="hasFacet" type="xs:Facet"/>
<xs:element name="hasProperty" type="xs:Property"/>
```
Make sure that processors that do not read the external subset will know about the various IDs we declare.

First the built-in primitive datatypes. These definitions are for information only, the real built-in definitions are magic. Note in particular that there is no type named 'anySimpleType'. The primitives should really be derived from no type at all, and anySimpleType should be derived as a union of all the primitives.

For each built-in datatype in this schema (both primitive and derived) can be uniquely addressed via a URI constructed as follows:

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

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

http://www.w3.org/2001/XMLSchema#nt

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#nt.maxInclusive
<xs:simpleType name="float" id="float">
<xs:annotation>
<xs:appinfo>
<hfp:hasFacet name="enumeration"/>
<hfp:hasFacet name="whiteSpace"/>
<hfp:hasFacet name="maxInclusive"/>
<hfp:hasFacet name="maxExclusive"/>
<hfp:hasFacet name="minInclusive"/>
<hfp:hasFacet name="minExclusive"/>
<hfp:hasProperty name="ordered" value="total"/>
<hfp:hasProperty name="bounded" value="true"/>
<hfp:hasProperty name="cardinality" value="finite"/>
<hfp:hasProperty name="numeric" value="true"/>
</xs:appinfo>
<xs:documentation source="http://www.w3.org/TR/xmlschema-2/#float"/>
</xs:annotation>
<xs:restriction base="xs:anySimpleType">
<xs:whiteSpace value="collapse" fixed="true" id="float.whiteSpace"/>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="double" id="double">
<xs:annotation>
<xs:appinfo>
<hfp:hasFacet name="pattern"/>
<hfp:hasFacet name="enumeration"/>
<hfp:hasFacet name="whiteSpace"/>
<hfp:hasFacet name="maxInclusive"/>
<hfp:hasFacet name="maxExclusive"/>
<hfp:hasFacet name="minInclusive"/>
<hfp:hasFacet name="minExclusive"/>
<hfp:hasProperty name="ordered" value="total"/>
<hfp:hasProperty name="bounded" value="true"/>
<hfp:hasProperty name="cardinality" value="finite"/>
<hfp:hasProperty name="numeric" value="true"/>
</xs:appinfo>
<xs:documentation source="http://www.w3.org/TR/xmlschema-2/#double"/>
</xs:annotation>
<xs:restriction base="xs:anySimpleType">
<xs:whiteSpace value="collapse" fixed="true" id="double.whiteSpace"/>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="decimal" id="decimal">
<xs:annotation>
<xs:appinfo>
<hfp:hasFacet name="totalDigits"/>
<hfp:hasFacet name="fractionDigits"/>
<hfp:hasFacet name="pattern"/>
<hfp:hasFacet name="whiteSpace"/>
<hfp:hasFacet name="enumeration"/>
<hfp:hasFacet name="maxInclusive"/>
<hfp:hasFacet name="maxExclusive"/>
<hfp:hasFacet name="minInclusive"/>
<hfp:hasFacet name="minExclusive"/>
<hfp:hasProperty name="ordered" value="total"/>
<hfp:hasProperty name="bounded" value="false"/>
<hfp:hasProperty name="cardinality" value="countably infinite"/>
<hfp:hasProperty name="numeric" value="true"/>
</xs:appinfo>
<xs:documentation source="http://www.w3.org/TR/xmlschema-2/#decimal"/>
</xs:annotation>
<xs:restriction base="xs:anySimpleType">
</xs:restriction>
</xs:simpleType>
<xs:restriction base="xs:anySimpleType">
  <xs:whiteSpace value="collapse" fixed="true"
    id="decimal.whiteSpace"/>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="duration" id="duration">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="partial"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#duration"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true"
      id="duration.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="dateTime" id="dateTime">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="partial"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#dateTime"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true"
      id="dateTime.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="time" id="time">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="false"/>
      <hfp:hasProperty name="numeric" value="true"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#time"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true"
      id="time.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="gYear" id="gYear">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="partial"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#gYear"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true" id="gYear.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="gMonthDay" id="gMonthDay">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="maxInclusive"/>
      <hfp:hasFacet name="maxExclusive"/>
      <hfp:hasFacet name="minInclusive"/>
      <hfp:hasFacet name="minExclusive"/>
      <hfp:hasProperty name="ordered" value="partial"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#gMonthDay"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true" id="gMonthDay.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="gMonth" id="gMonth">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="pattern" />
      <hfp:hasFacet name="enumeration" />
      <hfp:hasFacet name="WhiteSpace" />
      <hfp:hasFacet name="maxInclusive" />
      <hfp:hasFacet name="maxExclusive" />
      <hfp:hasFacet name="minInclusive" />
      <hfp:hasFacet name="minExclusive" />
      <hfp:hasProperty name="ordered" value="false" />
      <hfp:hasProperty name="bounded" value="false" />
      <hfp:hasProperty name="cardinality" value="countably infinite" />
      <hfp:hasProperty name="numeric" value="false" />
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#gMonth" />
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true" id="gMonth.whiteSpace" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="hexBinary" id="hexBinary">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="length" />
      <hfp:hasFacet name="minLength" />
      <hfp:hasFacet name="maxLength" />
      <hfp:hasFacet name="pattern" />
      <hfp:hasFacet name="enumeration" />
      <hfp:hasFacet name="WhiteSpace" />
      <hfp:hasProperty name="ordered" value="false" />
      <hfp:hasProperty name="bounded" value="false" />
      <hfp:hasProperty name="cardinality" value="countably infinite" />
      <hfp:hasProperty name="numeric" value="false" />
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#binary" />
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true" id="hexBinary.whiteSpace" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="base64Binary" id="base64Binary">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="length" />
      <hfp:hasFacet name="minLength" />
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#binary" />
  </xs:annotation>
</xs:simpleType>
<xs:simpleType name="base64Binary">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="maxLength"/>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasProperty name="ordered" value="false"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#base64Binary"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true" id="base64Binary.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="anyURI" id="anyURI">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="length"/>
      <hfp:hasFacet name="minLength"/>
      <hfp:hasFacet name="maxLength"/>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasProperty name="ordered" value="false"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#anyURI"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true" id="anyURI.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="QName" id="QName">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="length"/>
      <hfp:hasFacet name="minLength"/>
      <hfp:hasFacet name="maxLength"/>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasProperty name="ordered" value="false"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#QName"/>
  </xs:annotation>
  <xs:restriction base="xs:anySimpleType">
    <xs:whiteSpace value="collapse" fixed="true" id="QName.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>
NOTATION cannot be used directly in a schema; rather a type must be derived from it by specifying at least one enumeration facet whose value is the name of a NOTATION declared in the schema.

Now the derived primitive types

The derived primitive types

The derived primitive types

The derived primitive types

The derived primitive types

The derived primitive types
<xs:annotation>
  <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#NMTOKEN"/>
</xs:annotation>
<xs:restriction base="xs:token">
  <xs:pattern value="\c+" id="NMTOKEN.pattern">
    <xs:annotation>
      <xs:documentation source="http://www.w3.org/TR/REC-xml#NT-Nmtoken">
        pattern matches production 7 from the XML spec
      </xs:annotation>
    </xs:pattern>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NMTOKENS" id="NMTOKENS">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="length"/>
      <hfp:hasFacet name="minLength"/>
      <hfp:hasFacet name="maxLength"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasProperty name="ordered" value="false"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality" value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
  </xs:annotation>
  <xs:restriction>
    <xs:simpleType>
      <xs:list itemType="xs:NMTOKEN"/>
    </xs:simpleType>
    <xs:minLength value="1" id="NMTOKENS.minLength"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="Name" id="Name">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#Name"/>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value="\i\c*" id="Name.pattern">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/REC-xml#NT-Name">
          pattern matches production 5 from the XML spec
        </xs:documentation>
      </xs:pattern>
    </xs:restriction>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="NCName" id="NCName">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#NCName"/>
  </xs:annotation>
  <xs:restriction base="xs:Name">
    <xs:pattern value="[^-].*[^-]" id="NCName.pattern">
      <xs:annotation>
        <xs:documentation source="http://www.w3.org/TR/REC-xml#NT-NCName">
          pattern matches production 7 from the XML spec
        </xs:documentation>
      </xs:pattern>
    </xs:restriction>
  </xs:restriction>
</xs:simpleType>
<xs:annotation>
  <xs:documentation
    source="http://www.w3.org/TR/REC-xml-names/#NT-NCName">pattern matches production 4 from the Namespaces in XML spec</xs:documentation>
</xs:annotation>
</xs:pattern>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="ID" id="ID">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#ID"/>
  </xs:annotation>
  <xs:restriction base="xs:NCName"/>
</xs:simpleType>

<xs:simpleType name="IDREF" id="IDREF">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#IDREF"/>
  </xs:annotation>
  <xs:restriction base="xs:NCName"/>
</xs:simpleType>

<xs:simpleType name="ENTITY" id="ENTITY">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#ENTITY"/>
  </xs:annotation>
  <xs:restriction base="xs:NCName"/>
</xs:simpleType>

<xs:simpleType name="integer" id="integer">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#integer"/>
  </xs:annotation>
  <xs:restriction base="xs:decimal">
    <xs:fractionDigits value="0" fixed="true" id="integer.fractionDigits"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="nonPositiveInteger" id="nonPositiveInteger">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#nonPositiveInteger"/>
  </xs:annotation>
  <xs:restriction base="xs:integer">
    <xs:maxInclusive value="0" id="nonPositiveInteger.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="negativeInteger" id="negativeInteger">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#negativeInteger"/>
  </xs:annotation>
  <xs:restriction base="xs:nonPositiveInteger">
    <xs:maxInclusive value="-1" id="negativeInteger.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="long" id="long"/>
<xs:simpleType name="unsignedInt" id="unsignedInt">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#unsignedInt"/>
  </xs:annotation>
  <xs:restriction base="xs:unsignedLong">
    <xs:maxInclusive value="4294967295" id="unsignedInt.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="unsignedShort" id="unsignedShort">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#unsignedShort"/>
  </xs:annotation>
  <xs:restriction base="xs:unsignedInt">
    <xs:maxInclusive value="65535" id="unsignedShort.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="unsignedByte" id="unsignedByte">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#unsignedByte"/>
  </xs:annotation>
  <xs:restriction base="xs:unsignedShort">
    <xs:maxInclusive value="255" id="unsignedByte.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="positiveInteger" id="positiveInteger">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#positiveInteger"/>
  </xs:annotation>
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:minInclusive value="1" id="positiveInteger.minInclusive"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="derivationControl">
  <xs:annotation>
    <xs:documentation>A utility type, not for public use</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="substitution"/>
    <xs:enumeration value="extension"/>
    <xs:enumeration value="restriction"/>
    <xs:enumeration value="list"/>
    <xs:enumeration value="union"/>
  </xs:restriction>
</xs:simpleType>

<xs:group name="simpleDerivation">
  <xs:choice>
    <xs:element name="derivationControl"/>
  </xs:choice>
</xs:group>
<xs:element ref="xs:restriction"/>
<xs:element ref="xs:list"/>
<xs:element ref="xs:union"/>
</xs:choice>
</xs:group>

<xs:simpleType name="simpleDerivationSet">
<xs:annotation>
<xs:documentation>
#all or (possibly empty) subset of {restriction, union, list}
</xs:documentation>
<xs:documentation>
A utility type, not for public use</xs:documentation>
</xs:annotation>
<xs:union>
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="#all"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType>
<xs:restriction base="xs:derivationControl">
<xs:enumeration value="list"/>
<xs:enumeration value="union"/>
<xs:enumeration value="restriction"/>
</xs:restriction>
</xs:simpleType>
</xs:union>
</xs:simpleType>

<xs:complexType name="simpleType" abstract="true">
<xs:complexContent>
<xs:extension base="xs:annotated">
<xs:group ref="xs:simpleDerivation"/>
<xs:attribute name="final" type="xs:simpleDerivationSet"/>
<xs:attribute name="name" type="xs:NCName">
<xs:annotation>
<xs:documentation>
Can be restricted to required or forbidden
</xs:documentation>
</xs:annotation>
</xs:attribute>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:complexType name="topLevelSimpleType">
<xs:complexContent>
<xs:restriction base="xs:simpleType">
<xs:sequence>
<xs:element ref="xs:annotation" minOccurs="0"/>
<xs:group ref="xs:simpleDerivation"/>
</xs:sequence>
<xs:attribute name="name" use="required" type="xs:NCName">
<xs:annotation>
<xs:documentation>
Required at the top level
</xs:documentation>
</xs:annotation>
</xs:attribute>
</xs:restriction>
</xs:complexContent>
</xs:complexType>
</xs:complexType>

79a
<xs:complexType name="localSimpleType">
  <xs:complexContent>
    <xs:restriction base="xs:simpleType">
      <xs:sequence>
        <xs:element ref="xs:annotation" minOccurs="0"/>
        <xs:group ref="xs:simpleDerivation"/>
      </xs:sequence>
      <xs:attribute name="name" use="prohibited">
        <xs:annotation>
          <xs:documentation>
            Forbidden when nested
          </xs:documentation>
          <xs:attribute>
            <xs:attribute name="final" use="prohibited"/>
          </xs:attribute>
        </xs:annotation>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name="list" id="list">
  <xs:complexType>
    <xs:annotation>
      <xs:documentation>
        Item type attribute and simpleType child are mutually exclusive, but one or other is required.
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:sequence>
          <xs:element name="simpleType" type="xs:localSimpleType" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="itemType" type="xs:QName" use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="union" id="union">
  <xs:complexType>
    <xs:annotation>
      <xs:documentation>
        Member types attribute must be non-empty or there must be at least one simpleType child.
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:sequence>
          <xs:element name="simpleType" type="xs:localSimpleType" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="memberTypes" use="optional"/>
        <xs:list itemType="xs:QName"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:complexType name="facet">
  <xs:complexContent>
    <xs:extension base="xs:annotated">
      <xs:attribute name="value" use="required"/>
      <xs:attribute name="fixed" type="xs:boolean" use="optional" default="false"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="fractionDigits" id="fractionDigits" type="xs:numFacet">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#element-fractionDigits"/>
  </xs:annotation>
</xs:element>

<xs:element name="length" id="length" type="xs:numFacet">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#element-length"/>
  </xs:annotation>
</xs:element>

<xs:element name="minLength" id=" minLength" type="xs:numFacet">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#element-minLength"/>
  </xs:annotation>
</xs:element>

<xs:element name="maxLength" id="maxLength" type="xs:numFacet">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#element-maxLength"/>
  </xs:annotation>
</xs:element>

<xs:element name="enumeration" id="enumeration" type="xs:noFixedFacet">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#element-enumeration"/>
  </xs:annotation>
</xs:element>

<xs:element name="whiteSpace" id="whiteSpace" type="xs:facet">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#element-whiteSpace"/>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:restriction base="xs:facet">
        <xs:sequence>
          <xs:element ref="xs:annotation" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="value" use="required" type="xs:simpleType">
          <xs:restriction base="xs:NMTOKEN">
            <xs:enumeration value="preserve"/>
            <xs:enumeration value="replace"/>
            <xs:enumeration value="collapse"/>
          </xs:restriction>
        </xs:attribute>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

<xs:element name="pattern" id="pattern" type="xs:noFixedFacet">
  <xs:annotation>
    <xs:documentation source="http://www.w3.org/TR/xmlschema-2/#element-pattern"/>
  </xs:annotation>
</xs:element>
B DTD for Datatype Definitions (non-normative)

<-- DTD for XML Schemas: Part 2: Datatypes
Id: datatypes.dtd,v 1.23 2001/03/16 17:36:30 ht Exp
Note this DTD is NOT normative, or even definitive.
-->

<-- This DTD cannot be used on its own, it is intended only for incorporation in XMLSchema.dtd, q.v.
-->

<-- Define all the element names, with optional prefix -->
<ENTITY %simpleType "%p;simpleType">
<ENTITY %restriction "%p;restriction">
<ENTITY %list "%p;list">
<ENTITY %uni on "%p;uni on">
<ENTITY %maxExclusive "%p;maxExclusive">
<ENTITY %minExclusive "%p;minExclusive">
<ENTITY %maxInclusive "%p;maxInclusive">
<ENTITY %minInclusive "%p;minInclusive">
<ENTITY %totalDigits "%p;totalDigits">
<ENTITY %fractionDigits "%p;fractionDigits">
<ENTITY %length "%p;length">
<ENTITY %minLength "%p;minLength">
<ENTITY %maxLength "%p;maxLength">
<ENTITY %enumeration "%p;enumeration">
<ENTITY %whiteSpace "%p;whiteSpace">
<ENTITY %pattern "%p;pattern">

<-- Customisation entities for the ATTLIST of each element type. Define one of these if your schema takes advantage of the anyAttribute=##other in the schema for schemas -->

<ENTITY %simpleTypeAttrs "">
<ENTITY %restrictionAttrs "">
<ENTITY %listAttrs "">
<ENTITY %uni onAttrs "">
<ENTITY %maxExclusiveAttrs "">
<ENTITY %minExclusiveAttrs "">
<ENTITY %maxInclusiveAttrs "">
<ENTITY %minInclusiveAttrs "">
<ENTITY %totalDigitsAttrs "">
<ENTITY %fractionDigitsAttrs "">
<ENTITY %lengthAttrs "">
<ENTITY %minLengthAttrs "">
<ENTITY %maxLengthAttrs "">
<ENTITY %enumerationAttrs "">
<ENTITY %whiteSpaceAttrs "">
<ENTITY %patternAttrs "">

<-- Define some entities for informative use as attribute types -->
<ENTITY %URIref "CDATA">
<ENTITY %XPathExpr "CDATA">
<ENTITY %QName "NM_TOKEN">
<ENTITY %QNames "NM_TOKENS">
<ENTITY %NCName "NM_TOKEN">
<ENTITY %nonNegativeInteger "NM_TOKEN" >
Note that the use of 'facet' below is less restrictive than is really intended: There should in fact be no more than one of each of minInclusive, minExclusive, maxInclusive, maxExclusive, totalDigits, fractionDigits, length, maxLength, minLength within datatype, and the min- and max- variants of Inclusive and Exclusive are mutually exclusive. On the other hand, pattern and enumeration may repeat.

```xml
<ENTITY % minBound "(\%minInclusive; | \%minExclusive;)">
<ENTITY % maxBound "(\%maxInclusive; | \%maxExclusive;)">
<ENTITY % bounds "\%minBound; | \%maxBound;">
<ENTITY % numeric "\%totalDigits; | \%fractionDigits;">
<ENTITY % ordered "\%bounds; | \%numeric;">
<ENTITY % unordered "\%pattern; | \%enumeration; | \%WhiteSpace; | \%ength; | \%maxLength; | \%minLength;">
<ENTITY % facet "\%ordered; | \%unordered;">
<ENTITY % facetAttr "value CDATA #REQUIRED
id I D \# MPL ED">
<ENTITY % fixedAttr "fixed \%boolean; \# MPL ED">
<ENTITY % facetModel "(%annotation;)?">
<ELEMENT %impleType>
((%annotation;)?, (%restriction; | %list; | %union;))>
<ATTLIST %impleType;
name %NCName; \# MPL ED
final \%impleDerivationSet; \# MPL ED
id I D \# MPL ED
%impleTypeAttrs;>
<ENTITY %restriction; ((%annotation;)?,
(%restriction1; | %list; | %union;))>
<ATTLIST %restriction;
base %QName; \# MPL ED
id I D \# MPL ED
%restrictionAttrs;>
<ELEMENT %restriction;
base %QName; \# MPL ED
id I D \# MPL ED
%restrictionAttrs;>
< ELEMENT %simpleType; ((%annotation;)?, (%impleType; | %list; | %union;))>
<ATTLIST %impleType;
name %NCName; \# MPL ED
id I D \# MPL ED
%impleTypeAttrs;>
< ELEMENT %list; ((%annotation;)?, (%impleType;))>
<ATTLIST %list;
itemType %QName; \# IMPLIED
id I D \# MPL ED
%listAttrs;>
< ELEMENT %union; ((%annotation;)?, (%impleType;))>
```

85a
<!ATTIX %uni on;
    id ID # MPLI ED
  memberTypes %QNames; # MPLI ED
%uni onAtrrs;>
-->

<-- At least one item in memberTypes or one simpleType
  child is required
-->

< ELEMENT %maxExcl usive; %acetModel; >
< ATTLIX %maxExcl usive;
  %acetAttr;
  %fixedAttr;
  %maxExcl usiveAtrrs; >
< ELEMENT %minExcl usive; %acetModel; >
< ATTLIX %minExcl usive;
  %acetAttr;
  %fixedAttr;
  %minExcl usiveAtrrs; >
< ELEMENT %maxIncl usive; %acetModel; >
< ATTLIX %maxIncl usive;
  %acetAttr;
  %fixedAttr;
  %maxIncl usiveAtrrs; >
< ELEMENT %minIncl usive; %acetModel; >
< ATTLIX %minIncl usive;
  %acetAttr;
  %fixedAttr;
  %minIncl usiveAtrrs; >
< ELEMENT %total Di gits; %acetModel; >
< ATTLIX %total Di gits;
  %acetAttr;
  %fixedAttr;
  %total Di gitsAtrrs; >
< ELEMENT %fracDi gits; %acetModel; >
< ATTLIX %fracDi gits;
  %acetAttr;
  %fixedAttr;
  %fracDi gitsAtrrs; >
< ELEMENT %ength; %acetModel; >
< ATTLIX %ength;
  %acetAttr;
  %fixedAttr;
  %engthAtrrs; >
< ELEMENT %nLength; %acetModel ; >
< ATTLIX %nLength;
  %acetAttr;
  %fixedAttr;
  %nLengthAtrrs; >
< ELEMENT %maxLength; %acetModel ; >
< ATTLIX %maxLength;
  %acetAttr;
  %fixedAttr;
  %maxLengthAtrrs; >

<-- This one can be repeated -->
< ELEMENT %enumeration; %acetModel; >
< ATTLIX %enumeration;
  %acetAttr;
  %enumerationAtrrs; >

< ELEMENT %whiteSpace; %acetModel; >
C Datatypes and Facets

C.1 Fundamental Facets

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

<table>
<thead>
<tr>
<th>Datatype</th>
<th>ordered</th>
<th>bounded</th>
<th>cardinality</th>
<th>numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
<td>false</td>
<td>finite</td>
<td>false</td>
</tr>
<tr>
<td>float</td>
<td>total</td>
<td>true</td>
<td>finite</td>
<td>true</td>
</tr>
<tr>
<td>double</td>
<td>total</td>
<td>true</td>
<td>finite</td>
<td>true</td>
</tr>
<tr>
<td>decimal</td>
<td>total</td>
<td>false</td>
<td>countably infinite</td>
<td>true</td>
</tr>
<tr>
<td>duration</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>dateTime</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>time</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>date</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gYearMonth</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gYear</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gMonthDay</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gDay</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>gMonth</td>
<td>partial</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>hexBinary</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>base64Binary</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>anyURI</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>QName</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>NOTATION</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>normalizedString</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>token</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>language</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>IDREFS</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>NMTOKEN</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>NMTOKENS</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>Name</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>NCName</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>ID</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>IDREF</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
<tr>
<td>ENTITY</td>
<td>false</td>
<td>false</td>
<td>countably infinite</td>
<td>false</td>
</tr>
</tbody>
</table>
D ISO 8601 Date and Time Formats

D.1 ISO 8601 Conventions

The primitive datatypes duration, dateTime, time, date, gYearMonth, gMonthDay, gDay, gMonth and gYear use lexical formats inspired by [ISO 8601]. 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 digits. For the primitive datatypes dateTime, time, date, gYearMonth, gMonthDay, gDay, gMonth and gYear, 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 23.
- 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 dateTime.
- 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 dateTime, time, date, gYearMonth, gMonthDay, gDay, gMonth, and gYear.

In the lexical format for duration 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 duration 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 time datatype uses a truncated format for dateTime which represents an instant of time that recurs every day. Similarly, the gMonthDay and gDay datatypes use left-truncated formats for date. The datatype gMonth uses a right and left truncated format for date.

[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 dateTime are used as lexical representations for date, gMonth, gYear.

**D.3 Deviations from ISO 8601 Formats**

- **D.3.1 Sign Allowed**
- **D.3.2 No Year Zero**
- **D.3.3 More Than 9999 Years**

**D.3.1 Sign Allowed**

An optional minus sign is allowed immediately preceding, without a space, the lexical representations for duration, dateTime, date, gMonth, gYear.

**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 dateTime, date, gYearMonth, and gYear. This follows [ISO 8601 Draft Revision].

**E Adding durations to dateTimes**

Given a dateTime S and a duration D, this appendix specifies how to compute a dateTime E where E is the end of the time period with start S and duration D i.e. E = S + D. Such computations are used, for example, to determine whether a dateTime is within a specific time period. This appendix also addresses the addition of durations to the datatypes date, gYearMonth, gYear, gDay and gMonth, which can be viewed as a set of dateTimes. In such cases, the addition is made to the first or starting dateTime 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 $S[\text{year}]$ to represent the year field of S, $S[\text{month}]$ to represent the month field, and so on. It also depends on the following functions:

- $f\text{Quotient}(a, b) = \text{the greatest integer less than or equal to } a/b$
\[\text{fQuotient}(-1,3) = -1\]
\[\text{fQuotient}(0,3) = 0\]
\[\text{fQuotient}(3,3) = 1\]
\[\text{fQuotient}(3.123,3) = 1\]

\[\text{modulo}(a, b) = a - \text{fQuotient}(a,b) \times b\]
\[\text{modulo}(-1,3) = 2\]
\[\text{modulo}(0,3) = 0\]...
\[\text{modulo}(2,3) = 0\]...
\[\text{modulo}(3,3) = 0\]
\[\text{modulo}(3.123,3) = 0.123\]

\[\text{fQuotient}(a, low, high) = \text{fQuotient}(a - low, high - low)\]
\[\text{fQuotient}(0, 1, 13) = -1\]
\[\text{fQuotient}(1, 1, 13) = 0\]...
\[\text{fQuotient}(12, 1, 13) = 1\]
\[\text{fQuotient}(13.123, 1, 13) = 1\]

\[\text{modulo}(a, low, high) = \text{modulo}(a - low, high - low) + low\]
\[\text{modulo}(0, 1, 13) = 12\]
\[\text{modulo}(1, 1, 13) = 1\]
\[\text{modulo}(10, 1, 13) = 1\]
\[\text{modulo}(13, 1, 13) = 1.123\]

\[\text{maximumDayInMonthFor(yearValue, monthValue)} =\]
\[M := \text{modulo(monthValue, 1, 13)}\]
\[Y := \text{yearValue} + \text{fQuotient(monthValue, 1, 13)}\]
\[\text{Return a value based on } M \text{ and } Y:\]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>M = January, March, May, July, August, October, or December</td>
</tr>
<tr>
<td>30</td>
<td>M = April, June, September, or November</td>
</tr>
<tr>
<td>29</td>
<td>M = February AND (modulo(Y, 400) = 0 OR (modulo(Y, 100) = 0) AND modulo(Y, 4) = 0)</td>
</tr>
<tr>
<td>28</td>
<td>Otherwise</td>
</tr>
</tbody>
</table>

E.1 Algorithm

Essentially, this calculation is equivalent to separating D into \langle year,month\rangle and \langle day,hour,minute,second\rangle fields. The \langle year,month\rangle 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 \langle day,hour,minute,second\rangle 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)
- 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:

<table>
<thead>
<tr>
<th>dateTime</th>
<th>duration</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01</td>
<td>-P3M</td>
<td>1999-10</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>PT33H</td>
<td>2000-01-13</td>
</tr>
</tbody>
</table>

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:

\[(\text{dateTime} + \text{duration1}) + \text{duration2}) \neq ((\text{dateTime} + \text{duration2}) + \text{duration1})\]

Example:

\[(2000-03-30 + P1D) + P1M = 2000-03-31 + P1M = 2001-04-30\]
\[(2000-03-30 + P1M) + P1D = 2000-04-30 + P1D = 2000-05-01\]

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

For all branches $S$, and for all regular expressions $T$, valid regular expressions $R$ are:

1. $\text{empty string}$
2. $S$
3. $ST$

Denoting the set of strings $L(R)$ containing:

1. The set containing just the empty string
2. All strings in $L(S)$
3. All strings in $L(S)$ and all strings in $L(T)$

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

For all pieces $S$, and for all branches $T$, valid branches $R$ are:

1. $S$
2. $ST$

Denoting the set of strings $L(R)$ containing:

1. All strings in $L(S)$
2. All strings $st$ with $s$ in $L(S)$ and $t$ in $L(T)$

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

For all atoms $S$ and non-negative integers $n$, $m$ such that $n \leq m$, valid pieces $R$ are:

1. $S$
2. $S^*$
3. $S^+$
4. $S[n,m]$
5. $S(n)$
6. $S(n,)$
7. $S(0,m)$
8. $S(0,0)$

Denoting the set of strings $L(R)$ containing:

1. All strings in $L(S)$
2. 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 $L(S)$)
3. All strings $st$ with $s$ in $L(S)$ and $t$ in $L(S^*)$. (all concatenations of one or more strings from $L(S)$)
4. All strings $st$ with $s$ in $L(S)$ and $t$ in $L(S[n,m])$. (All sequences of at least $n$, and at most $m$, strings from $L(S)$)
5. All strings in $L(S(n,n))$. (All sequences of exactly $n$ strings from $L(S)$)
6. All strings in $L(S(n)n)$ (All sequences of at least $n$, strings from $L(S)$)
7. All strings $st$ with $s$ in $L(S)$ and $t$ in $L(S(0,m-1))$. (All sequences of at most $m$, strings from $L(S)$)
8. The set containing only the empty string

NOTE: The regular expression language in the Perl Programming Language [Perl] does not include a quantifier of the form $S\{,m\}$, since it is logically equivalent to $S\{0,m\}$. We have, therefore, left this logical possibility out of
the regular expression language defined by this specification. We welcome further input from implementors and
schema authors on this issue.

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

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>::=</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[*+]</code></td>
<td>`{ quantRange</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>quantRange</th>
<th>::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>quantMin</td>
<td>QuantExact</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>quantMin</th>
<th>::=</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0-9</code></td>
<td><code>+</code></td>
</tr>
</tbody>
</table>

For all normal characters `c`, character classes `C`, and regular expressions `S`, valid **atom** `R` are:

<table>
<thead>
<tr>
<th>Valid <strong>atom</strong> <code>R</code></th>
<th>Denoting the set of strings <code>L(R)</code> containing:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c</code></td>
<td>the single string consisting only of <code>c</code></td>
</tr>
<tr>
<td><code>C</code></td>
<td>all strings in <code>L(C)</code></td>
</tr>
<tr>
<td><code>S</code></td>
<td>all strings in <code>L(S)</code></td>
</tr>
</tbody>
</table>

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

[Definition:] A **normal character** is any XML character that is not a metacharacter. In regular expression `s`, a normal character is an **atom** that denotes the singleton set of strings containing only itself.

<table>
<thead>
<tr>
<th>Normal Character</th>
<th>::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>`[^.?*+()</td>
<td>]`</td>
</tr>
</tbody>
</table>

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

**F.1 Character Classes**

[Definition:] A **character class** is an **atom** `R` that identifies a **set of characters** `C(R)`. The set of strings `L(R)` denoted by a character class `R` contains one single-character string “c” for each character `c` in `C(R)`.

<table>
<thead>
<tr>
<th>Character Class</th>
<th>::=</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>charClassEsc</code></td>
<td><code>charClassExpr</code></td>
</tr>
</tbody>
</table>

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

[Definition:] A **character class expression** is a **character group** surrounded by `[` and `]` characters. For all character groups `G`,

---

93a
[G] is a valid character class expression, identifying the set of characters $C([G]) = C(G)$.

**Character Class Expression**

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

[Definition:] A character group is either a positive character group, a negative character group, or a character class subtraction.

**Character Group**

[13] charGroup ::= posCharGroup | negCharGroup | charClassSub

[Definition:] A positive character group consists of one or more character range or character class escape, 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 ::= ( charRange | charClassEsc )+

For all character range $R$, all character class escape $E$, and all positive character group $P$, valid positive character groups $G$ are:

<table>
<thead>
<tr>
<th>$R$</th>
<th>identifying the set of characters $C(G)$ containing:</th>
</tr>
</thead>
<tbody>
<tr>
<td>all characters in $C(R)$.</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>all characters in $C(E)$.</td>
</tr>
<tr>
<td>$RP$</td>
<td>all characters in $C(R)$ and all characters in $C(P)$.</td>
</tr>
<tr>
<td>$EP$</td>
<td>all characters in $C(E)$ and all characters in $C(P)$.</td>
</tr>
</tbody>
</table>

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

**Negative Character Group**

[15] negCharGroup ::= ' ^ ' posCharGroup

[Definition:] 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 ::= ( posCharGroup | negCharGroup ) ' - ' charClassExpr

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 $C(C)$.

[Definition:] A character range $R$ identifies a set of characters $C(R)$ containing all XML characters with UCS code points in a specified range.
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; and
- The - character is a valid character range only at the beginning or end of a positive character group.

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;

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

F.1.1 Character Class Escapes

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

Character Class Escape

\[ 23 \]

\[
\text{charClassEsc} ::= ( \text{SingleCharEsc} | \text{MultiCharEsc} | \text{catEsc} | \text{complEsc} )
\]

[Definition:] A single character escape identifies a set containing a single character -- usually because that character is difficult or impossible to write directly into a regular expression.

Single Character Escape

\[ 24 \]

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

The valid single character escape s are: identifying the set of characters \( C(R) \) containing:
[Definition:] [Unicode Database] specifies a number of possible values for the “General Category” property and provides mappings from code points to specific character properties. The set containing all characters that have property X, can be identified with a category escape \p{X}. The complement of this set is specified with the category escape \P{X}. (\[ \P{X} \] = [\^[p{X}]}).

NOTE: [Unicode Database] is subject to future revision. For example, the mapping from code points to specific 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.

<table>
<thead>
<tr>
<th>Category</th>
<th>Property</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letters</td>
<td>L</td>
<td>All Letters</td>
</tr>
<tr>
<td></td>
<td>Lu</td>
<td>uppercase</td>
</tr>
<tr>
<td></td>
<td>Li</td>
<td>lowercase</td>
</tr>
<tr>
<td></td>
<td>Lt</td>
<td>titlecase</td>
</tr>
<tr>
<td></td>
<td>Lm</td>
<td>modifier</td>
</tr>
<tr>
<td></td>
<td>Lo</td>
<td>other</td>
</tr>
<tr>
<td>Marks</td>
<td>M</td>
<td>All Marks</td>
</tr>
<tr>
<td></td>
<td>Mn</td>
<td>nonspace</td>
</tr>
</tbody>
</table>

\[
\begin{array}{|c|c|}
\hline
\text{n} & \text{the newline character (#xA)} \\
\text{r} & \text{the return character (#xD)} \\
\text{t} & \text{the tab character (#x9)} \\
\text{\\} & \text{\textbackslash} \\
\text{\|} & \text{\textbar} \\
\text{.} & \text{.} \\
\text{-} & \text{-} \\
\text{\^} & \text{^} \\
\text{\?} & \text{?} \\
\text{*} & \text{*} \\
\text{+} & \text{+} \\
\text{\{} & \text{\{} \\
\text{\}} & \text{\}} \\
\text{\(} & \text{\{} \\
\text{\)} & \text{\}} \\
\text{\[} & \text{\{} \\
\text{\]} & \text{\}} \\
\hline
\end{array}
\]
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mc</td>
<td>spacing combining</td>
</tr>
<tr>
<td></td>
<td>Me</td>
<td>enclosing</td>
</tr>
<tr>
<td>Numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>All Numbers</td>
<td></td>
</tr>
<tr>
<td>Nd</td>
<td>decimal digit</td>
<td></td>
</tr>
<tr>
<td>NL</td>
<td>letter</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>other</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>All Punctuation</td>
<td></td>
</tr>
<tr>
<td>Pc</td>
<td>connector</td>
<td></td>
</tr>
<tr>
<td>Pd</td>
<td>dash</td>
<td></td>
</tr>
<tr>
<td>Ps</td>
<td>open</td>
<td></td>
</tr>
<tr>
<td>Pe</td>
<td>close</td>
<td></td>
</tr>
<tr>
<td>Pi</td>
<td>initial quote (may behave like Ps or Pe depending on usage)</td>
<td></td>
</tr>
<tr>
<td>Pf</td>
<td>final quote (may behave like Ps or Pe depending on usage)</td>
<td></td>
</tr>
<tr>
<td>Po</td>
<td>other</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>All Separators</td>
<td></td>
</tr>
<tr>
<td>Zs</td>
<td>space</td>
<td></td>
</tr>
<tr>
<td>Zl</td>
<td>line</td>
<td></td>
</tr>
<tr>
<td>Zp</td>
<td>paragraph</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>All Symbols</td>
<td></td>
</tr>
<tr>
<td>Sm</td>
<td>math</td>
<td></td>
</tr>
<tr>
<td>Sc</td>
<td>currency</td>
<td></td>
</tr>
<tr>
<td>Sk</td>
<td>modifier</td>
<td></td>
</tr>
<tr>
<td>So</td>
<td>other</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>All Others</td>
<td></td>
</tr>
<tr>
<td>Cc</td>
<td>control</td>
<td></td>
</tr>
<tr>
<td>Cf</td>
<td>format</td>
<td></td>
</tr>
<tr>
<td>Co</td>
<td>private use</td>
<td></td>
</tr>
<tr>
<td>Cn</td>
<td>not assigned</td>
<td></td>
</tr>
</tbody>
</table>

**Categories**

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letters</td>
<td>'L' [ultmo]</td>
</tr>
<tr>
<td>Marks</td>
<td>'M' [nce]</td>
</tr>
<tr>
<td>Numbers</td>
<td>'N' [dlo]</td>
</tr>
<tr>
<td>Punctuation</td>
<td>'P' [cdseifo]</td>
</tr>
<tr>
<td>Separators</td>
<td>'Z' [slp]</td>
</tr>
<tr>
<td>Symbols</td>
<td>'S' [mcko]</td>
</tr>
<tr>
<td>Others</td>
<td>'C' [cfon]</td>
</tr>
</tbody>
</table>

**NOTE:** The properties mentioned above exclude the Cs property. The Cs property identifies "surrogate"
characters, which do not occur at the level of the "character abstraction" that XML instance documents operate on.

[Definition:] [Unicode Database] groups code points into a number of blocks such as Basic Latin (i.e., ASCII), Latin-1 Supplement, Hangul Jamo, CJK Compatibility, etc. The set containing all characters that have block name \( X \) (with all white space stripped out), can be identified with \( \text{a block escape} \backslash \text{p}\{1 \ sX\}. \) The complement of this set is specified with the \( \text{block escape} \backslash \text{P}\{1 \ sX\}. \ (\{1 \ \text{p}\{1 \ sX\}\} = \{\ \backslash \text{p}\{1 \ sX\}\}). \)

<table>
<thead>
<tr>
<th>Start Code</th>
<th>End Code</th>
<th>Block Name</th>
<th>Start Code</th>
<th>End Code</th>
<th>Block Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>#x0000</td>
<td>#x007F</td>
<td>Basic Latin</td>
<td>#x0080</td>
<td>#x00FF</td>
<td>Latin-1 Supplement</td>
</tr>
<tr>
<td>#x0100</td>
<td>#x017F</td>
<td>LatinExtended-A</td>
<td>#x0180</td>
<td>#x024F</td>
<td>LatinExtended-B</td>
</tr>
<tr>
<td>#x0250</td>
<td>#x02AF</td>
<td>IPAExtensions</td>
<td>#x0280</td>
<td>#x02FF</td>
<td>SpacingModifierLetters</td>
</tr>
<tr>
<td>#x0300</td>
<td>#x036F</td>
<td>CombiningDiacriticalMarks</td>
<td>#x0370</td>
<td>#x03FF</td>
<td>Greek</td>
</tr>
<tr>
<td>#x0400</td>
<td>#x04FF</td>
<td>Cyrillic</td>
<td>#x0530</td>
<td>#x058F</td>
<td>Armenian</td>
</tr>
<tr>
<td>#x0590</td>
<td>#x05FF</td>
<td>Hebrew</td>
<td>#x0600</td>
<td>#x06FF</td>
<td>Arabic</td>
</tr>
<tr>
<td>#x0700</td>
<td>#x074F</td>
<td>Syriac</td>
<td>#x0780</td>
<td>#x07BF</td>
<td>Thaana</td>
</tr>
<tr>
<td>#x0900</td>
<td>#x097F</td>
<td>Devanagari</td>
<td>#x0980</td>
<td>#x09FF</td>
<td>Bengali</td>
</tr>
<tr>
<td>#x0A00</td>
<td>#x0A7F</td>
<td>Gurumuki</td>
<td>#x0A80</td>
<td>#x0AFF</td>
<td>Gujarati</td>
</tr>
<tr>
<td>#x0B00</td>
<td>#x0B7F</td>
<td>Oriya</td>
<td>#x0B80</td>
<td>#x0BFF</td>
<td>Tamil</td>
</tr>
<tr>
<td>#x1000</td>
<td>#x107F</td>
<td>Telugu</td>
<td>#x1080</td>
<td>#x10FF</td>
<td>Kannada</td>
</tr>
<tr>
<td>#x10A0</td>
<td>#x10FF</td>
<td>Malayalam</td>
<td>#x10B0</td>
<td>#x10FF</td>
<td>Sinhala</td>
</tr>
<tr>
<td>#x1100</td>
<td>#x117F</td>
<td>Thai</td>
<td>#x1180</td>
<td>#x11FF</td>
<td>HangulJamo</td>
</tr>
<tr>
<td>#x1200</td>
<td>#x137F</td>
<td>Ethiopic</td>
<td>#x13A0</td>
<td>#x13FF</td>
<td>Cherokee</td>
</tr>
<tr>
<td>#x1400</td>
<td>#x167F</td>
<td>UnifiedCanadianAboriginalSyllabics</td>
<td>#x1680</td>
<td>#x16FF</td>
<td>Ogham</td>
</tr>
<tr>
<td>#x16A0</td>
<td>#x16FF</td>
<td>Runic</td>
<td>#x1780</td>
<td>#x17FF</td>
<td>Khmer</td>
</tr>
<tr>
<td>#x1800</td>
<td>#x18AF</td>
<td>Mongolian</td>
<td>#x1E00</td>
<td>#x1EAF</td>
<td>LatinExtendedAdditional</td>
</tr>
<tr>
<td>#x1F00</td>
<td>#x1FFF</td>
<td>GreekExtended</td>
<td>#x2000</td>
<td>#x20FF</td>
<td>GeneralPunctuation</td>
</tr>
<tr>
<td>#x2070</td>
<td>#x209F</td>
<td>SuperscriptsandSubscripts</td>
<td>#x2100</td>
<td>#x21FF</td>
<td>CurrencySymbols</td>
</tr>
<tr>
<td>#x2000</td>
<td>#x20FF</td>
<td>CombiningMarksforSymbols</td>
<td>#x2110</td>
<td>#x214F</td>
<td>LetterlikeSymbols</td>
</tr>
<tr>
<td>#x2150</td>
<td>#x218F</td>
<td>NumberForms</td>
<td>#x2190</td>
<td>#x21FF</td>
<td>Arrows</td>
</tr>
<tr>
<td>#x2200</td>
<td>#x22FF</td>
<td>MathematicalOperators</td>
<td>#x2300</td>
<td>#x23FF</td>
<td>MiscellaneousTechnical</td>
</tr>
<tr>
<td>#x2400</td>
<td>#x243F</td>
<td>ControlPictures</td>
<td>#x2440</td>
<td>#x245F</td>
<td>OpticalCharacterRecognition</td>
</tr>
<tr>
<td>#x2460</td>
<td>#x24FF</td>
<td>EnclosedAlphanumerics</td>
<td>#x2500</td>
<td>#x257F</td>
<td>BoxDrawing</td>
</tr>
<tr>
<td>#x2580</td>
<td>#x259F</td>
<td>BlockElements</td>
<td>#x25A0</td>
<td>#x25FF</td>
<td>GeometricShapes</td>
</tr>
<tr>
<td>#x2600</td>
<td>#x263F</td>
<td>MiscellaneousSymbols</td>
<td>#x2700</td>
<td>#x27BF</td>
<td>Dingbats</td>
</tr>
</tbody>
</table>

The following table specifies the recognized block names (for more information, see the "Blocks.txt" file in [Unicode Database]).
NOTE: [Unicode Database] is subject to future revision. For example, the grouping of code points into blocks might 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}.

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

Multi-Character Escape

[ 37] MultiCharEsc ::= \.

Character sequence | Equivalent character class
--- | ---
\ . | \^[\n\r\v\s\t\n\r\s\]| (\'\' [IsIc CdDwW])
\s | [Is20t
\r\s
\i the set of initial name characters, those matched by Letter
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.

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.

atomic
Atomic datatypes are those having values which are regarded by this specification as being indivisible.

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.

bounded
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 and an exclusive lower bound.

built-in
Built-in datatypes are those which are defined in this specification, and can be either primitive or derived;

canonical lexical representation
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.

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.

conformance to the XML Representation of Schemas
Processors which accept schemas in the form of XML documents as described in XML Representation of Simple Type Definition Schema Components (§4.1.2) (and other relevant portions of Datatype components (§4)) 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 Constraint s in this specification, and must adhere exactly to the specifications in XML Representation of Simple Type Definition Schema Components (§4.1.2) (and other relevant portions of Datatype components (§4)) for mapping the contents of such documents to schema components for use in validation.

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

Constraint on Schemas
Constraint on Schemas

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 facet s that characterize properties of the value space, individual values or lexical items.

derived
Derived datatypes are those that are defined in terms of other datatypes.

error
exclusive lower bound
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 \( V \), \( l < v \).

exclusive upper bound
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 \( V \), \( u > v \).

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

for compatibility
for compatibility

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

inclusive lower bound
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 \( V \), \( l \leq v \).

inclusive upper bound
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 \( V \), \( u \geq v \).

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

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

list
List datatypes are those having values each of which consists of a finite-length (possibly empty) sequence of values of an atomic datatype.

match
match

may
may

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

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

must
must

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

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

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

order-relation
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.

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

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

regular expression
A regular expression is composed from zero or more branch es, separated by | characters.

restriction
A datatype is said to be derived by restriction from another datatype when values for zero or more constraining facet s are specified that serve to constrain its value space and/or its lexical space to a subset of those of its base type.

Schema Representation Constraint
Schema Representation Constraint

total order
A total order is an partial order such that for no \( a \) and \( b \) is it the case that \( a <> b \).

union
Union datatypes are those whose value space s and lexical space s are the union of the value space s and lexical
space s of one or more other datatypes.

**User-derived** datatypes are those derived datatypes that are defined by individual schema designers.

**Validation Rule**

**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.

### H References

#### H.1 Normative

**Clinger, WD (1990)**


**IEEE 754-1985**


**Namespaces in XML**


**RFC 1766**


**RFC 2045**


**RFC 2396**


**RFC 2732**


**Unicode Database**

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

**XML 1.0 (Second Edition)**


**XML Linking Language**


**XML Schema Part 1: Structures**


**XML Schema Requirements**


#### H.2 Non-normative

**Character Model**


**Gay, DM (1990)**


**HTML 4.01**


**IETF INTERNET-DRAFT: IRIs**


**International Earth Rotation Service (IERS)**


102a
ISO 11404

ISO 8601

ISO 8601 Draft Revision

Perl

RDF Schema

Ruby

SQL

U.S. Naval Observatory Time Service Department
Information about Leap Seconds Available at: http://tycho.usno.navy.mil/leapsec.990505.html

Unicode Regular Expression Guidelines

XML Schema Language: Part 2 Primer

XSL

I Acknowledgements (non-normative)

The following have contributed material to this draft:

- 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.

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.

The current members of the XML Schema Working Group are:

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 Holland, 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