XML Path Language (XPath) Version 1.0

W3C Recommendation 16 November 1999

This version:

http://www.w3.org/TR/1999/REC-xpath-19991116
(available in XML or HTML)

Latest version:

http://www.w3.org/TR/xpath

Previous versions:

http://www.w3.org/TR/1999/PR-xpath-19991008
http://www.w3.org/1999/08/WD-xpath-19990813
http://www.w3.org/1999/07/WD-xpath-19990709
http://www.w3.org/TR/1999/WD-xslt-19990421

Editors:

James Clark <jjc@jclark.com>
Steve DeRose (Inso Corp. and Brown University) <Steven_DeRose@Brown.edu>

Abstract

XPath is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer.

Status of this document

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

The list of known errors in this specification is available at http://www.w3.org/1999/11/REC-xpath-19991116-errata.

Comments on this specification may be sent to www-xpath-comments@w3.org; archives of the comments are available.

The English version of this specification is the only normative version. However, for translations of this document, see http://www.w3.org/Style/XSL/translations.html.

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

This specification is joint work of the XSL Working Group and the XML Linking Working Group and so is part of the W3C Style activity and of the W3C XML activity.

Table of contents

Appendices

---

1 Introduction

XPath is the result of an effort to provide a common syntax and semantics for functionality shared between XSL Transformations [XSLT] and XPointer [XPointer]. The primary purpose of XPath is to address parts of an XML [XML] document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers and booleans. XPath uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document.

In addition to its use for addressing, XPath is also designed so that it has a natural subset that can be used for matching (testing whether or not a node matches a pattern); this use of XPath is described in XSLT.

XPath models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes. XPath defines a way to compute a string-value for each type of node. Some types of nodes also have names. XPath fully supports XML Namespaces [XML Names]. Thus, the name of a node is modeled as a pair consisting of a local part and a possibly null namespace URI; this is called an expanded-name. The data model is described in detail in **[5 Data Model]**.

The primary syntactic construct in XPath is the expression. An expression matches the production Expr. An expression is evaluated to yield an object, which has one of the following four basic types:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)

Expression evaluation occurs with respect to a context. XSLT and XPointer specify how the context is determined for XPath expressions used in XSLT and XPointer respectively. The context consists of:

- a node (the **context node**)
- a pair of non-zero positive integers (the **context position** and the **context size**)
- a set of variable bindings
- a function library
- the set of namespace declarations in scope for the expression

The context position is always less than or equal to the context size.

The variable bindings consist of a mapping from variable names to variable values. The value of a variable is an object, which can be of any of the types that are possible for the value of an expression, and may also be of additional types not specified here.

The function library consists of a mapping from function names to functions. Each function takes zero or more arguments and returns a single result. This document defines a core function library that all XPath implementations must support (see **[4 Core Function Library]**). For a function in the core function library, arguments and result are of the four basic types. Both XSLT and XPointer extend XPath by defining additional functions; some of these functions operate on the four basic types; others operate on additional data types defined by XSLT and XPointer.

The namespace declarations consist of a mapping from prefixes to namespace URIs.

The variable bindings, function library and namespace declarations used to evaluate a subexpression are always the same as those used to evaluate the containing expression. The context node, context position, and context size used to evaluate a subexpression are sometimes different from those used to evaluate the containing expression. Several kinds of expressions change the context node; only predicates change the context position and context size (see **[2.4 Predicates]**). When the evaluation of a kind of expression is described, it will always be explicitly stated if the context node, context position, and context size change for the evaluation of subexpressions; if nothing is said about the context node, context position, and context size, they remain unchanged for the evaluation of subexpressions of that kind of expression.

XPath expressions often occur in XML attributes. The grammar specified in this section applies to the attribute value after XML 1.0 normalization. So, for example, if the grammar uses the character <, this must not appear in the XML source as < but must be quoted according to XML 1.0 rules by, for example, entering it as `<`. Within expressions, literal strings are delimited by single or double quotation marks, which are also used to delimit XML attributes. To avoid a quotation mark in an expression being interpreted by the XML processor as terminating the attribute value the quotation mark can be entered as a character reference (`&quot;` or `&apos;`). Alternatively, the expression can use single quotation marks if the XML attribute is delimited with double quotation marks or vice-versa.

One important kind of expression is a location path. A location path selects a set of nodes relative to the context node. The result of evaluating an expression that is a location path is the node-set containing the nodes selected by the location path. Location paths can recursively contain expressions that are used to filter sets of nodes. A location path matches the production LocationPath.

In the following grammar, the non-terminals QName and NCName are defined in [XML Names], and S is defined in [XML]. The grammar uses the same EBNF notation as [XML] (except that grammar symbols always have initial capital letters).

Expressions are parsed by first dividing the character string to be parsed into tokens and then parsing the resulting sequence of tokens. Whitespace can be freely used between tokens. The tokenization process is described in **[3.7 Lexical Structure]**.

2 Location Paths

Although location paths are not the most general grammatical construct in the language (a LocationPath is a special case of an Expr), they are the most important construct and will therefore be described first.

Every location path can be expressed using a straightforward but rather verbose syntax. There are also a number of syntactic abbreviations that allow common cases to be expressed concisely. This section will explain the semantics of location paths using the unabbreviated syntax. The abbreviated syntax will then be explained by showing how it expands into the unabbreviated syntax (see **[2.5 Abbreviated Syntax]**).

Here are some examples of location paths using the unabbreviated syntax:

- `child::para` selects the `para` element children of the context node
- `child::*` selects all element children of the context node
- `child::text()` selects all text node children of the context node
- `child::node()` selects all the children of the context node, whatever their node type
- `attribute::name` selects the `name` attribute of the context node
- `attribute::*` selects all the attributes of the context node
- `descendant::para` selects the `para` element descendants of the context node
- `ancestor::div` selects all `div` ancestors of the context node
- `ancestor-or-self::div` selects the `div` ancestors of the context node and, if the context node is a `div` element, the context node as well
- `descendant-or-self::para` selects the `para` element descendants of the context node and, if the context node is a `para` element, the context node as well
- `self::para` selects the context node if it is a `para` element, and otherwise selects nothing
- `child::chapter/descendant::para` selects the `para` element descendants of the `chapter` element children of the context node
- `child::*/child::para` selects all `para` grandchildren of the context node
- `/` selects the document root (which is always the parent of the document element)
- `/descendant::para` selects all the `para` elements in the same document as the context node
- `/descendant::olist/child::item` selects all the `item` elements that have an `olist` parent and that are in the same document as the context node
- `child::para[position()=1]` selects the first `para` child of the context node
- `child::para[position()=last()]` selects the last `para` child of the context node
- `child::para[position()=last()-1]` selects the last but one `para` child of the context node
- `child::para[position()>1]` selects all the `para` children of the context node other than the

first `para` child of the context node

- `following-sibling::chapter[position()=1]` selects the next `chapter` sibling of the context node
- `preceding-sibling::chapter[position()=1]` selects the previous `chapter` sibling of the context node
- `/descendant::figure[position()=42]` selects the forty-second `figure` element in the document
- `/child::doc/child::chapter[position()=5]/child::section[position()=2]` selects the second `section` of the fifth `chapter` of the `doc` document element
- `child::para[attribute::type="warning"]` selects all `para` children of the context node that have a `type` attribute with value `warning`
- `child::para[attribute::type='warning'][position()=5]` selects the fifth `para` child of the context node that has a `type` attribute with value `warning`
- `child::para[position()=5][attribute::type="warning"]` selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning`
- `child::chapter[child::title='Introduction']` selects the `chapter` children of the context node that have one or more `title` children with string-value equal to `Introduction`
- `child::chapter[child::title]` selects the `chapter` children of the context node that have one or more `title` children
- `child::*[self::chapter or self::appendix]` selects the `chapter` and `appendix` children of the context node
- `child::*[self::chapter or self::appendix][position()=last()]` selects the last `chapter` or `appendix` child of the context node

There are two kinds of location path: relative location paths and absolute location paths.

A relative location path consists of a sequence of one or more location steps separated by `/`. The steps in a relative location path are composed together from left to right. Each step in turn selects a set of nodes relative to a context node. An initial sequence of steps is composed together with a following step as follows. The initial sequence of steps selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are unioned together. The set of nodes identified by the composition of the steps is this union. For example, `child::div/child::para` selects the `para` element children of the `div` element children of the context node, or, in other words, the `para` element grandchildren that have `div` parents.

An absolute location path consists of `/` optionally followed by a relative location path. A `/` by itself selects the root node of the document containing the context node. If it is followed by a relative location path, then the location path selects the set of nodes that would be selected by the relative location path relative to the root node of the document containing the context node.

Location Paths

[1]	LocationPath	::=	RelativeLocationPath
			| AbsoluteLocationPath
[2]	AbsoluteLocationPath	::=	'/' RelativeLocationPath?
			| AbbreviatedAbsoluteLocationPath
[3]	RelativeLocationPath	::=	Step
			| RelativeLocationPath '/' Step
			| AbbreviatedRelativeLocationPath

2.1 Location Steps

A location step has three parts:

- an axis, which specifies the tree relationship between the nodes selected by the location step and the context node,
- a node test, which specifies the node type and expanded-name of the nodes selected by the location step, and
- zero or more predicates, which use arbitrary expressions to further refine the set of nodes selected by the location step.

The syntax for a location step is the axis name and node test separated by a double colon, followed by zero or more expressions each in square brackets. For example, in `child::para[position()=1]`, `child` is the name of the axis, `para` is the node test and `[position()=1]` is a predicate.

The node-set selected by the location step is the node-set that results from generating an initial node-set from the axis and node-test, and then filtering that node-set by each of the predicates in turn.

The initial node-set consists of the nodes having the relationship to the context node specified by the axis, and having the node type and expanded-name specified by the node test. For example, a location step `descendant::para` selects the `para` element descendants of the context node: `descendant` specifies that each node in the initial node-set must be a descendant of the context; `para` specifies that each node in the initial node-set must be an element named `para`. The available axes are described in **[2.2 Axes]**. The available node tests are described in **[2.3 Node Tests]**. The meaning of some node tests is dependent on the axis.

The initial node-set is filtered by the first predicate to generate a new node-set; this new node-set is then filtered using the second predicate, and so on. The final node-set is the node-set selected by the location step. The axis affects how the expression in each predicate is evaluated and so the semantics of a predicate is defined with respect to an axis. See **[2.4 Predicates]**.

Location Steps

[4]	Step	::=	AxisSpecifier NodeTest Predicate*
			| AbbreviatedStep
[5]	AxisSpecifier	::=	AxisName '::'
			| AbbreviatedAxisSpecifier

2.2 Axes

The following axes are available:

- the `child` axis contains the children of the context node
- the `descendant` axis contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes
- the `parent` axis contains the parent of the context node, if there is one
- the `ancestor` axis contains the ancestors of the context node; the ancestors of the context node consist of the parent of context node and the parent's parent and so on; thus, the ancestor axis will always include the root node, unless the context node is the root node
- the `following-sibling` axis contains all the following siblings of the context node; if the context node is an attribute node or namespace node, the `following-sibling` axis is empty
- the `preceding-sibling` axis contains all the preceding siblings of the context node; if the context node is an attribute node or namespace node, the `preceding-sibling` axis is empty
- the `following` axis contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes
- the `preceding` axis contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and excluding attribute nodes and namespace nodes
- the `attribute` axis contains the attributes of the context node; the axis will be empty unless the context node is an element
- the `namespace` axis contains the namespace nodes of the context node; the axis will be empty unless the context node is an element
- the `self` axis contains just the context node itself
- the `descendant-or-self` axis contains the context node and the descendants of the context node
- the `ancestor-or-self` axis contains the context node and the ancestors of the context node; thus, the ancestor axis will always include the root node

  NOTE: The `ancestor`, `descendant`, `following`, `preceding` and `self` axes partition a document (ignoring attribute and namespace nodes): they do not overlap and together they contain all the nodes in the document.

Axes

```
[6]   AxisName   ::=   'ancestor'
                     | 'ancestor-or-self'
                     | 'attribute'
                     | 'child'
                     | 'descendant'
```

| 'descendant-or-self'
| 'following'
| 'following-sibling'
| 'namespace'
| 'parent'
| 'preceding'
| 'preceding-sibling'
| 'self'

2.3 Node Tests

Every axis has a **principal node type**. If an axis can contain elements, then the principal node type is element; otherwise, it is the type of the nodes that the axis can contain. Thus,

- For the attribute axis, the principal node type is attribute.
- For the namespace axis, the principal node type is namespace.
- For other axes, the principal node type is element.

A node test that is a QName is true if and only if the type of the node (see **[5 Data Model]**) is the principal node type and has an expanded-name equal to the expanded-name specified by the QName. For example, `child::para` selects the `para` element children of the context node; if the context node has no `para` children, it will select an empty set of nodes. `attribute::href` selects the `href` attribute of the context node; if the context node has no `href` attribute, it will select an empty set of nodes.

A QName in the node test is expanded into an expanded-name using the namespace declarations from the expression context. This is the same way expansion is done for element type names in start and end-tags except that the default namespace declared with `xmlns` is not used: if the QName does not have a prefix, then the namespace URI is null (this is the same way attribute names are expanded). It is an error if the QName has a prefix for which there is no namespace declaration in the expression context.

A node test `*` is true for any node of the principal node type. For example, `child::*` will select all element children of the context node, and `attribute::*` will select all attributes of the context node.

A node test can have the form NCName`:*`. In this case, the prefix is expanded in the same way as with a QName, using the context namespace declarations. It is an error if there is no namespace declaration for the prefix in the expression context. The node test will be true for any node of the principal type whose expanded-name has the namespace URI to which the prefix expands, regardless of the local part of the name.

The node test `text()` is true for any text node. For example, `child::text()` will select the text node children of the context node. Similarly, the node test `comment()` is true for any comment node, and the node test `processing-instruction()` is true for any processing instruction. The `processing-instruction()` test may have an argument that is Literal; in this case, it is true for any processing instruction that has a name equal to the value of the Literal.

A node test `node()` is true for any node of any type whatsoever.

[7] NodeTest ::= NameTest
| NodeType '(' ')'

| 'processing-instruction' '(' Literal ')'

2.4 Predicates

An axis is either a forward axis or a reverse axis. An axis that only ever contains the context node or nodes that are after the context node in document order is a forward axis. An axis that only ever contains the context node or nodes that are before the context node in document order is a reverse axis. Thus, the ancestor, ancestor-or-self, preceding, and preceding-sibling axes are reverse axes; all other axes are forward axes. Since the self axis always contains at most one node, it makes no difference whether it is a forward or reverse axis. The **proximity position** of a member of a node-set with respect to an axis is defined to be the position of the node in the node-set ordered in document order if the axis is a forward axis and ordered in reverse document order if the axis is a reverse axis. The first position is 1.

A predicate filters a node-set with respect to an axis to produce a new node-set. For each node in the node-set to be filtered, the PredicateExpr is evaluated with that node as the context node, with the number of nodes in the node-set as the context size, and with the proximity position of the node in the node-set with respect to the axis as the context position; if PredicateExpr evaluates to true for that node, the node is included in the new node-set; otherwise, it is not included.

A PredicateExpr is evaluated by evaluating the Expr and converting the result to a boolean. If the result is a number, the result will be converted to true if the number is equal to the context position and will be converted to false otherwise; if the result is not a number, then the result will be converted as if by a call to the **boolean** function. Thus a location path `para[3]` is equivalent to `para[position()=3]`.

Predicates

[8]	Predicate	::=	'[' PredicateExpr ']'
[9]	PredicateExpr	::=	Expr

2.5 Abbreviated Syntax

Here are some examples of location paths using abbreviated syntax:

- `para` selects the `para` element children of the context node
- `*` selects all element children of the context node
- `text()` selects all text node children of the context node
- `@name` selects the `name` attribute of the context node
- `@*` selects all the attributes of the context node
- `para[1]` selects the first `para` child of the context node
- `para[last()]` selects the last `para` child of the context node
- `*/para` selects all `para` grandchildren of the context node
- `/doc/chapter[5]/section[2]` selects the second `section` of the fifth `chapter` of the `doc`

- `chapter//para` selects the `para` element descendants of the `chapter` element children of the context node
- `//para` selects all the `para` descendants of the document root and thus selects all `para` elements in the same document as the context node
- `//olist/item` selects all the `item` elements in the same document as the context node that have an `olist` parent
- `.` selects the context node
- `.//para` selects the `para` element descendants of the context node
- `..` selects the parent of the context node
- `../@lang` selects the `lang` attribute of the parent of the context node
- `para[@type="warning"]` selects all `para` children of the context node that have a `type` attribute with value `warning`
- `para[@type="warning"][5]` selects the fifth `para` child of the context node that has a `type` attribute with value `warning`
- `para[5][@type="warning"]` selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning`
- `chapter[title="Introduction"]` selects the `chapter` children of the context node that have one or more `title` children with string-value equal to `Introduction`
- `chapter[title]` selects the `chapter` children of the context node that have one or more `title` children
- `employee[@secretary and @assistant]` selects all the `employee` children of the context node that have both a `secretary` attribute and an `assistant` attribute

The most important abbreviation is that `child::` can be omitted from a location step. In effect, `child` is the default axis. For example, a location path `div/para` is short for `child::div/child::para`.

There is also an abbreviation for attributes: `attribute::` can be abbreviated to @. For example, a location path `para[@type="warning"]` is short for `child::para[attribute::type="warning"]` and so selects `para` children with a `type` attribute with value equal to `warning`.

`//` is short for `/descendant-or-self::node()/`. For example, `//para` is short for `/descendant-or-self::node()/child::para` and so will select any `para` element in the document (even a `para` element that is a document element will be selected by `//para` since the document element node is a child of the root node); `div//para` is short for `div/descendant-or-self::node()/child::para` and so will select all `para` descendants of `div` children.

> **NOTE:** The location path `//para[1]` does *not* mean the same as the location path `/descendant::para[1]`. The latter selects the first descendant `para` element; the former selects all descendant `para` elements that are the first `para` children of their parents.

A location step of `.` is short for `self::node()`. This is particularly useful in conjunction with `//`. For

example, the location path `.//para` is short for
`self::node()/descendant-or-self::node()/child::para`

and so will select all `para` descendant elements of the context node.

Similarly, a location step of `..` is short for `parent::node()`. For example, `../title` is short for `parent::node()/child::title` and so will select the `title` children of the parent of the context node.

Abbreviations

[10]	AbbreviatedAbsoluteLocationPath	::=	'//' RelativeLocationPath
[11]	AbbreviatedRelativeLocationPath	::=	RelativeLocationPath '//' Step
[12]	AbbreviatedStep	::=	'.'
			| '..'
[13]	AbbreviatedAxisSpecifier	::=	'@'?

3 Expressions

3.1 Basics

A VariableReference evaluates to the value to which the variable name is bound in the set of variable bindings in the context. It is an error if the variable name is not bound to any value in the set of variable bindings in the expression context.

Parentheses may be used for grouping.

[14]	Expr	::=	OrExpr
[15]	PrimaryExpr	::=	VariableReference
			| '(' Expr ')'
			| Literal
			| Number
			| FunctionCall

3.2 Function Calls

A FunctionCall expression is evaluated by using the FunctionName to identify a function in the expression evaluation context function library, evaluating each of the Arguments, converting each argument to the type required by the function, and finally calling the function, passing it the converted arguments. It is an error if the number of arguments is wrong or if an argument cannot be converted to the required type. The result of the FunctionCall expression is the result returned by the function.

An argument is converted to type string as if by calling the **string** function. An argument is converted to type number as if by calling the **number** function. An argument is converted to type boolean as if by calling the **boolean** function. An argument that is not of type node-set cannot be converted to a node-set.

[16]	FunctionCall	::=	FunctionName '(' (Argument (',' Argument)*)? ')'

[17] Argument ::= Expr

3.3 Node-sets

A location path can be used as an expression. The expression returns the set of nodes selected by the path.

The | operator computes the union of its operands, which must be node-sets.

Predicates are used to filter expressions in the same way that they are used in location paths. It is an error if the expression to be filtered does not evaluate to a node-set. The Predicate filters the node-set with respect to the child axis.

> **NOTE:** The meaning of a Predicate depends crucially on which axis applies. For example, `preceding::foo[1]` returns the first `foo` element in *reverse document order*, because the axis that applies to the `[1]` predicate is the preceding axis; by contrast, `(preceding::foo)[1]` returns the first `foo` element in *document order*, because the axis that applies to the `[1]` predicate is the child axis.

The `/` and `//` operators compose an expression and a relative location path. It is an error if the expression does not evaluate to a node-set. The `/` operator does composition in the same way as when `/` is used in a location path. As in location paths, `//` is short for `/descendant-or-self::node()/`.

There are no types of objects that can be converted to node-sets.

[18] UnionExpr ::= PathExpr
| UnionExpr '|' PathExpr

[19] PathExpr ::= LocationPath
| FilterExpr
| FilterExpr '/' RelativeLocationPath
| FilterExpr '//' RelativeLocationPath

[20] FilterExpr ::= PrimaryExpr
| FilterExpr Predicate

3.4 Booleans

An object of type boolean can have one of two values, true and false.

An `or` expression is evaluated by evaluating each operand and converting its value to a boolean as if by a call to the **boolean** function. The result is true if either value is true and false otherwise. The right operand is not evaluated if the left operand evaluates to true.

An `and` expression is evaluated by evaluating each operand and converting its value to a boolean as if by a call to the **boolean** function. The result is true if both values are true and false otherwise. The right operand is not evaluated if the left operand evaluates to false.

An EqualityExpr (that is not just a RelationalExpr) or a RelationalExpr (that is not just an AdditiveExpr) is evaluated by comparing the objects that result from evaluating the two operands. Comparison of the resulting objects is defined in the following three paragraphs. First, comparisons that involve node-sets are defined in terms of comparisons that do not involve node-sets; this is defined uniformly for =, !=, <=, <, >= and >.

Second, comparisons that do not involve node-sets are defined for = and !=. Third, comparisons that do not involve node-sets are defined for <=, <, >= and >.

If both objects to be compared are node-sets, then the comparison will be true if and only if there is a node in the first node-set and a node in the second node-set such that the result of performing the comparison on the string-values of the two nodes is true. If one object to be compared is a node-set and the other is a number, then the comparison will be true if and only if there is a node in the node-set such that the result of performing the comparison on the number to be compared and on the result of converting the string-value of that node to a number using the **number** function is true. If one object to be compared is a node-set and the other is a string, then the comparison will be true if and only if there is a node in the node-set such that the result of performing the comparison on the string-value of the node and the other string is true. If one object to be compared is a node-set and the other is a boolean, then the comparison will be true if and only if the result of performing the comparison on the boolean and on the result of converting the node-set to a boolean using the **boolean** function is true.

When neither object to be compared is a node-set and the operator is = or !=, then the objects are compared by converting them to a common type as follows and then comparing them. If at least one object to be compared is a boolean, then each object to be compared is converted to a boolean as if by applying the **boolean** function. Otherwise, if at least one object to be compared is a number, then each object to be compared is converted to a number as if by applying the **number** function. Otherwise, both objects to be compared are converted to strings as if by applying the **string** function. The = comparison will be true if and only if the objects are equal; the != comparison will be true if and only if the objects are not equal. Numbers are compared for equality according to IEEE 754 [IEEE 754]. Two booleans are equal if either both are true or both are false. Two strings are equal if and only if they consist of the same sequence of UCS characters.

> **NOTE:** If `$x` is bound to a node-set, then `$x="foo"` does not mean the same as `not($x!="foo")`: the former is true if and only if *some* node in `$x` has the string-value `foo`; the latter is true if and only if *all* nodes in `$x` have the string-value `foo`.

When neither object to be compared is a node-set and the operator is <=, <, >= or >, then the objects are compared by converting both objects to numbers and comparing the numbers according to IEEE 754. The < comparison will be true if and only if the first number is less than the second number. The <= comparison will be true if and only if the first number is less than or equal to the second number. The > comparison will be true if and only if the first number is greater than the second number. The >= comparison will be true if and only if the first number is greater than or equal to the second number.

> **NOTE:** When an XPath expression occurs in an XML document, any < and <= operators must be quoted according to XML 1.0 rules by using, for example, `<` and `<=`. In the following example the value of the `test` attribute is an XPath expression:
>
> ```
> <xsl:if test="@value < 10">...</xsl:if>
> ```

[21]	OrExpr	::=	AndExpr
			| OrExpr 'or' AndExpr
[22]	AndExpr	::=	EqualityExpr
			| AndExpr 'and' EqualityExpr
[23]	EqualityExpr	::=	RelationalExpr
			| EqualityExpr '=' RelationalExpr
			| EqualityExpr '!=' RelationalExpr

[24] RelationalExpr ::= AdditiveExpr
| RelationalExpr '<' AdditiveExpr
| RelationalExpr '>' AdditiveExpr
| RelationalExpr '<=' AdditiveExpr
| RelationalExpr '>=' AdditiveExpr

NOTE: The effect of the above grammar is that the order of precedence is (lowest precedence first):

- `or`
- `and`
- `=, !=`
- `<=, <, >=, >`

and the operators are all left associative. For example, `3 > 2 > 1` is equivalent to `(3 > 2) > 1`, which evaluates to false.

3.5 Numbers

A number represents a floating-point number. A number can have any double-precision 64-bit format IEEE 754 value [IEEE 754]. These include a special "Not-a-Number" (NaN) value, positive and negative infinity, and positive and negative zero. See Section 4.2.3 of [JLS] for a summary of the key rules of the IEEE 754 standard.

The numeric operators convert their operands to numbers as if by calling the **number** function.

The `+` operator performs addition.

The `-` operator performs subtraction.

NOTE: Since XML allows `-` in names, the `-` operator typically needs to be preceded by whitespace. For example, `foo-bar` evaluates to a node-set containing the child elements named `foo-bar`; `foo - bar` evaluates to the difference of the result of converting the string-value of the first `foo` child element to a number and the result of converting the string-value of the first `bar` child to a number.

The `div` operator performs floating-point division according to IEEE 754.

The `mod` operator returns the remainder from a truncating division. For example,

- `5 mod 2` returns `1`
- `5 mod -2` returns `1`
- `-5 mod 2` returns `-1`
- `-5 mod -2` returns `-1`

NOTE: This is the same as the % operator in Java and ECMAScript.

NOTE: This is not the same as the IEEE 754 remainder operation, which returns the remainder

from a rounding division.

Numeric Expressions

[25]	AdditiveExpr	::=	MultiplicativeExpr
			| AdditiveExpr '+' MultiplicativeExpr
			| AdditiveExpr '-' MultiplicativeExpr
[26]	MultiplicativeExpr	::=	UnaryExpr
			| MultiplicativeExpr MultiplyOperator UnaryExpr
			| MultiplicativeExpr 'div' UnaryExpr
			| MultiplicativeExpr 'mod' UnaryExpr
[27]	UnaryExpr	::=	UnionExpr
			| '-' UnaryExpr

3.6 Strings

Strings consist of a sequence of zero or more characters, where a character is defined as in the XML Recommendation [XML]. A single character in XPath thus corresponds to a single Unicode abstract character with a single corresponding Unicode scalar value (see [Unicode]); this is not the same thing as a 16-bit Unicode code value: the Unicode coded character representation for an abstract character with Unicode scalar value greater that U+FFFF is a pair of 16-bit Unicode code values (a surrogate pair). In many programming languages, a string is represented by a sequence of 16-bit Unicode code values; implementations of XPath in such languages must take care to ensure that a surrogate pair is correctly treated as a single XPath character.

> **NOTE:** It is possible in Unicode for there to be two strings that should be treated as identical even though they consist of the distinct sequences of Unicode abstract characters. For example, some accented characters may be represented in either a precomposed or decomposed form. Therefore, XPath expressions may return unexpected results unless both the characters in the XPath expression and in the XML document have been normalized into a canonical form. See [Character Model].

3.7 Lexical Structure

When tokenizing, the longest possible token is always returned.

For readability, whitespace may be used in expressions even though not explicitly allowed by the grammar: ExprWhitespace may be freely added within patterns before or after any ExprToken.

The following special tokenization rules must be applied in the order specified to disambiguate the ExprToken grammar:

- If there is a preceding token and the preceding token is not one of `@`, `::`, `(`, `[`, `,` or an Operator, then a `*` must be recognized as a MultiplyOperator and an NCName must be recognized as an OperatorName.
- If the character following an NCName (possibly after intervening ExprWhitespace) is `(`, then the token must be recognized as a NodeType or a FunctionName.

- If the two characters following an NCName (possibly after intervening ExprWhitespace) are `::`, then the token must be recognized as an AxisName.
- Otherwise, the token must not be recognized as a MultiplyOperator, an OperatorName, a NodeType, a FunctionName, or an AxisName.

Expression Lexical Structure

[28]	ExprToken	::=	'(' | ')' | '[' | ']' | '.' | '..' | '@' | ',' | '::' | NameTest | NodeType | Operator | FunctionName | AxisName | Literal | Number | VariableReference
[29]	Literal	::=	'"' [^"]* '"' | "'" [^']* "'"
[30]	Number	::=	Digits ('.' Digits?)? | '.' Digits
[31]	Digits	::=	[0-9]+
[32]	Operator	::=	OperatorName | MultiplyOperator | '/' | '//' | '|' | '+' | '-' | '=' | '!=' | '<' | '<=' | '>' | '>='
[33]	OperatorName	::=	'and' | 'or' | 'mod' | 'div'
[34]	MultiplyOperator	::=	'*'
[35]	FunctionName	::=	QName - NodeType
[36]	VariableReference	::=	'$' QName
[37]	NameTest	::=	'*' | NCName ':' '*' | QName
[38]	NodeType	::=	'comment' | 'text' | 'processing-instruction' | 'node'
[39]	ExprWhitespace	::=	S

4 Core Function Library

This section describes functions that XPath implementations must always include in the function library that is used to evaluate expressions.

Each function in the function library is specified using a function prototype, which gives the return type, the name of the function, and the type of the arguments. If an argument type is followed by a question mark, then the argument is optional; otherwise, the argument is required.

4.1 Node Set Functions

Function: *number* **last**()

The **last** function returns a number equal to the context size from the expression evaluation context.

Function: *number* **position**()

The **position** function returns a number equal to the context position from the expression evaluation context.

Function: *number* **count**(*node-set*)

The **count** function returns the number of nodes in the argument node-set.

Function: *node-set* **id**(*object*)

The **id** function selects elements by their unique ID (see [**5.2.1 Unique IDs**]). When the argument to **id** is of type node-set, then the result is the union of the result of applying **id** to the string-value of each of the nodes in the argument node-set. When the argument to **id** is of any other type, the argument is converted to a string as if by a call to the **string** function; the string is split into a whitespace-separated list of tokens (whitespace is any sequence of characters matching the production S); the result is a node-set containing the elements in the same document as the context node that have a unique ID equal to any of the tokens in the list.

- `id("foo")` selects the element with unique ID `foo`
- `id("foo")/child::para[position()=5]` selects the fifth `para` child of the element with unique ID `foo`

Function: *string* **local-name**(*node-set*?)

The **local-name** function returns the local part of the expanded-name of the node in the argument node-set that is first in document order. If the argument node-set is empty or the first node has no expanded-name, an empty string is returned. If the argument is omitted, it defaults to a node-set with the context node as its only member.

Function: *string* **namespace-uri**(*node-set*?)

The **namespace-uri** function returns the namespace URI of the expanded-name of the node in the argument node-set that is first in document order. If the argument node-set is empty, the first node has no expanded-name, or the namespace URI of the expanded-name is null, an empty string is returned. If the argument is omitted, it defaults to a node-set with the context node as its only member.

> **NOTE:** The string returned by the **namespace-uri** function will be empty except for element nodes and attribute nodes.

Function: *string* **name**(*node-set*?)

The **name** function returns a string containing a QName representing the expanded-name of the node in the

argument node-set that is first in document order. The QName must represent the expanded-name with respect to the namespace declarations in effect on the node whose expanded-name is being represented. Typically, this will be the QName that occurred in the XML source. This need not be the case if there are namespace declarations in effect on the node that associate multiple prefixes with the same namespace. However, an implementation may include information about the original prefix in its representation of nodes; in this case, an implementation can ensure that the returned string is always the same as the QName used in the XML source. If the argument node-set is empty or the first node has no expanded-name, an empty string is returned. If the argument it omitted, it defaults to a node-set with the context node as its only member.

> **NOTE:** The string returned by the **name** function will be the same as the string returned by the **local-name** function except for element nodes and attribute nodes.

4.2 String Functions

Function: *string* **string**(*object*?)

The **string** function converts an object to a string as follows:

- A node-set is converted to a string by returning the string-value of the node in the node-set that is first in document order. If the node-set is empty, an empty string is returned.
- A number is converted to a string as follows
  - NaN is converted to the string `NaN`
  - positive zero is converted to the string `0`
  - negative zero is converted to the string `0`
  - positive infinity is converted to the string `Infinity`
  - negative infinity is converted to the string `-Infinity`
  - if the number is an integer, the number is represented in decimal form as a Number with no decimal point and no leading zeros, preceded by a minus sign (`-`) if the number is negative
  - otherwise, the number is represented in decimal form as a Number including a decimal point with at least one digit before the decimal point and at least one digit after the decimal point, preceded by a minus sign (`-`) if the number is negative; there must be no leading zeros before the decimal point apart possibly from the one required digit immediately before the decimal point; beyond the one required digit after the decimal point there must be as many, but only as many, more digits as are needed to uniquely distinguish the number from all other IEEE 754 numeric values.
- The boolean false value is converted to the string `false`. The boolean true value is converted to the string `true`.
- An object of a type other than the four basic types is converted to a string in a way that is dependent on that type.

If the argument is omitted, it defaults to a node-set with the context node as its only member.

> **NOTE:** The `string` function is not intended for converting numbers into strings for presentation to users. The `format-number` function and `xsl:number` element in [XSLT] provide this functionality.

Function: *string* **concat**(*string*, *string*, *string**)

The **concat** function returns the concatenation of its arguments.

Function: *boolean* **starts-with**(*string*, *string*)

The **starts-with** function returns true if the first argument string starts with the second argument string, and otherwise returns false.

Function: *boolean* **contains**(*string*, *string*)

The **contains** function returns true if the first argument string contains the second argument string, and otherwise returns false.

Function: *string* **substring-before**(*string*, *string*)

The **substring-before** function returns the substring of the first argument string that precedes the first occurrence of the second argument string in the first argument string, or the empty string if the first argument string does not contain the second argument string. For example, `substring-before("1999/04/01","/")` returns `1999`.

Function: *string* **substring-after**(*string*, *string*)

The **substring-after** function returns the substring of the first argument string that follows the first occurrence of the second argument string in the first argument string, or the empty string if the first argument string does not contain the second argument string. For example, `substring-after("1999/04/01","/")` returns `04/01`, and `substring-after("1999/04/01","19")` returns `99/04/01`.

Function: *string* **substring**(*string*, *number*, *number*?)

The **substring** function returns the substring of the first argument starting at the position specified in the second argument with length specified in the third argument. For example, `substring("12345",2,3)` returns `"234"`. If the third argument is not specified, it returns the substring starting at the position specified in the second argument and continuing to the end of the string. For example, `substring("12345",2)` returns `"2345"`.

More precisely, each character in the string (see **[3.6 Strings]**) is considered to have a numeric position: the position of the first character is 1, the position of the second character is 2 and so on.

> **NOTE:** This differs from Java and ECMAScript, in which the `String.substring` method treats the position of the first character as 0.

The returned substring contains those characters for which the position of the character is greater than or equal to the rounded value of the second argument and, if the third argument is specified, less than the sum of the rounded value of the second argument and the rounded value of the third argument; the comparisons and addition used for the above follow the standard IEEE 754 rules; rounding is done as if by a call to the **round** function. The following examples illustrate various unusual cases:

- `substring("12345", 1.5, 2.6)` returns `"234"`
- `substring("12345", 0, 3)` returns `"12"`
- `substring("12345", 0 div 0, 3)` returns `""`
- `substring("12345", 1, 0 div 0)` returns `""`
- `substring("12345", -42, 1 div 0)` returns `"12345"`
- `substring("12345", -1 div 0, 1 div 0)` returns `""`

Function: *number* **string-length**(*string*?)

The **string-length** returns the number of characters in the string (see **[3.6 Strings]**). If the argument is omitted, it defaults to the context node converted to a string, in other words the string-value of the context node.

Function: *string* **normalize-space**(*string*?)

The **normalize-space** function returns the argument string with whitespace normalized by stripping leading and trailing whitespace and replacing sequences of whitespace characters by a single space. Whitespace characters are the same as those allowed by the S production in XML. If the argument is omitted, it defaults to the context node converted to a string, in other words the string-value of the context node.

Function: *string* **translate**(*string*, *string*, *string*)

The **translate** function returns the first argument string with occurrences of characters in the second argument string replaced by the character at the corresponding position in the third argument string. For example, `translate("bar","abc","ABC")` returns the string `BAr`. If there is a character in the second argument string with no character at a corresponding position in the third argument string (because the second argument string is longer than the third argument string), then occurrences of that character in the first argument string are removed. For example, `translate("--aaa--","abc-","ABC")` returns `"AAA"`. If a character occurs more than once in the second argument string, then the first occurrence determines the replacement character. If the third argument string is longer than the second argument string, then excess characters are ignored.

> **NOTE:** The **translate** function is not a sufficient solution for case conversion in all languages. A future version of XPath may provide additional functions for case conversion.

4.3 Boolean Functions

Function: *boolean* **boolean**(*object*)

The **boolean** function converts its argument to a boolean as follows:

- a number is true if and only if it is neither positive or negative zero nor NaN
- a node-set is true if and only if it is non-empty
- a string is true if and only if its length is non-zero
- an object of a type other than the four basic types is converted to a boolean in a way that is dependent

on that type

Function: *boolean* **not**(*boolean*)

The **not** function returns true if its argument is false, and false otherwise.

Function: *boolean* **true**()

The **true** function returns true.

Function: *boolean* **false**()

The **false** function returns false.

Function: *boolean* **lang**(*string*)

The **lang** function returns true or false depending on whether the language of the context node as specified by `xml:lang` attributes is the same as or is a sublanguage of the language specified by the argument string. The language of the context node is determined by the value of the `xml:lang` attribute on the context node, or, if the context node has no `xml:lang` attribute, by the value of the `xml:lang` attribute on the nearest ancestor of the context node that has an `xml:lang` attribute. If there is no such attribute, then **lang** returns false. If there is such an attribute, then **lang** returns true if the attribute value is equal to the argument ignoring case, or if there is some suffix starting with `-` such that the attribute value is equal to the argument ignoring that suffix of the attribute value and ignoring case. For example, `lang("en")` would return true if the context node is any of these five elements:

```
<para xml:lang="en"/>
<div xml:lang="en"><para/></div>
<para xml:lang="EN"/>
<para xml:lang="en-us"/>
```

4.4 Number Functions

Function: *number* **number**(*object*?)

The **number** function converts its argument to a number as follows:

- a string that consists of optional whitespace followed by an optional minus sign followed by a Number followed by whitespace is converted to the IEEE 754 number that is nearest (according to the IEEE 754 round-to-nearest rule) to the mathematical value represented by the string; any other string is converted to NaN
- boolean true is converted to 1; boolean false is converted to 0
- a node-set is first converted to a string as if by a call to the **string** function and then converted in the same way as a string argument
- an object of a type other than the four basic types is converted to a number in a way that is dependent on that type

If the argument is omitted, it defaults to a node-set with the context node as its only member.

NOTE: The **number** function should not be used for conversion of numeric data occurring in an

element in an XML document unless the element is of a type that represents numeric data in a language-neutral format (which would typically be transformed into a language-specific format for presentation to a user). In addition, the **number** function cannot be used unless the language-neutral format used by the element is consistent with the XPath syntax for a Number.

Function: *number* **sum**(*node-set*)

The **sum** function returns the sum, for each node in the argument node-set, of the result of converting the string-values of the node to a number.

Function: *number* **floor**(*number*)

The **floor** function returns the largest (closest to positive infinity) number that is not greater than the argument and that is an integer.

Function: *number* **ceiling**(*number*)

The **ceiling** function returns the smallest (closest to negative infinity) number that is not less than the argument and that is an integer.

Function: *number* **round**(*number*)

The **round** function returns the number that is closest to the argument and that is an integer. If there are two such numbers, then the one that is closest to positive infinity is returned. If the argument is NaN, then NaN is returned. If the argument is positive infinity, then positive infinity is returned. If the argument is negative infinity, then negative infinity is returned. If the argument is positive zero, then positive zero is returned. If the argument is negative zero, then negative zero is returned. If the argument is less than zero, but greater than or equal to -0.5, then negative zero is returned.

> **NOTE:** For these last two cases, the result of calling the **round** function is not the same as the result of adding 0.5 and then calling the **floor** function.

5 Data Model

XPath operates on an XML document as a tree. This section describes how XPath models an XML document as a tree. This model is conceptual only and does not mandate any particular implementation. The relationship of this model to the XML Information Set [XML Infoset] is described in [**B XML Information Set Mapping**].

XML documents operated on by XPath must conform to the XML Namespaces Recommendation [XML Names].

The tree contains nodes. There are seven types of node:

- root nodes
- element nodes
- text nodes
- attribute nodes

- namespace nodes
- processing instruction nodes
- comment nodes

For every type of node, there is a way of determining a **string-value** for a node of that type. For some types of node, the string-value is part of the node; for other types of node, the string-value is computed from the string-value of descendant nodes.

> **NOTE:** For element nodes and root nodes, the string-value of a node is not the same as the string returned by the DOM `nodeValue` method (see [DOM]).

Some types of node also have an **expanded-name**, which is a pair consisting of a local part and a namespace URI. The local part is a string. The namespace URI is either null or a string. The namespace URI specified in the XML document can be a URI reference as defined in [RFC2396]; this means it can have a fragment identifier and can be relative. A relative URI should be resolved into an absolute URI during namespace processing: the namespace URIs of expanded-names of nodes in the data model should be absolute. Two expanded-names are equal if they have the same local part, and either both have a null namespace URI or both have non-null namespace URIs that are equal.

There is an ordering, **document order**, defined on all the nodes in the document corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities. Thus, the root node will be the first node. Element nodes occur before their children. Thus, document order orders element nodes in order of the occurrence of their start-tag in the XML (after expansion of entities). The attribute nodes and namespace nodes of an element occur before the children of the element. The namespace nodes are defined to occur before the attribute nodes. The relative order of namespace nodes is implementation-dependent. The relative order of attribute nodes is implementation-dependent. **Reverse document order** is the reverse of document order.

Root nodes and element nodes have an ordered list of child nodes. Nodes never share children: if one node is not the same node as another node, then none of the children of the one node will be the same node as any of the children of another node. Every node other than the root node has exactly one **parent**, which is either an element node or the root node. A root node or an element node is the parent of each of its child nodes. The **descendants** of a node are the children of the node and the descendants of the children of the node.

5.1 Root Node

The root node is the root of the tree. A root node does not occur except as the root of the tree. The element node for the document element is a child of the root node. The root node also has as children processing instruction and comment nodes for processing instructions and comments that occur in the prolog and after the end of the document element.

The string-value of the root node is the concatenation of the string-values of all text node descendants of the root node in document order.

The root node does not have an expanded-name.

5.2 Element Nodes

There is an element node for every element in the document. An element node has an expanded-name computed by expanding the QName of the element specified in the tag in accordance with the XML Namespaces Recommendation [XML Names]. The namespace URI of the element's expanded-name will be null if the QName has no prefix and there is no applicable default namespace.

> **NOTE:** In the notation of Appendix A.3 of [XML Names], the local part of the expanded-name corresponds to the `type` attribute of the `ExpEType` element; the namespace URI of the expanded-name corresponds to the `ns` attribute of the `ExpEType` element, and is null if the `ns` attribute of the `ExpEType` element is omitted.

The children of an element node are the element nodes, comment nodes, processing instruction nodes and text nodes for its content. Entity references to both internal and external entities are expanded. Character references are resolved.

The string-value of an element node is the concatenation of the string-values of all text node descendants of the element node in document order.

5.2.1 Unique IDs

An element node may have a unique identifier (ID). This is the value of the attribute that is declared in the DTD as type `ID`. No two elements in a document may have the same unique ID. If an XML processor reports two elements in a document as having the same unique ID (which is possible only if the document is invalid) then the second element in document order must be treated as not having a unique ID.

> **NOTE:** If a document does not have a DTD, then no element in the document will have a unique ID.

5.3 Attribute Nodes

Each element node has an associated set of attribute nodes; the element is the parent of each of these attribute nodes; however, an attribute node is not a child of its parent element.

> **NOTE:** This is different from the DOM, which does not treat the element bearing an attribute as the parent of the attribute (see [DOM]).

Elements never share attribute nodes: if one element node is not the same node as another element node, then none of the attribute nodes of the one element node will be the same node as the attribute nodes of another element node.

> **NOTE:** The = operator tests whether two nodes have the same value, *not* whether they are the same node. Thus attributes of two different elements may compare as equal using =, even though they are not the same node.

A defaulted attribute is treated the same as a specified attribute. If an attribute was declared for the element type in the DTD, but the default was declared as `#IMPLIED`, and the attribute was not specified on the element, then the element's attribute set does not contain a node for the attribute.

Some attributes, such as `xml:lang` and `xml:space`, have the semantics that they apply to all elements that are descendants of the element bearing the attribute, unless overridden with an instance of the same

attribute on another descendant element. However, this does not affect where attribute nodes appear in the tree: an element has attribute nodes only for attributes that were explicitly specified in the start-tag or empty-element tag of that element or that were explicitly declared in the DTD with a default value.

An attribute node has an expanded-name and a string-value. The expanded-name is computed by expanding the QName specified in the tag in the XML document in accordance with the XML Namespaces Recommendation [XML Names]. The namespace URI of the attribute's name will be null if the QName of the attribute does not have a prefix.

> **NOTE:** In the notation of Appendix A.3 of [XML Names], the local part of the expanded-name corresponds to the `name` attribute of the `ExpAName` element; the namespace URI of the expanded-name corresponds to the `ns` attribute of the `ExpAName` element, and is null if the `ns` attribute of the `ExpAName` element is omitted.

An attribute node has a string-value. The string-value is the normalized value as specified by the XML Recommendation [XML]. An attribute whose normalized value is a zero-length string is not treated specially: it results in an attribute node whose string-value is a zero-length string.

> **NOTE:** It is possible for default attributes to be declared in an external DTD or an external parameter entity. The XML Recommendation does not require an XML processor to read an external DTD or an external parameter unless it is validating. A stylesheet or other facility that assumes that the XPath tree contains default attribute values declared in an external DTD or parameter entity may not work with some non-validating XML processors.

There are no attribute nodes corresponding to attributes that declare namespaces (see [XML Names]).

5.4 Namespace Nodes

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 the XML Namespaces Recommendation [XML Names]) 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. Elements never share namespace nodes: if one element node is not the same node as another element node, then none of the namespace nodes of the one element node will be the same node as the namespace nodes of another element node. This means that an element will have a namespace node:

- for every attribute on the element whose name starts with `xmlns:`;
- for every attribute on an ancestor element whose name starts `xmlns:` unless the element itself or a nearer ancestor redeclares the prefix;
- for an `xmlns` attribute, if the element or some ancestor has an `xmlns` attribute, and the value of the `xmlns` attribute for the nearest such element is non-empty

  > **NOTE:** An attribute `xmlns=""` "undeclares" the default namespace (see [XML Names]).

A namespace node has an expanded-name: the local part is the namespace prefix (this is empty if the namespace node is for the default namespace); the namespace URI is always null.

The string-value of a namespace node is the namespace URI that is being bound to the namespace prefix; if it

is relative, it must be resolved just like a namespace URI in an expanded-name.

5.5 Processing Instruction Nodes

There is a processing instruction node for every processing instruction, except for any processing instruction that occurs within the document type declaration.

A processing instruction has an expanded-name: the local part is the processing instruction's target; the namespace URI is null. The string-value of a processing instruction node is the part of the processing instruction following the target and any whitespace. It does not include the terminating `?>`.

> **NOTE:** The XML declaration is not a processing instruction. Therefore, there is no processing instruction node corresponding to the XML declaration.

5.6 Comment Nodes

There is a comment node for every comment, except for any comment that occurs within the document type declaration.

The string-value of comment is the content of the comment not including the opening `<!--` or the closing `-->`.

A comment node does not have an expanded-name.

5.7 Text Nodes

Character data is grouped into text nodes. As much character data as possible is grouped into each text node: a text node never has an immediately following or preceding sibling that is a text node. The string-value of a text node is the character data. A text node always has at least one character of data.

Each character within a CDATA section is treated as character data. Thus, `<![CDATA[<]]>` in the source document will treated the same as `&lt;`. Both will result in a single < character in a text node in the tree. Thus, a CDATA section is treated as if the `<![CDATA[` and `]]>` were removed and every occurrence of < and & were replaced by `&lt;` and `&amp;` respectively.

> **NOTE:** When a text node that contains a < character is written out as XML, the < character must be escaped by, for example, using `&lt;`, or including it in a CDATA section.

Characters inside comments, processing instructions and attribute values do not produce text nodes. Line-endings in external entities are normalized to #xA as specified in the XML Recommendation [XML].

A text node does not have an expanded-name.

6 Conformance

XPath is intended primarily as a component that can be used by other specifications. Therefore, XPath relies on specifications that use XPath (such as [XPointer] and [XSLT]) to specify criteria for conformance of implementations of XPath and does not define any conformance criteria for independent implementations of XPath.

A References

A.1 Normative References

IEEE 754

Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754-1985.

RFC2396

T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. IETF RFC 2396. See http://www.ietf.org/rfc/rfc2396.txt.

XML

World Wide Web Consortium. *Extensible Markup Language (XML) 1.0*. W3C Recommendation. See http://www.w3.org/TR/1998/REC-xml-19980210

XML Names

World Wide Web Consortium. *Namespaces in XML*. W3C Recommendation. See http://www.w3.org/TR/REC-xml-names

A.2 Other References

Character Model

World Wide Web Consortium. *Character Model for the World Wide Web*. W3C Working Draft. See http://www.w3.org/TR/WD-charmod

DOM

World Wide Web Consortium. *Document Object Model (DOM) Level 1 Specification*. W3C Recommendation. See http://www.w3.org/TR/REC-DOM-Level-1

JLS

J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. See http://java.sun.com/docs/books/jls/index.html.

ISO/IEC 10646

ISO (International Organization for Standardization). *ISO/IEC 10646-1:1993, Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane*. International Standard. See http://www.iso.ch/cate/d18741.html.

TEI

C.M. Sperberg-McQueen, L. Burnard *Guidelines for Electronic Text Encoding and Interchange*. See http://etext.virginia.edu/TEI.html.

Unicode

Unicode Consortium. *The Unicode Standard*. See http://www.unicode.org/unicode/standard/standard.html.

XML Infoset

World Wide Web Consortium. *XML Information Set*. W3C Working Draft. See

http://www.w3.org/TR/xml-infoset

XPointer

World Wide Web Consortium. *XML Pointer Language (XPointer)*. W3C Working Draft. See http://www.w3.org/TR/WD-xptr

XQL

J. Robie, J. Lapp, D. Schach. *XML Query Language (XQL)*. See http://www.w3.org/TandS/QL/QL98/pp/xql.html

XSLT

World Wide Web Consortium. *XSL Transformations (XSLT)*. W3C Recommendation. See http://www.w3.org/TR/xslt

B XML Information Set Mapping (Non-Normative)

The nodes in the XPath data model can be derived from the information items provided by the XML Information Set [XML Infoset] as follows:

NOTE: A new version of the XML Information Set Working Draft, which will replace the May 17 version, was close to completion at the time when the preparation of this version of XPath was completed and was expected to be released at the same time or shortly after the release of this version of XPath. The mapping is given for this new version of the XML Information Set Working Draft. If the new version of the XML Information Set Working has not yet been released, W3C members may consult the internal Working Group version http://www.w3.org/XML/Group/1999/09/WD-xml-infoset-19990915.html (members only).

- The root node comes from the document information item. The children of the root node come from the *children* and *children - comments* properties.
- An element node comes from an element information item. The children of an element node come from the *children* and *children - comments* properties. The attributes of an element node come from the *attributes* property. The namespaces of an element node come from the *in-scope namespaces* property. The local part of the expanded-name of the element node comes from the *local name* property. The namespace URI of the expanded-name of the element node comes from the *namespace URI* property. The unique ID of the element node comes from the *children* property of the attribute information item in the *attributes* property that has an *attribute type* property equal to `ID`.
- An attribute node comes from an attribute information item. The local part of the expanded-name of the attribute node comes from the *local name* property. The namespace URI of the expanded-name of the attribute node comes from the *namespace URI* property. The string-value of the node comes from concatenating the *character code* property of each member of the *children* property.
- A text node comes from a sequence of one or more consecutive character information items. The string-value of the node comes from concatenating the *character code* property of each of the character information items.
- A processing instruction node comes from a processing instruction information item. The local part of the expanded-name of the node comes from the *target* property. (The namespace URI part of the

expanded-name of the node is null.) The string-value of the node comes from the *content* property. There are no processing instruction nodes for processing instruction items that are children of document type declaration information item.

- A comment node comes from a comment information item. The string-value of the node comes from the *content* property. There are no comment nodes for comment information items that are children of document type declaration information item.
- A namespace node comes from a namespace declaration information item. The local part of the expanded-name of the node comes from the *prefix* property. (The namespace URI part of the expanded-name of the node is null.) The string-value of the node comes from the *namespace URI* property.