XML Path Language (XPath) 2.0 (Second Edition)

W3C Recommendation 14 December 2010 *(Link errors corrected 3 January 2011)*

This version:
http://www.w3.org/TR/2010/REC-xpath20-20101214/
Latest version:
http://www.w3.org/TR/xpath20/
Previous versions:
http://www.w3.org/TR/2009/PER-xpath20-20090421/, http://www.w3.org/TR/2007/REC-xpath20-20070123/
Editors:
Anders Berglund (XSL WG), BC&TF <http://www.albconsults.com>
Scott Boag (XSL WG), IBM Research <scott_boag@us.ibm.com>
Don Chamberlin (XML Query WG) <dchamber@us.ibm.com>
Mary F. Fernández (XML Query WG), AT&T Labs <mff@research.att.com>
Michael Kay (XSL WG), Saxonica, via http://www.saxonica.com/
Jonathan Robie (XML Query WG), Red Hat, via http://www.ibiblio.org/jwrobie/
Jérôme Siméon (XML Query WG), IBM T.J. Watson Research Center <simeon@us.ibm.com>

Please refer to the **errata** for this document, which may include some normative corrections.

See also **translations**.

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

Abstract

XPath 2.0 is an expression language that allows the processing of values conforming to the data model defined in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)]. The data model provides a tree representation of XML documents as well as atomic values such as integers, strings, and booleans, and sequences that may contain both references to nodes in an XML document and atomic values. The result of an XPath expression may be a selection of nodes from the input documents, or an atomic value, or more generally, any sequence allowed by the data model. The name of the language derives from its most distinctive feature, the path expression, which provides a means of hierarchic addressing of the nodes in an XML tree. XPath 2.0 is a superset of [XPath 1.0], with the added capability to support a richer set of data types, and to take advantage of the type information that becomes available when documents are validated using XML Schema. A backwards compatibility mode is provided to ensure that nearly all XPath 1.0 expressions continue to deliver the same result with XPath 2.0; exceptions to this policy are noted in [**I Backwards Compatibility with XPath 1.0**].

Status of this Document

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

This is one document in a set of eight documents that are being progressed to Edited Recommendation together (XPath 2.0, XQuery 1.0, XQueryX 1.0, XSLT 2.0, Data Model (XDM), Functions and Operators, Formal Semantics, Serialization).

This document, published on 14 December 2010, is an Edited Recommendation of the W3C. It supersedes the previous W3C Recommendation of 23 January 2007. This second edition is not a new version of this specification; its purpose is to clarify a number of issues that have become apparent since the first edition was published. All of these clarifications (excepting trivial editorial fixes) have been published in a separate errata document, and published in a Proposed Edited Recommendation in April 2009. The changes are summarized in an appendix. On 3 January 2011, the original publication of this Recommendation was replaced by this version in which two HTML anchors that were omitted by the original publication have been restored; the W3C Team has retained a copy of the original publication. This document has been jointly developed by the W3C XSL Working Group and the W3C XML Query Working Group, each of which is part of the XML Activity.

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.

This document incorporates changes made against the Recommendation of 23 January 2007 that resolve all errata known at the date of publication. Changes to this document since the first edition are detailed in the **J Changes since the First Edition**. This document supersedes the first edition.

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 1.0 and XSLT 2.0 specifications; see the XQuery 1.0 implementation report and the XSLT 2.0 implementation report (member-only) for details.

Please report errors in and submit comments on this document using W3C's public Bugzilla system (instructions can be found at http://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 “[XPath]” in the subject line of your report, whether made in Bugzilla or in email. Each Bugzilla entry and email message should contain only one error report. Archives of the comments and responses are available at http://lists.w3.org/Archives/Public/public-qt-comments/.

This document was produced by groups operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the XML Query Working Group and also maintains a public list of any patent disclosures made in connection with the deliverables of the XSL Working Group; those pages also include instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

Appendices

---

1 Introduction

The primary purpose of XPath is to address the nodes of [XML 1.0] or [XML 1.1] trees. XPath gets its name from its use of a path notation for navigating through the hierarchical structure of an XML document. XPath uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values.

[Definition: XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. This logical structure, known as the **data model**, is defined in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)].]

XPath is designed to be embedded in a **host language** such as [XSL Transformations (XSLT) Version 2.0 (Second Edition)] or [XQuery 1.0: An XML Query Language (Second Edition)]. XPath 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 [XSL Transformations (XSLT) Version 2.0 (Second Edition)].

XQuery Version 1.0 is an extension of XPath Version 2.0. Any expression that is syntactically valid and executes successfully in both XPath 2.0 and XQuery 1.0 will return the same result in both languages. Since these languages are so closely related, their grammars and language descriptions are generated from a common source to ensure consistency, and the editors of these specifications work together closely.

XPath also depends on and is closely related to the following specifications:

- [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)] defines the data model that underlies all XPath expressions.
- [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)] defines the static semantics of XPath and also contains a formal but non-normative description of the dynamic semantics that may be useful for implementors and others who require a formal definition.
- The type system of XPath is based on [XML Schema].
- The built-in function library and the operators supported by XPath are defined in [XQuery 1.0 and

[XPath 2.0 Functions and Operators (Second Edition)].

This document specifies a grammar for XPath, using the same basic EBNF notation used in [XML 1.0]. Unless otherwise noted (see **A.2 Lexical structure**), whitespace is not significant in expressions. Grammar productions are introduced together with the features that they describe, and a complete grammar is also presented in the appendix [**A XPath Grammar**]. The appendix is the normative version.

In the grammar productions in this document, named symbols are underlined and literal text is enclosed in double quotes. For example, the following production describes the syntax of a function call:

```
[48]  FunctionCall  ::=  QName "(" (ExprSingle ("," ExprSingle)*)? ")"
```

The production should be read as follows: A function call consists of a QName followed by an open-parenthesis. The open-parenthesis is followed by an optional argument list. The argument list (if present) consists of one or more expressions, separated by commas. The optional argument list is followed by a close-parenthesis.

Certain aspects of language processing are described in this specification as **implementation-defined** or **implementation-dependent**.

- [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.]

A language aspect described in this specification as **implementation-defined** or **implementation dependent** may be further constrained by the specifications of a host language in which XPath is embedded.

This document normatively defines the dynamic semantics of XPath. The static semantics of XPath are normatively defined in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)]. In this document, examples and material labeled as "Note" are provided for explanatory purposes and are not normative.

2 Basics

The basic building block of XPath is the **expression**, which is a string of [Unicode] characters (the version of Unicode to be used is implementation-defined.) The language provides several kinds of expressions which may be constructed from keywords, symbols, and operands. In general, the operands of an expression are other expressions. XPath allows expressions to be nested with full generality.

> **Note:**
>
> This specification contains no assumptions or requirements regarding the character set encoding of strings of [Unicode] characters.

Like XML, XPath is a case-sensitive language. Keywords in XPath use lower-case characters and are not reserved—that is, names in XPath expressions are allowed to be the same as language keywords, except for certain unprefixed function-names listed in **A.3 Reserved Function Names**.

[Definition: In the data model, a **value** is always a sequence.] [Definition: A **sequence** is an ordered collection of zero or more items.] [Definition: An **item** is either an atomic value or a node.] [Definition: An **atomic value** is a value in the value space of an **atomic type**, as defined in [XML Schema].] [Definition: A **node** is an instance of one of the **node kinds** defined in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)].] Each node has a unique **node identity**, a **typed value**, and a **string value**. In addition, some nodes have a **name**. The **typed value** of a node is a sequence of zero or more atomic values. The **string value** of a node is a value of type `xs:string`. The **name** of a node is a value of type `xs:QName`. [Definition: In certain situations a value is said to be **undefined** (for example, the value of the context item, or the typed value of an element node). This term indicates that the property in question has no value and that any attempt to use its value results in an error.]

[Definition: A sequence containing exactly one item is called a **singleton**.] An item is identical to a singleton sequence containing that item. Sequences are never nested—for example, combining the values

1, (2, 3), and ( ) into a single sequence results in the sequence (1, 2, 3). [Definition: A sequence containing zero items is called an **empty sequence**.]

[Definition: The term **XDM instance** is used, synonymously with the term **value**, to denote an unconstrained sequence of nodes and/or atomic values in the data model.]

Names in XPath are called **QNames**, and conform to the syntax in [XML Names]. [Definition: Lexically, a **QName** consists of an optional namespace prefix and a local name. If the namespace prefix is present, it is separated from the local name by a colon.] A lexical QName can be converted into an **expanded QName** by resolving its namespace prefix to a namespace URI, using the statically known namespaces [err:XPST0081]. [Definition: An **expanded QName** consists of an optional namespace URI and a local name. An expanded QName also retains its original namespace prefix (if any), to facilitate casting the expanded QName into a string.] The namespace URI value is whitespace normalized according to the rules for the `xs:anyURI` type in [XML Schema]. Two expanded QNames are equal if their namespace URIs are equal and their local names are equal (even if their namespace prefixes are not equal). Namespace URIs and local names are compared on a codepoint basis, without further normalization.

This document uses the following namespace prefixes to represent the namespace URIs with which they are listed. Use of these namespace prefix bindings in this document is not normative.

- `xs = http://www.w3.org/2001/XMLSchema`
- `fn = http://www.w3.org/2005/xpath-functions`
- `err = http://www.w3.org/2005/xqt-errors` (see **2.3.2 Identifying and Reporting Errors**).

Element nodes have a property called **in-scope namespaces**. [Definition: The **in-scope namespaces** property of an element node is a set of **namespace bindings**, each of which associates a namespace prefix with a URI, thus defining the set of namespace prefixes that are available for interpreting QNames within the scope of the element. For a given element, one namespace binding may have an empty prefix; the URI of this namespace binding is the default namespace within the scope of the element.]

In [XPath 1.0], the in-scope namespaces of an element node are represented by a collection of **namespace nodes** arranged on a **namespace axis**. In XPath Version 2.0, the namespace axis is deprecated and need not be supported by a host language. A host language that does not support the namespace axis need not represent namespace bindings in the form of nodes.

[Definition: Within this specification, the term **URI** refers to a Universal Resource Identifier as defined in [RFC3986] and extended in [RFC3987] 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.

2.1 Expression Context

[Definition: The **expression context** for a given expression consists of all the information that can affect the result of the expression.] This information is organized into two categories called the static context and the dynamic context.

2.1.1 Static Context

[Definition: The **static context** of an expression is the information that is available during static analysis of the expression, prior to its evaluation.] This information can be used to decide whether the expression contains a static error. If analysis of an expression relies on some component of the static context that has not been assigned a value, a static error is raised [err:XPST0001].

The individual components of the static context are summarized below. A default initial value for each component may be specified by the host language. The scope of each component is specified in **C.1 Static Context Components**.

- [Definition: **XPath 1.0 compatibility mode.** This value is `true` if rules for backward compatibility with XPath Version 1.0 are in effect; otherwise it is `false`.]
- [Definition: **Statically known namespaces.** This is a set of (prefix, URI) pairs that define all the namespaces that are known during static processing of a given expression.] The URI value is whitespace normalized according to the rules for the `xs:anyURI` type in [XML Schema]. Note the

difference between in-scope namespaces, which is a dynamic property of an element node, and statically known namespaces, which is a static property of an expression.

- [Definition: **Default element/type namespace.** This is a namespace URI or "none". The namespace URI, if present, is used for any unprefixed QName appearing in a position where an element or type name is expected.] The URI value is whitespace normalized according to the rules for the `xs:anyURI` type in [XML Schema].
- [Definition: **Default function namespace.** This is a namespace URI or "none". The namespace URI, if present, is used for any unprefixed QName appearing in a position where a function name is expected.] The URI value is whitespace normalized according to the rules for the `xs:anyURI` type in [XML Schema].
- [Definition: **In-scope schema definitions.** This is a generic term for all the element declarations, attribute declarations, and schema type definitions that are in scope during processing of an expression.] It includes the following three parts:
    - [Definition: **In-scope schema types.** Each schema type definition is identified either by an expanded QName (for a **named type**) or by an implementation-dependent type identifier (for an **anonymous type**). The in-scope schema types include the predefined schema types described in **2.5.1 Predefined Schema Types**.]
    - [Definition: **In-scope element declarations.** Each element declaration is identified either by an expanded QName (for a top-level element declaration) or by an implementation-dependent element identifier (for a local element declaration).] An element declaration includes information about the element's substitution group affiliation.

      [Definition: **Substitution groups** are defined in [XML Schema] Part 1, Section 2.2.2.2. Informally, the substitution group headed by a given element (called the **head element**) consists of the set of elements that can be substituted for the head element without affecting the outcome of schema validation.]
    - [Definition: **In-scope attribute declarations.** Each attribute declaration is identified either by an expanded QName (for a top-level attribute declaration) or by an implementation-dependent attribute identifier (for a local attribute declaration).]
- [Definition: **In-scope variables.** This is a set of (expanded QName, type) pairs. It defines the set of variables that are available for reference within an expression. The expanded QName is the name of the variable, and the type is the static type of the variable.]

  An expression that binds a variable (such as a `for`, `some`, or `every` expression) extends the in-scope variables of its subexpressions with the new bound variable and its type.
- [Definition: **Context item static type.** This component defines the static type of the context item within the scope of a given expression.]
- [Definition: **Function signatures.** This component defines the set of functions that are available to be called from within an expression. Each function is uniquely identified by its expanded QName and its arity (number of parameters).] In addition to the name and arity, each function signature specifies the static types of the function parameters and result.

  The function signatures include the signatures of constructor functions, which are discussed in **3.10.4 Constructor Functions**.
- [Definition: **Statically known collations.** This is an implementation-defined set of (URI, collation) pairs. It defines the names of the collations that are available for use in processing expressions.] [Definition: A **collation** is a specification of the manner in which strings and URIs are compared and, by extension, ordered. For a more complete definition of collation, see [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].]
- [Definition: **Default collation.** This identifies one of the collations in statically known collations as the collation to be used by functions and operators for comparing and ordering values of type `xs:string` and `xs:anyURI` (and types derived from them) when no explicit collation is specified.]
- [Definition: **Base URI.** This is an absolute URI, used when necessary in the resolution of relative URIs (for example, by the `fn:resolve-uri` function.)] The URI value is whitespace normalized according to the rules for the `xs:anyURI` type in [XML Schema].
- [Definition: **Statically known documents.** This is a mapping from strings onto types. The string represents the absolute URI of a resource that is potentially available using the `fn:doc` function. The type is the static type of a call to `fn:doc` with the given URI as its literal argument.] If the argument to `fn:doc` is a string literal that is not present in **statically known documents**, then the static type of `fn:doc` is `document-node()?`.

Note:

The purpose of the **statically known documents** is to provide static type information, not to determine which documents are available. A URI need not be found in the **statically known documents** to be accessed using `fn:doc`.

- [Definition: **Statically known collections.** This is a mapping from strings onto types. The string represents the absolute URI of a resource that is potentially available using the `fn:collection` function. The type is the type of the sequence of nodes that would result from calling the `fn:collection` function with this URI as its argument.] If the argument to `fn:collection` is a string literal that is not present in **statically known collections**, then the static type of `fn:collection` is `node()*`.

  Note:

  The purpose of the **statically known collections** is to provide static type information, not to determine which collections are available. A URI need not be found in the **statically known collections** to be accessed using `fn:collection`.

- [Definition: **Statically known default collection type.** This is the type of the sequence of nodes that would result from calling the `fn:collection` function with no arguments.] Unless initialized to some other value by an implementation, the value of **statically known default collection type** is `node()*`.

2.1.2 Dynamic Context

[Definition: The **dynamic context** of an expression is defined as information that is available at the time the expression is evaluated.] If evaluation of an expression relies on some part of the dynamic context that has not been assigned a value, a dynamic error is raised [err:XPDY0002].

The individual components of the dynamic context are summarized below. Further rules governing the semantics of these components can be found in **C.2 Dynamic Context Components**.

The dynamic context consists of all the components of the static context, and the additional components listed below.

[Definition: The first three components of the dynamic context (context item, context position, and context size) are called the **focus** of the expression.] The focus enables the processor to keep track of which items are being processed by the expression.

Certain language constructs, notably the path expression `E1/E2` and the predicate `E1[E2]`, create a new focus for the evaluation of a sub-expression. In these constructs, `E2` is evaluated once for each item in the sequence that results from evaluating `E1`. Each time `E2` is evaluated, it is evaluated with a different focus. The focus for evaluating `E2` is referred to below as the **inner focus**, while the focus for evaluating `E1` is referred to as the **outer focus**. The inner focus exists only while `E2` is being evaluated. When this evaluation is complete, evaluation of the containing expression continues with its original focus unchanged.

- [Definition: The **context item** is the item currently being processed. An item is either an atomic value or a node.][Definition: When the context item is a node, it can also be referred to as the **context node**.] The context item is returned by an expression consisting of a single dot (`.`). When an expression `E1/E2` or `E1[E2]` is evaluated, each item in the sequence obtained by evaluating `E1` becomes the context item in the inner focus for an evaluation of `E2`.
- [Definition: The **context position** is the position of the context item within the sequence of items currently being processed.] It changes whenever the context item changes. When the focus is defined, the value of the context position is an integer greater than zero. The context position is returned by the expression `fn:position()`. When an expression `E1/E2` or `E1[E2]` is evaluated, the context position in the inner focus for an evaluation of `E2` is the position of the context item in the sequence obtained by evaluating `E1`. The position of the first item in a sequence is always 1 (one). The context position is always less than or equal to the context size.
- [Definition: The **context size** is the number of items in the sequence of items currently being processed.] Its value is always an integer greater than zero. The context size is returned by the expression `fn:last()`. When an expression `E1/E2` or `E1[E2]` is evaluated, the context size in the inner focus for an evaluation of `E2` is the number of items in the sequence obtained by evaluating `E1`.
- [Definition: **Variable values**. This is a set of (expanded QName, value) pairs. It contains the same expanded QNames as the in-scope variables in the static context for the expression. The expanded QName is the name of the variable and the value is the dynamic value of the variable, which includes

its dynamic type.]

- [Definition: **Function implementations**. Each function in function signatures has a function implementation that enables the function to map instances of its parameter types into an instance of its result type.]
- [Definition: **Current dateTime.** This information represents an implementation-dependent point in time during the processing of an expression, and includes an explicit timezone. It can be retrieved by the `fn:current-dateTime` function. If invoked multiple times during the execution of an expression, this function always returns the same result.]
- [Definition: **Implicit timezone.** This is the timezone to be used when a date, time, or dateTime value that does not have a timezone is used in a comparison or arithmetic operation. The implicit timezone is an implementation-defined value of type `xs:dayTimeDuration`. See [XML Schema] for the range of legal values of a timezone.]
- [Definition: **Available documents.** This is a mapping of strings onto document nodes. The string represents the absolute URI of a resource. The document node is the root of a tree that represents that resource using the data model. The document node is returned by the `fn:doc` function when applied to that URI.] The set of available documents is not limited to the set of statically known documents, and it may be empty.

  If there are one or more URIs in available documents that map to a document node `D`, then the document-uri property of `D` must either be absent, or must be one of these URIs.

  Note:

  This means that given a document node `$N`, the result of `fn:doc(fn:document-uri($N)) is $N` will always be True, unless `fn:document-uri($N)` is an empty sequence.
- [Definition: **Available collections.** This is a mapping of strings onto sequences of nodes. The string represents the absolute URI of a resource. The sequence of nodes represents the result of the `fn:collection` function when that URI is supplied as the argument.] The set of available collections is not limited to the set of statically known collections, and it may be empty.

  For every document node `D` that is in the target of a mapping in available collections, or that is the root of a tree containing such a node, the document-uri property of `D` must either be absent, or must be a URI `U` such that available documents contains a mapping from `U` to `D`."

  Note:

  This means that for any document node `$N` retrieved using the `fn:collection` function, either directly or by navigating to the root of a node that was returned, the result of `fn:doc(fn:document-uri($N)) is $N` will always be True, unless `fn:document-uri($N)` is an empty sequence. This implies a requirement for the `fn:doc` and `fn:collection` functions to be consistent in their effect. If the implementation uses catalogs or user-supplied URI resolvers to dereference URIs supplied to the `fn:doc` function, the implementation of the `fn:collection` function must take these mechanisms into account. For example, an implementation might achieve this by mapping the collection URI to a set of document URIs, which are then resolved using the same catalog or URI resolver that is used by the `fn:doc` function.
- [Definition: **Default collection.** This is the sequence of nodes that would result from calling the `fn:collection` function with no arguments.] The value of **default collection** may be initialized by the implementation.

2.2 Processing Model

XPath is defined in terms of the data model and the expression context.

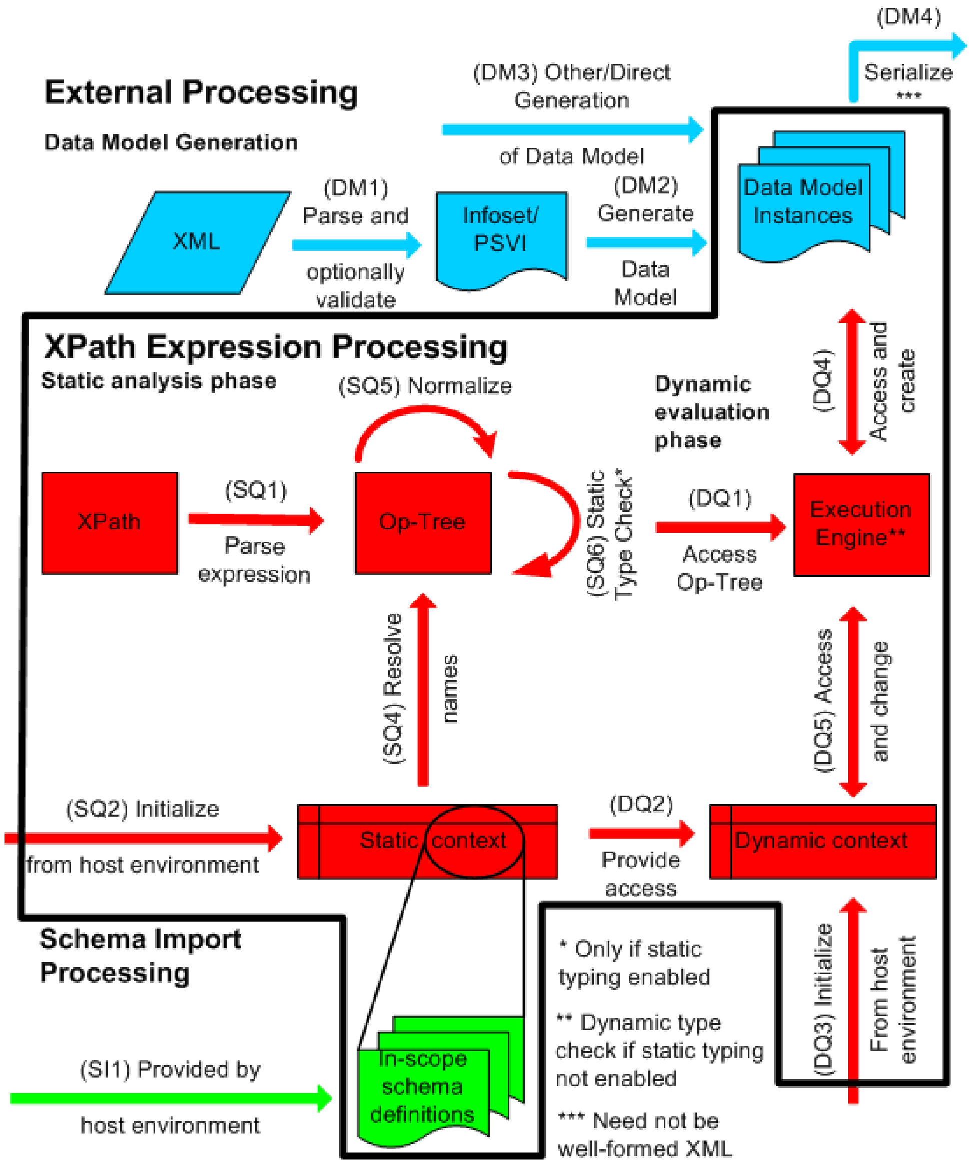

Figure 1: Processing Model Overview

Figure 1 provides a schematic overview of the processing steps that are discussed in detail below. Some of these steps are completely outside the domain of XPath; in Figure 1, these are depicted outside the line that represents the boundaries of the language, an area labeled **external processing**. The external processing domain includes generation of an XDM instance that represents the data to be queried (see **2.2.1 Data Model Generation**), schema import processing (see **2.2.2 Schema Import Processing**) and serialization (see **2.2.4 Serialization**). The area inside the boundaries of the language is known as the **XPath processing domain**, which includes the static analysis and dynamic evaluation phases (see **2.2.3 Expression Processing**). Consistency constraints on the XPath processing domain are defined in **2.2.5 Consistency Constraints**.

2.2.1 Data Model Generation

Before an expression can be processed, its input data must be represented as an XDM instance. This process occurs outside the domain of XPath, which is why Figure 1 represents it in the external processing domain. Here are some steps by which an XML document might be converted to an XDM instance:

1. A document may be parsed using an XML parser that generates an **XML Information Set** (see [XML Infoset]). The parsed document may then be validated against one or more schemas. This process, which is described in [XML Schema], results in an abstract information structure called the **Post-Schema Validation Infoset** (PSVI). If a document has no associated schema, its Information Set is preserved. (See DM1 in Fig. 1.)
2. The Information Set or PSVI may be transformed into an XDM instance by a process described in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)]. (See DM2 in Fig. 1.)

The above steps provide an example of how an XDM instance might be constructed. An XDM instance might also be synthesized directly from a relational database, or constructed in some other way (see DM3 in Fig. 1.) XPath is defined in terms of the data model, but it does not place any constraints on how XDM instances are constructed.

[Definition: Each element node and attribute node in an XDM instance has a **type annotation** (referred to in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)] as its `type-name` property.) The type annotation of a node is a schema type that describes the relationship between the string value of the node and its typed value.] If the XDM instance was derived from a validated XML document as described in Section 3.3 Construction from a PSVI *DM*, the type annotations of the element and attribute nodes are derived from schema validation. XPath does not provide a way to directly access the type annotation of an element or attribute node.

The value of an attribute is represented directly within the attribute node. An attribute node whose type is unknown (such as might occur in a schemaless document) is given the type annotation `xs:untypedAtomic`.

The value of an element is represented by the children of the element node, which may include text nodes and other element nodes. The type annotation of an element node indicates how the values in its child text nodes are to be interpreted. An element that has not been validated (such as might occur in a schemaless document) is annotated with the schema type `xs:untyped`. An element that has been validated and found to be partially valid is annotated with the schema type `xs:anyType`. If an element node is annotated as `xs:untyped`, all its descendant element nodes are also annotated as `xs:untyped`. However, if an element node is annotated as `xs:anyType`, some of its descendant element nodes may have a more specific type annotation.

2.2.2 Schema Import Processing

The in-scope schema definitions in the static context are provided by the host language (see step SI1 in Figure 1) and must satisfy the consistency constraints defined in **2.2.5 Consistency Constraints**.

2.2.3 Expression Processing

XPath defines two phases of processing called the static analysis phase and the dynamic evaluation phase (see Fig. 1). During the static analysis phase, static errors, dynamic errors, or type errors may be raised. During the dynamic evaluation phase, only dynamic errors or type errors may be raised. These kinds of errors are defined in **2.3.1 Kinds of Errors**.

Within each phase, an implementation is free to use any strategy or algorithm whose result conforms to the specifications in this document.

2.2.3.1 Static Analysis Phase

[Definition: The **static analysis phase** depends on the expression itself and on the static context. The **static analysis phase** does not depend on input data (other than schemas).]

During the static analysis phase, the XPath expression is parsed into an internal representation called the **operation tree** (step SQ1 in Figure 1). A parse error is raised as a static error [err:XPST0003]. The static context is initialized by the implementation (step SQ2). The static context is used to resolve schema type names, function names, namespace prefixes, and variable names (step SQ4). If a name of one of these kinds in the **operation tree** is not found in the static context, a static error ([err:XPST0008] or [err:XPST0017]) is raised (however, see exceptions to this rule in **2.5.4.3 Element Test** and **2.5.4.5 Attribute Test**.)

The **operation tree** is then **normalized** by making explicit the implicit operations such as atomization and extraction of Effective Boolean Values (step SQ5). The normalization process is described in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)].

Each expression is then assigned a static type (step SQ6). [Definition: The **static type** of an expression is a type such that, when the expression is evaluated, the resulting value will always conform to the static type.] If the Static Typing Feature is supported, the static types of various expressions are inferred according to the rules described in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)]. If the Static Typing

Feature is not supported, the static types that are assigned are implementation-dependent.

During the static analysis phase, if the Static Typing Feature is in effect and an operand of an expression is found to have a static type that is not appropriate for that operand, a type error is raised [err:XPTY0004]. If static type checking raises no errors and assigns a static type T to an expression, then execution of the expression on valid input data is guaranteed either to produce a value of type T or to raise a dynamic error.

The purpose of the Static Typing Feature is to provide early detection of type errors and to infer type information that may be useful in optimizing the evaluation of an expression.

2.2.3.2 Dynamic Evaluation Phase

[Definition: The **dynamic evaluation phase** is the phase during which the value of an expression is computed.] It occurs after completion of the static analysis phase.

The dynamic evaluation phase can occur only if no errors were detected during the static analysis phase. If the Static Typing Feature is in effect, all type errors are detected during static analysis and serve to inhibit the dynamic evaluation phase.

The dynamic evaluation phase depends on the **operation tree** of the expression being evaluated (step DQ1), on the input data (step DQ4), and on the dynamic context (step DQ5), which in turn draws information from the external environment (step DQ3) and the static context (step DQ2). The dynamic evaluation phase may create new data-model values (step DQ4) and it may extend the dynamic context (step DQ5)—for example, by binding values to variables.

[Definition: A **dynamic type** is associated with each value as it is computed. The dynamic type of a value may be more specific than the static type of the expression that computed it (for example, the static type of an expression might be `xs:integer*`, denoting a sequence of zero or more integers, but at evaluation time its value may have the dynamic type `xs:integer`, denoting exactly one integer.)]

If an operand of an expression is found to have a dynamic type that is not appropriate for that operand, a type error is raised [err:XPTY0004].

Even though static typing can catch many type errors before an expression is executed, it is possible for an expression to raise an error during evaluation that was not detected by static analysis. For example, an expression may contain a cast of a string into an integer, which is statically valid. However, if the actual value of the string at run time cannot be cast into an integer, a dynamic error will result. Similarly, an expression may apply an arithmetic operator to a value whose static type is `xs:untypedAtomic`. This is not a static error, but at run time, if the value cannot be successfully cast to a numeric type, a dynamic error will be raised.

When the Static Typing Feature is in effect, it is also possible for static analysis of an expression to raise a type error, even though execution of the expression on certain inputs would be successful. For example, an expression might contain a function that requires an element as its parameter, and the static analysis phase might infer the static type of the function parameter to be an optional element. This case is treated as a type error and inhibits evaluation, even though the function call would have been successful for input data in which the optional element is present.

2.2.4 Serialization

[Definition: **Serialization** is the process of converting an XDM instance into a sequence of octets (step DM4 in Figure 1.)] The general framework for serialization is described in [XSLT 2.0 and XQuery 1.0 Serialization (Second Edition)].

The host language may provide a serialization option.

2.2.5 Consistency Constraints

In order for XPath to be well defined, the input XDM instance, the static context, and the dynamic context must be mutually consistent. The consistency constraints listed below are prerequisites for correct functioning of an XPath implementation. Enforcement of these consistency constraints is beyond the scope of this specification. This specification does not define the result of an expression under any condition in which one or more of these constraints is not satisfied.

Some of the consistency constraints use the term **data model schema**. [Definition: For a given node in an XDM instance, the **data model schema** is defined as the schema from which the type annotation of that node was derived.] For a node that was constructed by some process other than schema validation, the **data model schema** consists simply of the schema type definition that is represented by the type annotation of the node.

- For every node that has a type annotation, if that type annotation is found in the in-scope schema definitions (ISSD), then its definition in the ISSD must be equivalent to its definition in the data model schema. Furthermore, all types that are derived by extension from the given type in the data model schema must also be known by equivalent definitions in the ISSD.
- For every element name *EN* that is found both in an XDM instance and in the in-scope schema definitions (ISSD), all elements that are known in the data model schema to be in the substitution group headed by *EN* must also be known in the ISSD to be in the substitution group headed by *EN*.
- Every element name, attribute name, or schema type name referenced in in-scope variables or function signatures must be in the in-scope schema definitions, unless it is an element name referenced as part of an ElementTest or an attribute name referenced as part of an AttributeTest.
- Any reference to a global element, attribute, or type name in the in-scope schema definitions must have a corresponding element, attribute or type definition in the in-scope schema definitions.
- For each mapping of a string to a document node in available documents, if there exists a mapping of the same string to a document type in statically known documents, the document node must match the document type, using the matching rules in **2.5.4 SequenceType Matching**.
- For each mapping of a string to a sequence of nodes in available collections, if there exists a mapping of the same string to a type in statically known collections, the sequence of nodes must match the type, using the matching rules in **2.5.4 SequenceType Matching**.
- The sequence of nodes in the default collection must match the statically known default collection type, using the matching rules in **2.5.4 SequenceType Matching**.
- The value of the context item must match the context item static type, using the matching rules in **2.5.4 SequenceType Matching**.
- For each (variable, type) pair in in-scope variables and the corresponding (variable, value) pair in variable values such that the variable names are equal, the value must match the type, using the matching rules in **2.5.4 SequenceType Matching**.
- In the statically known namespaces, the prefix `xml` must not be bound to any namespace URI other than `http://www.w3.org/XML/1998/namespace`, and no prefix other than `xml` may be bound to this namespace URI.

2.3 Error Handling

2.3.1 Kinds of Errors

As described in **2.2.3 Expression Processing**, XPath defines a static analysis phase, which does not depend on input data, and a dynamic evaluation phase, which does depend on input data. Errors may be raised during each phase.

[Definition: A **static error** is an error that must be detected during the static analysis phase. A syntax error is an example of a static error.]

[Definition: A **dynamic error** is an error that must be detected during the dynamic evaluation phase and may be detected during the static analysis phase. Numeric overflow is an example of a dynamic error.]

[Definition: A **type error** may be raised during the static analysis phase or the dynamic evaluation phase. During the static analysis phase, a type error occurs when the static type of an expression does not match the expected type of the context in which the expression occurs. During the dynamic evaluation phase, a type error occurs when the dynamic type of a value does not match the expected type of the context in which the value occurs.]

The outcome of the static analysis phase is either success or one or more type errors, static errors, or statically-detected dynamic errors. The result of the dynamic evaluation phase is either a result value, a type error, or a dynamic error.

If more than one error is present, or if an error condition comes within the scope of more than one error defined in this specification, then any non-empty subset of these errors may be reported.

During the static analysis phase, if the Static Typing Feature is in effect and the static type assigned to an expression other than `()` or `data(())` is `empty-sequence()`, a static error is raised [err:XPST0005]. This catches cases in which a query refers to an element or attribute that is not present in the in-scope schema definitions, possibly because of a spelling error.

Independently of whether the Static Typing Feature is in effect, if an implementation can determine during the static analysis phase that an expression, if evaluated, would necessarily raise a type error or a dynamic error, the implementation may (but is not required to) report that error during the static analysis phase. However, the `fn:error()` function must not be evaluated during the static analysis phase.

[Definition: In addition to static errors, dynamic errors, and type errors, an XPath implementation may raise **warnings**, either during the static analysis phase or the dynamic evaluation phase. The circumstances in which warnings are raised, and the ways in which warnings are handled, are implementation-defined.]

In addition to the errors defined in this specification, an implementation may raise a dynamic error for a reason beyond the scope of this specification. For example, limitations may exist on the maximum numbers or sizes of various objects. Any such limitations, and the consequences of exceeding them, are implementation-dependent.

2.3.2 Identifying and Reporting Errors

The errors defined in this specification are identified by QNames that have the form `err:XPYYnnnn`, where:

- `err` denotes the namespace for XPath and XQuery errors, `http://www.w3.org/2005/xqt-errors`. This binding of the namespace prefix `err` is used for convenience in this document, and is not normative.
- `XP` identifies the error as an XPath error.
- `YY` denotes the error category, using the following encoding:
  - `ST` denotes a static error.
  - `DY` denotes a dynamic error.
  - `TY` denotes a type error.
- `nnnn` is a unique numeric code.

> **Note:**
>
> The namespace URI for XPath and XQuery errors is not expected to change from one version of XPath to another. However, the contents of this namespace may be extended to include additional error definitions.

The method by which an XPath processor reports error information to the external environment is implementation-defined.

An error can be represented by a URI reference that is derived from the error QName as follows: an error with namespace URI *`NS`* and local part *`LP`* can be represented as the URI reference *`NS`*`#`*`LP`*. For example, an error whose QName is `err:XPST0017` could be represented as `http://www.w3.org/2005/xqt-errors#XPST0017`.

> **Note:**
>
> Along with a code identifying an error, implementations may wish to return additional information, such as the location of the error or the processing phase in which it was detected. If an implementation chooses to do so, then the mechanism that it uses to return this information is implementation-defined.

2.3.3 Handling Dynamic Errors

Except as noted in this document, if any operand of an expression raises a dynamic error, the expression also raises a dynamic error. If an expression can validly return a value or raise a dynamic error, the implementation may choose to return the value or raise the dynamic error. For example, the logical expression `expr1 and expr2` may return the value `false` if either operand returns `false`, or may raise a dynamic error if either operand raises a dynamic error.

If more than one operand of an expression raises an error, the implementation may choose which error is raised by the expression. For example, in this expression:

```
($x div $y) + xs:decimal($z)
```

both the sub-expressions `($x div $y)` and `xs:decimal($z)` may raise an error. The implementation may choose which error is raised by the "`+`" expression. Once one operand raises an error, the implementation is not required, but is permitted, to evaluate any other operands.

[Definition: In addition to its identifying QName, a dynamic error may also carry a descriptive string and one or more additional values called **error values**.] An implementation may provide a mechanism whereby an application-defined error handler can process error values and produce diagnostic messages.

A dynamic error may be raised by a built-in function or operator. For example, the `div` operator raises an error if its operands are `xs:decimal` values and its second operand is equal to zero. Errors raised by built-in functions and operators are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

A dynamic error can also be raised explicitly by calling the `fn:error` function, which only raises an error and never returns a value. This function is defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)]. For example, the following function call raises a dynamic error, providing a QName that identifies the error, a descriptive string, and a diagnostic value (assuming that the prefix `app` is bound to a namespace containing application-defined error codes):

```
fn:error(xs:QName("app:err057"), "Unexpected value", fn:string($v))
```

2.3.4 Errors and Optimization

Because different implementations may choose to evaluate or optimize an expression in different ways, certain aspects of the detection and reporting of dynamic errors are implementation-dependent, as described in this section.

An implementation is always free to evaluate the operands of an operator in any order.

In some cases, a processor can determine the result of an expression without accessing all the data that would be implied by the formal expression semantics. For example, the formal description of filter expressions suggests that `$s[1]` should be evaluated by examining all the items in sequence `$s`, and selecting all those that satisfy the predicate `position()=1`. In practice, many implementations will recognize that they can evaluate this expression by taking the first item in the sequence and then exiting. If `$s` is defined by an expression such as `//book[author eq 'Berners-Lee']`, then this strategy may avoid a complete scan of a large document and may therefore greatly improve performance. However, a consequence of this strategy is that a dynamic error or type error that would be detected if the expression semantics were followed literally might not be detected at all if the evaluation exits early. In this example, such an error might occur if there is a `book` element in the input data with more than one `author` subelement.

The extent to which a processor may optimize its access to data, at the cost of not detecting errors, is defined by the following rules.

Consider an expression *Q* that has an operand (sub-expression) *E*. In general the value of *E* is a sequence. At an intermediate stage during evaluation of the sequence, some of its items will be known and others will be unknown. If, at such an intermediate stage of evaluation, a processor is able to establish that there are only two possible outcomes of evaluating *Q*, namely the value *V* or an error, then the processor may deliver the result *V* without evaluating further items in the operand *E*. For this purpose, two values are considered to represent the same outcome if their items are pairwise the same, where nodes are the same if they have the same identity, and values are the same if they are equal and have exactly the same type.

There is an exception to this rule: If a processor evaluates an operand *E* (wholly or in part), then it is required to establish that the actual value of the operand *E* does not violate any constraints on its cardinality. For example, the expression `$e eq 0` results in a type error if the value of `$e` contains two or more items. A processor is not allowed to decide, after evaluating the first item in the value of `$e` and finding it equal to zero, that the only possible outcomes are the value `true` or a type error caused by the cardinality violation. It must establish that the value of `$e` contains no more than one item.

These rules apply to all the operands of an expression considered in combination: thus if an expression has two operands *E1* and *E2*, it may be evaluated using any samples of the respective sequences that satisfy the above rules.

The rules cascade: if *A* is an operand of *B* and *B* is an operand of *C*, then the processor needs to evaluate only a sufficient sample of *B* to determine the value of *C*, and needs to evaluate only a sufficient sample of *A* to determine this sample of *B*.

The effect of these rules is that the processor is free to stop examining further items in a sequence as soon as it can establish that further items would not affect the result except possibly by causing an error. For example, the processor may return `true` as the result of the expression `S1 = S2` as soon as it finds a pair of equal values from the two sequences.

Another consequence of these rules is that where none of the items in a sequence contributes to the result of an expression, the processor is not obliged to evaluate any part of the sequence. Again, however, the processor cannot dispense with a required cardinality check: if an empty sequence is not permitted in the relevant context, then the processor must ensure that the operand is not an empty sequence.

Examples:

- If an implementation can find (for example, by using an index) that at least one item returned by `$expr1` in the following example has the value `47`, it is allowed to return `true` as the result of the `some` expression, without searching for another item returned by `$expr1` that would raise an error if it were evaluated.

```
some $x in $expr1 satisfies $x = 47
```

- In the following example, if an implementation can find (for example, by using an index) the `product` element-nodes that have an `id` child with the value `47`, it is allowed to return these nodes as the result of the path expression, without searching for another `product` node that would raise an error because it has an `id` child whose value is not an integer.

```
//product[id = 47]
```

For a variety of reasons, including optimization, implementations may rewrite expressions into a different form. There are a number of rules that limit the extent of this freedom:

- Other than the raising or not raising of errors, the result of evaluating a rewritten expression must conform to the semantics defined in this specification for the original expression.

  Note:

  This allows an implementation to return a result in cases where the original expression would have raised an error, or to raise an error in cases where the original expression would have returned a result. The main cases where this is likely to arise in practice are (a) where a rewrite changes the order of evaluation, such that a subexpression causing an error is evaluated when the expression is written one way and is not evaluated when the expression is written a different way, and (b) where intermediate results of the evaluation cause overflow or other out-of-range conditions.

  Note:

  This rule does not mean that the result of the expression will always be the same in non-error cases as if it had not been rewritten, because there are many cases where the result of an expression is to some degree implementation-dependent or implementation-defined.

- Conditional and typeswitch expressions must not raise a dynamic error in respect of subexpressions occurring in a branch that is not selected, and must not return the value delivered by a branch unless that branch is selected. Thus, the following example must not raise a dynamic error if the document `abc.xml` does not exist:

```
if (doc-available('abc.xml')) then doc('abc.xml') else ()
```

- As stated earlier, an expression must not be rewritten to dispense with a required cardinality check: for example, `string-length(//title)` must raise an error if the document contains more than one title element.

- Expressions must not be rewritten in such a way as to create or remove static errors. For example, there is a rule that in casting a string to a QName the operand must be a string literal. This rule applies to the original expression and not to any rewritten form of the expression.

Expression rewrite is illustrated by the following examples.

- Consider the expression `//part[color eq "Red"]`. An implementation might choose to rewrite this expression as `//part[color = "Red"][color eq "Red"]`. The implementation might then process the expression as follows: First process the "`=`" predicate by probing an index on parts by color to quickly find all the parts that have a Red color; then process the "`eq`" predicate by checking each of these parts to make sure it has only a single color. The result would be as follows:
  - Parts that have exactly one color that is Red are returned.
  - If some part has color Red together with some other color, an error is raised.
  - The existence of some part that has no color Red but has multiple non-Red colors does not trigger an error.
- The expression in the following example cannot raise a casting error if it is evaluated exactly as written (i.e., left to right). Since neither predicate depends on the context position, an implementation might choose to reorder the predicates to achieve better performance (for example, by taking advantage of an index). This reordering could cause the expression to raise an error.

  ```
  $N[@x castable as xs:date][xs:date(@x) gt xs:date("2000-01-01")]
  ```

  To avoid unexpected errors caused by expression rewrite, tests that are designed to prevent dynamic errors should be expressed using conditional expressions. For example, the above expression can be written as follows:

  ```
  $N[if (@x castable as xs:date)
     then xs:date(@x) gt xs:date("2000-01-01")
     else false()]
  ```

2.4 Concepts

This section explains some concepts that are important to the processing of XPath expressions.

2.4.1 Document Order

An ordering called **document order** is defined among all the nodes accessible during processing of a given expression, which may consist of one or more **trees** (documents or fragments). Document order is defined in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)], and its definition is repeated here for convenience. [Definition: The node ordering that is the reverse of document order is called **reverse document order**.]

Document order is a total ordering, although the relative order of some nodes is implementation-dependent. [Definition: 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 expression, 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 node with which they are associated. 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 is before any node in a different tree T2, then all nodes in tree T1 are before all nodes in tree T2.

2.4.2 Atomization

The semantics of some XPath operators depend on a process called atomization. Atomization is applied to a value when the value is used in a context in which a sequence of atomic values is required. The result of atomization is either a sequence of atomic values or a type error [err:FOTY0012]. [Definition: **Atomization** of a sequence is defined as the result of invoking the `fn:data` function on the sequence, as defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].]

The semantics of `fn:data` are repeated here for convenience. The result of `fn:data` is the sequence of atomic values produced by applying the following rules to each item in the input sequence:

- If the item is an atomic value, it is returned.
- If the item is a node, its typed value is returned (err:FOTY0012 is raised if the node has no typed value.)

Atomization is used in processing the following types of expressions:

- Arithmetic expressions
- Comparison expressions
- Function calls and returns
- Cast expressions

2.4.3 Effective Boolean Value

Under certain circumstances (listed below), it is necessary to find the effective boolean value of a value. [Definition: The **effective boolean value** of a value is defined as the result of applying the `fn:boolean` function to the value, as defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].]

The dynamic semantics of `fn:boolean` are repeated here for convenience:

1. If its operand is an empty sequence, `fn:boolean` returns `false`.
2. If its operand is a sequence whose first item is a node, `fn:boolean` returns `true`.
3. If its operand is a singleton value of type `xs:boolean` or derived from `xs:boolean`, `fn:boolean` returns the value of its operand unchanged.
4. If its operand is a singleton value of type `xs:string`, `xs:anyURI`, `xs:untypedAtomic`, or a type derived from one of these, `fn:boolean` returns `false` if the operand value has zero length; otherwise it returns `true`.
5. If its operand is a singleton value of any numeric type or derived from a numeric type, `fn:boolean` returns `false` if the operand value is `NaN` or is numerically equal to zero; otherwise it returns `true`.
6. In all other cases, `fn:boolean` raises a type error [err:FORG0006].

Note:

The static semantics of `fn:boolean` are defined in Section 7.2.4 The fn:boolean and fn:not functions[FS].

The effective boolean value of a sequence is computed implicitly during processing of the following types of expressions:

- Logical expressions (`and`, `or`)
- The `fn:not` function
- Certain types of predicates, such as `a[b]`
- Conditional expressions (`if`)
- Quantified expressions (`some`, `every`)
- General comparisons, in XPath 1.0 compatibility mode.

Note:

The definition of effective boolean value is *not* used when casting a value to the type `xs:boolean`, for example in a `cast` expression or when passing a value to a function whose expected parameter is of type `xs:boolean`.

2.4.4 Input Sources

XPath has a set of functions that provide access to input data. These functions are of particular importance because they provide a way in which an expression can reference a document or a collection of documents. The input functions are described informally here; they are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

An expression can access input data either by calling one of the input functions or by referencing some part of the dynamic context that is initialized by the external environment, such as a variable or context item.

The input functions supported by XPath are as follows:

- The `fn:doc` function takes a string containing a URI. If that URI is associated with a document in available documents, `fn:doc` returns a document node whose content is the data model representation of the given document; otherwise it raises a dynamic error (see [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)] for details).
- The `fn:collection` function with one argument takes a string containing a URI. If that URI is associated with a collection in available collections, `fn:collection` returns the data model representation of that collection; otherwise it raises a dynamic error (see [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)] for details). A collection may be any sequence of nodes. For example, the expression `fn:collection("http://example.org")//customer` identifies all the `customer` elements that are descendants of nodes found in the collection whose URI is `http://example.org`.
- The `fn:collection` function with zero arguments returns the default collection, an implementation-dependent sequence of nodes.

2.5 Types

The type system of XPath is based on [XML Schema], and is formally defined in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)].

[Definition: A **sequence type** is a type that can be expressed using the SequenceType syntax. Sequence types are used whenever it is necessary to refer to a type in an XPath expression. The term **sequence type** suggests that this syntax is used to describe the type of an XPath value, which is always a sequence.]

[Definition: A **schema type** is a type that is (or could be) defined using the facilities of [XML Schema] (including the built-in types of [XML Schema]).] A schema type can be used as a type annotation on an element or attribute node (unless it is a non-instantiable type such as `xs:NOTATION` or `xs:anyAtomicType`, in which case its derived types can be so used). Every schema type is either a **complex type** or a **simple type**; simple types are further subdivided into **list types**, **union types**, and **atomic types** (see [XML Schema] for definitions and explanations of these terms.)

Atomic types represent the intersection between the categories of sequence type and schema type. An atomic type, such as `xs:integer` or `my:hatsize`, is both a sequence type and a schema type.

2.5.1 Predefined Schema Types

The in-scope schema types in the static context are initialized with a set of predefined schema types that is determined by the host language. This set may include some or all of the schema types in the namespace `http://www.w3.org/2001/XMLSchema`, represented in this document by the namespace prefix `xs`. The schema types in this namespace are defined in [XML Schema] and augmented by additional types defined in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)]. The schema types defined in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)] are summarized below.

1. [Definition: `xs:untyped` is used as the type annotation of an element node that has not been validated, or has been validated in `skip` mode.] No predefined schema types are derived from `xs:untyped`.
2. [Definition: `xs:untypedAtomic` is an atomic type that is used to denote 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 annotation `xs:untypedAtomic`. No predefined schema types are derived from `xs:untypedAtomic`.

3. [Definition: `xs:dayTimeDuration` is derived by restriction from `xs:duration`. The lexical representation of `xs:dayTimeDuration` is restricted to contain only day, hour, minute, and second components.]
4. [Definition: `xs:yearMonthDuration` is derived by restriction from `xs:duration`. The lexical representation of `xs:yearMonthDuration` is restricted to contain only year and month components.]
5. [Definition: `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.]

   Note:

   `xs:anyAtomicType` will not appear as the type of an actual value in an XDM instance.

The relationships among the schema types in the `xs` namespace are illustrated in Figure 2. A more complete description of the XPath type hierarchy can be found in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

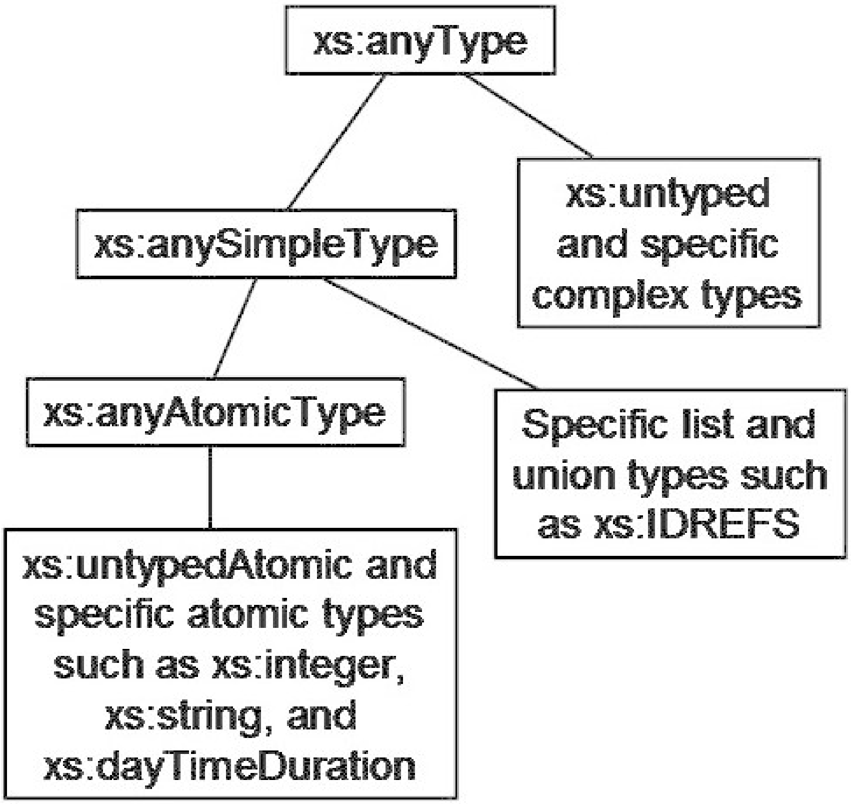

Figure 2: Hierarchy of Schema Types used in XPath

2.5.2 Typed Value and String Value

Every node has a **typed value** and a **string value**. [Definition: The **typed value** of a node is a sequence of atomic values and can be extracted by applying the `fn:data` function to the node.] [Definition: The **string value** of a node is a string and can be extracted by applying the `fn:string` function to the node.] Definitions of `fn:data` and `fn:string` can be found in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

An implementation may store both the typed value and the string value of a node, or it may store only one of these and derive the other as needed. The string value of a node must be a valid lexical representation of the typed value of the node, but the node is not required to preserve the string representation from the original source document. For example, if the typed value of a node is the `xs:integer` value `30`, its string value might be "`30`" or "`0030`".

The typed value, string value, and type annotation of a node are closely related. If the node was created by mapping from an Infoset or PSVI, the relationships among these properties are defined by rules in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)].

As a convenience to the reader, the relationship between typed value and string value for various kinds of nodes is summarized and illustrated by examples below.

1. For text and document nodes, the typed value of the node is the same as its string value, as an instance of the type `xs:untypedAtomic`. The string value of a document node is formed by concatenating the string values of all its descendant text nodes, in document order.
2. The typed value of a comment, namespace, or processing instruction node is the same as its string value. It is an instance of the type `xs:string`.
3. The typed value of an attribute node with the type annotation `xs:anySimpleType` or `xs:untypedAtomic` is the same as its string value, as an instance of `xs:untypedAtomic`. The typed value of an attribute node with any other type annotation is derived from its string value and type annotation using the lexical-to-value-space mapping defined in [XML Schema] Part 2 for the relevant type.

   Example: A1 is an attribute having string value `"3.14E-2"` and type annotation `xs:double`. The typed value of A1 is the `xs:double` value whose lexical representation is `3.14E-2`.

   Example: A2 is an attribute with type annotation `xs:IDREFS`, which is a list datatype whose item type is the atomic datatype `xs:IDREF`. Its string value is "`bar baz faz`". The typed value of A2 is a sequence of three atomic values ("`bar`", "`baz`", "`faz`"), each of type `xs:IDREF`. The typed value of a node is never treated as an instance of a named list type. Instead, if the type annotation of a node is a list type (such as `xs:IDREFS`), its typed value is treated as a sequence of the atomic type from which it is derived (such as `xs:IDREF`).
4. For an element node, the relationship between typed value and string value depends on the node's type annotation, as follows:
   a. If the type annotation is `xs:untyped` or `xs:anySimpleType` or denotes a complex type with mixed content (including `xs:anyType`), then the typed value of the node is equal to its string value, as an instance of `xs:untypedAtomic`. However, if the `nilled` property of the node is `true`, then its typed value is the empty sequence.

      Example: E1 is an element node having type annotation `xs:untyped` and string value "`1999-05-31`". The typed value of E1 is "`1999-05-31`", as an instance of `xs:untypedAtomic`.

      Example: E2 is an element node with the type annotation `formula`, which is a complex type with mixed content. The content of E2 consists of the character "`H`", a child element named `subscript` with string value "`2`", and the character "`O`". The typed value of E2 is "`H2O`" as an instance of `xs:untypedAtomic`.
   b. If the type annotation denotes a simple type or a complex type with simple content, then the typed value of the node is derived from its string value and its type annotation in a way that is consistent with schema validation. However, if the `nilled` property of the node is `true`, then its typed value is the empty sequence.

      Example: E3 is an element node with the type annotation `cost`, which is a complex type that has several attributes and a simple content type of `xs:decimal`. The string value of E3 is "`74.95`". The typed value of E3 is `74.95`, as an instance of `xs:decimal`.

      Example: E4 is an element node with the type annotation `hatsizelist`, which is a simple type derived from the atomic type `hatsize`, which in turn is derived from `xs:integer`. The string value of E4 is "`7 8 9`". The typed value of E4 is a sequence of three values (`7`, `8`, `9`), each of type `hatsize`.

      Example: E5 is an element node with the type annotation `my:integer-or-string` which is a union type with member types `xs:integer` and `xs:string`. The string value of E5 is "`47`". The typed value of E5 is `47` as an `xs:integer`, since `xs:integer` is the member type that validated the content of E5. In general, when the type annotation of a node is a union type, the typed value of the node will be an instance of one of the member types of the union.

      Note:

      If an implementation stores only the string value of a node, and the type annotation of the node is a union type, the implementation must be able to deliver the typed value of the node as an instance of the appropriate member type.
   c. If the type annotation denotes a complex type with empty content, then the typed value of the node is the empty sequence and its string value is the zero-length string.
   d. If the type annotation denotes a complex type with element-only content, then the typed value of the node is undefined. The `fn:data` function raises a type error [err:FOTY0012] when applied to

such a node. The string value of such a node is equal to the concatenated string values of all its text node descendants, in document order.

Example: E6 is an element node with the type annotation `weather`, which is a complex type whose content type specifies `element-only`. E6 has two child elements named `temperature` and `precipitation`. The typed value of E6 is undefined, and the `fn:data` function applied to E6 raises an error.

2.5.3 SequenceType Syntax

Whenever it is necessary to refer to a type in an XPath expression, the SequenceType syntax is used.

[50]	`SequenceType`	::=	`("empty-sequence" "(" ")")` `| (ItemType OccurrenceIndicator?)`
[52]	`ItemType`	::=	`KindTest | ("item" "(" ")") | AtomicType`
[51]	`OccurrenceIndicator`	::=	`"?" | "*" | "+"`
[53]	`AtomicType`	::=	`QName`

[54]	`KindTest`	::=	`DocumentTest`
			`| ElementTest`
			`| AttributeTest`
			`| SchemaElementTest`
			`| SchemaAttributeTest`
			`| PITest`
			`| CommentTest`
			`| TextTest`
			`| AnyKindTest`

[56]	`DocumentTest`	::=	`"document-node" "(" (ElementTest | SchemaElementTest)? ")"`
[64]	`ElementTest`	::=	`"element" "(" (ElementNameOrWildcard ("," TypeName "?"?)?)? ")"`
[66]	`SchemaElementTest`	::=	`"schema-element" "(" ElementDeclaration ")"`
[67]	`ElementDeclaration`	::=	`ElementName`
[60]	`AttributeTest`	::=	`"attribute" "(" (AttribNameOrWildcard ("," TypeName)?)? ")"`
[62]	`SchemaAttributeTest`	::=	`"schema-attribute" "(" AttributeDeclaration ")"`
[63]	`AttributeDeclaration`	::=	`AttributeName`
[65]	`ElementNameOrWildcard`	::=	`ElementName | "*"`
[69]	`ElementName`	::=	`QName`
[61]	`AttribNameOrWildcard`	::=	`AttributeName | "*"`
[68]	`AttributeName`	::=	`QName`
[70]	`TypeName`	::=	`QName`
[59]	`PITest`	::=	`"processing-instruction" "(" (NCName | StringLiteral)? ")"`
[58]	`CommentTest`	::=	`"comment" "(" ")"`
[57]	`TextTest`	::=	`"text" "(" ")"`
[55]	`AnyKindTest`	::=	`"node" "(" ")"`

With the exception of the special type `empty-sequence()`, a sequence type consists of an **item type** that constrains the type of each item in the sequence, and a **cardinality** that constrains the number of items in the sequence. Apart from the item type `item()`, which permits any kind of item, item types divide into **node types** (such as `element()`) and **atomic types** (such as `xs:integer`).

Item types representing element and attribute nodes may specify the required type annotations of those nodes, in the form of a schema type. Thus the item type `element(*, us:address)` denotes any element node whose type annotation is (or is derived from) the schema type named `us:address`.

Here are some examples of sequence types that might be used in XPath expressions:

- `xs:date` refers to the built-in atomic schema type named `xs:date`
- `attribute()?` refers to an optional attribute node

- `element()` refers to any element node
- `element(po:shipto, po:address)` refers to an element node that has the name `po:shipto` and has the type annotation `po:address` (or a schema type derived from `po:address`)
- `element(*, po:address)` refers to an element node of any name that has the type annotation `po:address` (or a type derived from `po:address`)
- `element(customer)` refers to an element node named `customer` with any type annotation
- `schema-element(customer)` refers to an element node whose name is `customer` (or is in the substitution group headed by `customer`) and whose type annotation matches the schema type declared for a `customer` element in the in-scope element declarations
- `node()*` refers to a sequence of zero or more nodes of any kind
- `item()+` refers to a sequence of one or more nodes or atomic values

2.5.4 SequenceType Matching

[Definition: During evaluation of an expression, it is sometimes necessary to determine whether a value with a known dynamic type "matches" an expected sequence type. This process is known as **SequenceType matching**.] For example, an `instance of` expression returns `true` if the dynamic type of a given value matches a given sequence type, or `false` if it does not.

QNames appearing in a sequence type have their prefixes expanded to namespace URIs by means of the statically known namespaces and (where applicable) the default element/type namespace. An unprefixed attribute QName is in no namespace. Equality of QNames is defined by the `eq` operator.

The rules for SequenceType matching compare the dynamic type of a value with an expected sequence type. These rules are a subset of the formal rules that match a value with an expected type defined in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)], because the Formal Semantics must be able to match values against types that are not expressible using the SequenceType syntax.

Some of the rules for SequenceType matching require determining whether a given schema type is the same as or derived from an expected schema type. The given schema type may be "known" (defined in the in-scope schema definitions), or "unknown" (not defined in the in-scope schema definitions). An unknown schema type might be encountered, for example, if a source document has been validated using a schema that was not imported into the static context. In this case, an implementation is allowed (but is not required) to provide an implementation-dependent mechanism for determining whether the unknown schema type is derived from the expected schema type. For example, an implementation might maintain a data dictionary containing information about type hierarchies.

[Definition: The use of a value whose dynamic type is derived from an expected type is known as **subtype substitution**.] Subtype substitution does not change the actual type of a value. For example, if an `xs:integer` value is used where an `xs:decimal` value is expected, the value retains its type as `xs:integer`.

The definition of SequenceType matching relies on a pseudo-function named `derives-from(`*AT, ET*`)`, which takes an actual simple or complex schema type *AT* and an expected simple or complex schema type *ET*, and either returns a boolean value or raises a type error [err:XPTY0004]. The pseudo-function `derives-from` is defined below and is defined formally in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)].

- `derives-from(`*AT*, *ET*`)` returns `true` if *ET* is a known type and any of the following three conditions is true:
  1. *AT* is a schema type found in the in-scope schema definitions, and is the same as *ET* or is derived by restriction or extension from *ET*
  2. *AT* is a schema type not found in the in-scope schema definitions, and an implementation-dependent mechanism is able to determine that *AT* is derived by restriction from *ET*
  3. There exists some schema type *IT* such that `derives-from(`*IT, ET*`)` and `derives-from(`*AT, IT*`)` are true.
- `derives-from(`*AT*, *ET*`)` returns `false` if *ET* is a known type and either the first and third or the second and third of the following conditions are true:
  1. *AT* is a schema type found in the in-scope schema definitions, and is not the same as *ET*, and is not derived by restriction or extension from *ET*

2. *AT* is a schema type not found in the in-scope schema definitions, and an implementation-dependent mechanism is able to determine that *AT* is not derived by restriction from *ET*
3. No schema type *IT* exists such that `derives-from(`*IT, ET*`)` and `derives-from(`*AT, IT*`)` are true.

- `derives-from(`*AT*, *ET*`)` raises a type error [err:XPTY0004] if:
  1. *ET* is an unknown type, or
  2. *AT* is an unknown type, and the implementation is not able to determine whether *AT* is derived by restriction from *ET*.

The rules for SequenceType matching are given below, with examples (the examples are for purposes of illustration, and do not cover all possible cases).

2.5.4.1 Matching a SequenceType and a Value

- The sequence type `empty-sequence()` matches a value that is the empty sequence.
- An ItemType with no OccurrenceIndicator matches any value that contains exactly one item if the ItemType matches that item (see **2.5.4.2 Matching an ItemType and an Item**).
- An ItemType with an OccurrenceIndicator matches a value if the number of items in the value matches the OccurrenceIndicator and the ItemType matches each of the items in the value.

An OccurrenceIndicator specifies the number of items in a sequence, as follows:

- `?` matches zero or one items
- `*` matches zero or more items
- `+` matches one or more items

As a consequence of these rules, any sequence type whose OccurrenceIndicator is `*` or `?` matches a value that is an empty sequence.

2.5.4.2 Matching an ItemType and an Item

- An ItemType consisting simply of a QName is interpreted as an AtomicType. An AtomicType *AtomicType* matches an atomic value whose actual type is *AT* if `derives-from(`*AT, AtomicType*`)` is `true`. If a QName that is used as an AtomicType is not defined as an atomic type in the in-scope schema types, a static error is raised [err:XPST0051].

  Example: The AtomicType `xs:decimal` matches the value `12.34` (a decimal literal). `xs:decimal` also matches a value whose type is `shoesize`, if `shoesize` is an atomic type derived by restriction from `xs:decimal`.

  Note:

  The names of non-atomic types such as `xs:IDREFS` are not accepted in this context, but can often be replaced by an atomic type with an occurrence indicator, such as `xs:IDREF+`.
- `item()` matches any single item.

  Example: `item()` matches the atomic value `1` or the element `<a/>`.
- `node()` matches any node.
- `text()` matches any text node.
- `processing-instruction()` matches any processing-instruction node.
- `processing-instruction(`*N*`)` matches any processing-instruction node whose PITarget is equal to `fn:normalize-space(N)`. If `fn:normalize-space(N)` is not in the lexical space of NCName, a type error is raised [err:XPTY0004]

  Example: `processing-instruction(xml-stylesheet)` matches any processing instruction whose PITarget is `xml-stylesheet`.

  For backward compatibility with XPath 1.0, the PITarget of a processing instruction may also be expressed as a string literal, as in this example: `processing-instruction("xml-stylesheet")`.
- `comment()` matches any comment node.
- `document-node()` matches any document node.
- `document-node(`*E*`)` matches any document node that contains exactly one element node, optionally

accompanied by one or more comment and processing instruction nodes, if *E* is an ElementTest or SchemaElementTest that matches the element node (see **2.5.4.3 Element Test** and **2.5.4.4 Schema Element Test**).

Example: `document-node(element(book))` matches a document node containing exactly one element node that is matched by the ElementTest `element(book)`.

- An ItemType that is an ElementTest, SchemaElementTest, AttributeTest, or SchemaAttributeTest matches an element or attribute node as described in the following sections.

2.5.4.3 Element Test

An ElementTest is used to match an element node by its name and/or type annotation. An ElementTest may take any of the following forms. In these forms, ElementName need not be present in the in-scope element declarations, but TypeName must be present in the in-scope schema types [err:XPST0008]. Note that substitution groups do not affect the semantics of ElementTest.

1. `element()` and `element(*)` match any single element node, regardless of its name or type annotation.
2. `element(`ElementName`)` matches any element node whose name is ElementName, regardless of its type annotation or `nilled` property.

   Example: `element(person)` matches any element node whose name is `person`.
3. `element(`ElementName`,` TypeName`)` matches an element node whose name is ElementName if `derives-from(`*AT*, TypeName `)` is `true`, where *AT* is the type annotation of the element node, and the `nilled` property of the node is `false`.

   Example: `element(person, surgeon)` matches a non-nilled element node whose name is `person` and whose type annotation is `surgeon` (or is derived from `surgeon`).
4. `element(`ElementName, TypeName `?)` matches an element node whose name is ElementName if `derives-from(`*AT*, TypeName`)` is `true`, where *AT* is the type annotation of the element node. The `nilled` property of the node may be either `true` or `false`.

   Example: `element(person, surgeon?)` matches a nilled or non-nilled element node whose name is `person` and whose type annotation is `surgeon` (or is derived from `surgeon`).
5. `element(*,` TypeName`)` matches an element node regardless of its name, if `derives-from(`*AT*, TypeName `)` is `true`, where *AT* is the type annotation of the element node, and the `nilled` property of the node is `false`.

   Example: `element(*, surgeon)` matches any non-nilled element node whose type annotation is `surgeon` (or is derived from `surgeon`), regardless of its name.
6. `element(*,` TypeName `?)` matches an element node regardless of its name, if `derives-from(`*AT*, TypeName `)` is `true`, where *AT* is the type annotation of the element node. The `nilled` property of the node may be either `true` or `false`.

   Example: `element(*, surgeon?)` matches any nilled or non-nilled element node whose type annotation is `surgeon` (or is derived from `surgeon`), regardless of its name.

2.5.4.4 Schema Element Test

A SchemaElementTest matches an element node against a corresponding element declaration found in the in-scope element declarations. It takes the following form:

`schema-element(`ElementName`)`

If the ElementName specified in the SchemaElementTest is not found in the in-scope element declarations, a static error is raised [err:XPST0008].

A SchemaElementTest matches a candidate element node if all three of the following conditions are satisfied:

1. The name of the candidate node matches the specified ElementName or matches the name of an element in a substitution group headed by an element named ElementName.
2. `derives-from(`*AT, ET*`)` is `true`, where *AT* is the type annotation of the candidate node and *ET* is the schema type declared for element ElementName in the in-scope element declarations.

3. If the element declaration for ElementName in the in-scope element declarations is not `nillable`, then the `nilled` property of the candidate node is `false`.

Example: The SchemaElementTest `schema-element(customer)` matches a candidate element node if `customer` is a top-level element declaration in the in-scope element declarations, the name of the candidate node is `customer` or is in a substitution group headed by `customer`, the type annotation of the candidate node is the same as or derived from the schema type declared for the `customer` element, and either the candidate node is not `nilled` or `customer` is declared to be `nillable`.

2.5.4.5 Attribute Test

An AttributeTest is used to match an attribute node by its name and/or type annotation. An AttributeTest any take any of the following forms. In these forms, AttributeName need not be present in the in-scope attribute declarations, but TypeName must be present in the in-scope schema types [err:XPST0008].

1. `attribute()` and `attribute(*)` match any single attribute node, regardless of its name or type annotation.
2. `attribute(`AttributeName`)` matches any attribute node whose name is AttributeName, regardless of its type annotation.

   Example: `attribute(price)` matches any attribute node whose name is `price`.
3. `attribute(`AttributeName, TypeName`)` matches an attribute node whose name is AttributeName if `derives-from(`*AT*, TypeName `)` is `true`, where *AT* is the type annotation of the attribute node.

   Example: `attribute(price, currency)` matches an attribute node whose name is `price` and whose type annotation is `currency` (or is derived from `currency`).
4. `attribute(*, `TypeName`)` matches an attribute node regardless of its name, if `derives-from(`*AT*, TypeName`)` is `true`, where *AT* is the type annotation of the attribute node.

   Example: `attribute(*, currency)` matches any attribute node whose type annotation is `currency` (or is derived from `currency`), regardless of its name.

2.5.4.6 Schema Attribute Test

A SchemaAttributeTest matches an attribute node against a corresponding attribute declaration found in the in-scope attribute declarations. It takes the following form:

`schema-attribute(`AttributeName`)`

If the AttributeName specified in the SchemaAttributeTest is not found in the in-scope attribute declarations, a static error is raised [err:XPST0008].

A SchemaAttributeTest matches a candidate attribute node if both of the following conditions are satisfied:

1. The name of the candidate node matches the specified AttributeName.
2. `derives-from(`*AT, ET*`)` is `true`, where *AT* is the type annotation of the candidate node and *ET* is the schema type declared for attribute AttributeName in the in-scope attribute declarations.

Example: The SchemaAttributeTest `schema-attribute(color)` matches a candidate attribute node if `color` is a top-level attribute declaration in the in-scope attribute declarations, the name of the candidate node is `color`, and the type annotation of the candidate node is the same as or derived from the schema type declared for the `color` attribute.

2.6 Comments

[77]	`Comment`	::=	`"(:" (CommentContents | Comment)* ":)"`
[82]	`CommentContents`	::=	`(Char+ - (Char* ('(:' | ':)') Char*))`

Comments may be used to provide informative annotation for an expression. Comments are lexical constructs only, and do not affect expression processing.

Comments are strings, delimited by the symbols `(:` and `:)`. Comments may be nested.

A comment may be used anywhere ignorable whitespace is allowed (see **A.2.4.1 Default Whitespace Handling**).

The following is an example of a comment:

```
(: Houston, we have a problem :)
```

3 Expressions

This section discusses each of the basic kinds of expression. Each kind of expression has a name such as `PathExpr`, which is introduced on the left side of the grammar production that defines the expression. Since XPath is a composable language, each kind of expression is defined in terms of other expressions whose operators have a higher precedence. In this way, the precedence of operators is represented explicitly in the grammar.

The order in which expressions are discussed in this document does not reflect the order of operator precedence. In general, this document introduces the simplest kinds of expressions first, followed by more complex expressions. For the complete grammar, see Appendix [**A XPath Grammar**].

The highest-level symbol in the XPath grammar is XPath.

[1]	`XPath`	::=	`Expr`
[2]	`Expr`	::=	`ExprSingle ("," ExprSingle)*`

[3]	`ExprSingle`	::=	`ForExpr`
			`| QuantifiedExpr`
			`| IfExpr`
			`| OrExpr`

The XPath operator that has lowest precedence is the comma operator, which is used to combine two operands to form a sequence. As shown in the grammar, a general expression (Expr) can consist of multiple ExprSingle operands, separated by commas. The name ExprSingle denotes an expression that does not contain a top-level comma operator (despite its name, an ExprSingle may evaluate to a sequence containing more than one item.)

The symbol ExprSingle is used in various places in the grammar where an expression is not allowed to contain a top-level comma. For example, each of the arguments of a function call must be an ExprSingle, because commas are used to separate the arguments of a function call.

After the comma, the expressions that have next lowest precedence are ForExpr, QuantifiedExpr, IfExpr, and OrExpr. Each of these expressions is described in a separate section of this document.

3.1 Primary Expressions

[Definition: **Primary expressions** are the basic primitives of the language. They include literals, variable references, context item expressions, and function calls. A primary expression may also be created by enclosing any expression in parentheses, which is sometimes helpful in controlling the precedence of operators.]

[41]	`PrimaryExpr`	::=	`Literal	VarRef	ParenthesizedExpr	ContextItemExpr	FunctionCall`

3.1.1 Literals

[Definition: A **literal** is a direct syntactic representation of an atomic value.] XPath supports two kinds of literals: numeric literals and string literals.

[42]	`Literal`	::=	`NumericLiteral	StringLiteral`
[43]	`NumericLiteral`	::=	`IntegerLiteral	DecimalLiteral	DoubleLiteral`
[71]	`IntegerLiteral`	::=	`Digits`
[72]	`DecimalLiteral`	::=	`("." Digits)	(Digits "." [0-9]*)`

[73]	DoubleLiteral	::=	`(("." Digits) | (Digits ("." [0-9]*)?)) [eE] [+-]? Digits`
[74]	StringLiteral	::=	`('"' (EscapeQuot | [^"])* '"') | ("'" (EscapeApos | [^'])* "'")`
[75]	EscapeQuot	::=	`'""'`
[76]	EscapeApos	::=	`"''"`
[81]	Digits	::=	`[0-9]+`

The value of a **numeric literal** containing no "`.`" and no `e` or `E` character is an atomic value of type `xs:integer`. The value of a numeric literal containing "`.`" but no `e` or `E` character is an atomic value of type `xs:decimal`. The value of a numeric literal containing an `e` or `E` character is an atomic value of type `xs:double`. The value of the numeric literal is determined by casting it to the appropriate type according to the rules for casting from `xs:untypedAtomic` to a numeric type as specified in Section 17.1.1 Casting from xs:string and xs:untypedAtomic[FO].

The value of a **string literal** is an atomic value whose type is `xs:string` and whose value is the string denoted by the characters between the delimiting apostrophes or quotation marks. If the literal is delimited by apostrophes, two adjacent apostrophes within the literal are interpreted as a single apostrophe. Similarly, if the literal is delimited by quotation marks, two adjacent quotation marks within the literal are interpreted as one quotation mark.

Here are some examples of literal expressions:

- `"12.5"` denotes the string containing the characters '1', '2', '.', and '5'.
- `12` denotes the `xs:integer` value twelve.
- `12.5` denotes the `xs:decimal` value twelve and one half.
- `125E2` denotes the `xs:double` value twelve thousand, five hundred.
- `"He said, ""I don't like it."""` denotes a string containing two quotation marks and one apostrophe.

  Note:

  When XPath expressions are embedded in contexts where quotation marks have special significance, such as inside XML attributes, additional escaping may be needed.

The `xs:boolean` values `true` and `false` can be represented by calls to the built-in functions `fn:true()` and `fn:false()`, respectively.

Values of other atomic types can be constructed by calling the constructor function for the given type. The constructor functions for XML Schema built-in types are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)]. In general, the name of a constructor function for a given type is the same as the name of the type (including its namespace). For example:

- `xs:integer("12")` returns the integer value twelve.
- `xs:date("2001-08-25")` returns an item whose type is `xs:date` and whose value represents the date 25th August 2001.
- `xs:dayTimeDuration("PT5H")` returns an item whose type is `xs:dayTimeDuration` and whose value represents a duration of five hours.

Constructor functions can also be used to create special values that have no literal representation, as in the following examples:

- `xs:float("NaN")` returns the special floating-point value, "Not a Number."
- `xs:double("INF")` returns the special double-precision value, "positive infinity."

It is also possible to construct values of various types by using a `cast` expression. For example:

- `9 cast as hatsize` returns the atomic value `9` whose type is `hatsize`.

3.1.2 Variable References

[44]	VarRef	::=	`"$" VarName`
[45]	VarName	::=	`QName`

[Definition: A **variable reference** is a QName preceded by a $-sign.] Two variable references are equivalent if their local names are the same and their namespace prefixes are bound to the same namespace URI in the statically known namespaces. An unprefixed variable reference is in no namespace.

Every variable reference must match a name in the in-scope variables, which include variables from the following sources:

1. The in-scope variables may be augmented by implementation-defined variables.
2. A variable may be bound by an XPath expression. The kinds of expressions that can bind variables are `for` expressions (**3.7 For Expressions**) and quantified expressions (**3.9 Quantified Expressions**).

Every variable binding has a static scope. The scope defines where references to the variable can validly occur. It is a static error [err:XPST0008] to reference a variable that is not in scope. If a variable is bound in the static context for an expression, that variable is in scope for the entire expression.

If a variable reference matches two or more variable bindings that are in scope, then the reference is taken as referring to the inner binding, that is, the one whose scope is smaller. At evaluation time, the value of a variable reference is the value of the expression to which the relevant variable is bound. The scope of a variable binding is defined separately for each kind of expression that can bind variables.

3.1.3 Parenthesized Expressions

[46]	`ParenthesizedExpr`	::=	`"(" Expr? ")"`

Parentheses may be used to enforce a particular evaluation order in expressions that contain multiple operators. For example, the expression `(2 + 4) * 5` evaluates to thirty, since the parenthesized expression `(2 + 4)` is evaluated first and its result is multiplied by five. Without parentheses, the expression `2 + 4 * 5` evaluates to twenty-two, because the multiplication operator has higher precedence than the addition operator.

Empty parentheses are used to denote an empty sequence, as described in **3.3.1 Constructing Sequences**.

3.1.4 Context Item Expression

[47]	`ContextItemExpr`	::=	`"."`

A **context item expression** evaluates to the context item, which may be either a node (as in the expression `fn:doc("bib.xml")/books/book[fn:count(./author)>1]`) or an atomic value (as in the expression `(1 to 100)[. mod 5 eq 0]`).

If the context item is undefined, a context item expression raises a dynamic error [err:XPDY0002].

3.1.5 Function Calls

[Definition: The **built-in functions** supported by XPath are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].] Additional functions may be provided in the static context. XPath per se does not provide a way to declare functions, but a host language may provide such a mechanism.

[48]	`FunctionCall`	::=	`QName "(" (ExprSingle ("," ExprSingle)*)? ")"`

A **function call** consists of a QName followed by a parenthesized list of zero or more expressions, called **arguments**. If the QName in the function call has no namespace prefix, it is considered to be in the default function namespace.

If the expanded QName and number of arguments in a function call do not match the name and arity of a function signature in the static context, a static error is raised [err:XPST0017].

A function call is evaluated as follows:

1. Argument expressions are evaluated, producing argument values. The order of argument evaluation is implementation-dependent and a function need not evaluate an argument if the function can evaluate its body without evaluating that argument.
2. Each argument value is converted by applying the function conversion rules listed below.
3. The function is evaluated using the converted argument values. The result is either an instance of the function's declared return type or a dynamic error. The dynamic type of a function result may be a type that is derived from the declared return type. Errors raised by functions are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

The **function conversion rules** are used to convert an argument value to its expected type; that is, to the declared type of the function parameter. The expected type is expressed as a sequence type. The function conversion rules are applied to a given value as follows:

- If XPath 1.0 compatibility mode is `true` and an argument is not of the expected type, then the following conversions are applied sequentially to the argument value V:
  1. If the expected type calls for a single item or optional single item (examples: `xs:string`, `xs:string?`, `xs:untypedAtomic`, `xs:untypedAtomic?`, `node()`, `node()?`, `item()`, `item()?`), then the value V is effectively replaced by V[1].
  2. If the expected type is `xs:string` or `xs:string?`, then the value `V` is effectively replaced by `fn:string(V)`.
  3. If the expected type is `xs:double` or `xs:double?`, then the value `V` is effectively replaced by `fn:number(V)`.
- If the expected type is a sequence of an atomic type (possibly with an occurrence indicator `*`, `+`, or `?`), the following conversions are applied:
  1. Atomization is applied to the given value, resulting in a sequence of atomic values.
  2. Each item in the atomic sequence that is of type `xs:untypedAtomic` is cast to the expected atomic type. For built-in functions where the expected type is specified as numeric, arguments of type `xs:untypedAtomic` are cast to `xs:double`.
  3. For each numeric item in the atomic sequence that can be promoted to the expected atomic type using numeric promotion as described in **B.1 Type Promotion**, the promotion is done.
  4. For each item of type `xs:anyURI` in the atomic sequence that can be promoted to the expected atomic type using URI promotion as described in **B.1 Type Promotion**, the promotion is done.
- If, after the above conversions, the resulting value does not match the expected type according to the rules for SequenceType Matching, a type error is raised [err:XPTY0004]. Note that the rules for SequenceType Matching permit a value of a derived type to be substituted for a value of its base type.

Since the arguments of a function call are separated by commas, any argument expression that contains a top-level comma operator must be enclosed in parentheses. Here are some illustrative examples of function calls:

- `my:three-argument-function(1, 2, 3)` denotes a function call with three arguments.
- `my:two-argument-function((1, 2), 3)` denotes a function call with two arguments, the first of which is a sequence of two values.
- `my:two-argument-function(1, ())` denotes a function call with two arguments, the second of which is an empty sequence.
- `my:one-argument-function((1, 2, 3))` denotes a function call with one argument that is a sequence of three values.
- `my:one-argument-function(( ))` denotes a function call with one argument that is an empty sequence.
- `my:zero-argument-function( )` denotes a function call with zero arguments.

3.2 Path Expressions

[25]	`PathExpr`	::=	`("/" RelativePathExpr?)`
			`| ("//" RelativePathExpr)`
			`| RelativePathExpr`

[26]	`RelativePathExpr`	::=	`StepExpr (("/" | "//") StepExpr)*`

[Definition: A **path expression** can be used to locate nodes within trees. A path expression consists of a series of one or more steps, separated by "/" or "//", and optionally beginning with "/" or "//".] An initial "/" or "//" is an abbreviation for one or more initial steps that are implicitly added to the beginning of the path expression, as described below.

A path expression consisting of a single step is evaluated as described in **3.2.1 Steps**.

A "/" at the beginning of a path expression is an abbreviation for the initial step `(fn:root(self::node()) treat as document-node())/` (however, if the "/" is the entire path expression, the trailing "/" is omitted from the expansion.) The effect of this initial step is to begin the path at the root node of the tree that contains the context node. If the context item is not a node, a type error is raised [err:XPTY0020]. At evaluation time, if the root node above the context node is not a document node, a dynamic error is raised [err:XPDY0050].

A "//" at the beginning of a path expression is an abbreviation for the initial steps `(fn:root(self::node()) treat as document-node())/descendant-or-self::node()/` (however, "//" by itself is not a valid path expression [err:XPST0003].) The effect of these initial steps is to establish an initial node sequence that contains the root of the tree in which the context node is found, plus all nodes descended from this root. This node sequence is used as the input to subsequent steps in the path expression. If the context item is not a node, a type error is raised [err:XPTY0020]. At evaluation time, if the root node above the context node is not a document node, a dynamic error is raised [err:XPDY0050].

Note:

The descendants of a node do not include attribute nodes or namespace nodes.

Each non-initial occurrence of "//" in a path expression is expanded as described in **3.2.4 Abbreviated Syntax**, leaving a sequence of steps separated by "/". This sequence of steps is then evaluated from left to right. Each operation `E1/E2` is evaluated as follows: Expression `E1` is evaluated, and if the result is not a (possibly empty) sequence of nodes, a type error is raised [err:XPTY0019]. Each node resulting from the evaluation of `E1` then serves in turn to provide an **inner focus** for an evaluation of `E2`, as described in **2.1.2 Dynamic Context**. The sequences resulting from all the evaluations of `E2` are combined as follows:

1. If every evaluation of `E2` returns a (possibly empty) sequence of nodes, these sequences are combined, and duplicate nodes are eliminated based on node identity. The resulting node sequence is returned in document order.
2. If every evaluation of `E2` returns a (possibly empty) sequence of atomic values, these sequences are concatenated, in order, and returned.
3. If the multiple evaluations of `E2` return at least one node and at least one atomic value, a type error is raised [err:XPTY0018].

Note:

Since each step in a path provides context nodes for the following step, in effect, only the last step in a path is allowed to return a sequence of atomic values.

As an example of a path expression, `child::div1/child::para` selects the `para` element children of the `div1` element children of the context node, or, in other words, the `para` element grandchildren of the context node that have `div1` parents.

Note:

The "/" character can be used either as a complete path expression or as the beginning of a longer path expression such as "/*". Also, "*" is both the multiply operator and a wildcard in path expressions. This can cause parsing difficulties when "/" appears on the left hand side of "*". This is resolved using the leading-lone-slash constraint. For example, "/*" and "/ *" are valid path expressions containing wildcards, but "/*5" and "/ * 5" raise syntax errors. Parentheses must be used when "/" is used on the left hand side of an operator, as in "(/) * 5". Similarly, "4 + / * 5" raises a syntax error, but "4 + (/) * 5" is a valid expression. The expression "4 + /" is also valid, because / does not occur on the left hand side of the operator.

3.2.1 Steps

[27]	`StepExpr`	::=	`FilterExpr | AxisStep`
[28]	`AxisStep`	::=	`(ReverseStep | ForwardStep) PredicateList`
[29]	`ForwardStep`	::=	`(ForwardAxis NodeTest) | AbbrevForwardStep`
[32]	`ReverseStep`	::=	`(ReverseAxis NodeTest) | AbbrevReverseStep`
[39]	`PredicateList`	::=	`Predicate*`

[Definition: A **step** is a part of a path expression that generates a sequence of items and then filters the sequence by zero or more predicates. The value of the step consists of those items that satisfy the predicates, working from left to right. A step may be either an axis step or a filter expression.] Filter expressions are described in **3.3.2 Filter Expressions**.

[Definition: An **axis step** returns a sequence of nodes that are reachable from the context node via a specified axis. Such a step has two parts: an **axis**, which defines the "direction of movement" for the step, and a node test, which selects nodes based on their kind, name, and/or type annotation.] If the context item is a node, an axis step returns a sequence of zero or more nodes; otherwise, a type error is raised [err:XPTY0020]. The resulting node sequence is returned in document order. An axis step may be either a **forward step** or a **reverse step**, followed by zero or more predicates.

In the **abbreviated syntax** for a step, the axis can be omitted and other shorthand notations can be used as described in **3.2.4 Abbreviated Syntax**.

The unabbreviated syntax for an axis step consists of the axis name and node test separated by a double colon. The result of the step consists of the nodes reachable from the context node via the specified axis that have the node kind, name, and/or type annotation specified by the node test. For example, the step `child::para` selects the `para` element children of the context node: `child` is the name of the axis, and `para` is the name of the element nodes to be selected on this axis. The available axes are described in **3.2.1.1 Axes**. The available node tests are described in **3.2.1.2 Node Tests**. Examples of steps are provided in **3.2.3 Unabbreviated Syntax** and **3.2.4 Abbreviated Syntax**.

3.2.1.1 Axes

```
[30] ForwardAxis ::= ("child" "::")
                     | ("descendant" "::")
                     | ("attribute" "::")
                     | ("self" "::")
                     | ("descendant-or-self" "::")
                     | ("following-sibling" "::")
                     | ("following" "::")
                     | ("namespace" "::")
[33] ReverseAxis ::= ("parent" "::")
                     | ("ancestor" "::")
                     | ("preceding-sibling" "::")
                     | ("preceding" "::")
                     | ("ancestor-or-self" "::")
```

XPath defines a full set of **axes** for traversing documents, but a **host language** may define a subset of these axes. The following axes are defined:

- The `child` axis contains the children of the context node, which are the nodes returned by the `dm:children` accessor in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)].

  Note:

  Only document nodes and element nodes have children. If the context node is any other kind of node, or if the context node is an empty document or element node, then the child axis is an empty sequence. The children of a document node or element node may be element, processing instruction, comment, or text nodes. Attribute, namespace, and document nodes can never appear as children.
- the `descendant` axis is defined as the transitive closure of the child axis; it contains the descendants of the context node (the children, the children of the children, and so on)
- the `parent` axis contains the sequence returned by the `dm:parent` accessor in [XQuery 1.0 and XPath

2.0 Data Model (Second Edition)], which returns the parent of the context node, or an empty sequence if the context node has no parent

Note:

An attribute node may have an element node as its parent, even though the attribute node is not a child of the element node.

- the `ancestor` axis is defined as the transitive closure of the parent axis; it contains the ancestors of the context node (the parent, the parent of the parent, and so on)

  Note:

  The ancestor axis includes the root node of the tree in which the context node is found, unless the context node is the root node.

- the `following-sibling` axis contains the context node's following siblings, those children of the context node's parent that occur after the context node in document order; if the context node is an attribute or namespace node, the `following-sibling` axis is empty
- the `preceding-sibling` axis contains the context node's preceding siblings, those children of the context node's parent that occur before the context node in document order; if the context node is an attribute or namespace node, the `preceding-sibling` axis is empty
- the `following` axis contains all nodes that are descendants of the root of the tree in which the context node is found, are not descendants of the context node, and occur after the context node in document order
- the `preceding` axis contains all nodes that are descendants of the root of the tree in which the context node is found, are not ancestors of the context node, and occur before the context node in document order
- the `attribute` axis contains the attributes of the context node, which are the nodes returned by the `dm:attributes` accessor in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)]; 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-or-self axis will always include the root node
- the `namespace` axis contains the namespace nodes of the context node, which are the nodes returned by the `dm:namespace-nodes` accessor in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)]; this axis is empty unless the context node is an element node. The `namespace` axis is deprecated in XPath 2.0. If XPath 1.0 compatibility mode is `true`, the `namespace` axis must be supported. If XPath 1.0 compatibility mode is `false`, then support for the `namespace` axis is implementation-defined. An implementation that does not support the `namespace` axis when XPath 1.0 compatibility mode is `false` must raise a static error [err:XPST0010] if it is used. Applications needing information about the in-scope namespaces of an element should use the functions `fn:in-scope-prefixes` and `fn:namespace-uri-for-prefix` defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

Axes can be categorized as **forward axes** and **reverse axes**. 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.

The `parent`, `ancestor`, `ancestor-or-self`, `preceding`, and `preceding-sibling` axes are reverse axes; all other axes are forward axes. 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.

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

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

3.2.1.2 Node Tests

[Definition: A **node test** is a condition that must be true for each node selected by a step.] The condition may be based on the kind of the node (element, attribute, text, document, comment, or processing instruction), the name of the node, or (in the case of element, attribute, and document nodes), the type annotation of the node.

```
[35] NodeTest  ::= KindTest | NameTest
[36] NameTest  ::= QName | Wildcard
[37] Wildcard  ::= "*"
                 | (NCName ":" "*")
                 | ("*" ":" NCName)
```

[Definition: A node test that consists only of a QName or a Wildcard is called a **name test**.] A name test is true if and only if the **kind** of the node is the principal node kind for the step axis and the expanded QName of the node is equal (as defined by the `eq` operator) to the expanded QName specified by the name test. For example, `child::para` selects the `para` element children of the context node; if the context node has no `para` children, it selects an empty set of nodes. `attribute::abc:href` selects the attribute of the context node with the QName `abc:href`; if the context node has no such attribute, it selects an empty set of nodes.

A QName in a name test is resolved into an expanded QName using the statically known namespaces in the expression context. It is a static error [err:XPST0081] if the QName has a prefix that does not correspond to any statically known namespace. An unprefixed QName, when used as a name test on an axis whose principal node kind is element, has the namespace URI of the default element/type namespace in the expression context; otherwise, it has no namespace URI.

A name test is not satisfied by an element node whose name does not match the expanded QName of the name test, even if it is in a substitution group whose head is the named element.

A node test `*` is true for any node of the principal node kind of the step axis. 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 statically known namespaces in the static context. If the prefix is not found in the statically known namespaces, a static error is raised [err:XPST0081]. The node test is true for any node of the principal node kind of the step axis whose expanded QName has the namespace URI to which the prefix is bound, regardless of the local part of the name.

A node test can also have the form `*:NCName`. In this case, the node test is true for any node of the principal node kind of the step axis whose local name matches the given NCName, regardless of its namespace or lack of a namespace.

[Definition: An alternative form of a node test called a **kind test** can select nodes based on their kind, name, and type annotation.] The syntax and semantics of a kind test are described in **2.5.3 SequenceType Syntax** and **2.5.4 SequenceType Matching**. When a kind test is used in a node test, only those nodes on the designated axis that match the kind test are selected. Shown below are several examples of kind tests that might be used in path expressions:

- `node()` matches any node.
- `text()` matches any text node.
- `comment()` matches any comment node.
- `element()` matches any element node.
- `schema-element(person)` matches any element node whose name is `person` (or is in the substitution group headed by `person`), and whose type annotation is the same as (or is derived from) the declared type of the `person` element in the in-scope element declarations.
- `element(person)` matches any element node whose name is `person`, regardless of its type annotation.
- `element(person, surgeon)` matches any non-nilled element node whose name is `person`, and whose type annotation is `surgeon` or is derived from `surgeon`.
- `element(*, surgeon)` matches any non-nilled element node whose type annotation is `surgeon` (or is derived from `surgeon`), regardless of its name.
- `attribute()` matches any attribute node.

- `attribute(price)` matches any attribute whose name is `price`, regardless of its type annotation.
- `attribute(*, xs:decimal)` matches any attribute whose type annotation is `xs:decimal` (or is derived from `xs:decimal`), regardless of its name.
- `document-node()` matches any document node.
- `document-node(element(book))` matches any document node whose content consists of a single element node that satisfies the kind test `element(book)`, interleaved with zero or more comments and processing instructions.

3.2.2 Predicates

[40] Predicate ::= "[" Expr "]"

[Definition: A **predicate** consists of an expression, called a **predicate expression**, enclosed in square brackets. A predicate serves to filter a sequence, retaining some items and discarding others.] In the case of multiple adjacent predicates, the predicates are applied from left to right, and the result of applying each predicate serves as the input sequence for the following predicate.

For each item in the input sequence, the predicate expression is evaluated using an **inner focus**, defined as follows: The context item is the item currently being tested against the predicate. The context size is the number of items in the input sequence. The context position is the position of the context item within the input sequence. For the purpose of evaluating the context position within a predicate, the input sequence is considered to be sorted as follows: into document order if the predicate is in a forward-axis step, into reverse document order if the predicate is in a reverse-axis step, or in its original order if the predicate is not in a step.

For each item in the input sequence, the result of the predicate expression is coerced to an `xs:boolean` value, called the **predicate truth value**, as described below. Those items for which the predicate truth value is `true` are retained, and those for which the predicate truth value is `false` are discarded.

The predicate truth value is derived by applying the following rules, in order:

1. If the value of the predicate expression is a singleton atomic value of a numeric type or derived from a numeric type, the predicate truth value is `true` if the value of the predicate expression is equal (by the `eq` operator) to the **context position**, and is `false` otherwise. [Definition: A predicate whose predicate expression returns a numeric type is called a **numeric predicate**.]
2. Otherwise, the predicate truth value is the effective boolean value of the predicate expression.

Here are some examples of axis steps that contain predicates:

- This example selects the second `chapter` element that is a child of the context node:

```
child::chapter[2]
```

- This example selects all the descendants of the context node that are elements named `"toy"` and whose `color` attribute has the value `"red"`:

```
descendant::toy[attribute::color = "red"]
```

- This example selects all the `employee` children of the context node that have both a `secretary` child element and an `assistant` child element:

```
child::employee[secretary][assistant]
```

Note:

When using predicates with a sequence of nodes selected using a **reverse axis**, it is important to remember that the the context positions for such a sequence are assigned in reverse document order. For example, `preceding::foo[1]` returns the first qualifying `foo` element in reverse document order, because the predicate is part of an axis step using a reverse axis. By contrast, `(preceding::foo)[1]` returns the first qualifying `foo` element in document order, because the parentheses cause `(preceding::foo)` to be parsed as a primary expression in which context positions are assigned in document order. Similarly, `ancestor::*[1]` returns the nearest ancestor element, because the `ancestor` axis is a reverse axis, whereas `(ancestor::*)[1]` returns the root element (first ancestor in document order).

The fact that a reverse-axis step assigns context positions in reverse document order for the purpose of evaluating predicates does not alter the fact that the final result of the step is always in document order.

3.2.3 Unabbreviated Syntax

This section provides a number of examples of path expressions in which the axis is explicitly specified in each step. The syntax used in these examples is called the **unabbreviated syntax**. In many common cases, it is possible to write path expressions more concisely using an **abbreviated syntax**, as explained in **3.2.4 Abbreviated 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. Note that no attribute nodes are returned, because attributes are not children.
- `attribute::name` selects the `name` attribute of the context node
- `attribute::*` selects all the attributes of the context node
- `parent::node()` selects the parent of the context node. If the context node is an attribute node, this expression returns the element node (if any) to which the attribute node is attached.
- `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 returns an empty sequence
- `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 root of the tree that contains the context node, but raises a dynamic error if this root is not a document node
- `/descendant::para` selects all the `para` elements in the same document as the context node
- `/descendant::list/child::member` selects all the `member` elements that have a `list` parent and that are in the same document as the context node
- `child::para[fn:position() = 1]` selects the first `para` child of the context node
- `child::para[fn:position() = fn:last()]` selects the last `para` child of the context node
- `child::para[fn:position() = fn:last()-1]` selects the last but one `para` child of the context node
- `child::para[fn:position() > 1]` selects all the `para` children of the context node other than the first `para` child of the context node
- `following-sibling::chapter[fn:position() = 1]`selects the next `chapter` sibling of the context node
- `preceding-sibling::chapter[fn:position() = 1]`selects the previous `chapter` sibling of the context node
- `/descendant::figure[fn:position() = 42]` selects the forty-second `figure` element in the document containing the context node
- `/child::book/child::chapter[fn:position() = 5]/child::section[fn:position() = 2]` selects the second `section` of the fifth `chapter` of the `book` whose parent is the document node that contains the context node
- `child::para[attribute::type eq "warning"]`selects all `para` children of the context node that have a `type` attribute with value `warning`
- `child::para[attribute::type eq 'warning'][fn:position() = 5]`selects the fifth `para` child of the context node that has a `type` attribute with value `warning`
- `child::para[fn:position() = 5][attribute::type eq "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 whose [typed value](#) is equal to the string `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][fn:position() = fn:last()]` selects the last `chapter` or `appendix` child of the context node

3.2.4 Abbreviated Syntax

[31]	`AbbrevForwardStep`	::=	`"@"? NodeTest`
[34]	`AbbrevReverseStep`	::=	`".."`

The abbreviated syntax permits the following abbreviations:

1. The attribute axis `attribute::` can be abbreviated by `@`. For example, a path expression `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`.
2. If the axis name is omitted from an [axis step](#), the default axis is `child` unless the axis step contains an [AttributeTest](#) or [SchemaAttributeTest](#); in that case, the default axis is `attribute`. For example, the path expression `section/para` is an abbreviation for `child::section/child::para`, and the path expression `section/@id` is an abbreviation for `child::section/attribute::id`. Similarly, `section/attribute(id)` is an abbreviation for `child::section/attribute::attribute(id)`. Note that the latter expression contains both an axis specification and a [node test](#).
3. Each non-initial occurrence of `//` is effectively replaced by `/descendant-or-self::node()/` during processing of a path expression. For example, `div1//para` is short for `child::div1/descendant-or-self::node()/child::para` and so will select all `para` descendants of `div1` children.

   Note:

   The path expression `//para[1]` does *not* mean the same as the path expression `/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 respective parents.
4. A step consisting 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.

   Note:

   The expression `.`, known as a **context item expression**, is a [primary expression](#), and is described in **[3.1.4 Context Item Expression](#)**.

Here are some examples of path expressions that use the 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[fn:last()]` selects the last `para` child of the context node
- `*/para` selects all `para` grandchildren of the context node
- `/book/chapter[5]/section[2]` selects the second `section` of the fifth `chapter` of the `book` whose parent is the document node that contains the context node
- `chapter//para` selects the `para` element descendants of the `chapter` element children of the context node
- `//para` selects all the `para` descendants of the root document node and thus selects all `para` elements in the same document as the context node
- `//@version` selects all the `version` attribute nodes that are in the same document as the context node
- `//list/member` selects all the `member` elements in the same document as the context node that have a

`list` parent

- `.//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 whose typed value is equal to the string `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
- `book/(chapter|appendix)/section` selects every `section` element that has a parent that is either a `chapter` or an `appendix` element, that in turn is a child of a `book` element that is a child of the context node.
- If `E` is any expression that returns a sequence of nodes, then the expression `E/.` returns the same nodes in document order, with duplicates eliminated based on node identity.

3.3 Sequence Expressions

XPath supports operators to construct, filter, and combine sequences of items. Sequences are never nested—for example, combining the values `1`, `(2, 3)`, and `( )` into a single sequence results in the sequence `(1, 2, 3)`.

3.3.1 Constructing Sequences

[2]	`Expr`	::=	`ExprSingle ("," ExprSingle)*`
[11]	`RangeExpr`	::=	`AdditiveExpr ( "to" AdditiveExpr )?`

[Definition: One way to construct a sequence is by using the **comma operator**, which evaluates each of its operands and concatenates the resulting sequences, in order, into a single result sequence.] Empty parentheses can be used to denote an empty sequence.

A sequence may contain duplicate atomic values or nodes, but a sequence is never an item in another sequence. When a new sequence is created by concatenating two or more input sequences, the new sequence contains all the items of the input sequences and its length is the sum of the lengths of the input sequences.

> **Note:**
>
> In places where the grammar calls for ExprSingle, such as the arguments of a function call, any expression that contains a top-level comma operator must be enclosed in parentheses.

Here are some examples of expressions that construct sequences:

- The result of this expression is a sequence of five integers:

```
(10, 1, 2, 3, 4)
```

- This expression combines four sequences of length one, two, zero, and two, respectively, into a single sequence of length five. The result of this expression is the sequence `10, 1, 2, 3, 4`.

```
(10, (1, 2), (), (3, 4))
```

- The result of this expression is a sequence containing all `salary` children of the context node followed by all `bonus` children.

```
(salary, bonus)
```

- Assuming that `$price` is bound to the value `10.50`, the result of this expression is the sequence `10.50, 10.50`.

```
($price, $price)
```

A **range expression** can be used to construct a sequence of consecutive integers. Each of the operands of the `to` operator is converted as though it was an argument of a function with the expected parameter type `xs:integer?`. If either operand is an empty sequence, or if the integer derived from the first operand is greater than the integer derived from the second operand, the result of the range expression is an empty sequence. If the two operands convert to the same integer, the result of the range expression is that integer. Otherwise, the result is a sequence containing the two integer operands and every integer between the two operands, in increasing order.

- This example uses a range expression as one operand in constructing a sequence. It evaluates to the sequence `10, 1, 2, 3, 4`.

```
(10, 1 to 4)
```

- This example constructs a sequence of length one containing the single integer `10`.

```
10 to 10
```

- The result of this example is a sequence of length zero.

```
15 to 10
```

- This example uses the `fn:reverse` function to construct a sequence of six integers in decreasing order. It evaluates to the sequence `15, 14, 13, 12, 11, 10`.

```
fn:reverse(10 to 15)
```

3.3.2 Filter Expressions

[38]	`FilterExpr`	::=	`PrimaryExpr PredicateList`
[39]	`PredicateList`	::=	`Predicate*`

[Definition: A **filter expression** consists simply of a **primary expression** followed by zero or more predicates. The result of the filter expression consists of the items returned by the primary expression, filtered by applying each predicate in turn, working from left to right.] If no predicates are specified, the result is simply the result of the primary expression. The ordering of the items returned by a filter expression is the same as their order in the result of the primary expression. Context positions are assigned to items based on their ordinal position in the result sequence. The first context position is 1.

Here are some examples of filter expressions:

- Given a sequence of products in a variable, return only those products whose price is greater than 100.

```
$products[price gt 100]
```

- List all the integers from 1 to 100 that are divisible by 5. (See **3.3.1 Constructing Sequences** for an explanation of the `to` operator.)

```
(1 to 100)[. mod 5 eq 0]
```

- The result of the following expression is the integer 25:

```
(21 to 29)[5]
```

- The following example returns the fifth through ninth items in the sequence bound to variable `$orders`.

```
$orders[fn:position() = (5 to 9)]
```

- The following example illustrates the use of a filter expression as a step in a path expression. It returns the last chapter or appendix within the book bound to variable `$book`:

```
$book/(chapter | appendix)[fn:last()]
```

- The following example also illustrates the use of a filter expression as a step in a path expression. It returns the element node within the specified document whose ID value is `tiger`:

```
fn:doc("zoo.xml")/fn:id('tiger')
```

3.3.3 Combining Node Sequences

[14]	UnionExpr	::=	IntersectExceptExpr (("union" | "|") IntersectExceptExpr)*
[15]	IntersectExceptExpr	::=	InstanceofExpr (("intersect" | "except") InstanceofExpr)*

XPath provides the following operators for combining sequences of nodes:

- The `union` and `|` operators are equivalent. They take two node sequences as operands and return a sequence containing all the nodes that occur in either of the operands.
- The `intersect` operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in both operands.
- The `except` operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in the first operand but not in the second operand.

All these operators eliminate duplicate nodes from their result sequences based on node identity. The resulting sequence is returned in document order.

If an operand of `union`, `intersect`, or `except` contains an item that is not a node, a type error is raised [err:XPTY0004].

Here are some examples of expressions that combine sequences. Assume the existence of three element nodes that we will refer to by symbolic names A, B, and C. Assume that the variables `$seq1`, `$seq2` and `$seq3` are bound to the following sequences of these nodes:

- `$seq1` is bound to (A, B)
- `$seq2` is bound to (A, B)
- `$seq3` is bound to (B, C)

Then:

- `$seq1 union $seq2` evaluates to the sequence (A, B).
- `$seq2 union $seq3` evaluates to the sequence (A, B, C).
- `$seq1 intersect $seq2` evaluates to the sequence (A, B).
- `$seq2 intersect $seq3` evaluates to the sequence containing B only.
- `$seq1 except $seq2` evaluates to the empty sequence.
- `$seq2 except $seq3` evaluates to the sequence containing A only.

In addition to the sequence operators described here, [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)] includes functions for indexed access to items or sub-sequences of a sequence, for indexed insertion or removal of items in a sequence, and for removing duplicate items from a sequence.

3.4 Arithmetic Expressions

XPath provides arithmetic operators for addition, subtraction, multiplication, division, and modulus, in their usual binary and unary forms.

[12]	AdditiveExpr	::=	MultiplicativeExpr (("+" | "-") MultiplicativeExpr)*
[13]	MultiplicativeExpr	::=	UnionExpr (("*" | "div" | "idiv" | "mod") UnionExpr)*
[20]	UnaryExpr	::=	("-" | "+")* ValueExpr
[21]	ValueExpr	::=	PathExpr

A subtraction operator must be preceded by whitespace if it could otherwise be interpreted as part of the previous token. For example, `a-b` will be interpreted as a name, but `a - b` and `a -b` will be interpreted as arithmetic expressions. (See **A.2.4 Whitespace Rules** for further details on whitespace handling.)

The first step in evaluating an arithmetic expression is to evaluate its operands. The order in which the operands are evaluated is implementation-dependent.

If XPath 1.0 compatibility mode is `true`, each operand is evaluated by applying the following steps, in order:

1. Atomization is applied to the operand. The result of this operation is called the **atomized operand**.
2. If the atomized operand is an empty sequence, the result of the arithmetic expression is the `xs:double` value `NaN`, and the implementation need not evaluate the other operand or apply the operator. However, an implementation may choose to evaluate the other operand in order to determine whether it raises an error.
3. If the atomized operand is a sequence of length greater than one, any items after the first item in the sequence are discarded.
4. If the atomized operand is now an instance of type `xs:boolean`, `xs:string`, `xs:decimal` (including `xs:integer`), `xs:float`, or `xs:untypedAtomic`, then it is converted to the type `xs:double` by applying the `fn:number` function. (Note that `fn:number` returns the value `NaN` if its operand cannot be converted to a number.)

If XPath 1.0 compatibility mode is `false`, each operand is evaluated by applying the following steps, in order:

1. Atomization is applied to the operand. The result of this operation is called the **atomized operand**.
2. If the atomized operand is an empty sequence, the result of the arithmetic expression is an empty sequence, and the implementation need not evaluate the other operand or apply the operator. However, an implementation may choose to evaluate the other operand in order to determine whether it raises an error.
3. If the atomized operand is a sequence of length greater than one, a type error is raised [err:XPTY0004].
4. If the atomized operand is of type `xs:untypedAtomic`, it is cast to `xs:double`. If the cast fails, a dynamic error is raised. [err:FORG0001]

After evaluation of the operands, if the types of the operands are a valid combination for the given arithmetic operator, the operator is applied to the operands, resulting in an atomic value or a dynamic error (for example, an error might result from dividing by zero.) The combinations of atomic types that are accepted by the various arithmetic operators, and their respective result types, are listed in **B.2 Operator Mapping** together with the operator functions that define the semantics of the operator for each type combination, including the dynamic errors that can be raised by the operator. The definitions of the operator functions are found in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

If the types of the operands, after evaluation, are not a valid combination for the given operator, according to the rules in **B.2 Operator Mapping**, a type error is raised [err:XPTY0004].

XPath supports two division operators named `div` and `idiv`. Each of these operators accepts two operands of any numeric type. As described in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)], `$arg1 idiv $arg2` is equivalent to `($arg1 div $arg2) cast as xs:integer?` except for error cases.

Here are some examples of arithmetic expressions:

- The first expression below returns the `xs:decimal` value `-1.5`, and the second expression returns the `xs:integer` value `-1`:

```
-3 div 2
-3 idiv 2
```

- Subtraction of two date values results in a value of type `xs:dayTimeDuration`:

```
$emp/hiredate - $emp/birthdate
```

- This example illustrates the difference between a subtraction operator and a hyphen:

```
$unit-price - $unit-discount
```

- Unary operators have higher precedence than binary operators, subject of course to the use of parentheses. Therefore, the following two examples have different meanings:

```
-$bellcost + $whistlecost
-($bellcost + $whistlecost)
```

Note:

Multiple consecutive unary arithmetic operators are permitted by XPath for compatibility with [XPath 1.0].

3.5 Comparison Expressions

Comparison expressions allow two values to be compared. XPath provides three kinds of comparison expressions, called value comparisons, general comparisons, and node comparisons.

[10]	`ComparisonExpr`	::=	`RangeExpr ( (ValueComp`
			`| GeneralComp`
			`| NodeComp) RangeExpr )?`

[23]	`ValueComp`	::=	`"eq" | "ne" | "lt" | "le" | "gt" | "ge"`
[22]	`GeneralComp`	::=	`"=" | "!=" | "<" | "<=" | ">" | ">="`
[24]	`NodeComp`	::=	`"is" | "<<" | ">>"`

Note:

When an XPath expression is written within an XML document, the XML escaping rules for special characters must be followed; thus "`<`" must be written as "`<`".

3.5.1 Value Comparisons

The value comparison operators are `eq`, `ne`, `lt`, `le`, `gt`, and `ge`. Value comparisons are used for comparing single values.

The first step in evaluating a value comparison is to evaluate its operands. The order in which the operands are evaluated is implementation-dependent. Each operand is evaluated by applying the following steps, in order:

1. Atomization is applied to the operand. The result of this operation is called the **atomized operand**.
2. If the atomized operand is an empty sequence, the result of the value comparison is an empty sequence, and the implementation need not evaluate the other operand or apply the operator. However, an implementation may choose to evaluate the other operand in order to determine whether it raises an error.
3. If the atomized operand is a sequence of length greater than one, a type error is raised [err:XPTY0004].
4. If the atomized operand is of type `xs:untypedAtomic`, it is cast to `xs:string`.

   Note:

   The purpose of this rule is to make value comparisons transitive. Users should be aware that the general comparison operators have a different rule for casting of `xs:untypedAtomic` operands. Users should also be aware that transitivity of value comparisons may be compromised by loss of precision during type conversion (for example, two `xs:integer` values that differ slightly may both be considered equal to the same `xs:float` value because `xs:float` has less precision than `xs:integer`).

Next, if possible, the two operands are converted to their least common type by a combination of type promotion and subtype substitution. For example, if the operands are of type `hatsize` (derived from `xs:integer`) and `shoesize` (derived from `xs:float`), their least common type is `xs:float`.

Finally, if the types of the operands are a valid combination for the given operator, the operator is applied to the operands. The combinations of atomic types that are accepted by the various value comparison operators, and their respective result types, are listed in **B.2 Operator Mapping** together with the operator functions that define the semantics of the operator for each type combination. The definitions of the operator functions are found in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

Informally, if both atomized operands consist of exactly one atomic value, then the result of the comparison is `true` if the value of the first operand is (equal, not equal, less than, less than or equal, greater than, greater than or equal) to the value of the second operand; otherwise the result of the comparison is `false`.

If the types of the operands, after evaluation, are not a valid combination for the given operator, according to the rules in **B.2 Operator Mapping**, a type error is raised [err:XPTY0004].

Here are some examples of value comparisons:

- The following comparison atomizes the node(s) that are returned by the expression `$book/author`. The comparison is true only if the result of atomization is the value "Kennedy" as an instance of `xs:string` or `xs:untypedAtomic`. If the result of atomization is an empty sequence, the result of the comparison is an empty sequence. If the result of atomization is a sequence containing more than one value, a type error is raised [err:XPTY0004].

```
$book1/author eq "Kennedy"
```

- The following path expression contains a predicate that selects products whose weight is greater than 100. For any product that does not have a `weight` subelement, the value of the predicate is the empty sequence, and the product is not selected. This example assumes that `weight` is a validated element with a numeric type.

```
//product[weight gt 100]
```

- The following comparison is true if `my:hatsize` and `my:shoesize` are both user-defined types that are derived by restriction from a primitive numeric type:

```
my:hatsize(5) eq my:shoesize(5)
```

- The following comparison is true. The `eq` operator compares two QNames by performing codepoint-comparisons of their namespace URIs and their local names, ignoring their namespace prefixes.

```
fn:QName("http://example.com/ns1", "this:color")
   eq fn:QName("http://example.com/ns1", "that:color")
```

3.5.2 General Comparisons

The general comparison operators are `=`, `!=`, `<`, `<=`, `>`, and `>=`. General comparisons are existentially quantified comparisons that may be applied to operand sequences of any length. The result of a general comparison that does not raise an error is always `true` or `false`.

If XPath 1.0 compatibility mode is `true`, a general comparison is evaluated by applying the following rules, in order:

1. If either operand is a single atomic value that is an instance of `xs:boolean`, then the other operand is converted to `xs:boolean` by taking its effective boolean value.
2. Atomization is applied to each operand. After atomization, each operand is a sequence of atomic values.
3. If the comparison operator is `<`, `<=`, `>`, or `>=`, then each item in both of the operand sequences is converted to the type `xs:double` by applying the `fn:number` function. (Note that `fn:number` returns the value `NaN` if its operand cannot be converted to a number.)
4. The result of the comparison is `true` if and only if there is a pair of atomic values, one in the first operand sequence and the other in the second operand sequence, that have the required **magnitude relationship**. Otherwise the result of the comparison is `false`. The **magnitude relationship** between two atomic values is determined by applying the following rules. If a `cast` operation called for by these rules is not successful, a dynamic error is raised. [err:FORG0001]
   a. If at least one of the two atomic values is an instance of a numeric type, then both atomic values are converted to the type `xs:double` by applying the `fn:number` function.
   b. If both atomic values are instances of `xs:untypedAtomic`, then the values are cast to the type `xs:string`.
   c. If exactly one of the atomic values is an instance of `xs:untypedAtomic`, and the previous rule does not apply (that is, the other value is not numeric), then it is cast to a type depending on the other value's dynamic type T according to the following rules, in which V denotes the value to be cast:

i. If T is `xs:dayTimeDuration` or is derived from `xs:dayTimeDuration`, then V is cast to `xs:dayTimeDuration`.
ii. If T is `xs:yearMonthDuration` or is derived from `xs:yearMonthDuration`, then V is cast to `xs:yearMonthDuration`.
iii. In all other cases, V is cast to the primitive base type of T.

Note:

The special treatment of the duration types is required to avoid errors that may arise when comparing the primitive type `xs:duration` with any duration type.

d. After performing the conversions described above, the atomic values are compared using one of the value comparison operators `eq`, `ne`, `lt`, `le`, `gt`, or `ge`, depending on whether the general comparison operator was `=`, `!=`, `<`, `<=`, `>`, or `>=`. The values have the required **magnitude relationship** if and only if the result of this value comparison is `true`.

If XPath 1.0 compatibility mode is `false`, a general comparison is evaluated by applying the following rules, in order:

1. Atomization is applied to each operand. After atomization, each operand is a sequence of atomic values.
2. The result of the comparison is `true` if and only if there is a pair of atomic values, one in the first operand sequence and the other in the second operand sequence, that have the required **magnitude relationship**. Otherwise the result of the comparison is `false`. The **magnitude relationship** between two atomic values is determined by applying the following rules. If a `cast` operation called for by these rules is not successful, a dynamic error is raised. [err:FORG0001]
   a. If both atomic values are instances of `xs:untypedAtomic`, then the values are cast to the type `xs:string`.
   b. If exactly one of the atomic values is an instance of `xs:untypedAtomic`, it is cast to a type depending on the other value's dynamic type T according to the following rules, in which V denotes the value to be cast:
      i. If T is a numeric type or is derived from a numeric type, then V is cast to `xs:double`.
      ii. If T is `xs:dayTimeDuration` or is derived from `xs:dayTimeDuration`, then V is cast to `xs:dayTimeDuration`.
      iii. If T is `xs:yearMonthDuration` or is derived from `xs:yearMonthDuration`, then V is cast to `xs:yearMonthDuration`.
      iv. In all other cases, V is cast to the primitive base type of T.

      Note:

      The special treatment of the duration types is required to avoid errors that may arise when comparing the primitive type `xs:duration` with any duration type.
   c. After performing the conversions described above, the atomic values are compared using one of the value comparison operators `eq`, `ne`, `lt`, `le`, `gt`, or `ge`, depending on whether the general comparison operator was `=`, `!=`, `<`, `<=`, `>`, or `>=`. The values have the required **magnitude relationship** if and only if the result of this value comparison is `true`.

When evaluating a general comparison in which either operand is a sequence of items, an implementation may return `true` as soon as it finds an item in the first operand and an item in the second operand that have the required **magnitude relationship**. Similarly, a general comparison may raise a dynamic error as soon as it encounters an error in evaluating either operand, or in comparing a pair of items from the two operands. As a result of these rules, the result of a general comparison is not deterministic in the presence of errors.

Here are some examples of general comparisons:

- The following comparison is true if the typed value of any `author` subelement of `$book1` is "Kennedy" as an instance of `xs:string` or `xs:untypedAtomic`:

```
$book1/author = "Kennedy"
```

- The following example contains three general comparisons. The value of the first two comparisons is `true`, and the value of the third comparison is `false`. This example illustrates the fact that general comparisons are not transitive.

```
(1, 2) = (2, 3)
(2, 3) = (3, 4)
(1, 2) = (3, 4)
```

- The following example contains two general comparisons, both of which are `true`. This example illustrates the fact that the `=` and `!=` operators are not inverses of each other.

```
(1, 2) = (2, 3)
(1, 2) != (2, 3)
```

- Suppose that `$a`, `$b`, and `$c` are bound to element nodes with type annotation `xs:untypedAtomic`, with string values "`1`", "`2`", and "`2.0`" respectively. Then `($a, $b) = ($c, 3.0)` returns `false`, because `$b` and `$c` are compared as strings. However, `($a, $b) = ($c, 2.0)` returns `true`, because `$b` and `2.0` are compared as numbers.

3.5.3 Node Comparisons

Node comparisons are used to compare two nodes, by their identity or by their document order. The result of a node comparison is defined by the following rules:

1. The operands of a node comparison are evaluated in implementation-dependent order.
2. If either operand is an empty sequence, the result of the comparison is an empty sequence, and the implementation need not evaluate the other operand or apply the operator. However, an implementation may choose to evaluate the other operand in order to determine whether it raises an error.
3. Each operand must be either a single node or an empty sequence; otherwise a type error is raised [err:XPTY0004].
4. A comparison with the `is` operator is `true` if the two operand nodes have the same identity, and are thus the same node; otherwise it is `false`. See [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)] for a definition of node identity.
5. A comparison with the `<<` operator returns `true` if the left operand node precedes the right operand node in document order; otherwise it returns `false`.
6. A comparison with the `>>` operator returns `true` if the left operand node follows the right operand node in document order; otherwise it returns `false`.

Here are some examples of node comparisons:

- The following comparison is true only if the left and right sides each evaluate to exactly the same single node:

```
/books/book[isbn="1558604820"] is /books/book[call="QA76.9 C3845"]
```

- The following comparison is true only if the node identified by the left side occurs before the node identified by the right side in document order:

```
/transactions/purchase[parcel="28-451"]
   << /transactions/sale[parcel="33-870"]
```

3.6 Logical Expressions

A **logical expression** is either an **and-expression** or an **or-expression**. If a logical expression does not raise an error, its value is always one of the boolean values `true` or `false`.

[8]	`OrExpr`	::=	`AndExpr ( "or" AndExpr )*`
[9]	`AndExpr`	::=	`ComparisonExpr ( "and" ComparisonExpr )*`

The first step in evaluating a logical expression is to find the effective boolean value of each of its operands (see **2.4.3 Effective Boolean Value**).

The value of an and-expression is determined by the effective boolean values (EBV's) of its operands, as shown in the following table:

	EBV_2		

AND:	EBV_2 = true	EBV_2 = false	error in EBV_2
EBV_1 = true	true	false	error
EBV_1 = false	false	false	if XPath 1.0 compatibility mode is true, then false; otherwise either false or error.
error in EBV_1	error	if XPath 1.0 compatibility mode is true, then error; otherwise either false or error.	error

The value of an or-expression is determined by the effective boolean values (EBV's) of its operands, as shown in the following table:

OR:	EBV_2 = true	EBV_2 = false	error in EBV_2
EBV_1 = true	true	true	if XPath 1.0 compatibility mode is true, then true; otherwise either true or error.
EBV_1 = false	true	false	error
error in EBV_1	if XPath 1.0 compatibility mode is true, then error; otherwise either true or error.	error	error

If XPath 1.0 compatibility mode is `true`, the order in which the operands of a logical expression are evaluated is effectively prescribed. Specifically, it is defined that when there is no need to evaluate the second operand in order to determine the result, then no error can occur as a result of evaluating the second operand.

If XPath 1.0 compatibility mode is `false`, the order in which the operands of a logical expression are evaluated is implementation-dependent. In this case, an or-expression can return `true` if the first expression evaluated is true, and it can raise an error if evaluation of the first expression raises an error. Similarly, an and-expression can return `false` if the first expression evaluated is false, and it can raise an error if evaluation of the first expression raises an error. As a result of these rules, a logical expression is not deterministic in the presence of errors, as illustrated in the examples below.

Here are some examples of logical expressions:

- The following expressions return `true`:

```
1 eq 1 and 2 eq 2
```

```
1 eq 1 or 2 eq 3
```

- The following expression may return either `false` or raise a dynamic error (in XPath 1.0 compatibility mode, the result must be `false`):

```
1 eq 2 and 3 idiv 0 = 1
```

- The following expression may return either `true` or raise a dynamic error (in XPath 1.0 compatibility mode, the result must be `true`):

```
1 eq 1 or 3 idiv 0 = 1
```

- The following expression must raise a dynamic error:

```
1 eq 1 and 3 idiv 0 = 1
```

In addition to and- and or-expressions, XPath provides a function named `fn:not` that takes a general sequence as parameter and returns a boolean value. The `fn:not` function is defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)]. The `fn:not` function reduces its parameter to an effective boolean value. It then returns `true` if the effective boolean value of its parameter is `false`, and `false` if the effective boolean value of its parameter is `true`. If an error is encountered in finding the effective boolean value of its operand, `fn:not` raises the same error.

3.7 For Expressions

XPath provides an iteration facility called a **for expression**.

[4]	ForExpr	::=	SimpleForClause "return" ExprSingle
[5]	SimpleForClause	::=	"for" "$" VarName "in" ExprSingle ("," "$" VarName "in" ExprSingle)*

A `for` expression is evaluated as follows:

1. If the `for` expression uses multiple variables, it is first expanded to a set of nested `for` expressions, each of which uses only one variable. For example, the expression `for $x in X, $y in Y return $x + $y` is expanded to `for $x in X return for $y in Y return $x + $y`.
2. In a single-variable `for` expression, the variable is called the **range variable**, the value of the expression that follows the `in` keyword is called the **binding sequence**, and the expression that follows the `return` keyword is called the **return expression**. The result of the `for` expression is obtained by evaluating the `return` expression once for each item in the binding sequence, with the range variable bound to that item. The resulting sequences are concatenated (as if by the comma operator) in the order of the items in the binding sequence from which they were derived.

The following example illustrates the use of a `for` expression in restructuring an input document. The example is based on the following input:

```
<bib>
  <book>
    <title>TCP/IP Illustrated</title>
    <author>Stevens</author>
    <publisher>Addison-Wesley</publisher>
  </book>
  <book>
    <title>Advanced Programming in the Unix Environment</title>
    <author>Stevens</author>
    <publisher>Addison-Wesley</publisher>
  </book>
  <book>
    <title>Data on the Web</title>
    <author>Abiteboul</author>
    <author>Buneman</author>
    <author>Suciu</author>
  </book>
</bib>
```

The following example transforms the input document into a list in which each author's name appears only once, followed by a list of titles of books written by that author. This example assumes that the context item is the `bib` element in the input document.

```
for $a in fn:distinct-values(book/author)
return (book/author[. = $a][1], book[author = $a]/title)
```

The result of the above expression consists of the following sequence of elements. The titles of books written by a given author are listed after the name of the author. The ordering of `author` elements in the result is implementation-dependent due to the semantics of the `fn:distinct-values` function.

```
<author>Stevens</author>
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix environment</title>
```

```
<author>Abiteboul</author>
<title>Data on the Web</title>
<author>Buneman</author>
<title>Data on the Web</title>
<author>Suciu</author>
<title>Data on the Web</title>
```

The following example illustrates a `for` expression containing more than one variable:

```
for $i in (10, 20),
    $j in (1, 2)
return ($i + $j)
```

The result of the above expression, expressed as a sequence of numbers, is as follows: `11, 12, 21, 22`

The scope of a variable bound in a `for` expression comprises all subexpressions of the `for` expression that appear after the variable binding. The scope does not include the expression to which the variable is bound. The following example illustrates how a variable binding may reference another variable bound earlier in the same `for` expression:

```
for $x in $z, $y in f($x)
return g($x, $y)
```

Note:

The focus for evaluation of the `return` clause of a `for` expression is the same as the focus for evaluation of the `for` expression itself. The following example, which attempts to find the total value of a set of order-items, is therefore incorrect:

```
fn:sum(for $i in order-item return @price *
@qty)
```

Instead, the expression must be written to use the variable bound in the `for` clause:

```
fn:sum(for $i in order-item
    return $i/@price * $i/@qty)
```

3.8 Conditional Expressions

XPath supports a conditional expression based on the keywords `if`, `then`, and `else`.

[7] `IfExpr` ::= `"if" "(" Expr ")" "then" ExprSingle "else" ExprSingle`

The expression following the `if` keyword is called the **test expression**, and the expressions following the `then` and `else` keywords are called the **then-expression** and **else-expression**, respectively.

The first step in processing a conditional expression is to find the effective boolean value of the test expression, as defined in **2.4.3 Effective Boolean Value**.

The value of a conditional expression is defined as follows: If the effective boolean value of the test expression is `true`, the value of the then-expression is returned. If the effective boolean value of the test expression is `false`, the value of the else-expression is returned.

Conditional expressions have a special rule for propagating dynamic errors. If the effective value of the test expression is `true`, the conditional expression ignores (does not raise) any dynamic errors encountered in the else-expression. In this case, since the else-expression can have no observable effect, it need not be evaluated. Similarly, if the effective value of the test expression is `false`, the conditional expression ignores any dynamic errors encountered in the then-expression, and the then-expression need not be evaluated.

Here are some examples of conditional expressions:

- In this example, the test expression is a comparison expression:

```
if ($widget1/unit-cost < $widget2/unit-cost)
  then $widget1
  else $widget2
```

- In this example, the test expression tests for the existence of an attribute named `discounted`, independently of its value:

```
if ($part/@discounted)
  then $part/wholesale
  else $part/retail
```

3.9 Quantified Expressions

Quantified expressions support existential and universal quantification. The value of a quantified expression is always `true` or `false`.

[6] QuantifiedExpr ::= ("some" | "every") "$" VarName "in" ExprSingle ("," "$" VarName "in" ExprSingle)* "satisfies" ExprSingle

A **quantified expression** begins with a **quantifier**, which is the keyword `some` or `every`, followed by one or more in-clauses that are used to bind variables, followed by the keyword `satisfies` and a test expression. Each in-clause associates a variable with an expression that returns a sequence of items, called the **binding sequence** for that variable. The in-clauses generate tuples of variable bindings, including a tuple for each combination of items in the binding sequences of the respective variables. Conceptually, the test expression is evaluated for each tuple of variable bindings. Results depend on the effective boolean value of the test expressions, as defined in **2.4.3 Effective Boolean Value**. The value of the quantified expression is defined by the following rules:

1. If the quantifier is `some`, the quantified expression is `true` if at least one evaluation of the test expression has the effective boolean value `true`; otherwise the quantified expression is `false`. This rule implies that, if the in-clauses generate zero binding tuples, the value of the quantified expression is `false`.
2. If the quantifier is `every`, the quantified expression is `true` if every evaluation of the test expression has the effective boolean value `true`; otherwise the quantified expression is `false`. This rule implies that, if the in-clauses generate zero binding tuples, the value of the quantified expression is `true`.

The scope of a variable bound in a quantified expression comprises all subexpressions of the quantified expression that appear after the variable binding. The scope does not include the expression to which the variable is bound.

The order in which test expressions are evaluated for the various binding tuples is implementation-dependent. If the quantifier is `some`, an implementation may return `true` as soon as it finds one binding tuple for which the test expression has an effective boolean value of `true`, and it may raise a dynamic error as soon as it finds one binding tuple for which the test expression raises an error. Similarly, if the quantifier is `every`, an implementation may return `false` as soon as it finds one binding tuple for which the test expression has an effective boolean value of `false`, and it may raise a dynamic error as soon as it finds one binding tuple for which the test expression raises an error. As a result of these rules, the value of a quantified expression is not deterministic in the presence of errors, as illustrated in the examples below.

Here are some examples of quantified expressions:

- This expression is `true` if every `part` element has a `discounted` attribute (regardless of the values of these attributes):

```
every $part in /parts/part satisfies $part/@discounted
```

- This expression is `true` if at least one `employee` element satisfies the given comparison expression:

```
some $emp in /emps/employee satisfies
     ($emp/bonus > 0.25 * $emp/salary)
```

- In the following examples, each quantified expression evaluates its test expression over nine tuples of variable bindings, formed from the Cartesian product of the sequences `(1, 2, 3)` and `(2, 3, 4)`. The expression beginning with `some` evaluates to `true`, and the expression beginning with `every` evaluates to `false`.

```
some $x in (1, 2, 3), $y in (2, 3, 4)
     satisfies $x + $y = 4
```

```
every $x in (1, 2, 3), $y in (2, 3, 4)
     satisfies $x + $y = 4
```

- This quantified expression may either return `true` or raise a type error, since its test expression returns `true` for one variable binding and raises a type error for another:

```
some $x in (1, 2, "cat") satisfies $x * 2 = 4
```

- This quantified expression may either return `false` or raise a type error, since its test expression returns `false` for one variable binding and raises a type error for another:

```
every $x in (1, 2, "cat") satisfies $x * 2 = 4
```

3.10 Expressions on SequenceTypes

sequence types are used in `instance of`, `cast`, `castable`, and `treat` expressions.

3.10.1 Instance Of

[16]	`InstanceofExpr`	::=	`TreatExpr ( "instance" "of" SequenceType )?`

The boolean operator `instance of` returns `true` if the value of its first operand matches the SequenceType in its second operand, according to the rules for SequenceType matching; otherwise it returns `false`. For example:

- `5 instance of xs:integer`

  This example returns `true` because the given value is an instance of the given type.
- `5 instance of xs:decimal`

  This example returns `true` because the given value is an integer literal, and `xs:integer` is derived by restriction from `xs:decimal`.
- `(5, 6) instance of xs:integer+`

  This example returns `true` because the given sequence contains two integers, and is a valid instance of the specified type.
- `. instance of element()`

  This example returns `true` if the context item is an element node or `false` if the context item is defined but is not an element node. If the context item is undefined, a dynamic error is raised [err:XPDY0002].

3.10.2 Cast

[19]	`CastExpr`	::=	`UnaryExpr ( "cast" "as" SingleType )?`
[49]	`SingleType`	::=	`AtomicType "?"?`

Occasionally it is necessary to convert a value to a specific datatype. For this purpose, XPath provides a `cast` expression that creates a new value of a specific type based on an existing value. A `cast` expression takes two operands: an **input expression** and a **target type**. The type of the input expression is called the **input type**. The target type must be an atomic type that is in the in-scope schema types [err:XPST0051]. In addition, the target type cannot be `xs:NOTATION` or `xs:anyAtomicType` [err:XPST0080]. The optional occurrence indicator "`?`" denotes that an empty sequence is permitted. If the target type has no namespace prefix, it is considered to be in the default element/type namespace. The semantics of the `cast` expression are as follows:

1. Atomization is performed on the input expression.
2. If the result of atomization is a sequence of more than one atomic value, a type error is raised [err:XPTY0004].
3. If the result of atomization is an empty sequence:
   a. If `?` is specified after the target type, the result of the `cast` expression is an empty sequence.

   b. If `?` is not specified after the target type, a type error is raised [err:XPTY0004].

4. If the result of atomization is a single atomic value, the result of the cast expression depends on the input type and the target type. In general, the cast expression attempts to create a new value of the target type based on the input value. Only certain combinations of input type and target type are supported. A summary of the rules are listed below— the normative definition of these rules is given in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)]. For the purpose of these rules, an implementation may determine that one type is derived by restriction from another type either by examining the in-scope schema definitions or by using an alternative, implementation-dependent mechanism such as a data dictionary.

   a. `cast` is supported for the combinations of input type and target type listed in Section 17.1 Casting from primitive types to primitive types[FO]. For each of these combinations, both the input type and the target type are primitive schema types. For example, a value of type `xs:string` can be cast into the schema type `xs:decimal`. For each of these built-in combinations, the semantics of casting are specified in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

      If the target type of a `cast` expression is `xs:QName`, or is a type that is derived from `xs:QName` or `xs:NOTATION`, and if the base type of the input is not the same as the base type of the target type, then the input expression must be a string literal [err:XPTY0004].

      Note:

      The reason for this rule is that construction of an instance of one of these target types from a string requires knowledge about namespace bindings. If the input expression is a non-literal string, it might be derived from an input document whose namespace bindings are different from the statically known namespaces.

   b. `cast` is supported if the input type is a non-primitive atomic type that is derived by restriction from the target type. In this case, the input value is mapped into the value space of the target type, unchanged except for its type. For example, if `shoesize` is derived by restriction from `xs:integer`, a value of type `shoesize` can be cast into the schema type `xs:integer`.

   c. `cast` is supported if the target type is a non-primitive atomic type and the input type is `xs:string` or `xs:untypedAtomic`. The input value is first converted to a value in the lexical space of the target type by applying the whitespace normalization rules for the target type (as defined in [XML Schema]). The lexical value is then converted to the value space of the target type using the schema-defined rules for the target type. If the input value fails to satisfy some facet of the target type, a dynamic error may be raised as specified in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

   d. `cast` is supported if the target type is a non-primitive atomic type that is derived by restriction from the input type. The input value must satisfy all the facets of the target type (in the case of the pattern facet, this is checked by generating a string representation of the input value, using the rules for casting to `xs:string`). The resulting value is the same as the input value, but with a different dynamic type.

   e. If a primitive type P1 can be cast into a primitive type P2, then any type derived by restriction from P1 can be cast into any type derived by restriction from P2, provided that the facets of the target type are satisfied. First the input value is cast to P1 using rule (b) above. Next, the value of type P1 is cast to the type P2, using rule (a) above. Finally, the value of type P2 is cast to the target type, using rule (d) above.

   f. For any combination of input type and target type that is not in the above list, a `cast` expression raises a type error [err:XPTY0004].

If casting from the input type to the target type is supported but nevertheless it is not possible to cast the input value into the value space of the target type, a dynamic error is raised. [err:FORG0001] This includes the case when any facet of the target type is not satisfied. For example, the expression `"2003-02-31" cast as xs:date` would raise a dynamic error.

3.10.3 Castable

[18]	`CastableExpr`	::=	`CastExpr ( "castable" "as" SingleType )?`
[49]	`SingleType`	::=	`AtomicType "?"?`

XPath provides an expression that tests whether a given value is castable into a given target type. The target type must be an atomic type that is in the in-scope schema types [err:XPST0051]. In addition, the target type cannot be `xs:NOTATION` or `xs:anyAtomicType` [err:XPST0080]. The optional occurrence indicator "`?`" denotes that an empty sequence is permitted.

The expression `E castable as T` returns `true` if the result of evaluating `E` can be successfully cast into the target type `T` by using a `cast` expression; otherwise it returns `false`. If evaluation of `E` fails with a dynamic error, the `castable` expression as a whole fails. The `castable` expression can be used as a predicate to avoid errors at evaluation time. It can also be used to select an appropriate type for processing of a given value, as illustrated in the following example:

```
if ($x castable as hatsize)
   then $x cast as hatsize
   else if ($x castable as IQ)
   then $x cast as IQ
   else $x cast as xs:string
```

Note:

If the target type of a `castable` expression is `xs:QName`, or is a type that is derived from `xs:QName` or `xs:NOTATION`, and the input argument of the expression is of type `xs:string` but it is not a literal string, the result of the `castable` expression is `false`.

3.10.4 Constructor Functions

For every atomic type in the in-scope schema types (except `xs:NOTATION` and `xs:anyAtomicType`, which are not instantiable), a **constructor function** is implicitly defined. In each case, the name of the constructor function is the same as the name of its target type (including namespace). The signature of the constructor function for type *T* is as follows:

```
T($arg as xs:anyAtomicType?) as T?
```

[Definition: The **constructor function** for a given type is used to convert instances of other atomic types into the given type. The semantics of the constructor function call `T($arg)` are defined to be equivalent to the expression `(($arg) cast as T?)`.]

The constructor functions for `xs:QName` and for types derived from `xs:QName` and `xs:NOTATION` require their arguments to be string literals or to have a base type that is the same as the base type of the target type; otherwise a type error [err:XPTY0004] is raised. This rule is consistent with the semantics of `cast` expressions for these types, as defined in **3.10.2 Cast**.

The following examples illustrate the use of constructor functions:

- This example is equivalent to `("2000-01-01" cast as xs:date?)`.

  ```
  xs:date("2000-01-01")
  ```

- This example is equivalent to `(($floatvalue * 0.2E-5) cast as xs:decimal?)`.

  ```
  xs:decimal($floatvalue * 0.2E-5)
  ```

- This example returns a `xs:dayTimeDuration` value equal to 21 days. It is equivalent to `("P21D" cast as xs:dayTimeDuration?)`.

  ```
  xs:dayTimeDuration("P21D")
  ```

- If `usa:zipcode` is a user-defined atomic type in the in-scope schema types, then the following expression is equivalent to the expression `("12345" cast as usa:zipcode?)`.

  ```
  usa:zipcode("12345")
  ```

Note:

An instance of an atomic type that is not in a namespace can be constructed in either of the following ways:

- By using a `cast` expression, if the default element/type namespace is "none".

```
17 cast as apple
```

- By using a constructor function, if the default function namespace is "none".

```
apple(17)
```

3.10.5 Treat

[17] `TreatExpr` ::= `CastableExpr ( "treat" "as" SequenceType )?`

XPath provides an expression called `treat` that can be used to modify the static type of its operand.

Like `cast`, the `treat` expression takes two operands: an expression and a SequenceType. Unlike `cast`, however, `treat` does not change the dynamic type or value of its operand. Instead, the purpose of `treat` is to ensure that an expression has an expected dynamic type at evaluation time.

The semantics of *`expr1`* `treat as` *`type1`* are as follows:

- During static analysis:

  The static type of the `treat` expression is *`type1`*. This enables the expression to be used as an argument of a function that requires a parameter of *`type1`*.
- During expression evaluation:

  If *`expr1`* matches *`type1`*, using the rules for SequenceType matching, the `treat` expression returns the value of *`expr1`*; otherwise, it raises a dynamic error [err:XPDY0050]. If the value of *`expr1`* is returned, its identity is preserved. The `treat` expression ensures that the value of its expression operand conforms to the expected type at run-time.
- Example:

  ```
  $myaddress treat as element(*, USAddress)
  ```

  The static type of `$myaddress` may be `element(*, Address)`, a less specific type than `element(*, USAddress)`. However, at run-time, the value of `$myaddress` must match the type `element(*, USAddress)` using rules for SequenceType matching; otherwise a dynamic error is raised [err:XPDY0050].

A XPath Grammar

A.1 EBNF

The grammar of XPath uses the same simple Extended Backus-Naur Form (EBNF) notation as [XML 1.0] with the following minor differences.

- All named symbols have a name that begins with an uppercase letter.
- It adds a notation for referring to productions in external specs.
- Comments or extra-grammatical constraints on grammar productions are between '/*' and '*/' symbols.
  - A 'xgc:' prefix is an extra-grammatical constraint, the details of which are explained in **A.1.2 Extra-grammatical Constraints**
  - A 'ws:' prefix explains the whitespace rules for the production, the details of which are explained in **A.2.4 Whitespace Rules**
  - A 'gn:' prefix means a 'Grammar Note', and is meant as a clarification for parsing rules, and is explained in **A.1.3 Grammar Notes**. These notes are not normative.

The terminal symbols for this grammar include the quoted strings used in the production rules below, and the terminal symbols defined in section **A.2.1 Terminal Symbols**.

The EBNF notation is described in more detail in **A.1.1 Notation**.

To increase readability, the EBNF in the main body of this document omits some of these notational features. This appendix is the normative version of the EBNF.

```
[1]  XPath               ::= Expr
[2]  Expr                ::= ExprSingle ("," ExprSingle)*
[3]  ExprSingle          ::= ForExpr
                             | QuantifiedExpr
                             | IfExpr
                             | OrExpr
[4]  ForExpr             ::= SimpleForClause "return" ExprSingle
[5]  SimpleForClause     ::= "for" "$" VarName "in" ExprSingle ("," "$"
                             VarName "in" ExprSingle)*
[6]  QuantifiedExpr      ::= ("some" | "every") "$" VarName "in"
                             ExprSingle ("," "$" VarName "in" ExprSingle)*
                             "satisfies" ExprSingle
[7]  IfExpr              ::= "if" "(" Expr ")" "then" ExprSingle "else"
                             ExprSingle
[8]  OrExpr              ::= AndExpr ( "or" AndExpr )*
[9]  AndExpr             ::= ComparisonExpr ( "and" ComparisonExpr )*
[10] ComparisonExpr      ::= RangeExpr ( (ValueComp
                             | GeneralComp
                             | NodeComp) RangeExpr )?
[11] RangeExpr           ::= AdditiveExpr ( "to" AdditiveExpr )?
[12] AdditiveExpr        ::= MultiplicativeExpr ( ("+" | "-")
                             MultiplicativeExpr )*
[13] MultiplicativeExpr  ::= UnionExpr ( ("*" | "div" | "idiv" | "mod")
                             UnionExpr )*
[14] UnionExpr           ::= IntersectExceptExpr ( ("union" | "|")
                             IntersectExceptExpr )*
[15] IntersectExceptExpr ::= InstanceofExpr ( ("intersect" | "except")
                             InstanceofExpr )*
[16] InstanceofExpr      ::= TreatExpr ( "instance" "of" SequenceType )?
[17] TreatExpr           ::= CastableExpr ( "treat" "as" SequenceType )?
[18] CastableExpr        ::= CastExpr ( "castable" "as" SingleType )?
[19] CastExpr            ::= UnaryExpr ( "cast" "as" SingleType )?
[20] UnaryExpr           ::= ("-" | "+")* ValueExpr
[21] ValueExpr           ::= PathExpr
[22] GeneralComp         ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[23] ValueComp           ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[24] NodeComp            ::= "is" | "<<" | ">>"
[25] PathExpr            ::= ("/" RelativePathExpr?)          /* xgs:
                             | ("//" RelativePathExpr)           leading-lone-
                             | RelativePathExpr                  slash */
[26] RelativePathExpr    ::= StepExpr (("/" | "//") StepExpr)*
[27] StepExpr            ::= FilterExpr | AxisStep
[28] AxisStep            ::= (ReverseStep | ForwardStep) PredicateList
[29] ForwardStep         ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[30] ForwardAxis         ::= ("child" "::")
                             | ("descendant" "::")
                             | ("attribute" "::")
                             | ("self" "::")
                             | ("descendant-or-self" "::")
                             | ("following-sibling" "::")
                             | ("following" "::")
                             | ("namespace" "::")
[31] AbbrevForwardStep   ::= "@"? NodeTest
[32] ReverseStep         ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[33] ReverseAxis         ::= ("parent" "::")
```

			| ("ancestor" "::") | ("preceding-sibling" "::") | ("preceding" "::") | ("ancestor-or-self" "::")	
[34]	AbbrevReverseStep	::=	".."	
[35]	NodeTest	::=	KindTest | NameTest	
[36]	NameTest	::=	QName | Wildcard	
[37]	Wildcard	::=	"*" | (NCName ":" "*") | ("*" ":" NCName)	/* ws: explicit */
[38]	FilterExpr	::=	PrimaryExpr PredicateList	
[39]	PredicateList	::=	Predicate*	
[40]	Predicate	::=	"[" Expr "]"	
[41]	PrimaryExpr	::=	Literal | VarRef | ParenthesizedExpr | ContextItemExpr | FunctionCall	
[42]	Literal	::=	NumericLiteral | StringLiteral	
[43]	NumericLiteral	::=	IntegerLiteral | DecimalLiteral | DoubleLiteral	
[44]	VarRef	::=	"$" VarName	
[45]	VarName	::=	QName	
[46]	ParenthesizedExpr	::=	"(" Expr? ")"	
[47]	ContextItemExpr	::=	"."	
[48]	FunctionCall	::=	QName "(" (ExprSingle ("," ExprSingle)*)? ")"	/* xgs: reserved-function-names */ /* gn: parens */
[49]	SingleType	::=	AtomicType "?"?	
[50]	SequenceType	::=	("empty-sequence" "(" ")") | (ItemType OccurrenceIndicator?)	
[51]	OccurrenceIndicator	::=	"?" | "*" | "+"	/* xgs: occurrence-indicators */
[52]	ItemType	::=	KindTest | ("item" "(" ")") | AtomicType	
[53]	AtomicType	::=	QName	
[54]	KindTest	::=	DocumentTest | ElementTest | AttributeTest | SchemaElementTest | SchemaAttributeTest | PITest | CommentTest | TextTest | AnyKindTest	
[55]	AnyKindTest	::=	"node" "(" ")"	
[56]	DocumentTest	::=	"document-node" "(" (ElementTest | SchemaElementTest)? ")"	
[57]	TextTest	::=	"text" "(" ")"	
[58]	CommentTest	::=	"comment" "(" ")"	
[59]	PITest	::=	"processing-instruction" "(" (NCName | StringLiteral)? ")"	
[60]	AttributeTest	::=	"attribute" "(" (AttribNameOrWildcard ("," TypeName)?)? ")"	
[61]	AttribNameOrWildcard	::=	AttributeName | "*"	
[62]	SchemaAttributeTest	::=	"schema-attribute" "(" AttributeDeclaration ")"	

[63]	`AttributeDeclaration`	::=	`AttributeName`
[64]	`ElementTest`	::=	`"element" "(" (ElementNameOrWildcard ("," TypeName "?"?)?)? ")"`
[65]	`ElementNameOrWildcard`	::=	`ElementName | "*"`
[66]	`SchemaElementTest`	::=	`"schema-element" "(" ElementDeclaration ")"`
[67]	`ElementDeclaration`	::=	`ElementName`
[68]	`AttributeName`	::=	`QName`
[69]	`ElementName`	::=	`QName`
[70]	`TypeName`	::=	`QName`

A.1.1 Notation

The following definitions will be helpful in defining precisely this exposition.

[Definition: Each rule in the grammar defines one **symbol**, using the following format:

```
symbol ::= expression
```

]

[Definition: A **terminal** is a symbol or string or pattern that can appear in the right-hand side of a rule, but never appears on the left hand side in the main grammar, although it may appear on the left-hand side of a rule in the grammar for terminals.] The following constructs are used to match strings of one or more characters in a terminal:

[a-zA-Z]

matches any Char with a value in the range(s) indicated (inclusive).

[abc]

matches any Char with a value among the characters enumerated.

[^abc]

matches any Char with a value not among the characters given.

"string"

matches the sequence of characters that appear inside the double quotes.

'string'

matches the sequence of characters that appear inside the single quotes.

[http://www.w3.org/TR/REC-example/#NT-Example]

matches any string matched by the production defined in the external specification as per the provided reference.

Patterns (including the above constructs) can be combined with grammatical operators to form more complex patterns, matching more complex sets of character strings. In the examples that follow, A and B represent (sub-)patterns.

(A)

`A` is treated as a unit and may be combined as described in this list.

A?

matches `A` or nothing; optional `A`.

A B

matches `A` followed by `B`. This operator has higher precedence than alternation; thus `A B | C D` is identical to `(A B) | (C D)`.

A | B

matches `A` or `B` but not both.

A - B

matches any string that matches `A` but does not match `B`.

A+

matches one or more occurrences of `A`. Concatenation has higher precedence than alternation; thus `A+ | B+` is identical to `(A+) | (B+)`.

A*

matches zero or more occurrences of `A`. Concatenation has higher precedence than alternation; thus `A* | B*` is identical to `(A*) | (B*)`

A.1.2 Extra-grammatical Constraints

This section contains constraints on the EBNF productions, which are required to parse legal sentences. The notes below are referenced from the right side of the production, with the notation: */* xgc: <id> */*.

Constraint: leading-lone-slash

A single slash may appear either as a complete path expression or as the first part of a path expression in which it is followed by a RelativePathExpr. In some cases, the next token after the slash is insufficient to allow a parser to distinguish these two possibilities: the `*` token and keywords like `union` could be either an operator or a NameTest . For example, without lookahead the first part of the expression `/ * 5` is easily taken to be a complete expression, `/ *`, which has a very different interpretation (the child nodes of `/`).

Therefore to reduce the need for lookahead, if the token immediately following a slash can form the start of a RelativePathExpr, then the slash must be the beginning of a PathExpr, not the entirety of it.

A single slash may be used as the left-hand argument of an operator by parenthesizing it: `(/) * 5`. The expression `5 * /`, on the other hand, is legal without parentheses.

Constraint: xml-version

An implementation's choice to support the [XML 1.0] and [XML Names], or [XML 1.1] and [XML Names 1.1] lexical specification determines the external document from which to obtain the definition for this production. The EBNF only has references to the 1.0 versions. In some cases, the XML 1.0 and XML 1.1 definitions may be exactly the same. Also please note that these external productions follow the whitespace rules of their respective specifications, and not the rules of this specification, in particular **A.2.4.1 Default Whitespace Handling**. Thus `prefix : localname` is not a valid QName for purposes of this specification, just as it is not permitted in a XML document. Also, comments are not permissible on either side of the colon. Also extra-grammatical constraints such as well-formedness constraints must be taken into account.

Constraint: reserved-function-names

Unprefixed function names spelled the same way as language keywords could make the language harder to recognize. For instance, `if(foo)` could be taken either as a FunctionCall or as the beginning of an IfExpr. Therefore it is not legal syntax for a user to invoke functions with unprefixed names which match any of the names in **A.3 Reserved Function Names**.

A function named "if" can be called by binding its namespace to a prefix and using the prefixed form: "library:if(foo)" instead of "if(foo)".

Constraint: occurrence-indicators

As written, the grammar in **A XPath Grammar** is ambiguous for some forms using the '+' and '*' Kleene operators. The ambiguity is resolved as follows: these operators are tightly bound to the SequenceType expression, and have higher precedence than other uses of these symbols. Any occurrence of '+' and '*', as well as '?', following a sequence type is assumed to be an occurrence indicator. That is, a "+", "*", or "?" immediately following an ItemType must be an OccurrenceIndicator. Thus, `4 treat as item() + - 5` must be interpreted as `(4 treat as item()+) - 5`, taking the '+' as an OccurrenceIndicator and the '-' as a subtraction operator. To force the interpretation of "+" as an addition operator (and the corresponding interpretation of the "-" as a unary minus), parentheses may be used: the form `(4 treat as item()) + -5` surrounds the SequenceType expression with parentheses and leads to the desired interpretation.

This rule has as a consequence that certain forms which would otherwise be legal and unambiguous are not recognized: in "4 treat as item() + 5", the "+" is taken as an OccurrenceIndicator, and not as an operator, which means this is not a legal expression.

A.1.3 Grammar Notes

This section contains general notes on the EBNF productions, which may be helpful in understanding how to interpret and implement the EBNF. These notes are not normative. The notes below are referenced from the right side of the production, with the notation: */* gn: <id> */*.

Note:

grammar-note: parens

Look-ahead is required to distinguish FunctionCall from a QName or keyword followed by a Comment. For example: `address (: this may be empty :)` may be mistaken for a call to a function named "address" unless this lookahead is employed. Another example is `for (: whom the bell :) $tolls in 3 return $tolls`, where the keyword "for" must not be mistaken for a function name.

grammar-note: comments

Comments are allowed everywhere that ignorable whitespace is allowed, and the Comment symbol does not explicitly appear on the right-hand side of the grammar (except in its own production). See **A.2.4.1 Default Whitespace Handling**.

A comment can contain nested comments, as long as all "(:" and ":)" patterns are balanced, no matter where they occur within the outer comment.

Note:

Lexical analysis may typically handle nested comments by incrementing a counter for each "(:" pattern, and decrementing the counter for each ":)" pattern. The comment does not terminate until the counter is back to zero.

Some illustrative examples:

- `(: commenting out a (: comment :) may be confusing, but often helpful :)` is a legal Comment, since balanced nesting of comments is allowed.
- `"this is just a string :)"` is a legal expression. However, `(: "this is just a string :)" :)` will cause a syntax error. Likewise, `"this is another string (:"` is a legal expression, but `(: "this is another string (:" :)` will cause a syntax error. It is a limitation of nested comments that literal content can cause unbalanced nesting of comments.
- `for (: set up loop :) $i in $x return $i` is syntactically legal, ignoring the comment.
- `5 instance (: strange place for a comment :) of xs:integer` is also syntactically valid.

A.2 Lexical structure

The terminal symbols assumed by the grammar above are described in this section.

Quoted strings appearing in production rules are terminal symbols.

Other terminal symbols are defined in **A.2.1 Terminal Symbols**.

A **host language** may choose whether the lexical rules of [XML 1.0] and [XML Names] are followed, or alternatively, the lexical rules of [XML 1.1] and [XML Names 1.1] are followed.

When tokenizing, the longest possible match that is valid in the current context is used.

All keywords are case sensitive. Keywords are not reserved—that is, any QName may duplicate a keyword except as noted in **A.3 Reserved Function Names**.

A.2.1 Terminal Symbols

[71]	`IntegerLiteral`	::=	`Digits`	
[72]	`DecimalLiteral`	::=	`("." Digits) | (Digits "." [0-9]*)`	/* ws: explicit */
[73]	`DoubleLiteral`	::=	`(("." Digits) | (Digits ("." [0-9]*)?)) [eE] [+-]? Digits`	/* ws: explicit */
[74]	`StringLiteral`	::=	`('"' (EscapeQuot | [^"])* '"') | ("'" (EscapeApos | [^'])* "'")`	/* ws: explicit */
[75]	`EscapeQuot`	::=	`'""'`	
[76]	`EscapeApos`	::=	`"''"`	
[77]	`Comment`	::=	`"(:" (CommentContents | Comment)* ":)"`	/* ws: explicit */ /* gn: comments */
[78]	`QName`	::=	`[http://www.w3.org/TR/REC-xml-names/#NT-QName]`Names	/* xgs: xml-version */
[79]	`NCName`	::=	`[http://www.w3.org/TR/REC-xml-names/#NT-NCName]`Names	/* xgs: xml-version */
[80]	`Char`	::=	`[http://www.w3.org/TR/REC-xml#NT-Char]`XML	/* xgs: xml-version */

The following symbols are used only in the definition of terminal symbols; they are not terminal symbols in the grammar of **A.1 EBNF**.

[81]	`Digits`	::=	`[0-9]+`
[82]	`CommentContents`	::=	`(Char+ - (Char* ('(:' | ':)') Char*))`

A.2.2 Terminal Delimitation

XPath 2.0 expressions consist of terminal symbols and symbol separators.

Terminal symbols that are not used exclusively in /* ws: explicit */ productions are of two kinds: delimiting and non-delimiting.

[Definition: The **delimiting terminal symbols** are: "!=", StringLiteral, "$", "(", ")", "*", "+", (comma), "-", (dot), "..", "/", "//", (colon), "::", "<", "<<", "<=", "=", ">", ">=", ">>", "?", "@", "[", "]", "|"]

[Definition: The **non-delimiting terminal symbols** are: IntegerLiteral, NCName, DecimalLiteral, DoubleLiteral, QName, "ancestor", "ancestor-or-self", "and", "as", "attribute", "cast", "castable", "child", "comment", "descendant", "descendant-or-self", "div", "document-node", "element", "else", "empty-sequence", "eq", "every", "except", "external", "following", "following-sibling", "for", "ge", "gt", "idiv", "if", "in", "instance", "intersect", "is", "item", "le", "lt", "mod", "namespace", "ne", "node", "of", "or", "parent", "preceding", "preceding-sibling", "processing-instruction", "return", "satisfies", "schema-attribute", "schema-element", "self", "some", "text", "then", "to", "treat", "union"]

[Definition: Whitespace and Comments function as **symbol separators**. For the most part, they are not mentioned in the grammar, and may occur between any two terminal symbols mentioned in the grammar, except where that is forbidden by the /* ws: explicit */ annotation in the EBNF, or by the /* xgs: xml-version */ annotation.]

It is customary to separate consecutive terminal symbols by whitespace and Comments, but this is required only when otherwise two non-delimiting symbols would be adjacent to each other. There are two exceptions to this, that of "." and "-", which do require a symbol separator if they follow a QName or NCName. Also, "." requires a separator if it precedes or follows a numeric literal.

A.2.3 End-of-Line Handling

The XPath processor must behave as if it normalized all line breaks on input, before parsing. The normalization should be done according to the choice to support either [XML 1.0] or [XML 1.1] lexical processing.

A.2.3.1 XML 1.0 End-of-Line Handling

For [XML 1.0] processing, all of the following must be translated to a single #xA character:

1. the two-character sequence #xD #xA
2. any #xD character that is not immediately followed by #xA.

A.2.3.2 XML 1.1 End-of-Line Handling

For [XML 1.1] processing, all of the following must be translated to a single #xA character:

1. the two-character sequence #xD #xA
2. the two-character sequence #xD #x85
3. the single character #x85
4. the single character #x2028
5. any #xD character that is not immediately followed by #xA or #x85.

A.2.4 Whitespace Rules

A.2.4.1 Default Whitespace Handling

[Definition: A **whitespace** character is any of the characters defined by [http://www.w3.org/TR/REC-xml/#NT-S].]

[Definition: **Ignorable whitespace** consists of any whitespace characters that may occur between terminals, unless these characters occur in the context of a production marked with a ws:explicit annotation, in which case they can occur only where explicitly specified (see **A.2.4.2 Explicit Whitespace Handling**).] Ignorable whitespace characters are not significant to the semantics of an expression. Whitespace is allowed before the first terminal and after the last terminal of a module. Whitespace is allowed between any two terminals. Comments may also act as "whitespace" to prevent two adjacent terminals from being recognized as one. Some illustrative examples are as follows:

- `foo- foo` results in a syntax error. "foo-" would be recognized as a QName.
- `foo -foo` is syntactically equivalent to `foo - foo`, two QNames separated by a subtraction operator.
- `foo(: This is a comment :)- foo` is syntactically equivalent to `foo - foo`. This is because the comment prevents the two adjacent terminals from being recognized as one.
- `foo-foo` is syntactically equivalent to single QName. This is because "-" is a valid character in a QName. When used as an operator after the characters of a name, the "-" must be separated from the name, e.g. by using whitespace or parentheses.
- `10div 3` results in a syntax error.
- `10 div3` also results in a syntax error.
- `10div3` also results in a syntax error.

A.2.4.2 Explicit Whitespace Handling

Explicit whitespace notation is specified with the EBNF productions, when it is different from the default rules, using the notation shown below. This notation is not inherited. In other words, if an EBNF rule is marked as /* ws: explicit */, the notation does not automatically apply to all the 'child' EBNF productions of that rule.

ws: explicit

/* ws: explicit */ means that the EBNF notation explicitly notates, with `S` or otherwise, where whitespace characters are allowed. In productions with the /* ws: explicit */ annotation, **A.2.4.1 Default Whitespace Handling** does not apply. Comments are also not allowed in these productions.

A.3 Reserved Function Names

The following names are not allowed as function names in an unprefixed form because expression syntax takes precedence.

- `attribute`
- `comment`
- `document-node`
- `element`
- `empty-sequence`
- `if`
- `item`
- `node`
- `processing-instruction`
- `schema-attribute`
- `schema-element`
- `text`
- `typeswitch`

Note:

Although the keyword `typeswitch` is not used in XPath, it is considered a reserved function name for compatibility with XQuery.

A.4 Precedence Order

The grammar in **A.1 EBNF** normatively defines built-in precedence among the operators of XPath. These operators are summarized here to make clear the order of their precedence from lowest to highest. The associativity column indicates the order in which operators of equal precedence in an expression are applied.

#	Operator	Associativity
1	, (comma)	left-to-right
3	for, some, every, if	left-to-right
4	or	left-to-right
5	and	left-to-right
6	eq, ne, lt, le, gt, ge, =, !=, <, <=, >, >=, is, <<, >>	left-to-right
7	to	left-to-right
8	+, -	left-to-right
9	*, div, idiv, mod	left-to-right
10	union, |	left-to-right
11	intersect, except	left-to-right

12	instance of	left-to-right
13	treat	left-to-right
14	castable	left-to-right
15	cast	left-to-right
16	-(unary), +(unary)	right-to-left
17	?, *(OccurrenceIndicator), +(OccurrenceIndicator)	left-to-right
18	/, //	left-to-right
19	[ ]	left-to-right

Note:

Parentheses can be used to override the operator precedence in the usual way. Square brackets in an expression such as A[B] serve two roles: they act as an operator causing B to be evaluated once for each item in the value of A, and they act as parentheses enclosing the expression B.

B Type Promotion and Operator Mapping

B.1 Type Promotion

[Definition: Under certain circumstances, an atomic value can be promoted from one type to another. **Type promotion** is used in evaluating function calls (see **3.1.5 Function Calls**) and operators that accept numeric or string operands (see **B.2 Operator Mapping**).] The following type promotions are permitted:

1. Numeric type promotion:
   a. A value of type `xs:float` (or any type derived by restriction from `xs:float`) can be promoted to the type `xs:double`. The result is the `xs:double` value that is the same as the original value.
   b. A value of type `xs:decimal` (or any type derived by restriction from `xs:decimal`) can be promoted to either of the types `xs:float` or `xs:double`. The result of this promotion is created by casting the original value to the required type. This kind of promotion may cause loss of precision.
2. URI type promotion: A value of type `xs:anyURI` (or any type derived by restriction from `xs:anyURI`) can be promoted to the type `xs:string`. The result of this promotion is created by casting the original value to the type `xs:string`.

   Note:

   Since `xs:anyURI` values can be promoted to `xs:string`, functions and operators that compare strings using the default collation also compare `xs:anyURI` values using the default collation. This ensures that orderings that include strings, `xs:anyURI` values, or any combination of the two types are consistent and well-defined.

Note that type promotion is different from subtype substitution. For example:

- A function that expects a parameter `$p` of type `xs:float` can be invoked with a value of type `xs:decimal`. This is an example of type promotion. The value is actually converted to the expected type. Within the body of the function, `$p instance of xs:decimal` returns `false`.
- A function that expects a parameter `$p` of type `xs:decimal` can be invoked with a value of type `xs:integer`. This is an example of subtype substitution. The value retains its original type. Within the body of the function, `$p instance of xs:integer` returns `true`.

B.2 Operator Mapping

The operator mapping tables in this section list the combinations of types for which the various operators of XPath are defined. [Definition: For each operator and valid combination of operand types, the operator mapping tables specify a result type and an **operator function** that implements the semantics of the operator for the given types.] The definitions of the operator functions are given in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)]. The result of an operator may be the raising of an error by its operator function, as defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)]. In some cases, the operator function does not implement the full semantics of a given operator. For the definition of each operator (including its behavior for empty sequences or sequences of length greater than one), see the descriptive material in the main part of this document.

The `and` and `or` operators are defined directly in the main body of this document, and do not occur in the operator mapping tables.

If an operator in the operator mapping tables expects an operand of type *ET*, that operator can be applied to an operand of type *AT* if type *AT* can be converted to type *ET* by a combination of type promotion and subtype substitution. For example, a table entry indicates that the `gt` operator may be applied to two `xs:date` operands, returning `xs:boolean`. Therefore, the `gt` operator may also be applied to two (possibly different) subtypes of `xs:date`, also returning `xs:boolean`.

[Definition: When referring to a type, the term **numeric** denotes the types `xs:integer`, `xs:decimal`, `xs:float`, and `xs:double`.] An operator whose operands and result are designated as numeric might be thought of as representing four operators, one for each of the numeric types. For example, the numeric `+` operator might be thought of as representing the following four operators:

Operator	First operand type	Second operand type	Result type
+	xs:integer	xs:integer	xs:integer
+	xs:decimal	xs:decimal	xs:decimal
+	xs:float	xs:float	xs:float
+	xs:double	xs:double	xs:double

A numeric operator may be validly applied to an operand of type *AT* if type *AT* can be converted to any of the four numeric types by a combination of type promotion and subtype substitution. If the result type of an operator is listed as numeric, it means "the first type in the ordered list `(xs:integer, xs:decimal, xs:float, xs:double)` into which all operands can be converted by subtype substitution and type promotion." As an example, suppose that the type `hatsize` is derived from `xs:integer` and the type `shoesize` is derived from `xs:float`. Then if the `+` operator is invoked with operands of type `hatsize` and `shoesize`, it returns a result of type `xs:float`. Similarly, if `+` is invoked with two operands of type `hatsize` it returns a result of type `xs:integer`.

[Definition: In the operator mapping tables, the term **Gregorian** refers to the types `xs:gYearMonth`, `xs:gYear`, `xs:gMonthDay`, `xs:gDay`, and `xs:gMonth`.] For binary operators that accept two Gregorian-type operands, both operands must have the same type (for example, if one operand is of type `xs:gDay`, the other operand must be of type `xs:gDay`.)

Binary Operators

Operator	Type(A)	Type(B)	Function	Result type
A + B	numeric	numeric	op:numeric-add(A, B)	numeric
A + B	xs:date	xs:yearMonthDuration	op:add-yearMonthDuration-to-date(A, B)	xs:date
A + B	xs:yearMonthDuration	xs:date	op:add-yearMonthDuration-to-date(B, A)	xs:date
A + B	xs:date	xs:dayTimeDuration	op:add-dayTimeDuration-to-date(A, B)	xs:date
A + B	xs:dayTimeDuration	xs:date	op:add-dayTimeDuration-to-date(B, A)	xs:date
A + B	xs:time	xs:dayTimeDuration	op:add-dayTimeDuration-to-time(A, B)	xs:time
A + B	xs:dayTimeDuration	xs:time	op:add-dayTimeDuration-to-time(B, A)	xs:time
A + B	xs:dateTime	xs:yearMonthDuration	op:add-yearMonthDuration-to-dateTime(A, B)	xs:dateTime
A + B	xs:yearMonthDuration	xs:dateTime	op:add-yearMonthDuration-to-dateTime(B, A)	xs:dateTime
A + B	xs:dateTime	xs:dayTimeDuration	op:add-dayTimeDuration-to-dateTime(A, B)	xs:dateTime
			op:add-dayTimeDuration-to-	

A + B	xs:dayTimeDuration	xs:dateTime	dateTime(B, A)	xs:dateTime
A + B	xs:yearMonthDuration	xs:yearMonthDuration	op:add-yearMonthDurations(A, B)	xs:yearMonthDuration
A + B	xs:dayTimeDuration	xs:dayTimeDuration	op:add-dayTimeDurations(A, B)	xs:dayTimeDuration
A - B	numeric	numeric	op:numeric-subtract(A, B)	numeric
A - B	xs:date	xs:date	op:subtract-dates(A, B)	xs:dayTimeDuration
A - B	xs:date	xs:yearMonthDuration	op:subtract-yearMonthDuration-from-date(A, B)	xs:date
A - B	xs:date	xs:dayTimeDuration	op:subtract-dayTimeDuration-from-date(A, B)	xs:date
A - B	xs:time	xs:time	op:subtract-times(A, B)	xs:dayTimeDuration
A - B	xs:time	xs:dayTimeDuration	op:subtract-dayTimeDuration-from-time(A, B)	xs:time
A - B	xs:dateTime	xs:dateTime	op:subtract-dateTimes(A, B)	xs:dayTimeDuration
A - B	xs:dateTime	xs:yearMonthDuration	op:subtract-yearMonthDuration-from-dateTime(A, B)	xs:dateTime
A - B	xs:dateTime	xs:dayTimeDuration	op:subtract-dayTimeDuration-from-dateTime(A, B)	xs:dateTime
A - B	xs:yearMonthDuration	xs:yearMonthDuration	op:subtract-yearMonthDurations(A, B)	xs:yearMonthDuration
A - B	xs:dayTimeDuration	xs:dayTimeDuration	op:subtract-dayTimeDurations(A, B)	xs:dayTimeDuration
A * B	numeric	numeric	op:numeric-multiply(A, B)	numeric
A * B	xs:yearMonthDuration	numeric	op:multiply-yearMonthDuration(A, B)	xs:yearMonthDuration
A * B	numeric	xs:yearMonthDuration	op:multiply-yearMonthDuration(B, A)	xs:yearMonthDuration
A * B	xs:dayTimeDuration	numeric	op:multiply-dayTimeDuration(A, B)	xs:dayTimeDuration
A * B	numeric	xs:dayTimeDuration	op:multiply-dayTimeDuration(B, A)	xs:dayTimeDuration
A idiv B	numeric	numeric	op:numeric-integer-divide(A, B)	xs:integer
A div B	numeric	numeric	op:numeric-divide(A, B)	numeric; but xs:decimal if both operands are xs:integer
A div B	xs:yearMonthDuration	numeric	op:divide-yearMonthDuration(A, B)	xs:yearMonthDuration
A div B	xs:dayTimeDuration	numeric	op:divide-dayTimeDuration(A, B)	xs:dayTimeDuration
A div B	xs:yearMonthDuration	xs:yearMonthDuration	op:divide-yearMonthDuration-by-yearMonthDuration (A, B)	xs:decimal
A div B	xs:dayTimeDuration	xs:dayTimeDuration	op:divide-dayTimeDuration-by-dayTimeDuration (A, B)	xs:decimal
A mod B	numeric	numeric	op:numeric-mod(A, B)	numeric
A eq B	numeric	numeric	op:numeric-equal(A, B)	xs:boolean

A eq B	xs:boolean	xs:boolean	op:boolean-equal(A, B)	xs:boolean
A eq B	xs:string	xs:string	op:numeric-equal(fn:compare(A, B), 0)	xs:boolean
A eq B	xs:date	xs:date	op:date-equal(A, B)	xs:boolean
A eq B	xs:time	xs:time	op:time-equal(A, B)	xs:boolean
A eq B	xs:dateTime	xs:dateTime	op:dateTime-equal(A, B)	xs:boolean
A eq B	xs:duration	xs:duration	op:duration-equal(A, B)	xs:boolean
A eq B	Gregorian	Gregorian	op:gYear-equal(A, B) etc.	xs:boolean
A eq B	xs:hexBinary	xs:hexBinary	op:hex-binary-equal(A, B)	xs:boolean
A eq B	xs:base64Binary	xs:base64Binary	op:base64-binary-equal(A, B)	xs:boolean
A eq B	xs:anyURI	xs:anyURI	op:numeric-equal(fn:compare(A, B), 0)	xs:boolean
A eq B	xs:QName	xs:QName	op:QName-equal(A, B)	xs:boolean
A eq B	xs:NOTATION	xs:NOTATION	op:NOTATION-equal(A, B)	xs:boolean
A ne B	numeric	numeric	fn:not(op:numeric-equal(A, B))	xs:boolean
A ne B	xs:boolean	xs:boolean	fn:not(op:boolean-equal(A, B))	xs:boolean
A ne B	xs:string	xs:string	fn:not(op:numeric-equal(fn:compare(A, B), 0))	xs:boolean
A ne B	xs:date	xs:date	fn:not(op:date-equal(A, B))	xs:boolean
A ne B	xs:time	xs:time	fn:not(op:time-equal(A, B))	xs:boolean
A ne B	xs:dateTime	xs:dateTime	fn:not(op:dateTime-equal(A, B))	xs:boolean
A ne B	xs:duration	xs:duration	fn:not(op:duration-equal(A, B))	xs:boolean
A ne B	Gregorian	Gregorian	fn:not(op:gYear-equal(A, B)) etc.	xs:boolean
A ne B	xs:hexBinary	xs:hexBinary	fn:not(op:hex-binary-equal(A, B))	xs:boolean
A ne B	xs:base64Binary	xs:base64Binary	fn:not(op:base64-binary-equal(A, B))	xs:boolean
A ne B	xs:anyURI	xs:anyURI	fn:not(op:numeric-equal(fn:compare(A, B), 0))	xs:boolean
A ne B	xs:QName	xs:QName	fn:not(op:QName-equal(A, B))	xs:boolean
A ne B	xs:NOTATION	xs:NOTATION	fn:not(op:NOTATION-equal(A, B))	xs:boolean
A gt B	numeric	numeric	op:numeric-greater-than(A, B)	xs:boolean
A gt B	xs:boolean	xs:boolean	op:boolean-greater-than(A, B)	xs:boolean
A gt B	xs:string	xs:string	op:numeric-greater-than(fn:compare(A, B), 0)	xs:boolean
A gt B	xs:date	xs:date	op:date-greater-than(A, B)	xs:boolean
A gt B	xs:time	xs:time	op:time-greater-than(A, B)	xs:boolean
A gt B	xs:dateTime	xs:dateTime	op:dateTime-greater-than(A, B)	xs:boolean

A gt B	xs:yearMonthDuration	xs:yearMonthDuration	op:yearMonthDuration-greater-than(A, B)	xs:boolean
A gt B	xs:dayTimeDuration	xs:dayTimeDuration	op:dayTimeDuration-greater-than(A, B)	xs:boolean
A gt B	xs:anyURI	xs:anyURI	op:numeric-greater-than(fn:compare(A, B), 0)	xs:boolean
A lt B	numeric	numeric	op:numeric-less-than(A, B)	xs:boolean
A lt B	xs:boolean	xs:boolean	op:boolean-less-than(A, B)	xs:boolean
A lt B	xs:string	xs:string	op:numeric-less-than(fn:compare(A, B), 0)	xs:boolean
A lt B	xs:date	xs:date	op:date-less-than(A, B)	xs:boolean
A lt B	xs:time	xs:time	op:time-less-than(A, B)	xs:boolean
A lt B	xs:dateTime	xs:dateTime	op:dateTime-less-than(A, B)	xs:boolean
A lt B	xs:yearMonthDuration	xs:yearMonthDuration	op:yearMonthDuration-less-than(A, B)	xs:boolean
A lt B	xs:dayTimeDuration	xs:dayTimeDuration	op:dayTimeDuration-less-than(A, B)	xs:boolean
A lt B	xs:anyURI	xs:anyURI	op:numeric-less-than(fn:compare(A, B), 0)	xs:boolean
A ge B	numeric	numeric	op:numeric-greater-than(A, B) or op:numeric-equal(A, B)	xs:boolean
A ge B	xs:boolean	xs:boolean	fn:not(op:boolean-less-than(A, B))	xs:boolean
A ge B	xs:string	xs:string	op:numeric-greater-than(fn:compare(A, B), -1)	xs:boolean
A ge B	xs:date	xs:date	fn:not(op:date-less-than(A, B))	xs:boolean
A ge B	xs:time	xs:time	fn:not(op:time-less-than(A, B))	xs:boolean
A ge B	xs:dateTime	xs:dateTime	fn:not(op:dateTime-less-than(A, B))	xs:boolean
A ge B	xs:yearMonthDuration	xs:yearMonthDuration	fn:not(op:yearMonthDuration-less-than(A, B))	xs:boolean
A ge B	xs:dayTimeDuration	xs:dayTimeDuration	fn:not(op:dayTimeDuration-less-than(A, B))	xs:boolean
A ge B	xs:anyURI	xs:anyURI	op:numeric-greater-than(fn:compare(A, B), -1)	xs:boolean
A le B	numeric	numeric	op:numeric-less-than(A, B) or op:numeric-equal(A, B)	xs:boolean
A le B	xs:boolean	xs:boolean	fn:not(op:boolean-greater-than(A, B))	xs:boolean
A le B	xs:string	xs:string	op:numeric-less-than(fn:compare(A, B), 1)	xs:boolean
A le B	xs:date	xs:date	fn:not(op:date-greater-than(A, B))	xs:boolean
A le B	xs:time	xs:time	fn:not(op:time-greater-than(A, B))	xs:boolean
A le B	xs:dateTime	xs:dateTime	fn:not(op:dateTime-greater-than(A, B))	xs:boolean
A le B	xs:yearMonthDuration	xs:yearMonthDuration	fn:not(op:yearMonthDuration-greater-than(A, B))	xs:boolean
A le B	xs:dayTimeDuration	xs:dayTimeDuration	fn:not(op:dayTimeDuration-	xs:boolean

			greater-than(A, B))	
A le B	xs:anyURI	xs:anyURI	op:numeric-less-than(fn:compare(A, B), 1)	xs:boolean
A is B	node()	node()	op:is-same-node(A, B)	xs:boolean
A << B	node()	node()	op:node-before(A, B)	xs:boolean
A >> B	node()	node()	op:node-after(A, B)	xs:boolean
A union B	node()*	node()*	op:union(A, B)	node()*
A | B	node()*	node()*	op:union(A, B)	node()*
A intersect B	node()*	node()*	op:intersect(A, B)	node()*
A except B	node()*	node()*	op:except(A, B)	node()*
A to B	xs:integer	xs:integer	op:to(A, B)	xs:integer*
A , B	item()*	item()*	op:concatenate(A, B)	item()*

Unary Operators

Operator	Operand type	Function	Result type
+ A	numeric	op:numeric-unary-plus(A)	numeric
- A	numeric	op:numeric-unary-minus(A)	numeric

C Context Components

The tables in this section describe the scope (range of applicability) of the various components in the static context and dynamic context.

C.1 Static Context Components

The following table describes the components of the **static context**. For each component, "global" indicates that the value of the component applies throughout an XPath expression, whereas "lexical" indicates that the value of the component applies only within the subexpression in which it is defined.

Static Context Components

Component	Scope
XPath 1.0 Compatibility Mode	global
Statically known namespaces	global
Default element/type namespace	global
Default function namespace	global
In-scope schema types	global
In-scope element declarations	global
In-scope attribute declarations	global
In-scope variables	lexical; for-expressions and quantified expressions can bind new variables
Context item static type	lexical
Function signatures	global
Statically known	

collations	global
Default collation	global
Base URI	global
Statically known documents	global
Statically known collections	global
Statically known default collection type	global

C.2 Dynamic Context Components

The following table describes how values are assigned to the various components of the **dynamic context**. All these components are initialized by mechanisms defined by the host language. For each component, "global" indicates that the value of the component remains constant throughout evaluation of the XPath expression, whereas "dynamic" indicates that the value of the component can be modified by the evaluation of subexpressions.

Dynamic Context Components

Component	Scope
Context item	dynamic; changes during evaluation of path expressions and predicates
Context position	dynamic; changes during evaluation of path expressions and predicates
Context size	dynamic; changes during evaluation of path expressions and predicates
Variable values	dynamic; for-expressions and quantified expressions can bind new variables
Current date and time	global; must be initialized by implementation
Implicit timezone	global; must be initialized by implementation
Available documents	global; must be initialized by implementation
Available collections	global; must be initialized by implementation
Default collection	global; overwriteable by implementation

D Implementation-Defined Items

The following items in this specification are implementation-defined:

1. The version of Unicode that is used to construct expressions.
2. The statically-known collations.
3. The implicit timezone.
4. The circumstances in which warnings are raised, and the ways in which warnings are handled.
5. The method by which errors are reported to the external processing environment.
6. Whether the implementation is based on the rules of [XML 1.0] and [XML Names] or the rules of [XML 1.1] and [XML Names 1.1]. One of these sets of rules must be applied consistently by all aspects of the implementation. If the implementation is based on the rules of [XML 1.0], the edition used must be at least Third Edition; the edition used is implementation-defined, but we recommend that implementations use the latest version.
7. Whether the implementation supports the namespace axis.

8. Any static typing extensions supported by the implementation, if the Static Typing Feature is supported.

Note:

Additional implementation-defined items are listed in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)] and [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

E References

E.1 Normative References

RFC 2119
: S. Bradner. *Key Words for use in RFCs to Indicate Requirement Levels.* IETF RFC 2119. See http://www.ietf.org/rfc/rfc2119.txt.

RFC3986
: T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. IETF RFC 3986. See http://www.ietf.org/rfc/rfc3986.txt.

RFC3987
: M. Duerst and M. Suignard. *Internationalized Resource Identifiers (IRIs)*. IETF RFC 3987. See http://www.ietf.org/rfc/rfc3987.txt.

ISO/IEC 10646
: ISO (International Organization for Standardization). *ISO/IEC 10646:2003. Information technology—Universal Multiple-Octet Coded Character Set (UCS)*, as, from time to time, amended, replaced by a new edition, or expanded by the addition of new parts. [Geneva]: International Organization for Standardization. (See http://www.iso.org for the latest version.)

Unicode
: The Unicode Consortium. *The Unicode Standard* Reading, Mass.: Addison-Wesley, 2003, as updated from time to time by the publication of new versions. See http://www.unicode.org/standard/versions/ for the latest version and additional information on versions of the standard and of the Unicode Character Database. The version of Unicode to be used is implementation-defined, but implementations are recommended to use the latest Unicode version.

XML 1.0
: World Wide Web Consortium. *Extensible Markup Language (XML) 1.0.* W3C Recommendation. See http://www.w3.org/TR/REC-xml/. The edition of XML 1.0 must be no earlier than the Third Edition; the edition used is implementation-defined, but we recommend that implementations use the latest version.

XML 1.1
: World Wide Web Consortium. *Extensible Markup Language (XML) 1.1.* W3C Recommendation. See http://www.w3.org/TR/xml11/

XML Base
: World Wide Web Consortium. *XML Base.* W3C Recommendation. See http://www.w3.org/TR/xmlbase/

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

XML Names 1.1
: World Wide Web Consortium. *Namespaces in XML 1.1.* W3C Recommendation. See http://www.w3.org/TR/xml-names11/

XML ID
: World Wide Web Consortium. *xml:id Version 1.0.* W3C Recommendation. See http://www.w3.org/TR/xml-id/

XML Schema
: World Wide Web Consortium. *XML Schema, Parts 0, 1, and 2 (Second Edition)*. W3C Recommendation, 28 October 2004. See http://www.w3.org/TR/xmlschema-0/, http://www.w3.org/TR/xmlschema-1/, and http://www.w3.org/TR/xmlschema-2/.

XQuery 1.0 and XPath 2.0 Data Model (Second Edition)
: World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. W3C Recommendation, 14 December 2010. See http://www.w3.org/TR/xpath-datamodel/.

XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)
: World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)*.

W3C Recommendation, 14 December 2010. See http://www.w3.org/TR/xquery-semantics/.

XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)
World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)* W3C Recommendation, 14 December 2010. See http://www.w3.org/TR/xpath-functions/.

XSLT 2.0 and XQuery 1.0 Serialization (Second Edition)
World Wide Web Consortium. *XSLT 2.0 and XQuery 1.0 Serialization (Second Edition)*. W3C Recommendation, 14 December 2010. See http://www.w3.org/TR/xslt-xquery-serialization/.

E.2 Non-normative References

XPath 2.0 Requirements
World Wide Web Consortium. *XPath Requirements Version 2.0*. W3C Working Draft 22 August 2003. See http://www.w3.org/TR/xpath20req/.

XQuery 1.0: An XML Query Language (Second Edition)
World Wide Web Consortium. *XQuery 1.0: An XML Query Language (Second Edition)*. W3C Recommendation, 14 December 2010. See http://www.w3.org/TR/xquery/.

XSL Transformations (XSLT) Version 2.0 (Second Edition)
World Wide Web Consortium. *XSL Transformations (XSLT) 2.0 (Second Edition)* W3C Recommendation, 14 December 2010. See http://www.w3.org/TR/xslt20/

Document Object Model
World Wide Web Consortium. *Document Object Model (DOM) Level 3 Core Specification.* W3C Recommendation, April 7, 2004. See http://www.w3.org/TR/DOM-Level-3-Core/.

XML Infoset
World Wide Web Consortium. *XML Information Set.* W3C Recommendation 24 October 2001. See http://www.w3.org/TR/xml-infoset/

XPath 1.0
World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, Nov. 16, 1999. See http://www.w3.org/TR/xpath/

XPointer
World Wide Web Consortium. *XML Pointer Language (XPointer).* W3C Last Call Working Draft 8 January 2001. See http://www.w3.org/TR/WD-xptr

E.3 Background Material

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

XSLT 1.0
World Wide Web Consortium. *XSL Transformations (XSLT) 1.0.* W3C Recommendation. See http://www.w3.org/TR/xslt

F Conformance

XPath is intended primarily as a component that can be used by other specifications. Therefore, XPath relies on specifications that use it (such as [XPointer] and [XSL Transformations (XSLT) Version 2.0 (Second Edition)]) to specify conformance criteria for XPath in their respective environments. Specifications that set conformance criteria for their use of XPath must not change the syntactic or semantic definitions of XPath as given in this specification, except by subsetting and/or compatible extensions.

The specification of such a language may describe it as an extension of XPath provided that every expression that conforms to the XPath grammar behaves as described in this specification.

F.1 Static Typing Feature

[Definition: The **Static Typing Feature** is an optional feature of XPath that provides support for the static semantics defined in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)], and requires implementations to detect and report type errors during the static analysis phase.] Specifications that use XPath may specify conformance criteria for use of the Static Typing Feature.

If an implementation does not support the Static Typing Feature, but can nevertheless determine during the static analysis phase that an expression will necessarily raise a type error if evaluated at run time, the implementation may raise that error during the static analysis phase. The choice of whether to raise such an error at analysis time is implementation dependent.

F.1.1 Static Typing Extensions

In some cases, the static typing rules defined in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)] are not very precise (see, for example, the type inference rules for the ancestor axes—parent, ancestor, and ancestor-or-self—and for the function `fn:root`). Some implementations may wish to support more precise static typing rules.

A conforming implementation that implements the Static Typing Feature may also provide one or more **static typing extensions**. [Definition: A **static typing extension** is an implementation-defined type inference rule that infers a more precise static type than that inferred by the type inference rules in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)].] See Section 6.1.1 Static Typing Extensions[FS] for a formal definition of the constraints on static typing extensions.

G Error Conditions

err:XPST0001

It is a static error if analysis of an expression relies on some component of the static context that has not been assigned a value.

err:XPDY0002

It is a dynamic error if evaluation of an expression relies on some part of the dynamic context that has not been assigned a value.

err:XPST0003

It is a static error if an expression is not a valid instance of the grammar defined in **A.1 EBNF**.

err:XPTY0004

It is a type error if, during the static analysis phase, an expression is found to have a static type that is not appropriate for the context in which the expression occurs, or during the dynamic evaluation phase, the dynamic type of a value does not match a required type as specified by the matching rules in **2.5.4 SequenceType Matching**.

err:XPST0005

During the analysis phase, it is a static error if the static type assigned to an expression other than the expression `()` or `data(())` is `empty-sequence()`.

err:XPTY0006

(Not currently used.)

err:XPTY0007

(Not currently used.)

err:XPST0008

It is a static error if an expression refers to an element name, attribute name, schema type name, namespace prefix, or variable name that is not defined in the static context, except for an ElementName in an ElementTest or an AttributeName in an AttributeTest.

err:XPST0010

An implementation must raise a static error if it encounters a reference to an axis that it does not support.

err:XPST0017

It is a static error if the expanded QName and number of arguments in a function call do not match the name and arity of a function signature in the static context.

err:XPTY0018

It is a type error if the result of the last step in a path expression contains both nodes and atomic values.

err:XPTY0019

It is a type error if the result of a step (other than the last step) in a path expression contains an atomic value.

err:XPTY0020

It is a type error if, in an axis step, the context item is not a node.

err:XPDY0021

(Not currently used.)

err:XPDY0050

It is a dynamic error if the dynamic type of the operand of a `treat` expression does not match the sequence type specified by the `treat` expression. This error might also be raised by a path expression beginning with "`/`" or "`//`" if the context node is not in a tree that is rooted at a document node. This is because a leading "`/`" or "`//`" in a path expression is an abbreviation for an initial step that includes the clause `treat as document-node()`.

err:XPST0051

It is a static error if a QName that is used as an AtomicType in a SequenceType is not defined in the in-scope schema types as an atomic type.

err:XPST0080

It is a static error if the target type of a `cast` or `castable` expression is `xs:NOTATION` or `xs:anyAtomicType`.

err:XPST0081

It is a static error if a QName used in an expression contains a namespace prefix that cannot be expanded into a namespace URI by using the statically known namespaces.

err:XPST0083

(Not currently used.)

H Glossary (Non-Normative)

Gregorian

In the operator mapping tables, the term **Gregorian** refers to the types `xs:gYearMonth`, `xs:gYear`, `xs:gMonthDay`, `xs:gDay`, and `xs:gMonth`.

QName

Lexically, a **QName** consists of an optional namespace prefix and a local name. If the namespace prefix is present, it is separated from the local name by a colon.

SequenceType matching

During evaluation of an expression, it is sometimes necessary to determine whether a value with a known dynamic type "matches" an expected sequence type. This process is known as **SequenceType matching**.

URI

Within this specification, the term **URI** refers to a Universal Resource Identifier as defined in [RFC3986] and extended in [RFC3987] with the new name **IRI**.

XDM instance

The term **XDM instance** is used, synonymously with the term **value**, to denote an unconstrained sequence of nodes and/or atomic values in the data model.

XPath 1.0 compatibility mode

XPath 1.0 compatibility mode. This value is `true` if rules for backward compatibility with XPath Version 1.0 are in effect; otherwise it is `false`.

atomic value

An **atomic value** is a value in the value space of an **atomic type**, as defined in [XML Schema].

atomization

Atomization of a sequence is defined as the result of invoking the `fn:data` function on the sequence, as defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

available collections

Available collections. This is a mapping of strings onto sequences of nodes. The string represents the absolute URI of a resource. The sequence of nodes represents the result of the `fn:collection` function when that URI is supplied as the argument.

available documents

Available documents. This is a mapping of strings onto document nodes. The string represents the absolute URI of a resource. The document node is the root of a tree that represents that resource using the data model. The document node is returned by the `fn:doc` function when applied to that URI.

axis step

An **axis step** returns a sequence of nodes that are reachable from the context node via a specified axis. Such a step has two parts: an **axis**, which defines the "direction of movement" for the step, and a node test, which selects nodes based on their kind, name, and/or type annotation.

base URI

Base URI. This is an absolute URI, used when necessary in the resolution of relative URIs (for example, by the `fn:resolve-uri` function.)

built-in function

The **built-in functions** supported by XPath are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

collation

A **collation** is a specification of the manner in which strings and URIs are compared and, by extension, ordered. For a more complete definition of collation, see [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

comma operator

One way to construct a sequence is by using the **comma operator**, which evaluates each of its operands and concatenates the resulting sequences, in order, into a single result sequence.

constructor function

The **constructor function** for a given type is used to convert instances of other atomic types into the given type. The semantics of the constructor function call `T($arg)` are defined to be equivalent to the expression `(($arg) cast as T?)`.

context item

The **context item** is the item currently being processed. An item is either an atomic value or a node.

context item static type

Context item static type. This component defines the static type of the context item within the scope of a given expression.

context node

When the context item is a node, it can also be referred to as the **context node**.

context position

The **context position** is the position of the context item within the sequence of items currently being processed.

context size

The **context size** is the number of items in the sequence of items currently being processed.

current dateTime

Current dateTime. This information represents an implementation-dependent point in time during the processing of an expression, and includes an explicit timezone. It can be retrieved by the `fn:current-dateTime` function. If invoked multiple times during the execution of an expression, this function always returns the same result.

data model

XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. This logical structure, known as the **data model**, is defined in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)].

data model schema

For a given node in an XDM instance, the **data model schema** is defined as the schema from which the type annotation of that node was derived.

default collation

Default collation. This identifies one of the collations in statically known collations as the collation to be used by functions and operators for comparing and ordering values of type `xs:string` and `xs:anyURI` (and types derived from them) when no explicit collation is specified.

default collection

Default collection. This is the sequence of nodes that would result from calling the `fn:collection` function with no arguments.

default element/type namespace

Default element/type namespace. This is a namespace URI or "none". The namespace URI, if present, is used for any unprefixed QName appearing in a position where an element or type name is expected.

default function namespace

Default function namespace. This is a namespace URI or "none". The namespace URI, if present, is used for any unprefixed QName appearing in a position where a function name is expected.

delimiting terminal symbol

The **delimiting terminal symbols** are: "!=", StringLiteral, "$", "(", ")", "*", "+", (comma), "-", (dot), "..", "/", "//", (colon), "::", "<", "<<", "<=", "=", ">", ">=", ">>", "?", "@", "[", "]", "|"

document order

Informally, **document order** is the order in which nodes appear in the XML serialization of a document.

dynamic context

The **dynamic context** of an expression is defined as information that is available at the time the expression is evaluated.

dynamic error

A **dynamic error** is an error that must be detected during the dynamic evaluation phase and may be detected during the static analysis phase. Numeric overflow is an example of a dynamic error.

dynamic evaluation phase

The **dynamic evaluation phase** is the phase during which the value of an expression is computed.

dynamic type

A **dynamic type** is associated with each value as it is computed. The dynamic type of a value may be more specific than the static type of the expression that computed it (for example, the static type of an expression might be `xs:integer*`, denoting a sequence of zero or more integers, but at evaluation time its value may have the dynamic type `xs:integer`, denoting exactly one integer.)

effective boolean value

The **effective boolean value** of a value is defined as the result of applying the `fn:boolean` function to the value, as defined in [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

empty sequence

A sequence containing zero items is called an **empty sequence**.

error value

In addition to its identifying QName, a dynamic error may also carry a descriptive string and one or more additional values called **error values**.

expanded QName

An **expanded QName** consists of an optional namespace URI and a local name. An expanded QName also retains its original namespace prefix (if any), to facilitate casting the expanded QName into a string.

expression context

The **expression context** for a given expression consists of all the information that can affect the result of the expression.

filter expression

A **filter expression** consists simply of a **primary expression** followed by zero or more predicates. The result of the filter expression consists of the items returned by the primary expression, filtered by applying each predicate in turn, working from left to right.

focus

The first three components of the dynamic context (context item, context position, and context size) are called the **focus** of the expression.

function implementation

Function implementations. Each function in function signatures has a function implementation that enables the function to map instances of its parameter types into an instance of its result type.

function signature

Function signatures. This component defines the set of functions that are available to be called from within an expression. Each function is uniquely identified by its expanded QName and its arity (number of parameters).

ignorable whitespace

Ignorable whitespace consists of any whitespace characters that may occur between terminals, unless these characters occur in the context of a production marked with a ws:explicit annotation, in which case they can occur only where explicitly specified (see **A.2.4.2 Explicit Whitespace Handling**).

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.

implementation defined

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

implicit timezone

Implicit timezone. This is the timezone to be used when a date, time, or dateTime value that does not have a timezone is used in a comparison or arithmetic operation. The implicit timezone is an implementation-defined value of type `xs:dayTimeDuration`. See [XML Schema] for the range of legal values of a timezone.

in-scope attribute declarations

In-scope attribute declarations. Each attribute declaration is identified either by an expanded QName (for a top-level attribute declaration) or by an implementation-dependent attribute identifier (for a local attribute declaration).

in-scope element declarations

In-scope element declarations. Each element declaration is identified either by an expanded QName (for a top-level element declaration) or by an implementation-dependent element identifier (for a local element declaration).

in-scope namespaces

The **in-scope namespaces** property of an element node is a set of **namespace bindings**, each of which associates a namespace prefix with a URI, thus defining the set of namespace prefixes that are available for interpreting QNames within the scope of the element. For a given element, one namespace binding may have an empty prefix; the URI of this namespace binding is the default namespace within the scope of the element.

in-scope schema definitions

In-scope schema definitions. This is a generic term for all the element declarations, attribute declarations, and schema type definitions that are in scope during processing of an expression.

in-scope schema type

In-scope schema types. Each schema type definition is identified either by an expanded QName (for a **named type**) or by an implementation-dependent type identifier (for an **anonymous type**). The in-scope schema types include the predefined schema types described in **2.5.1 Predefined Schema Types**.

in-scope variables

In-scope variables. This is a set of (expanded QName, type) pairs. It defines the set of variables that are available for reference within an expression. The expanded QName is the name of the variable, and the type is the static type of the variable.

item

An **item** is either an atomic value or a node.

kind test

An alternative form of a node test called a **kind test** can select nodes based on their kind, name, and type annotation.

literal

A **literal** is a direct syntactic representation of an atomic value.

name test

A node test that consists only of a QName or a Wildcard is called a **name test**.

node

A **node** is an instance of one of the **node kinds** defined in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)].

node test

A **node test** is a condition that must be true for each node selected by a step.

non-delimiting terminal symbol

The **non-delimiting terminal symbols** are: IntegerLiteral, NCName, DecimalLiteral, DoubleLiteral, QName, "ancestor", "ancestor-or-self", "and", "as", "attribute", "cast", "castable", "child", "comment", "descendant", "descendant-or-self", "div", "document-node", "element", "else", "empty-sequence", "eq", "every", "except", "external", "following", "following-sibling", "for", "ge", "gt", "idiv", "if", "in", "instance", "intersect", "is", "item", "le", "lt", "mod", "namespace", "ne", "node", "of", "or", "parent", "preceding", "preceding-sibling", "processing-instruction", "return", "satisfies", "schema-attribute", "schema-element", "self", "some", "text", "then", "to", "treat", "union"

numeric

When referring to a type, the term **numeric** denotes the types `xs:integer`, `xs:decimal`, `xs:float`, and `xs:double`.

numeric predicate

A predicate whose predicate expression returns a numeric type is called a **numeric predicate**.

operator function

For each operator and valid combination of operand types, the operator mapping tables specify a result type and an **operator function** that implements the semantics of the operator for the given types.

path expression

A **path expression** can be used to locate nodes within trees. A path expression consists of a series of one or more steps, separated by "`/`" or "`//`", and optionally beginning with "`/`" or "`//`".

predicate

A **predicate** consists of an expression, called a **predicate expression**, enclosed in square brackets. A predicate serves to filter a sequence, retaining some items and discarding others.

primary expression

Primary expressions are the basic primitives of the language. They include literals, variable references, context item expressions, and function calls. A primary expression may also be created by enclosing any expression in parentheses, which is sometimes helpful in controlling the precedence of operators.

principal node kind

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

reverse document order

The node ordering that is the reverse of document order is called **reverse document order**.

schema type

A **schema type** is a type that is (or could be) defined using the facilities of [XML Schema] (including the built-in types of [XML Schema]).

sequence

A **sequence** is an ordered collection of zero or more items.

sequence type

A **sequence type** is a type that can be expressed using the SequenceType syntax. Sequence types are used whenever it is necessary to refer to a type in an XPath expression. The term **sequence type** suggests that this syntax is used to describe the type of an XPath value, which is always a sequence.

serialization

Serialization is the process of converting an XDM instance into a sequence of octets (step DM4 in Figure 1.)

singleton

A sequence containing exactly one item is called a **singleton**.

stable

Document order is **stable**, which means that the relative order of two nodes will not change during the processing of a given expression, even if this order is implementation-dependent.

static analysis phase

The **static analysis phase** depends on the expression itself and on the static context. The **static analysis phase** does not depend on input data (other than schemas).

static context

The **static context** of an expression is the information that is available during static analysis of the expression, prior to its evaluation.

static error

A **static error** is an error that must be detected during the static analysis phase. A syntax error is an example of a static error.

static type

The **static type** of an expression is a type such that, when the expression is evaluated, the resulting value will always conform to the static type.

static typing extension

A **static typing extension** is an implementation-defined type inference rule that infers a more precise static type than that inferred by the type inference rules in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)].

static typing feature

The **Static Typing Feature** is an optional feature of XPath that provides support for the static semantics defined in [XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)], and requires implementations to detect and report type errors during the static analysis phase.

statically known collections

Statically known collections. This is a mapping from strings onto types. The string represents the absolute URI of a resource that is potentially available using the `fn:collection` function. The type is the type of the sequence of nodes that would result from calling the `fn:collection` function with this URI as its argument.

statically known documents

Statically known documents. This is a mapping from strings onto types. The string represents the absolute URI of a resource that is potentially available using the `fn:doc` function. The type is the static type of a call to `fn:doc` with the given URI as its literal argument.

statically known collations

Statically known collations. This is an implementation-defined set of (URI, collation) pairs. It defines the names of the collations that are available for use in processing expressions.

statically known default collection type

Statically known default collection type. This is the type of the sequence of nodes that would result from calling the `fn:collection` function with no arguments.

statically known namespaces

Statically known namespaces. This is a set of (prefix, URI) pairs that define all the namespaces that are known during static processing of a given expression.

step

A **step** is a part of a path expression that generates a sequence of items and then filters the sequence by zero or more predicates. The value of the step consists of those items that satisfy the predicates, working from left to right. A step may be either an axis step or a filter expression.

string value

The **string value** of a node is a string and can be extracted by applying the `fn:string` function to the node.

substitution group

Substitution groups are defined in [XML Schema] Part 1, Section 2.2.2.2. Informally, the substitution group headed by a given element (called the **head element**) consists of the set of elements that can be substituted for the head element without affecting the outcome of schema validation.

subtype substitution

The use of a value whose dynamic type is derived from an expected type is known as **subtype substitution**.

symbol

Each rule in the grammar defines one **symbol**, using the following format:

```
symbol ::= expression
```

symbol separators

Whitespace and Comments function as **symbol separators**. For the most part, they are not mentioned in the grammar, and may occur between any two terminal symbols mentioned in the grammar, except where that is forbidden by the /* ws: explicit */ annotation in the EBNF, or by the /* xgs: xml-version */ annotation.

terminal

A **terminal** is a symbol or string or pattern that can appear in the right-hand side of a rule, but never appears on the left hand side in the main grammar, although it may appear on the left-hand side of a rule in the grammar for terminals.

type annotation

Each element node and attribute node in an XDM instance has a **type annotation** (referred to in [XQuery 1.0 and XPath 2.0 Data Model (Second Edition)] as its `type-name` property.) The type annotation of a node is a schema type that describes the relationship between the string value of the node and its typed value.

type error

A **type error** may be raised during the static analysis phase or the dynamic evaluation phase. During the static analysis phase, a type error occurs when the static type of an expression does not match the expected type of the context in which the expression occurs. During the dynamic evaluation phase, a type error occurs when the dynamic type of a value does not match the expected type of the context in which the value occurs.

type promotion

Under certain circumstances, an atomic value can be promoted from one type to another. **Type promotion** is used in evaluating function calls (see **3.1.5 Function Calls**) and operators that accept numeric or string operands (see **B.2 Operator Mapping**).

typed value

The **typed value** of a node is a sequence of atomic values and can be extracted by applying the `fn:data` function to the node.

undefined

In certain situations a value is said to be **undefined** (for example, the value of the context item, or the typed value of an element node). This term indicates that the property in question has no value and that any attempt to use its value results in an error.

value

In the data model, a **value** is always a sequence.

variable reference

A **variable reference** is a QName preceded by a $-sign.

variable values

Variable values. This is a set of (expanded QName, value) pairs. It contains the same expanded QNames as the in-scope variables in the static context for the expression. The expanded QName is the name of the variable and the value is the dynamic value of the variable, which includes its dynamic type.

warning

In addition to static errors, dynamic errors, and type errors, an XPath implementation may raise **warnings**, either during the static analysis phase or the dynamic evaluation phase. The circumstances in which warnings are raised, and the ways in which warnings are handled, are implementation-defined.

whitespace

A **whitespace** character is any of the characters defined by [http://www.w3.org/TR/REC-xml/#NT-S].

xs:anyAtomicType

`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.

xs:dayTimeDuration

`xs:dayTimeDuration` is derived by restriction from `xs:duration`. The lexical representation of `xs:dayTimeDuration` is restricted to contain only day, hour, minute, and second components.

xs:untyped

`xs:untyped` is used as the type annotation of an element node that has not been validated, or has been validated in `skip` mode.

xs:untypedAtomic

`xs:untypedAtomic` is an atomic type that is used to denote untyped atomic data, such as text that has not been assigned a more specific type.

xs:yearMonthDuration

`xs:yearMonthDuration` is derived by restriction from `xs:duration`. The lexical representation of `xs:yearMonthDuration` is restricted to contain only year and month components.

I Backwards Compatibility with XPath 1.0 (Non-Normative)

This appendix provides a summary of the areas of incompatibility between XPath 2.0 and [XPath 1.0].

Three separate cases are considered:

1. Incompatibilities that exist when source documents have no schema, and when running with XPath 1.0 compatibility mode set to true. This specification has been designed to reduce the number of incompatibilities in this situation to an absolute minimum, but some differences remain and are listed individually.
2. Incompatibilities that arise when XPath 1.0 compatibility mode is set to false. In this case, the number of expressions where compatibility is lost is rather greater.
3. Incompatibilities that arise when the source document is processed using a schema (whether or not XPath 1.0 compatibility mode is set to true). Processing the document with a schema changes the way that the values of nodes are interpreted, and this can cause an XPath expression to return different results.

I.1 Incompatibilities when Compatibility Mode is true

The list below contains all known areas, within the scope of this specification, where an XPath 2.0 processor running with compatibility mode set to true will produce different results from an XPath 1.0 processor evaluating the same expression, assuming that the expression was valid in XPath 1.0, and that the nodes in the source document have no type annotations other than `xs:untyped` and `xs:untypedAtomic`.

Incompatibilities in the behavior of individual functions are not listed here, but are included in an appendix of [XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)].

Since both XPath 1.0 and XPath 2.0 leave some aspects of the specification implementation-defined, there may be incompatiblities in the behavior of a particular implementation that are outside the scope of this specification. Equally, some aspects of the behavior of XPath are defined by the host language.

1. Consecutive comparison operators such as `A < B < C` were supported in XPath 1.0, but are not permitted by the XPath 2.0 grammar. In most cases such comparisons in XPath 1.0 did not have the intuitive meaning, so it is unlikely that they have been widely used in practice. If such a construct is found, an XPath 2.0 processor will report a syntax error, and the construct can be rewritten as `(A < B) < C`
2. When converting strings to numbers (either explicitly when using the `number` function, or implicitly say on a function call), certain strings that converted to the special value `NaN` under XPath 1.0 will convert to values other than `NaN` under XPath 2.0. These include any number written with a leading `+` sign, any number in exponential floating point notation (for example `1.0e+9`), and the strings `INF` and `-INF`.

   Furthermore, the strings `Infinity` and `-Infinity`, which were accepted by XPath 1.0 as representations of the floating-point values positive and negative infinity, are no longer recognized. They are converted to `NaN` when running under XPath 2.0 with compatibility mode set to true, and cause a dynamic error when compatibility mode is set to false.
3. XPath 2.0 does not allow a token starting with a letter to follow immediately after a numeric literal, without intervening whitespace. For example, `10div 3` was permitted in XPath 1.0, but in XPath 2.0 must be written as `10 div 3`.
4. The namespace axis is deprecated in XPath 2.0. Implementations may support the namespace axis for backward compatibility with XPath 1.0, but they are not required to do so. (XSLT 2.0 requires that if XPath backwards compatibility mode is supported, then the namespace axis must also be supported; but other host languages may define the conformance rules differently.)
5. If one operand in a general comparison is a single atomic value of type `xs:boolean`, the other operand is converted to `xs:boolean` when XPath 1.0 compatibility mode is set to true. In XPath 1.0, if neither operand of a comparison operation using the <, <=, > or >= operator was a node set, both operands were converted to numbers. The result of the expression `true() > number('0.5')` is therefore true in XPath 1.0, but is false in XPath 2.0 even when compatibility mode is set to true.
6. In XPath 2.0, a type error is raised if, for a PITarget specified in a SequenceType of form `processing-instruction(N)`, *fn:normalize-space(N)* is not in the lexical space of NCName. In XPath 1.0, this condition was not treated as an error.
7. In XPath 1.0, the expression `-x|y` parsed as `-(x|y)`, and returned the negation of the numeric value of the first node in the union of `x` and `y`. In XPath 2.0, this expression parses as `(-x)|y`. When XPath 1.0 Compatibility Mode is true, this will always cause a type error.
8. The rules for converting numbers to strings have changed. These may affect the way numbers are displayed in the output of a stylesheet. For numbers whose absolute value is in the range `1E-6` to `1E+6`, the result should be the same, but outside this range, scientific format is used for non-integral `xs:float` and `xs:double` values.

I.2 Incompatibilities when Compatibility Mode is false

Even when the setting of the XPath 1.0 compatibility mode is false, many XPath expressions will still produce the same results under XPath 2.0 as under XPath 1.0. The exceptions are described in this section.

In all cases it is assumed that the expression in question was valid under XPath 1.0, that XPath 1.0 compatibility mode is false, and that all elements and attributes are annotated with the types `xs:untyped` and `xs:untypedAtomic` respectively.

In the description below, the terms *node-set* and *number* are used with their XPath 1.0 meanings, that is, to describe expressions which according to the rules of XPath 1.0 would have generated a node-set or a number respectively.

1. When a node-set containing more than one node is supplied as an argument to a function or operator that expects a single node or value, the XPath 1.0 rule was that all nodes after the first were discarded. Under XPath 2.0, a type error occurs if there is more than one node. The XPath 1.0 behavior can always be restored by using the predicate `[1]` to explicitly select the first node in the node-set.
2. In XPath 1.0, the `<` and `>` operators, when applied to two strings, attempted to convert both the strings to numbers and then made a numeric comparison between the results. In XPath 2.0, these operators perform a string comparison using the default collating sequence. (If either value is numeric, however, the results are compatible with XPath 1.0)
3. When an empty node-set is supplied as an argument to a function or operator that expects a number, the value is no longer converted implicitly to NaN. The XPath 1.0 behavior can always be restored by using the `number` function to perform an explicit conversion.
4. More generally, the supplied arguments to a function or operator are no longer implicitly converted to the required type, except in the case where the supplied argument is of type `xs:untypedAtomic` (which will commonly be the case when a node in a schemaless document is supplied as the argument). For example, the function call `substring-before(10 div 3, ".")` raises a type error under XPath 2.0, because the arguments to the `substring-before` function must be strings rather than numbers. The XPath 1.0 behavior can be restored by performing an explicit conversion to the required type using a constructor function or cast.
5. The rules for comparing a node-set to a boolean have changed. In XPath 1.0, an expression such as `$node-set = true()` was evaluated by converting the node-set to a boolean and then performing a boolean comparison: so this expression would return `true` if `$node-set` was non-empty. In XPath 2.0, this expression is handled in the same way as other comparisons between a sequence and a singleton: it is `true` if `$node-set` contains at least one node whose value, after atomization and conversion to a boolean using the casting rules, is `true`.

   This means that if `$node-set` is empty, the result under XPath 2.0 will be `false` regardless of the value of the boolean operand, and regardless of which operator is used. If `$node-set` is non-empty, then in most cases the comparison with a boolean is likely to fail, giving a dynamic error. But if a node has the value "0", "1", "true", or "false", evaluation of the expression may succeed.
6. Comparisons of a number to a boolean, a number to a string, or a string to a boolean are not allowed in XPath 2.0: they result in a type error. In XPath 1.0 such comparisons were allowed, and were handled by converting one of the operands to the type of the other. So for example in XPath 1.0 `4 = true()` was true; `4 = "+4"` was false (because the string `+4` converts to `NaN`), and `false = "false"` was false (because the string `"false"` converts to the boolean `true`). In XPath 2.0 all these comparisons are type errors.
7. Additional numeric types have been introduced, with the effect that arithmetic may now be done as an integer, decimal, or single- or double-precision floating point calculation where previously it was always performed as double-precision floating point. The result of the `div` operator when dividing two integers is now a value of type decimal rather than double. The expression `10 div 0` raises an error rather than returning positive infinity.
8. The rules for converting strings to numbers have changed. The implicit conversion that occurs when passing an `xs:untypedAtomic` value as an argument to a function that expects a number no longer converts unrecognized strings to the value `NaN`; instead, it reports a dynamic error. This is in addition to the differences that apply when backwards compatibility mode is set to true.
9. Many operations in XPath 2.0 produce an empty sequence as their result when one of the arguments or operands is an empty sequence. Where the operation expects a string, an empty sequence is usually considered equivalent to a zero-length string, which is compatible with the XPath 1.0 behavior. Where the operation expects a number, however, the result is not the same. For example, if `@width` returns an empty sequence, then in XPath 1.0 the result of `@width+1` was `NaN`, while with XPath 2.0 it is `()`. This has the effect that a filter expression such as `item[@width+1 != 2]` will select items having no `width` attribute under XPath 1.0, and will not select them under XPath 2.0.
10. The typed value of a comment node, processing instruction node, or namespace node under XPath 2.0 is of type `xs:string`, not `xs:untypedAtomic`. This means that no implicit conversions are applied if the value is used in a context where a number is expected. If a processing-instruction node is used as an operand of an arithmetic operator, for example, XPath 1.0 would attempt to convert the string value of the node to a number (and deliver `NaN` if unsuccessful), while XPath 2.0 will report a type error.

11. In XPath 1.0, it was defined that with an expression of the form `A and B`, B would not be evaluated if A was false. Similarly in the case of `A or B`, B would not be evaluated if A was true. This is no longer guaranteed with XPath 2.0: the implementation is free to evaluate the two operands in either order or in parallel. This change has been made to give more scope for optimization in situations where XPath expressions are evaluated against large data collections supported by indexes. Implementations may choose to retain backwards compatibility in this area, but they are not obliged to do so.
12. In XPath 1.0, the expression `-x|y` parsed as `-(x|y)`, and returned the negation of the numeric value of the first node in the union of `x` and `y`. In XPath 2.0, this expression parses as `(-x)|y`. When XPath 1.0 Compatibility Mode is false, this will cause a type error, except in the situation where `x` evaluates to an empty sequence. In that situation, XPath 2.0 will return the value of `y`, whereas XPath 1.0 returned the negation of the numeric value of `y`.

I.3 Incompatibilities when using a Schema

An XPath expression applied to a document that has been processed against a schema will not always give the same results as the same expression applied to the same document in the absence of a schema. Since schema processing had no effect on the result of an XPath 1.0 expression, this may give rise to further incompatibilities. This section gives a few examples of the differences that can arise.

Suppose that the context node is an element node derived from the following markup: `<background color="red green blue"/>`. In XPath 1.0, the predicate `[@color="blue"]` would return `false`. In XPath 2.0, if the `color` attribute is defined in a schema to be of type `xs:NMTOKENS`, the same predicate will return `true`.

Similarly, consider the expression `@birth < @death` applied to the element `<person birth="1901-06-06" death="1991-05-09"/>`. With XPath 1.0, this expression would return false, because both attributes are converted to numbers, which returns `NaN` in each case. With XPath 2.0, in the presence of a schema that annotates these attributes as dates, the expression returns `true`.

Once schema validation is applied, elements and attributes cannot be used as operands and arguments of expressions that expect a different data type. For example, it is no longer possible to apply the `substring` function to a date to extract the year component, or to a number to extract the integer part. Similarly, if an attribute is annotated as a boolean then it is not possible to compare it with the strings `"true"` or `"false"`. All such operations lead to type errors. The remedy when such errors occur is to introduce an explicit conversion, or to do the computation in a different way. For example, `substring-after(@temperature, "-")` might be rewritten as `abs(@temperature)`.

In the case of an XPath 2.0 implementation that provides the static typing feature, many further type errors will be reported in respect of expressions that worked under XPath 1.0. For example, an expression such as `round(../@price)` might lead to a static type error because the processor cannot infer statically that `../@price` is guaranteed to be numeric.

Schema validation will in many cases perform whitespace normalization on the contents of elements (depending on their type). This will change the result of operations such as the `string-length` function.

Schema validation augments the data model by adding default values for omitted attributes and empty elements.

J Changes since the First Edition (Non-Normative)

This version of the XPath specification was created by applying the errata from Errata for XML Path Language (XPath) 2.0 to the XPath 2.0 Recommendation. No other substantive changes have been made.

Erratum	Bugzilla	Category	Description
XP.E1	4298	editorial	Spelling mistake: minimum
XP.E2	4855	editorial	Some incompatibilities from XPath 1.0 are undocumented; others are wrongly classified as applying only when compatibility mode is false.
XP.E3	4868	editorial	For valid syntax, parentheses need to be added to the expansion for leading "/" and leading "//" in a path expression.

XP.E4	4446	substantive	This erratum adds more details to the rules defining permissible expression rewrites for optimization and other purposes.
XP.E5	4873	substantive	This erratum clarifies the conditions under which a castable expression may raise an error.
XP.E6	5445	editorial	Undocumented incompatibility when the operators <, >, <=, or >= are used to compare a number to a boolean.
XP.E7	5351	substantive	Specifies that an error results if the PITarget specified in a SequenceType of form processing-instruction(PITarget) is not a syntactically valid NCName.
XP.E8	5261	editorial	Removes references to error code FORG0001 from description of cast expression. Replaces them with a reference to Functions and Operators for normative description of error behavior.
XP.E9	5471	editorial	Deletes unnecessary reference to RFC2396 from Normative References. This item is never referenced in the normative text.
XP.E10	5223	substantive	Specifies that general comparisons cast an untyped operand to the primitive base type of the other operand rather than to the most specific type of the other operand.
XP.E11	5984	editorial	Corrects a list of examples of primitive atomic types.
XP.E13	5347	substantive	Allows (and encourages) the use of XML 1.0 editions newer than the Third Edition.
XP.E14	6027	substantive	Specifies conformance criteria for syntax extensions.
XP.E15	6287	editorial	Defines the meaning of "undefined" for Data Model properties.
XP.E16	5727	substantive	Clarifications on parsing leading / in XPath expressions.
XP.E18	5876	substantive	Corrects the description of precedence with respect to parentheses and square brackets.
XP.E19	5351	substantive	Specifies that leading and trailing whitespace are stripped from a PITarget specified in a SequenceType of form processing-instruction(PITarget) before it is tested to see if it is a syntactically valid NCName. Also makes the description of the error introduced in E12 more precise. If accepted, this supersedes E12.