W3C Recommendation

XQuery and XPath Data Model 3.1

W3C Recommendation 21 March 2017

This version:
https://www.w3.org/TR/2017/REC-xpath-datamodel-31-20170321/

Latest version of XQuery and XPath Data Model 3.1:
https://www.w3.org/TR/xpath-datamodel-31/

Previous versions of XQuery and XPath Data Model 3.1:
https://www.w3.org/TR/2017/PR-xpath-datamodel-31-20170117/
https://www.w3.org/TR/2016/CR-xpath-datamodel-31-20161213/
https://www.w3.org/TR/2014/CR-xpath-datamodel-31-20141218/

Most recent version of XQuery and XPath Data Model 3:
https://www.w3.org/TR/xpath-datamodel-3/

Most recent version of XQuery and XPath Data Model:
https://www.w3.org/TR/xpath-datamodel/

Most recent Recommendation of XQuery and XPath Data Model:
https://www.w3.org/TR/2014/REC-xpath-datamodel-30-20140408/

Editors:
Norman Walsh (XML Query WG), MarkLogic Corporation <Norman.Walsh@marklogic.com>
John Snelson (XML Query WG) , MarkLogic Corporation <john.snelson@marklogic.com>
Andrew Coleman (XML Query WG), IBM Hursley Laboratories <andrew_coleman@uk.ibm.com>

Please check the **errata** for any errors or issues reported since publication.

See also **translations**.

This document is also available in these non-normative formats: XML and Change markings relative to previous edition.

Abstract

This document defines the XQuery and XPath Data Model 3.1, which is the data model of [XML Path Language (XPath) 3.1], [XSL Transformations (XSLT) Version 3.0], and [XQuery 3.1: An XML Query Language] , and any other specifications that reference it. This document is the result of joint work by the [XSLT Working Group] and the [XML Query Working Group].

Status of this Document

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

This document is governed by the 1 March 2017 W3C Process Document.

This is a Recommendation of the W3C.

This document was published by the W3C XML Query Working Group and the W3C XSLT Working Group, each of which is part of the XML Activity.

This Recommendation specifies the XQuery and XPath Data Model (XDM) version 3.1, a fully compatible extension of XDM version 3.0.

This specification is designed to be referenced normatively from other specifications defining a host language for it; it is not intended to be implemented outside a host language. The implementability of this specification has been tested in the context of its normative inclusion in host languages defined by the XQuery 3.1 and XSLT 3.0 (expected in 2017) specifications; see the XQuery 3.1 implementation report (and, in the future, the WGs expect that there will also be an XSLT 3.0 implementation report) for details.

No substantive changes have been made to this specification since its publication as a Proposed Recommendation.

Please report errors in this document using W3C's public Bugzilla system (instructions can be found at https://www.w3.org/XML/2005/04/qt-bugzilla). If access to that system is not feasible, you may send your comments to the W3C XSLT/XPath/XQuery public comments mailing list, public-qt-comments@w3.org. It will be very helpful if you include the string “[XDM31]” in the subject line of your report, whether made in Bugzilla or in email. Please use multiple Bugzilla entries (or, if necessary, multiple email messages) if you have more than one comment to make. Archives of the comments and responses are available at https://lists.w3.org/Archives/Public/public-qt-comments/.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited 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.

Table of Contents

✧ ✧ ✧

1 Introduction

This document defines the XQuery and XPath Data Model 3.1, which is the data model of [XML Path Language (XPath) 3.1], [XSL Transformations (XSLT) Version 3.0], and [XQuery 3.1: An XML Query Language] .

The XQuery and XPath Data Model 3.1 (henceforth "data model") serves two purposes. First, it defines the information contained in the input to an XSLT or XQuery processor. Second, it defines all permissible values of expressions in the XSLT, XQuery, and XPath languages. A language is *closed* with respect to a data model if the value of every expression in the language is guaranteed to be in the data model. XSLT 3.0, XQuery 3.1, and XPath 3.1 are all closed with respect to the data model.

The data model describes items similar to those of the [Infoset] (henceforth "Infoset") . It is written to provide a data model suitable for XPath, XQuery and XSLT, which was not a goal of the Infoset, and this leads to a number of differences, some of which are:

- Support for XML Schema types. The XML Schema recommendations define features, such as structures ([Schema Part 1]) and simple data types ([Schema Part 2]), that extend the Infoset with precise type information.
- Representation of collections of documents and of complex values. ([XQuery 3.1 Requirements])
- Support for typed atomic values.
- Support for ordered, heterogeneous sequences.

As with the Infoset, the XQuery and XPath Data Model 3.1 specifies what information in the documents is accessible but does not specify the programming-language interfaces or bindings used to represent or access the data.

The data model can represent various values including not only the input and the output of a stylesheet or query, but all values of expressions used during the intermediate calculations. Examples include the input document or document repository (represented as a Document Node or a sequence of Document Nodes), the result of a path expression (represented as a sequence of nodes), the result of an arithmetic or a logical expression (represented as an atomic value), a sequence expression resulting in a sequence of items, etc.

This document provides a precise definition of the properties of nodes in the XQuery and XPath Data Model 3.1, how they are accessed, and how they relate to values in the Infoset and PSVI.

2 Concepts

This section outlines a number of general concepts that apply throughout this specification.

In this document, examples and material labeled as "Note" are provided for explanatory purposes and are not normative.

2.1 Terminology

For a full glossary of terms, see **D Glossary**.

In this specification the words **must**, **must not**, **should**, **should not**, **may** and **recommended** are to be interpreted as described in [RFC 2119].

This specification distinguishes between the data model as a general concept and specific items (documents, elements, atomic values, etc.) that are concrete examples of the data model by identifying all concrete examples as instances of the data model.

[Definition: Every **instance of the data model** is a sequence.]

[Definition: A **sequence** is an ordered collection of zero or more items.] A sequence cannot be a member of a sequence. A single item appearing on its own is modeled as a sequence containing one item. Sequences are defined in **2.5 Sequences**.

[Definition: An **item** is either a node, a function, or an atomic value.]

Sometimes it is necessary to distinguish the case where a particular property has no value in the data model. The canonical example of such a case is the namespace URI property of an expanded QName that is not in any namespace. For such properties, it is convenient to be able to speak of "the state of having no value". [Definition: When a property has no value, we say that it is **absent**.]

Every node is one of the seven kinds of nodes defined in Section **6 Nodes**. Nodes form a tree. Each node has at most one parent (reachable via the `dm:parent` accessor) and descendant nodes that are reachable directly or indirectly via the `dm:children`, `dm:attributes`, and `dm:namespace-nodes` accessors.

[Definition: The **root node** is the topmost node of a tree, the node with no parent.] Every tree has exactly one root node and every other node can be reached from exactly one root node.

> **Note:**
>
> The term “root node” is merely a designator, based on position, for one of the nodes in the tree without implying what kind of a node it is. In the XPath 1.0 datamodel the root node was a kind of node.

[Definition: A tree whose root node is a Document Node is referred to as a **document**.]

[Definition: A tree whose root node is not a Document Node is referred to as a **fragment**.]

[Definition: An **atomic value** is a value in the value space of an atomic type and is labeled with the name of that atomic type.]

[Definition: An **atomic type** is a primitive simple type or a type derived by restriction from another atomic type.] (Types derived by list or union are not atomic.)

[Definition: The **primitive simple types** are the types defined in **2.1.1 Types adopted from XML Schema**.]

A type is represented in the data model by an expanded-QName.

[Definition: An **expanded-QName** is a set of three values consisting of a possibly empty prefix, a possibly empty namespace URI, and a local name.] See **3.3.3 QNames and NOTATIONS**.

[Definition: **Implementation-defined** indicates an aspect that may differ between implementations, but must be specified by the implementor for each particular implementation.]

[Definition: **Implementation-dependent** indicates an aspect that may differ between implementations, is not specified by this or any W3C specification, and is not required to be specified by the implementor for any particular implementation.]

Within this specification, the term URI refers to a Uniform Resource Identifier as defined in [RFC 3986] and extended in [RFC 3987] with the new name IRI. The term URI has been retained in preference to IRI to avoid introducing new names for concepts such as “Base URI” that are defined or referenced across the whole family of XML specifications.

In all cases where this specification leaves the behavior implementation-defined or implementation-dependent, the implementation has the option of providing mechanisms that allow the user to influence the behavior.

2.1.1 Types adopted from XML Schema

The data model adopts the following types:

- The 19 types defined in Section 3.2 Primitive datatypes*XS2* of [Schema Part 2].
- Three built-in list types: `xs:NMTOKENS`, `xs:IDREFS`, and `xs:ENTITIES`.

  The following types which were originally defined in [XQuery 1.0 and XPath 2.0 Data Model (XDM)] and were subsequently adopted by [Schema 1.1 Part 2]: `xs:anyAtomicType`, `xs:dayTimeDuration`, `xs:yearMonthDuration`.
- In the case of a processor that supports [Schema 1.1 Part 2], the data model also includes: the new union type `xs:error` (a type with no instances) and the new derived type `xs:dateTimeStamp`.
- The following types, although they use the `xs:` namespace, are defined here in the data model and not in XML Schema: `xs:untypedAtomic` , and `xs:numeric`, a union type whose members are `xs:double`, `xs:float` and `xs:decimal`.

2.2 Notation

In addition to prose, this specification defines a set of accessor functions to explain the data model. The accessors are shown with the prefix *dm:*. This prefix is always shown in italics to emphasize that these functions are abstract; they exist to explain the interface between the data model and specifications that rely on the data model: they are not accessible directly from the host language.

Several prefixes are used throughout this document for notational convenience. The following bindings are assumed.

1. `xs:` bound to `http://www.w3.org/2001/XMLSchema`
2. `xsi:` bound to `http://www.w3.org/2001/XMLSchema-instance`
3. `fn:` bound to `http://www.w3.org/2005/xpath-functions`

In practice, any prefix that is bound to the appropriate URI may be used.

The signature of accessor functions is shown using the same style as [XQuery and XPath Functions and Operators 3.1], described in Section 1.4 Function signatures and descriptions *FO31*.

This document relies on the [Infoset] and Post-Schema-Validation Infoset (PSVI). Information items and properties are indicated by the styles **information item** and **[infoset property]**, respectively.

Some aspects of type assignment rely on the ability to access properties of the schema components. Such properties are indicated by the style {component property}. Note that this does not mean a lightweight schema processor cannot be used, it only means that the application must have some mechanism to access the necessary properties.

2.3 Node Identity

Each node has a unique identity. The identity of a node is distinct from its value or other visible properties; nodes may be distinct even when they have the same values for all intrinsic properties other than their identity. (The identity of atomic values, by contrast, is determined solely by their intrinsic properties. No two distinct integers, for example, have the same value; every instance of the value “5” as an integer is identical to every other instance of the value “5” as an integer.)

Note:

The concept of node identity should not be confused with the concept of a unique ID, which is a unique name assigned to an element by the author to represent references using ID/IDREF correlation.

2.4 Document Order

[Definition: A **document order** is defined among all the nodes accessible during a given query or transformation. Document order is a total ordering, although the relative order of some nodes is implementation-dependent. Informally, document order is the order in which nodes appear in the XML serialization of a document.] [Definition: Document order is **stable**, which means that the relative order of two nodes will not change during the processing of a given query or transformation, even if this order is implementation-dependent.]

Within a tree, document order satisfies the following constraints:

1. The root node is the first node.
2. Every node occurs before all of its children and descendants.
3. Namespace Nodes immediately follow the Element Node with which they are associated. The relative order of Namespace Nodes is stable but implementation-dependent.
4. Attribute Nodes immediately follow the Namespace Nodes of the element with which they are associated. If there are no Namespace Nodes associated with a given element, then the Attribute Nodes associated with that element immediately follow the element. The relative order of Attribute Nodes is stable but implementation-dependent.
5. The relative order of siblings is the order in which they occur in the **children** property of their parent node.
6. Children and descendants occur before following siblings.

The relative order of nodes in distinct trees is stable but implementation-dependent, subject to the following constraint: If any node in a given tree, `T1`, occurs before any node in a different tree, `T2`, then all nodes in `T1` are before all nodes in `T2`.

2.5 Sequences

An important characteristic of the data model is that there is no distinction between an item (a node, function, or atomic value) and a singleton sequence containing that item. An item is equivalent to a singleton sequence containing that item and vice versa.

A sequence may contain any mixture of nodes, functions, and atomic values. When a node is added to a sequence its identity remains the same. Consequently a node may occur in more than one sequence and a sequence may contain duplicate items.

Sequences never contain other sequences; if sequences are combined, the result is always a “flattened” sequence. In other words, appending “(d e)” to “(a b c)” produces a sequence of length 5: “(a b c d e)”. It *does not* produce a sequence of length 4: “(a b c (d e))”, such a nested sequence never occurs.

Note:

Sequences replace node-sets from XPath 1.0. In XPath 1.0, node-sets do not contain duplicates. In generalizing node-sets to sequences in XPath 3.0 and XPath 3.1, duplicate removal is provided by functions on node sequences.

Note:

Arrays and maps are derived from functions and therefore can also be contained within sequences.

2.6 Namespace Names

The specifications [Namespaces in XML] and [Namespaces in XML 1.1] introduce the concept of a namespace name. In [Namespaces in XML] a namespace name is required to be a URI; in [Namespaces in XML 1.1] it is required to be an IRI; but both specifications explicitly do not require a processor to check that namespace names appearing in an instance document are in fact valid URIs or IRIs.

[Definition: This specification uses the term **Namespace URI** to refer to a namespace name, whether or not it is a valid URI or IRI]. Following the lead of [Namespaces in XML] and [Namespaces in XML 1.1], processors implementing this data model **may** raise an error if a namespace name is not a valid URI or IRI (depending on whether they support [Namespaces in XML] or [Namespaces in XML 1.1]), but they are *not required* to make any checks. Note that the use of a relative reference as a namespace name is deprecated and is defined to be meaningless, but it is not an error. Namespace names, whatever form they take, are treated as character strings and compared for equality using codepoint-by-codepoint comparison, subject only to whitespace normalization if they appear in a context (for example, within an attribute value) where this is appropriate.

In some interfaces, namespace names are held as values of type `xs:anyURI`. For example, the namespace part of an expanded QName is defined to be a value of type `xs:anyURI`. In [Schema Part 2], the type `xs:anyURI` imposes some restrictions on the value space, but there is latitude for implementors to decide exactly what these restrictions are. In [Schema 1.1 Part 2] there are no restrictions on the form of an `xs:anyURI` value, so any sequence of characters is acceptable within the value space. In this and related specifications, the use of the type `xs:anyURI` to hold a namespace name does not imply any restrictions on the value space beyond those described in this section: implementations **may** reject character strings that are not valid URIs or IRIs, but they are *not required* to do so.

2.7 Schema Information

The data model supports strongly typed languages such as [XML Path Language (XPath) 3.1] and [XQuery 3.1: An XML Query Language] that have a type system based on [Schema Part 1]. To achieve this, the data model includes (by reference) the Schema Component Model described in [Schema Part 1].

Note:

The Schema Component Model includes a number of kinds of component, such as type definitions and element and attribute declarations, and defines the properties and relationships of these components. Many of these components and properties are not used by the language specifications that rely on XDM, and where this is the case, there is no requirement for implementations to make them visible. However, this specification makes no attempt to define the minimal subset of the schema component model that is needed to support the semantics of XPath and XQuery processing.

There are two main areas where the language semantics depend on information in schema components:

1. Expressions are evaluated with respect to a static context, which includes schema components, specifically type definitions, element declarations, and attribute declarations. The names of such components may be used in language constructs only if the components are present in the static context.
2. Values including element and attribute nodes, and atomic values, have a property called a type annotation whose value is a type: this is a reference to a type definition in the Schema Component Model.

Every item in the data model has both a value and a type. In addition to nodes, the data model can represent atomic values like the number 5 or the string “Hello World.” For each of these atomic values, the data model contains both the value of the item (such as 5 or “Hello World”) and its type. The property that holds the type is sometimes referred to as the type annotation: its value is a type definition component as defined in the Schema Component Model. This may be a built-in type (a type with a name such as `xs:integer` or `xs:string`), or a user-defined type.

There is a constraint that the total set of components used during expression processing (both statically and dynamically) must constitute a valid schema. This implies, for example, that this total set does not include two different types with the same expanded name.

Note:

This makes it the responsibility of the processor to ensure that the schema components used in the static context of a query or expression during static analysis are consistent with the schema components used to validate documents during query or expression evaluation. This specification does not say how this should be achieved.

It is also a constraint that the schema available to the processor must contain at least the components and properties needed to correctly implement the semantics of the XPath and XQuery language. For example, this means that given a node with a particular type annotation T, and a function that expects an argument of type S, there must be sufficient information available to the processor to establish whether or not T is derived from S. As with other consistency constraints described in this data model, it is a precondition that these constraints are satisfied; the specifications do not speculate on what happens if they are not.

2.7.1 Representation of Types

The data model uses expanded-QNames to represent the names of schema types, which include the built-in types defined by [Schema Part 2], five additional types defined by this specification, and may include other user- or implementation-defined types.

For XML Schema types, the namespace name of the expanded-QName is the {target namespace} property of the type definition, and its local name is the {name} property of the type definition.

The data model relies on the fact that an expanded-QName uniquely identifies every named type. Although it is possible for different schemas to define different types with the same expanded-QName, at most one of them can be used in any given validation episode. The data model cannot support environments where different types with the same expanded-QName are available.

The scope over which the names of anonymous types must be meaningful and distinct is depends on the processing context. It is the responsibility of the host language to define the size and scope of the processing context.

2.7.2 Predefined Types

In addition to the 19 types defined in Section 3.2 Primitive datatypes^{XS2} of [Schema Part 2], the data model defines five additional types: `xs:anyAtomicType`, `xs:untyped`, `xs:untypedAtomic`, `xs:dayTimeDuration`, and `xs:yearMonthDuration`. These types are defined in the XML Schema namespace with permission of the XML Schema Working Group; in implementations that support [Schema 1.1 Part 2], the XSD 1.1 definitions of `xs:anyAtomicType`, `xs:dayTimeDuration`, and `xs:yearMonthDuration` supersede the definitions in this specification.

`xs:untyped`
The datatype **xs:untyped** denotes the dynamic type of an element node that has not been validated, or has been validated in skip mode. The properties of `xs:untyped` are the same as the properties of `xs:anyType` except for the base type and name. The base type of `xs:untyped` is `xs:anyType`. No predefined types are derived from `xs:untyped` and no such derivations are allowed.

`xs:untypedAtomic`
The datatype **xs:untypedAtomic** denotes untyped atomic data, such as text that has not been assigned a more specific type. An attribute that has been validated in skip mode is represented in the Data Model by an attribute node with the type `xs:untypedAtomic`. No predefined types are derived from `xs:untypedAtomic` and no such derivations are allowed.

`xs:anyAtomicType`
The datatype **xs:anyAtomicType** is an atomic type that includes all atomic values (and no values that are not atomic). Its base type is `xs:anySimpleType` from which all simple types, including atomic, list, and union types are derived. All primitive atomic types, such as `xs:decimal` and `xs:string`, have `xs:anyAtomicType` as their base type.

No type may be derived from `xs:anyAtomicType` by restriction, union, or list.

`xs:dayTimeDuration`
The type `xs:dayTimeDuration` is derived from `xs:duration` by restricting its lexical representation to contain only the days, hours, minutes and seconds components. The value space of `xs:dayTimeDuration` is the set of fractional

second values. The components of `xs:dayTimeDuration` correspond to the day, hour, minute and second components defined in Section 5.5.3.2 of [ISO 8601]. An `xs:dayTimeDuration` is derived from `xs:duration` as follows:

```
<xs:simpleType name='dayTimeDuration'>
  <xs:restriction base='xs:duration'>
    <xs:pattern value="[^YM]*[DT].*"/>
  </xs:restriction>
</xs:simpleType>
```

`xs:yearMonthDuration`

The type `xs:yearMonthDuration` is derived from `xs:duration` by restricting its lexical representation to contain only the year and month components. The value space of `xs:yearMonthDuration` is the set of `xs:integer` month values. The year and month components of `xs:yearMonthDuration` correspond to the Gregorian year and month components defined in section 5.5.3.2 of [ISO 8601], respectively.

The type `xs:yearMonthDuration` is derived from `xs:duration` as follows:

```
<xs:simpleType name='yearMonthDuration'>
  <xs:restriction base='xs:duration'>
    <xs:pattern value="[^DT]*"/>
  </xs:restriction>
</xs:simpleType>
```

A schema for `xs:dayTimeDuration` and `xs:yearMonthDuration` is provided in **C Schema for the Extended XS Namespace**.

2.7.3 XML and XSD Versions

Some of the types defined in XML Schema have differing definitions in XSD 1.0 and XSD 1.1; furthermore, some types are defined by reference to other specifications including XML and XML Namespaces, and these too may vary from one version of the specification to the next.

As a general policy, implementations of data types **should** support the latest definitive version of any referenced specification, even if that is published after the date of this specification.

This means, for example, that the xs:string data type **should** (at the time of writing) support the set of characters defined by the Char production in XML 1.1 Second Edition. Similarly, the xs:anyURI data type **should** support the definition used in XSD 1.1 (which allows any sequence of characters), and the xs:NCName data type **should** support the definition based on the syntax of a name as defined in both XML 1.1 Second Edition and XML 1.0 Fifth Edition (which are the same).

In practice interoperability problems can arise both because specifications are not always in synchronization with each other (for example, XSD 1.0 contains referenced to dated versions of XML 1.0 other than the latest version), and also because implementations **may** use third-party components (such as XML parsers, serializers, and schema validators) that were built against different versions of the base specifications. For these reasons, use of the latest version of referenced specifications is generally *recommended* but not *required*. It is implementation-dependent how a processor handles any such conflicts.

[Definition: A **string** is a sequence of zero or more characters, or equivalently, a value in the value space of the `xs:string` data type.]

[Definition: A **character** is an instance of the `Char` production in [XML] .]

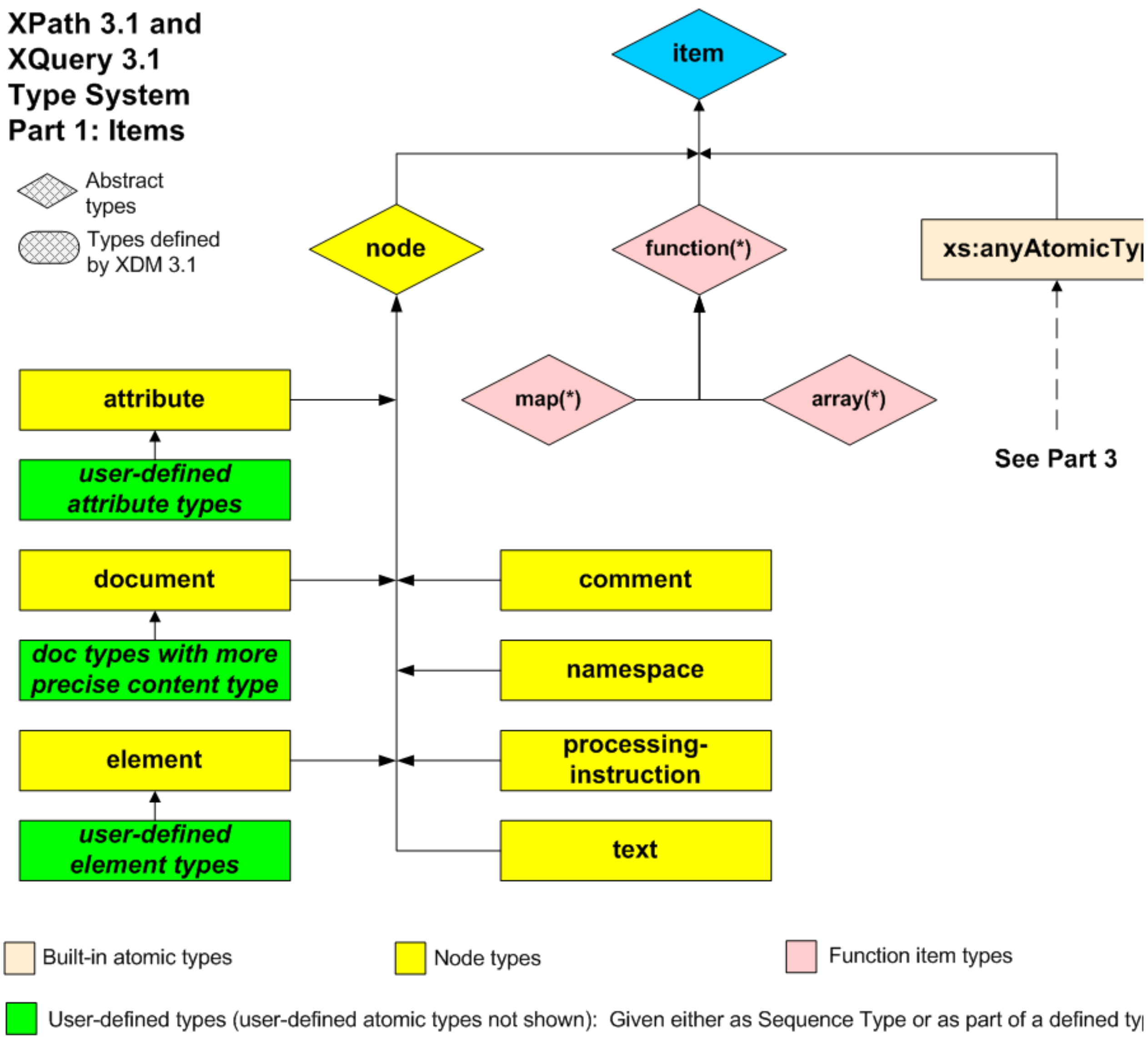
[Definition: A **codepoint** is a non-negative integer assigned to a character by the Unicode consortium, or reserved for future assignment to a character.]

2.7.4 Type System

The diagrams below show how nodes, functions, primitive simple types, and user defined types fit together into a type system. This type system comprises two distinct subsystems that both include the primitive simple types. In the diagrams, connecting lines represent relationships between derived types and the types from which they are derived; the arrowheads point toward the type from which they are derived. The dashed line represents relationships not present in this diagram, but that appear in one of the other diagrams. Dotted lines represent additional relationships that follow an evident pattern. The information that appears in each diagram is recapitulated in tabular form.

The `xs:IDREFS`, `xs:NMTOKENS`, `xs:ENTITIES` types, and `xs:numeric` and both the `user-defined list types` and `user-defined union types` are special types in that these types are lists or unions rather than types derived by extension or restriction.

The first diagram and its corresponding table illustrate the relationship of various item types. Item types in the data model form a directed graph, rather than a hierarchy or lattice: in the relationship defined by the `derived-from(A, B)` function, some types are derived from more than one other type. Examples include functions (`function(xs:string) as xs:int` is substitutable for `function(xs:NCName) as xs:int` and also for `function(xs:string) as xs:decimal`), and union types (`A` is substitutable for `union(A, B)` and also for `union(A, C)`. In XDM, item types include node types, function types, and built-in atomic types. The diagram, which shows only hierarchic relationships, is therefore a simplification of the full model.

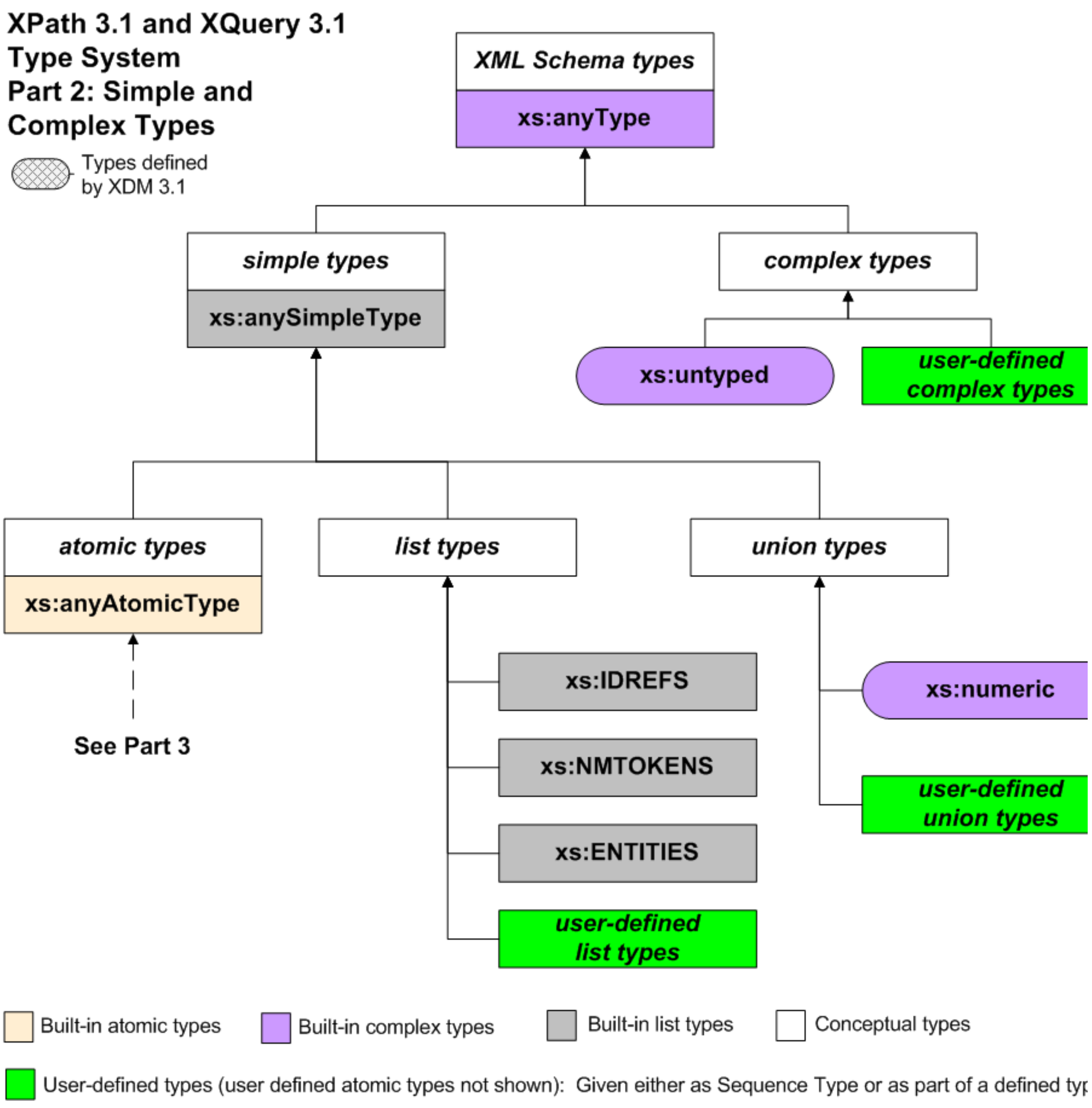

XPath 3.1 and XQuery 3.1 Type System Part 1: Items
Abstract types
Types defined by XDM 3.1
item
node
function(*)
xs:anyAtomicTy
map(*)
array(*)
See Part 3
attribute
user-defined attribute types
document
doc types with more precise content type
element
user-defined element types
comment
namespace
processing-instruction
text
Built-in atomic types
Node types
Function item types
User-defined types (user-defined atomic types not shown): Given either as Sequence Type or as part of a defined ty

In the table, each type whose name is indented is derived from the type whose name appears nearest above it with one less level of indentation.

item			
	xs:anyAtomicType		
	node		
		attribute	
			user-defined attribute types
		comment	
		document	
			user-defined document types
		element	
			user-defined element types
		namespace	
		processing-instruction	
		text	
	function(*)		

		array(*)
		map(*)

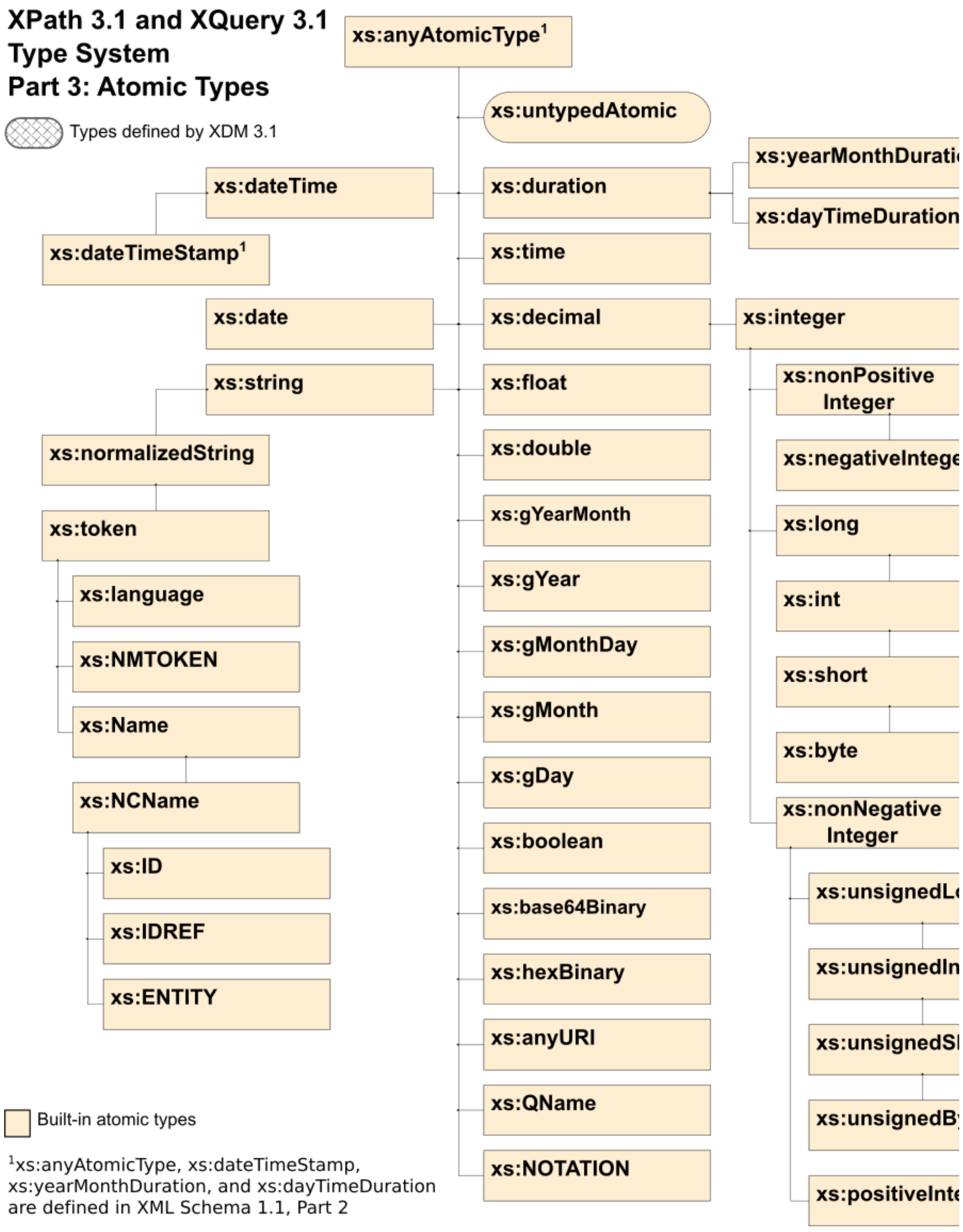
The next diagram and table illustrate the "any type" type subsystem, in which all types are derived from distinguished type `xs:anyType`.



In the table, each type whose name is indented is derived from the type whose name appears nearest above it with one less level of indentation.

xs:anyType			
	xs:anySimpleType		
		xs:anyAtomicType	
		list types	
			xs:IDREFS
			xs:NMTOKENS
			xs:ENTITIES
			user-defined list types
		union types	
			xs:numeric
			user-defined union types
	complex types		
		xs:untyped	
		user-defined complex types	

The final diagram and table show all of the atomic types, including the primitive simple types and the built-in types derived from the primitive simple types. This includes all the built-in datatypes defined in [Schema Part 2].



In the table, each type whose name is indented is derived from the type whose name appears nearest above it with one less level of indentation.

xs:untypedAtomic	
xs:dateTime	
	xs:dateTimeStamp
xs:date	
xs:time	
xs:duration	
	xs:yearMonthDuration
	xs:dayTimeDuration
xs:float	
xs:double	
xs:decimal	

	xs:integer					
		xs:nonPositiveInteger				
			xs:negativeInteger			
		xs:long				
			xs:int			
				xs:short		
					xs:byte	
		xs:nonNegativeInteger				
			xs:unsignedLong			
				xs:unsignedInt		
					xs:unsignedShort	
						xs:unsignedByte
			xs:positiveInteger			
xs:gYearMonth						
xs:gYear						
xs:gMonthDay						
xs:gDay						
xs:gMonth						
xs:string						
	xs:normalizedString					
		xs:token				
			xs:language			
			xs:NMTOKEN			
			xs:Name			
				xs:NCName		
					xs:ID	
					xs:IDREF	
					xs:ENTITY	
xs:boolean						
xs:base64Binary						
xs:hexBinary						
xs:anyURI						
xs:QName						
xs:NOTATION						

2.7.5 Atomic Values

An atomic value can be constructed from a lexical representation. Given a string and an atomic type, the atomic value is constructed in such a way as to be consistent with schema validation. If the string does not represent a valid value of the type, an error is raised. When `xs:untypedAtomic` is specified as the type, no validation takes place. The details of the construction are described in Section 18 Constructor functions *FO31* and the related Section 19 Casting *FO31* section of [XQuery and XPath Functions and Operators 3.1].

2.7.6 String Values

A string value can be constructed from an atomic value. Such a value is constructed by converting the atomic value to its string representation as described in Section 19 Casting *FO31* . Using the canonical lexical representation for atomic values is not always compatible with XPath 1.0. These and other backwards incompatibilities are described in [TITLE OF XP31 SPEC, TITLE OF id-backwards-compatibility SECTION]*XP31* .

2.7.7 Negative Zero

The `xs:float` and `xs:double` data types in the data model have the same value space as in XML Schema 1.1 ([Schema 1.1 Part 2]). Specifically they include both negative and positive zero, and in this respect they differ from XML Schema 1.0.

To accommodate this difference, when converting from an `xs:string` to an `xs:float` or `xs:double`, it is implementation-defined whether the lexical value "-0" (and similar forms such as "-0.0") convert to negative zero or to positive zero in the value space.

2.8 Other Items

The XPath Data Model is the abstraction over which XPath expressions are evaluated. Historically, all of the items in the data model could be derived directly (nodes) or indirectly (typed values, sequences) from an XML document. However, as the XPath expression language has matured, new features have been added which require additional types of items to appear in the data model. These items have no direct XML serialization, but they are never the less part of the data model.

2.8.1 Functions

[Definition: A **function** is an item that can be **called**.] Functions cannot be compared for identity, equality, or otherwise, and have no serialization.

A function has the following properties:

- **name** (`xs:QName`): An expanded QName, possibly absent.
- **parameter names** (`xs:QName*`): A list of distinct names, one for each of the function's parameters.
- **signature** (a `FunctionTest` of the form `Annotation* TypedFunctionTest`): The TypedFunctionTest*XP31* has one SequenceType*XP31* for each parameter, and one SequenceType for the function's result. [Definition: A **function signature** represents the type of a function.] The presence of annotations is language dependent; functions defined in languages, such as XPath, that have no mechanism for defining annotations will create functions in the data model with zero annotations.
- **implementation** This enables the function, when it's called, to map instances of its parameter types into an instance of its result type. The implementation is either:
  - a host language expression, which is always associated with a static context, or
  - an implementation-dependent function implementation, which is optionally associated with both a static and a dynamic context.
- **nonlocal variable bindings** (a mapping from `xs:QName` to `item()*`): This provides a value for each of the function's free variables (i.e., variables referenced by the function's implementation, other than locals and parameters).

[Definition: A function's **arity** is the number of its parameters.] The number of names in a function's parameter names, and the number of parameter types in its signature, must equal the function's arity.

The space of all possible function signatures forms a hierarchy of function types. All function types are a subtype of `function(*)`, which is itself a subtype of `item()`. Subtypes of `function(*)` are partitioned into discrete types, each representing functions that accept a particular number of arguments. Function types which have the same arity can be subtypes of each other based on their argument and return types. This subtype relationship is defined in Section 2.5.6 SequenceType Subtype Relationships *XP31* . For example:

- `function(item()) as item()` is a subtype of `function(*)`
- `function(item()) as xs:integer` is a subtype of `function(item()) as item()`
- `function(item()) as item()` is a subtype of `function(xs:string) as item()`

2.8.2 Map Items

[Definition: A **map item** is a value that represents a map (sometimes called a hash or an associative array).] A map is logically a collection of key/value pairs. Each key in the map is unique (there is no other key to which it is equal) and has associated with it a value that is a single item or sequence of items. There is no uniqueness constraint on values. The semantics of equality are described in Section 17.1.1 op:same-key *FO31*.

> **Note:**
>
> This version of the XPath Data Model does not specify whether or not maps have any identity other than their values. No operations currently defined on maps depend on the notion of map identity. Other specifications, for example, the *XQuery Update Facility*, may make identity of maps explicit.

There is a single accessor associated with maps, it is defined in the following section.

2.8.2.1 `map-entries` Accessor

```
dm:map-entries($map as map()) as array(array(item()))
```

The *dm:*`map-entries` accessor returns an array of arrays. For each key/value pair in the `$map`, an array will be constructed with the key in position 1 and the value in position 2. The array returned by *dm:*`map-entries` is the array of the arrays constructed for the key/value pairs. The order of the members in the array returned is implementation-dependent.

2.8.3 Array Items

[Definition: An **array item** is a value that represents an array.] An array is an ordered list of values; these values are called the members of the array. Unlike sequences, a member of an array can be any value (including a sequence or an array). The number of members in an array is called its size, and they are referenced by their position, in the range 1 to the size of the array.

Note:

This version of the XPath Data Model does not specify whether or not arrays have any identity other than their values. No operations currently defined on arrays depend on the notion of array identity. Other specifications, for example, the XQuery Update Facility, may make identity of arrays explicit.

The accessors associated with arrays are defined in the following sections.

2.8.3.1 `array-size` *Accessor*

```
dm:array-size($array as array()) as xs:nonNegativeInteger
```

The *dm:*`array-size` accessor returns the number of items in the array.

2.8.3.2 `array-get` *Accessor*

```
dm:array-get($array as array(), $position as xs:positiveInteger) as item()*
```

The *dm:*`array-get` accessor returns the value in the array at the position `$position`. An error is raised if the array does not contain a value at that position. For all positions greater than 0 and less than or equal to the array size, *dm:*`array-get` will return a value.

3 Data Model Construction

This section describes the constraints on instances of the data model.

The data model supports well-formed XML documents conforming to [Namespaces in XML] or [Namespaces in XML 1.1]. Documents that are not well-formed are, by definition, not XML. XML documents that do not conform to [Namespaces in XML] or [Namespaces in XML 1.1] are not supported (nor are they supported by [Infoset]).

In other words, the data model supports the following classes of XML documents:

- Well-formed documents conforming to [Namespaces in XML] or [Namespaces in XML 1.1].
- DTD-valid documents conforming to [Namespaces in XML] or [Namespaces in XML 1.1], and
- W3C XML Schema-validated documents.

This document describes how to construct an instance of the data model from an infoset ([Infoset]) or a Post Schema Validation Infoset (PSVI), the augmented infoset produced by an XML Schema validation episode.

An instance of the data model can also be constructed directly through application APIs, or from non-XML sources such as relational tables in a database. Data model construction from sources other than an Infoset or PSVI is implementation-defined. Regardless of how an instance of the data model is constructed, every node and atomic value in the data model must have a typed-value that is consistent with its type.

The data model supports some kinds of values that are not supported by [Infoset]. Examples of these are document fragments and sequences of Document Nodes. The data model also supports values that are not nodes. Examples of these are sequences of atomic values, or sequences mixing nodes and atomic values. These are necessary to be able to represent the results of intermediate expressions in the data model during expression processing.

3.1 Direct Construction

Although this document describes construction of an instance of the data model in terms of infoset properties, an infoset is not a necessary precondition for building an instance of the data model.

There are no constraints on how an instance of the data model may be constructed directly, save that the resulting instance **must** satisfy all of the constraints described in this document.

3.2 Construction from an Infoset

An instance of the data model can be constructed from an infoset that satisfies the following general constraints:

- All general and external parsed entities must be fully expanded. The Infoset must not contain any **unexpanded entity reference information items**.
- The infoset **must** provide all of the properties identified as "required" in this document. The properties identified as "optional" may be used, if they are present. All other properties are ignored.

An instance of the data model constructed from an information set **must** be consistent with the description provided for each node kind.

Furthermore, construction of an instance of the data model from an Infoset is only guaranteed to be well-defined for Infosets that could have been derived from a conforming XML document.

3.3 Construction from a PSVI

An instance of the data model can be constructed from a PSVI, whose element and attribute information items have been strictly assessed, laxly assessed, or have not been assessed. Constructing an instance of the data model from a PSVI **must** be consistent with the description provided in this section and with the description provided for each node kind.

Data model construction requires that the PSVI provide unique names for all anonymous schema types.

> **Note:**
>
> [Schema Part 1] does not require all schema processors to provide unique names for anonymous schema types. In order to build an instance of the data model from a PSVI produced by a processor that does not provide the names, some post-processing will be required in order to assure that they are all uniquely identified before construction begins.

[Definition: An **incompletely validated** document is an XML document that has a corresponding schema but whose schema-validity assessment has resulted in one or more element or attribute information items being assigned values other than 'valid' for the **[validity]** property in the PSVI.]

The data model supports incompletely validated documents. Elements and attributes that are not valid are treated as having unknown types.

The most significant difference between Infoset construction and PSVI construction occurs in the area of schema type assignment. Other differences can also arise from schema processing: default attribute and element values may be provided, white space normalization of element content may occur, and the user-supplied lexical form of elements and attributes with atomic schema types may be lost.

3.3.1 Mapping PSVI Additions to Node Properties

A PSVI element or attribute information item may have a **[validity]** property. The **[validity]** property may be " *valid* ", " *invalid* ", or " *notKnown* " and reflects the outcome of schema-validity assessment. In the data model, precise schema type information is exposed for Element and Attribute Nodes that are " *valid* ". Nodes that are not " *valid* " are treated as if they were simply well-formed XML and only very general schema type information is associated with them.

3.3.1.1 Element and Attribute Node Types

The precise definition of the schema type of an element or attribute information item depends on the properties of the PSVI. In the PSVI, [Schema Part 1] defines a **[type definition]** property as well as the **[type definition namespace]**, **[type definition name]** and **[type definition anonymous]** properties, which are effectively short-cut terms for properties of the type definition. Further, the **[element declaration]** and **[attribute declaration]** properties are defined for elements and attributes, respectively. These declarations in turn will identify the **[type definition]** declared for the element or attribute. To distinguish the **[type definition]** given in the PSVI for the element or attribute instance from the **[type definition]** associated with the declaration, the former is referred to below as the actual type and the latter as the declared type of the element or attribute instance in question.

The type depends on the declared type, the actual type, and the **[validity]** and **[validation attempted]** properties in the PSVI. If:

- The **[validity]** and **[validation attempted]** properties exist and have the values " *valid* " and " *full* ", respectively, the schema type of an element or attribute information item is represented by an expanded-QName whose namespace and local name correspond to the first applicable items in the following list:
  - If the declared type exists and is a union and the actual type is (not the same as the declared type, and not a type derived from the declared type, but) one of the member types of the union, or derived from one of its member types:
    - If the {name} property of the declared type is present: the {target namespace} and {name} properties of the declared type.
    - If the {name} property of the declared type is absent: the namespace and local name of the anonymous type name supplied for the declared type.
  - If there is no declared type, and the actual type is a union, then:
    - If the {name} property of the actual type is present: the {target namespace} and {name} properties of the actual type.
    - If the {name} property of the actual type is absent: the namespace and local name of the anonymous type name supplied for the actual type.
  - Otherwise:
    - If **[type definition anonymous]** is false: the {target namespace} and {name} properties of the actual type.
    - If **[type definition anonymous]** is true: the namespace and local name of the anonymous type name supplied for the actual type.
- The **[validity]** property exists and is " *invalid* ", or the **[validation attempted]** property exists and is " *partial* ", the schema type of an element is `xs:anyType` and the type of an attribute is `xs:anySimpleType`.
- The **[validity]** property exists and is " *notKnown* ", the schema type of an element is `xs:anyType` and the type of an attribute is `xs:anySimpleType`.
- The **[validity]** or **[validation attempted]** properties do not exist, the schema type of an element is `xs:untyped` and the type of an attribute is `xs:untypedAtomic`.

The prefix associated with the type names is implementation-dependent.

3.3.1.2 Typed Value Determination

This section describes how the typed value of an Element or Attribute Node is computed from an element or attribute PSVI information item, where the information item has either a simple type or a complex type with simple content. For other kinds of Element Nodes, see **6.2.4 Construction from a PSVI**; for other kinds of Attribute Nodes, see **6.3.4 Construction from a PSVI**.

The typed value of Attribute Nodes and some Element Nodes is a sequence of atomic values. The types of the items in the typed value of a node may differ from the type of the node itself. This section describes how the typed value of a node is derived from the properties of an information item in a PSVI.

The types of the items in the typed value of a node are determined as follows. The process begins with a type, `T`. If the schema type of the node itself, as represented in the PSVI, is a complex type with simple content, then `T` is the {content type} of the schema type of the node; otherwise, `T` is the schema type of the node itself. For each primitive or ordinary simple type `T`, the W3C XML Schema specification defines a function `M` mapping the lexical representation of a value onto the value itself.

Note:

For atomic and list types, the mapping is the “lexical mapping” defined for `T` in [Schema Part 2]; for union types, the mapping is the lexical mapping defined in [Schema Part 2] modified as appropriate by any applicable rules in [Schema Part 1]. The mapping, so modified, is a function (in the mathematical sense) which maps to a single value even in cases where the lexical mapping proper maps to multiple values.

The typed value is determined as follows:

- If the **nilled** property of the node in question is `true`, then the typed value is the empty sequence.
- If `T` is `xs:anySimpleType` or `xs:anyAtomicType`, the typed value is the **[schema normalized value]** as an instance of `xs:untypedAtomic`.
- Otherwise, the typed value is the result of applying `M` to the string value as an instance of the appropriate value type, where the appropriate value type is the **[member type definition]** if `T` is a union type, otherwise it is simply `T`.

The typed value determination process is guaranteed to result in a sequence of atomic values, each having a well-defined atomic type. This sequence of atomic values, in turn, determines the typed-value property of the node in the data model.

3.3.1.3 Relationship Between Typed-Value and String-Value

Element and attribute nodes have both typed-value and string-value properties. However, implementations are allowed some flexibility in how these properties are stored. An implementation may choose to store the string-value only and derive the typed-value from it, or to store the typed-value only and derive the string-value from it, or to store both the string-value and the typed-value.

In order to permit these various implementation strategies, some variations in the string value of a node are defined as insignificant. Implementations that store only the typed value of a node are permitted to return a string value that is different from the original lexical form of the node content. For example, consider the following element:

```
<offset xsi:type="xs:integer">0030</offset>
```

Assuming that the node is valid, it has a typed value of 30 as an `xs:integer`. An implementation may return either "30" or "0030" as the string value of the node. Any string that is a valid lexical representation of the typed value is acceptable. In this specification, we express this rule by saying that the relationship between the string value of a node and its typed value must be "consistent with schema validation."

If an implementation stores only the string-value of a node, the following considerations apply:

- Where union types occur, the implementation must be able to deliver the typed-value as an instance of the appropriate member type. For example, if the type of an element node is my:integer-or-string, which is defined as a union of xs:integer and xs:string, and the string-value of the node is "47", the implementation must be able to deliver the typed-value of the node as either the integer 47 or the string "47", depending on which member type validated the element.
- Where types of `xs:QName`, `xs:NOTATION`, or types derived from one of these types occur, the implementation must be able to deliver the typed-value as a triple including a local name, a namespace prefix, and a namespace URI, even though the namespace URI is not part of the string-value (see **3.3.3 QNames and NOTATIONS**).
- Where an element with a complex type and element-only content occurs, it is an error to attempt to access the typed-value of the Element Node.

If an implementation stores only the typed-value of a node, it must be prepared to construct string values from not only the node, but in some cases also the descendants of that node. For example, an element with a complex type and element-only content has no typed-value but does have a string-value that is the concatenation of the string-values of all its Text Node descendants in document order.

A further caveat applies if an implementation stores the typed value of a node. If a new data model is constructed by copying portions of another data model, and the copy operation does not preserve inherited namespaces, and the type is a union type that is sensitive to the namespace context, then the typed value may be different than what would be obtained by revalidating the node within its new namespace context. Although this may stretch the semantics of “consistent with schema validation”, we accept this possibility; it is not an error.

3.3.1.4 Pattern Facets

Creating a subtype by restriction generally reduces the *value* space of the original schema type. For example, expressing a hat size as a restriction of decimal with a minimum value of 6.5 and maximum value of 8.0 creates a schema type whose valid values are only those in the range 6.5 to 8.0.

The pattern facet is different because it restricts the *lexical* space of the schema type, not its value space. Expressing a three-digit number as a restriction of integer with the pattern facet “[0-9]{3}” creates a schema type whose valid values are only those with a lexical form consisting of three digits.

The pattern facet is not reversible in practice. A given point in the value space might have several lexical representations. In general, there's no practical way to determine which, if any, of these representations satisfies the pattern facet of the type.

As a consequence, pattern facets are not respected when mapping to an Infoset or during serialization and values in the data model that were originally valid with respect to a schema that contains pattern-based restrictions may be invalid after serialization.

3.3.2 Dates and Times

The date and time types require special attention. This section applies to implementations that store the typed value of `xs:dateTime`, `xs:date`, `xs:time`, `xs:gYearMonth`, `xs:gYear`, `xs:gMonthDay`, `xs:gMonth`, `xs:gDay`, and types that are derived from them. These are known collectively as the date/time types in this specification.

The values of the date/time types are represented in the data model using seven components:

year
An `xs:integer`.

month
An `xs:integer` between 1 and 12, inclusive.

day
An `xs:integer` between 1 and 31, inclusive, possibly restricted further depending on the values of month and year.

hour
An `xs:integer` between 0 and 23, inclusive.

minute
An `xs:integer` between 0 and 59, inclusive.

second
An `xs:decimal` greater than or equal to zero and less than 60. Leap seconds are not supported.

timezone
An `xs:dayTimeDuration` between -PT14H00M and PT14H00M, inclusive. All timezone values must be an integral number of minutes.

Components that are intrinsic to the datatype (for example, day, month, and year in a `xs:date`) are required; components that can never be part of a datatype (for example, years in a `xs:time`) must be missing. Missing components are represented by the empty sequence. When a component is present, it contains the “local value” that has not been normalized in any way. The timezone component is optional for all the date/time datatypes.

Thus, the lexical `xs:dateTime` representation “`2003-01-02T11:30:00-05:00`” is stored as “`{2003, 1, 2, 11, 30, 0.0, -PT05H00M}`”. The value of the lexical representation “`2003-01-16T16:30:00`” is stored as “`{2003, 1, 16, 16, 30, 0, ()}`” because it has no timezone. The value of the lexical `xs:gDay` representation “`---30+10:30`” is stored as “`{(), (), 30, (), (), (), PT10H30M}`”.

The lexical form “`24:00:00`” is normalized in the component model. As a `xs:time`, it is stored as “`{(), (), (), 0, 0, 0.0, ()}`” and the `xs:dateTime` representation “`1999-12-31T24:00:00`” is stored as “`{2000, 1, 1, 0, 0, 0.0, ()}`”.

Note:

Implementations are permitted to store date/time values in any representation that's convenient for them, provided that the individual properties can be accessed and modified.

3.3.3 QNames and NOTATIONS

The `QName` and `NOTATION` data types require special attention. The following sections apply to `xs:QName`, `xs:NOTATION`, and types derived from them. These types are referred to collectively as “qualified names”.

As defined in XML Schema, the lexical space for qualified names includes a local name and an optional namespace prefix. The value space for qualified names contains a local name and an optional namespace URI. Therefore, it is not possible to derive a lexical value from the typed value, or vice versa, without access to some context that defines the namespace bindings.

When qualified names exist as values of nodes in a well-formed document, it is always possible to determine such a namespace context. However, the data model also allows qualified names to exist as freestanding atomic values, or as the name or value of a parentless attribute node, and in these cases no namespace context is available.

In this Data Model, therefore, the value space for qualified names contains a local-name, an optional namespace URI, and an optional prefix. The prefix is used only when producing a lexical representation of the value, that is, when casting the value to a string. The prefix plays no part in other operations involving qualified names: in particular, two qualified names are equal if their local names and namespace URIs match, regardless whether they have the same prefix.

The following consistency constraints apply:

- If the namespace URI of a qualified name is absent, then the prefix must also be absent.
- For every element node whose name has a prefix, the prefix must be one that has a binding to the namespace URI of the element name in the namespaces property of the element.
- For every element node whose name has no prefix, the element must have a a binding for the empty prefix to the namespace URI of the element name, or must have no binding for the empty prefix in the case where the name of the element has no namespace URI.
- For every attribute node whose name has a prefix, the attribute node must either be parentless, or the prefix must be one that has a binding to the namespace URI of the attribute name in the namespaces property of the parent element.
- For every qualified name that contains a prefix and that is included in the typed value of an element node, or of an attribute node that has an element node as its parent, the prefix must be one that is bound to the namespace URI of the qualified name in the namespaces property of that element.
- For every qualified name that contains a namespace URI and no prefix, and that is included in the typed value of an element node, or of an attribute node that has an element node as its parent, that element node must have a binding for the empty prefix to that namespace URI in its namespace property.
- For every qualified name that contains neither a namespace URI nor a prefix, and that is included in the typed value of an element node, or of an attribute node that has an element node as its parent, that node must not have a binding for the empty prefix.
- No qualified name that contains a prefix may be included in the typed value of an attribute node that has no parent.

4 Infoset Mapping

This specification describes how to map each kind of node to the corresponding information item. This mapping produces an Infoset; it does not and cannot produce a PSVI. Validation must be used to obtain a PSVI for a (portion of a) data model instance.

5 Accessors

A set of accessors is defined on nodes in the data model. For consistency, all the accessors are defined on every kind of node, although several accessors return a constant empty sequence on some kinds of nodes.

In order for processors to be able to operate on instances of the data model, the model must expose the properties of the items it contains. The data model does this by defining a family of accessor functions. These are not functions in the literal sense; they are not available for users or applications to call directly. Rather they are descriptions of the information that an implementation of the data model must expose to applications. Functions and operators available to end-users are described in [XQuery and XPath Functions and Operators 3.1].

Some typed values in the data model are absent. Attempting to access an absent typed value is an error. Behavior in these cases is implementation-defined and the host language is responsible for determining the result.

5.1 `attributes` Accessor

```
dm:attributes($n as node()) as attribute()*
```

The `dm:attributes` accessor returns the attributes of a node as a sequence containing zero or more Attribute Nodes. The order of Attribute Nodes is stable but implementation dependent.

It is defined on all seven node kinds.

5.2 `base-uri` Accessor

```
dm:base-uri($n as node()) as xs:anyURI?
```

The `dm:base-uri` accessor returns the base URI of a node as a sequence containing zero or one URI reference. For more information about base URIs, see [XML Base].

It is defined on all seven node kinds.

5.3 `children` Accessor

```
dm:children($n as node()) as node()*
```

The `dm:children` accessor returns the children of a node as a sequence containing zero or more nodes.

It is defined on all seven node kinds.

5.4 `document-uri` Accessor

```
dm:document-uri($node as node()) as xs:anyURI?
```

The `dm:document-uri` accessor returns the absolute URI of the resource from which the Document Node was constructed, if the absolute URI is available. If there is no URI available, or if it cannot be made absolute when the Document Node is constructed, or if it is used on a node other than a Document Node, the empty sequence is returned.

It is defined on all seven node kinds.

5.5 `is-id` Accessor

```
dm:is-id($node as node()) as xs:boolean?
```

The `dm:is-id` accessor returns true if the node is an XML ID. Exactly what constitutes an ID depends in part on how the data model was constructed, see **6.2 Element Nodes** and **6.3 Attribute Nodes**.

It is defined on all seven node kinds.

5.6 `is-idrefs` Accessor

```
dm:is-idrefs($node as node()) as xs:boolean?
```

The `dm:is-idrefs` accessor returns true if the node is an XML IDREF or IDREFS. Exactly what constitutes an IDREF or IDREFS depends in part on how the data model was constructed, see **6.2 Element Nodes** and **6.3 Attribute Nodes**.

It is defined on all seven node kinds.

5.7 `namespace-nodes` Accessor

```
dm:namespace-nodes($n as node()) as node()*
```

The `dm:namespace-nodes` accessor returns the dynamic, in-scope namespaces associated with a node as a sequence containing zero or more Namespace Nodes. The order of Namespace Nodes is stable but implementation dependent.

It is defined on all seven node kinds.

5.8 `nilled` Accessor

```
dm:nilled($n as node()) as xs:boolean?
```

The `dm:nilled` accessor returns true if the node is "nilled". [Schema Part 1] introduced the nilled mechanism to signal that an element should be accepted as valid when it has no content even when it has a content type which does not require or even necessarily allow empty content.

It is defined on all seven node kinds.

5.9 `node-kind` Accessor

```
dm:node-kind($n as node()) as xs:string
```

The `dm:node-kind` accessor returns a string identifying the kind of node. It will be one of the following, depending on the kind of node: “attribute”, “comment”, “document”, “element”, “namespace” “processing-instruction”, or “text”.

It is defined on all seven node kinds.

5.10 `node-name` Accessor

```
dm:node-name($n as node()) as xs:QName?
```

The `dm:node-name` accessor returns the name of the node as a sequence of zero or one `xs:QName`s. Note that the QName value includes an optional prefix as described in **3.3.3 QNames and NOTATIONS**.

It is defined on all seven node kinds.

5.11 `parent` Accessor

```
dm:parent($n as node()) as node()?
```

The `dm:parent` accessor returns the parent of a node as a sequence containing zero or one nodes.

It is defined on all seven node kinds.

5.12 `string-value` Accessor

```
dm:string-value($n as node()) as xs:string
```

The `dm:string-value` accessor returns the string value of a node.

It is defined on all seven node kinds.

5.13 `type-name` Accessor

```
dm:type-name($n as node()) as xs:QName?
```

The `dm:type-name` accessor returns the name of the schema type of a node as a sequence of zero or one `xs:QNames`.

It is defined on all seven node kinds.

5.14 `typed-value` Accessor

```
dm:typed-value($n as node()) as xs:anyAtomicType*
```

The `dm:typed-value` accessor returns the typed-value of the node as a sequence of zero or more atomic values.

It is defined on all seven node kinds.

5.15 `unparsed-entity-public-id` Accessor

```
dm:unparsed-entity-public-id($node       as node(),
                             $entityname as xs:string) as xs:string?
```

The `dm:unparsed-entity-public-id` accessor returns the public identifier of an unparsed external entity declared in the specified document. If no entity with the name specified in `$entityname` exists, or if the entity is not an external unparsed entity, or if the entity has no public identifier, the empty sequence is returned.

It is defined on all seven node kinds.

5.16 `unparsed-entity-system-id` Accessor

```
dm:unparsed-entity-system-id($node       as node(),
                             $entityname as xs:string) as xs:anyURI?
```

The `dm:unparsed-entity-system-id` accessor returns the system identifier of an unparsed external entity declared in the specified document. The value is an absolute URI, and is obtained by resolving the **[system identifier]** of the unparsed entity information item against the **[declaration base URI]** of the same item. If no entity with the name specified in `$entityname` exists, or if the entity is not an external unparsed entity, the empty sequence is returned.

It is defined on all seven node kinds.

6 Nodes

[Definition: There are seven kinds of **Nodes** in the data model: document, element, attribute, text, namespace, processing instruction, and comment.] Each kind of node is described in the following sections.

All nodes **must** satisfy the following general constraints:

1. Every node **must** have a unique identity, distinct from all other nodes.
2. The **children** property of a node **must not** contain two consecutive Text Nodes.
3. The **children** property of a node **must not** contain any empty Text Nodes.

4. No node may appear more than once in the **children** or **attributes** properties of a node.

6.1 Document Nodes

6.1.1 Overview

Document Nodes encapsulate XML documents. Documents have the following properties:

- **base-uri**, possibly empty.
- **children**, possibly empty.
- **unparsed-entities**, possibly empty.
- **document-uri**, possibly empty.
- **string-value**
- **typed-value**

Document Nodes **must** satisfy the following constraints.

1. The **children must** consist exclusively of Element, Processing Instruction, Comment, and Text Nodes if it is not empty. Attribute, Namespace, and Document Nodes can never appear as children
2. If a node *N* is among the **children** of a Document Node *D*, then the **parent** of *N* **must** be *D*.
3. If a node *N* has a **parent** Document Node *D*, then *N* **must** be among the **children** of *D*.
4. The **string-value** property of a Document Node must be the concatenation of the **string-value**s of all its Text Node descendants in document order or, if the document has no such descendants, the zero-length string.

In the [Infoset], a **document information item** must have at least one child, its children must consist exclusively of **element information item**s, **processing instruction information item**s and **comment information item**s, and exactly one of the children must be an **element information item**. This data model is more permissive: a Document Node may be empty, it may have more than one Element Node as a child, and it also permits Text Nodes as children.

Implementations that support DTD processing and access to the unparsed entity accessors use the **unparsed-entities** property to associate information about an unordered collection of unparsed entities with a Document Node. This property is accessed indirectly through the `dm:unparsed-entity-system-id` and `dm:unparsed-entity-public-id` functions. There is at most one unparsed entity associated with any given name. Conforming XML documents may include more than one unparsed entity declaration for the same name, but XML mandates that only the first such declaration is significant.

6.1.2 Accessors

`dm:attributes`
Returns the empty sequence

`dm:base-uri`
Returns the value of the **base-uri** property.

`dm:children`
Returns the value of the **children** property.

`dm:document-uri`
Returns the absolute URI of the resource from which the Document Node was constructed, or the empty sequence if no such absolute URI is available.

`dm:is-id`
Returns the empty sequence.

`dm:is-idrefs`
Returns the empty sequence.

`dm:namespace-nodes`
Returns the empty sequence

`dm:nilled`

Returns the empty sequence

`dm:node-kind`
: Returns “`document`”.

`dm:node-name`
: Returns the empty sequence.

`dm:parent`
: Returns the empty sequence

`dm:string-value`
: Returns the value of the **string-value** property.

`dm:type-name`
: Returns the empty sequence.

`dm:typed-value`
: Returns the value of the **typed-value** property.

`dm:unparsed-entity-public-id`
: Returns the public identifier of the specified unparsed entity or the empty sequence if no such entity exists.

`dm:unparsed-entity-system-id`
: Returns the system identifier of the specified unparsed entity or the empty sequence if no such entity exists.

6.1.3 Construction from an Infoset

The **document information item** is required. A Document Node is constructed for each **document information item**.

The following infoset properties are required: **[children]** and **[base URI]**.

The following infoset properties are optional: **[unparsed entities]**.

Document Node properties are derived from the infoset as follows:

base-uri

The value of the **[base URI]** property, if available. Note that the base URI property, if available, is always an absolute URI (if an absolute URI can be computed) though it may contain Unicode characters that are not allowed in URIs. These characters, if they occur, are present in the **base-uri** property and will have to be encoded and escaped by the application to obtain a URI suitable for retrieval, if retrieval is required.

In practice a **[base URI]** is not always known. In this case the value of the **base-uri** property of the document node will be the empty sequence. This is not intrinsically an error, though it may cause some operations that depend on the base URI to fail.

children

The sequence of nodes constructed from the information items found in the **[children]** property.

For each element, processing instruction, and comment found in the **[children]** property, a corresponding Element, Processing Instruction, or Comment Node is constructed and that sequence of nodes is used as the value of the **children** property.

If present among the **[children]**, the **document type declaration information item** is ignored.

unparsed-entities

If the **[unparsed entities]** property is present and is not the empty set, the values of the **unparsed entity information items** must be used to support the `dm:unparsed-entity-system-id` and `dm:unparsed-entity-public-id` accessors.

The internal structure of the values of the **unparsed-entities** property is implementation defined.

string-value

The concatenation of the string-values of all its Text Node descendants in document order. If the document has no such descendants, the zero-length string.

typed-value

The `dm:string-value` of the node as an `xs:untypedAtomic` value.

document-uri

The **document-uri** property holds the absolute URI for the resource from which the document node was constructed, if one is available and can be made absolute. For example, if a collection of documents is returned by the `fn:collection` function, the **document-uri** property may serve to distinguish between them even though each has the same **base-uri** property.

If the **document-uri** is not the empty sequence, then the following constraint must hold: the node returned by evaluating `fn:doc()` with the **document-uri** as its argument must return the document node that provided the value of the **document-uri** property.

In other words, for any Document Node `$arg`, either `fn:document-uri($arg)` must return the empty sequence or `fn:doc(fn:document-uri($arg))` must return `$arg`.

6.1.4 Construction from a PSVI

Construction from a PSVI is identical to construction from the Infoset.

6.1.5 Infoset Mapping

A Document Node maps to a **document information item**. The mapping fails and produces no value if the Document Node contains Text Node children that do not consist entirely of white space or if the Document Node contains more than one Element Node child.

The following properties are specified by this mapping:

[children]
A list of information items obtained by processing each of the *dm:*`children` in order and mapping each to the appropriate information item(s).

[document element]
The **element information item** that is among the **[children]**.

[unparsed entities]
An unordered set of **unparsed entity information item**s constructed from the **unparsed-entities**.

Each unparsed entity maps to an **unparsed entity information item**. The following properties are specified by this mapping:

[name]
The name of the entity.

[system identifier]
The system identifier of the entity.

[public identifier]
The public identifier of the entity.

[declaration base URI]
Implementation defined. In many cases, the **document-uri** is the correct answer and implementations **must** use this value if they have no better information. Implementations that keep track of the original **[declaration base URI]** for entities should use that value.

The following properties of the **unparsed entity information item** have no value: **[notation name]**, **[notation]**.

The following properties of the **document information item** have no value: **[notations] [character encoding scheme] [standalone] [version] [all declarations processed]**.

6.2 Element Nodes

6.2.1 Overview

Element Nodes encapsulate XML elements. Elements have the following properties:

- **base-uri**, possibly empty.

- **node-name**
- **parent**, possibly empty
- **schema-type**
- **children**, possibly empty
- **attributes**, possibly empty
- **namespaces**
- **nilled**
- **string-value**
- **typed-value**
- **is-id**
- **is-idrefs**

Element Nodes **must** satisfy the following constraints.

1. The **children must** consist exclusively of Element, Processing Instruction, Comment, and Text Nodes if it is not empty. Attribute, Namespace, and Document Nodes can never appear as children
2. The Attribute Nodes of an element **must** have distinct `xs:QNames`.
3. If a node *N* is among the **children** of an element *E*, then the **parent** of *N* **must** be *E*.
4. Exclusive of Attribute and Namespace Nodes, if a node *N* has a **parent** element *E*, then *N* **must** be among the **children** of *E*. (Attribute and Namespace Nodes have a parent, but they do not appear among the children of their parent.)

   The data model permits Element Nodes without parents (to represent partial results during expression processing, for example). Such Element Nodes **must not** appear among the **children** of any other node.

   An element may not be its own child or its own parent.
5. If an Attribute Node *A* is among the **attributes** of an element *E*, then the **parent** of *A* **must** be *E*.
6. If an Attribute Node *A* has a **parent** element *E*, then *A* **must** be among the **attributes** of *E*.

   The data model permits Attribute Nodes without parents. Such Attribute Nodes **must not** appear among the **attributes** of any Element Node.
7. If a Namespace Node *N* is among the namespaces of an element *E*, then the parent of *N* **must** be *E*.
8. If a Namespace Node *N* has a parent element *E*, then *N* **must** be among the **namespaces** of *E*.

   The data model permits Namespace Nodes without parents. Such Namespace Nodes **must not** appear among the **namespaces** of any Element Node. This constraint is irrelevant for implementations that do not support Namespace Nodes.
9. If the `dm:type-name` of an Element Node is `xs:untyped`, then the `dm:type-name` of all its descendant elements **must** also be `xs:untyped` and the `dm:type-name` of all its Attribute Nodes **must** be `xs:untypedAtomic`.
10. If the `dm:type-name` of an Element Node is `xs:untyped`, then the **nilled** property **must** be `false`.
11. If the **nilled** property is `true`, then the **children** property **must not** contain Element Nodes or Text Nodes.
12. For every expanded QName that appears in the `dm:node-name` of the element, the `dm:node-name` of any Attribute Node among the **attributes** of the element, or in any value of type `xs:QName` or `xs:NOTATION` (or any type derived from those types) that appears in the typed-value of the element or the typed-value of any of its attributes, if the expanded QName has a non-empty URI, then there **must** be a prefix binding for this URI among the **namespaces** of this Element Node.

    If any of the expanded QNames has an empty URI, then there **must not** be any binding among the **namespaces** of this Element Node which binds the empty prefix to a URI.
13. Every element must include a Namespace Node and/or namespace binding for the prefix `xml` bound to the URI `http://www.w3.org/XML/1998/namespace` and there must be no other prefix bound to that URI.
14. The **string-value** property of an Element Node must be the concatenation of the **string-value**s of all its Text Node descendants in document order or, if the element has no such descendants, the zero-length string.

6.2.2 Accessors

`dm:attributes`

Returns the value of the **attributes** property. The order of Attribute Nodes is stable but implementation dependent.

dm:`base-uri`
: Returns the value of the **base-uri** property.

dm:`children`
: Returns the value of the **children** property.

dm:`document-uri`
: Returns the empty sequence.

dm:`is-id`
: Returns the value of the **is-id** property.

dm:`is-idrefs`
: Returns the value of the **is-idrefs** property.

dm:`namespace-nodes`
: Returns the value of the **namespaces** property as a sequence of Namespace Nodes. The order of Namespace Nodes is stable but implementation dependent.

dm:`nilled`
: Returns the value of the **nilled** property.

dm:`node-kind`
: Returns “`element`”.

dm:`node-name`
: Returns the value of the **node-name** property.

dm:`parent`
: Returns the value of the **parent** property.

dm:`string-value`
: Returns the value of the **string-value** property.

dm:`type-name`
: Returns the value of the **schema-type** property.

dm:`typed-value`
: Returns the value of the **typed-value** property.

dm:`unparsed-entity-public-id`
: Returns the empty sequence.

dm:`unparsed-entity-system-id`
: Returns the empty sequence.

6.2.3 Construction from an Infoset

The **element information items** are required. An Element Node is constructed for each **element information item**.

The following infoset properties are required: **[namespace name]**, **[local name]**, **[children]**, **[attributes]**, **[in-scope namespaces]**, **[base URI]**, and **[parent]**.

Element Node properties are derived from the infoset as follows:

base-uri
: The value of the **[base URI]** property, if available. Note that the base URI property, if available, is always an absolute URI (if an absolute URI can be computed) though it may contain Unicode characters that are not allowed in URIs. These characters, if they occur, are present in the **base-uri** property and will have to be encoded and escaped by the application to obtain a URI suitable for retrieval, if retrieval is required.

  In practice a **[base URI]** is not always known. In this case the value of the **base-uri** property of the document node will be the empty sequence. This is not intrinsically an error, though it may cause some operations that depend on the base URI to fail.

node-name
: An `xs:QName` constructed from the **[prefix]**, **[local name]**, and **[namespace name]** properties.

parent
The node that corresponds to the value of the **[parent]** property or the empty sequence if there is no parent.

schema-type
All Element Nodes constructed from an infoset have the type `xs:untyped`.

children
The sequence of nodes constructed from the information items found in the **[children]** property.

For each element, processing instruction, comment, and maximal sequence of adjacent **character information items** found in the **[children]** property, a corresponding Element, Processing Instruction, Comment, or Text Node is constructed and that sequence of nodes is used as the value of the **children** property.

Because the data model requires that all general entities be expanded, there will never be **unexpanded entity reference information item** children.

attributes
A set of Attribute Nodes constructed from the **attribute information items** appearing in the **[attributes]** property. This includes all of the "special" attributes (`xml:lang`, `xml:space`, `xsi:type`, etc.) but does not include namespace declarations (because they are not attributes).

Default and fixed attributes provided by the DTD are added to the **[attributes]** and are therefore included in the data model **attributes** of an element.

namespaces
A set of Namespace Nodes constructed from the **namespace information items** appearing in the **[in-scope namespaces]** property. Implementations that do not support Namespace Nodes may simply preserve the relevant bindings in this property.

Implementations **may** ignore **namespace information items** for namespaces which are not known to be used. A namespace is known to be used if:

- It appears in the expanded QName of the **node-name** of the element.
- It appears in the expanded QName of the **node-name** of any of the element's attributes.

Note: applications may rely on namespaces that are not known to be used, for example when QNames are used in content and that content does not have a type of `xs:QName` Such applications may have difficulty processing data models where some namespaces have been ignored.

nilled
All Element Nodes constructed from an infoset have a **nilled** property of " *false* ".

string-value
The **string-value** is constructed from the **character information item [children]** of the element and all its descendants. The precise rules for selecting significant **character information items** and constructing characters from them is described in **6.7.3 Construction from an Infoset** of **6.7 Text Nodes**.

This process is equivalent to concatenating the *`dm:`*`string-value`s of all of the Text Node descendants of the resulting Element Node.

If the element has no such descendants, the **string-value** is the empty string.

typed-value
The **string-value** as an `xs:untypedAtomic`.

is-id
All Element Nodes constructed from an infoset have a **is-id** property of " *false* ".

is-idrefs
All Element Nodes constructed from an infoset have a **is-idrefs** property of " *false* ".

6.2.4 Construction from a PSVI

The following Element Node properties are affected by PSVI properties.

schema-type
The **schema-type** is determined as described in **3.3.1.1 Element and Attribute Node Types**.

children

The sequence of nodes constructed from the information items found in the **[children]** property.

For each element, processing instruction, comment, and maximal sequence of adjacent **character information items** found in the **[children]** property, a corresponding Element, Processing Instruction, Comment, or Text Node is constructed and that sequence of nodes is used as the value of the **children** property.

For elements with schema simple types, or complex types with simple content, if the **[schema normalized value]** PSVI property exists, the processor **may** use a sequence of nodes containing the Processing Instruction and Comment Nodes corresponding to the **processing instruction** and **comment information items** found in the **[children]** property, plus an optional single Text Node whose string value is the **[schema normalized value]** for the **children** property. If the **[schema normalized value]** is the empty string, the Text Node **must not** be present, otherwise it **must** be present.

The relative order of Processing Instruction and Comment Nodes must be preserved, but the position of the Text Node, if it is present, among them is implementation defined.

The effect of the above rules is that where a fixed or default value for an element is defined in the schema, and the element takes this default value, a text node will be created to contain the value, even though there are no character information items representing the value in the PSVI. The position of this text node relative to any comment or processing instruction children is implementation-dependent.

[Schema Part 1] also permits an element with mixed content to take a default or fixed value (which will always be a simple value), but it is unclear how such a defaulted value is represented in the PSVI. Implementations therefore **may** represent such a default value by creating a text node, but are not required to do so.

> **Note:**
>
> Section 3.3.1 in [Schema 1.1 Part 1] clarifies the PSVI contributions of element default or fixed values in mixed content: additional character information items are not added to the PSVI.

Because the data model requires that all general entities be expanded, there will never be **unexpanded entity reference information item** children.

attributes

A set of Attribute Nodes constructed from the **attribute information items** appearing in the **[attributes]** property. This includes all of the "special" attributes (`xml:lang`, `xml:space`, `xsi:type`, etc.) but does not include namespace declarations (because they are not attributes).

Default and fixed attributes provided by XML Schema processing are added to the **[attributes]** and are therefore included in the data model **attributes** of an element.

namespaces

A set of Namespace Nodes constructed from the **namespace information items** appearing in the **[in-scope namespaces]** property. Implementations that do not support Namespace Nodes may simply preserve the relevant bindings in this property.

Implementations **may** ignore **namespace information items** for namespaces which are not known to be used. A namespace is known to be used if:

- It appears in the expanded QName of the **node-name** of the element.
- It appears in the expanded QName of the **node-name** of any of the element's attributes.
- It appears in the expanded QName of any values of type `xs:QName` that appear among the element's children or the typed values of its attributes.

Note: applications may rely on namespaces that are not known to be used, for example when QNames are used in content and that content does not have a type of `xs:QName` Such applications may have difficulty processing data models where some namespaces have been ignored.

nilled

If the **[validity]** property exists on an information item and is " *valid* " then if the **[nil]** property exists and is true, then the **nilled** property is " *true* ". In all other cases, including all cases where schema validity assessment was not attempted or did not succeed, the **nilled** property is " *false* ".

string-value

The string-value is calculated as follows:

- If the element is empty: its string value is the zero length string.
- If the element has a type of `xs:untyped`, a complex type with element-only content, or a complex type with mixed content: its string-value is the concatenation of the **string-value**s of all its Text Node descendants in document order.
- If the element has a simple type or a complex type with simple content: its string-value is the **[schema normalized value]** of the node.

If an implementation stores only the typed value of an element, it may use any valid lexical representation of the typed value for the **string-value** property.

typed-value

The typed-value is calculated as follows:

- If the element is of type `xs:untyped`, its typed-value is its *dm:*`string-value` as an `xs:untypedAtomic`.
- If the element has a complex type with empty content, its typed-value is the empty sequence.
- If the element has a simple type or a complex type with simple content: its typed value is computed as described in **3.3.1.2 Typed Value Determination**. The result is a sequence of zero or more atomic values. The relationship between the type-name, typed-value, and string-value of an element node is consistent with XML Schema validation.

  Note that in the case of dates and times, the timezone is preserved as described in **3.3.2 Dates and Times**, and in the case of `xs:QName`s and `xs:NOTATION`s, the prefix is preserved as described in **3.3.3 QNames and NOTATIONS**.
- If the element has a complex type with mixed content (including `xs:anyType`), its typed-value is its *dm:*`string-value` as an `xs:untypedAtomic`.
- Otherwise, the element must be a complex type with element-only content. The typed-value of such an element is absent. Attempting to access this property with the *dm:*`typed-value` accessor always raises an error.

is-id

If the element has a complex type with element-only content, the **is-id** property is `false`. Otherwise, if the typed-value of the element consists of exactly one atomic value and that value is of type `xs:ID`, or a type derived from `xs:ID`, the **is-id** property is `true`, otherwise it is `false`.

> **Note:**
>
> This means that in the case of a type constructed by list from `xs:ID`, the ID is recognized provided that the list is of length one. A type constructed as a union involving `xs:ID` is recognized provided the actual value is of type `xs:ID`.

> **Note:**
>
> The element that is marked with the **is-id** property, and which will therefore be retrieved by the fn:id function, is the node whose typed value contains the `xs:ID` value. This node is a child of the element node that, according to XML Schema, is uniquely identified by this ID.

is-idrefs

If the element has a complex type with element-only content, the **is-idrefs** property is `false`. Otherwise, if any of the atomic values in the typed-value of the element is of type `xs:IDREF` or `xs:IDREFS`, or a type derived from one of those types, the **is-idrefs** property is `true`, otherwise it is `false`.

All other properties have values that are consistent with construction from an infoset.

6.2.5 Infoset Mapping

An Element Node maps to an **element information item**.

The following properties are specified by this mapping:

[namespace name]
The namespace name of the value of `dm:node-name`.

[local name]
The local part of the value of `dm:node-name`.

[prefix]
The prefix associated with the value of `dm:node-name`.

[children]
A list of information items obtained by processing each of the `dm:children` in order and mapping each to the appropriate information item(s).

[attributes]
An unordered set of information items obtained by processing each of the `dm:attributes` and mapping each to the appropriate information item(s).

[in-scope namespaces]
An unordered set of **namespace information items** constructed from the **namespaces**.

Each in-scope namespace maps to a **namespace information item**. The following properties are specified by this mapping:

[prefix]
The prefix associated with the namespace.

[namespace name]
The URI associated with the namespace.

[base URI]
The value of `dm:base-uri`.

[parent]

- If this node is the root of the infoset mapping operation, *unknown*.
- If this node has a parent, the information item that corresponds to the node returned by `dm:parent`.
- Otherwise *no value*.

The following property has no value: **[namespace attributes]**.

6.3 Attribute Nodes

6.3.1 Overview

Attribute Nodes represent XML attributes. Attributes have the following properties:

- **node-name**
- **parent**, possibly empty
- **schema-type**
- **string-value**
- **typed-value**
- **is-id**
- **is-idrefs**

Attribute Nodes **must** satisfy the following constraints.

1. If an Attribute Node *A* is among the **attributes** of an element *E*, then the **parent** of *A* **must** be *E*.
2. If a Attribute Node *A* has a parent element *E*, then *A* **must** be among the **attributes** of *E*.

   The data model permits Attribute Nodes without parents (to represent partial results during expression processing, for example). Such attributes **must not** appear among the **attributes** of any Element Node.
3. In the node-name of an attribute node, if a namespace URI is present then a prefix **must** also be present.

For convenience, the Element Node that owns this attribute is called its "parent" even though an Attribute Node is not a "child" of its parent element.

6.3.2 Accessors

dm:`attributes`
: Returns the empty sequence.

dm:`base-uri`
: If the attribute has a parent, returns the value of the *dm:*`base-uri` of its parent; otherwise it returns the empty sequence.

dm:`children`
: Returns the empty sequence.

dm:`document-uri`
: Returns the empty sequence.

dm:`is-id`
: Returns the value of the **is-id** property.

dm:`is-idrefs`
: Returns the value of the **is-idrefs** property.

dm:`namespace-nodes`
: Returns the empty sequence.

dm:`nilled`
: Returns the empty sequence.

dm:`node-kind`
: Returns “`attribute`”.

dm:`node-name`
: Returns the value of the **node-name** property.

dm:`parent`
: Returns the value of the **parent** property.

dm:`string-value`
: Returns the value of the **string-value** property.

dm:`type-name`
: Returns the value of the **schema-type** property.

dm:`typed-value`
: Returns the value of the **typed-value** property.

dm:`unparsed-entity-public-id`
: Returns the empty sequence.

dm:`unparsed-entity-system-id`
: Returns the empty sequence.

6.3.3 Construction from an Infoset

The **attribute information items** are required. An Attribute Node is constructed for each **attribute information item**.

The following infoset properties are required: **[namespace name]**, **[local name]**, **[normalized value]**, **[attribute type]**, and **[owner element]**.

Attribute Node properties are derived from the infoset as follows:

node-name
: An `xs:QName` constructed from the **[prefix]**, **[local name]**, and **[namespace name]** properties.

parent
: The Element Node that corresponds to the value of the **[owner element]** property or the empty sequence if there is no owner.

schema-type
The value `xs:untypedAtomic`.

string-value
The **[normalized value]** of the attribute.

typed-value
The attribute’s typed-value is its *dm:*`string-value` as an `xs:untypedAtomic`.

is-id
If the attribute is named `xml:id` and its **[attribute type]** property does not have the value `ID`, then [xml:id] processing is performed. This will assure that the value does have the type `ID` and that it is properly normalized. If an error is encountered during xml:id processing, an implementation **may** raise a dynamic error. The **is-id** property is always `true` for attributes named `xml:id`.

If the **[attribute type]** property has the value `ID`, `true`, otherwise `false`.

is-idrefs
If the **[attribute type]** property has the value `IDREF` or `IDREFS`, `true`, otherwise `false`.

6.3.4 Construction from a PSVI

The following Attribute Node properties are affected by PSVI properties.

string-value

- The **[schema normalized value]** PSVI property if that exists.
- Otherwise, the **[normalized value]** property.

If an implementation stores only the typed value of an attribute, it may use any valid lexical representation of the typed value for the **string-value** property.

schema-type
The **schema-type** is determined as described in **3.3.1.1 Element and Attribute Node Types**.

typed-value
The typed-value is calculated as follows:

- If the attribute is of type `xs:untypedAtomic`: its typed-value is its *dm:*`string-value` as an `xs:untypedAtomic`.
- Otherwise, a sequence of zero or more atomic values as described in **3.3.1.2 Typed Value Determination**. The relationship between the type-name, typed-value, and string-value of an attribute node is consistent with XML Schema validation.

is-id
If the attribute is named `xml:id` and its **[attribute type]** property does not have the value `xs:ID` or a type derived from `xs:ID`, then [xml:id] processing is performed. This will assure that the value does have the type `xs:ID` and that it is properly normalized. If an error is encountered during xml:id processing, an implementation **may** raise a dynamic error. The **is-id** property is always true for attributes named `xml:id`.

Otherwise, if the typed-value of the attribute consists of exactly one atomic value and that value is of type `xs:ID`, or a type derived from `xs:ID`, the **is-id** property is `true`, otherwise it is `false`.

> **Note:**
>
> This means that in the case of a type constructed by list from `xs:ID`, the ID is recognized provided that the list is of length one. A type constructed as a union involving `xs:ID` is recognized provided the actual value is of type `xs:ID`.

is-idrefs
If any of the atomic values in the typed-value of the attribute is of type `xs:IDREF` or `xs:IDREFS`, or a type derived from one of those types, the **is-idrefs** property is `true`, otherwise it is `false`.

Note:

This rule means that a node whose type is constructed by list with an item type of `xs:IDREF` (or a type derived from `xs:IDREF`) may have the **is-idrefs** property, whether or not the list type is named `xs:IDREFS` or is derived from `xs:IDREFS`. Because union types are allowed, it also means that an element or attribute with the **is-idrefs** property can contain atomic values of type `xs:IDREF` alongside values of other types. A node has the **is-idrefs** property only if the typed value contains at least one atomic value that is an instance of `xs:IDREF`; it is not sufficient that the type annotation permits such values.

All other properties have values that are consistent with construction from an infoset.

Note: attributes from the XML Schema instance namespace, " `http://www.w3.org/2001/XMLSchema-instance` ", (`xsi:schemaLocation`, `xsi:type`, etc.) appear as ordinary attributes in the data model.

6.3.5 Infoset Mapping

An Attribute Node maps to an **attribute information item**.

The following properties are specified by this mapping:

[namespace name]
The namespace name of the value of *dm:*`node-name`.

[local name]
The local part of the value of *dm:*`node-name`.

[prefix]
The prefix associated with the value of *dm:*`node-name`.

[normalized value]
The value of *dm:*`string-value`.

[owner element]

- If this node has a parent, the information item that corresponds to the node returned by *dm:*`parent`.
- Otherwise *no value*.

The following properties have no value: **[specified] [attribute type] [references]**.

6.4 Namespace Nodes

6.4.1 Overview

Each Namespace Node represents the binding of a namespace URI to a namespace prefix or to the default namespace. Implementations that do not use Namespace Nodes may represent the same information using the **namespaces** property of an element node. Namespaces have the following properties:

- **prefix**, possibly empty
- **uri**
- **parent**, possibly empty

Namespace Nodes **must** satisfy the following constraints.

1. If a Namespace Node *N* is among the namespaces of an element *E*, then the parent of *N* **must** be *E*.
2. If a Namespace Node *N* has a parent element *E*, then *N* **must** be among the namespaces of *E*.
3. A Namespace Node **must** not have the name `xmlns` nor the string-value `http://www.w3.org/2000/xmlns/`.

The data model permits Namespace Nodes without parents, see below.

In XPath 1.0, Namespace Nodes were directly accessible by applications, by means of the namespace axis. In XPath 3.1 the namespace axis is deprecated, and it is not available at all in XQuery 3.1. XPath 3.1 implementations are not required to expose the namespace axis, though they may do so if they wish to offer backwards compatibility.

The information held in namespace nodes is instead made available to applications using functions defined in [XQuery and XPath Functions and Operators 3.1]. Some properties of Namespace Nodes are not exposed by these functions: in particular, properties related to the identity of Namespace Nodes, their parentage, and their position in document order. Implementations that do not expose the namespace axis can therefore avoid the overhead of maintaining this information.

Implementations that expose the namespace axis **must** provide unique Namespace Nodes for each element. Each element has an associated set of Namespace Nodes, one for each distinct namespace prefix that is in scope for the element (including the `xml` prefix, which is implicitly declared by [Namespaces in XML] and one for the default namespace if one is in scope for the element. The element is the parent of each of these Namespace Nodes; however, a Namespace Node is not a child of its parent element. In implementations that expose the namespace axis, elements never share namespace nodes.

> **Note:**
>
> In implementations that do not expose the namespace axis, there is no means by which the host language can tell if namespace nodes are shared or not and, in such circumstances, sharing namespace nodes may be a very reasonable implementation strategy.

6.4.2 Accessors

dm:`attributes`
  Returns the empty sequence.

dm:`base-uri`
  Returns the empty sequence.

dm:`children`
  Returns the empty sequence.

dm:`document-uri`
  Returns the empty sequence.

dm:`is-id`
  Returns the empty sequence.

dm:`is-idrefs`
  Returns the empty sequence.

dm:`namespace-nodes`
  Returns the empty sequence.

dm:`nilled`
  Returns the empty sequence.

dm:`node-kind`
  Returns “`namespace`”.

dm:`node-name`
  If the **prefix** is not empty, returns an `xs:QName` with the value of the **prefix** property in the local-name and an empty namespace name, otherwise returns the empty sequence.

dm:`parent`
  Returns the value of the **parent** property.

dm:`string-value`
  Returns the value of the **uri** property.

dm:`type-name`
  Returns the empty sequence.

dm:`typed-value`
  Returns the value of the **uri** property as an `xs:string`.

dm:`unparsed-entity-public-id`
  Returns the empty sequence.

`dm:unparsed-entity-system-id`
Returns the empty sequence.

6.4.3 Construction from an Infoset

The **namespace information items** are required.

The following infoset properties are required: **[prefix]**, **[namespace name]**.

Namespace Node properties are derived from the infoset as follows:

prefix
The **[prefix]** property.

uri
The **[namespace name]** property.

parent
The element in whose **[in-scope namespaces]** property the **namespace information item** appears, if the implementation exposes any mechanism for accessing the `dm:parent` accessor of Namespace Nodes.

6.4.4 Construction from a PSVI

Construction from a PSVI is identical to construction from the Infoset.

6.4.5 Infoset Mapping

A Namespace Node maps to a **namespace information item**.

The following properties are specified by this mapping:

[prefix]
The prefix associated with the namespace.

[namespace name]
The value of `dm:string-value`.

6.5 Processing Instruction Nodes

6.5.1 Overview

Processing Instruction Nodes encapsulate XML processing instructions. Processing instructions have the following properties:

- **target**
- **content**
- **base-uri**, possibly empty
- **parent**, possibly empty

Processing Instruction Nodes **must** satisfy the following constraints.

1. The string "?>" **must not** occur within the **content**.
2. The **target must** be an `NCName`.

6.5.2 Accessors

`dm:attributes`
Returns the empty sequence.

`dm:base-uri`

Returns the value of the **base-uri** property.

dm:`children`
Returns the empty sequence.

dm:`document-uri`
Returns the empty sequence.

dm:`is-id`
Returns the empty sequence.

dm:`is-idrefs`
Returns the empty sequence.

dm:`namespace-nodes`
Returns the empty sequence.

dm:`nilled`
Returns the empty sequence.

dm:`node-kind`
Returns “`processing-instruction`”.

dm:`node-name`
Returns an `xs:QName` with the value of the **target** property in the local-name and an empty namespace URI and empty prefix.

dm:`parent`
Returns the value of the **parent** property.

dm:`string-value`
Returns the value of the **content** property.

dm:`type-name`
Returns the empty sequence.

dm:`typed-value`
Returns the value of the **content** property as a `xs:string`.

dm:`unparsed-entity-public-id`
Returns the empty sequence.

dm:`unparsed-entity-system-id`
Returns the empty sequence.

6.5.3 Construction from an Infoset

A Processing Instruction Node is constructed for each **processing instruction information item** that is not ignored.

The following infoset properties are required: **[target]**, **[content]**, **[base URI]**, and **[parent]**.

Processing Instruction Node properties are derived from the infoset as follows:

target
The value of the **[target]** property.

content
The value of the **[content]** property.

base-uri
The value of the **[base URI]** property, if available. Note that the base URI property, if available, is always an absolute URI (if an absolute URI can be computed) though it may contain Unicode characters that are not allowed in URIs. These characters, if they occur, are present in the **base-uri** property and will have to be encoded and escaped by the application to obtain a URI suitable for retrieval, if retrieval is required.

In practice a **[base URI]** is not always known. In this case the value of the **base-uri** property of the document node will be the empty sequence. This is not intrinsically an error, though it may cause some operations that depend on the base URI to fail.

parent

The node corresponding to the value of the **[parent]** property.

There are no Processing Instruction Nodes for processing instructions that are children of a **document type declaration information item**.

6.5.4 Construction from a PSVI

Construction from a PSVI is identical to construction from the Infoset.

6.5.5 Infoset Mapping

An Processing Instruction Node maps to a **processing instruction information item**.

The following properties are specified by this mapping:

[target]
The local part of the value of `dm:node-name`.

[content]
The value of `dm:string-value`.

[base URI]
The value of `dm:base-uri`.

[parent]

- If this node is the root of the infoset mapping operation, *unknown*.
- If this node has a parent, the information item that corresponds to the node returned by `dm:parent`.
- Otherwise *no value*.

[notation]
no value.

6.6 Comment Nodes

6.6.1 Overview

Comment Nodes encapsulate XML comments. Comments have the following properties:

- **content**
- **parent**, possibly empty

Comment Nodes **must** satisfy the following constraints.

1. The string "--" **must not** occur within the **content**.
2. The character "-" **must not** occur as the last character of the **content**.

6.6.2 Accessors

`dm:attributes`
Returns the empty sequence.

`dm:base-uri`
If the comment has a parent, returns the value of the `dm:base-uri` of its parent; otherwise, returns the empty sequence.

`dm:children`
Returns the empty sequence.

`dm:document-uri`
Returns the empty sequence.

`dm:is-id`

Returns the empty sequence.

dm:`is-idrefs`
Returns the empty sequence.

dm:`namespace-nodes`
Returns the empty sequence.

dm:`nilled`
Returns the empty sequence.

dm:`node-kind`
Returns “`comment`”.

dm:`node-name`
Returns the empty sequence.

dm:`parent`
Returns the value of the **parent** property.

dm:`string-value`
Returns the value of the **content** property.

dm:`type-name`
Returns the empty sequence.

dm:`typed-value`
Returns the value of the **content** property as a `xs:string`.

dm:`unparsed-entity-public-id`
Returns the empty sequence.

dm:`unparsed-entity-system-id`
Returns the empty sequence.

6.6.3 Construction from an Infoset

The **comment information items** are optional.

A Comment Node is constructed for each **comment information item**.

The following infoset properties are required: **[content]** and **[parent]**.

Comment Node properties are derived from the infoset as follows:

content
The value of the **[content]** property.

parent
The node corresponding to the value of the **[parent]** property.

There are no Comment Nodes for comments that are children of a **document type declaration information item**.

6.6.4 Construction from a PSVI

Construction from a PSVI is identical to construction from the Infoset.

6.6.5 Infoset Mapping

A Comment Node maps to a **comment information item**.

The following properties are specified by this mapping:

[content]
The value of the *dm:*`string-value`.

[parent]

- If this node is the root of the infoset mapping operation, *unknown*.
- If this node has a parent, the information item that corresponds to the node returned by `dm:parent`.
- Otherwise *no value*.

6.7 Text Nodes

6.7.1 Overview

Text Nodes encapsulate XML character content. Text has the following properties:

- **content**
- **parent**, possibly empty.

Text Nodes **must** satisfy the following constraint:

1. If the **parent** of a text node is not empty, the Text Node **must not** contain the zero-length string as its **content**.

In addition, Document and Element Nodes impose the constraint that two consecutive Text Nodes can never occur as adjacent siblings. When a Document or Element Node is constructed, Text Nodes that would be adjacent **must** be combined into a single Text Node. If the resulting Text Node is empty, it **must** never be placed among the children of its parent, it is simply discarded.

6.7.2 Accessors

`dm:attributes`
Returns the empty sequence.

`dm:base-uri`
If the Text Node has a parent, returns the value of the `dm:base-uri` of its parent; otherwise, returns the empty sequence.

`dm:children`
Returns the empty sequence.

`dm:document-uri`
Returns the empty sequence.

`dm:is-id`
Returns the empty sequence.

`dm:is-idrefs`
Returns the empty sequence.

`dm:namespace-nodes`
Returns the empty sequence.

`dm:nilled`
Returns the empty sequence.

`dm:node-kind`
Returns “`text`”.

`dm:node-name`
Returns the empty sequence.

`dm:parent`
Returns the value of the **parent** property.

`dm:string-value`
Returns the value of the **content** property.

`dm:type-name`
Returns `xs:untypedAtomic`.

`dm:typed-value`

Returns the value of the **content** property as an `xs:untypedAtomic`.

dm:`unparsed-entity-public-id`
Returns the empty sequence.

dm:`unparsed-entity-system-id`
Returns the empty sequence.

6.7.3 Construction from an Infoset

The **character information items** are required. A Text Node is constructed for each maximal sequence of **character information items** in document order.

The following infoset properties are required: **[character code]** and **[parent]**.

The following infoset properties are optional: **[element content whitespace]**.

A sequence of **character information items** is maximal if it satisfies the following constraints:

1. All of the information items in the sequence have the same parent.
2. The sequence consists of adjacent **character information items** uninterrupted by other types of information item.
3. No other such sequence exists that contains any of the same **character information items** and is longer.

Text Node properties are derived from the infoset as follows:

content
A string comprised of characters that correspond to the **[character code]** properties of each of the **character information items**.

If the resulting Text Node consists entirely of whitespace and the **[element content whitespace]** property of the **character information items** used to construct this node are `true`, the **content** of the Text Node is the zero-length string. Text Nodes are only allowed to be empty if they have no parents; an empty Text Node will be discarded when its parent is constructed, if it has a parent.

The content of the Text Node is not necessarily normalized as described in the [Character Model]. It is the responsibility of data producers to provide normalized text, and the responsibility of applications to make sure that operations do not de-normalize text.

parent
The node corresponding to the value of the **[parent]** property.

6.7.4 Construction from a PSVI

For Text Nodes constructed from the **[schema normalized value]** of elements, **content** contains the value of the **[schema normalized value]**.

Otherwise, construction from a PSVI is the same as construction from the Infoset except for the **content** property. When constructing the **content** property, **[element content whitespace]** is not used to test if whitespace is collapsed. Instead, if the resulting Text Node consists entirely of whitespace and the **character information items** used to construct this node have a parent and that parent is an element and its {content type} is not "`mixed`", then the **content** of the Text Node is the zero-length string.

Text Nodes are only allowed to be empty if they have no parents; an empty Text Node will be discarded when its parent is constructed, if it has a parent.

6.7.5 Infoset Mapping

A Text Node maps to a sequence of **character information items**.

Each character of the *dm:*`string-value` of the node is converted into a **character information item** as specified by this mapping:

[character code]

The Unicode code point value of the character.

[parent]

- If this node is the root of the infoset mapping operation, *unknown*.
- If this node has a parent, the information item that corresponds to the node returned by `dm:parent`.
- Otherwise *no value*.

[element content whitespace]
Unknown.

This sequence of characters constitutes the infoset mapping.

7 Conformance

The data model is intended primarily as a component that can be used by other specifications. Therefore, the data model relies on specifications that use it (such as [XML Path Language (XPath) 3.1], [XSL Transformations (XSLT) Version 3.0], and [XQuery 3.1: An XML Query Language]) to specify conformance criteria for the data model in their respective environments. Specifications that set conformance criteria for their use of the data model must not relax the constraints expressed in this specification.

Authors of conformance criteria for the use of the data model should pay particular attention to the following features of the data model:

1. Support for the normative construction from an infoset described in **3.2 Construction from an Infoset**.
2. Support for the normative construction from a PSVI described in **3.3 Construction from a PSVI**.
3. Support for XML 1.0 and XML 1.1.
4. Support for data types in XML Schema 1.0 and XML Schema 1.1.
5. How namespaces are supported, through nodes or through the alternative, implementation-dependent representation.

Note:

In addition, the `dm:is-id` and `dm:base-uri` accessors are required by functions in [XQuery and XPath Functions and Operators 3.1]. These refer to the specifications [xml:id] and [XML Base] respectively.

A XML Information Set Conformance

This specification conforms to the XML Information Set [Infoset]. The following information items **must** be exposed by the infoset producer to construct a data model unless they are explicitly identified as optional:

- The **Document Information Item** with **[base URI]**, **[children]**, and, optionally, **[unparsed entities]** properties. If the **[unparsed entities]** property is supported, the **Unparsed Entity Information Items** must also be supported.
- **Element Information Items** with **[base URI]**, **[children]**, **[attributes]**, **[in-scope namespaces]**, **[prefix]**, **[local name]**, **[namespace name]**, **[parent]** properties.
- **Attribute Information Items** with **[namespace name]**, **[prefix]**, **[local name]**, **[normalized value]**, **[attribute type]**, and **[owner element]** properties.
- **Character Information Items** with **[character code]**, **[parent]**, and, optionally, **[element content whitespace]** properties.
- **Processing Instruction Information Items** with **[base URI]**, **[target]**, **[content]** and **[parent]** properties.
- **Comment Information Items** with **[content]** and **[parent]** properties.
- **Namespace Information Items** with **[prefix]** and **[namespace name]** properties.

Other information items and properties made available by the Infoset processor are ignored. In addition to the properties above, the following PSVI properties are required on **Element Information Items** and **Attribute Information Items** if the data model is constructed from a PSVI:

- **[validity]**, **[validation attempted]**, **[type definition]**, **[type definition namespace]**, **[type definition name]**, **[type definition anonymous]**, **[nil]**, **[member type definition]**, **[member type definition namespace]**, **[member type definition name]**, **[member type definition anonymous]** and **[schema normalized value]**.

B References

B.1 Normative References

XML
Extensible Markup Language (XML) 1.0 (Fifth Edition), Tim Bray, Jean Paoli, Michael Sperberg-McQueen, *et. al.*, Editors. World Wide Web Consortium, 26 Nov 2008. This version is http://www.w3.org/TR/2008/REC-xml-20081126/. The latest version is available at http://www.w3.org/TR/xml.

Infoset
XML Information Set (Second Edition), John Cowan and Richard Tobin, Editors. World Wide Web Consortium, 04 Feb 2004. This version is http://www.w3.org/TR/2004/REC-xml-infoset-20040204. The latest version is available at http://www.w3.org/TR/xml-infoset.

Namespaces in XML
Namespaces in XML 1.0 (Third Edition), Tim Bray, Dave Hollander, Andrew Layman, *et. al.*, Editors. World Wide Web Consortium, 08 Dec 2009. This version is http://www.w3.org/TR/2009/REC-xml-names-20091208/. The latest version is available at http://www.w3.org/TR/xml-names/.

Namespaces in XML 1.1
Namespaces in XML 1.1 (Second Edition), Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin, Editors. World Wide Web Consortium, 16 Aug 2006. This version is http://www.w3.org/TR/2006/REC-xml-names11-20060816. The latest version is available at http://www.w3.org/TR/xml-names11/.

xml:id
xml:id Version 1.0, Jonathan Marsh, Daniel Veillard, and Norman Walsh, Editors. World Wide Web Consortium, 09 Sep 2005. This version is http://www.w3.org/TR/2005/REC-xml-id-20050909/. The latest version is available at http://www.w3.org/TR/xml-id/.

XQuery 1.0 and XPath 2.0 Data Model (XDM)
XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition), Norman Walsh, Mary Fernández, Ashok Malhotra, *et. al.*, Editors. World Wide Web Consortium, 14 December 2010. This version is https://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/. The latest version is available at https://www.w3.org/TR/xpath-datamodel/.

XML Path Language (XPath) 3.1
XML Path Language (XPath) 3.1, Jonathan Robie, Michael Dyck and Josh Spiegel, Editors. World Wide Web Consortium, 21 March 2017. This version is https://www.w3.org/TR/2017/REC-xpath-31-20170321/. The latest version is available at https://www.w3.org/TR/xpath-31/.

XQuery and XPath Functions and Operators 3.1
XQuery and XPath Functions and Operators 3.1, Michael Kay, Editor. World Wide Web Consortium, 21 March 2017. This version is https://www.w3.org/TR/2017/REC-xpath-functions-31-20170321/. The latest version is available at https://www.w3.org/TR/xpath-functions-31/.

Schema Part 1
XML Schema Part 1: Structures Second Edition, Henry Thompson, David Beech, Murray Maloney, and Noah Mendelsohn, Editors. World Wide Web Consortium, 28 Oct 2004. This version is http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/. The latest version is available at http://www.w3.org/TR/xmlschema-1/.

Schema Part 2
XML Schema Part 2: Datatypes Second Edition, Paul V. Biron and Ashok Malhotra, Editors. World Wide Web Consortium, 28 Oct 2004. This version is http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/. The latest version is available at http://www.w3.org/TR/xmlschema-2/.

Schema 1.1 Part 1
W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures, Sandy Gao, Michael Sperberg-McQueen, Henry Thompson, *et. al.*, Editors. World Wide Web Consortium, 05 Apr 2012. This version is http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/. The latest version is available at http://www.w3.org/TR/xmlschema11-1/.

Schema 1.1 Part 2
W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, David Peterson, Sandy Gao, Ashok Malhotra, *et. al.*, Editors. World Wide Web Consortium, 05 Apr 2012. This version is http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/. The latest version is available at http://www.w3.org/TR/xmlschema11-2/.

XSLT and XQuery Serialization 3.1
XSLT and XQuery Serialization 3.1, Andrew Coleman and Michael Sperberg-McQueen, Editors. World Wide Web Consortium, 21 March 2017. This version is https://www.w3.org/TR/2017/REC-xslt-xquery-serialization-31-20170321/. The latest version is available at https://www.w3.org/TR/xslt-xquery-serialization-31/.

XQuery 1.0 and XPath 2.0 Formal Semantics
XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition), Jérôme Siméon, Denise Draper, Peter Frankhauser, *et. al.*, Editors. World Wide Web Consortium, 14 December 2010. This version is https://www.w3.org/TR/2010/REC-xquery-semantics-20101214/. The latest version is available at https://www.w3.org/TR/xquery-semantics/.

RFC 2119
Key words for use in RFCs to Indicate Requirement Levels, S. Bradner. Network Working Group, IETF, Mar 1997.

RFC 3986
Uniform Resource Identifier (URI): Generic Syntax, T. Berners-Lee, R. Fielding, and L. Masinter. Network Working Group, IETF, Jan 2005.

RFC 3987
Internationalized Resource Identifiers (IRIs), M. Duerst and M. Suignard. Network Working Group, IETF, Jan 2005.

Character Model
Character Model for the World Wide Web 1.0: Fundamentals, Martin Dürst, François Yergeau, Richard Ishida, *et. al.*, Editors. World Wide Web Consortium, 15 Feb 2005. This version is http://www.w3.org/TR/2005/REC-charmod-20050215/. The latest version is available at http://www.w3.org/TR/charmod/.

B.2 Other References

XML Base
XML Base (Second Edition), Jonathan Marsh and Richard Tobin, Editors. World Wide Web Consortium, 28 Jan 2009. This version is http://www.w3.org/TR/2009/REC-xmlbase-20090128/. The latest version is available at http://www.w3.org/TR/xmlbase/.

XSL Transformations (XSLT) Version 3.0
XSL Transformations (XSLT) Version 3.0, Michael Kay, Editor. World Wide Web Consortium, 7 February 2017. This version is https://www.w3.org/TR/2017/CR-xslt-30-20170207/. The latest version is available at https://www.w3.org/TR/xslt-30/.

XML Query Working Group
XML Query Working Group, World Wide Web Consortium. Home page: https://www.w3.org/XML/Query/

XSLT Working Group
XSL Working Group, World Wide Web Consortium. Home page: https://www.w3.org/Style/XSL/

XQuery 3.1: An XML Query Language
XQuery 3.1: An XML Query Language, Jonathan Robie, Michael Dyck and Josh Spiegel, Editors. World Wide Web Consortium, 21 March 2017. This version is https://www.w3.org/TR/2017/REC-xquery-31-20170321/. The latest version is available at https://www.w3.org/TR/xquery-31/.

XQuery 3.1 Requirements
XQuery 3.1 Requirements and Use Cases, Jonathan Robie, Editor. World Wide Web Consortium, 13 December 2016. This version is https://www.w3.org/TR/2016/NOTE-xquery-31-requirements-20161213/. The latest version is available at https://www.w3.org/TR/xquery-31-requirements/.

ISO 8601
ISO (International Organization for Standardization). *Representations of dates and times, 2000-08-03.* Available from: http://www.iso.org/

C Schema for the Extended XS Namespace

The following schema defines the additional types in the `xs:` namespace identified by this document.

You can retrieve the normative schema document for this namespace from http://www.w3.org/2014/10/xpath-datamodel/xpath-datatypes.xsd.

```
<?xml version='1.0'?>

<!--
This is an XML Schema document for the XML Schema namespace,
http://www.w3.org/2001/XMLSchema, that has been extended to
include definitions for the xs:dayTimeDuration and
xs:yearMonthDuration types.

The other xs: types defined in XDM are not described here because
xs:untyped and xs:anyAtomicType are special types that cannot be
properly defined using XML Schema itself and because xs:untypedAtomic
should not be used for validation, but only used for unvalidated
elements and attributes.
-->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           xml:lang="en">

<xs:include schemaLocation="http://www.w3.org/2001/XMLSchema.xsd"/>

<xs:simpleType name='dayTimeDuration'>
  <xs:annotation>
    <xs:documentation
        source="http://www.w3.org/TR/xpath-datamodel#dayTimeDuration"/>
  </xs:annotation>
  <xs:restriction base='xs:duration'>
    <xs:pattern value="[^YM]*[DT].*"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name='yearMonthDuration'>
  <xs:annotation>
    <xs:documentation
        source="http://www.w3.org/TR/xpath-datamodel#yearMonthDuration"/>
  </xs:annotation>
  <xs:restriction base='xs:duration'>
    <xs:pattern value="[^DT]*"/>
  </xs:restriction>
</xs:simpleType>

</xs:schema>
```

D Glossary (Non-Normative)

Namespace URI	This specification uses the term **Namespace URI** to refer to a namespace name, whether or not it is a valid URI or IRI
Node	There are seven kinds of **Nodes** in the data model: document, element, attribute, text, namespace, processing instruction, and comment.
absent	When a property has no value, we say that it is **absent**.
array item	An **array item** is a value that represents an array.
atomic type	An **atomic type** is a primitive simple type or a type derived by restriction from another atomic type.
atomic value	An **atomic value** is a value in the value space of an atomic type and is labeled with the name of

	that atomic type.
character	A **character** is an instance of the `Char` production in [XML] .
codepoint	A **codepoint** is a non-negative integer assigned to a character by the Unicode consortium, or reserved for future assignment to a character.
document	A tree whose root node is a Document Node is referred to as a **document**.
document order	A **document order** is defined among all the nodes accessible during a given query or transformation. Document order is a total ordering, although the relative order of some nodes is implementation-dependent. Informally, document order is the order in which nodes appear in the XML serialization of a document.
expanded-QName	An **expanded-QName** is a set of three values consisting of a possibly empty prefix, a possibly empty namespace URI, and a local name.
fragment	A tree whose root node is not a Document Node is referred to as a **fragment**.
function	A **function** is an item that can be **called**.
function arity	A function's **arity** is the number of its parameters.
function signature	A **function signature** represents the type of a function.
implementation defined	**Implementation-defined** indicates an aspect that may differ between implementations, but must be specified by the implementor for each particular implementation.
implementation dependent	**Implementation-dependent** indicates an aspect that may differ between implementations, is not specified by this or any W3C specification, and is not required to be specified by the implementor for any particular implementation.
incompletely validated	An **incompletely validated** document is an XML document that has a corresponding schema but whose schema-validity assessment has resulted in one or more element or attribute information items being assigned values other than 'valid' for the **[validity]** property in the PSVI.
instance of the data model	Every **instance of the data model** is a sequence.
item	An **item** is either a node, a function, or an atomic value.
map item	A **map item** is a value that represents a map (sometimes called a hash or an associative array).
primitive simple type	The **primitive simple types** are the types defined in **2.1.1 Types adopted from XML Schema**.
root node	The **root node** is the topmost node of a tree, the node with no parent.
sequence	A **sequence** is an ordered collection of zero or more items.
stable	Document order is **stable**, which means that the relative order of two nodes will not change during the processing of a given query or transformation, even if this order is implementation-dependent.

string	A **string** is a sequence of zero or more characters, or equivalently, a value in the value space of the `xs:string` data type.

E Example (Non-Normative)

The following XML document is used to illustrate the information contained in a data model:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="dm-example.xsl"?>
<catalog xmlns="http://www.example.com/catalog"
         xmlns:html="http://www.w3.org/1999/xhtml"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.example.com/catalog
                             dm-example.xsd"
         xml:lang="en"
         version="0.1">

<!-- This example is for data model illustration only.
     It does not demonstrate good schema design. -->

<tshirt code="T1534017" label=" Staind : Been Awhile "
        xlink:href="http://example.com/0,,1655091,00.html"
        sizes="M L XL">
  <title> Staind: Been Awhile Tee Black (1-sided) </title>
  <description>
    <html:p>
      Lyrics from the hit song 'It's Been Awhile'
      are shown in white, beneath the large
      'Flock &amp; Weld' Staind logo.
    </html:p>
  </description>
  <price> 25.00 </price>
</tshirt>

<album code="A1481344" label=" Staind : Its Been A While "
       formats="CD">
  <title> It's Been A While </title>
  <description xsi:nil="true" />
  <price currency="USD"> 10.99 </price>
  <artist> Staind </artist>
</album>

</catalog>
```

The document is associated with the URI "http://www.example.com/catalog.xml", and is valid with respect to the following XML schema:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:cat="http://www.example.com/catalog"
           xmlns:xlink="http://www.w3.org/1999/xlink"
           targetNamespace="http://www.example.com/catalog"
           elementFormDefault="qualified">

<xs:import namespace="http://www.w3.org/XML/1998/namespace"
           schemaLocation="http://www.w3.org/2001/xml.xsd" />

<xs:import namespace="http://www.w3.org/1999/xlink"
           schemaLocation="http://www.cs.rpi.edu/~puninj/XGMML/xlinks-2001.xsd" />

<xs:element name="catalog">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="cat:_item" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="version" type="xs:string" fixed="0.1" use="required" />
    <xs:attribute ref="xml:base" />
    <xs:attribute ref="xml:lang" />
  </xs:complexType>
</xs:element>

<xs:element name="_item" type="cat:itemType" abstract="true" />

<xs:complexType name="itemType">
  <xs:sequence>
    <xs:element name="title" type="xs:token" />
    <xs:element name="description" type="cat:description" nillable="true" />
    <xs:element name="price" type="cat:price" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="label" type="xs:token" />
  <xs:attribute name="code" type="xs:ID" use="required" />
  <xs:attributeGroup ref="xlink:simpleLink" />
</xs:complexType>

<xs:element name="tshirt" type="cat:tshirtType" substitutionGroup="cat:_item" />

<xs:complexType name="tshirtType">
  <xs:complexContent>
    <xs:extension base="cat:itemType">
      <xs:attribute name="sizes" type="cat:clothesSizes" use="required" />
      <xs:attribute ref="xml:lang" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:simpleType name="clothesSizes">
  <xs:union memberTypes="cat:sizeList">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="oneSize" />
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>

<xs:simpleType name="sizeList">
  <xs:restriction>
    <xs:simpleType>
      <xs:list itemType="cat:clothesSize" />
    </xs:simpleType>
    <xs:minLength value="1" />
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="clothesSize">
```

```
    <xs:union memberTypes="cat:numberedSize cat:categorySize" />
  </xs:simpleType>

  <xs:simpleType name="numberedSize">
    <xs:restriction base="xs:integer">
      <xs:enumeration value="4" />
      <xs:enumeration value="6" />
      <xs:enumeration value="8" />
      <xs:enumeration value="10" />
      <xs:enumeration value="12" />
      <xs:enumeration value="14" />
      <xs:enumeration value="16" />
      <xs:enumeration value="18" />
      <xs:enumeration value="20" />
      <xs:enumeration value="22" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="categorySize">
    <xs:restriction base="xs:token">
      <xs:enumeration value="XS" />
      <xs:enumeration value="S" />
      <xs:enumeration value="M" />
      <xs:enumeration value="L" />
      <xs:enumeration value="XL" />
      <xs:enumeration value="XXL" />
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="album" type="cat:albumType" substitutionGroup="cat:_item" />

  <xs:complexType name="albumType">
    <xs:complexContent>
      <xs:extension base="cat:itemType">
        <xs:sequence>
          <xs:element name="artist" type="xs:string" />
        </xs:sequence>
        <xs:attribute name="formats" type="cat:formatsType" use="required" />
        <xs:attribute ref="xml:lang" use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:simpleType name="formatsType">
    <xs:list itemType="cat:formatType" />
  </xs:simpleType>

  <xs:simpleType name="formatType">
    <xs:restriction base="xs:token">
      <xs:enumeration value="CD" />
      <xs:enumeration value="MiniDisc" />
      <xs:enumeration value="tape" />
      <xs:enumeration value="vinyl" />
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="description" mixed="true">
    <xs:sequence>
      <xs:any namespace="http://www.w3.org/1999/xhtml" processContents="lax"
              minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute ref="xml:lang" use="optional"/>
  </xs:complexType>

  <xs:complexType name="price">
    <xs:simpleContent>
      <xs:extension base="cat:monetaryAmount">
        <xs:attribute name="currency" type="cat:currencyType" default="USD" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:simpleType name="currencyType">
    <xs:restriction base="xs:token">
      <xs:pattern value="[A-Z]{3}" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="monetaryAmount">
    <xs:restriction base="xs:decimal">
      <xs:fractionDigits value="3" />
      <xs:pattern value="\d+(\.\d{2,3})?" />
    </xs:restriction>
  </xs:simpleType>

  </xs:schema>
```

The schema is associated with the URI "http://www.example.com/dm-example.xsd".

This example exposes the data model for a document that has an associated schema and has been validated successfully against it. In general, an XML Schema is not required, that is, the data model can represent a schemaless, well-formed XML document with the rules described in **2.7 Schema Information**.

The XML document is represented by the nodes described below. The value *D1* represents a Document Node; the values *E1, E2, etc.* represent Element Nodes; the values *A1, A2, etc.* represent Attribute Nodes; the values *N1, N2, etc.* represent Namespace Nodes; the values *P1, P2, etc.* represent Processing Instruction Nodes; the values *T1, T2, etc.* represent Text Nodes.

For brevity:

- Text Nodes in the data model that contain only white space are not shown.
- Literal strings are shown in quotes without the `xs:string()` constructor
- Literal decimals are shown without the `xs:decimal()` constructor
- Nodes are referred to using the syntax `[nodeID]`
- xs:QNames are used with the following prefixes bindings:

xs	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
cat	http://www.example.com/catalog
xlink	http://www.w3.org/1999/xlink
html	http://www.w3.org/1999/xhtml
anon	An implementation-dependent prefix associated with anonymous type names

- The abbreviation " `\n` " is used in string literals to represent a newline character; this isn't supported in XPath, but it makes this presentation clearer.
- Accessors that return the empty sequence have been omitted.
- To simplify the presentation, we're assuming an implementation that does not expose the namespace axis. Therefore, Namespace Nodes are shared across multiple elements. See **6.4 Namespace Nodes**.

// Document node D1		
dm:base-uri(D1)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(D1)	=	"document"
dm:string-value(D1)	=	" Staind: Been Awhile Tee Black (1-sided) \n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n 25.00 It's Been A While 10.99 Staind "
dm:typed-value(D1)	=	xs:untypedAtomic(" Staind: Been Awhile Tee Black (1-sided) \n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n 25.00 It's Been A While 10.99 Staind ")
dm:children(D1)	=	([P1], [E1])

// Namespace node N1		
dm:node-kind(N1)	=	"namespace"
dm:node-name(N1)	=	xs:QName("", "xml")
dm:string-value(N1)	=	"http://www.w3.org/XML/1998/namespace"
dm:typed-value(N1)	=	"http://www.w3.org/XML/1998/namespace"

// Namespace node N2		
dm:node-kind(N2)	=	"namespace"
dm:node-name(N2)	=	()
dm:string-value(N2)	=	"http://www.example.com/catalog"
dm:typed-value(N2)	=	"http://www.example.com/catalog"

// Namespace node N3		
dm:node-kind(N3)	=	"namespace"
dm:node-name(N3)	=	xs:QName("", "html")
dm:string-value(N3)	=	"http://www.w3.org/1999/xhtml"
dm:typed-value(N3)	=	"http://www.w3.org/1999/xhtml"

// Namespace node N4		
dm:node-kind(N4)	=	"namespace"
dm:node-name(N4)	=	xs:QName("", "xlink")
dm:string-value(N4)	=	"http://www.w3.org/1999/xlink"
dm:typed-value(N4)	=	"http://www.w3.org/1999/xlink"

// Namespace node N5		
dm:node-kind(N5)	=	"namespace"
dm:node-name(N5)	=	xs:QName("", "xsi")
dm:string-value(N5)	=	"http://www.w3.org/2001/XMLSchema-instance"
dm:typed-value(N5)	=	"http://www.w3.org/2001/XMLSchema-instance"

// Processing Instruction node P1

dm:base-uri(P1)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(P1)	=	"processing-instruction"
dm:node-name(P1)	=	xs:QName("", "xml-stylesheet")
dm:string-value(P1)	=	"type="text/xsl" href="dm-example.xsl""
dm:typed-	=	"type="text/xsl" href="dm-example.xsl""

value(P1)		
dm:parent(P1)	=	([D1])

// Element node E1		
dm:base-uri(E1)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E1)	=	"element"
dm:node-name(E1)	=	xs:QName("http://www.example.com/catalog", "catalog")
dm:string-value(E1)	=	" Staind: Been Awhile Tee Black (1-sided) \n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n 25.00 It's Been A While 10.99 Staind "
dm:typed-value(E1)	=	fn:error()
dm:type-name(E1)	=	anon:TYP000001
dm:is-id(E1)	=	false
dm:is-idrefs(E1)	=	false
dm:parent(E1)	=	([D1])
dm:children(E1)	=	([C1], [E2], [E7])
dm:attributes(E1)	=	([A1], [A2], [A3])
dm:namespace-nodes(E1)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E1)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Attribute node A1		
dm:node-kind(A1)	=	"attribute"
dm:node-name(A1)	=	xs:QName("http://www.w3.org/2001/XMLSchema-instance", "xsi:schemaLocation")
dm:string-value(A1)	=	"http://www.example.com/catalog dm-example.xsd"
dm:typed-value(A1)	=	(xs:anyURI("http://www.example.com/catalog"), xs:anyURI("catalog.xsd"))
dm:type-name(A1)	=	anon:TYP000002
dm:is-id(A1)	=	false
dm:is-idrefs(A1)	=	false
dm:parent(A1)	=	([E1])

// Attribute node A2		
dm:node-kind(A2)	=	"attribute"
dm:node-name(A2)	=	xs:QName("http://www.w3.org/XML/1998/namespace", "xml:lang")
dm:string-value(A2)	=	"en"
dm:typed-value(A2)	=	"en"
dm:type-name(A2)	=	xs:NMTOKEN
dm:is-id(A2)	=	false
dm:is-idrefs(A2)	=	false
dm:parent(A2)	=	([E1])

// Attribute node A3		
dm:node-kind(A3)	=	"attribute"
dm:node-name(A3)	=	xs:QName("", "version")
dm:string-value(A3)	=	"0.1"
dm:typed-value(A3)	=	"0.1"
dm:type-name(A3)	=	xs:string
dm:is-id(A3)	=	false
dm:is-idrefs(A3)	=	false
dm:parent(A3)	=	([E1])

// Comment node C1		
dm:base-uri(C1)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(C1)	=	"comment"
dm:string-value(C1)	=	" This example is for data model illustration only.\n It does not demonstrate good schema design. "
dm:typed-value(C1)	=	" This example is for data model illustration only.\n It does not demonstrate good schema design. "
dm:parent(C1)	=	([E1])

// Element node E2		
dm:base-uri(E2)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E2)	=	"element"
dm:node-name(E2)	=	xs:QName("http://www.example.com/catalog", "tshirt")
dm:string-value(E2)	=	" Staind: Been Awhile Tee Black (1-sided) \n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n 25.00 "
dm:typed-value(E2)	=	fn:error()
dm:type-name(E2)	=	cat:tshirtType
dm:is-id(E2)	=	false
dm:is-idrefs(E2)	=	false
dm:parent(E2)	=	([E1])
dm:children(E2)	=	([E3], [E4], [E6])
dm:attributes(E2)	=	([A4], [A5], [A6], [A7])
dm:namespace-nodes(E2)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E2)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Attribute node A4		
dm:node-kind(A4)	=	"attribute"
dm:node-name(A4)	=	xs:QName("", "code")
dm:string-value(A4)	=	"T1534017"
dm:typed-value(A4)	=	xs:ID("T1534017")
dm:type-		

name(A4)	=	xs:ID
dm:is-id(A4)	=	true
dm:is-idrefs(A4)	=	false
dm:parent(A4)	=	([E2])

// Attribute node A5		
dm:node-kind(A5)	=	"attribute"
dm:node-name(A5)	=	xs:QName("", "label")
dm:string-value(A5)	=	"Staind : Been Awhile"
dm:typed-value(A5)	=	xs:token("Staind : Been Awhile")
dm:type-name(A5)	=	xs:token
dm:is-id(A5)	=	false
dm:is-idrefs(A5)	=	false
dm:parent(A5)	=	([E2])

// Attribute node A6		
dm:node-kind(A6)	=	"attribute"
dm:node-name(A6)	=	xs:QName("http://www.w3.org/1999/xlink", "xlink:href")
dm:string-value(A6)	=	"http://example.com/0,,1655091,00.html"
dm:typed-value(A6)	=	xs:anyURI("http://example.com/0,,1655091,00.html")
dm:type-name(A6)	=	xs:anyURI
dm:is-id(A6)	=	false
dm:is-idrefs(A6)	=	false
dm:parent(A6)	=	([E2])

// Attribute node A7		
dm:node-kind(A7)	=	"attribute"
dm:node-name(A7)	=	xs:QName("", "sizes")
dm:string-value(A7)	=	"M L XL"
dm:typed-value(A7)	=	(xs:token("M"), xs:token("L"), xs:token("XL"))
dm:type-name(A7)	=	cat:sizeList
dm:is-id(A7)	=	false
dm:is-idrefs(A7)	=	false
dm:parent(A7)	=	([E2])

// Element node E3		
dm:base-uri(E3)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E3)	=	"element"
dm:node-name(E3)	=	xs:QName("http://www.example.com/catalog", "title")
dm:string-	=	"Staind: Been Awhile Tee Black (1-sided)"

value(E3)		
dm:typed-value(E3)	=	xs:token("Staind: Been Awhile Tee Black (1-sided)")
dm:type-name(E3)	=	xs:token
dm:is-id(E3)	=	false
dm:is-idrefs(E3)	=	false
dm:parent(E3)	=	([E2])
dm:children(E3)	=	([T1])
dm:attributes(E3)	=	()
dm:namespace-nodes(E3)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E3)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Text node T1		
dm:base-uri(T1)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(T1)	=	"text"
dm:string-value(T1)	=	"Staind: Been Awhile Tee Black (1-sided)"
dm:typed-value(T1)	=	xs:untypedAtomic("Staind: Been Awhile Tee Black (1-sided)")
dm:type-name(T1)	=	xs:untypedAtomic
dm:parent(T1)	=	([E3])

// Element node E4		
dm:base-uri(E4)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E4)	=	"element"
dm:node-name(E4)	=	xs:QName("http://www.example.com/catalog", "description")
dm:string-value(E4)	=	"\n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n "
dm:typed-value(E4)	=	xs:untypedAtomic("\n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n ")
dm:type-name(E4)	=	cat:description
dm:is-id(E4)	=	false
dm:is-idrefs(E4)	=	false
dm:parent(E4)	=	([E2])
dm:children(E4)	=	([E5])
dm:attributes(E4)	=	()
dm:namespace-nodes(E4)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E4)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Element node E5		
dm:base-uri(E5)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E5)	=	"element"
dm:node-name(E5)	=	xs:QName("http://www.w3.org/1999/xhtml", "html:p")
dm:string-value(E5)	=	"\n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n "

dm:typed-value(E5)	=	xs:untypedAtomic("\n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n ")
dm:type-name(E5)	=	xs:anyType
dm:is-id(E5)	=	false
dm:is-idrefs(E5)	=	false
dm:parent(E5)	=	([E4])
dm:children(E5)	=	([T2])
dm:attributes(E5)	=	()
dm:namespace-nodes(E5)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E5)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Text node T2		
dm:base-uri(T2)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(T2)	=	"text"
dm:string-value(T2)	=	"\n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n "
dm:typed-value(T2)	=	xs:untypedAtomic("\n Lyrics from the hit song 'It's Been Awhile'\n are shown in white, beneath the large\n 'Flock & Weld' Staind logo.\n ")
dm:type-name(T2)	=	xs:untypedAtomic
dm:parent(T2)	=	([E5])

// Element node E6		
dm:base-uri(E6)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E6)	=	"element"
dm:node-name(E6)	=	xs:QName("http://www.example.com/catalog", "price")
dm:string-value(E6)	=	"25.00"
// The typed-value is based on the content type of the complex type for the element		
dm:typed-value(E6)	=	cat:monetaryAmount(25.0)
dm:type-name(E6)	=	cat:price
dm:is-id(E6)	=	false
dm:is-idrefs(E6)	=	false
dm:parent(E6)	=	([E2])
dm:children(E6)	=	([T3])
dm:attributes(E6)	=	()
dm:namespace-nodes(E6)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E6)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Text node T3		
dm:base-uri(T3)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(T3)	=	"text"
dm:string-value(T3)	=	"25.00"
dm:typed-value(T3)	=	xs:untypedAtomic("25.00")

dm:type-name(T3)	=	xs:untypedAtomic
dm:parent(T3)	=	([E6])

// Element node E7		
dm:base-uri(E7)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E7)	=	"element"
dm:node-name(E7)	=	xs:QName("http://www.example.com/catalog", "album")
dm:string-value(E7)	=	" It's Been A While 10.99 Staind "
dm:typed-value(E7)	=	fn:error()
dm:type-name(E7)	=	cat:albumType
dm:is-id(E7)	=	false
dm:is-idrefs(E7)	=	false
dm:parent(E7)	=	([E1])
dm:children(E7)	=	([E8], [E9], [E10], [E11])
dm:attributes(E7)	=	([A8], [A9], [A10])
dm:namespace-nodes(E7)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E7)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Attribute node A8		
dm:node-kind(A8)	=	"attribute"
dm:node-name(A8)	=	xs:QName("", "code")
dm:string-value(A8)	=	"A1481344"
dm:typed-value(A8)	=	xs:ID("A1481344")
dm:type-name(A8)	=	xs:ID
dm:is-id(A8)	=	true
dm:is-idrefs(A8)	=	false
dm:parent(A8)	=	([E7])

// Attribute node A9		
dm:node-kind(A9)	=	"attribute"
dm:node-name(A9)	=	xs:QName("", "label")
dm:string-value(A9)	=	"Staind : Its Been A While"
dm:typed-value(A9)	=	xs:token("Staind : Its Been A While")
dm:type-name(A9)	=	xs:token
dm:is-id(A9)	=	false
dm:is-idrefs(A9)	=	false
dm:parent(A9)	=	([E7])

// Attribute node A10		

dm:node-kind(A10)	=	"attribute"
dm:node-name(A10)	=	xs:QName("", "formats")
dm:string-value(A10)	=	"CD"
dm:typed-value(A10)	=	cat:formatType("CD")
dm:type-name(A10)	=	cat:formatType
dm:is-id(A10)	=	false
dm:is-idrefs(A10)	=	false
dm:parent(A10)	=	([E7])

// Element node E8		
dm:base-uri(E8)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E8)	=	"element"
dm:node-name(E8)	=	xs:QName("http://www.example.com/catalog", "title")
dm:string-value(E8)	=	"It's Been A While"
dm:typed-value(E8)	=	xs:token("It's Been A While")
dm:type-name(E8)	=	xs:token
dm:is-id(E8)	=	false
dm:is-idrefs(E8)	=	false
dm:parent(E8)	=	([E7])
dm:children(E8)	=	([T4])
dm:attributes(E8)	=	()
dm:namespace-nodes(E8)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E8)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Text node T4		
dm:base-uri(T4)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(T4)	=	"text"
dm:string-value(T4)	=	"It's Been A While"
dm:typed-value(T4)	=	xs:untypedAtomic("It's Been A While")
dm:type-name(T4)	=	xs:untypedAtomic
dm:parent(T4)	=	([E8])

// Element node E9		
dm:base-uri(E9)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E9)	=	"element"
dm:node-name(E9)	=	xs:QName("http://www.example.com/catalog", "description")
dm:string-value(E9)	=	""
// xsi:nil is true so the typed value is the empty sequence		

dm:typed-value(E9)	=	()
dm:type-name(E9)	=	cat:description
dm:is-id(E9)	=	false
dm:is-idrefs(E9)	=	false
dm:parent(E9)	=	([E7])
dm:children(E9)	=	()
dm:attributes(E9)	=	([A11])
dm:namespace-nodes(E9)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E9)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Attribute node A11		
dm:node-kind(A11)	=	"attribute"
dm:node-name(A11)	=	xs:QName("http://www.w3.org/2001/XMLSchema-instance", "xsi:nil")
dm:string-value(A11)	=	"true"
dm:typed-value(A11)	=	xs:boolean("true")
dm:type-name(A11)	=	xs:boolean
dm:is-id(A11)	=	false
dm:is-idrefs(A11)	=	false
dm:parent(A11)	=	([E9])

// Element node E10		
dm:base-uri(E10)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E10)	=	"element"
dm:node-name(E10)	=	xs:QName("http://www.example.com/catalog", "price")
dm:string-value(E10)	=	"10.99"
dm:typed-value(E10)	=	cat:monetaryAmount(10.99)
dm:type-name(E10)	=	cat:price
dm:is-id(E10)	=	false
dm:is-idrefs(E10)	=	false
dm:parent(E10)	=	([E7])
dm:children(E10)	=	([T5])
dm:attributes(E10)	=	([A12])
dm:namespace-nodes(E10)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E10)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Attribute node A12		
dm:node-	=	"attribute"

kind(A12)		
dm:node-name(A12)	=	xs:QName("", "currency")
dm:string-value(A12)	=	"USD"
dm:typed-value(A12)	=	cat:currencyType("USD")
dm:type-name(A12)	=	cat:currencyType
dm:is-id(A12)	=	false
dm:is-idrefs(A12)	=	false
dm:parent(A12)	=	([E10])

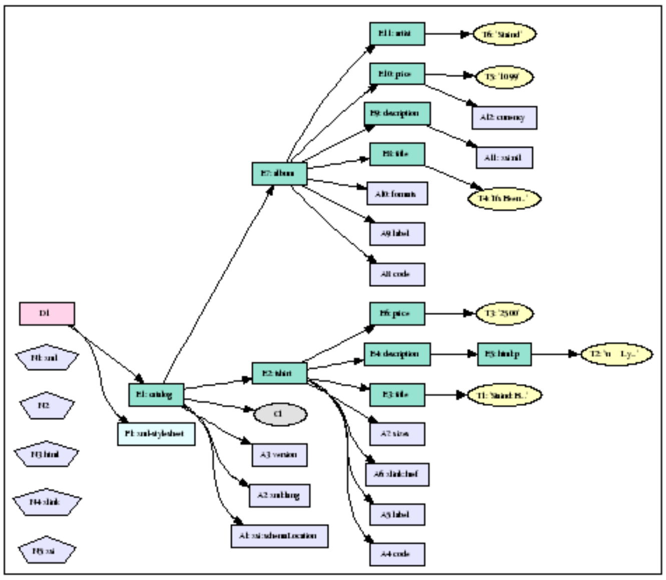
// Text node T5		
dm:base-uri(T5)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(T5)	=	"text"
dm:string-value(T5)	=	"10.99"
dm:typed-value(T5)	=	xs:untypedAtomic("10.99")
dm:type-name(T5)	=	xs:untypedAtomic
dm:parent(T5)	=	([E10])

// Element node E11		
dm:base-uri(E11)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(E11)	=	"element"
dm:node-name(E11)	=	xs:QName("http://www.example.com/catalog", "artist")
dm:string-value(E11)	=	" Staind "
dm:typed-value(E11)	=	" Staind "
dm:type-name(E11)	=	xs:string
dm:is-id(E11)	=	false
dm:is-idrefs(E11)	=	false
dm:parent(E11)	=	([E7])
dm:children(E11)	=	([T6])
dm:attributes(E11)	=	()
dm:namespace-nodes(E11)	=	([N1], [N2], [N3], [N4], [N5])
dm:namespace-bindings(E11)	=	("xml", "http://www.w3.org/XML/1998/namespace", "", "http://www.example.com/catalog", "html", "http://www.w3.org/1999/xhtml", "xlink", "http://www.w3.org/1999/xlink", "xsi", "http://www.w3.org/2001/XMLSchema-instance")

// Text node T6		
dm:base-uri(T6)	=	xs:anyURI("http://www.example.com/catalog.xml")
dm:node-kind(T6)	=	"text"
dm:string-value(T6)	=	" Staind "
dm:typed-value(T6)	=	xs:untypedAtomic(" Staind ")
dm:type-name(T6)	=	xs:untypedAtomic

dm:parent(T6)	=	([E11])

A graphical representation of the data model for the preceding example is shown below. Document order in this representation can be found by following the traditional left-to-right, depth-first traversal; however, because the image has been rotated for easier presentation, this appears to be bottom-to-top, depth-first order.



Graphic representation of the data model. [large view, SVG]

F Implementation-Defined and Implementation-Dependent Items (Non-Normative)

F.1 Implementation-Defined Items

The following items are implementation-defined.

1. Support for additional user-defined or implementation-defined types is implementation-defined. (See **2.7.1 Representation of Types**)
2. When converting from an `xs:string` to an `xs:float` or `xs:double`, it is implementation-defined whether the lexical value “-0” (and similar forms such as “-0.0”) convert to negative zero or to positive zero in the value space. (See **2.7.7 Negative Zero**)
3. Data model construction from sources other than an Infoset or PSVI is implementation-defined. (See **3 Data Model Construction**)
4. Some typed values in the data model are *absent*. Attempting to access an absent typed value is an error. Behavior in these cases is implementation-defined and the host language is responsible for determining the result. (See **5 Accessors**)

F.2 Implementation-Dependent Items

The following items are implementation-dependent.

1. The relative order of Namespace Nodes nodes is stable but implementation-dependent. (See **2.4 Document Order**)
2. The relative order of Attribute Nodes nodes is stable but implementation-dependent. (See **2.4 Document Order**)
3. The relative order of distinct trees is stable but implementation-dependent. (See **2.4 Document Order**)
4. The names of anonymous types are implementation-dependent. (See **2.7.1 Representation of Types**)
5. The prefix associated with type names is implementation-dependent. (See **3.3.1.1 Element and Attribute Node Types**)

G Change Log (Non-Normative)

The following changes have been made to this document since the XPath and XQuery Data Model 3.0 Recommendation of 8 April 2014.

G.1 Substantive changes

- Update the text and diagrams that specify the type system for XPath, XQuery, and XSLT.
- Incorporate new material that defines a new kind of item — map item — in the data model. See **2.8.2 Map Items**.
- Incorporate new material that defines a new kind of item — array item — in the data model. See **2.8.3 Array Items**.

G.2 Minor changes

- Update document text to reflect version 3.1
- Clarify the meaning of the word "identity" in the context of this specification.
- Clarify that there exists in the data model only one unparsed entity associated with any given name.

H Accessor Summary (Non-Normative)

This section summarizes the return values of each accessor by node type.

H.1 *dm:*attributes Accessor

Document Nodes
Returns the empty sequence

Element Nodes
Returns the value of the **attributes** property. The order of Attribute Nodes is stable but implementation dependent.

Attribute Nodes
Returns the empty sequence.

Namespace Nodes
Returns the empty sequence.

Processing Instruction Nodes
Returns the empty sequence.

Comment Nodes
Returns the empty sequence.

Text Nodes
Returns the empty sequence.

H.2 *dm:*base-uri Accessor

Document Nodes
Returns the value of the **base-uri** property.

Element Nodes
Returns the value of the **base-uri** property.

Attribute Nodes
If the attribute has a parent, returns the value of the `dm:base-uri` of its parent; otherwise it returns the empty sequence.

Namespace Nodes
Returns the empty sequence.

Processing Instruction Nodes
Returns the value of the **base-uri** property.

Comment Nodes

If the comment has a parent, returns the value of the `dm:base-uri` of its parent; otherwise, returns the empty sequence.

Text Nodes

If the Text Node has a parent, returns the value of the `dm:base-uri` of its parent; otherwise, returns the empty sequence.

H.3 *dm:*children Accessor

Document Nodes

Returns the value of the **children** property.

Element Nodes

Returns the value of the **children** property.

Attribute Nodes

Returns the empty sequence.

Namespace Nodes

Returns the empty sequence.

Processing Instruction Nodes

Returns the empty sequence.

Comment Nodes

Returns the empty sequence.

Text Nodes

Returns the empty sequence.

H.4 *dm:*document-uri Accessor

Document Nodes

Returns the absolute URI of the resource from which the Document Node was constructed, or the empty sequence if no such absolute URI is available.

Element Nodes

Returns the empty sequence.

Attribute Nodes

Returns the empty sequence.

Namespace Nodes

Returns the empty sequence.

Processing Instruction Nodes

Returns the empty sequence.

Comment Nodes

Returns the empty sequence.

Text Nodes

Returns the empty sequence.

H.5 *dm:*is-id Accessor

Document Nodes

Returns the empty sequence.

Element Nodes

Returns the value of the **is-id** property.

Attribute Nodes

Returns the value of the **is-id** property.

Namespace Nodes

Returns the empty sequence.

Processing Instruction Nodes
Returns the empty sequence.

Comment Nodes
Returns the empty sequence.

Text Nodes
Returns the empty sequence.

H.6 *dm:*is-idrefs Accessor

Document Nodes
Returns the empty sequence.

Element Nodes
Returns the value of the **is-idrefs** property.

Attribute Nodes
Returns the value of the **is-idrefs** property.

Namespace Nodes
Returns the empty sequence.

Processing Instruction Nodes
Returns the empty sequence.

Comment Nodes
Returns the empty sequence.

Text Nodes
Returns the empty sequence.

H.7 *dm:*namespace-nodes Accessor

Document Nodes
Returns the empty sequence

Element Nodes
Returns the value of the **namespaces** property as a sequence of Namespace Nodes. The order of Namespace Nodes is stable but implementation dependent.

Attribute Nodes
Returns the empty sequence.

Namespace Nodes
Returns the empty sequence.

Processing Instruction Nodes
Returns the empty sequence.

Comment Nodes
Returns the empty sequence.

Text Nodes
Returns the empty sequence.

H.8 *dm:*nilled Accessor

Document Nodes
Returns the empty sequence

Element Nodes
Returns the value of the **nilled** property.

Attribute Nodes
Returns the empty sequence.

Namespace Nodes
Returns the empty sequence.

Processing Instruction Nodes
Returns the empty sequence.

Comment Nodes
Returns the empty sequence.

Text Nodes
Returns the empty sequence.

H.9 *dm:*node-kind Accessor

Document Nodes
Returns “`document`”.

Element Nodes
Returns “`element`”.

Attribute Nodes
Returns “`attribute`”.

Namespace Nodes
Returns “`namespace`”.

Processing Instruction Nodes
Returns “`processing-instruction`”.

Comment Nodes
Returns “`comment`”.

Text Nodes
Returns “`text`”.

H.10 *dm:*node-name Accessor

Document Nodes
Returns the empty sequence.

Element Nodes
Returns the value of the **node-name** property.

Attribute Nodes
Returns the value of the **node-name** property.

Namespace Nodes
If the **prefix** is not empty, returns an `xs:QName` with the value of the **prefix** property in the local-name and an empty namespace name, otherwise returns the empty sequence.

Processing Instruction Nodes
Returns an `xs:QName` with the value of the **target** property in the local-name and an empty namespace URI and empty prefix.

Comment Nodes
Returns the empty sequence.

Text Nodes
Returns the empty sequence.

H.11 *dm:*parent Accessor

Document Nodes
Returns the empty sequence

Element Nodes
Returns the value of the **parent** property.

Attribute Nodes
Returns the value of the **parent** property.

Namespace Nodes
Returns the value of the **parent** property.

Processing Instruction Nodes
Returns the value of the **parent** property.

Comment Nodes
Returns the value of the **parent** property.

Text Nodes
Returns the value of the **parent** property.

H.12 *dm:*string-value Accessor

Document Nodes
Returns the value of the **string-value** property.

Element Nodes
Returns the value of the **string-value** property.

Attribute Nodes
Returns the value of the **string-value** property.

Namespace Nodes
Returns the value of the **uri** property.

Processing Instruction Nodes
Returns the value of the **content** property.

Comment Nodes
Returns the value of the **content** property.

Text Nodes
Returns the value of the **content** property.

H.13 *dm:*type-name Accessor

Document Nodes
Returns the empty sequence.

Element Nodes
Returns the value of the **schema-type** property.

Attribute Nodes
Returns the value of the **schema-type** property.

Namespace Nodes
Returns the empty sequence.

Processing Instruction Nodes
Returns the empty sequence.

Comment Nodes
Returns the empty sequence.

Text Nodes
Returns `xs:untypedAtomic`.

H.14 *dm:*typed-value Accessor

Document Nodes
Returns the value of the **typed-value** property.

Element Nodes

Returns the value of the **typed-value** property.

Attribute Nodes
Returns the value of the **typed-value** property.

Namespace Nodes
Returns the value of the **uri** property as an `xs:string`.

Processing Instruction Nodes
Returns the value of the **content** property as a `xs:string`.

Comment Nodes
Returns the value of the **content** property as a `xs:string`.

Text Nodes
Returns the value of the **content** property as an `xs:untypedAtomic`.

H.15 *dm:*unparsed-entity-public-id Accessor

Document Nodes
Returns the public identifier of the specified unparsed entity or the empty sequence if no such entity exists.

Element Nodes
Returns the empty sequence.

Attribute Nodes
Returns the empty sequence.

Namespace Nodes
Returns the empty sequence.

Processing Instruction Nodes
Returns the empty sequence.

Comment Nodes
Returns the empty sequence.

Text Nodes
Returns the empty sequence.

H.16 *dm:*unparsed-entity-system-id Accessor

Document Nodes
Returns the system identifier of the specified unparsed entity or the empty sequence if no such entity exists.

Element Nodes
Returns the empty sequence.

Attribute Nodes
Returns the empty sequence.

Namespace Nodes
Returns the empty sequence.

Processing Instruction Nodes
Returns the empty sequence.

Comment Nodes
Returns the empty sequence.

Text Nodes
Returns the empty sequence.

I Infoset Construction Summary (Non-Normative)

This section summarizes data model construction from an Infoset for each kind of information item. General notes occur elsewhere.

I.1 Document Nodes Information Items

The **document information item** is required. A Document Node is constructed for each **document information item**.

The following infoset properties are required: **[children]** and **[base URI]**.

The following infoset properties are optional: **[unparsed entities]**.

Document Node properties are derived from the infoset as follows:

base-uri
The value of the **[base URI]** property, if available. Note that the base URI property, if available, is always an absolute URI (if an absolute URI can be computed) though it may contain Unicode characters that are not allowed in URIs. These characters, if they occur, are present in the **base-uri** property and will have to be encoded and escaped by the application to obtain a URI suitable for retrieval, if retrieval is required.

In practice a **[base URI]** is not always known. In this case the value of the **base-uri** property of the document node will be the empty sequence. This is not intrinsically an error, though it may cause some operations that depend on the base URI to fail.

children
The sequence of nodes constructed from the information items found in the **[children]** property.

For each element, processing instruction, and comment found in the **[children]** property, a corresponding Element, Processing Instruction, or Comment Node is constructed and that sequence of nodes is used as the value of the **children** property.

If present among the **[children]**, the **document type declaration information item** is ignored.

unparsed-entities
If the **[unparsed entities]** property is present and is not the empty set, the values of the **unparsed entity information items** must be used to support the `dm:unparsed-entity-system-id` and `dm:unparsed-entity-public-id` accessors.

The internal structure of the values of the **unparsed-entities** property is implementation defined.

string-value
The concatenation of the string-values of all its Text Node descendants in document order. If the document has no such descendants, the zero-length string.

typed-value
The `dm:string-value` of the node as an `xs:untypedAtomic` value.

document-uri
The **document-uri** property holds the absolute URI for the resource from which the document node was constructed, if one is available and can be made absolute. For example, if a collection of documents is returned by the `fn:collection` function, the **document-uri** property may serve to distinguish between them even though each has the same **base-uri** property.

If the **document-uri** is not the empty sequence, then the following constraint must hold: the node returned by evaluating `fn:doc()` with the **document-uri** as its argument must return the document node that provided the value of the **document-uri** property.

In other words, for any Document Node `$arg`, either `fn:document-uri($arg)` must return the empty sequence or `fn:doc(fn:document-uri($arg))` must return `$arg`.

I.2 Element Nodes Information Items

The **element information items** are required. An Element Node is constructed for each **element information item**.

The following infoset properties are required: **[namespace name]**, **[local name]**, **[children]**, **[attributes]**, **[in-scope namespaces]**, **[base URI]**, and **[parent]**.

Element Node properties are derived from the infoset as follows:

base-uri
The value of the **[base URI]** property, if available. Note that the base URI property, if available, is always an absolute URI (if an absolute URI can be computed) though it may contain Unicode characters that are not allowed in URIs.

These characters, if they occur, are present in the **base-uri** property and will have to be encoded and escaped by the application to obtain a URI suitable for retrieval, if retrieval is required.

In practice a **[base URI]** is not always known. In this case the value of the **base-uri** property of the document node will be the empty sequence. This is not intrinsically an error, though it may cause some operations that depend on the base URI to fail.

node-name
An `xs:QName` constructed from the **[prefix]**, **[local name]**, and **[namespace name]** properties.

parent
The node that corresponds to the value of the **[parent]** property or the empty sequence if there is no parent.

schema-type
All Element Nodes constructed from an infoset have the type `xs:untyped`.

children
The sequence of nodes constructed from the information items found in the **[children]** property.

For each element, processing instruction, comment, and maximal sequence of adjacent **character information items** found in the **[children]** property, a corresponding Element, Processing Instruction, Comment, or Text Node is constructed and that sequence of nodes is used as the value of the **children** property.

Because the data model requires that all general entities be expanded, there will never be **unexpanded entity reference information item** children.

attributes
A set of Attribute Nodes constructed from the **attribute information items** appearing in the **[attributes]** property. This includes all of the "special" attributes (`xml:lang`, `xml:space`, `xsi:type`, etc.) but does not include namespace declarations (because they are not attributes).

Default and fixed attributes provided by the DTD are added to the **[attributes]** and are therefore included in the data model **attributes** of an element.

namespaces
A set of Namespace Nodes constructed from the **namespace information items** appearing in the **[in-scope namespaces]** property. Implementations that do not support Namespace Nodes may simply preserve the relevant bindings in this property.

Implementations **may** ignore **namespace information items** for namespaces which are not known to be used. A namespace is known to be used if:

- It appears in the expanded QName of the **node-name** of the element.
- It appears in the expanded QName of the **node-name** of any of the element's attributes.

Note: applications may rely on namespaces that are not known to be used, for example when QNames are used in content and that content does not have a type of `xs:QName` Such applications may have difficulty processing data models where some namespaces have been ignored.

nilled
All Element Nodes constructed from an infoset have a **nilled** property of " *false* ".

string-value
The **string-value** is constructed from the **character information item [children]** of the element and all its descendants. The precise rules for selecting significant **character information items** and constructing characters from them is described in **6.7.3 Construction from an Infoset** of **6.7 Text Nodes**.

This process is equivalent to concatenating the *dm:*`string-value`s of all of the Text Node descendants of the resulting Element Node.

If the element has no such descendants, the **string-value** is the empty string.

typed-value
The **string-value** as an `xs:untypedAtomic`.

is-id
All Element Nodes constructed from an infoset have a **is-id** property of " *false* ".

is-idrefs
All Element Nodes constructed from an infoset have a **is-idrefs** property of " *false* ".

I.3 Attribute Nodes Information Items

The **attribute information items** are required. An Attribute Node is constructed for each **attribute information item**.

The following infoset properties are required: **[namespace name]**, **[local name]**, **[normalized value]**, **[attribute type]**, and **[owner element]**.

Attribute Node properties are derived from the infoset as follows:

node-name
An `xs:QName` constructed from the **[prefix]**, **[local name]**, and **[namespace name]** properties.

parent
The Element Node that corresponds to the value of the **[owner element]** property or the empty sequence if there is no owner.

schema-type
The value `xs:untypedAtomic`.

string-value
The **[normalized value]** of the attribute.

typed-value
The attribute's typed-value is its *dm:*`string-value` as an `xs:untypedAtomic`.

is-id
If the attribute is named `xml:id` and its **[attribute type]** property does not have the value `ID`, then [xml:id] processing is performed. This will assure that the value does have the type `ID` and that it is properly normalized. If an error is encountered during xml:id processing, an implementation **may** raise a dynamic error. The **is-id** property is always `true` for attributes named `xml:id`.

If the **[attribute type]** property has the value `ID`, `true`, otherwise `false`.

is-idrefs
If the **[attribute type]** property has the value `IDREF` or `IDREFS`, `true`, otherwise `false`.

I.4 Namespace Nodes Information Items

The **namespace information items** are required.

The following infoset properties are required: **[prefix]**, **[namespace name]**.

Namespace Node properties are derived from the infoset as follows:

prefix
The **[prefix]** property.

uri
The **[namespace name]** property.

parent
The element in whose **[in-scope namespaces]** property the **namespace information item** appears, if the implementation exposes any mechanism for accessing the *dm:*`parent` accessor of Namespace Nodes.

I.5 Processing Instruction Nodes Information Items

A Processing Instruction Node is constructed for each **processing instruction information item** that is not ignored.

The following infoset properties are required: **[target]**, **[content]**, **[base URI]**, and **[parent]**.

Processing Instruction Node properties are derived from the infoset as follows:

target

The value of the **[target]** property.

content
The value of the **[content]** property.

base-uri
The value of the **[base URI]** property, if available. Note that the base URI property, if available, is always an absolute URI (if an absolute URI can be computed) though it may contain Unicode characters that are not allowed in URIs. These characters, if they occur, are present in the **base-uri** property and will have to be encoded and escaped by the application to obtain a URI suitable for retrieval, if retrieval is required.

In practice a **[base URI]** is not always known. In this case the value of the **base-uri** property of the document node will be the empty sequence. This is not intrinsically an error, though it may cause some operations that depend on the base URI to fail.

parent
The node corresponding to the value of the **[parent]** property.

There are no Processing Instruction Nodes for processing instructions that are children of a **document type declaration information item**.

I.6 Comment Nodes Information Items

The **comment information items** are optional.

A Comment Node is constructed for each **comment information item**.

The following infoset properties are required: **[content]** and **[parent]**.

Comment Node properties are derived from the infoset as follows:

content
The value of the **[content]** property.

parent
The node corresponding to the value of the **[parent]** property.

There are no Comment Nodes for comments that are children of a **document type declaration information item**.

I.7 Text Nodes Information Items

The **character information items** are required. A Text Node is constructed for each maximal sequence of **character information items** in document order.

The following infoset properties are required: **[character code]** and **[parent]**.

The following infoset properties are optional: **[element content whitespace]**.

A sequence of **character information items** is maximal if it satisfies the following constraints:

1. All of the information items in the sequence have the same parent.
2. The sequence consists of adjacent **character information items** uninterrupted by other types of information item.
3. No other such sequence exists that contains any of the same **character information items** and is longer.

Text Node properties are derived from the infoset as follows:

content
A string comprised of characters that correspond to the **[character code]** properties of each of the **character information items**.

If the resulting Text Node consists entirely of whitespace and the **[element content whitespace]** property of the **character information items** used to construct this node are `true`, the **content** of the Text Node is the zero-length string. Text Nodes are only allowed to be empty if they have no parents; an empty Text Node will be discarded when its parent is constructed, if it has a parent.

The content of the Text Node is not necessarily normalized as described in the [Character Model]. It is the responsibility of data producers to provide normalized text, and the responsibility of applications to make sure that operations do not de-normalize text.

parent
The node corresponding to the value of the **[parent]** property.

J PSVI Construction Summary (Non-Normative)

This section summarizes data model construction from a PSVI for each kind of information item. General notes occur elsewhere.

J.1 Document Nodes Information Items

Construction from a PSVI is identical to construction from the Infoset.

J.2 Element Nodes Information Items

The following Element Node properties are affected by PSVI properties.

schema-type
The **schema-type** is determined as described in **3.3.1.1 Element and Attribute Node Types**.

children
The sequence of nodes constructed from the information items found in the **[children]** property.

For each element, processing instruction, comment, and maximal sequence of adjacent **character information items** found in the **[children]** property, a corresponding Element, Processing Instruction, Comment, or Text Node is constructed and that sequence of nodes is used as the value of the **children** property.

For elements with schema simple types, or complex types with simple content, if the **[schema normalized value]** PSVI property exists, the processor **may** use a sequence of nodes containing the Processing Instruction and Comment Nodes corresponding to the **processing instruction** and **comment information items** found in the **[children]** property, plus an optional single Text Node whose string value is the **[schema normalized value]** for the **children** property. If the **[schema normalized value]** is the empty string, the Text Node **must not** be present, otherwise it **must** be present.

The relative order of Processing Instruction and Comment Nodes must be preserved, but the position of the Text Node, if it is present, among them is implementation defined.

The effect of the above rules is that where a fixed or default value for an element is defined in the schema, and the element takes this default value, a text node will be created to contain the value, even though there are no character information items representing the value in the PSVI. The position of this text node relative to any comment or processing instruction children is implementation-dependent.

[Schema Part 1] also permits an element with mixed content to take a default or fixed value (which will always be a simple value), but it is unclear how such a defaulted value is represented in the PSVI. Implementations therefore **may** represent such a default value by creating a text node, but are not required to do so.

> **Note:**
>
> Section 3.3.1 in [Schema 1.1 Part 1] clarifies the PSVI contributions of element default or fixed values in mixed content: additional character information items are not added to the PSVI.

Because the data model requires that all general entities be expanded, there will never be **unexpanded entity reference information item** children.

attributes
A set of Attribute Nodes constructed from the **attribute information items** appearing in the **[attributes]** property. This includes all of the "special" attributes (`xml:lang`, `xml:space`, `xsi:type`, etc.) but does not include namespace declarations (because they are not attributes).

Default and fixed attributes provided by XML Schema processing are added to the **[attributes]** and are therefore included in the data model **attributes** of an element.

namespaces

A set of Namespace Nodes constructed from the **namespace information items** appearing in the **[in-scope namespaces]** property. Implementations that do not support Namespace Nodes may simply preserve the relevant bindings in this property.

Implementations **may** ignore **namespace information items** for namespaces which are not known to be used. A namespace is known to be used if:

- It appears in the expanded QName of the **node-name** of the element.
- It appears in the expanded QName of the **node-name** of any of the element's attributes.
- It appears in the expanded QName of any values of type `xs:QName` that appear among the element's children or the typed values of its attributes.

Note: applications may rely on namespaces that are not known to be used, for example when QNames are used in content and that content does not have a type of `xs:QName` Such applications may have difficulty processing data models where some namespaces have been ignored.

nilled

If the **[validity]** property exists on an information item and is " *valid* " then if the **[nil]** property exists and is true, then the **nilled** property is " *true* ". In all other cases, including all cases where schema validity assessment was not attempted or did not succeed, the **nilled** property is " *false* ".

string-value

The string-value is calculated as follows:

- If the element is empty: its string value is the zero length string.
- If the element has a type of `xs:untyped`, a complex type with element-only content, or a complex type with mixed content: its string-value is the concatenation of the **string-value**s of all its Text Node descendants in document order.
- If the element has a simple type or a complex type with simple content: its string-value is the **[schema normalized value]** of the node.

If an implementation stores only the typed value of an element, it may use any valid lexical representation of the typed value for the **string-value** property.

typed-value

The typed-value is calculated as follows:

- If the element is of type `xs:untyped`, its typed-value is its *`dm:`*`string-value` as an `xs:untypedAtomic`.
- If the element has a complex type with empty content, its typed-value is the empty sequence.
- If the element has a simple type or a complex type with simple content: its typed value is computed as described in **3.3.1.2 Typed Value Determination**. The result is a sequence of zero or more atomic values. The relationship between the type-name, typed-value, and string-value of an element node is consistent with XML Schema validation.

  Note that in the case of dates and times, the timezone is preserved as described in **3.3.2 Dates and Times**, and in the case of `xs:QName`s and `xs:NOTATION`s, the prefix is preserved as described in **3.3.3 QNames and NOTATIONS**.
- If the element has a complex type with mixed content (including `xs:anyType`), its typed-value is its *`dm:`*`string-value` as an `xs:untypedAtomic`.
- Otherwise, the element must be a complex type with element-only content. The typed-value of such an element is absent. Attempting to access this property with the *`dm:`*`typed-value` accessor always raises an error.

is-id

If the element has a complex type with element-only content, the **is-id** property is `false`. Otherwise, if the typed-value of the element consists of exactly one atomic value and that value is of type `xs:ID`, or a type derived from `xs:ID`, the **is-id** property is `true`, otherwise it is `false`.

Note:

This means that in the case of a type constructed by list from `xs:ID`, the ID is recognized provided that the list is of length one. A type constructed as a union involving `xs:ID` is recognized provided the actual value is of type `xs:ID`.

Note:

The element that is marked with the **is-id** property, and which will therefore be retrieved by the fn:id function, is the node whose typed value contains the `xs:ID` value. This node is a child of the element node that, according to XML Schema, is uniquely identified by this ID.

is-idrefs

If the element has a complex type with element-only content, the **is-idrefs** property is `false`. Otherwise, if any of the atomic values in the typed-value of the element is of type `xs:IDREF` or `xs:IDREFS`, or a type derived from one of those types, the **is-idrefs** property is `true`, otherwise it is `false`.

All other properties have values that are consistent with construction from an infoset.

J.3 Attribute Nodes Information Items

The following Attribute Node properties are affected by PSVI properties.

string-value

- The **[schema normalized value]** PSVI property if that exists.
- Otherwise, the **[normalized value]** property.

If an implementation stores only the typed value of an attribute, it may use any valid lexical representation of the typed value for the **string-value** property.

schema-type

The **schema-type** is determined as described in **3.3.1.1 Element and Attribute Node Types**.

typed-value

The typed-value is calculated as follows:

- If the attribute is of type `xs:untypedAtomic`: its typed-value is its *dm:*`string-value` as an `xs:untypedAtomic`.
- Otherwise, a sequence of zero or more atomic values as described in **3.3.1.2 Typed Value Determination**. The relationship between the type-name, typed-value, and string-value of an attribute node is consistent with XML Schema validation.

is-id

If the attribute is named `xml:id` and its **[attribute type]** property does not have the value `xs:ID` or a type derived from `xs:ID`, then [xml:id] processing is performed. This will assure that the value does have the type `xs:ID` and that it is properly normalized. If an error is encountered during xml:id processing, an implementation **may** raise a dynamic error. The **is-id** property is always true for attributes named `xml:id`.

Otherwise, if the typed-value of the attribute consists of exactly one atomic value and that value is of type `xs:ID`, or a type derived from `xs:ID`, the **is-id** property is `true`, otherwise it is `false`.

Note:

This means that in the case of a type constructed by list from `xs:ID`, the ID is recognized provided that the list is of length one. A type constructed as a union involving `xs:ID` is recognized provided the actual value is of type `xs:ID`.

is-idrefs

If any of the atomic values in the typed-value of the attribute is of type `xs:IDREF` or `xs:IDREFS`, or a type derived from one of those types, the **is-idrefs** property is `true`, otherwise it is `false`.

Note:

This rule means that a node whose type is constructed by list with an item type of `xs:IDREF` (or a type derived from `xs:IDREF`) may have the **is-idrefs** property, whether or not the list type is named `xs:IDREFS` or is derived from `xs:IDREFS`. Because union types are allowed, it also means that an element or attribute with the **is-idrefs** property can contain atomic values of type `xs:IDREF` alongside values of other types. A node has the **is-idrefs** property only if the typed value contains at least one atomic value that is an instance of `xs:IDREF`; it is not sufficient that the type annotation permits such values.

All other properties have values that are consistent with construction from an infoset.

Note: attributes from the XML Schema instance namespace, " `http://www.w3.org/2001/XMLSchema-instance` ", (`xsi:schemaLocation`, `xsi:type`, etc.) appear as ordinary attributes in the data model.

J.4 Namespace Nodes Information Items

Construction from a PSVI is identical to construction from the Infoset.

J.5 Processing Instruction Nodes Information Items

Construction from a PSVI is identical to construction from the Infoset.

J.6 Comment Nodes Information Items

Construction from a PSVI is identical to construction from the Infoset.

J.7 Text Nodes Information Items

For Text Nodes constructed from the **[schema normalized value]** of elements, **content** contains the value of the **[schema normalized value]**.

Otherwise, construction from a PSVI is the same as construction from the Infoset except for the **content** property. When constructing the **content** property, **[element content whitespace]** is not used to test if whitespace is collapsed. Instead, if the resulting Text Node consists entirely of whitespace and the **character information items** used to construct this node have a parent and that parent is an element and its {content type} is not “`mixed`”, then the **content** of the Text Node is the zero-length string.

Text Nodes are only allowed to be empty if they have no parents; an empty Text Node will be discarded when its parent is constructed, if it has a parent.

K Infoset Mapping Summary (Non-Normative)

This section summarizes the infoset mapping for each kind of node. General notes occur elsewhere.

K.1 Document Nodes Information Items

A Document Node maps to a **document information item**. The mapping fails and produces no value if the Document Node contains Text Node children that do not consist entirely of white space or if the Document Node contains more than one Element Node child.

The following properties are specified by this mapping:

[children]
: A list of information items obtained by processing each of the *`dm:`*`children` in order and mapping each to the appropriate information item(s).

[document element]

The **element information item** that is among the **[children]**.

[unparsed entities]
An unordered set of **unparsed entity information item**s constructed from the **unparsed-entities**.

Each unparsed entity maps to an **unparsed entity information item**. The following properties are specified by this mapping:

[name]
The name of the entity.

[system identifier]
The system identifier of the entity.

[public identifier]
The public identifier of the entity.

[declaration base URI]
Implementation defined. In many cases, the **document-uri** is the correct answer and implementations **must** use this value if they have no better information. Implementations that keep track of the original **[declaration base URI]** for entities should use that value.

The following properties of the **unparsed entity information item** have no value: **[notation name]**, **[notation]**.

The following properties of the **document information item** have no value: **[notations] [character encoding scheme] [standalone] [version] [all declarations processed]**.

K.2 Element Nodes Information Items

An Element Node maps to an **element information item**.

The following properties are specified by this mapping:

[namespace name]
The namespace name of the value of `dm:node-name`.

[local name]
The local part of the value of `dm:node-name`.

[prefix]
The prefix associated with the value of `dm:node-name`.

[children]
A list of information items obtained by processing each of the `dm:children` in order and mapping each to the appropriate information item(s).

[attributes]
An unordered set of information items obtained by processing each of the `dm:attributes` and mapping each to the appropriate information item(s).

[in-scope namespaces]
An unordered set of **namespace information items** constructed from the **namespaces**.

Each in-scope namespace maps to a **namespace information item**. The following properties are specified by this mapping:

[prefix]
The prefix associated with the namespace.

[namespace name]
The URI associated with the namespace.

[base URI]
The value of `dm:base-uri`.

[parent]

- If this node is the root of the infoset mapping operation, *unknown*.
- If this node has a parent, the information item that corresponds to the node returned by `dm:parent`.

- Otherwise *no value*.

The following property has no value: **[namespace attributes]**.

K.3 Attribute Nodes Information Items

An Attribute Node maps to an **attribute information item**.

The following properties are specified by this mapping:

[namespace name]
The namespace name of the value of `dm:node-name`.

[local name]
The local part of the value of `dm:node-name`.

[prefix]
The prefix associated with the value of `dm:node-name`.

[normalized value]
The value of `dm:string-value`.

[owner element]

- If this node has a parent, the information item that corresponds to the node returned by `dm:parent`.
- Otherwise *no value*.

The following properties have no value: **[specified] [attribute type] [references]**.

K.4 Namespace Nodes Information Items

A Namespace Node maps to a **namespace information item**.

The following properties are specified by this mapping:

[prefix]
The prefix associated with the namespace.

[namespace name]
The value of `dm:string-value`.

K.5 Processing Instruction Nodes Information Items

An Processing Instruction Node maps to a **processing instruction information item**.

The following properties are specified by this mapping:

[target]
The local part of the value of `dm:node-name`.

[content]
The value of `dm:string-value`.

[base URI]
The value of `dm:base-uri`.

[parent]

- If this node is the root of the infoset mapping operation, *unknown*.
- If this node has a parent, the information item that corresponds to the node returned by `dm:parent`.
- Otherwise *no value*.

[notation]
no value.

K.6 Comment Nodes Information Items

A Comment Node maps to a **comment information item**.

The following properties are specified by this mapping:

[content]
The value of the `dm:string-value`.

[parent]

- If this node is the root of the infoset mapping operation, *unknown*.
- If this node has a parent, the information item that corresponds to the node returned by `dm:parent`.
- Otherwise *no value*.

K.7 Text Nodes Information Items

A Text Node maps to a sequence of **character information items**.

Each character of the `dm:string-value` of the node is converted into a **character information item** as specified by this mapping:

[character code]
The Unicode code point value of the character.

[parent]

- If this node is the root of the infoset mapping operation, *unknown*.
- If this node has a parent, the information item that corresponds to the node returned by `dm:parent`.
- Otherwise *no value*.

[element content whitespace]
Unknown.

This sequence of characters constitutes the infoset mapping.