XQuery 1.0: An XML Query Language

W3C Recommendation 23 January 2007

This version:

http://www.w3.org/TR/2007/REC-xquery-20070123/

Latest version:

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

Previous version:

http://www.w3.org/TR/2006/PR-xquery-20061121/

Authors and Contributors:

Scott Boag (XSL WG) (IBM Research) <scott_boag@us.ibm.com>
Don Chamberlin (XML Query WG) (IBM Almaden Research Center)
Mary F. Fernández (XML Query WG) (AT&T Labs) <mff@research.att.com>
Daniela Florescu (XML Query WG) (Oracle) <dana.florescu@oracle.com>
Jonathan Robie (XML Query WG) (DataDirect Technologies)
Jérôme Siméon (XML Query WG) (IBM T.J. Watson Research Center) <simeon@us.ibm.com>

Abstract

XML is a versatile markup language, capable of labeling the information content of diverse data sources including structured and semi-structured documents, relational databases, and object repositories. A query language that uses the structure of XML intelligently can express queries across all these kinds of data, whether physically stored in XML or viewed as XML via middleware. This specification describes a query language called XQuery, which is designed to be broadly applicable across many types of XML data sources.

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 have progressed to Recommendation together (XQuery 1.0, XQueryX 1.0, XSLT 2.0, Data Model, Functions and Operators, Formal Semantics, Serialization, XPath 2.0).

This is a Recommendation of the W3C. It has been developed by the W3C XML Query Working Group, 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 minor changes made against the Proposed Recommendation of 21 November 2006; please see the public disposition of comments for details. Changes to this document since the Proposed Recommendation are detailed in the Appendix J – Revision Log on page 149.

Please report errors in 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 "[XQuery]" 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 a group 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 group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

Appendices

This page is intentionally left blank.

1. Introduction

As increasing amounts of information are stored, exchanged, and presented using XML, the ability to intelligently query XML data sources becomes increasingly important. One of the great strengths of XML is its flexibility in representing many different kinds of information from diverse sources. To exploit this flexibility, an XML query language must provide features for retrieving and interpreting information from these diverse sources.

XQuery is designed to meet the requirements identified by the W3C XML Query Working Group [XML Query 1.0 Requirements] and the use cases in [XML Query Use Cases]. It is designed to be a language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents. The Query Working Group has identified a requirement for both a non-XML query syntax and an XML-based query syntax. XQuery is designed to meet the first of these requirements. XQuery is derived from an XML query language called Quilt [Quilt], which in turn borrowed features from several other languages, including XPath 1.0 [XPath 1.0], XQL [XQL], XML-QL [XML-QL], SQL [SQL], and OQL [ODMG].

XQuery 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/XPath Data Model (XDM)].

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.

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

- [XQuery/XPath Data Model (XDM)] defines the data model that underlies all XQuery expressions.
- [XQuery 1.0 and XPath 2.0 Formal Semantics] defines the static semantics of XQuery 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 XQuery is based on [XML Schema].
- The built-in function library and the operators supported by XQuery are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators].
- One requirement in [XML Query 1.0 Requirements] is that an XML query language have both a human-readable syntax and an XML-based syntax. The XML-based syntax for XQuery is described in [XQueryX 1.0].

This document specifies a grammar for XQuery, using the same basic EBNF notation used in [XML 1.0]. Unless otherwise noted (see Appendix A.2 – Lexical structure on page 118), whitespace is not significant in queries. Grammar productions are introduced together with the features that they describe, and a complete grammar is also presented in the appendix [Appendix A – XQuery Grammar on page 109]. 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:

```
[1]    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*.

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

This document normatively defines the dynamic semantics of XQuery. The static semantics of XQuery are normatively defined in [XQuery 1.0 and XPath 2.0 Formal Semantics]. 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 XQuery 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. XQuery allows expressions to be nested with full generality. (However, unlike a pure functional language, it does not allow variable substitution if the variable declaration contains construction of new nodes.)

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

Like XML, XQuery is a case-sensitive language. Keywords in XQuery use lower-case characters and are not reserved—that is, names in XQuery expressions are allowed to be the same as language keywords, except for certain unprefixed function-names listed in Appendix A.3 – Reserved Function Names on page 122.

In the data model, a *value* is always a sequence. A *sequence* is an ordered collection of zero or more items. An *item* is either an atomic value or a node. An *atomic value* is a value in the value space of an *atomic type*, as defined in [XML Schema]. A *node* is an instance of one of the *node kinds* defined in [XQuery/XPath Data Model (XDM)]. 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`.

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

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 XQuery are called *QNames*, and conform to the syntax in [XML Names]. 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 . 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.

Certain namespace prefixes are predeclared by XQuery and bound to fixed namespace URIs. These namespace prefixes are as follows:

- `xml = http://www.w3.org/XML/1998/namespace`
- `xs = http://www.w3.org/2001/XMLSchema`
- `xsi = http://www.w3.org/2001/XMLSchema-instance`
- `fn = http://www.w3.org/2005/xpath-functions`
- `local = http://www.w3.org/2005/xquery-local-functions` (see § 4.15 – Function Declaration on page 103.)

In addition to the prefixes in the above list, this document uses the prefix `err` to represent the namespace URI `http://www.w3.org/2005/xqt-errors` (see § 2.3.2 – Identifying and Reporting Errors on page 14). This namespace prefix is not predeclared and its use in this document is not normative.

Element nodes have a property called *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 [XPath 1.0], the in-scope namespaces of an element node are represented by a collection of *namespace nodes* arranged on a *namespace axis*, which is optional and deprecated in [XPath 2.0]. XQuery does not support the namespace axis and does not represent namespace bindings in the form of nodes. However, where other specifications such as [XSLT 2.0 and XQuery 1.0 Serialization] refer to namespace nodes, these nodes may be synthesized from the in-scope namespaces of an element node by interpreting each namespace binding as a namespace node.

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

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

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 .

The individual components of the static context are summarized below. Rules governing the scope and initialization of these components can be found in Appendix C.1 – Static Context Components on page 128.

- *XPath 1.0 compatibility mode.* This component must be set by all host languages that include XPath 2.0 as a subset, indicating whether rules for compatibility with XPath 1.0 are in effect. XQuery sets the value of this component to `false`.
- *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.

  Some namespaces are predefined; additional namespaces can be added to the statically known namespaces by namespace declarations in a Prolog and by namespace declaration attributes in direct element constructors.
- *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].
- *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].
- *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:
  - *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 on page 21. If the Schema Import Feature is supported, in-scope schema types also include all type definitions found in imported schemas.
  - *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). If the Schema Import Feature is supported, in-scope element declarations include all element declarations found in imported schemas. An element declaration includes information about the element's substitution group affiliation.

    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.
  - *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). If the Schema Import Feature is supported, in-scope attribute declarations include all attribute declarations found in imported schemas.

- *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.

  Variable declarations in a Prolog are added to in-scope variables. An expression that binds a variable (such as a `let`, `for`, `some`, or `every` expression) extends the in-scope variables of its subexpressions with the new bound variable and its type. Within a *function declaration*, the in-scope variables are extended by the names and types of the *function parameters*.

  The static type of a variable may be either declared in a query or (if the Static Typing Feature is enabled) inferred by static type inference rules as described in [XQuery 1.0 and XPath 2.0 Formal Semantics].

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

- *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 § 3.12.5 – Constructor Functions on page 88.

- *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 queries and expressions. 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].

- *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.

- *Construction mode.* The construction mode governs the behavior of element and document node constructors. If construction mode is `preserve`, the type of a constructed element node is `xs:anyType`, and all attribute and element nodes copied during node construction retain their original types. If construction mode is `strip`, the type of a constructed element node is `xs:untyped`; all element nodes copied during node construction receive the type `xs:untyped`, and all attribute nodes copied during node construction receive the type `xs:untypedAtomic`.

- *Ordering mode.* Ordering mode, which has the value `ordered` or `unordered`, affects the ordering of the result sequence returned by certain path expressions, `union`, `intersect`, and `except` expressions, and FLWOR expressions that have no `order by` clause. Details are provided in the descriptions of these expressions.

- *Default order for empty sequences.* This component controls the processing of empty sequences and `NaN` values as ordering keys in an `order by` clause in a FLWOR expression, as described in § 3.8.3 – Order By and Return Clauses on page 76. Its value may be `greatest` or `least`.

- *Boundary-space policy.* This component controls the processing of boundary whitespace by direct element constructors, as described in § 3.7.1.4 – Boundary Whitespace on page 62. Its value may be `preserve` or `strip`.

- *Copy-namespaces mode.* This component controls the namespace bindings that are assigned when an existing element node is copied by an element constructor, as described in § 3.7.1 – Direct Element

Constructors on page 55. Its value consists of two parts: `preserve` or `no-preserve`, and `inherit` or `no-inherit`.

- *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].
- *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()?`.

  
  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`.

- *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()*`.

  
  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`.

- *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

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 .

The individual components of the dynamic context are summarized below. Further rules governing the semantics of these components can be found in Appendix C.2 – Dynamic Context Components on page 130.

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

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.

- The *context item* is the item currently being processed. An item is either an atomic value or a node.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`.
- 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.
- 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`.
- *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.
- *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. For a user-defined function, the function implementation is an XQuery expression. For a built-in function or external function, the function implementation is implementation-dependent.
- *Current dateTime.* This information represents an implementation-dependent point in time during the processing of a query, and includes an explicit timezone. It can be retrieved by the `fn:current-dateTime` function. If invoked multiple times during the execution of a query, this function always returns the same result.
- *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.
- *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.

  

  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.

- *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`."

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.

- *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

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

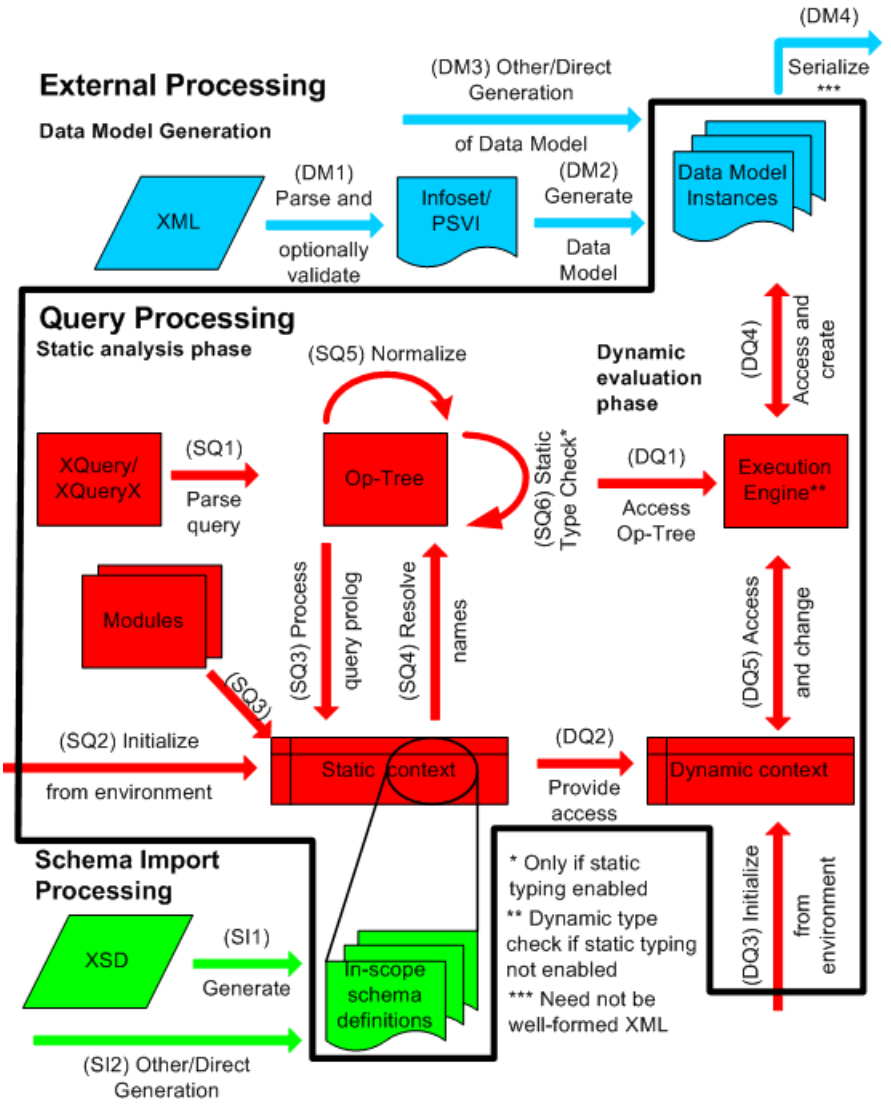

Figure 1: Processing Model Overview

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 XQuery; 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 on page 9), schema import processing (see § 2.2.2 – Schema Import Processing on page 10) and serialization (see § 2.2.4 – Serialization on page 12). The area inside the boundaries of the language is known as the *query processing domain*, which includes the static analysis and dynamic evaluation phases (see § 2.2.3 – Expression Processing on page 10). Consistency constraints on the query processing domain are defined in § 2.2.5 – Consistency Constraints on page 12.

2.2.1. Data Model Generation

Before a query can be processed, its input data must be represented as an XDM instance. This process occurs outside the domain of XQuery, 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/XPath Data Model (XDM)]. (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.) XQuery is defined in terms of the data model, but it does not place any constraints on how XDM instances are constructed.

Each element node and attribute node in an XDM instance has a *type annotation* (referred to in [XQuery/XPath Data Model (XDM)] 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 , the type annotations of the element and attribute nodes are derived from schema validation. XQuery 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 may be extracted from actual XML schemas as described in [XQuery 1.0 and XPath 2.0 Formal Semantics] (see step SI1 in Figure 1) or may be generated by some other mechanism (see step SI2 in Figure 1). In either case, the result must satisfy the consistency constraints defined in § 2.2.5 – Consistency Constraints on page 12.

2.2.3. Expression Processing

XQuery 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 on page 14.

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

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 query is parsed into an internal representation called the *operation tree* (step SQ1 in Figure 1). A parse error is raised as a static error . The static context is initialized by the implementation (step SQ2). The static context is then changed and augmented based on information in the *prolog* (step SQ3). If the Schema Import Feature is supported, the in-scope schema definitions are populated with information from imported schemas. If the Module Feature is supported, the static context is extended with function declarations and variable declarations from imported modules. 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 (or) is raised (however, see exceptions to this rule in § 2.5.4.3 – Element Test on page 28 and § 2.5.4.5 – Attribute Test on page 29.)

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].

Each expression is then assigned a static type (step SQ6). 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]. 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 . 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

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.

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 .

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

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].

An XQuery implementation is not required to provide a serialization interface. For example, an implementation may only provide a DOM interface (see [Document Object Model]) or an interface based on an event stream. In these cases, serialization would be outside of the scope of this specification.

[XSLT 2.0 and XQuery 1.0 Serialization] defines a set of *serialization parameters* that govern the serialization process. If an XQuery implementation provides a serialization interface, it may support (and may expose to users) any of the serialization parameters listed (with default values) in Appendix C.3 – Serialization Parameters on page 131. An XQuery implementation that provides a serialization interface must support some combination of serialization parameters in which `method = "xml"` and `version = "1.0"`.

The data model permits an element node to have fewer in-scope namespaces than its parent. Correct serialization of such an element node would require "undeclaration" of namespaces, which is a feature of [XML Names 1.1]. An implementation that does not support [XML Names 1.1] is permitted to serialize such an element without "undeclaration" of namespaces, which effectively causes the element to inherit the in-scope namespaces of its parent.

2.2.5. Consistency Constraints

In order for XQuery 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 XQuery implementation. Enforcement of these consistency constraints is beyond the scope of this specification. This specification does not define the result of a query 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*. 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 on page 25.
- 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 on page 25.
- 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 on page 25.
- The value of the context item must match the context item static type, using the matching rules in § 2.5.4 – SequenceType Matching on page 25.
- 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 on page 25.
- For each variable declared as `external`: If the variable declaration includes a declared type, the external environment must provide a value for the variable that matches the declared type, using the matching rules in § 2.5.4 – SequenceType Matching on page 25. If the variable declaration does not include a declared type, the external environment must provide a type and a matching value, using the same matching rules.
- For each function declared as external: the function implementation must either return a value that matches the declared result type, using the matching rules in § 2.5.4 – SequenceType Matching on page 25, or raise an implementation-defined error.
- For a given query, define a *participating ISSD* as the in-scope schema definitions of a module that is used in evaluating the query. If two participating ISSDs contain a definition for the same schema type, element name, or attribute name, the definitions must be equivalent in both ISSDs. Furthermore, if two participating ISSDs each contain a definition of a schema type *T*, the set of types derived by extension from *T* must be equivalent in both ISSDs. Also, if two participating ISSDs each contain a definition of an element name *E*, the substitution group headed by *E* must be equivalent in both ISSDs.
- 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 on page 10, XQuery 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.

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.

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.

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 . 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.

In addition to static errors, dynamic errors, and type errors, an XQuery 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:XXYYnnnn`, 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.
- `XX` denotes the language in which the error is defined, using the following encoding:

  - `XP` denotes an error defined by XPath. Such an error may also occur XQuery since XQuery includes XPath as a subset.
  - `XQ` denotes an error defined by XQuery.
- `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.

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

The method by which an XQuery 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`.

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.

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].

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]. 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 are free to rewrite expressions into equivalent expressions. Other than the raising or not raising of errors, the result of evaluating an equivalent expression must be the same as the result of evaluating the original 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 or `typeswitch` expressions. Conditional and `typeswitch` expressions raise only dynamic errors that occur in the branch that is actually selected. Thus, unlike the previous example, the following example cannot raise a dynamic error if `@x` is not castable into an `xs:date`:

```
$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 XQuery expressions.

2.4.1. Document Order

An ordering called *document order* is defined among all the nodes accessible during processing of a given query, which may consist of one or more *trees* (documents or fragments). Document order is defined in [XQuery/XPath Data Model (XDM)], and its definition is repeated here for convenience. 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. Informally, *document order* is the order in which nodes appear in the XML serialization of a document. Document order is *stable*, which means that the relative order of two nodes will not change during the processing of a given query, 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. Attribute nodes immediately follow the element node with which they are associated. The relative order of attribute nodes is stable but implementation-dependent.
4. The relative order of siblings is the order in which they occur in the `children` property of their parent node.
5. 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 XQuery 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]. *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].

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
- Constructor expressions for various kinds of nodes
- `order by` clauses in FLWOR expressions

2.4.3. Effective Boolean Value

Under certain circumstances (listed below), it is necessary to find the effective boolean value of a 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].

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].

The static semantics of `fn:boolean` are defined in .

The effective boolean value of a sequence that contains at least one node and at least one atomic value may be nondeterministic in regions of a query where ordering mode is `unordered`.

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
- The `where` clause of a FLWOR expression
- Certain types of predicates, such as `a[b]`
- Conditional expressions (`if`)
- Quantified expressions (`some`, `every`)

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

XQuery 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].

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 XQuery 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] 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] 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.

2.4.5. URI Literals

In certain places in the XQuery grammar, a statically known valid URI is required. These places are denoted by the grammatical symbol **URILiteral**. For example, URILiterals are used to specify namespaces and collations, both of which must be statically known.

[2] URILiteral ::= **StringLiteral**

Syntactically, a URILiteral is identical to a **StringLiteral**: a sequence of zero or more characters enclosed in single or double quotes. However, an implementation MAY raise a static error if the value of a URILiteral is of nonzero length and is not in the lexical space of `xs:anyURI`.

As in a string literal, any predefined entity reference (such as `&`), character reference (such as `•`), or **EscapeQuot** or **EscapeApos** (for example, `""`) is replaced by its appropriate expansion. Certain characters, notably the ampersand, can only be represented using a predefined entity reference or a character reference.

The URILiteral is subjected to whitespace normalization as defined for the `xs:anyURI` type in [XML Schema]: this means that leading and trailing whitespace is removed, and any other sequence of whitespace characters is replaced by a single space (#x20) character. Whitespace normalization is done after the expansion of character references, so writing a newline (for example) as `
` does not prevent its being normalized to a space character.

The URILiteral is not automatically subjected to percent-encoding or decoding as defined in [RFC3986]. Any process that attempts to resolve the URI against a base URI, or to dereference the URI, may however apply percent-encoding or decoding as defined in the relevant RFCs.

The `xs:anyURI` type is designed to anticipate the introduction of Internationalized Resource Identifiers (IRI's) as defined in [RFC3987].

The following is an example of a valid URILiteral:

```
"http://www.w3.org/2005/xpath-functions/collation/codepoint"
```

2.5. Types

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

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 XQuery expression. The term *sequence type* suggests that this syntax is used to describe the type of an XQuery value, which is always a sequence.

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 certain predefined schema types, including the built-in schema types in the namespace `http://www.w3.org/2001/XMLSchema`, which has the predefined namespace prefix `xs`. The schema types in this namespace are defined in [XML Schema] and augmented by additional types defined in [XQuery/XPath Data Model (XDM)]. Element and attribute declarations in the `xs` namespace are not implicitly included in the static context. The schema types defined in [XQuery/XPath Data Model (XDM)] are summarized below.

1. `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. `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. `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. `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. `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:integer`, `xs:string`, and `xs:untypedAtomic`, have `xs:anyAtomicType` as their base type.

`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 XQuery type hierarchy can be found in [XQuery 1.0 and XPath 2.0 Functions and Operators].

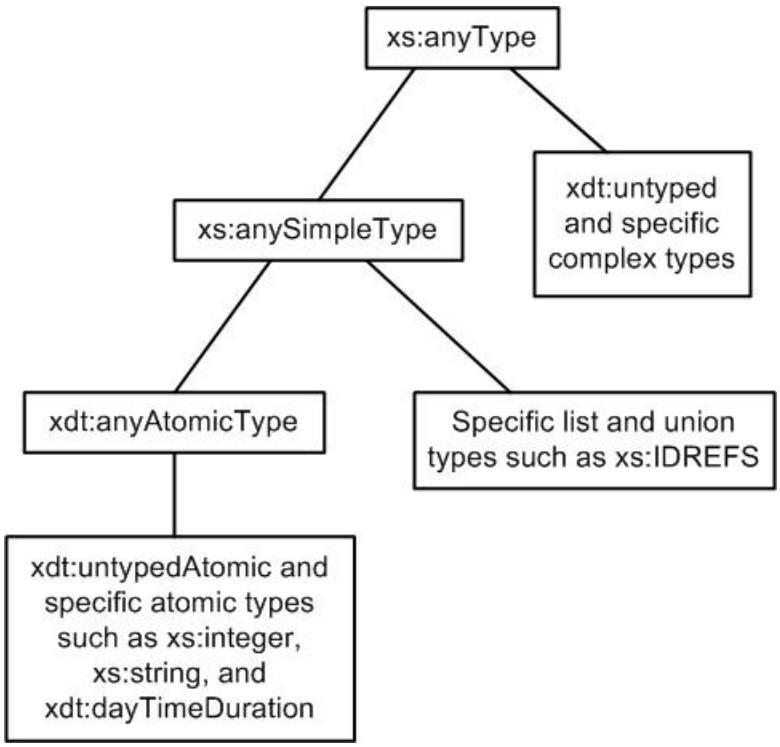

Figure 2: Hierarchy of Schema Types used in XQuery

2.5.2. Typed Value and String Value

Every node has a *typed value* and a *string 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. 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].

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, and are defined by rules found in the following locations:

- If the node was created by mapping from an Infoset or PSVI, see rules in [XQuery/XPath Data Model (XDM)].

- If the node was created by an XQuery node constructor, see rules in § 3.7.1 – Direct Element Constructors on page 55, § 3.7.3.1 – Computed Element Constructors on page 65, or § 3.7.3.2 – Computed Attribute Constructors on page 67.
- If the node was created by a `validate` expression, see rules in § 3.13 – Validate Expressions on page 89.

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 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.

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 XQuery expression, the **SequenceType** syntax is used.

[3]	SequenceType	::=	("empty-sequence" "(" ")")| (**ItemType** **OccurrenceIndicator**?)
[4]	ItemType	::=	**KindTest** | ("item" "(" ")") | **AtomicType**
[5]	OccurrenceIndicator	::=	"?" | "*" | "+"
[6]	AtomicType	::=	**QName**
[7]	KindTest	::=	**DocumentTest**| **ElementTest**| **AttributeTest**| **SchemaElementTest**| **SchemaAttributeTest**| **PITest**| **CommentTest**| **TextTest**| **AnyKindTest**
[8]	DocumentTest	::=	"document-node" "(" (**ElementTest** | **SchemaElementTest**)? ")"
[9]	ElementTest	::=	"element" "(" (**ElementNameOrWildcard** ("," **TypeName** "?"?)?)? ")"
[10]	SchemaElementTest	::=	"schema-element" "(" **ElementDeclaration** ")"
[11]	ElementDeclaration	::=	**ElementName**
[12]	AttributeTest	::=	"attribute" "(" (**AttribNameOrWildcard** ("," **TypeName**)?)? ")"
[13]	SchemaAttributeTest	::=	"schema-attribute" "(" **AttributeDeclaration** ")"

```
[14]  AttributeDeclaration  ::=  AttributeName
[15]  ElementNameOrWildcard ::=  ElementName | "*"
[16]  ElementName           ::=  QName
[17]  AttribNameOrWildcard  ::=  AttributeName | "*"
[18]  AttributeName         ::=  QName
[19]  TypeName              ::=  QName
[20]  PITest                ::=  "processing-instruction" "(" (NCName | StringLiteral)? ")"
[21]  CommentTest           ::=  "comment" "(" ")"
[22]  TextTest              ::=  "text" "(" ")"
[23]  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 XQuery 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

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], 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.

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 . The pseudo-function `derives-from` is defined below and is defined formally in [XQuery 1.0 and XPath 2.0 Formal Semantics].

- `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 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 `derives-from` pseudo-function cannot be written as a real XQuery function, because types are not valid function parameters.

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 on page 27).
- 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 .

  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`.

  

  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 name (called its "PITarget" in XML) is equal to *N*, where *N* is an NCName.

  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 on page 28 and § 2.5.4.4 – Schema Element Test on page 29).

  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 . 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 .

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 .

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 .

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

[24] Comment ::= "(:" (**CommentContents** | **Comment**)* ":)"

[25] CommentContents ::= (**Char**+ - (Char* (':' | ':)') Char*))

Comments may be used to provide informative annotation for a query, either in the Prolog or in the Query Body. Comments are lexical constructs only, and do not affect query processing.

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

A comment may be used anywhere ignorable whitespace is allowed (see Appendix A.2.4.1 – Default Whitespace Handling on page 121).

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 XQuery 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 [Appendix A – XQuery Grammar on page 109].

A *query* consists of one or more modules. If a query is executable, one of its modules has a Query Body containing an expression whose value is the result of the query. An expression is represented in the XQuery grammar by the symbol **Expr**.

[26] Expr ::= **ExprSingle** ("," **ExprSingle**)*

[27] ExprSingle ::= **FLWORExpr**| **QuantifiedExpr**| **TypeswitchExpr**| **IfExpr**| **OrExpr**

The XQuery 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 **FLWORExpr**, **QuantifiedExpr**, **TypeswitchExpr**, **IfExpr**, and **OrExpr**. Each of these expressions is described in a separate section of this document.

3.1. Primary Expressions

Primary expressions are the basic primitives of the language. They include literals, variable references, context item expressions, constructors, 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. Constructors are described in § 3.7 – Constructors on page 54.

[28] PrimaryExpr ::= **Literal** | **VarRef** | **ParenthesizedExpr** | **ContextItemExpr** | **Function-Call** | **OrderedExpr** | **UnorderedExpr** | **Constructor**

3.1.1. Literals

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

[29] Literal ::= **NumericLiteral** | **StringLiteral**

[30] NumericLiteral ::= **IntegerLiteral** | **DecimalLiteral** | **DoubleLiteral**

[31] IntegerLiteral ::= **Digits**

[32] DecimalLiteral ::= ("." **Digits**) | (**Digits** "." [0-9]*)

[33] DoubleLiteral ::= (("." **Digits**) | (**Digits** ("." [0-9]*)?)) [eE] [+-]? **Digits**

[34] StringLiteral ::= ("" (**PredefinedEntityRef** | **CharRef** | **EscapeQuot** | [^"&])* "") | ("'" (**PredefinedEntityRef** | **CharRef** | **EscapeApos** | [^'&])* "'")

[35] PredefinedEntityRef ::= "&" ("lt" | "gt" | "amp" | "quot" | "apos") ";"

[36] 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 .

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.

A string literal may contain a *predefined entity reference*. A *predefined entity reference* is a short sequence of characters, beginning with an ampersand, that represents a single character that might otherwise have syntactic significance. Each predefined entity reference is replaced by the character it represents when the string literal is processed. The predefined entity references recognized by XQuery are as follows:

Entity Reference	Character Represented
`<`	`<`
`>`	`>`
`&`	`&`
`&quot;`	`"`
`&apos;`	`'`

A string literal may also contain a *character reference*. A *character reference* is an XML-style reference to a [Unicode] character, identified by its decimal or hexadecimal code point. For example, the Euro symbol (€) can be represented by the character reference `&#8364;`. Character references are normatively defined in Section 4.1 of the XML specification (it is implementation-defined whether the rules in [XML 1.0] or [XML 1.1] apply.) A static error is raised if a character reference does not identify a valid character in the version of XML that is in use.

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.
- `"Ben & Jerry&apos;s"` denotes the `xs:string` value "`Ben & Jerry's`".
- `"&#8364;99.50"` denotes the `xs:string` value "`€99.50`".

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]. 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

[37] VarRef ::= "$" **VarName**

[38] VarName ::= **QName**

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. A variable may be declared in a Prolog, in the current module or an *imported module*. See § 4 – Modules and Prologs on page 92 for a discussion of modules and Prologs.
2. The in-scope variables may be augmented by implementation-defined variables.
3. A variable may be bound by an XQuery expression. The kinds of expressions that can bind variables are FLWOR expressions (§ 3.8 – FLWOR Expressions on page 73), quantified expressions (§ 3.11 – Quantified Expressions on page 83), and `typeswitch` expressions (§ 3.12.2 – Typeswitch on page 85). Function calls also bind values to the formal parameters of functions before executing the function body.

Every variable binding has a static scope. The scope defines where references to the variable can validly occur. It is a static error 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.

A reference to a variable that was declared `external`, but was not bound to a value by the external environment, raises a dynamic error .

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

[39] 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 on page 45.

3.1.4. Context Item Expression

[40] 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 .

3.1.5. Function Calls

The *built-in functions* supported by XQuery are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators]. Additional functions may be declared in a Prolog, imported from a library module, or provided by the external environment as part of the static context.

[41] 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 .

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. If the function is a built-in function, it is evaluated using the converted argument values. The result is either an instance of the function's declared return type or a dynamic error. Errors raised by built-in functions are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators].
4. If the function is a user-declared function that has a body, the converted argument values are bound to the formal parameters of the function, and the function body is evaluated. The value returned by the function body is then converted to the declared return type of the function by applying the function conversion rules.

   When a converted argument value is bound to a function parameter, the argument value retains its most specific dynamic type, even though this type may be derived from the type of the formal parameter. For example, a function with a parameter `$p` of type `xs:decimal` can be invoked with an argument of type `xs:integer`, which is derived from `xs:decimal`. During the processing of this function invocation, the dynamic type of `$p` inside the body of the function is considered to be

`xs:integer`. Similarly, the value returned by a function retains its most specific type, which may be derived from the declared return type of the function. For example, a function that has a declared return type of `xs:decimal` may in fact return a value of dynamic type `xs:integer`.

During evaluation of a function body, the static context and dynamic context for expression evaluation are defined by the module in which the function is declared, which is not necessarily the same as the module in which the function is called. For example, the variables in scope while evaluating a function body are defined by in-scope variables of the module that declares the function rather than the module in which the function is called. During evaluation of a function body, the focus (context item, context position, and context size) is undefined, except where it is defined by some expression inside the function body.

5. If the function is a user-declared external function, its function implementation is invoked with the converted argument values. The result is either a value of the declared type or an implementation-defined error (see § 2.2.5 – Consistency Constraints on page 12).

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

- 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 Appendix B.1 – Type Promotion on page 123, 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 Appendix B.1 – Type Promotion on page 123, 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 . If the function call takes place in a module other than the module in which the function is defined, this rule must be satisfied in both the module where the function is called and the module where the function is defined (the test is repeated because the two modules may have different in-scope schema definitions.) 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

[42] PathExpr ::= ("/" **RelativePathExpr**?)| ("//" **RelativePathExpr**)| **RelativePathExpr**

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

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 on page 37.

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 . At evaluation time, if the root node above the context node is not a document node, a dynamic error is raised .

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 .) 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 . At evaluation time, if the root node above the context node is not a document node, a dynamic error is raised .

The descendants of a node do not include attribute nodes .

Each non-initial occurrence of "`//`" in a path expression is expanded as described in § 3.2.4 – Abbreviated Syntax on page 43, 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 . 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 on page 6. 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. If ordering mode is `ordered`, the resulting node sequence is returned in document order; otherwise it is returned in implementation-dependent order.

2. If every evaluation of `E2` returns a (possibly empty) sequence of atomic values, these sequences are concatenated and returned. If ordering mode is `ordered`, the returned sequence preserves the orderings within and among the subsequences generated by the evaluations of `E2`; otherwise the order of the returned sequence is implementation-dependent.
3. If the multiple evaluations of `E2` return at least one node and at least one atomic value, a type error is raised .

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.

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

[44]	StepExpr	::=	**FilterExpr** | **AxisStep**
[45]	AxisStep	::=	(**ReverseStep** | **ForwardStep**) **PredicateList**
[46]	ForwardStep	::=	(**ForwardAxis NodeTest**) | **AbbrevForwardStep**
[47]	ReverseStep	::=	(**ReverseAxis NodeTest**) | **AbbrevReverseStep**
[48]	PredicateList	::=	**Predicate***

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 on page 46.

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 . If ordering mode is `ordered`, the resulting node sequence is returned in document order; otherwise it is returned in implementation-dependent 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 on page 43.

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 on page 38. The available node tests are described in § 3.2.1.2 – Node Tests on page 39. Examples of steps are provided in § 3.2.3 – Unabbreviated Syntax on page 42 and § 3.2.4 – Abbreviated Syntax on page 43.

3.2.1.1. Axes

[49] ForwardAxis ::= ("child" "::")| ("descendant" "::")| ("attribute" "::")| ("self" "::")| ("descendant-or-self" "::")| ("following-sibling" "::")| ("following" "::")

[50] ReverseAxis ::= ("parent" "::")| ("ancestor" "::")| ("preceding-sibling" "::")| ("preceding" "::")| ("ancestor-or-self" "::")

XQuery supports the following axes (subject to limitations as described in § 5.2.4 – Full Axis Feature on page 108):

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

  

  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 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/XPath Data Model (XDM)], which returns the parent of the context node, or an empty sequence if the context node has no parent

  

  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)

  

  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 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 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/XPath Data Model (XDM)]; 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

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 nodes): they do not overlap and together they contain all the nodes in the document.

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 all other axes, the principal node kind is element.

3.2.1.2. Node Tests

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.

[51] NodeTest ::= **KindTest** | **NameTest**

[52] NameTest ::= **QName** | **Wildcard**

[53] Wildcard ::= "*"| (**NCName** ":" "*")| ("*" ":" **NCName**)

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 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 . 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.

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 on page 24 and § 2.5.4 – SequenceType Matching on page 25. 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

[54] Predicate ::= "[" **Expr** "]"

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

   

   In a region of a query where ordering mode is `unordered`, the result of a numeric predicate is nondeterministic, as explained in § 3.9 – Ordered and Unordered Expressions on page 81.

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]
  ```

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 (when in ordered mode) 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 on page 43.

- `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

[55] AbbrevForwardStep ::= "@"? **NodeTest**

[56] 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.

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.

The expression `.`, known as a *context item expression*, is a primary expression, and is described in § 3.1.4 – Context Item Expression on page 34.

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

XQuery 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

[57] Expr ::= **ExprSingle** ("," **ExprSingle**)*

[58] RangeExpr ::= **AdditiveExpr** ("to" **AdditiveExpr**)?

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.

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

[59] FilterExpr ::= **PrimaryExpr PredicateList**

[60] PredicateList ::= **Predicate***

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 on page 45 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

[61] UnionExpr ::= **IntersectExceptExpr** (("union" | "|") **IntersectExceptExpr**)*

[62] IntersectExceptExpr ::= **InstanceofExpr** (("intersect" | "except") **InstanceofExpr**)*

XQuery 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. If ordering mode is `ordered`, the resulting sequence is returned in document order; otherwise it is returned in implementation-dependent order.

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

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] 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

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

[63] AdditiveExpr ::= **MultiplicativeExpr** (("+" | "-") **MultiplicativeExpr**)*

[64] MultiplicativeExpr ::= **UnionExpr** (("*" | "div" | "idiv" | "mod") **UnionExpr**)*

[65] UnaryExpr ::= ("-" | "+")* **ValueExpr**

[66] ValueExpr ::= **ValidateExpr** | **PathExpr** | **ExtensionExpr**

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 Appendix A.2.4 – Whitespace Rules on page 121 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.

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 .
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 Appendix B.2 – Operator Mapping on page 124 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].

If the types of the operands, after evaluation, are not a valid combination for the given operator, according to the rules in Appendix B.2 – Operator Mapping on page 124, a type error is raised .

XQuery 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], `$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)
```

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

3.5. Comparison Expressions

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

[67]	ComparisonExpr	::=	**RangeExpr** ((**ValueComp**| **GeneralComp**| **NodeComp**) **RangeExpr**)?
[68]	ValueComp	::=	"eq" | "ne" | "lt" | "le" | "gt" | "ge"
[69]	GeneralComp	::=	"=" | "!=" | "<" | "<=" | ">" | ">="
[70]	NodeComp	::=	"is" | "<<" | ">>"

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 .
4. If the atomized operand is of type `xs:untypedAtomic`, it is cast to `xs:string`.

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 Appendix B.2 – Operator Mapping on page 124 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].

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 Appendix B.2 – Operator Mapping on page 124, a type error is raised .

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 .

```
$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 comparisons are true because, in each case, the two constructed nodes have the same value after atomization, even though they have different identities and/or names:

```
<a>5</a> eq <a>5</a>
```

```
<a>5</a> eq <b>5</b>
```

- 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`.

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]

   

   The purpose of these rules is to preserve compatibility with XPath 1.0, in which (for example) `x < 17` is a numeric comparison if `x` is an untyped value. Users should be aware that the value comparison operators have different rules for casting of `xs:untypedAtomic` operands.

   A. If one of the atomic values is an instance of `xs:untypedAtomic` and the other is an instance of a numeric type, then the `xs:untypedAtomic` value is cast to the type `xs:double`.

   B. If one of the atomic values is an instance of `xs:untypedAtomic` and the other is an instance of `xs:untypedAtomic` or `xs:string`, then the `xs:untypedAtomic` value (or values) is (are) cast to the type `xs:string`.

   C. If one of the atomic values is an instance of `xs:untypedAtomic` and the other is not an instance of `xs:string`, `xs:untypedAtomic`, or any numeric type, then the `xs:untypedAtomic` value is cast to the dynamic type of the other value.

   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`.

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 .
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/XPath Data Model (XDM)] 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 false because each constructed node has its own identity:

```
<a>5</a> is <a>5</a>
```

- 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`.

[71] OrExpr ::= **AndExpr** ("or" **AndExpr**)*

[72] 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 on page 19).

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

AND:	EBV_2 = `true`	EBV_2 = `false`	error in EBV_2
EBV_1 = `true`	`true`	`false`	error
EBV_1 = `false`	`false`	`false`	either `false` or error
error in EBV_1	error	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`	either `true` or error
EBV_1 = `false`	`true`	`false`	error
error in EBV_1	either `true` or error	error	error

The order in which the operands of a logical expression are evaluated is implementation-dependent. The tables above are defined in such a way that 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:

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

- The following expression may return either `true` or raise a dynamic error:

```
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, XQuery 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]. 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. Constructors

XQuery provides constructors that can create XML structures within a query. Constructors are provided for element, attribute, document, text, comment, and processing instruction nodes. Two kinds of constructors are provided: *direct constructors*, which use an XML-like notation, and *computed constructors*, which use a notation based on enclosed expressions.

[73]	Constructor	::=	**DirectConstructor**| **ComputedConstructor**
[74]	DirectConstructor	::=	**DirElemConstructor**| **DirCommentConstructor**| **DirPIConstructor**
[75]	DirElemConstructor	::=	"<" **QName DirAttributeList** ("/>" | (">" **DirElemContent*** "</" **QName S**? ">"))
[76]	DirElemContent	::=	**DirectConstructor**| **CDataSection**| **CommonContent**| **ElementContentChar**
[77]	ElementContentChar	::=	**Char** - [{}<&]
[78]	CommonContent	::=	**PredefinedEntityRef** | **CharRef** | "{{" | "}}" | **EnclosedExpr**
[79]	CDataSection	::=	"<![CDATA[" **CDataSectionContents** "]]>"
[80]	CDataSectionContents	::=	(**Char*** - (Char* ']]>' Char*))
[81]	DirAttributeList	::=	(**S** (**QName S**? "=" **S**? **DirAttributeValue**)?)*
[82]	DirAttributeValue	::=	('"' (**EscapeQuot** | **QuotAttrValueContent**)* '"')| ("'" (**EscapeApos** | **AposAttrValueContent**)* "'")
[83]	QuotAttrValueContent	::=	**QuotAttrContentChar**| **CommonContent**
[84]	AposAttrValueContent	::=	**AposAttrContentChar**| **CommonContent**
[85]	QuotAttrContentChar	::=	**Char** - ["{}<&]
[86]	AposAttrContentChar	::=	**Char** - ['{}<&]
[87]	EscapeQuot	::=	'"'
[88]	EscapeApos	::=	"'"
[89]	EnclosedExpr	::=	"{" **Expr** "}"

This section contains a conceptual description of the semantics of various kinds of constructor expressions. An XQuery implementation is free to use any implementation technique that produces the same result as the processing steps described in this section.

3.7.1. Direct Element Constructors

An *element constructor* creates an element node. A *direct element constructor* is a form of element constructor in which the name of the constructed element is a constant. Direct element constructors are based on standard XML notation. For example, the following expression is a direct element constructor that creates a `book` element containing an attribute and some nested elements:

```
<book isbn="isbn-0060229357">
    <title>Harold and the Purple Crayon</title>
    <author>
        <first>Crockett</first>
        <last>Johnson</last>
    </author>
</book>
```

If the element name in a direct element constructor has a namespace prefix, the namespace prefix is resolved to a namespace URI using the statically known namespaces. If the element name has no namespace prefix, it is implicitly qualified by the default element/type namespace. Note that both the statically known namespaces and the default element/type namespace may be affected by namespace declaration attributes found inside the element constructor. The namespace prefix of the element name is retained after expansion of the QName, as described in [XQuery/XPath Data Model (XDM)]. The resulting expanded QName becomes the `node-name` property of the constructed element node.

In a direct element constructor, the name used in the end tag must exactly match the name used in the corresponding start tag, including its prefix or absence of a prefix.

In a direct element constructor, curly braces { } delimit *enclosed expressions*, distinguishing them from literal text. Enclosed expressions are evaluated and replaced by their value, as illustrated by the following example:

```
<example>
   <p> Here is a query. </p>
   <eg> $b/title </eg>
   <p> Here is the result of the query. </p>
   <eg>{ $b/title }</eg>
</example>
```

The above query might generate the following result (whitespace has been added for readability to this result and other result examples in this document):

```
<example>
  <p> Here is a query. </p>
  <eg> $b/title </eg>
  <p> Here is the result of the query. </p>
  <eg><title>Harold and the Purple Crayon</title></eg>
</example>
```

Since XQuery uses curly braces to denote enclosed expressions, some convention is needed to denote a curly brace used as an ordinary character. For this purpose, a pair of identical curly brace characters within the content of an element or attribute are interpreted by XQuery as a single curly brace character (that is, the pair "`{{`" represents the character "`{`" and the pair "`}}`" represents the character "`}`".) Alternatively, the character references `&#x7b;` and `&#x7d;` can be used to denote curly brace characters. A single left curly brace ("`{`") is interpreted as the beginning delimiter for an enclosed expression. A single right curly brace ("`}`") without a matching left curly brace is treated as a static error .

The result of an element constructor is a new element node, with its own node identity. All the attribute and descendant nodes of the new element node are also new nodes with their own identities, even if they are copies of existing nodes.

3.7.1.1. Attributes

The start tag of a direct element constructor may contain one or more attributes. As in XML, each attribute is specified by a name and a value. In a direct element constructor, the name of each attribute is specified by a constant QName, and the value of the attribute is specified by a string of characters enclosed in single or double quotes. As in the main content of the element constructor, an attribute value may contain expressions enclosed in curly braces, which are evaluated and replaced by their value during processing of the element constructor.

Each attribute in a direct element constructor creates a new attribute node, with its own node identity, whose parent is the constructed element node. However, note that namespace declaration attributes (see § 3.7.1.2 – Namespace Declaration Attributes on page 57) do not create attribute nodes.

If an attribute name has a namespace prefix, the prefix is resolved to a namespace URI using the statically known namespaces. If the attribute name has no namespace prefix, the attribute is in no namespace. Note that the statically known namespaces used in resolving an attribute name may be affected by namespace declaration attributes that are found inside the same element constructor. The namespace prefix of the attribute name is retained after expansion of the QName, as described in [XQuery/XPath Data Model (XDM)]. The resulting expanded QName becomes the `node-name` property of the constructed attribute node.

If the attributes in a direct element constructor do not have distinct expanded QNames as their respective `node-name` properties, a static error is raised .

Conceptually, an attribute (other than a namespace declaration attribute) in a direct element constructor is processed by the following steps:

1. Each consecutive sequence of literal characters in the attribute content is treated as a string containing those characters. Attribute value normalization is then applied to normalize whitespace and expand character references and predefined entity references. An XQuery processor that supports XML 1.0 uses the rules for attribute value normalization in Section 3.3.3 of [XML 1.0]; an XQuery processor that supports XML 1.1 uses the rules for attribute value normalization in Section 3.3.3 of [XML 1.1]. In either case, the normalization rules are applied as though the type of the attribute were CDATA (leading and trailing whitespace characters are not stripped.) The choice between XML 1.0 and XML 1.1 rules is implementation-defined.
2. Each enclosed expression is converted to a string as follows:
   A. Atomization is applied to the value of the enclosed expression, converting it to a sequence of atomic values.

   B. If the result of atomization is an empty sequence, the result is the zero-length string. Otherwise, each atomic value in the atomized sequence is cast into a string.

   C. The individual strings resulting from the previous step are merged into a single string by concatenating them with a single space character between each pair.

3. Adjacent strings resulting from the above steps are concatenated with no intervening blanks. The resulting string becomes the `string-value` property of the attribute node. The attribute node is given a type annotation (`type-name` property) of `xs:untypedAtomic` (this type annotation may change if the parent element is validated). The `typed-value` property of the attribute node is the same as its `string-value`, as an instance of `xs:untypedAtomic`.

4. The `parent` property of the attribute node is set to the element node constructed by the direct element constructor that contains this attribute.

5. If the attribute name is `xml:id`, then `xml:id` processing is performed as defined in [XML ID]. This ensures that the attribute has the type `xs:ID` and that its value is properly normalized. If an error is encountered during `xml:id` processing, an implementation MAY raise a dynamic error .

6. If the attribute name is `xml:id`, the `is-id` property of the resulting attribute node is set to `true`; otherwise the `is-id` property is set to `false`. The `is-idrefs` property of the attribute node is unconditionally set to `false`.

- Example:

  ```
  <shoe size="7"/>
  ```

  The string value of the `size` attribute is "`7`".

- Example:

  ```
  <shoe size="{7}"/>
  ```

  The string value of the `size` attribute is "`7`".

- Example:

  ```
  <shoe size="{()}"/>
  ```

  The string value of the `size` attribute is the zero-length string.

- Example:

  ```
  <chapter ref="[{1, 5 to 7, 9}]"/>
  ```

  The string value of the `ref` attribute is "`[1 5 6 7 9]`".

- Example:

  ```
  <shoe size="As big as {$hat/@size}"/>
  ```

  The string value of the `size` attribute is the string "`As big as `", concatenated with the string value of the node denoted by the expression `$hat/@size`.

3.7.1.2. Namespace Declaration Attributes

The names of a constructed element and its attributes may be QNames that include *namespace prefixes*. Namespace prefixes can be bound to namespaces in the Prolog or by *namespace declaration attributes*. It is a static error to use a namespace prefix that has not been bound to a namespace .

A *namespace declaration attribute* is used inside a direct element constructor. Its purpose is to bind a namespace prefix or to set the default element/type namespace for the constructed element node, including its attributes. Syntactically, a namespace declaration attribute has the form of an attribute with namespace prefix `xmlns`, or with name `xmlns` and no namespace prefix. The value of a namespace declaration attribute must be a **URILiteral**; otherwise a static error is raised . All the namespace declaration attributes of a given element must have distinct names . Each namespace declaration attribute is processed as follows:

- The local part of the attribute name is interpreted as a namespace prefix and the value of the attribute is interpreted as a namespace URI. This prefix and URI are added to the statically known namespaces of the constructor expression (overriding any existing binding of the given prefix), and are also added as a namespace binding to the in-scope namespaces of the constructed element. If the namespace URI is a zero-length string and the implementation supports [XML Names 1.1], any existing namespace binding for the given prefix is removed from the in-scope namespaces of the constructed element and from the statically known namespaces of the constructor expression. If the namespace URI is a zero-length string and the implementation does not support [XML Names 1.1], a static error is raised . It is implementation-defined whether an implementation supports [XML Names] or [XML Names 1.1].
- If the name of the namespace declaration attribute is `xmlns` with no prefix, the value of the attribute is interpreted as a namespace URI. This URI specifies the default element/type namespace of the constructor expression (overriding any existing default), and is added (with no prefix) to the in-scope namespaces of the constructed element (overriding any existing namespace binding with no prefix). If the namespace URI is a zero-length string, the default element/type namespace of the constructor expression is set to "none," and any no-prefix namespace binding is removed from the in-scope namespaces of the constructed element.
- It is a static error if a namespace declaration attribute binds a namespace URI to the predefined prefix `xmlns`. It is also a static error if a namespace declaration attribute binds a namespace URI other than `http://www.w3.org/XML/1998/namespace` to the prefix `xml`, or binds a prefix other than `xml` to the namespace URI `http://www.w3.org/XML/1998/namespace`.

A namespace declaration attribute does not cause an attribute node to be created.

The following examples illustrate namespace declaration attributes:

- In this element constructor, a namespace declaration attribute is used to set the default element/type namespace to `http://example.org/animals`:

```
<cat xmlns = "http://example.org/animals">
  <breed>Persian</breed>
</cat>
```

- In this element constructor, namespace declaration attributes are used to bind the namespace prefixes `metric` and `english`:

```
<box xmlns:metric = "http://example.org/metric/units"
     xmlns:english = "http://example.org/english/units">
  <height> <metric:meters>3</metric:meters> </height>
  <width> <english:feet>6</english:feet> </width>
  <depth> <english:inches>18</english:inches> </depth>
</box>
```

3.7.1.3. Content

The part of a direct element constructor between the start tag and the end tag is called the *content* of the element constructor. This content may consist of text characters (parsed as **ElementContentChar**), nested direct constructors, **CdataSections**, character and predefined entity references, and expressions enclosed in curly braces. In general, the value of an enclosed expression may be any sequence of nodes and/or atomic values. Enclosed expressions can be used in the content of an element constructor to compute both the content and the attributes of the constructed node.

Conceptually, the content of an element constructor is processed as follows:

1. The content is evaluated to produce a sequence of nodes called the *content sequence*, as follows:
   A. If the boundary-space policy in the static context is `strip`, boundary whitespace is identified and deleted (see § 3.7.1.4 – Boundary Whitespace on page 62 for a definition of boundary whitespace.)
   B. Predefined entity references and character references are expanded into their referenced strings, as described in § 3.1.1 – Literals on page 31. Characters inside a **CDataSection**, including special characters such as `<` and `&`, are treated as literal characters rather than as markup characters (except for the sequence `]]>`, which terminates the CDataSection).
   C. Each consecutive sequence of literal characters evaluates to a single text node containing the characters.
   D. Each nested direct constructor is evaluated according to the rules in § 3.7.1 – Direct Element Constructors on page 55 or § 3.7.2 – Other Direct Constructors on page 63, resulting in a new element, comment, or processing instruction node. Then:
      i. The `parent` property of the resulting node is then set to the newly constructed element node.
      ii. The `base-uri` property of the resulting node, and of each of its descendants, is set to be the same as that of its new parent, unless it (the child node) has an `xml:base` attribute, in which case its `base-uri` property is set to the value of that attribute, resolved (if it is relative) against the `base-uri` property of its new parent node.
   E. Enclosed expressions are evaluated as follows:
      i. For each adjacent sequence of one or more atomic values returned by an enclosed expression, a new text node is constructed, containing the result of casting each atomic value to a string, with a single space character inserted between adjacent values.

         
         The insertion of blank characters between adjacent values applies even if one or both of the values is a zero-length string.

      ii. For each node returned by an enclosed expression, a new copy is made of the given node and all nodes that have the given node as an ancestor, collectively referred to as *copied nodes*. The properties of the copied nodes are as follows:
         a) Each copied node receives a new node identity.
         b) The `parent`, `children`, and `attributes` properties of the copied nodes are set so as to preserve their inter-node relationships. For the topmost node (the node directly returned by the enclosed expression), the `parent` property is set to the node constructed by this constructor.

c) If construction mode in the static context is `strip`:

1) If the copied node is an element node, its `type-name` property is set to `xs:untyped`. Its `nilled`, `is-id`, and `is-idrefs` properties are set to `false`.

2) If the copied node is an attribute node, its `type-name` property is set to `xs:untypedAtomic`. Its `is-idrefs` property is set to `false`. Its `is-id` property is set to `true` if the qualified name of the attribute node is `xml:id`; otherwise it is set to `false`.

3) The `string-value` of each copied element and attribute node remains unchanged, and its `typed-value` becomes equal to its `string-value` as an instance of `xs:untypedAtomic`.

Implementations that store only the typed value of a node are required at this point to convert the typed value to a string form.

On the other hand, if construction mode in the static context is `preserve`, the `type-name`, `nilled`, `string-value`, `typed-value`, `is-id`, and `is-idrefs` properties of the copied nodes are preserved.

d) The `in-scope-namespaces` property of a copied element node is determined by the following rules. In applying these rules, the default namespace or absence of a default namespace is treated like any other namespace binding:

1) If copy-namespaces mode specifies `preserve`, all in-scope-namespaces of the original element are retained in the new copy. If copy-namespaces mode specifies `no-preserve`, the new copy retains only those in-scope namespaces of the original element that are used in the names of the element and its attributes. It is a type error in this case if the typed value of the copied element or of any of its attributes is namespace-sensitive. A value is *namespace-sensitive* if it includes an item whose dynamic type is `xs:QName` or `xs:NOTATION` or is derived by restriction from `xs:QName` or `xs:NOTATION`.

Error can occur only if construction mode is `preserve`, since otherwise the typed value of the copied node is never namespace-sensitive.

2) If copy-namespaces mode specifies `inherit`, the copied node inherits all the in-scope namespaces of the constructed node, augmented and overridden by the in-scope namespaces of the original element that were preserved by the preceding rule. If copy-namespaces mode specifies `no-inherit`, the copied node does not inherit any in-scope namespaces from the constructed node.

e) When an element or processing instruction node is copied, its `base-uri` property is set to be the same as that of its new parent, with the following exception: if a copied element node has an `xml:base` attribute, its `base-uri` property is set to the value of that attribute, resolved (if it is relative) against the `base-uri` property of the new parent node.

f) All other properties of the copied nodes are preserved.

2. If the content sequence contains a document node, the document node is replaced in the content sequence by its children.

3. Adjacent text nodes in the content sequence are merged into a single text node by concatenating their contents, with no intervening blanks. After concatenation, any text node whose content is a zero-length string is deleted from the content sequence.
4. If the content sequence contains an attribute node following a node that is not an attribute node, a type error is raised .
5. The properties of the newly constructed element node are determined as follows:
   A. `node-name` is the expanded QName resulting from resolving the element name in the start tag, including its original namespace prefix (if any), as described in § 3.7.1 – Direct Element Constructors on page 55.
   B. `parent` is set to empty.
   C. `attributes` consist of all the attributes specified in the start tag as described in § 3.7.1.1 – Attributes on page 56, together with all the attribute nodes in the content sequence, in implementation-dependent order. Note that the `parent` property of each of these attribute nodes has been set to the newly constructed element node. If two or more attributes have the same `node-name`, a dynamic error is raised . If an attribute named `xml:space` has a value other than `preserve` or `default`, a dynamic error MAY be raised .
   D. `children` consist of all the element, text, comment, and processing instruction nodes in the content sequence. Note that the `parent` property of each of these nodes has been set to the newly constructed element node.
   E. `base-uri` is set to the following value:
      i. If the constructed node has an attribute named `xml:base`, then the value of this attribute, resolved if it is relative against the base URI in the static context. The value of the `xml:base` attribute is normalized as described in [XML Base].
      ii. Otherwise, the value of the base URI in the static context.
   F. `in-scope-namespaces` consist of all the namespace bindings resulting from namespace declaration attributes as described in § 3.7.1.2 – Namespace Declaration Attributes on page 57, and possibly additional namespace bindings as described in § 3.7.4 – In-scope Namespaces of a Constructed Element on page 71.
   G. The `nilled` property is `false`.
   H. The `string-value` property is equal to the concatenated contents of the text-node descendants in document order. If there are no text-node descendants, the `string-value` property is a zero-length string.
   I. The `typed-value` property is equal to the `string-value` property, as an instance of `xs:untypedAtomic`.
   J. If construction mode in the static context is `strip`, the `type-name` property is `xs:untyped`. On the other hand, if construction mode is `preserve`, the `type-name` property is `xs:anyType`.
   K. The `is-id` and `is-idrefs` properties are set to `false`.

- Example:

```
<a>{1}</a>
```

The constructed element node has one child, a text node containing the value "`1`".

- Example:

  ```
  <a>{1, 2, 3}</a>
  ```

  The constructed element node has one child, a text node containing the value "`1 2 3`".

- Example:

  ```
  <c>{1}{2}{3}</c>
  ```

  The constructed element node has one child, a text node containing the value "`123`".

- Example:

  ```
  <b>{1, "2", "3"}</b>
  ```

  The constructed element node has one child, a text node containing the value "`1 2 3`".

- Example:

  ```
  <fact>I saw 8 cats.</fact>
  ```

  The constructed element node has one child, a text node containing the value "`I saw 8 cats.`".

- Example:

  ```
  <fact>I saw {5 + 3} cats.</fact>
  ```

  The constructed element node has one child, a text node containing the value "`I saw 8 cats.`".

- Example:

  ```
  <fact>I saw <howmany>{5 + 3}</howmany> cats.</fact>
  ```

  The constructed element node has three children: a text node containing "`I saw `", a child element node named `howmany`, and a text node containing "` cats.`". The child element node in turn has a single text node child containing the value "`8`".

3.7.1.4. Boundary Whitespace

In a direct element constructor, whitespace characters may appear in the content of the constructed element. In some cases, enclosed expressions and/or nested elements may be separated only by whitespace characters. For example, in the expression below, the end-tag `</title>` and the start-tag `<author>` are separated by a newline character and four space characters:

```
<book isbn="isbn-0060229357">
    <title>Harold and the Purple Crayon</title>
    <author>
        <first>Crockett</first>
        <last>Johnson</last>
    </author>
</book>
```

Boundary whitespace is a sequence of consecutive whitespace characters within the content of a direct element constructor, that is delimited at each end either by the start or end of the content, or by a **DirectConstructor**, or by an **EnclosedExpr**. For this purpose, characters generated by character references such as `&#x20;` or by **CdataSections** are not considered to be whitespace characters.

The boundary-space policy in the static context controls whether boundary whitespace is preserved by element constructors. If boundary-space policy is `strip`, boundary whitespace is not considered significant and is discarded. On the other hand, if boundary-space policy is `preserve`, boundary whitespace is considered significant and is preserved.

- Example:

```
<cat>
   <breed>{$b}</breed>
   <color>{$c}</color>
</cat>
```

  The constructed `cat` element node has two child element nodes named `breed` and `color`. Whitespace surrounding the child elements will be stripped away by the element constructor if boundary-space policy is `strip`.

- Example:

```
<a>  {"abc"}  </a>
```

  If boundary-space policy is `strip`, this example is equivalent to `<a>abc</a>`. However, if boundary-space policy is `preserve`, this example is equivalent to `<a>  abc  </a>`.

- Example:

```
<a> z {"abc"}</a>
```

  Since the whitespace surrounding the `z` is not boundary whitespace, it is always preserved. This example is equivalent to `<a> z abc</a>`.

- Example:

```
<a>&#x20;{"abc"}</a>
```

  This example is equivalent to `<a> abc</a>`, regardless of the boundary-space policy, because the space generated by the character reference is not treated as a whitespace character.

- Example:

```
<a>{"  "}</a>
```

  This example constructs an element containing two space characters, regardless of the boundary-space policy, because whitespace inside an enclosed expression is never considered to be boundary whitespace.

Element constructors treat attributes named `xml:space` as ordinary attributes. An `xml:space` attribute does not affect the handling of whitespace by an element constructor.

3.7.2. Other Direct Constructors

XQuery allows an expression to generate a processing instruction node or a comment node. This can be accomplished by using a *direct processing instruction constructor* or a *direct comment constructor*. In each case, the syntax of the constructor expression is based on the syntax of a similar construct in XML.

[90] DirPIConstructor ::= "<?" **PITarget** (**S DirPIContents**)? "?>"

[91] DirPIContents ::= (**Char*** - (Char* '?>' Char*))

[92] DirCommentConstructor ::= "<!--" **DirCommentContents** "-->"

[93] DirCommentContents ::= ((**Char** - '-') | ('-' (**Char** - '-')))*

A direct processing instruction constructor creates a processing instruction node whose `target` property is **PITarget** and whose `content` property is **DirPIContents**. The `base-uri` property of the node is empty. The `parent` property of the node is empty.

The **PITarget** of a processing instruction may not consist of the characters "XML" in any combination of upper and lower case. The **DirPIContents** of a processing instruction may not contain the string "?>".

The following example illustrates a direct processing instruction constructor:

```
<?format role="output" ?>
```

A direct comment constructor creates a comment node whose `content` property is **DirCommentContents**. Its `parent` property is empty.

The **DirCommentContents** of a comment may not contain two consecutive hyphens or end with a hyphen. These rules are syntactically enforced by the grammar shown above.

The following example illustrates a direct comment constructor:

```
<!-- Tags are ignored in the following section -->
```

A direct comment constructor is different from a **comment**, since a direct comment constructor actually constructs a comment node, whereas a **comment** is simply used in documenting a query and is not evaluated.

3.7.3. Computed Constructors

[94] ComputedConstructor ::= **CompDocConstructor**| **CompElemConstructor**| **CompAttrConstructor**| **CompTextConstructor**| **CompCommentConstructor**| **CompPIConstructor**

An alternative way to create nodes is by using a *computed constructor*. A computed constructor begins with a keyword that identifies the type of node to be created: `element`, `attribute`, `document`, `text`, `processing-instruction`, or `comment`.

For those kinds of nodes that have names (element, attribute, and processing instruction nodes), the keyword that specifies the node kind is followed by the name of the node to be created. This name may be specified either as a QName or as an expression enclosed in braces. When an expression is used to specify the name of a constructed node, that expression is called the *name expression* of the constructor.

The final part of a computed constructor is an expression enclosed in braces, called the *content expression* of the constructor, that generates the content of the node.

The following example illustrates the use of computed element and attribute constructors in a simple case where the names of the constructed nodes are constants. This example generates exactly the same result as the first example in § 3.7.1 – Direct Element Constructors on page 55:

```
element book {
   attribute isbn {"isbn-0060229357" },
   element title { "Harold and the Purple Crayon"},
   element author {
      element first { "Crockett" },
      element last {"Johnson" }
```

```
    }
}
```

3.7.3.1. Computed Element Constructors

[95] CompElemConstructor ::= "element" (**QName** | ("{" **Expr** "}")) "{" **ContentExpr**? "}"

[96] ContentExpr ::= **Expr**

A *computed element constructor* creates an element node, allowing both the name and the content of the node to be computed.

If the keyword `element` is followed by a QName, it is expanded using the statically known namespaces, and the resulting expanded QName is used as the `node-name` property of the constructed element node. If expansion of the QName is not successful, a static error is raised .

If the keyword `element` is followed by a name expression, the name expression is processed as follows:

1. Atomization is applied to the value of the name expression. If the result of atomization is not a single atomic value of type `xs:QName`, `xs:string`, or `xs:untypedAtomic`, a type error is raised .
2. If the atomized value of the name expression is of type `xs:QName`, that expanded QName is used as the `node-name` property of the constructed element, retaining the prefix part of the QName.
3. If the atomized value of the name expression is of type `xs:string` or `xs:untypedAtomic`, that value is converted to an expanded QName. If the string value contains a namespace prefix, that prefix is resolved to a namespace URI using the statically known namespaces. If the string value contains no namespace prefix, it is treated as a local name in the default element/type namespace. The resulting expanded QName is used as the `node-name` property of the constructed element, retaining the prefix part of the QName. If conversion of the atomized name expression to an expanded QName is not successful, a dynamic error is raised .

The content expression of a computed element constructor (if present) is processed in exactly the same way as an enclosed expression in the content of a direct element constructor, as described in Step 1e of § 3.7.1.3 – Content on page 59. The result of processing the content expression is a sequence of nodes called the *content sequence*. If the content expression is absent, the content sequence is an empty sequence.

Processing of the computed element constructor proceeds as follows:

1. Adjacent text nodes in the content sequence are merged into a single text node by concatenating their contents, with no intervening blanks. After concatenation, any text node whose content is a zero-length string is deleted from the content sequence.
2. If the content sequence contains a document node, the document node is replaced in the content sequence by its children.
3. If the content sequence contains an attribute node following a node that is not an attribute node, a type error is raised .
4. The properties of the newly constructed element node are determined as follows:
   A. `node-name` is the expanded QName resulting from processing the specified QName or name expression, as described above.
   B. `parent` is empty.
   C. `attributes` consist of all the attribute nodes in the content sequence, in implementation-dependent order. Note that the `parent` property of each of these attribute nodes has been set to

the newly constructed element node. If two or more attributes have the same `node-name`, a dynamic error is raised . If an attribute named `xml:space` has a value other than `preserve` or `default`, a dynamic error MAY be raised .

D. `children` consist of all the element, text, comment, and processing instruction nodes in the content sequence. Note that the `parent` property of each of these nodes has been set to the newly constructed element node.

E. `base-uri` is set to the following value:

   i. If the constructed node has an attribute named `xml:base`, then the value of this attribute, resolved if it is relative against the base URI in the static context. The value of the `xml:base` attribute is normalized as described in [XML Base].

   ii. Otherwise, the value of the base URI in the static context.

F. `in-scope-namespaces` are computed as described in § 3.7.4 – In-scope Namespaces of a Constructed Element on page 71.

G. The `nilled` property is `false`.

H. The `string-value` property is equal to the concatenated contents of the text-node descendants in document order.

I. The `typed-value` property is equal to the `string-value` property, as an instance of `xs:untypedAtomic`.

J. If construction mode in the static context is `strip`, the `type-name` property is `xs:untyped`. On the other hand, if construction mode is `preserve`, the `type-name` property is `xs:anyType`.

K. The `is-id` and `is-idrefs` properties are set to `false`.

A computed element constructor might be used to make a modified copy of an existing element. For example, if the variable `$e` is bound to an element with numeric content, the following constructor might be used to create a new element with the same name and attributes as `$e` and with numeric content equal to twice the value of `$e`:

```
element {fn:node-name($e)}
   {$e/@*, 2 * fn:data($e)}
```

In this example, if `$e` is bound by the expression `let $e := <length units="inches">{5}</length>`, then the result of the example expression is the element `<length units="inches">10</length>`.

The static type of the expression `fn:node-name($e)` is `xs:QName?`, denoting zero or one QName. Therefore, if the Static Typing Feature is in effect, the above example raises a static type error, since the name expression in a computed element constructor is required to return exactly one string or QName. In order to avoid the static type error, the name expression `fn:node-name($e)` could be rewritten as `fn:exactly-one(fn:node-name($e))`. If the Static Typing Feature is not in effect, the example can be successfully evaluated as written, provided that `$e` is bound to exactly one element node with numeric content.

One important purpose of computed constructors is to allow the name of a node to be computed. We will illustrate this feature by an expression that translates the name of an element from one language to another. Suppose that the variable `$dict` is bound to a `dictionary` element containing a sequence of `entry` elements, each of which encodes translations for a specific word. Here is an example entry that encodes the German and Italian variants of the word "address":

```
<entry word="address">
   <variant xml:lang="de">Adresse</variant>
   <variant xml:lang="it">indirizzo</variant>
</entry>
```

Suppose further that the variable `$e` is bound to the following element:

```
<address>123 Roosevelt Ave. Flushing, NY 11368</address>
```

Then the following expression generates a new element in which the name of `$e` has been translated into Italian and the content of `$e` (including its attributes, if any) has been preserved. The first enclosed expression after the `element` keyword generates the name of the element, and the second enclosed expression generates the content and attributes:

```
element
  {$dict/entry[@word=name($e)]/variant[@xml:lang="it"]}
  {$e/@*, $e/node()}
```

The result of this expression is as follows:

```
<indirizzo>123 Roosevelt Ave. Flushing, NY 11368</indirizzo>
```

As in the previous example, if the Static Typing Feature is in effect, the enclosed expression that computes the element name in the above computed element constructor must be wrapped in a call to the `fn:exactly-one` function in order to avoid a static type error.

Additional examples of computed element constructors can be found in Appendix I.4 – Recursive Transformations on page 148.

3.7.3.2. Computed Attribute Constructors

[97] CompAttrConstructor ::= "attribute" (**QName** | ("{" **Expr** "}")) "{" **Expr**? "}"

A computed attribute constructor creates a new attribute node, with its own node identity.

If the keyword `attribute` is followed by a QName, that QName is expanded using the statically known namespaces, and the resulting expanded QName (including its prefix) is used as the `node-name` property of the constructed attribute node. If expansion of the QName is not successful, a static error is raised .

If the keyword `attribute` is followed by a name expression, the name expression is processed as follows:

1. Atomization is applied to the result of the name expression. If the result of atomization is not a single atomic value of type `xs:QName`, `xs:string`, or `xs:untypedAtomic`, a type error is raised .
2. If the atomized value of the name expression is of type `xs:QName`, that expanded QName (including its prefix) is used as the `node-name` property of the constructed attribute node.
3. If the atomized value of the name expression is of type `xs:string` or `xs:untypedAtomic`, that value is converted to an expanded QName. If the string value contains a namespace prefix, that prefix is resolved to a namespace URI using the statically known namespaces. If the string value contains no namespace prefix, it is treated as a local name in no namespace. The resulting expanded QName (including its prefix) is used as the `node-name` property of the constructed attribute. If conversion of the atomized name expression to an expanded QName is not successful, a dynamic error is raised .

The `node-name` property of the constructed attribute (an expanded QName) is checked as follows: If its URI part is `http://www.w3.org/2000/xmlns/` (corresponding to namespace prefix `xmlns`) or if it is in no namespace and its local name is `xmlns`, a dynamic error is raised.

The content expression of a computed attribute constructor is processed as follows:

1. Atomization is applied to the result of the content expression, converting it to a sequence of atomic values. (If the content expression is absent, the result of this step is an empty sequence.)
2. If the result of atomization is an empty sequence, the value of the attribute is the zero-length string. Otherwise, each atomic value in the atomized sequence is cast into a string.
3. The individual strings resulting from the previous step are merged into a single string by concatenating them with a single space character between each pair. The resulting string becomes the `string-value` property of the new attribute node. The type annotation (`type-name` property) of the new attribute node is `xs:untypedAtomic`. The `typed-value` property of the attribute node is the same as its `string-value`, as an instance of `xs:untypedAtomic`.
4. The `parent` property of the attribute node is set to empty.
5. If the attribute name is `xml:id`, then `xml:id` processing is performed as defined in [XML ID]. This ensures that the attribute node has the type `xs:ID` and that its value is properly normalized. If an error is encountered during `xml:id` processing, an implementation MAY raise a dynamic error .
6. If the attribute name is `xml:id`, the `is-id` property of the resulting attribute node is set to `true`; otherwise the `is-id` property is set to `false`. The `is-idrefs` property of the attribute node is unconditionally set to `false`.
7. If the attribute name is `xml:space` and the attribute value is other than `preserve` or `default`, a dynamic error MAY be raised .

- Example:

```
attribute size {4 + 3}
```

The string value of the `size` attribute is "`7`" and its type is `xs:untypedAtomic`.

- Example:

```
attribute
   { if ($sex = "M") then "husband" else "wife" }
   { <a>Hello</a>, 1 to 3, <b>Goodbye</b> }
```

The name of the constructed attribute is either `husband` or `wife`. Its string value is "`Hello 1 2 3 Goodbye`".

3.7.3.3. Document Node Constructors

[98] CompDocConstructor ::= "document" "{" **Expr** "}"

All document node constructors are computed constructors. The result of a document node constructor is a new document node, with its own node identity.

A document node constructor is useful when the result of a query is to be a document in its own right. The following example illustrates a query that returns an XML document containing a root element named `author-list`:

```
document
   {
      <author-list>
         {fn:doc("bib.xml")/bib/book/author}
      </author-list>
   }
```

The content expression of a document node constructor is processed in exactly the same way as an enclosed expression in the content of a direct element constructor, as described in Step 1e of § 3.7.1.3 – Content on page 59. The result of processing the content expression is a sequence of nodes called the *content sequence*. Processing of the document node constructor then proceeds as follows:

1. Adjacent text nodes in the content sequence are merged into a single text node by concatenating their contents, with no intervening blanks. After concatenation, any text node whose content is a zero-length string is deleted from the content sequence.
2. If the content sequence contains a document node, the document node is replaced in the content sequence by its children.
3. If the content sequence contains an attribute node, a type error is raised .
4. The properties of the newly constructed document node are determined as follows:
   A. `base-uri` is taken from base URI in the static context. If no base URI is defined in the static context, the `base-uri` property is empty.
   B. `children` consist of all the element, text, comment, and processing instruction nodes in the content sequence. Note that the `parent` property of each of these nodes has been set to the newly constructed document node.
   C. The `unparsed-entities` and `document-uri` properties are empty.
   D. The `string-value` property is equal to the concatenated contents of the text-node descendants in document order.
   E. The `typed-value` property is equal to the `string-value` property, as an instance of `xs:untypedAtomic`.

No validation is performed on the constructed document node. The [XML 1.0] rules that govern the structure of an XML document (for example, the document node must have exactly one child that is an element node) are not enforced by the XQuery document node constructor.

3.7.3.4. Text Node Constructors

[99] CompTextConstructor ::= "text" "{" **Expr** "}"

All text node constructors are computed constructors. The result of a text node constructor is a new text node, with its own node identity.

The content expression of a text node constructor is processed as follows:

1. Atomization is applied to the value of the content expression, converting it to a sequence of atomic values.
2. If the result of atomization is an empty sequence, no text node is constructed. Otherwise, each atomic value in the atomized sequence is cast into a string.

3. The individual strings resulting from the previous step are merged into a single string by concatenating them with a single space character between each pair. The resulting string becomes the `content` property of the constructed text node.

The `parent` property of the constructed text node is set to empty.

It is possible for a text node constructor to construct a text node containing a zero-length string. However, if used in the content of a constructed element or document node, such a text node will be deleted or merged with another text node.

The following example illustrates a text node constructor:

```
text {"Hello"}
```

3.7.3.5. Computed Processing Instruction Constructors

[100] CompPIConstructor ::= "processing-instruction" (**NCName** | ("{" **Expr** "}")) "{" **Expr**? "}"

A computed processing instruction constructor (**CompPIConstructor**) constructs a new processing instruction node with its own node identity.

If the keyword `processing-instruction` is followed by an NCName, that NCName is used as the `target` property of the constructed node. If the keyword `processing-instruction` is followed by a name expression, the name expression is processed as follows:

1. Atomization is applied to the value of the name expression. If the result of atomization is not a single atomic value of type `xs:NCName`, `xs:string`, or `xs:untypedAtomic`, a type error is raised .
2. If the atomized value of the name expression is of type `xs:string` or `xs:untypedAtomic`, that value is cast to the type `xs:NCName`. If the value cannot be cast to `xs:NCName`, a dynamic error is raised .
3. The resulting NCName is then used as the `target` property of the newly constructed processing instruction node. However, a dynamic error is raised if the NCName is equal to "`XML`" (in any combination of upper and lower case) .

The content expression of a computed processing instruction constructor is processed as follows:

1. Atomization is applied to the value of the content expression, converting it to a sequence of atomic values. (If the content expression is absent, the result of this step is an empty sequence.)
2. If the result of atomization is an empty sequence, it is replaced by a zero-length string. Otherwise, each atomic value in the atomized sequence is cast into a string. If any of the resulting strings contains the string "`?>`", a dynamic error is raised.
3. The individual strings resulting from the previous step are merged into a single string by concatenating them with a single space character between each pair. Leading whitespace is removed from the resulting string. The resulting string then becomes the `content` property of the constructed processing instruction node.

The remaining properties of the new processing instruction node are determined as follows:

1. The `parent` property is empty.
2. The `base-uri` property is empty.

The following example illustrates a computed processing instruction constructor:

```
let $target := "audio-output",
    $content := "beep"
return processing-instruction {$target} {$content}
```

The processing instruction node constructed by this example might be serialized as follows:

```
<?audio-output beep?>
```

3.7.3.6. Computed Comment Constructors

[101] CompCommentConstructor ::= "comment" "{" **Expr** "}"

A computed comment constructor (**CompCommentConstructor**) constructs a new comment node with its own node identity. The content expression of a computed comment constructor is processed as follows:

1. Atomization is applied to the value of the content expression, converting it to a sequence of atomic values.
2. If the result of atomization is an empty sequence, it is replaced by a zero-length string. Otherwise, each atomic value in the atomized sequence is cast into a string.
3. The individual strings resulting from the previous step are merged into a single string by concatenating them with a single space character between each pair. The resulting string becomes the `content` property of the constructed comment node.
4. It is a dynamic error if the result of the content expression of a computed comment constructor contains two adjacent hyphens or ends with a hyphen.

The `parent` property of the constructed comment node is set to empty.

The following example illustrates a computed comment constructor:

```
let $homebase := "Houston"
return comment {fn:concat($homebase, ", we have a problem.")}
```

The comment node constructed by this example might be serialized as follows:

```
<!--Houston, we have a problem.-->
```

3.7.4. In-scope Namespaces of a Constructed Element

An element node constructed by a direct or computed element constructor has an in-scope namespaces property that consists of a set of namespace bindings. The in-scope namespaces of an element node may affect the way the node is serialized (see § 2.2.4 – Serialization on page 12), and may also affect the behavior of certain functions that operate on nodes, such as `fn:name`. 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. Also note that one of the namespace bindings in the in-scope namespaces may have no prefix (denoting the default namespace for the given element). The in-scope namespaces of a constructed element node consist of the following namespace bindings:

- A namespace binding is created for each namespace declared in the current element constructor by a namespace declaration attribute.
- A namespace binding is created for each namespace that is declared in a namespace declaration attribute of an enclosing direct element constructor and not overridden by the current element constructor or an intermediate constructor.

- A namespace binding is always created to bind the prefix `xml` to the namespace URI `http://www.w3.org/XML/1998/namespace`.
- For each namespace used in the name of the constructed element or in the names of its attributes, a namespace binding must exist. If a namespace binding does not already exist for one of these namespaces, a new namespace binding is created for it. If the name of the node includes a prefix, that prefix is used in the namespace binding; if the name has no prefix, then a binding is created for the empty prefix. If this would result in a conflict, because it would require two different bindings of the same prefix, then the prefix used in the node name is changed to an arbitrary implementation-dependent prefix that does not cause such a conflict, and a namespace binding is created for this new prefix.

Copy-namespaces mode does not affect the namespace bindings of a newly constructed element node. It applies only to existing nodes that are copied by a constructor expression.

The following query serves as an example:

```
declare namespace p="http://example.com/ns/p";
declare namespace q="http://example.com/ns/q";
declare namespace f="http://example.com/ns/f";

<p:a q:b="{f:func(2)}" xmlns:r="http://example.com/ns/r"/>
```

The in-scope namespaces of the resulting `p:a` element consists of the following namespace bindings:

- `p = "http://example.com/ns/p"`
- `q = "http://example.com/ns/q"`
- `r = "http://example.com/ns/r"`
- `xml = "http://www.w3.org/XML/1998/namespace"`

The namespace bindings for `p` and `q` are added to the result element because their respective namespaces are used in the names of the element and its attributes. The namespace binding `r="http://example.com/ns/r"` is added to the in-scope namespaces of the constructed element because it is defined by a namespace declaration attribute, even though it is not used in a name.

No namespace binding corresponding to `f="http://example.com/ns/f"` is created, because the namespace prefix `f` appears only in the query prolog and is not used in an element or attribute name of the constructed node. This namespace binding does not appear in the query result, even though it is present in the statically known namespaces and is available for use during processing of the query.

Note that the following constructed element, if nested within a `validate` expression, cannot be validated:

```
<p xsi:type="xs:integer">3</p>
```

The constructed element will have namespace bindings for the prefixes `xsi` (because it is used in a name) and `xml` (because it is defined for every constructed element node). During validation of the constructed element, the validator will be unable to interpret the namespace prefix `xs` because it is has no namespace binding. Validation of this constructed element could be made possible by providing a namespace declaration attribute, as in the following example:

```
<p xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xsi:type="xs:integer">3</p>
```

3.8. FLWOR Expressions

XQuery provides a feature called a FLWOR expression that supports iteration and binding of variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents and for restructuring data. The name FLWOR, pronounced "flower", is suggested by the keywords `for`, `let`, `where`, `order by`, and `return`.

[102] FLWORExpr ::= (**ForClause** | **LetClause**)+ **WhereClause**? **OrderByClause**? "return" **ExprSingle**

[103] ForClause ::= "for" "$" **VarName TypeDeclaration**? **PositionalVar**? "in" **ExprSingle** ("," "$" **VarName TypeDeclaration**? **PositionalVar**? "in" **ExprSingle**)*

[104] LetClause ::= "let" "$" **VarName TypeDeclaration**? ":=" **ExprSingle** ("," "$" **VarName TypeDeclaration**? ":=" **ExprSingle**)*

[105] TypeDeclaration ::= "as" **SequenceType**

[106] PositionalVar ::= "at" "$" **VarName**

[107] WhereClause ::= "where" **ExprSingle**

[108] OrderByClause ::= (("order" "by") | ("stable" "order" "by")) **OrderSpecList**

[109] OrderSpecList ::= **OrderSpec** ("," **OrderSpec**)*

[110] OrderSpec ::= **ExprSingle OrderModifier**

[111] OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest" | "least"))? ("collation" **URILiteral**)?

The `for` and `let` clauses in a FLWOR expression generate an ordered sequence of tuples of bound variables, called the *tuple stream*. The optional `where` clause serves to filter the tuple stream, retaining some tuples and discarding others. The optional `order by` clause can be used to reorder the tuple stream. The `return` clause constructs the result of the FLWOR expression. The `return` clause is evaluated once for every tuple in the tuple stream, after filtering by the `where` clause, using the variable bindings in the respective tuples. The result of the FLWOR expression is an ordered sequence containing the results of these evaluations, concatenated as if by the comma operator.

The following example of a FLWOR expression includes all of the possible clauses. The `for` clause iterates over all the departments in an input document, binding the variable `$d` to each department number in turn. For each binding of `$d`, the `let` clause binds variable `$e` to all the employees in the given department, selected from another input document. The result of the `for` and `let` clauses is a tuple stream in which each tuple contains a pair of bindings for `$d` and `$e` (`$d` is bound to a department number and `$e` is bound to a set of employees in that department). The `where` clause filters the tuple stream by keeping only those binding-pairs that represent departments having at least ten employees. The `order by` clause orders the surviving tuples in descending order by the average salary of the employees in the department. The `return` clause constructs a new `big-dept` element for each surviving tuple, containing the department number, headcount, and average salary.

```
for $d in fn:doc("depts.xml")/depts/deptno
let $e := fn:doc("emps.xml")/emps/emp[deptno = $d]
where fn:count($e) >= 10
order by fn:avg($e/salary) descending
return
   <big-dept>
```

```
      {
      $d,
      <headcount>{fn:count($e)}</headcount>,
      <avgsal>{fn:avg($e/salary)}</avgsal>
      }
   </big-dept>
```

The clauses in a FLWOR expression are described in more detail below.

3.8.1. For and Let Clauses

The purpose of the `for` and `let` clauses in a FLWOR expression is to produce a tuple stream in which each tuple consists of one or more bound variables.

The simplest example of a `for` clause contains one variable and an associated expression. The value of the expression associated with a variable in a `for` clause is called the *binding sequence* for that variable. The `for` clause iterates over the items in the binding sequence, binding the variable to each item in turn. If ordering mode is `ordered`, the resulting sequence of variable bindings is ordered according to the order of values in the binding sequence; otherwise the ordering of the variable bindings is implementation-dependent.

A `for` clause may also contain multiple variables, each with an associated expression whose value is the binding sequence for that variable. In this case, the `for` clause iterates each variable over its binding sequence. The resulting tuple stream contains one tuple for each combination of values in the respective binding sequences. If ordering mode is `ordered`, the order of the tuple stream is determined primarily by the order of the binding sequence of the leftmost variable, and secondarily by the binding sequences of the other variables, working from left to right. Otherwise, the ordering of the variable bindings is implementation-dependent.

A `let` clause may also contain one or more variables, each with an associated expression. Unlike a `for` clause, however, a `let` clause binds each variable to the result of its associated expression, without iteration. The variable bindings generated by `let` clauses are added to the binding tuples generated by the `for` clauses. If there are no `for` clauses, the `let` clauses generate one tuple containing all the variable bindings.

Although `for` and `let` clauses both bind variables, the manner in which variables are bound is quite different, as illustrated by the following examples. The first example uses a `let` clause:

```
let $s := (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

The variable `$s` is bound to the result of the expression `(<one/>, <two/>, <three/>)`. Since there are no `for` clauses, the `let` clause generates one tuple that contains the binding of `$s`. The `return` clause is invoked for this tuple, creating the following output:

```
<out>
   <one/>
   <two/>
   <three/>
</out>
```

The next example is a similar query that contains a `for` clause instead of a `let` clause:

```
for $s in (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

In this example, the variable `$s` iterates over the given expression. If ordering mode is `ordered`, `$s` is first bound to `<one/>`, then to `<two/>`, and finally to `<three/>`. One tuple is generated for each of these bindings, and the `return` clause is invoked for each tuple, creating the following output:

```
<out>
   <one/>
</out>
<out>
   <two/>
</out>
<out>
   <three/>
</out>
```

The following example illustrates how binding tuples are generated by a `for` clause that contains multiple variables when ordering mode is `ordered`.

```
for $i in (1, 2), $j in (3, 4)
```

The tuple stream generated by the above `for` clause is as follows:

```
($i = 1, $j = 3)
($i = 1, $j = 4)
($i = 2, $j = 3)
($i = 2, $j = 4)
```

If ordering mode were `unordered`, the `for` clause in the above example would generate the same tuple stream but the order of the tuples would be implementation-dependent.

The scope of a variable bound in a `for` or `let` clause comprises all subexpressions of the containing FLWOR 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 bindings in `for` and `let` clauses may reference variables that were bound in earlier clauses, or in earlier bindings in the same clause of the FLWOR expression:

```
for $x in $w, $a in f($x)
let $y := g($a)
for $z in p($x, $y)
return q($x, $y, $z)
```

The `for` and `let` clauses of a given FLWOR expression may bind the same variable name more than once. In this case, each new binding occludes the previous one, which becomes inaccessible in the remainder of the FLWOR expression.

Each variable bound in a `for` or `let` clause may have an optional *type declaration*, which is a type declared using the syntax in § 2.5.3 – SequenceType Syntax on page 24. If the type of a value bound to the variable does not match the declared type according to the rules for SequenceType matching, a type error is raised . For example, the following expression raises a type error because the variable `$salary` has a type declaration that is not satisfied by the value that is bound to the variable:

```
let $salary as xs:decimal :=  "cat"
return $salary * 2
```

Each variable bound in a `for` clause may have an associated *positional variable* that is bound at the same time. The name of the positional variable is preceded by the keyword `at`. The positional variable always has an implied type of `xs:integer`. As a variable iterates over the items in its binding sequence, its positional variable iterates over the integers that represent the ordinal positions of those items in the binding sequence, starting with 1. The expanded QName of a positional variable must be distinct from the expanded QName of the variable with which it is associated .

Positional variables are illustrated by the following `for` clause:

```
for $car at $i in ("Ford", "Chevy"),
    $pet at $j in ("Cat", "Dog")
```

If ordering mode is `ordered`, the tuple stream generated by the above `for` clause is as follows:

```
($i = 1, $car = "Ford", $j = 1, $pet = "Cat")
($i = 1, $car = "Ford", $j = 2, $pet = "Dog")
($i = 2, $car = "Chevy", $j = 1, $pet = "Cat")
($i = 2, $car = "Chevy", $j = 2, $pet = "Dog")
```

If ordering mode is `unordered`, the order of the tuple stream is implementation-dependent. In addition, if a `for` clause contains subexpressions that are affected by ordering mode, the association of positional variables with items returned by these subexpressions is implementation-dependent if ordering mode is `unordered`.

3.8.2. Where Clause

The optional `where` clause serves as a filter for the tuples of variable bindings generated by the `for` and `let` clauses. The expression in the `where` clause, called the *where-expression*, is evaluated once for each of these tuples. If the effective boolean value of the where-expression is `true`, the tuple is retained and its variable bindings are used in an execution of the `return` clause. If the effective boolean value of the where-expression is `false`, the tuple is discarded. The effective boolean value of an expression is defined in § 2.4.3 – Effective Boolean Value on page 19.

The following expression illustrates how a `where` clause might be applied to a *positional variable* in order to perform sampling on an input sequence. This expression approximates the average value in a sequence by sampling one value out of each one hundred input values.

```
fn:avg(for $x at $i in $inputvalues
    where $i mod 100 = 0
    return $x)
```

3.8.3. Order By and Return Clauses

The `return` clause of a FLWOR expression is evaluated once for each tuple in the tuple stream, and the results of these evaluations are concatenated, as if by the comma operator, to form the result of the FLWOR expression.

If no `order by` clause is present, the order of the tuple stream is determined by the `for` and `let` clauses and by ordering mode. If an `order by` clause is present, it reorders the tuples in the tuple stream into a new, value-based order. In either case, the resulting order determines the order in which the `return` clause is evaluated, once for each tuple, using the variable bindings in the respective tuples. Note that

ordering mode has no effect on a FLWOR expression if an `order by` clause is present, since `order by` takes precedence over ordering mode.

An `order by` clause contains one or more ordering specifications, called **orderspecs**, as shown in the grammar above. For each tuple in the tuple stream, after filtering by the `where` clause, the orderspecs are evaluated, using the variable bindings in that tuple. The relative order of two tuples is determined by comparing the values of their orderspecs, working from left to right until a pair of unequal values is encountered. If an orderspec specifies a collation, that collation is used in comparing values of type `xs:string`, `xs:anyURI`, or types derived from them (otherwise, the default collation is used). If an orderspec specifies a collation by a relative URI, that relative URI is resolved to an absolute URI using the base URI in the static context. If an orderspec specifies a collation that is not found in statically known collations, an error is raised .

The process of evaluating and comparing the orderspecs is based on the following rules:

- Atomization is applied to the result of the expression in each orderspec. If the result of atomization is neither a single atomic value nor an empty sequence, a type error is raised .
- If the value of an orderspec has the dynamic type `xs:untypedAtomic` (such as character data in a schemaless document), it is cast to the type `xs:string`.

  ☞ Consistently treating untyped values as strings enables the sorting process to begin without complete knowledge of the types of all the values to be sorted.

- All the non-empty orderspec values must be convertible to a common type by subtype substitution and/or type promotion. The ordering is performed in the least common type that has a `gt` operator. If two or more non-empty orderspec values are not convertible to a common type that has a `gt` operator, a type error is raised .
  - Example: The orderspec values include a value of type `hatsize`, which is derived from `xs:integer`, and a value of type `shoesize`, which is derived from `xs:decimal`. The least common type reachable by subtype substitution and type promotion is `xs:decimal`.
  - Example: The orderspec values include a value of type `xs:string` and a value of type `xs:anyURI`. The least common type reachable by subtype substitution and type promotion is `xs:string`.

When two orderspec values are compared to determine their relative position in the ordering sequence, the *greater-than* relationship is defined as follows:

- When the orderspec specifies `empty least`, a value W is considered to be *greater-than* a value V if one of the following is true:
  - V is an empty sequence and W is not an empty sequence.
  - V is `NaN`, and W is neither `NaN` nor an empty sequence.
  - No collation is specified, and W `gt` V is true.
  - A specific collation C is specified, and `fn:compare(V, W, C)` is less than zero.
- When the orderspec specifies `empty greatest`, a value W is considered to be *greater-than* a value V if one of the following is true:
  - W is an empty sequence and V is not an empty sequence.
  - W is `NaN`, and V is neither `NaN` nor an empty sequence.

  - No collation is specified, and W `gt` V is true.
  - A specific collation C is specified, and `fn:compare(V, W, C)` is less than zero.
- When the orderspec specifies neither `empty least` nor `empty greatest`, the default order for empty sequences in the static context determines whether the rules for `empty least` or `empty greatest` are used.

If T1 and T2 are two tuples in the tuple stream, and V1 and V2 are the first pair of values encountered when evaluating their orderspecs from left to right for which one value is *greater-than* the other (as defined above), then:

1. If V1 is *greater-than* V2: If the orderspec specifies `descending`, then T1 precedes T2 in the tuple stream; otherwise, T2 precedes T1 in the tuple stream.
2. If V2 is *greater-than* V1: If the orderspec specifies `descending`, then T2 precedes T1 in the tuple stream; otherwise, T1 precedes T2 in the tuple stream.

If neither V1 nor V2 is *greater-than* the other for any pair of orderspecs for tuples T1 and T2, the following rules apply.

1. If `stable` is specified, the original order of T1 and T2 is preserved in the tuple stream.
2. If `stable` is not specified, the order of T1 and T2 in the tuple stream is implementation-dependent.

If two orderspecs return the special floating-point values positive and negative zero, neither of these values is *greater-than* the other, since `+0.0 gt -0.0` and `-0.0 gt +0.0` are both `false`.

An `order by` clause makes it easy to sort the result of a FLWOR expression, even if the sort key is not included in the result of the expression. For example, the following expression returns employee names in descending order by salary, without returning the actual salaries:

```
for $e in $employees
order by $e/salary descending
return $e/name
```

Since the `order by` clause in a FLWOR expression is the only facility provided by XQuery for specifying a value ordering, a FLWOR expression must be used in some queries where iteration would not otherwise be necessary. For example, a list of books with price less than 100 might be obtained by a simple path expression such as `$books/book[price < 100]`. But if these books are to be returned in alphabetic order by title, the query must be expressed as follows:

```
for $b in $books/book[price < 100]
order by $b/title
return $b
```

The following example illustrates an `order by` clause that uses several options. It causes a collection of books to be sorted in primary order by title, and in secondary descending order by price. A specific collation is specified for the title ordering, and in the ordering by price, books with no price are specified to occur last (as though they have the least possible price). Whenever two books with the same title and price occur, the keyword `stable` indicates that their input order is preserved.

```
for $b in $books/book
stable order by $b/title
      collation "http://www.example.org/collations/fr-ca",
```

```
   $b/price descending empty least
return $b
```

Parentheses are helpful in `return` clauses that contain comma operators, since FLWOR expressions have a higher precedence than the comma operator. For instance, the following query raises an error because after the comma, `$j` is no longer within the FLWOR expression, and is an undefined variable:

```
let $i := 5,
    $j := 20 * $i
return $i, $j
```

Parentheses can be used to bring `$j` into the `return` clause of the FLWOR expression, as the programmer probably intended:

```
let $i := 5,
    $j := 20 * $i
return ($i, $j)
```

3.8.4. Example

The following example illustrates how FLWOR expressions can be nested, and how ordering can be specified at multiple levels of an element hierarchy. The example query inverts a document hierarchy to transform a bibliography into an author list. The input (bound to the variable `$bib`) is a `bib` element containing a list of books, each of which in turn contains a list of authors. 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 query 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. The `fn:distinct-values` function is used to eliminate duplicates (by value) from a list of author nodes. The author list, and the lists of books published by each author, are returned in alphabetic order using the default collation.

```
<authlist>
 {
   for $a in fn:distinct-values($bib/book/author)
   order by $a
   return
     <author>
        <name> {$a} </name>
        <books>
          {
            for $b in $bib/book[author = $a]
            order by $b/title
            return $b/title
          }
        </books>
     </author>
 }
</authlist>
```

The result of the above expression is as follows:

```
<authlist>
   <author>
      <name>Abiteboul</name>
      <books>
         <title>Data on the Web</title>
      </books>
   </author>
   <author>
      <name>Buneman</name>
      <books>
         <title>Data on the Web</title>
      </books>
   </author>
   <author>
      <name>Stevens</name>
      <books>
         <title>Advanced Programming
                in the Unix Environment</title>
         <title>TCP/IP Illustrated</title>
      </books>
   </author>
   <author>
      <name>Suciu</name>
      <books>
         <title>Data on the Web</title>
      </books>
```

```
   </author>
</authlist>
```

3.9. Ordered and Unordered Expressions

[112] OrderedExpr ::= "ordered" "{" **Expr** "}"

[113] UnorderedExpr ::= "unordered" "{" **Expr** "}"

The purpose of `ordered` and `unordered` expressions is to set the ordering mode in the static context to `ordered` or `unordered` for a certain region in a query. The specified ordering mode applies to the expression nested inside the curly braces. For expressions where the ordering of the result is not significant, a performance advantage may be realized by setting the ordering mode to `unordered`, thereby granting the system flexibility to return the result in the order that it finds most efficient.

Ordering mode affects the behavior of path expressions that include a "`/`" or "`//`" operator or an axis step; `union`, `intersect`, and `except` expressions; the `fn:id` and `fn:idref` functions; and FLWOR expressions that have no `order by` clause. If ordering mode is `ordered`, node sequences returned by path expressions, `union`, `intersect`, and `except` expressions, and the `fn:id` and `fn:idref` functions are in document order; otherwise the order of these return sequences is implementation-dependent. The effect of ordering mode on FLWOR expressions is described in § 3.8 – FLWOR Expressions on page 73. Ordering mode has no effect on duplicate elimination.

In a region of a query where ordering mode is `unordered`, the result of an expression may be nondeterministic if the expression invokes certain functions that are affected by the ordering of node sequences. These functions include `fn:position`, `fn:last`, `fn:index-of`, `fn:insert-before`, `fn:remove`, `fn:reverse`, and `fn:subsequence`. Also, within a path expression in an unordered region, numeric predicates are nondeterministic. For example, in an ordered region, the path expression `(//a/b)[5]` will return the fifth qualifying `b`-element in document order. In an unordered region, the same expression will return an implementation-dependent qualifying `b`-element.

The `fn:id` and `fn:idref` functions are described in [XQuery 1.0 and XPath 2.0 Functions and Operators] as returning their results in document order. Since ordering mode is a feature of XQuery, relaxation of the ordering requirement for function results when ordering mode is `unordered` is a feature of XQuery rather than of the functions themselves.

The use of an `unordered` expression is illustrated by the following example, which joins together two documents named `parts.xml` and `suppliers.xml`. The example returns the part numbers of red parts, paired with the supplier numbers of suppliers who supply these parts. If an `unordered` expression were not used, the resulting list of (part number, supplier number) pairs would be required to have an ordering that is controlled primarily by the document order of `parts.xml` and secondarily by the document order of `suppliers.xml`. However, this might not be the most efficient way to process the query if the ordering of the result is not important. An XQuery implementation might be able to process the query more efficiently by using an index to find the red parts, or by using `suppliers.xml` rather than `parts.xml` to control the primary ordering of the result. The `unordered` expression gives the query evaluator freedom to make these kinds of optimizations.

```
unordered {
  for $p in fn:doc("parts.xml")/parts/part[color = "Red"],
      $s in fn:doc("suppliers.xml")/suppliers/supplier
  where $p/suppno = $s/suppno
```

```
  return
    <ps>
       { $p/partno, $s/suppno }
    </ps>
}
```

In addition to `ordered` and `unordered` expressions, XQuery provides a function named `fn:unordered` that operates on any sequence of items and returns the same sequence in a nondeterministic order. A call to the `fn:unordered` function may be thought of as giving permission for the argument expression to be materialized in whatever order the system finds most efficient. The `fn:unordered` function relaxes ordering only for the sequence that is its immediate operand, whereas an `unordered` expression sets the ordering mode for its operand expression and all nested expressions.

3.10. Conditional Expressions

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

[114] 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 on page 19.

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
```

3.11. Quantified Expressions

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

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

[16] TypeDeclaration ::= "as" **SequenceType**

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 on page 19. 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.

Each variable bound in an in-clause of a quantified expression may have an optional **type declaration**. If the type of a value bound to the variable does not match the declared type according to the rules for SequenceType matching, a type error is raised .

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
```

- This quantified expression contains a **type declaration** that is not satisfied by every item in the test expression. If the Static Typing Feature is implemented, this expression raises a type error during the static analysis phase. Otherwise, the expression may either return `true` or raise a type error during the dynamic evaluation phase.

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

3.12. Expressions on SequenceTypes

In addition to their use in function parameters and results, sequence types are used in `instance of`, `typeswitch`, `cast`, `castable`, and `treat` expressions.

3.12.1. Instance Of

[17] 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`.

- `<a>{5}</a> instance of xs:integer`

  This example returns `false` because the given value is an element rather than an integer.

- `(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 .

3.12.2. Typeswitch

[18] TypeswitchExpr ::= "typeswitch" "(" **Expr** ")" **CaseClause**+ "default" ("$" **VarName**)? "return" **ExprSingle**

[19] CaseClause ::= "case" ("$" **VarName** "as")? **SequenceType** "return" **ExprSingle**

The *typeswitch* expression chooses one of several expressions to evaluate based on the dynamic type of an input value.

In a `typeswitch` expression, the `typeswitch` keyword is followed by an expression enclosed in parentheses, called the *operand expression*. This is the expression whose type is being tested. The remainder of the `typeswitch` expression consists of one or more `case` clauses and a `default` clause.

Each `case` clause specifies a **SequenceType** followed by a `return` expression. The *effective case* in a `typeswitch` expression is the first `case` clause such that the value of the operand expression matches the **SequenceType** in the `case` clause, using the rules of SequenceType matching. The value of the `typeswitch` expression is the value of the `return` expression in the effective case. If the value of the operand expression does not match any **SequenceType** named in a `case` clause, the value of the `typeswitch` expression is the value of the `return` expression in the `default` clause.

In a `case` or `default` clause, if the value to be returned depends on the value of the operand expression, the clause must specify a variable name. Within the `return` expression of the `case` or `default` clause, this variable name is bound to the value of the operand expression. Inside a `case` clause, the static type of the variable is the **SequenceType** named in the `case` clause. Inside a `default` clause, the static type of the variable is the same as the static type of the operand expression. If the value to be returned by a `case` or `default` clause does not depend on the value of the operand expression, the clause need not specify a variable.

The scope of a variable binding in a `case` or `default` clause comprises that clause. It is not an error for more than one `case` or `default` clause in the same `typeswitch` expression to bind variables with the same name.

A special rule applies to propagation of dynamic errors by `typeswitch` expressions. A `typeswitch` expression ignores (does not raise) any dynamic errors encountered in `case` clauses other than the effective case. Dynamic errors encountered in the `default` clause are raised only if there is no effective case.

The following example shows how a `typeswitch` expression might be used to process an expression in a way that depends on its dynamic type.

```
typeswitch($customer/billing-address)
   case $a as element(*, USAddress) return $a/state
   case $a as element(*, CanadaAddress) return $a/province
   case $a as element(*, JapanAddress) return $a/prefecture
   default return "unknown"
```

3.12.3. Cast

[120] CastExpr ::= **UnaryExpr** ("cast" "as" **SingleType**)?

[121] SingleType ::= **AtomicType** "?"?

Occasionally it is necessary to convert a value to a specific datatype. For this purpose, XQuery 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 . In addition, the target type cannot be `xs:NOTATION` or `xs:anyAtomicType` . 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 .
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 .
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]. 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 . 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].

      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 .

      

      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]); a dynamic error [err:FORG0001] is raised if the resulting lexical value does not satisfy the pattern facet of the target type. The lexical value is then converted to the value space of the target type using the schema-defined rules for the target type; a dynamic error [err:FORG0001] is raised if the resulting value does not satisfy all the facets of the target type.

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 .

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.

3.12.4. Castable

[122] CastableExpr ::= **CastExpr** ("castable" "as" **SingleType**)?

[123] SingleType ::= **AtomicType** "?"?

XQuery 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 . In addition, the target type cannot be `xs:NOTATION` or `xs:anyAtomicType` . The optional occurrence indicator "`?`" denotes that an empty sequence is permitted.

The expression `V castable as T` returns `true` if the value `V` can be successfully cast into the target type `T` by using a `cast` expression; otherwise it returns `false`. 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
```

☞ 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`.

3.12.5. 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?
```

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 is raised. This rule is consistent with the semantics of `cast` expressions for these types, as defined in § 3.12.3 – Cast on page 86.

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")
  ```

☞ 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". (See § 4.13 – Default Namespace Declaration on page 100 for how to undeclare the default element/type namespace).

  ```
  17 cast as apple
  ```

- By using a constructor function, if the default function namespace is "none". (See § 4.13 – Default Namespace Declaration on page 100 for how to undeclare the default function namespace).

  ```
  apple(17)
  ```

3.12.6. Treat

[24] TreatExpr ::= **CastableExpr** ("treat" "as" **SequenceType**)?

XQuery 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 . 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 .

3.13. Validate Expressions

[125] ValidateExpr ::= "validate" **ValidationMode**? "{" **Expr** "}"

[126] ValidationMode ::= "lax" | "strict"

A `validate` expression can be used to validate a document node or an element node with respect to the in-scope schema definitions, using the schema validation process defined in [XML Schema]. If the operand of a `validate` expression does not evaluate to exactly one document or element node, a type error is raised . In this specification, the node that is the operand of a `validate` expression is called the *operand node*.

A `validate` expression returns a new node with its own identity and with no parent. The new node and its descendants are given type annotations that are generated by applying a validation process to the operand node. In some cases, default values may also be generated by the validation process.

A `validate` expression may optionally specify a **validation mode**. The default **validation mode** is `strict`. The result of a `validate` expression is defined by the following rules.

1. If the operand node is a document node, its children must consist of exactly one element node and zero or more comment and processing instruction nodes, in any order; otherwise, a dynamic error is raised.
2. The operand node is converted to an XML Information Set ([XML Infoset]) according to the "Infoset Mapping" rules defined in [XQuery/XPath Data Model (XDM)]. Note that this process discards any existing type annotations.

3. Validity assessment is carried out on the root element information item of the resulting Infoset, using the in-scope schema definitions as the effective schema. The process of validation applies recursively to contained elements and attributes to the extent required by the effective schema. During validity assessment, the following special rules are in effect:

   A. If **validation mode** is `strict`, then there must be a top-level element declaration in the in-scope element declarations that matches the root element information item in the Infoset, and schema-validity assessment is carried out using that declaration in accordance with item 2 of [XML Schema] Part 1, section 5.2, "Assessing Schema-Validity." If there is no such element declaration, a dynamic error is raised .

   B. If **validation mode** is `lax`, then schema-validity assessment is carried out in accordance with item 3 of [XML Schema] Part 1, section 5.2, "Assessing Schema-Validity."

      If **validation mode** is `lax` and the root element information item has neither a top-level element declaration nor an `xsi:type` attribute, [XML Schema] defines the recursive checking of children and attributes as optional. During processing of an XQuery `validate` expression, this recursive checking is required.

   C. If the operand node is an element node, the validation rules named "Validation Root Valid (ID/IDREF)" and "Identity-constraint Satisfied" are not applied. This means that document-level constraints relating to uniqueness and referential integrity are not enforced.

   D. There is no check that the document contains unparsed entities whose names match the values of nodes of type `xs:ENTITY` or `xs:ENTITIES`.

   E. There is no check that the document contains notations whose names match the values of nodes of type `xs:NOTATION`.

Validity assessment is affected by the presence or absence of `xsi:type` attributes on the elements being validated, and may generate new information items such as default attributes.

4. The next step depends on **validation mode** and on the `validity` property of the root element information item in the PSVI that results from the validation process.

   A. If the `validity` property of the root element information item is `valid` (for any **validation mode**), or if **validation mode** is `lax` and the `validity` property of the root element information item is `notKnown`, the PSVI is converted back into an XDM instance as described in [XQuery/XPath Data Model (XDM)] Section 3.3, "Construction from a PSVI". The resulting node (a new node of the same kind as the operand node) is returned as the result of the `validate` expression.

   B. Otherwise, a dynamic error is raised .

The effect of these rules is as follows: If **validation mode** is `strict`, the validated element must have a top-level element declaration in the effective schema, and must conform to this declaration. If **validation mode** is `lax`, the validated element must conform to its top-level element declaration if such a declaration exists in the effective schema. If **validation mode** is `lax` and there is no top-level element declaration for the element, and the element has an `xsi:type` attribute, then the `xsi:type` attribute must name a top-level type definition in the effective schema, and the element must conform to that type. The validated element corresponds either to the operand node or (if the operand node is a document node) to its element child.

☞ During conversion of the PSVI into an XDM instance after validation, any element information items whose validity property is `notKnown` are converted into element nodes with type annotation `xs:anyType`, and any attribute information items whose validity property is `notKnown` are converted into attribute nodes with type annotation `xs:untypedAtomic`, as described in .

3.14. Extension Expressions

An *extension expression* is an expression whose semantics are implementation-defined. Typically a particular extension will be recognized by some implementations and not by others. The syntax is designed so that extension expressions can be successfully parsed by all implementations, and so that fallback behavior can be defined for implementations that do not recognize a particular extension.

[127] ExtensionExpr ::= **Pragma**+ "{" **Expr**? "}"

[128] Pragma ::= "(#" **S**? **QName** (**S PragmaContents**)? "#)"

[129] PragmaContents ::= (**Char*** - (Char* '#)' Char*))

An extension expression consists of one or more *pragmas*, followed by an expression enclosed in curly braces. A *pragma* is denoted by the delimiters (# and #), and consists of an identifying QName followed by implementation-defined content. The content of a pragma may consist of any string of characters that does not contain the ending delimiter #). The QName of a pragma must resolve to a namespace URI and local name, using the statically known namespaces .

☞ Since there is no default namespace for pragmas, a pragma QName must include a namespace prefix.

Each implementation recognizes an implementation-defined set of namespace URIs used to denote pragmas.

If the namespace part of a pragma QName is not recognized by the implementation as a pragma namespace, then the pragma is ignored. If all the pragmas in an **ExtensionExpr** are ignored, then the value of the **ExtensionExpr** is the value of the expression enclosed in curly braces; if this expression is absent, then a static error is raised .

If an implementation recognizes the namespace of one or more pragmas in an **ExtensionExpr**, then the value of the **ExtensionExpr**, including its error behavior, is implementation-defined. For example, an implementation that recognizes the namespace of a pragma QName, but does not recognize the local part of the QName, might choose either to raise an error or to ignore the pragma.

It is a static error if an implementation recognizes a pragma but determines that its content is invalid.

If an implementation recognizes a pragma, it must report any static errors in the following expression even if it will not evaluate that expression (however, static type errors are raised only if the Static Typing Feature is in effect.)

☞ The following examples illustrate three ways in which extension expressions might be used.

- A pragma can be used to furnish a hint for how to evaluate the following expression, without actually changing the result. For example:

```
declare namespace exq = "http://example.org/XQueryImplementation";
   (# exq:use-index #)
      { $bib/book/author[name='Berners-Lee'] }
```

An implementation that recognizes the `exq:use-index` pragma might use an index to evaluate the expression that follows. An implementation that does not recognize this pragma would evaluate the expression in its normal way.

- A pragma might be used to modify the semantics of the following expression in ways that would not (in the absence of the pragma) be conformant with this specification. For example, a pragma might be used to permit comparison of `xs:duration` values using implementation-defined semantics (this would normally be an error). Such changes to the language semantics must be scoped to the expression contained within the curly braces following the pragma.
- A pragma might contain syntactic constructs that are evaluated in place of the following expression. In this case, the following expression itself (if it is present) provides a fallback for use by implementations that do not recognize the pragma. For example:

```
declare namespace exq = "http://example.org/XQueryImplementation";
   for $x in
      (# exq:distinct //city by @country #)
      { //city[not(@country = preceding::city/@country)] }
   return f:show-city($x)
```

Here an implementation that recognizes the pragma will return the result of evaluating the proprietary syntax `exq:distinct //city by @country`, while an implementation that does not recognize the pragma will instead return the result of the expression `//city[not(@country = preceding::city/@country)]`. If no fallback expression is required, or if none is feasible, then the expression between the curly braces may be omitted, in which case implementations that do not recognize the pragma will raise a static error.

4. Modules and Prologs

[30]	Module	::=	**VersionDecl**? (**LibraryModule** | **MainModule**)
[31]	MainModule	::=	**Prolog QueryBody**
[32]	LibraryModule	::=	**ModuleDecl Prolog**
[33]	Prolog	::=	((**DefaultNamespaceDecl** | **Setter** | **NamespaceDecl** | **Import**) **Separator**)* ((**VarDecl** | **FunctionDecl** | **OptionDecl**) **Separator**)*
[34]	Setter	::=	**BoundarySpaceDecl** | **DefaultCollationDecl** | **BaseURIDecl** | **ConstructionDecl** | **OrderingModeDecl** | **EmptyOrderDecl** | **CopyNamespacesDecl**
[35]	Import	::=	**SchemaImport** | **ModuleImport**
[36]	Separator	::=	";"
[37]	QueryBody	::=	**Expr**

A query can be assembled from one or more fragments called *modules*. A *module* is a fragment of XQuery code that conforms to the **Module** grammar and can independently undergo the static analysis phase described in § 2.2.3 – Expression Processing on page 10. Each module is either a main module or a library module.

A *main module* consists of a Prolog followed by a Query Body. A query has exactly one main module. In a main module, the Query Body can be evaluated, and its value is the result of the query.

A module that does not contain a Query Body is called a *library module*. A library module consists of a module declaration followed by a Prolog. A library module cannot be evaluated directly; instead, it provides function and variable declarations that can be imported into other modules.

The XQuery syntax does not allow a module to contain both a module declaration and a Query Body.

A *Prolog* is a series of declarations and imports that define the processing environment for the module that contains the Prolog. Each declaration or import is followed by a semicolon. A Prolog is organized into two parts.

The first part of the Prolog consists of setters, imports, namespace declarations, and default namespace declarations. *Setters* are declarations that set the value of some property that affects query processing, such as construction mode, ordering mode, or default collation. Namespace declarations and default namespace declarations affect the interpretation of QNames within the query. Imports are used to import definitions from schemas and modules. Each imported schema or module is identified by its *target namespace*, which is the namespace of the objects (such as elements or functions) that are defined by the schema or module.

The second part of the Prolog consists of declarations of variables, functions, and options. These declarations appear at the end of the Prolog because they may be affected by declarations and imports in the first part of the Prolog.

The *Query Body*, if present, consists of an expression that defines the result of the query. Evaluation of expressions is described in § 3 – Expressions on page 30. A module can be evaluated only if it has a Query Body.

4.1. Version Declaration

[18] VersionDecl ::= "xquery" "version" **StringLiteral** ("encoding" **StringLiteral**)? **Separator**

Any module may contain a *version declaration*. If present, the version declaration occurs at the beginning of the module and identifies the applicable XQuery syntax and semantics for the module. The version number "1.0" indicates a requirement that the module must be processed by an implementation that supports XQuery Version 1.0. If the version declaration is not present, the version is presumed to be "1.0". An XQuery implementation must raise a static error when processing a module labeled with a version that the implementation does not support. It is the intent of the XQuery working group to give later versions of this specification numbers other than "1.0", but this intent does not indicate a commitment to produce any future versions of XQuery, nor if any are produced, to use any particular numbering scheme.

If present, a version declaration may optionally include an *encoding declaration*. The value of the string literal following the keyword `encoding` is an encoding name, and must conform to the definition of `EncName` specified in [XML 1.0]. The purpose of an encoding declaration is to allow the writer of a query to provide a string that indicates how the query is encoded, such as "`UTF-8`", "`UTF-16`", or "`US-ASCII`". Since the encoding of a query may change as the query moves from one environment to another, there can be no guarantee that the encoding declaration is correct.

The handling of an encoding declaration is implementation-dependent. If an implementation has *a priori* knowledge of the encoding of a query, it may use this knowledge and disregard the encoding declaration. The semantics of a query are not affected by the presence or absence of an encoding declaration.

If a version declaration is present, no **Comment** may occur before the end of the version declaration. If such a **Comment** is present, the result is implementation-dependent.

☞ The effect of a Comment before the end of a version declaration is implementation-dependent because it may suppress query processing by interfering with detection of the encoding declaration.

The following examples illustrate version declarations:

```
xquery version "1.0";
```

```
xquery version "1.0" encoding "utf-8";
```

4.2. Module Declaration

[39] ModuleDecl ::= "module" "namespace" **NCName** "=" **URILiteral Separator**

A *module declaration* serves to identify a module as a library module. A module declaration begins with the keyword `module` and contains a namespace prefix and a **URILiteral**. The URILiteral must be of nonzero length . The URILiteral identifies the target namespace of the library module, which is the namespace for all variables and functions exported by the library module. The name of every variable and function declared in a library module must have a namespace URI that is the same as the target namespace of the module; otherwise a static error is raised . In the statically known namespaces of the library module, the namespace prefix specified in the module declaration is bound to the module's target namespace.

The namespace prefix specified in a module declaration must not be `xml` or `xmlns` , and must not be the same as any namespace prefix bound in the same module by a schema import, by a namespace declaration, or by a module import with a different target namespace .

Any module may import one or more library modules by means of a module import that specifies the target namespace of the library modules to be imported. When a module imports one or more library modules, the variables and functions declared in the imported modules are added to the static context and (where applicable) to the dynamic context of the importing module.

The following is an example of a module declaration:

```
module namespace math = "http://example.org/math-functions";
```

4.3. Boundary-space Declaration

[40] BoundarySpaceDecl ::= "declare" "boundary-space" ("preserve" | "strip")

A *boundary-space declaration* sets the boundary-space policy in the static context, overriding any implementation-defined default. Boundary-space policy controls whether boundary whitespace is preserved by element constructors during processing of the query. If boundary-space policy is `preserve`, boundary whitespace is preserved. If boundary-space policy is `strip`, boundary whitespace is stripped (deleted). A further discussion of whitespace in constructed elements can be found in § 3.7.1.4 – Boundary Whitespace on page 62.

The following example illustrates a boundary-space declaration:

```
declare boundary-space preserve;
```

If a Prolog contains more than one boundary-space declaration, a static error is raised .

4.4. Default Collation Declaration

[41] DefaultCollationDecl ::= "declare" "default" "collation" **URILiteral**

A *default collation declaration* sets the value of the default collation in the static context, overriding any implementation-defined default. The default collation is the collation that is used by functions and operators

that require a collation if no other collation is specified. For example, the `gt` operator on strings is defined by a call to the `fn:compare` function, which takes an optional collation parameter. Since the `gt` operator does not specify a collation, the `fn:compare` function implements `gt` by using the default collation.

If neither the implementation nor the Prolog specifies a default collation, the Unicode codepoint collation (`http://www.w3.org/2005/xpath-functions/collation/codepoint`) is used.

The following example illustrates a default collation declaration:

```
declare default collation
  "http://example.org/languages/Icelandic";
```

If a default collation declaration specifies a collation by a relative URI, that relative URI is resolved to an absolute URI using the base URI in the static context. If a Prolog contains more than one default collation declaration, or the value specified by a default collation declaration (after resolution of a relative URI, if necessary) is not present in statically known collations, a static error is raised .

4.5. Base URI Declaration

[142] BaseURIDecl ::= "declare" "base-uri" **URILiteral**

A *base URI declaration* specifies the base URI property of the static context. The base URI property is used when resolving relative URIs within a module. For example, the `fn:doc` function resolves a relative URI using the base URI of the calling module.

The following is an example of a base URI declaration:

```
declare base-uri "http://example.org";
```

If a Prolog contains more than one base URI declaration, a static error is raised .

In the terminology of [RFC3986] Section 5.1, the URILiteral of the base URI declaration is considered to be a "base URI embedded in content". If no base URI declaration is present, the base URI in the static context is established according to the principles outlined in [RFC3986] Section 5.1—that is, it defaults first to the base URI of the encapsulating entity, then to the URI used to retrieve the entity, and finally to an implementation-defined default. If the URILiteral in the base URI declaration is a relative URI, then it is made absolute by resolving it with respect to this same hierarchy. For example, if the URILiteral in the base URI declaration is `../data/`, and the query is contained in a file whose URI is `file:///C:/temp/queries/query.xq`, then the base URI in the static context is `file:///C:/temp/data/`.

It is not intrinsically an error if this process fails to establish an absolute base URI; however, the base URI in the static context is then undefined, and any attempt to use its value may result in an error .

4.6. Construction Declaration

[148] ConstructionDecl ::= "declare" "construction" ("strip" | "preserve")

A *construction declaration* sets the construction mode in the static context, overriding any implementation-defined default. The construction mode governs the behavior of element and document node constructors. If construction mode is `preserve`, the type of a constructed element node is `xs:anyType`, and all attribute and element nodes copied during node construction retain their original types. If construction mode is `strip`, the type of a constructed element node is `xs:untyped`; all element nodes copied during

node construction receive the type `xs:untyped`, and all attribute nodes copied during node construction receive the type `xs:untypedAtomic`.

The following example illustrates a construction declaration:

```
declare construction strip;
```

If a Prolog specifies more than one construction declaration, a static error is raised .

4.7. Ordering Mode Declaration

[44] OrderingModeDecl ::= "declare" "ordering" ("ordered" | "unordered")

An *ordering mode declaration* sets the ordering mode in the static context, overriding any implementation-defined default. This ordering mode applies to all expressions in a module (including both the Prolog and the Query Body, if any), unless overridden by an `ordered` or `unordered` expression.

Ordering mode affects the behavior of path expressions that include a "`/`" or "`//`" operator or an axis step; `union`, `intersect`, and `except` expressions; and FLWOR expressions that have no `order by` clause. If ordering mode is `ordered`, node sequences returned by path, `union`, `intersect`, and `except` expressions are in document order; otherwise the order of these return sequences is implementation-dependent. The effect of ordering mode on FLWOR expressions is described in § 3.8 – FLWOR Expressions on page 73.

The following example illustrates an ordering mode declaration:

```
declare ordering unordered;
```

If a Prolog contains more than one ordering mode declaration, a static error is raised .

4.8. Empty Order Declaration

[45] EmptyOrderDecl ::= "declare" "default" "order" "empty" ("greatest" | "least")

An *empty order declaration* sets the default order for empty sequences in the static context, overriding any implementation-defined default. This declaration controls the processing of empty sequences and `NaN` values as ordering keys in an `order by` clause in a FLWOR expression. An individual `order by` clause may override the default order for empty sequences by specifying `empty greatest` or `empty least`.

The following example illustrates an empty order declaration:

```
declare default order empty least;
```

If a Prolog contains more than one empty order declaration, a static error is raised .

It is important to distinguish an empty order declaration from an ordering mode declaration. An empty order declaration applies only when an `order by` clause is present, and specifies how empty sequences are treated by the `order by` clause (unless overridden). An ordering mode declaration, on the other hand, applies only in the absence of an `order by` clause.

4.9. Copy-Namespaces Declaration

[46] CopyNamespacesDecl ::= "declare" "copy-namespaces" **PreserveMode** "," **InheritMode**

[147] PreserveMode ::= "preserve" | "no-preserve"

[148] InheritMode ::= "inherit" | "no-inherit"

A *copy-namespaces declaration* sets the value of copy-namespaces mode in the static context, overriding any implementation-defined default. Copy-namespaces mode controls the namespace bindings that are assigned when an existing element node is copied by an element constructor or document constructor. Handling of namespace bindings by element constructors is described in § 3.7.1 – Direct Element Constructors on page 55.

The following example illustrates a copy-namespaces declaration:

```
declare copy-namespaces preserve, no-inherit;
```

If a Prolog contains more than one copy-namespaces declaration, a static error is raised .

4.10. Schema Import

[149] SchemaImport ::= "import" "schema" **SchemaPrefix**? **URILiteral** ("at" **URILiteral** ("," **URILiteral**)*)?

[150] SchemaPrefix ::= ("namespace" **NCName** "=") | ("default" "element" "namespace")

A *schema import* imports the element declarations, attribute declarations, and type definitions from a schema into the in-scope schema definitions. The schema to be imported is identified by its target namespace. The schema import may bind a namespace prefix to the target namespace of the imported schema, or may declare that target namespace to be the default element/type namespace. The schema import may also provide optional hints for locating the schema.

The namespace prefix specified in a schema import must not be `xml` or `xmlns` , and must not be the same as any namespace prefix bound in the same module by another schema import, a module import, a namespace declaration, or a module declaration .

The first **URILiteral** in a schema import specifies the target namespace of the schema to be imported. The URILiterals that follow the `at` keyword are optional location hints, and can be interpreted or disregarded in an implementation-dependent way. Multiple location hints might be used to indicate more than one possible place to look for the schema or multiple physical resources to be assembled to form the schema.

A schema import that specifies a zero-length string as target namespace is considered to import a schema that has no target namespace. Such a schema import may not bind a namespace prefix , but it may set the default element/type namespace to a zero-length string (representing "no namespace"), thus enabling the definitions in the imported namespace to be referenced. If the default element/type namespace is not set to "no namespace", there is no way to reference the definitions in an imported schema that has no target namespace.

It is a static error if more than one schema import in the same Prolog specifies the same target namespace. It is a static error if the implementation is not able to process a schema import by finding a valid schema with the specified target namespace. It is a static error if multiple imported schemas, or multiple physical resources within one schema, contain definitions for the same name in the same symbol space (for example, two definitions for the same element name, even if the definitions are consistent). However, it is not an error to import the schema with target namespace `http://www.w3.org/2001/XMLSchema` (predeclared prefix `xs`), even though the built-in types defined in this schema are implicitly included in the in-scope schema types.

It is a static error if the set of definitions contained in all schemas imported by a Prolog do not satisfy the conditions for schema validity specified in Sections 3 and 5 of [XML Schema] Part 1--i.e., each definition must be valid, complete, and unique.

The following example imports a schema, specifying both its target namespace and its location, and binding the prefix `soap` to the target namespace:

```
import schema namespace soap="http://www.w3.org/2003/05/soap-envelope"
at "http://www.w3.org/2003/05/soap-envelope/";
```

The following example imports a schema by specifying only its target namespace, and makes it the default element/type namespace:

```
import schema default element namespace "http://example.org/abc";
```

The following example imports a schema that has no target namespace, providing a location hint, and sets the default element/type namespace to "no namespace" so that the definitions in the imported schema can be referenced:

```
import schema default element namespace ""
at "http://example.org/xyz.xsd";
```

The following example imports a schema that has no target namespace and sets the default element/type namespace to "no namespace". Since no location hint is provided, it is up to the implementation to find the schema to be imported.

```
import schema default element namespace "";
```

4.11. Module Import

[51] ModuleImport ::= "import" "module" ("namespace" **NCName** "=")? **URILiteral** ("at" **URILiteral** ("," **URILiteral**)*)?

A *module import* imports the function declarations and variable declarations from one or more library modules into the function signatures and in-scope variables of the importing module. Each module import names a target namespace and imports an implementation-defined set of modules that share this target namespace. The module import may bind a namespace prefix to the target namespace, and it may provide optional hints for locating the modules to be imported.

The namespace prefix specified in a module import must not be `xml` or `xmlns` , and must not be the same as any namespace prefix bound in the same module by another module import, a schema import, a namespace declaration, or a module declaration with a different target namespace .

The first **URILiteral** in a module import must be of nonzero length , and specifies the target namespace of the modules to be imported. The URILiterals that follow the `at` keyword are optional location hints, and can be interpreted or disregarded in an implementation-defined way.

It is a static error if more than one module import in a Prolog specifies the same target namespace. It is a static error if the implementation is not able to process a module import by finding a valid module definition with the specified target namespace. It is a static error if the expanded QName and arity of a function declared in an imported module are respectively equal to the expanded QName and arity of a function declared in the importing module or in another imported module (even if the declarations are consistent) . It is a static error if the expanded QName of a variable declared in an imported module is equal (as defined

by the `eq` operator) to the expanded QName of a variable declared in the importing module or in another imported module (even if the declarations are consistent) .

Each module has its own static context. A module import imports only functions and variable declarations; it does not import other objects from the imported modules, such as in-scope schema definitions or statically known namespaces. Module imports are not transitive—that is, importing a module provides access only to function and variable declarations contained directly in the imported module. For example, if module A imports module B, and module B imports module C, module A does not have access to the functions and variables declared in module C.

A module may import its own target namespace (this is interpreted as importing an implementation-defined set of other modules that share its target namespace.) However, it is a static error if the graph of module imports contains a cycle (that is, if there exists a sequence of modules M_1 ... M_n such that each M_i imports M_{i+1} and M_n imports M_1), unless all the modules in the cycle share a common namespace.

It is a static error to import a module if the importing module's in-scope schema types do not include definitions for the schema type names that appear in the declarations of variables and functions (whether in an argument type or return type) that are present in the imported module and are referenced in the importing module.

To illustrate the above rules, suppose that a certain schema defines a type named `triangle`. Suppose that a library module imports the schema, binds its target namespace to the prefix `geometry`, and declares a function with the following function signature: `math:area($t as geometry:triangle) as xs:double`. If a query wishes to use this function, it must import *both* the library module and the schema on which it is based. Importing the library module alone would not provide access to the definition of the type `geometry:triangle` used in the signature of the `area` function.

A module M_1 *directly depends* on another module M_2 (different from M_1) if a variable or function declared in M_1 depends on a variable or function declared in M_2. It is a static error to import a module M_1 if there exists a sequence of modules M_1 ... M_i ... M_1 such that each module directly depends on the next module in the sequence (informally, if M_1 depends on itself through some chain of module dependencies.)

The following example illustrates a module import:

```
import module namespace math = "http://example.org/math-functions";
```

4.12. Namespace Declaration

[52] NamespaceDecl ::= "declare" "namespace" **NCName** "=" **URILiteral**

A *namespace declaration* declares a namespace prefix and associates it with a namespace URI, adding the (prefix, URI) pair to the set of statically known namespaces. The namespace declaration is in scope throughout the query in which it is declared, unless it is overridden by a namespace declaration attribute in a direct element constructor.

If the URILiteral part of a namespace declaration is a zero-length string, any existing namespace binding for the given prefix is removed from the statically known namespaces. This feature provides a way to remove predeclared namespace prefixes such as `local`.

The following query illustrates a namespace declaration:

```
declare namespace foo = "http://example.org";
<foo:bar> Lentils </foo:bar>
```

In the query result, the newly created node is in the namespace associated with the namespace URI `http://example.org`.

The namespace prefix specified in a namespace declaration must not be `xml` or `xmlns` , and must not be the same as any namespace prefix bound in the same module by a module import, schema import, module declaration, or another namespace declaration .

It is a static error if an expression contains a QName with a namespace prefix that is not in the statically known namespaces.

XQuery has several predeclared namespace prefixes that are present in the statically known namespaces before each query is processed. These prefixes may be used without an explicit declaration. They may be overridden by namespace declarations in a Prolog or by namespace declaration attributes on constructed elements (however, the prefix `xml` may not be redeclared, and no other prefix may be bound to the namespace URI associated with the prefix `xml`). The predeclared namespace prefixes are as follows:

- `xml = http://www.w3.org/XML/1998/namespace`
- `xs = http://www.w3.org/2001/XMLSchema`
- `xsi = http://www.w3.org/2001/XMLSchema-instance`
- `fn = http://www.w3.org/2005/xpath-functions`
- `local = http://www.w3.org/2005/xquery-local-functions` (see § 4.15 – Function Declaration on page 103.)

Additional predeclared namespace prefixes may be added to the statically known namespaces by an implementation.

When element or attribute names are compared, they are considered identical if the local parts and namespace URIs match on a codepoint basis. Namespace prefixes need not be identical for two names to match, as illustrated by the following example:

```
declare namespace xx = "http://example.org";

let $i := <foo:bar xmlns:foo = "http://example.org">
              <foo:bing> Lentils </foo:bing>
          </foo:bar>
return $i/xx:bing
```

Although the namespace prefixes `xx` and `foo` differ, both are bound to the namespace URI `"http://example.org"`. Since `xx:bing` and `foo:bing` have the same local name and the same namespace URI, they match. The output of the above query is as follows.

```
<foo:bing xmlns:foo = "http://example.org"> Lentils </foo:bing>
```

4.13. Default Namespace Declaration

[53] DefaultNamespaceDecl ::= "declare" "default" ("element" | "function") "namespace" **URILiteral**

Default namespace declarations can be used in a Prolog to facilitate the use of unprefixed QNames. The following kinds of default namespace declarations are supported:

- A *default element/type namespace declaration* declares a namespace URI that is associated with unprefixed names of elements and types. This declaration is recorded as the default element/type

namespace in the static context. A Prolog may contain at most one default element/type namespace declaration . If the **URILiteral** in a default element/type namespace declaration is a zero-length string, the default element/type namespace is undeclared (set to "none"), and unprefixed names of elements and types are considered to be in no namespace. The following example illustrates the declaration of a default namespace for elements and types:

```
declare default element namespace "http://example.org/names";
```

A default element/type namespace declaration may be overridden by a namespace declaration attribute in a direct element constructor.

If no default element/type namespace declaration is present, unprefixed element and type names are in no namespace (however, an implementation may define a different default as specified in Appendix C.1 – Static Context Components on page 128.)

- A *default function namespace declaration* declares a namespace URI that is associated with unprefixed function names in function calls and function declarations. This declaration is recorded as the default function namespace in the static context. A Prolog may contain at most one default function namespace declaration . If the StringLiteral in a default function namespace declaration is a zero-length string, the default function namespace is undeclared (set to "none"). In that case, any functions that are associated with a namespace can be called only by using an explicit namespace prefix.

  If no default function namespace declaration is present, the default function namespace is the namespace of XPath/XQuery functions, `http://www.w3.org/2005/xpath-functions` (however, an implementation may define a different default as specified in Appendix C.1 – Static Context Components on page 128.)

  The following example illustrates the declaration of a default function namespace:

```
declare default function namespace
     "http://example.org/math-functions";
```

  The effect of declaring a default function namespace is that all functions in the default function namespace, including implicitly-declared constructor functions, can be invoked without specifying a namespace prefix. When a function call uses a function name with no prefix, the local name of the function must match a function (including implicitly-declared constructor functions) in the default function namespace .

  
  Only constructor functions can be in no namespace.

Unprefixed attribute names and variable names are in no namespace.

4.14. Variable Declaration

[54] VarDecl ::= "declare" "variable" "$" **QName** **TypeDeclaration**? ((":=" **ExprSingle**) | "external")

[55] VarName ::= **QName**

[56] TypeDeclaration ::= "as" **SequenceType**

A *variable declaration* adds the static type of a variable to the in-scope variables, and may also add a value for the variable to the variable values. If the expanded QName of the variable is equal (as defined by the `eq` operator) to the name of another variable in in-scope variables, a static error is raised .

If a variable declaration includes a type, that type is added to the static context as the type of the variable. If a variable declaration includes an expression but not an explicit type, the type of the variable is inferred from static analysis of the expression and is added to the static context. If a variable declaration includes both a type and an expression, the value returned by the expression must match the declared type according to the rules for SequenceType matching; otherwise a type error is raised .

If a variable declaration includes an expression, the expression is called an *initializing expression*. The initializing expression for a given variable must be evaluated before the evaluation of any expression that references the variable. The static context for an initializing expression includes all functions that are declared or imported anywhere in the Prolog, but it includes only those variables and namespaces that are declared or imported earlier in the Prolog than the variable that is being initialized.

A variable `$x` *depends* on a variable `$y` or a function `f2` if a reference to `$y` or `f2` appears in the initializing expression of `$x`, or if there exists a variable `$z` or a function `f3` such that `$x` depends on `$z` or `f3` and `$z` or `f3` depends on `$y` or `f2`.

A function `f1` *depends* on a variable `$y` or a function `f2` if a reference to `$y` or `f2` appears in the body of `f1`, or if there exists a variable `$z` or a function `f3` such that `f1` depends on `$z` or `f3` and `$z` or `f3` depends on `$y` or `f2`.

If a variable depends on itself, a static error is raised .

If the variable declaration includes the keyword `external`, a value must be provided for the variable by the external environment before the query can be evaluated. If an external variable declaration also includes a declared type, the value provided by the external environment must match the declared type according to the rules for SequenceType matching (see § 2.2.5 – Consistency Constraints on page 12). If an external variable declaration does not include a declared type, the type and a matching value must be provided by the external environment at evaluation time. The static type of such a variable is considered to be `item()*`. Any reference to a variable that was declared `external`, but was not bound to a value by the external environment, raises a dynamic error .

All variable names declared in a library module must (when expanded) be in the target namespace of the library module . When a library module is imported, variables declared in the imported module are added to the in-scope variables of the importing module.

Variable names that have no namespace prefix are in no namespace. Variable declarations that have no namespace prefix may appear only in a main module.

The term *variable declaration* always refers to a declaration of a variable in a Prolog. The binding of a variable to a value in a query expression, such as a FLWOR expression, is known as a *variable binding*, and does not make the variable visible to an importing module.

Here are some examples of variable declarations:

- The following declaration specifies both the type and the value of a variable. This declaration causes the type `xs:integer` to be associated with variable `$x` in the static context, and the value `7` to be associated with variable `$x` in the dynamic context.

  ```
  declare variable $x as xs:integer := 7;
  ```

- The following declaration specifies a value but not a type. The static type of the variable is inferred from the static type of its value. In this case, the variable `$x` has a static type of `xs:decimal`, inferred from its value which is 7.5.

  ```
  declare variable $x := 7.5;
  ```

- The following declaration specifies a type but not a value. The keyword `external` indicates that the value of the variable will be provided by the external environment. At evaluation time, if the variable `$x` in the dynamic context does not have a value of type `xs:integer`, a type error is raised.

  ```
  declare variable $x as xs:integer external;
  ```

- The following declaration specifies neither a type nor a value. It simply declares that the query depends on the existence of a variable named `$x`, whose type and value will be provided by the external environment. During query analysis, the type of `$x` is considered to be `item()*`. During query evaluation, the dynamic context must include a type and a value for `$x`, and its value must be compatible with its type.

  ```
  declare variable $x external;
  ```

- The following declaration, which might appear in a library module, declares a variable whose name includes a namespace prefix:

  ```
  declare variable $math:pi as xs:double := 3.14159E0;
  ```

4.15. Function Declaration

In addition to the built-in functions described in [XQuery 1.0 and XPath 2.0 Functions and Operators], XQuery allows users to declare functions of their own. A function declaration specifies the name of the function, the names and datatypes of the parameters, and the datatype of the result. All datatypes are specified using the syntax described in § 2.5 – Types on page 21. A function declaration causes the declared function to be added to the function signatures of the module in which it appears.

[57]	FunctionDecl	::=	"declare" "function" **QName** "(" **ParamList**? ")" ("as" **SequenceType**)? (**EnclosedExpr** | "external")
[58]	ParamList	::=	**Param** ("," **Param**)*
[59]	Param	::=	"$" **QName TypeDeclaration**?
[60]	TypeDeclaration	::=	"as" **SequenceType**

A function declaration specifies whether a function is user-defined or external. For a *user-defined function*, the function declaration includes an expression called the *function body* that defines how the result of the function is computed from its parameters.. The static context for a function body includes all functions that are declared or imported anywhere in the Prolog, but it includes only those variables and namespaces that are declared or imported earlier in the Prolog than the function that is being defined.

External functions are functions that are implemented outside the query environment. For example, an XQuery implementation might provide a set of external functions in addition to the core function library described in [XQuery 1.0 and XPath 2.0 Functions and Operators]. External functions are identified by the keyword `external`. The purpose of a function declaration for an external function is to declare the datatypes of the function parameters and result, for use in type checking of the query that contains or imports the function declaration.

An XQuery implementation may provide a facility whereby external functions can be implemented using a host programming language, but it is not required to do so. If such a facility is provided, the protocols by which parameters are passed to an external function, and the result of the function is returned to the invoking query, are implementation-defined. An XQuery implementation may augment the type system of [XQuery/XPath Data Model (XDM)] with additional types that are designed to facilitate exchange of

data with host programming languages, or it may provide mechanisms for the user to define such types. For example, a type might be provided that encapsulates an object returned by an external function, such as an SQL database connection. These additional types, if defined, are considered to be derived by restriction from `xs:anyAtomicType`.

Every user-defined function must be in a namespace--that is, every declared function name must (when expanded) have a non-null namespace URI . If the function name in a function declaration has no namespace prefix, it is considered to be in the default function namespace. Every function name declared in a library module must (when expanded) be in the target namespace of the library module . It is a static error if the function name in a function declaration (when expanded) is in any of the following namespaces:

- `http://www.w3.org/XML/1998/namespace`
- `http://www.w3.org/2001/XMLSchema`
- `http://www.w3.org/2001/XMLSchema-instance`
- `http://www.w3.org/2005/xpath-functions`

It is a static error if the expanded QName and arity (number of arguments) of the declared function are equal (as defined by the `eq` operator) to the expanded QName and arity of another function in function signatures.

In order to allow main modules to declare functions for local use within the module without defining a new namespace, XQuery predefines the namespace prefix `local` to the namespace `http://www.w3.org/2005/xquery-local-functions`. It is suggested (but not required) that this namespace be used for defining local functions.

If a function parameter is declared using a name but no type, its default type is `item()*`. If the result type is omitted from a function declaration, its default result type is `item()*`.

The parameters of a function declaration are considered to be variables whose scope is the function body. It is an static error for a function declaration to have more than one parameter with the same name. The type of a function parameter can be any type that can be expressed as a sequence type.

The following example illustrates the declaration and use of a local function that accepts a sequence of `employee` elements, summarizes them by department, and returns a sequence of `dept` elements.

- Using a function, prepare a summary of employees that are located in Denver.

```
declare function local:summary($emps as element(employee)*)
   as element(dept)*
{
   for $d in fn:distinct-values($emps/deptno)
   let $e := $emps[deptno = $d]
   return
      <dept>
         <deptno>{$d}</deptno>
         <headcount> {fn:count($e)} </headcount>
         <payroll> {fn:sum($e/salary)} </payroll>
      </dept>
};

local:summary(fn:doc("acme_corp.xml")//employee[location = "Denver"])
```

Rules for converting function arguments to their declared parameter types, and for converting the result of a function to its declared result type, are described in § 3.1.5 – Function Calls on page 34.

A function declaration may be recursive—that is, it may reference itself. Mutually recursive functions, whose bodies reference each other, are also allowed. The following example declares a recursive function that computes the maximum depth of a node hierarchy, and calls the function to find the maximum depth of a particular document. In its declaration, the user-declared function `local:depth` calls the built-in functions `empty` and `max`, which are in the default function namespace.

- Find the maximum depth of the document named `partlist.xml`.

```
declare function local:depth($e as node()) as xs:integer
{
   (: A node with no children has depth 1 :)
   (: Otherwise, add 1 to max depth of children :)
   if (fn:empty($e/*)) then 1
   else fn:max(for $c in $e/* return local:depth($c)) + 1
};

local:depth(fn:doc("partlist.xml"))
```

Since a constructor function is effectively declared for every user-defined atomic type in the in-scope schema types, a static error is raised if the Prolog attempts to declare a single-parameter function with the same expanded QName as any of these types.

4.16. Option Declaration

An *option declaration* declares an option that affects the behavior of a particular implementation. Each option consists of an identifying QName and a StringLiteral.

[16] OptionDecl ::= "declare" "option" **QName StringLiteral**

Typically, a particular option will be recognized by some implementations and not by others. The syntax is designed so that option declarations can be successfully parsed by all implementations.

The QName of an option must resolve to a namespace URI and local name, using the statically known namespaces .

There is no default namespace for options.

Each implementation recognizes an implementation-defined set of namespace URIs used to denote option declarations.

If the namespace part of the QName is not a namespace recognized by the implementation as one used to denote option declarations, then the option declaration is ignored.

Otherwise, the effect of the option declaration, including its error behavior, is implementation-defined. For example, if the local part of the QName is not recognized, or if the StringLiteral does not conform to the rules defined by the implementation for the particular option declaration, the implementation may choose whether to report an error, ignore the option declaration, or take some other action.

Implementations may impose rules on where particular option declarations may appear relative to variable declarations and function declarations, and the interpretation of an option declaration may depend on its position.

An option declaration must not be used to change the syntax accepted by the processor, or to suppress the detection of static errors. However, it may be used without restriction to modify the semantics of the query. The scope of the option declaration is implementation-defined—for example, an option declaration might apply to the whole query, to the current module, or to the immediately following function declaration.

The following examples illustrate several possible uses for option declarations:

- This option declaration might be used to set a serialization parameter:

```
declare namespace exq = "http://example.org/XQueryImplementation";
declare option exq:output "encoding = iso-8859-1";
```

- This option declaration might be used to specify how comments in source documents returned by the `fn:doc()` function should be handled:

```
declare option exq:strip-comments "true";
```

- This option declaration might be used to associate a namespace used in function names with a Java class:

```
declare namespace math = "http://example.org/MathLibrary";
declare option exq:java-class "math = java.lang.Math";
```

5. Conformance

This section defines the conformance criteria for an XQuery processor. In this section, the following terms are used to indicate the requirement levels defined in [RFC 2119]. *MUST* means that the item is an absolute requirement of the specification. *MAY* means that an item is truly optional. *SHOULD* means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

An XQuery processor that claims to conform to this specification MUST include a claim of Minimal Conformance as defined in § 5.1 – Minimal Conformance on page 106. In addition to a claim of Minimal Conformance, it MAY claim conformance to one or more optional features defined in § 5.2 – Optional Features on page 107.

5.1. Minimal Conformance

Minimal Conformance to this specification MUST include all of the following items:

1. Support for everything specified in this document except those features specified in § 5.2 – Optional Features on page 107 to be optional. If an implementation does not provide a given optional feature, it MUST implement any requirements specified in § 5.2 – Optional Features on page 107 for implementations that do not provide that feature.
2. A definition of every item specified to be implementation-defined, unless that item is part of an optional feature that is not supported by the implementation. A list of implementation-defined items can be found in Appendix D – Implementation-Defined Items on page 132.

Implementations are not required to define items specified to be implementation-dependent.

3. Support for [XQuery/XPath Data Model (XDM)], as specified in § 5.3 – Data Model Conformance on page 108.
4. Support for all functions defined in [XQuery 1.0 and XPath 2.0 Functions and Operators].

5.2. Optional Features

5.2.1. Schema Import Feature

The *Schema Import Feature* permits the query Prolog to contain a schema import.

If an XQuery implementation does not support the Schema Import Feature, it MUST raise a static error if it encounters a schema import.

If an implementation does not support the Schema Import Feature, the in-scope schema types consist only of built-in and implementation-defined schema type definitions, as described in Appendix C.1 – Static Context Components on page 128.

5.2.2. Schema Validation Feature

The *Schema Validation Feature* permits a query to contain a `validate` expression (see § 3.13 – Validate Expressions on page 89.)

If an XQuery implementation does not support the Schema Validation Feature, it MUST raise a static error if it encounters a `validate` expression.

5.2.3. Static Typing Feature

The *Static Typing Feature* provides support for the static semantics defined in [XQuery 1.0 and XPath 2.0 Formal Semantics], and requires implementations to detect and report type errors during the static analysis phase.

If an implementation does not support the Static Typing Feature, but can nevertheless determine during the static analysis phase that an expression, if evaluated, will necessarily raise a type error 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.

An implementation that does not support the Static Typing Feature is not required to raise type errors during the static analysis phase; however, it MUST detect and raise non-type-related static errors during the static analysis phase.

5.2.3.1. Static Typing Extensions

In some cases, the static typing rules defined in [XQuery 1.0 and XPath 2.0 Formal Semantics] 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*. 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]. See for a formal definition of the constraints on static typing extensions.

5.2.4. Full Axis Feature

The following axes are designated as *optional axes*: `ancestor`, `ancestor-or-self`, `following`, `following-sibling`, `preceding`, and `preceding-sibling`.

A conforming XQuery implementation that supports the *Full Axis Feature* MUST support all the optional axes.

Conforming XQuery implementations that do not support the Full Axis Feature MAY support one or more optional axes; it is implementation-defined which optional axes are supported by such implementations. A conforming implementation that encounters a reference to an optional axis that it does not support MUST raise a static error .

XQuery does not recognize the `namespace` axis (defined by XPath 1.0 and deprecated by XPath 2.0).

5.2.5. Module Feature

A conforming XQuery implementation that supports the *Module Feature* allows a query Prolog to contain a *Module Import* and allows *library modules* to be created.

A conforming implementation that does not support the Module Feature MUST raise a static error if it encounters a module declaration or a module import. Since a module declaration is required in a library module, the Module Feature is required in order to create a library module.

In the absence of the Module Feature, each query consists of a single main module.

5.2.6. Serialization Feature

A conforming XQuery implementation that supports the *Serialization Feature* MUST provide means for serializing the result of a query, as specified in § 2.2.4 – Serialization on page 12.

A conforming XQuery implementation that supports the Serialization Feature MUST conform to Appendix C.3 – Serialization Parameters on page 131. The means by which serialization is invoked is implementation-defined.

If an error is raised during the serialization process as specified in [XSLT 2.0 and XQuery 1.0 Serialization], an conforming XQuery implementation MUST report the error to the calling environment.

Not all implementations need to serialize. For instance, an implementation might provide results via an XML API instead of producing a textual representation.

5.3. Data Model Conformance

All XQuery implementations process data represented in the data model as specified in [XQuery/XPath Data Model (XDM)]. The data model specification relies on languages such as XQuery to specify conformance criteria for the data model in their respective environments, and suggests that the following issues should be considered:

1. *Support for normative construction from an infoset.* A conforming implementation MAY choose to claim conformance to , which defines a normative way to construct an XDM instance from an XML document that is merely well-formed or is governed by a DTD.

2. *Support for normative construction from a PSVI.* A conforming implementation MAY choose to claim conformance to , which defines a normative way to construct an XDM instance from an XML document that is governed by a W3C XML Schema.

3. *Support for XML 1.0 and XML 1.1.* The [XQuery/XPath Data Model (XDM)] supports either [XML 1.0] or [XML 1.1]. In XQuery, the choice of which XML version to support is implementation-defined.

   At the time of writing there is no published version of XML Schema that references the XML 1.1 specifications. This means that datatypes such as `xs:NCName` and `xs:ID` are constrained by the XML 1.0 rules. It is recommended that an XQuery 1.0 processor should implement the rules defined by later versions of XML Schema as they become available.

   
   For suggestions on processing XML 1.1 documents, see [XML 1.1 and Schema 1.0].

4. *Ranges of data values.* In XQuery, the following limits are implementation-defined:

   A. For the `xs:decimal` type, the maximum number of decimal digits (`totalDigits` facet) (must be at least 18).

   B. For the types `xs:date`, `xs:time`, `xs:dateTime`, `xs:gYear`, and `xs:gYearMonth`: the maximum value of the year component and the maximum number of fractional second digits (must be at least 3).

   C. For the `xs:duration type`: the maximum absolute values of the years, months, days, hours, minutes, and seconds components.

   D. For the `xs:yearMonthDuration` type: the maximum absolute value, expressed as an integer number of months.

   E. For the `xs:dayTimeDuration` type: the maximum absolute value, expressed as a decimal number of seconds.

   F. For the types `xs:string`, `xs:hexBinary`, `xs:base64Binary`, `xs:QName`, `xs:anyURI`, `xs:NOTATION`, and types derived from them: limitations (if any) imposed by the implementation on lengths of values.

   The limits listed above need not be fixed, but may depend on environmental factors such as system resources. For example, the length of a value of type `xs:string` may be limited by available memory.

Appendix A. XQuery Grammar

A.1. EBNF

The grammar of XQuery 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 Appendix A.1.2 – Extra-grammatical Constraints on page 116
  - A 'ws:' prefix explains the whitespace rules for the production, the details of which are explained in Appendix A.2.4 – Whitespace Rules on page 121
  - A 'gn:' prefix means a 'Grammar Note', and is meant as a clarification for parsing rules, and is explained in Appendix A.1.3 – Grammar Notes on page 117. 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 Appendix A.2.1 – Terminal Symbols on page 119.

The EBNF notation is described in more detail in Appendix A.1.1 – Notation on page 115.

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.

[62]	Module	::=	**VersionDecl**? (**LibraryModule** | **MainModule**)
[63]	VersionDecl	::=	"xquery" "version" **StringLiteral** ("encoding" **StringLiteral**)? **Separator**
[64]	MainModule	::=	**Prolog QueryBody**
[65]	LibraryModule	::=	**ModuleDecl Prolog**
[66]	ModuleDecl	::=	"module" "namespace" **NCName** "=" **URILiteral Separator**
[67]	Prolog	::=	((**DefaultNamespaceDecl** | **Setter** | **NamespaceDecl** | **Import**) **Separator**)* ((**VarDecl** | **FunctionDecl** | **OptionDecl**) **Separator**)*
[68]	Setter	::=	**BoundarySpaceDecl** | **DefaultCollationDecl** | **BaseURIDecl** | **ConstructionDecl** | **OrderingModeDecl** | **EmptyOrderDecl** | **CopyNamespacesDecl**
[69]	Import	::=	**SchemaImport** | **ModuleImport**
[70]	Separator	::=	";"
[71]	NamespaceDecl	::=	"declare" "namespace" **NCName** "=" **URILiteral**
[72]	BoundarySpaceDecl	::=	"declare" "boundary-space" ("preserve" | "strip")
[73]	DefaultNamespaceDecl	::=	"declare" "default" ("element" | "function") "namespace" **URILiteral**
[74]	OptionDecl	::=	"declare" "option" **QName StringLiteral**
[75]	OrderingModeDecl	::=	"declare" "ordering" ("ordered" | "unordered")
[76]	EmptyOrderDecl	::=	"declare" "default" "order" "empty" ("greatest" | "least")
[77]	CopyNamespacesDecl	::=	"declare" "copy-namespaces" **PreserveMode** "," **InheritMode**
[78]	PreserveMode	::=	"preserve" | "no-preserve"
[79]	InheritMode	::=	"inherit" | "no-inherit"
[80]	DefaultCollationDecl	::=	"declare" "default" "collation" **URILiteral**
[81]	BaseURIDecl	::=	"declare" "base-uri" **URILiteral**
[82]	SchemaImport	::=	"import" "schema" **SchemaPrefix**? **URILiteral** ("at" **URILiteral** ("," **URILiteral**)*)?
[83]	SchemaPrefix	::=	("namespace" **NCName** "=") | ("default" "element" "namespace")

[184]	ModuleImport	::=	"import" "module" ("namespace" **NCName** "=")? **URILiteral** ("at" **URILiteral** ("," **URILiteral**)*)?
[185]	VarDecl	::=	"declare" "variable" "$" **QName** **TypeDeclaration**? ((":=" **ExprSingle**) | "external")
[186]	ConstructionDecl	::=	"declare" "construction" ("strip" | "preserve")
[187]	FunctionDecl	::=	"declare" "function" **QName** "(" **ParamList**? ")" ("as" **SequenceType**)? (**EnclosedExpr** | "external")
[188]	ParamList	::=	**Param** ("," **Param**)*
[189]	Param	::=	"$" **QName** **TypeDeclaration**?
[190]	EnclosedExpr	::=	"{" **Expr** "}"
[191]	QueryBody	::=	**Expr**
[192]	Expr	::=	**ExprSingle** ("," **ExprSingle**)*
[193]	ExprSingle	::=	**FLWORExpr**| **QuantifiedExpr**| **TypeswitchExpr**| **IfExpr**| **OrExpr**
[194]	FLWORExpr	::=	(**ForClause** | **LetClause**)+ **WhereClause**? **OrderByClause**? "return" **ExprSingle**
[195]	ForClause	::=	"for" "$" **VarName** **TypeDeclaration**? **PositionalVar**? "in" **ExprSingle** ("," "$" **VarName** **TypeDeclaration**? **PositionalVar**? "in" **ExprSingle**)*
[196]	PositionalVar	::=	"at" "$" **VarName**
[197]	LetClause	::=	"let" "$" **VarName** **TypeDeclaration**? ":=" **ExprSingle** ("," "$" **VarName** **TypeDeclaration**? ":=" **ExprSingle**)*
[198]	WhereClause	::=	"where" **ExprSingle**
[199]	OrderByClause	::=	(("order" "by") | ("stable" "order" "by")) **OrderSpecList**
[200]	OrderSpecList	::=	**OrderSpec** ("," **OrderSpec**)*
[201]	OrderSpec	::=	**ExprSingle** **OrderModifier**
[202]	OrderModifier	::=	("ascending" | "descending")? ("empty" ("greatest" | "least"))? ("collation" **URILiteral**)?
[203]	QuantifiedExpr	::=	("some" | "every") "$" **VarName** **TypeDeclaration**? "in" **ExprSingle** ("," "$" **VarName** **TypeDeclaration**? "in" **ExprSingle**)* "satisfies" **ExprSingle**
[204]	TypeswitchExpr	::=	"typeswitch" "(" **Expr** ")" **CaseClause**+ "default" ("$" **VarName**)? "return" **ExprSingle**
[205]	CaseClause	::=	"case" ("$" **VarName** "as")? **SequenceType** "return" **ExprSingle**
[206]	IfExpr	::=	"if" "(" **Expr** ")" "then" **ExprSingle** "else" **ExprSingle**
[207]	OrExpr	::=	**AndExpr** ("or" **AndExpr**)*
[208]	AndExpr	::=	**ComparisonExpr** ("and" **ComparisonExpr**)*
[209]	ComparisonExpr	::=	**RangeExpr** ((**ValueComp**| **GeneralComp**| **NodeComp**) **RangeExpr**)?
[210]	RangeExpr	::=	**AdditiveExpr** ("to" **AdditiveExpr**)?
[211]	AdditiveExpr	::=	**MultiplicativeExpr** (("+" | "-") **MultiplicativeExpr**)*
[212]	MultiplicativeExpr	::=	**UnionExpr** (("*" | "div" | "idiv" | "mod") **UnionExpr**)*

[23]	UnionExpr	::=	**IntersectExceptExpr** (("union" | "|") **IntersectExceptExpr**)*	
[24]	IntersectExceptExpr	::=	**InstanceofExpr** (("intersect" | "except") **InstanceofExpr**)*	
[25]	InstanceofExpr	::=	**TreatExpr** ("instance" "of" **SequenceType**)?	
[26]	TreatExpr	::=	**CastableExpr** ("treat" "as" **SequenceType**)?	
[27]	CastableExpr	::=	**CastExpr** ("castable" "as" **SingleType**)?	
[28]	CastExpr	::=	**UnaryExpr** ("cast" "as" **SingleType**)?	
[29]	UnaryExpr	::=	("-" | "+")* **ValueExpr**	
[20]	ValueExpr	::=	**ValidateExpr** | **PathExpr** | **ExtensionExpr**	
[21]	GeneralComp	::=	"=" | "!=" | "<" | "<=" | ">" | ">="	
[22]	ValueComp	::=	"eq" | "ne" | "lt" | "le" | "gt" | "ge"	
[23]	NodeComp	::=	"is" | "<<" | ">>"	
[24]	ValidateExpr	::=	"validate" **ValidationMode**? "{" **Expr** "}"	
[25]	ValidationMode	::=	"lax" | "strict"	
[26]	ExtensionExpr	::=	**Pragma**+ "{" **Expr**? "}"	
[27]	Pragma	::=	"(#" **S**? **QName** (**S PragmaContents**)? "#)"	/* ws: explicit */
[28]	PragmaContents	::=	(**Char*** - (Char* '#)' Char*))	
[29]	PathExpr	::=	("/" **RelativePathExpr**?)| ("//" **RelativePathExpr**)| **RelativePathExpr**	/* xgs: leading-lone-slash */
[30]	RelativePathExpr	::=	**StepExpr** (("/" | "//") **StepExpr**)*	
[31]	StepExpr	::=	**FilterExpr** | **AxisStep**	
[32]	AxisStep	::=	(**ReverseStep** | **ForwardStep**) **PredicateList**	
[33]	ForwardStep	::=	(**ForwardAxis NodeTest**) | **AbbrevForwardStep**	
[34]	ForwardAxis	::=	("child" "::")| ("descendant" "::")| ("attribute" "::")| ("self" "::")| ("descendant-or-self" "::")| ("following-sibling" "::")| ("following" "::")	
[35]	AbbrevForwardStep	::=	"@"? **NodeTest**	
[36]	ReverseStep	::=	(**ReverseAxis NodeTest**) | **AbbrevReverseStep**	
[37]	ReverseAxis	::=	("parent" "::")| ("ancestor" "::")| ("preceding-sibling" "::")| ("preceding" "::")| ("ancestor-or-self" "::")	
[38]	AbbrevReverseStep	::=	".."	
[39]	NodeTest	::=	**KindTest** | **NameTest**	
[40]	NameTest	::=	**QName** | **Wildcard**	
[41]	Wildcard	::=	"*"| (**NCName** ":" "*")| ("*" ":" **NCName**)	/* ws: explicit */
[42]	FilterExpr	::=	**PrimaryExpr PredicateList**	

[243]	PredicateList	::=	**Predicate***	
[244]	Predicate	::=	"[" **Expr** "]"	
[245]	PrimaryExpr	::=	**Literal** | **VarRef** | **ParenthesizedExpr** | **ContextItemExpr** | **FunctionCall** | **OrderedExpr** | **UnorderedExpr** | **Constructor**	
[246]	Literal	::=	**NumericLiteral** | **StringLiteral**	
[247]	NumericLiteral	::=	**IntegerLiteral** | **DecimalLiteral** | **DoubleLiteral**	
[248]	VarRef	::=	"$" **VarName**	
[249]	VarName	::=	**QName**	
[250]	ParenthesizedExpr	::=	"(" **Expr**? ")"	
[251]	ContextItemExpr	::=	"."	
[252]	OrderedExpr	::=	"ordered" "{" **Expr** "}"	
[253]	UnorderedExpr	::=	"unordered" "{" **Expr** "}"	
[254]	FunctionCall	::=	**QName** "(" (**ExprSingle** ("," **ExprSingle**)*)? ")"	/* xgs: reserved-function-names */ /* gn: parens */
[255]	Constructor	::=	**DirectConstructor**| **ComputedConstructor**	
[256]	DirectConstructor	::=	**DirElemConstructor**| **DirCommentConstructor**| **DirPIConstructor**	
[257]	DirElemConstructor	::=	"<" **QName DirAttributeList** ("/>" | (">" **DirElemContent*** "</" **QName S**? ">"))	/* ws: explicit */
[258]	DirAttributeList	::=	(**S** (**QName S**? "=" **S**? **DirAttributeValue**)?)*	/* ws: explicit */
[259]	DirAttributeValue	::=	('"' (**EscapeQuot** | **QuotAttrValueContent**)* '"')| ("'" (**EscapeApos** | **AposAttrValueContent**)* "'")	/* ws: explicit */
[260]	QuotAttrValueContent	::=	**QuotAttrContentChar**| **CommonContent**	
[261]	AposAttrValueContent	::=	**AposAttrContentChar**| **CommonContent**	
[262]	DirElemContent	::=	**DirectConstructor**| **CDataSection**| **CommonContent**| **ElementContentChar**	
[263]	CommonContent	::=	**PredefinedEntityRef** | **CharRef** | "{{" | "}}" | **EnclosedExpr**	
[264]	DirCommentConstructor	::=	"<!--" **DirCommentContents** "-->"	/* ws: explicit */
[265]	DirCommentContents	::=	((**Char** - '-') | ('-' (**Char** - '-')))*	/* ws: explicit */
[266]	DirPIConstructor	::=	"<?" **PITarget** (**S DirPIContents**)? "?>"	/* ws: explicit */

[267]	DirPIContents	::=	(**Char*** - (Char* '?>' Char*))	/* ws: explicit */
[268]	CDataSection	::=	"<![CDATA[" **CDataSectionContents** "]]>"	/* ws: explicit */
[269]	CDataSectionContents	::=	(**Char*** - (Char* ']]>' Char*))	/* ws: explicit */
[270]	ComputedConstructor	::=	**CompDocConstructor**| **CompElemConstructor**| **CompAttrConstructor**| **CompTextConstructor**| **CompCommentConstructor**| **CompPIConstructor**	
[271]	CompDocConstructor	::=	"document" "{" **Expr** "}"	
[272]	CompElemConstructor	::=	"element" (**QName** | ("{" **Expr** "}")) "{" **ContentExpr**? "}"	
[273]	ContentExpr	::=	**Expr**	
[274]	CompAttrConstructor	::=	"attribute" (**QName** | ("{" **Expr** "}")) "{" **Expr**? "}"	
[275]	CompTextConstructor	::=	"text" "{" **Expr** "}"	
[276]	CompCommentConstructor	::=	"comment" "{" **Expr** "}"	
[277]	CompPIConstructor	::=	"processing-instruction" (**NCName** | ("{" **Expr** "}")) "{" **Expr**? "}"	
[278]	SingleType	::=	**AtomicType** "?"?	
[279]	TypeDeclaration	::=	"as" **SequenceType**	
[280]	SequenceType	::=	("empty-sequence" "(" ")")| (**ItemType** **OccurrenceIndicator**?)	
[281]	OccurrenceIndicator	::=	"?" | "*" | "+"	/* xgs: occurrence-indicators */
[282]	ItemType	::=	**KindTest** | ("item" "(" ")") | **AtomicType**	
[283]	AtomicType	::=	**QName**	
[284]	KindTest	::=	**DocumentTest**| **ElementTest**| **AttributeTest**| **SchemaElementTest**| **SchemaAttributeTest**| **PITest**| **CommentTest**| **TextTest**| **AnyKindTest**	
[285]	AnyKindTest	::=	"node" "(" ")"	
[286]	DocumentTest	::=	"document-node" "(" (**ElementTest** | **SchemaElementTest**)? ")"	
[287]	TextTest	::=	"text" "(" ")"	
[288]	CommentTest	::=	"comment" "(" ")"	
[289]	PITest	::=	"processing-instruction" "(" (**NCName** | **StringLiteral**)? ")"	
[290]	AttributeTest	::=	"attribute" "(" (**AttribNameOrWildcard** "," **TypeName**)?)? ")"	
[291]	AttribNameOrWildcard	::=	**AttributeName** | "*"	
[292]	SchemaAttributeTest	::=	"schema-attribute" "(" **AttributeDeclaration** ")"	
[293]	AttributeDeclaration	::=	**AttributeName**	

[294] ElementTest ::= "element" "(" (**ElementNameOrWildcard** ("," **TypeName** "?"?)?)? ")"

[295] ElementNameOrWildcard ::= **ElementName** | "*"

[296] SchemaElementTest ::= "schema-element" "(" **ElementDeclaration** ")"

[297] ElementDeclaration ::= **ElementName**

[298] AttributeName ::= **QName**

[299] ElementName ::= **QName**

[300] TypeName ::= **QName**

[301] URILiteral ::= **StringLiteral**

A.1.1. Notation

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

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

```
symbol ::= expression
```

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> */*.

xgc: 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**, which can take the form of a **NameTest** ("*" or a QName). In contexts where operators like "*", "union", etc., can occur, parsers may have difficulty distinguishing operators from NameTests. For example, without lookahead the first part of the expression "/ * 5", for example is easily taken to be a complete expression, "/ *", which has a very different interpretation (the child nodes of "/").

To reduce the need for lookahead, therefore, if the token immediately following a slash is "*" or a keyword, then the slash must be the beginning, but not the entirety, of a **PathExpr** (and the following token must be a **NameTest**, not an operator).

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.

xgc: 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 Appendix A.2.4.1 – Default Whitespace Handling on page 121. 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.

xgc: 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 Appendix A.3 – Reserved Function Names on page 122.

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)".

xgc: occurrence-indicators

As written, the grammar in Appendix A – XQuery Grammar on page 109 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> */*.

grammar-note: parens

Look-ahead is required to distinguish **FunctionCall** from a QName or keyword followed by a **Pragma** or **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 explicity appear on the right-hand side of the grammar (except in its own production). See Appendix A.2.4.1 – Default Whitespace Handling on page 121. Note that comments are not allowed in direct constructor content, though they are allowed in nested **EnclosedExprs**.

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

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 $i` is syntactically legal, ignoring the comment.
- `5 instance (: strange place for a comment :) of xs:integer` is also syntactically legal.
- `<eg (: an example:)>{$i//title}</eg>` is not syntactically legal.
- `<eg> (: an example:) </eg>` is syntactically legal, but the characters that look like a comment are in fact literal element content.

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 Appendix A.2.1 – Terminal Symbols on page 119.

It is implementation-defined 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. Implementations that support the full [XML 1.1] character set SHOULD, for purposes of interoperability, provide a mode that follows only the [XML 1.0] and [XML Names] lexical rules.

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 Appendix A.3 – Reserved Function Names on page 122.

A.2.1. Terminal Symbols

[32]	IntegerLiteral	::=	**Digits**	
[33]	DecimalLiteral	::=	("." **Digits**) | (**Digits** "." [0-9]*)	/* ws: explicit */
[34]	DoubleLiteral	::=	(("." **Digits**) | (**Digits** ("." [0-9]*)?)) [eE] [+-]? **Digits**	/* ws: explicit */
[35]	StringLiteral	::=	("" (**PredefinedEntityRef** | **CharRef** | **EscapeQuot** | [^"&])* "") | ("' (**PredefinedEntityRef** | **CharRef** | **EscapeApos** | [^'&])* "')	/* ws: explicit */
[36]	PredefinedEntityRef	::=	"&" ("lt" | "gt" | "amp" | "quot" | "apos") ";"	/* ws: explicit */
[37]	EscapeQuot	::=	""	
[38]	EscapeApos	::=	"'	
[39]	ElementContentChar	::=	**Char** - [{}<&]	
[40]	QuotAttrContentChar	::=	**Char** - ["{}<&]	
[41]	AposAttrContentChar	::=	**Char** - ['{}<&]	
[42]	Comment	::=	"(:" (**CommentContents** | **Comment**)* ":)"	/* ws: explicit */ /* gn: comments */
[43]	PITarget	::=	[http://www.w3.org/TR/REC-xml#NT-PITarget]	/* xgs: xml-version */
[44]	CharRef	::=	[http://www.w3.org/TR/REC-xml#NT-CharRef]	/* xgs: xml-version */
[45]	QName	::=	[http://www.w3.org/TR/REC-xml-names/#NT-QName]	/* xgs: xml-version */
[46]	NCName	::=	[http://www.w3.org/TR/REC-xml-names/#NT-NCName]	/* xgs: xml-version */
[47]	S	::=	[http://www.w3.org/TR/REC-xml#NT-S]	/* xgs: xml-

version */

[38] Char ::= [http://www.w3.org/TR/REC-xml#NT-Char] /* xgs: xml-version */

The following symbols are used only in the definition of terminal symbols; they are not terminal symbols in the grammar of Appendix A.1 – EBNF on page 109.

[39] Digits ::= [0-9]+

[40] CommentContents ::= (**Char**+ - (Char* ('(:' | ':)') Char*))

A.2.2. Terminal Delimitation

XQuery 1.0 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.

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

The *non-delimiting terminal symbols* are: **IntegerLiteral**, **NCName**, **QName**, **DecimalLiteral**, **DoubleLiteral**, "ancestor", "ancestor-or-self", "and", "as", "ascending", "at", "attribute", "base-uri", "boundary-space", "by", "case", "cast", "castable", "child", "collation", "comment", "construction", "copy-namespaces", "declare", "default", "descendant", "descendant-or-self", "descending", "div", "document", "document-node", "element", "else", "empty", "empty-sequence", "encoding", "eq", "every", "except", "external", "following", "following-sibling", "for", "function", "ge", "greatest", "gt", "idiv", "if", "import", "in", "inherit", "instance", "intersect", "is", "item", "lax", "le", "least", "let", "lt", "mod", "module", "namespace", "ne", "node", "no-inherit", "no-preserve", "of", "option", "or", "order", "ordered", "ordering", "parent", "preceding", "preceding-sibling", "preserve", "processing-instruction", "return", "satisfies", "schema", "schema-attribute", "schema-element", "self", "some", "stable", "strict", "strip", "text", "then", "to", "treat", "typeswitch", "union", "unordered", "validate", "variable", "version", "where", "xquery"

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 XQuery 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.

The characters #x85 and #x2028 cannot be reliably recognized and translated until the **VersionDecl** declaration (if present) has been read.

A.2.4. Whitespace Rules

A.2.4.1. Default Whitespace Handling

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

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 Appendix A.2.4.2 – Explicit Whitespace Handling on page 122). 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, Appendix A.2.4.1 – Default Whitespace Handling on page 121 does not apply. **Comments** are also not allowed in these productions.

For example, whitespace is not freely allowed by the direct constructor productions, but is specified explicitly in the grammar, in order to be more consistent with XML.

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`

A.4. Precedence Order

The grammar in Appendix A.1 – EBNF on page 109 normatively defines built-in precedence among the operators of XQuery. These operators are summarized here to make clear the order of their precedence from lowest to highest. Operators that have a lower precedence number cannot be contained by operators with a higher precedence number. The associativity column indicates the order in which operators of equal precedence in an expression are applied.

#	Operator	Associativity
1	, (comma)	left-to-right
2	:= (assignment)	right-to-left
3	**for**, **some, every**, **typeswitch**, **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

Appendix B. Type Promotion and Operator Mapping

B.1. 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 on page 34), `order by` clauses (see § 3.8.3 – Order By and Return Clauses on page 76), and operators that accept numeric or string operands (see Appendix B.2 – Operator Mapping on page 124). 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`.

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 XQuery are defined. 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]. 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]. 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`.

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`.

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-date-Time(A, B)	xs:dateTime
A + B	xs:yearMonthDuration	xs:dateTime	op:add-yearMonthDuration-to-date-Time(B, A)	xs:dateTime
A + B	xs:dateTime	xs:dayTimeDuration	op:add-dayTimeDuration-to-dateTime(A, B)	xs:dateTime
A + B	xs:dayTimeDuration	xs:dateTime	op:add-dayTimeDuration-to-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-year-MonthDuration (A, B)	xs:decimal
A div B	xs:dayTimeDuration	xs:dayTimeDuration	op:divide-dayTimeDuration-by-dayTime-Duration (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-greater-than(A, B))	xs:boolean
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

Appendix C. Context Components

The tables in this section describe how values are assigned to the various components of the static context and dynamic context, and to the parameters that control the serialization process.

C.1. Static Context Components

The following table describes the components of the *static context*. The following aspects of each component are described:

- *Default initial value:* This is the initial value of the component if it is not overridden or augmented by the implementation or by a query.
- *Can be overwritten or augmented by implementation:* Indicates whether an XQuery implementation is allowed to replace the default initial value of the component by a different, implementation-defined value and/or to augment the default initial value by additional implementation-defined values.
- *Can be overwritten or augmented by a query:* Indicates whether a query is allowed to replace and/or augment the initial value provided by default or by the implementation. If so, indicates how this is accomplished (for example, by a declaration in the prolog).
- *Scope:* Indicates where the component is applicable. "Global" indicates that the component applies globally, throughout all the modules used in a query. "Module" indicates that the component applies throughout a module. "Lexical" indicates that the component applies within the expression in which it is defined (equivalent to "module" if the component is declared in a Prolog.)
- *Consistency Rules:* Indicates rules that must be observed in assigning values to the component. Additional consistency rules may be found in § 2.2.5 – Consistency Constraints on page 12.

Static Context Components

Component	Default initial value	Can be overwritten or augmented by implementation?	Can be overwritten or augmented by a query?	Scope	Consistency rules
XPath 1.0 Compatibility Mode	`false`	no	no	global	Must be `false`.
Statically known namespaces	`fn`, `xml`, `xs`, `xsi`, `local`	overwriteable and augmentable (except for `xml`)	overwriteable and augmentable by prolog or element constructor	lexical	Only one namespace can be assigned to a given prefix per lexical scope.
Default element/type namespace	no namespace	overwriteable	overwriteable by prolog or element constructor	lexical	Only one default namespace per lexical scope.
Default function namespace	`fn`	overwriteable (not recommended)	overwriteable by prolog	module	None.
In-scope schema types	built-in types in `xs`	augmentable	augmentable by schema import in prolog	module	Only one definition per global or local type.
In-scope element declarations	none	augmentable	augmentable by schema import in prolog	module	Only one definition per global or local element name.
In-scope attribute declarations	none	augmentable	augmentable by schema import in prolog	module	Only one definition per global or local attribute name.
In-scope variables	none	augmentable	overwriteable and augmentable by prolog and by variable-binding expressions	lexical	Only one definition per variable per lexical scope.
Context item static type	`none` (raises error on access)	overwriteable	not explicitly, but can be influenced by expressions	lexical	None.
Function signatures	functions in `fn` namespace, and constructors for built-in atomic types	augmentable	augmentable by module import and by function declaration in prolog	module	Each function must have a unique expanded QName and number of arguments.
Statically known collations	only the default collation	augmentable	no	module	Each URI uniquely identifies a collation.
Default collation	Unicode codepoint collation	overwriteable	overwriteable by prolog	module	None.
Construction mode	`preserve`	overwriteable	overwriteable by prolog	module	Value must be `preserve` or `strip`.
Ordering mode	`ordered`	overwriteable	overwriteable by prolog or expression	lexical	Value must be `ordered` or `unordered`.
Default order for empty sequences	implementation-defined	overwriteable	overwriteable by prolog	module	Value must be `greatest` or `least`.
Boundary-space policy	`strip`	overwriteable	overwriteable by prolog	module	Value must be `preserve` or `strip`.
Copy-namespaces mode	`inherit, preserve`	overwriteable	overwriteable by prolog	module	Value consists of `inherit` or `no-inherit`, and `preserve` or `no-preserve`.

Base URI	See rules in § 4.5 – Base URI Declaration on page 95	overwriteable	overwriteable by prolog	module	Value must be a valid lexical representation of the type xs:anyURI.
Statically known documents	none	augmentable	no	module	None.
Statically known collections	none	augmentable	no	module	None.
Statically known default collection type	`node()*`	overwriteable	no	module	None.

C.2. Dynamic Context Components

The following table describes the components of the *dynamic context*. The following aspects of each component are described:

- *Default initial value:* This is the initial value of the component if it is not overridden or augmented by the implementation or by a query.
- *Can be overwritten or augmented by implementation:* Indicates whether an XQuery implementation is allowed to replace the default initial value of the component by a different implementation-defined value and/or to augment the default initial value by additional implementation-defined values.
- *Can be overwritten or augmented by a query:* Indicates whether a query is allowed to replace and/or augment the initial value provided by default or by the implementation. If so, indicates how this is accomplished.
- *Scope:* Indicates where the component is applicable. "Global" indicates that the component applies globally, throughout all the modules used in a query, and remains constant during evaluation of a query. "Dynamic" indicates that evalation of an expression may influence the value of the component for that expression and for nested expressions.
- *Consistency Rules:* Indicates rules that must be observed in assigning values to the component. Additional consistency rules may be found in § 2.2.5 – Consistency Constraints on page 12.

Dynamic Context Components

Component	Default initial value	Can be overwritten or augmented by implementation?	Can be overwritten or augmented by a query?	Scope	Consistency rules
Context item	none	overwriteable	overwritten during evaluation of path expressions and predicates	dynamic	None
Context position	none	overwriteable	overwritten during evaluation of path expressions and predicates	dynamic	If context item is defined, context position must be >0 and <= context size; else context position is undefined.
Context size	none	overwriteable	overwritten during evaluation of path expressions and predicates	dynamic	If context item is defined, context size must be >0; else context size is undefined.

Variable values	none	augmentable	overwriteable and augmentable by prolog and by variable-binding expressions	dynamic	Names and values must be consistent with in-scope variables.
Function implementations	functions in `fn` namespace, and constructors for built-in atomic types	augmentable	augmentable by module import and by function declaration in prolog	global	Must be consistent with function signatures
Current dateTime	none	must be initialized by implementation	no	global	Must include a timezone. Remains constant during evaluation of a query.
Implicit timezone	none	must be initialized by implementation	no	global	Remains constant during evaluation of a query.
Available documents	none	must be initialized by implementation	no	global	None
Available collections	none	must be initialized by implementation	no	global	None
Default collection	none	overwriteable	no	global	None

C.3. Serialization Parameters

The following table specifies default values for the parameters that control the process of serializing an XDM instance into XML notation (`method = "xml"`). The meanings of the various parameters are defined in [XSLT 2.0 and XQuery 1.0 Serialization]. For each parameter, an XQuery implementation may (but is not required to) provide a means whereby a user can override the default value.

Serialization Parameters

Parameter	Default Value
byte-order-mark	implementation-defined
cdata-section-elements	empty
doctype-public	(none)
doctype-system	(none)
encoding	implementation-defined choice between "utf-8" and "utf-16"
escape-uri-attributes	(not applicable when method = xml)
include-content-type	(not applicable when method = xml)
indent	no
media-type	implementation-defined
method	xml
normalization-form	implementation-defined
omit-xml-declaration	implementation-defined
standalone	implementation-defined
undeclare-prefixes	no
use-character-maps	empty
version	implementation-defined

Appendix 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.
7. Any components of the static context or dynamic context that are overwritten or augmented by the implementation.
8. Which of the optional axes are supported by the implementation, if the Full-Axis Feature is not supported.
9. The default handling of empty sequences returned by an ordering key (sortspec) in an `order by` clause (`empty least` or `empty greatest`).
10. The names and semantics of any extension expressions (pragmas) recognized by the implementation.
11. The names and semantics of any option declarations recognized by the implementation.
12. Protocols (if any) by which parameters can be passed to an external function, and the result of the function can returned to the invoking query.
13. The process by which the specific modules to be imported by a module import are identified, if the Module Feature is supported (includes processing of location hints, if any.)
14. Any static typing extensions supported by the implementation, if the Static Typing Feature is supported.
15. The means by which serialization is invoked, if the Serialization Feature is supported.
16. The default values for the `byte-order-mark`, `encoding`, `media-type`, `normalization-form`, `omit-xml-declaration`, `standalone`, and `version` parameters, if the Serialization Feature is supported.
17. The result of an unsuccessful call to an external function (for example, if the function implementation cannot be found or does not return a value of the declared type).
18. Limits on ranges of values for various data types, as enumerated in § 5.3 – Data Model Conformance on page 108.

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

Appendix 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.

RFC2396

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

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/unicode/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. (Third Edition)* W3C Recommendation. See http://www.w3.org/TR/REC-xml

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/XPath Data Model (XDM)

World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*. W3C Recommendation, 23 Jan. 2007. See http://www.w3.org/TR/xpath-datamodel/.

XQuery 1.0 and XPath 2.0 Formal Semantics

World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C Recommendation, 23 Jan. 2007. See http://www.w3.org/TR/xquery-semantics/.

XQuery 1.0 and XPath 2.0 Functions and Operators

World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Functions and Operators* W3C Recommendation, 23 Jan. 2007. See http://www.w3.org/TR/xpath-functions/.

XSLT 2.0 and XQuery 1.0 Serialization

World Wide Web Consortium. *XSLT 2.0 and XQuery 1.0 Serialization*. W3C Recommendation, 23 Jan. 2007. See http://www.w3.org/TR/xslt-xquery-serialization/.

E.2. Non-normative References

XML Query 1.0 Requirements

World Wide Web Consortium. *XML Query 1.0 Requirements*. W3C Working Draft, 14 Nov 2003. See http://www.w3.org/TR/xquery-requirements/.

XPath 2.0

World Wide Web Consortium. *XML Path Language (XPath) Version 2.0*. W3C Recommendation, 23 Jan. 2007. See http://www.w3.org/TR/xpath20/.

XQueryX 1.0

World Wide Web Consortium. *XQueryX, Version 1.0*. W3C Recommendation, 23 Jan. 2007. See http://www.w3.org/TR/xqueryx/.

XSLT 2.0

World Wide Web Consortium. *XSL Transformations (XSLT) 2.0*. W3C Recommendation, 23 Jan. 2007. 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.html

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

XML Query Use Cases

World Wide Web Consortium. *XML Query Use Cases*. W3C Working Draft, 8 June 2006. See http://www.w3.org/TR/xquery-use-cases/.

XML 1.1 and Schema 1.0

World Wide Web Consortium. *Processing XML 1.0 Documents with XML Schema 1.0 Processors*. W3C Working Group Note, 11 May 2005. See http://www.w3.org/TR/xml11schema10/.

Uniform Resource Locators (URL)

Internet Engineering Task Force (IETF). *Uniform Resource Locators (URL)*. Request For Comment No. 1738, Dec. 1994. See http://www.ietf.org/rfc/rfc1738.txt.

ODMG

Rick Cattell et al. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann Publishers, San Francisco, 1996.

Quilt

Don Chamberlin, Jonathan Robie, and Daniela Florescu. *Quilt: an XML Query Language for Heterogeneous Data Sources*. In *Lecture Notes in Computer Science*, Springer-Verlag, Dec. 2000. Also available at http://www.almaden.ibm.com/cs/people/chamberlin/quilt_lncs.pdf. See also http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html.

XML-QL

Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. *A Query Language for XML*.

SQL

International Organization for Standardization (ISO). *Information Technology-Database Language SQL*. Standard No. ISO/IEC 9075:2003. (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.)

XQL

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

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

Use Case Sample Queries

Queries from the XQuery 1.0 Use Cases, presented in a single file. See http://www.w3.org/TR/xquery-use-cases/xquery-use-case-queries.txt.

XQuery Sample Queries

Queries from this document, presented in a single file. See http://www.w3.org/TR/xquery-use-cases/xquery-wd-queries.txt.

Appendix F. Error Conditions

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.

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.

It is a static error if an expression is not a valid instance of the grammar defined in Appendix A.1 – EBNF on page 109.

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 on page 25.

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()`.

(Not currently used.)

(Not currently used.)

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**.

An implementation that does not support the Schema Import Feature must raise a static error if a Prolog contains a schema import.

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

It is a static error if the set of definitions contained in all schemas imported by a Prolog do not satisfy the conditions for schema validity specified in Sections 3 and 5 of [XML Schema] Part 1--i.e., each definition must be valid, complete, and unique.

It is a static error if an implementation recognizes a pragma but determines that its content is invalid.

(Not currently used.)

(Not currently used.)

An implementation that does not support the Module Feature raises a static error if it encounters a module declaration or a module import.

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.

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

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

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

(Not currently used.)

It is a static error if the value of a namespace declaration attribute is not a **URILiteral**.

(Not currently used.)

It is a type error if the content sequence in an element constructor contains an attribute node following a node that is not an attribute node.

It is a dynamic error if any attribute of a constructed element does not have a name that is distinct from the names of all other attributes of the constructed element.

It is a dynamic error if the result of the content expression of a computed processing instruction constructor contains the string "`?>`".

In a validate expression, it is a dynamic error if the root element information item in the PSVI resulting from validation does not have the expected validity property: `valid` if validation mode is `strict`, or either `valid` or `notKnown` if validation mode is `lax`.

(Not currently used.)

(Not currently used.)

It is a type error if the argument of a `validate` expression does not evaluate to exactly one document or element node.

It is a static error if the version number specified in a version declaration is not supported by the implementation.

A static error is raised if a Prolog contains more than one base URI declaration.

It is a static error if a module contains multiple bindings for the same namespace prefix.

It is a static error if multiple functions declared or imported by a module have the number of arguments and their expanded QNames are equal (as defined by the `eq` operator).

It is a static error to import two schema components that both define the same name in the same symbol space and in the same scope.

It is a static error to import a module if the importing module's in-scope schema types do not include definitions for the schema type names that appear in the declarations of variables and functions (whether in an argument type or return type) that are present in the imported module and are referenced in the importing module.

(Not currently used.)

It is a static error if a Prolog contains more than one default collation declaration, or the value specified by a default collation declaration is not present in statically known collations.

It is a static error for a function declaration to have more than one parameter with the same name.

It is a static error if the attributes specified by a direct element constructor do not have distinct expanded QNames.

It is a dynamic error if the value of the name expression in a computed processing instruction constructor cannot be cast to the type `xs:NCName`.

(Not currently used.)

(Not currently used.)

It is a dynamic error if the `node-name` property of the node constructed by a computed attribute constructor is in the namespace `http://www.w3.org/2000/xmlns/` (corresponding to namespace prefix `xmlns`), or is in no namespace and has local name `xmlns`.

It is a static error if the function name in a function declaration is in one of the following namespaces: `http://www.w3.org/XML/1998/namespace`, `http://www.w3.org/2001/XMLSchema`, `http://www.w3.org/2001/XMLSchema-instance`, `http://www.w3.org/2005/xpath-functions`.

An implementation MAY raise a static error if the value of a **URILiteral** is of nonzero length and is not in the lexical space of `xs:anyURI`.

It is a static error if multiple module imports in the same Prolog specify the same target namespace.

It is a static error if a function or variable declared in a library module is not in the target namespace of the library module.

It is a static error if two or more variables declared or imported by a module have equal expanded QNames (as defined by the `eq` operator.)

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()`.

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.

(Not currently used.)

(Not currently used.)

It is a static error if a variable depends on itself.

It is a static error if a Prolog contains more than one copy-namespaces declaration.

(Not currently used.)

It is a static error if a schema import binds a namespace prefix but does not specify a target namespace other than a zero-length string.

It is a static error if multiple schema imports specify the same target namespace.

It is a static error if an implementation is unable to process a schema or module import by finding a schema or module with the specified target namespace.

It is a static error if the name of a function in a function declaration is not in a namespace (expanded QName has a null namespace URI).

It is a dynamic error if the operand of a validate expression is a document node whose children do not consist of exactly one element node and zero or more comment and processing instruction nodes, in any order.

(Not currently used.)

(Not currently used.)

It is a dynamic error if the value of the name expression in a computed processing instruction constructor is equal to "XML" (in any combination of upper and lower case).

A static error is raised if a Prolog contains more than one ordering mode declaration.

A static error is raised if a Prolog contains more than one default element/type namespace declaration, or more than one default function namespace declaration.

A static error is raised if a Prolog contains more than one construction declaration.

A static error is raised if a Prolog contains more than one boundary-space declaration.

A static error is raised if a Prolog contains more than one empty order declaration.

A static error is raised if a namespace URI is bound to the predefined prefix `xmlns`, or if a namespace URI other than `http://www.w3.org/XML/1998/namespace` is bound to the prefix `xml`, or if the prefix `xml` is bound to a namespace URI other than `http://www.w3.org/XML/1998/namespace`.

A static error is raised if the namespace declaration attributes of a direct element constructor do not have distinct names.

It is a dynamic error if the result of the content expression of a computed comment constructor contains two adjacent hyphens or ends with a hyphen.

It is a static error if the graph of module imports contains a cycle (that is, if there exists a sequence of modules M_1 ... M_n such that each M_i imports M_{i+1} and M_n imports M_1), unless all the modules in the cycle share a common namespace.

It is a dynamic error if the value of the name expression in a computed element or attribute constructor cannot be converted to an expanded QName (for example, because it contains a namespace prefix not found in statically known namespaces.)

An implementation that does not support the Validation Feature must raise a static error if it encounters a `validate` expression.

It is a static error if a `collation` subclause in an `order by` clause of a FLWOR expression does not identify a collation that is present in statically known collations.

(Not currently used.)

(Not currently used.)

It is a static error if an extension expression contains neither a pragma that is recognized by the implementation nor an expression enclosed in curly braces.

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

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

(Not currently used.)

(Not currently used.)

It is a dynamic error if the element validated by a `validate` statement does not have a top-level element declaration in the in-scope element declarations, if validation mode is `strict`.

It is a static error if the namespace URI in a namespace declaration attribute is a zero-length string, and the implementation does not support [XML Names 1.1].

It is a type error if the typed value of a copied element or attribute node is namespace-sensitive when construction mode is `preserve` and copy-namespaces mode is `no-preserve`.

It is a static error if the encoding specified in a Version Declaration does not conform to the definition of `EncName` specified in [XML 1.0].

It is a static error if the literal that specifies the target namespace in a module import or a module declaration is of zero length.

It is a static error if a variable bound in a for clause of a FLWOR expression, and its associated positional variable, do not have distinct names (expanded QNames).

It is a static error if a character reference does not identify a valid character in the version of XML that is in use.

An implementation MAY raise a dynamic error if an `xml:id` error, as defined in [XML ID], is encountered during construction of an attribute named `xml:id`.

An implementation MAY raise a dynamic error if a constructed attribute named `xml:space` has a value other than `preserve` or `default`.

It is a static error to import a module M_1 if there exists a sequence of modules M_1 ... M_i ... M_1 such that each module directly depends on the next module in the sequence (informally, if M_1 depends on itself through some chain of module dependencies.)

Appendix G. The `application/xquery` Media Type

This Appendix specifies the media type for XQuery Version 1.0. XQuery is a language for querying over collections of data from XML data sources, as specified in the main body of this document. This media type is being submitted to the IESG (Internet Engineering Steering Group) for review, approval, and registration with IANA (Internet Assigned Numbers Authority.)

G.1. Introduction

This document, found at http://www.w3.org/TR/xquery/, together with its normative references, defines the language XQuery Version 1.0. This Appendix provides information about the `application/xquery` media type, which is intended to be used for transmitting queries written in the XQuery language.

This document was prepared by members of the W3C XML Query Working Group. Please send comments to public-qt-comments@w3.org, a public mailing list with archives at http://lists.w3.org/Archives/Public/public-qt-comments.

G.2. Registration of MIME Media Type `application/xquery`

MIME media type name: `application`

MIME subtype name: `xquery`

Required parameters: none

Optional parameters: none

The syntax of XQuery is expressed in Unicode but may be written with any Unicode-compatible character encoding, including UTF-8 or UTF-16, or transported as US-ASCII or Latin-1 with Unicode characters outside the range of the given encoding represented using an XML-style `&#xddd;` syntax.

G.2.1. Interoperability Considerations

None known.

G.2.2. Applications Using this Media Type

The public XQuery Web page lists more than two dozen implementations of the XQuery language, both proprietary and open source.

This new media type is being registered to allow for deployment of XQuery on the World Wide Web.

G.2.3. File Extensions

The most common file extensions in use for XQuery are `.xq` and `.xquery`.

The appropriate Macintosh file type code is `TEXT`.

G.2.4. Intended Usage

The intended usage of this media type is for interchange of XQuery expressions.

G.2.5. Author/Change Controller

XQuery was produced by, and is maintained by, the World Wide Web Consortium's XML Query Working Group. The W3C has change control over this specification.

G.3. Encoding Considerations

For use with transports that are not 8-bit clean, quoted-printable encoding is recommended since the XQuery syntax itself uses the US-ASCII-compatible subset of Unicode.

An XQuery document may contain an encoding declaration as part of its version declaration:

```
xquery version "1.0" encoding "utf-8";
```

G.4. Recognizing XQuery Files

An XQuery file may have the string `xquery version "V.V"` near the beginning of the document, where `"V.V"` is a version number. Currently the version number, if present, must be `"1.0"`.

G.5. Charset Default Rules

XQuery documents use the Unicode character set and, by default, the UTF-8 encoding.

G.6. Security Considerations

Queries written in XQuery may cause arbitrary URIs or IRIs to be dereferenced. Therefore, the security issues of [RFC3987] Section 8 should be considered. In addition, the contents of resources identified by `file:` URIs can in some cases be accessed, processed and returned as results. XQuery expressions can invoke any of the functions defined in [XQuery 1.0 and XPath 2.0 Functions and Operators]. For example, the `fn:doc()` and `fn:doc-available()` functions allow local filesystem probes as well as access to any URI-defined resource accessible from the system evaluating the XQuery expression.

XQuery is a full declarative programming language, and supports user-defined functions, external function libraries (modules) referenced by URI, and system-specific "native" functions.

Arbitrary recursion is possible, as is arbitrarily large memory usage, and implementations may place limits on CPU and memory usage, as well as restricting access to system-defined functions.

The XML Query Working group is working on a facility to allow XQuery expressions to create and update persistent data. Untrusted queries should not be given write access to data.

Furthermore, because the XQuery language permits extensions, it is possible that `application/xquery` may describe content that has security implications beyond those described here.

Appendix H. Glossary (Non-Normative)

Appendix I. Example Applications (Non-Normative)

This section contains examples of several important classes of queries that can be expressed using XQuery. The applications described here include joins across multiple data sources, grouping and aggregation, queries based on sequential relationships, recursive transformations, and selection of distinct combinations of values.

I.1. Joins

Joins, which combine data from multiple sources into a single result, are a very important type of query. In this section we will illustrate how several types of joins can be expressed in XQuery. We will base our examples on the following three documents:

1. A document named `parts.xml` that contains many `part` elements; each `part` element in turn contains `partno` and `description` subelements.

2. A document named `suppliers.xml` that contains many `supplier` elements; each `supplier` element in turn contains `suppno` and `suppname` subelements.
3. A document named `catalog.xml` that contains information about the relationships between suppliers and parts. The catalog document contains many `item` elements, each of which in turn contains `partno`, `suppno`, and `price` subelements.

A conventional ("inner") join returns information from two or more related sources, as illustrated by the following example, which combines information from three documents. The example generates a "descriptive catalog" derived from the catalog document, but containing part descriptions instead of part numbers and supplier names instead of supplier numbers. The new catalog is ordered alphabetically by part description and secondarily by supplier name.

```
<descriptive-catalog>
   {
     for $i in fn:doc("catalog.xml")/items/item,
         $p in fn:doc("parts.xml")/parts/part[partno = $i/partno],
         $s in fn:doc("suppliers.xml")/suppliers
                   /supplier[suppno = $i/suppno]
     order by $p/description, $s/suppname
     return
        <item>
           {
           $p/description,
           $s/suppname,
           $i/price
           }
        </item>
   }
</descriptive-catalog>
```

The previous query returns information only about parts that have suppliers and suppliers that have parts. An *outer join* is a join that preserves information from one or more of the participating sources, including elements that have no matching element in the other source. For example, a *left outer join* between suppliers and parts might return information about suppliers that have no matching parts.

The following query demonstrates a left outer join. It returns names of all the suppliers in alphabetic order, including those that supply no parts. In the result, each supplier element contains the descriptions of all the parts it supplies, in alphabetic order.

```
for $s in fn:doc("suppliers.xml")/suppliers/supplier
order by $s/suppname
return
   <supplier>
      {
        $s/suppname,
        for $i in fn:doc("catalog.xml")/items/item
                  [suppno = $s/suppno],
            $p in fn:doc("parts.xml")/parts/part
                  [partno = $i/pno]
```

```
        order by $p/description
        return $p/description
      }
   </supplier>
```

The previous query preserves information about suppliers that supply no parts. Another type of join, called a *full outer join*, might be used to preserve information about both suppliers that supply no parts and parts that have no supplier. The result of a full outer join can be structured in any of several ways. The following query generates a list of `supplier` elements, each containing nested `part` elements for the parts that it supplies (if any), followed by a list of `part` elements for the parts that have no supplier. This might be thought of as a "supplier-centered" full outer join. Other forms of outer join queries are also possible.

```
<master-list>
 {
    for $s in fn:doc("suppliers.xml")/suppliers/supplier
    order by $s/suppname
    return
        <supplier>
           {
             $s/suppname,
             for $i in fn:doc("catalog.xml")/items/item
                     [suppno = $s/suppno],
                 $p in fn:doc("parts.xml")/parts/part
                     [partno = $i/partno]
             order by $p/description
             return
                <part>
                   {
                     $p/description,
                     $i/price
                   }
                </part>
           }
        </supplier>
    ,
    (: parts that have no supplier :)
    <orphan-parts>
       { for $p in fn:doc("parts.xml")/parts/part
         where fn:empty(fn:doc("catalog.xml")/items/item
               [partno = $p/partno] )
         order by $p/description
         return $p/description
       }
    </orphan-parts>
 }
</master-list>
```

The previous query uses an element constructor to enclose its output inside a `master-list` element. The concatenation operator (",") is used to combine the two main parts of the query. The result is an ordered sequence of `supplier` elements followed by an `orphan-parts` element that contains descriptions of all the parts that have no supplier.

I.2. Grouping

Many queries involve forming data into groups and applying some aggregation function such as `fn:count` or `fn:avg` to each group. The following example shows how such a query might be expressed in XQuery, using the catalog document defined in the previous section.

This query finds the part number and average price for parts that have at least 3 suppliers.

```
for $pn in fn:distinct-values(
      fn:doc("catalog.xml")/items/item/partno)
let $i := fn:doc("catalog.xml")/items/item[partno = $pn]
where fn:count($i) >= 3
order by $pn
return
   <well-supplied-item>
      <partno> {$p} </partno>
      <avgprice> {fn:avg($i/price)} </avgprice>
   </well-supplied-item>
```

The `fn:distinct-values` function in this query eliminates duplicate part numbers from the set of all part numbers in the catalog document. The result of `fn:distinct-values` is a sequence in which order is not significant.

Note that `$pn`, bound by a for clause, represents an individual part number, whereas `$i`, bound by a let clause, represents a set of items which serves as argument to the aggregate functions `fn:count($i)` and `fn:avg($i/price)`. The query uses an element constructor to enclose each part number and average price in a containing element called `well-supplied-item`.

The method illustrated above generalizes easily to grouping by more than one data value. For example, consider a census document containing a sequence of `person` elements, each with subelements named `state`, `job`, and `income`. A census analyst might need to prepare a report listing the average `income` for each combination of `state` and `job`. This report might be produced using the following query:

```
for $s in fn:distinct-values(
        fn:doc("census.xml")/census/person/state),
    $j in fn:distinct-values(
        fn:doc("census.xml")/census/person/job)
let $p := fn:doc("census.xml")/census/person
        [state = $s and job = $j]
order by $s, $j
return
   if (fn:exists($p)) then
      <group>
         <state> {$s} </state>
         <job> {$j} </job>
```

```
         <avgincome> {fn:avg($p/income)} </avgincome>
      </group>
   else ()
```

The `if-then-else` expression in the above example prevents generation of groups that contain no data. For example, the census data may contain some persons who live in Nebraska, and some persons whose job is Deep Sea Fisherman, but no persons who live in Nebraska and have the job of Deep Sea Fisherman. If output groups are desired for all possible combinations of states and jobs, the `if-then-else` expression can be omitted from the query. In this case, the output may include "empty" groups such as the following:

```
<group>
   <state>Nebraska</state>
   <job>Deep Sea Fisherman</job>
   <avgincome/>
</group>
```

I.3. Queries on Sequence

XQuery uses the << and >> operators to compare nodes based on document order. Although these operators are quite simple, they can be used to express complex queries for XML documents in which sequence is meaningful. The first two queries in this section involve a surgical report that contains `procedure`, `incision`, `instrument`, `action`, and `anesthesia` elements.

The following query returns all the `action` elements that occur between the first and second `incision` elements inside the first procedure. The original document order among these nodes is preserved in the result of the query.

```
let $proc := /report/procedure[1]
for $i in $proc//action
where $i >> ($proc//incision)[1]
   and $i << ($proc//incision)[2]
return $i
```

It is worth noting here that document order is defined in such a way that a node is considered to precede its descendants in document order. In the surgical report, an `action` is never part of an `incision`, but an `instrument` is. Since the `>>` operator is based on document order, the predicate `$i >> ($proc//incision)[1]` is true for any `instrument` element that is a descendant of the first `incision` element in the first procedure.

For some queries, it may be helpful to define a function that can test whether a node precedes another node without being its ancestor. The following function returns `true` if its first operand precedes its second operand but is not an ancestor of its second operand; otherwise it returns `false`:

```
declare function local:precedes($a as node(), $b as node())
   as boolean
   {
      $a << $b
        and
      fn:empty($a//node() intersect $b)
   };
```

Similarly, a `local:follows` function could be written:

```
declare function local:follows($a as node(), $b as node())
   as boolean
   {
      $a >> $b
        and
      fn:empty($b//node() intersect $a)
   };
```

Using the `local:precedes` function, we can write a query that finds `instrument` elements between the first two incisions, excluding from the query result any `instrument` that is a descendant of the first `incision`:

```
let $proc := /report/procedure[1]
for $i in $proc//instrument
where local:precedes(($proc//incision)[1], $i)
   and local:precedes($i, ($proc//incision)[2])
return $i
```

The following query reports incisions for which no prior anesthesia was recorded in the surgical report. Since an `anesthesia` is never part of an `incision`, we can use `<<` instead of the less-efficient `local:precedes` function:

```
for $proc in /report/procedure
where some $i in $proc//incision satisfies
         fn:empty($proc//anesthesia[. << $i])
return $proc
```

In some documents, particular sequences of elements may indicate a logical hierarchy. This is most commonly true of HTML. The following query returns the introduction of an XHTML document, wrapping it in a `div` element. In this example, we assume that an `h2` element containing the text "Introduction" marks the beginning of the introduction, and the introduction continues until the next `h2` or `h1` element, or the end of the document, whichever comes first.

```
let $intro := //h2[text()="Introduction"],
    $next-h := //(h1|h2)[. >> $intro][1]
return
   <div>
     {
       $intro,
       if (fn:empty($next-h))
         then //node()[. >> $intro]
         else //node()[. >> $intro and . << $next-h]
     }
   </div>
```

Note that the above query makes explicit the hierarchy that was implicit in the original document. In this example, we assume that the `h2` element containing the text "Introduction" has no subelements.

I.4. Recursive Transformations

Occasionally it is necessary to scan over a hierarchy of elements, applying some transformation at each level of the hierarchy. In XQuery this can be accomplished by defining a recursive function. In this section we will present two examples of such recursive functions.

Suppose that we need to compute a table of contents for a given document by scanning over the document, retaining only elements named `section` or `title`, and preserving the hierarchical relationships among these elements. For each `section`, we retain subelements named `section` or `title`; but for each `title`, we retain the full content of the element. This might be accomplished by the following recursive function:

```
declare function local:sections-and-titles($n as node()) as node()?
   {
   if (fn:local-name($n) = "section")
   then element
          { fn:local-name($n) }
          { for $c in $n/* return local:sections-and-titles($c) }
   else if (fn:local-name($n) = "title")
   then $n
   else ( )
   };
```

The "skeleton" of a given document, containing only its sections and titles, can then be obtained by invoking the `local:sections-and-titles` function on the root node of the document, as follows:

```
local:sections-and-titles(fn:doc("cookbook.xml"))
```

As another example of a recursive transformation, suppose that we wish to scan over a document, transforming every attribute named `color` to an element named `color`, and every element named `size` to an attribute named `size`. This can be accomplished by the following recursive function (note that the element constructor in case `$e` generates attributes before child elements):

```
declare function local:swizzle($n as node()) as node()
  {
   typeswitch($n)
     case $a as attribute(color)
       return element color { fn:string($a) }
     case $es as element(size)
       return attribute size { fn:string($es) }
     case $e as element()
       return element
         { fn:local-name($e) }
         { for $c in
             ($e/@* except $e/@color,     (: attr -> attr :)
              $e/size,                    (: elem -> attr :)
              $e/@color,                  (: attr -> elem :)
              $e/node() except $e/size )  (: elem -> elem :)
           return local:swizzle($c) }
     case $d as document-node()
```

```
      return document
        { for $c in $d/* return local:swizzle($c) }
    default return $n
};
```

The transformation can be applied to a whole document by invoking the `local:swizzle` function on the root node of the document, as follows:

```
local:swizzle(fn:doc("plans.xml"))
```

I.5. Selecting Distinct Combinations

It is sometimes necessary to search through a set of data to find all the distinct combinations of a given list of properties. For example, an input data set might consist of a large set of `order` elements, each of which has the same basic structure, as illustrated by the following example:

```
<order>
   <date>2003-10-15</date>
   <product>Dress Shirt</product>
   <size>M</size>
   <color>Blue</color>
   <supplier>Fashion Trends</supplier>
   <quantity>50</quantity>
</order>
```

From this data set, a user might wish to find all the distinct combinations of `product`, `size`, and `color` that occur together in an `order`. The following query returns this list, enclosing each distinct combination in a new element named `option`:

```
for $p in fn:distinct-values(/orders/order/product),
    $s in fn:distinct-values(/orders/order/size),
    $c in fn:distinct-values(/orders/order/color)
    order by $p, $s, $c
    return
       if (fn:exists(/orders/order[product eq $p
                and size eq $s and color eq $c]))
       then
          <option>
             <product>{$p}</product>
             <size>{$s}</size>
             <color>{$c}</color>
          </option>
       else ()
```

Appendix J. Revision Log (Non-Normative)

This log records changes that have been made to this document since the Proposed Recommendation Draft of 21 November 2006.

1. Additional details have been added to Appendix G.6 – Security Considerations on page 142.
2. Rules preventing multiple bindings of a namespace prefix within a module have been clarified.

   This change closes Bugzilla entry 3951. Sections affected: § 4.2 – Module Declaration on page 94, § 4.10 – Schema Import on page 97, § 4.11 – Module Import on page 98, § 4.12 – Namespace Declaration on page 99, and error code XQST0033.