XQuery 1.0: An XML Query Language

W3C Working Draft 30 April 2002

This version:
http://www.w3.org/TR/2002/WD-xquery-20020430
Latest version:
http://www.w3.org/TR/xquery
Previous versions:
http://www.w3.org/TR/2001/WD-xquery-20011220 http://www.w3.org/TR/2001/WD-xquery-20010607

Editors:
Scott Boag (XSL WG), IBM Research <scott_boag@us.ibm.com>
Don Chamberlin (XML Query WG), IBM Almaden Research Center <chamberlin@almaden.ibm.com>
Mary F. Fernandez (XML Query WG), AT&T Labs <mff@research.att.com>
Daniela Florescu (XML Query WG), XQRL <dana@xqrl.com>
Jonathan Robie (XML Query WG), Invited Expert <jonathan.robie@datadirect-technologies.com>
Jérôme Siméon (XML Query WG), Bell Labs, Lucent Technologies <simeon@research.bell-labs.com>
Mugur Stefanescu (XML Query WG), Concentric Visions <MStefanescu@Concentricvisions.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 is a public W3C Working Draft for review by W3C Members and other interested parties. This section describes the status of this document at the time of its publication. It is a draft document and may be updated, replaced, or made obsolete by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress." A list of current public W3C technical reports can be found at http://www.w3.org/TR/.

Much of this document is the result of joint work by the XML Query and XSL Working Groups, which are jointly responsible for XPath 2.0, a language derived from both XPath 1.0 and XQuery. The XPath 2.0 and XQuery 1.0 Working Drafts are generated from a common source. These languages are closely related, sharing much of the same expression syntax and semantics, and much of the text found in the two Working Drafts is identical.

This version of the document contains new details about the type system of XQuery, including a syntax for declaring types in function signatures and other expressions. It describes the semantics of several expressions that operate on types, including `treat`, `assert`, and `validate` expressions. It also describes in greater detail the semantics of element and attribute constructors and how they operate on the underlying data model.

This document is a work in progress. It contains many open issues, and should not be considered to be fully stable. Vendors who wish to create preview implementations based on this document do so at their own risk. While this document reflects the general consensus of the working groups, there are still controversial areas that may be subject to change.

Public comments on this document and its open issues are welcome. Of particular interest are comments on error handling (see issues 97 and 98.) Comments should be sent to the W3C XPath/XQuery mailing list, public-qt-comments@w3.org (archived at http://lists.w3.org/Archives/Public/public-qt-comments/).

XQuery 1.0 has been defined jointly by the XML Query Working Group (part of the XML Activity) and the XSL Working Group (part of the Style Activity).

Table of Contents

---

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 small, easily implementable 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 human-readable 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 Version 1.0 contains XPath Version 2.0 as a subset. 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:

- The XQuery data model defines the information in an XML document that is available to an XQuery processor. The data model is defined in [XQuery 1.0 and XPath 2.0 Data Model].
- The static and dynamic semantics of XQuery are formally defined in [XQuery 1.0 Formal Semantics]. This is done by mapping the full XQuery language into a "core" subset for which the semantics is defined. This document is useful for implementors and others who require a rigorous definition of XQuery. [**Ed. Note:** The current edition of [XQuery 1.0 Formal Semantics] has not incorporated recent language changes; it will be made consistent with this document in its next edition.]
- XQuery's type system is based on [XML Schema]. Work is in progress to ensure that the type systems of XQuery, the XQuery Core, and XML Schema are completely aligned.
- The library of functions and operators supported by XQuery is 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]. [**Ed. Note:** The current edition of [XQueryX 1.0] has not incorporated recent language changes; it will be made consistent with this document in its next edition.]

XQuery is a strongly typed language. Rules for assigning types to XQuery expressions are described in [XQuery 1.0 Formal Semantics]. A query may import type definitions from a Schema, according to rules that will be specified in a future edition of [XQuery 1.0 Formal Semantics].

Processing a query involves a static typing phase and a dynamic evaluation phase. The type system of XQuery is sound, in that the value yielded by dynamic evaluation is guaranteed to be an instance of the type assigned by static typing.

At user option, static typing can be disabled. A query that passes type checking will return the same result regardless of whether type checking is enabled or disabled. [**Ed. Note:** See Issue 41 for a further discussion of static vs. dynamic semantics.]

XQuery is likely to have multiple conformance levels. There may be a conformance level that does not include static type checking. There may be a conformance level that does not support Schema import, so that only built-in types and node types may be used in declarations. [**Ed. Note:** See Issue 42 for a further discussion of conformance levels.]

[**Ed. Note:** A future version of this document will include links between terms (in bold font) and their definitions.]

2 Expressions

2.1 Basics

The basic building block of XQuery is the **expression**. 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 is a **functional language** which allows various kinds of expressions to be nested with full generality. It is also a **strongly-typed language** in which the operands of various expressions, operators, and functions must conform to designated types.

Like XML, XQuery is a case-sensitive language. All keywords in XQuery use lower-case characters.

[4]	Expr	::=	SortExpr
			| OrExpr
			| AndExpr
			| FLWRExpr
			| QuantifiedExpr
			| TypeswitchExpr
			| IfExpr
			| GeneralComp
			| ValueComp
			| NodeComp
			| OrderComp
			| InstanceofExpr
			| RangeExpr
			| AdditiveExpr
			| MultiplicativeExpr
			| UnionExpr
			| IntersectExceptExpr

```
| UnaryExpr
| ValidateExpr
| CastExpr
| Constructor
| RootedPathExpr
| RelativePathExpr
| PrimaryExpr
| StepExpr
```

Note:

For the grammar productions in the main body of this document, the same Basic EBNF notation is used as in [XML], except that grammar symbols always have initial capital letters. The EBNF contains only non-terminals, and all terminal tokens are expanded for readability. Whitespace is not represented. A normative version of the EBNF that includes tokens and token states is presented in the appendix [**A Complete BNF**].

Note:

The use of a prefix token ":" is allowed on unprefixed QNames to visually distinguish tokens that are spelled the same as keywords. For instance, the following is a path expression containing the element names "for", "if", and "return": `:for/:if/:return`.

The value of an expression is either a **sequence** or the **error** value. 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 XML Schema **atomic type**, as defined in [XML Schema] (that is, a simple type that is not a list type or a union type). A **node** conforms to one of the seven **node kinds** described in [XQuery 1.0 and XPath 2.0 Data Model]. Some kinds of nodes have typed values, string values, and names, which can be extracted from the node. 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`.

Error is a special value indicating that an error has been encountered during the evaluation of an expression. Except as noted in this document, if any operand of an expression is the error value, the value of the expression is also the error value.

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

In this document, the namespace prefixes `xs:` and `xsi:` are considered to be bound to the XML Schema namespaces `http://www.w3.org/2001/XMLSchema` and `http://www.w3.org/2001/XMLSchema-instance`, respectively (as described in [XML Schema]), and the prefix `xf:` is considered to be bound to the namespace of XPath/XQuery functions and operators, `http://www.w3.org/2002/04/xquery-operators` (described in [XQuery 1.0 and XPath 2.0 Functions and Operators]). In some cases, where the meaning is clear and namespaces are not important to the discussion, built-in XML Schema typenames such as `integer` and `string` will be used without a namespace prefix.

2.1.1 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 **evaluation context**.

This section describes the context information used by XQuery expressions, including the functions in the core function library. Other functions, outside the core function library, may require additional context information.

2.1.1.1 Static Context

The **static context** of an expression is defined as all 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.

In XQuery, the information in the static context is provided by declarations in the **query prolog** (except as noted below). Static context consists of the following components:

- **In-scope namespaces.** This is a set of (prefix, URI) pairs. The in-scope namespaces are used for resolving prefixes used in QNames within the expression.
- **Default namespace for element and type names.** This is a namespace URI. This namespace is used for any unprefixed QName appearing in a position where an element or type name is expected.
- **Default namespace for function names.** This is a namespace URI. This namespace is used for any unprefixed QName appearing as the function name in a function call.
- **In-scope schema definitions.** This is a set of (QName, type definition) pairs. It defines the set of types that are available for reference within the expression. It includes the built-in schema types and all globally-declared types in imported schemas.
- **In-scope variables.** This is a set of (QName, type) pairs. It defines the set of variables that have been declared and are available for reference within the XPath expression. The QName represents the name of the variable, and the type represents its static data type.

  Unlike the other parts of the static context, variable types are not declared in the query prolog. Instead, they are derived from static analysis of the expressions in which the variables are bound.
- **In-scope functions.** This is a set of (QName, function signature) pairs. It defines the set of functions that are available to be called from within the expression. The QName represents the name of the function, and the function signature specifies the static types of the function parameters and function result.
- **In-scope collations.** This is a set of (URI, collation) pairs. It defines the names of the collations that are available for use in function calls that take a collation name as an argument. A collation may be regarded as an object that supports two functions: a function that given a set of strings, returns a sequence containing those strings in sorted order; and a function that given two strings, returns true if they are considered equal, and false if not.
- **Default collation.** This is a collation. This collation is used by string comparison functions when no explicit collation is specified.
- **Base URI.** This is an absolute URI, used by the `xf:document` function when resolving the relative URI of a document to be loaded, if no explicit base URI is supplied in the function call.

2.1.1.2 Evaluation Context

The **evaluation context** of an expression is defined as information that is available at the time the expression is evaluated. The evaluation context consists of all the components of the static context, and the additional components listed below.

The first four components of the dynamic context (context item, context position, context size, and context document) are called the **focus** of the expression. The focus enables the processor to keep track of which nodes are being processed by the expression.

The focus for the outermost expression is supplied by the environment in which the expression is evaluated. Certain language constructs, notably the path expression `E1/E2`, the filter expression `E1[E2]`, and the ordering expression

`E1 sortby 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. [**Ed. Note:** See issue 216.]

- 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 the expression "`.`". When an expression `E1/E2`, `E1[E2]` or `E2 sortby 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. Its value is always an integer greater than zero. The context position is returned by the expression `position()`. When an expression `E1/E2`, `E1[E2]`, or `E1 sortby 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 `last()`. When an expression `E1/E2`, `E1[E2]`, or `E1 sortby 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`.

- The **context document** is the document currently being processed. Its value is a node, which is always a document node. If there is a context node, and if the tree containing the context node has a document node as its root, then this document node will be the context document. When an inner focus is created and the context item in the inner focus is not a node, or is a node belonging to a tree whose root is not a document node, then the context document in this inner focus will be the same as the context document in the outer focus. The context document is returned by the XPath expression "`/`", and is used as the base for any **rooted path expression**, as well being used implicitly by certain functions such as `xf:id`. [**Ed. Note:** See Issue 217.]

- **Dynamic variables.** This is a set of (QName, type, value) triples. It contains the same QNames as the in-scope variables in the static context for the expression. Each QName is associated with the dynamic type and value of the corresponding variable. The dynamic type associated with a variable may be more specific than the static type associated with the same variable. The value of a variable is, in general, a sequence.

  The dynamic types and values of variables are provided by execution of the XQuery expressions in which the variables are bound.

- **Current date and time.** This information represents a point in time during processing of a query. It can be retrieved by the `xf:current-dateTime` function. If invoked multiple times during the execution of a query, this function always returns the same result.

- **Input sequence.** The input sequence is sequence of nodes that can be accessed by the `input` function. It might be thought of as an "implicit input". The content of the input sequence is determined in an implementation-dependent way.

2.1.1.3 Input Functions

XQuery has three functions that provide access to input data. These functions are of particular importance because they provide the only way in which an expression can reference a document or a collection of documents. The three input functions are described informally here, and in more detail in [XQuery 1.0 and XPath 2.0 Functions and Operators].

The **input sequence** is a part of the evaluation context for an expression. The way in which nodes are assigned to the input sequence is implementation-dependent. For instance, one implementation might provide a fixed mapping from a directory system to the input sequence, another implementation might provide a graphical user interface that allows users to choose a data source for the input sequence, and a third implementation might support UNIX-style pipes, allowing the output of one query to become the input sequence for another query.

The `input` function returns the input sequence. For example, the expression `input()//customer` returns all the `customer` elements that are descendants of nodes in the input sequence. If no input sequence has been bound, the `input` function returns the error value.

The `collection` function returns the nodes found in a collection. A collection may be any sequence of nodes. A collection is identified by a string, which must be a valid URI. For example, the expression `collection("www.example.com/invoices")//customer` identifies all the `customer` elements that are descendants of nodes found in the collection whose URI is `www.example.com/invoices`.

The `document` function, when its first argument is a string containing a single URI that refers to an XML document, returns the root of that document. The `document` function can also be used to address multiple documents or document fragments; see [XQuery 1.0 and XPath 2.0 Functions and Operators] for details.

2.1.2 Expression Processing

XQuery is defined in terms of the [XQuery 1.0 and XPath 2.0 Data Model] (referred to in this document simply as the Data Model), which represents information in the form of nodes, atomic values, and the error value. Before an XQuery expression can be processed, the input documents to be operated on by the expression must be represented in the Data Model. For example, an XML document can be converted to the Data Model by the following steps:

1. The document can be parsed using an XML parser.
2. The parsed document can 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, it can be validated against a permissive default schema that accepts any well-formed document.
3. The PSVI can be transformed into the Data Model by a process described in [XQuery 1.0 and XPath 2.0 Data Model].

The above steps provide an example of how a Data Model instance might be constructed. A Data Model instance might also be synthesized directly from a relational database, or constructed in some other way. XQuery is defined in terms of operations on the Data Model, but it does not place any constraints on how the input Data Model instance is constructed.

Each element or attribute node in the Data Model can be annotated with a dynamic type. If the Data Model was derived from an input XML document, the types of the elements and attributes are derived from schema validation. A newly constructed element or attribute node has no type annotation, but can be given one by a `validate` expression. The type of an element or attribute indicates its range of values--for example, an attribute named `version` might have the type `xs:decimal`, indicating that it contains a decimal value.

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 annotated with the type `xs:anySimpleType`.

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 containing text of unknown type, possibly interleaved with comments and/or processing instructions (such as might occur in a schemaless document), is annotated with the type `xs:anySimpleType`. However, if an element of unknown type contains subelements, it is annotated with the type `xs:anyType`.

Atomic values in the Data Model also carry type annotations. An atomic value of unknown type is annotated with the type `xs:anySimpleType`. Under certain circumstances (such as during processing of an arithmetic operator), an atomic value of `xs:anySimpleType` may be cast into a more specific type (such as `xs:double`).

This document provides a user-oriented description of how various kinds of expressions are processed. For each expression, the operands and result are instances of the Data Model. The details of transforming XML documents into the Data Model are described in [XQuery 1.0 and XPath 2.0 Data Model]. Transformation of a Data Model instance into an XML document is currently an open issue.

Two Data Model terms are described here briefly because they are of particular importance for the processing of expressions: **document order** and **typed value**. For a more detailed explanation of these terms, see [XQuery 1.0 and XPath 2.0 Data Model].

2.1.2.1 Document Order

Document order defines a total ordering among all the nodes seen by the language processor. Within a given source document, the document node is the first node, followed by element nodes, text nodes, comment nodes, and processing instruction nodes in the order of their representation in the XML document (after expansion of entities). Element nodes occur before their children. The namespace nodes of an element immediately follow the element node, in implementation-defined order. The attribute nodes of an element immediately follow its namespace nodes, and are also in implementation-defined order.

Each input document, and each node or node hierarchy constructed during expression processing, is a separate **data model fragment**. Document order among data model fragments is implementation-defined but stable within a query.

2.1.2.2 Typed Value

Every node has a **typed value**, which is a sequence of atomic values. The typed value of any node can be extracted by calling the `data` function with the node as argument. The typed value for the various kinds of nodes is defined as follows:

1. The typed value of a document, namespace, comment, or processing instruction node is the error value.
2. The typed value of a text node is the string content of the node, as an instance of `xs:anySimpleType`.
3. The typed value of an element or attribute node that has no type annotation is a sequence of atomic values that is stored in the Data Model. For details on how atomic values are created by element and attribute constructors, see **2.8.3 Data Model Representation**.
4. The typed value of an element or attribute node whose type annotation denotes either a simple type or a complex type with simple content is a sequence of atomic values that is obtained by applying the type annotation to the content of the node, as in the following examples:
    - Example: N is an element node of type `hatsizelist`, which is a complex type that includes a `country` attribute. The content of the type `hatsizelist` is a sequence of items of type `hatsize`, which is derived from `xs:decimal`. In XML Schema, this content is considered to have a simple type. The typed value of N is a sequence of values of type `hatsize`.
    - Example: A is an attribute of type `IDREFS`, a list type derived from `IDREF`, and its value is "`bar baz faz`". The typed value of A is a sequence of three atomic values of type `IDREF`.
5. The typed value of an element node whose type annotation denotes a complex type with complex content is the error value.

2.1.3 Types

XQuery is a strongly typed language with a type system based on [XML Schema]. The built-in types of XQuery include the node kinds of XML (such as element, attribute, and text nodes), the built-in atomic types of [XML Schema] (such as `xs:integer` and `xs:string`), and the following special derived types: `xs:dayTimeDuration` and `xs:yearMonthDuration` (described in [XQuery 1.0 and XPath 2.0 Functions and Operators]). Additional types may be defined in schemas and imported into a query by means of a **schema import**, as discussed in **3.1 Namespace Declarations and Schema Imports**.

When the type of a value is not appropriate for the context in which it is used, a **type exception** is raised. Languages that are based on XPath 2.0 may handle type exceptions with either a `strict` or a `flexible` policy, as described in [XPath 2.0]. XQuery has a `strict` type exception policy, which means that any expression that raises a type exception returns the error value.

Like XML Schema, XQuery distinguishes **named types** from **anonymous types**. The set of named types includes all the built-in types and all user-defined simple or complex types for which the type declaration contains a name. Named types are associated with values in one of the following ways:

- A literal value has a named type; for example, the type of the value `47` is `xs:integer`.
- The constructor functions described in [XQuery 1.0 and XPath 2.0 Functions and Operators] return values with named types; for example, `xf:date("2002-5-31")` returns a value of type `xs:date`.
- When an instance of the Data Model is constructed from a validated document, type names assigned by a schema processor are preserved in the Data Model.
- The `validate` expression invokes schema validation within a query, assigning type names in the same way that a schema processor would.
- The `cast` expression creates an atomic value with a specific named type.
- Some functions, such as `data()`, extract typed values from the nodes of a document, preserving the named types of these values.

The XQuery type system is formally defined in [XQuery 1.0 Formal Semantics]. This section presents a summary of types from a user's perspective.

2.1.3.1 Type Checking

XQuery provides two kinds of type checking, called **static type checking** and **dynamic type checking**.

Static type checking is performed during the query analysis phase (also known as "compile time.") Static type checking of an expression is based on the expression itself and on the **in-scope schema definitions**. Static type checking does not depend on the actual values found in any input document. The purpose of static type checking is to provide early detection of type errors and to compute the type of a query result.

During static type checking, each expression is assigned a static type. In some cases, the static type is derived from the lexical form of the expression; for example, the static type of the literal `5` is `xs:integer`. In other cases, the static type of an expression is inferred according to rules based on the static types of its operands; for example, the static type of the expression `size < 5` is `xs:boolean`. The static type of an expression may be either a named type or a structural description--for example, `xs:boolean?` denotes an optional occurrence of the `xs:boolean` type. The rules for inferring the static types of various expressions are described in [XQuery 1.0 Formal Semantics]. During the analysis phase, if an operand of an expression is found to have a static type that is not appropriate for that operand, a static 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 to produce either a value of type T or the error value.

Dynamic type checking is performed during the query execution phase (also known as "run time.") Dynamic checking depends on the actual values found in input documents. At run time, 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 "zero or more integers or strings," but at run time its value may have the dynamic type "integer.") If an operand of an expression is found to have a dynamic type that is not appropriate for that operand, a type exception is raised.

The dynamic type of a value may be either a structural type (such as "sequence of integers") or a named type. In many cases, a value that has a structural type can be given a named type by means of a `validate` expression, which matches the value against known types in the **in-scope schema definitions**.

It is possible for static type checking of an expression to raise a static type error, even though the expression might evaluate successfully on some valid input data. For example, an expression might contain a function that requires an element as its parameter, and static type checking might infer the static type of the function parameter to be an optional element. In this case, a static type error would result, even though the function call would be successful for input data in which the optional element is present.

It is also possible for an expression to return the error value, even though static type checking of the expression raised no error. 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, the error value will result. Similarly, an expression may apply an arithmetic operator to a value extracted from a schemaless document. This is not a static error, but at run time, if the value cannot be successfully cast to a numeric type, the error value will result.

If an implementation can determine by static analysis that an expression will necessarily return the error value (for example, because it contains a division by the constant zero), the implementation is allowed to report this error at query analysis time (as well as at query execution time).

At user option, static type checking can be disabled. Also, a conformance level may be defined for which implementions need not provide static type checking.

2.1.3.2 SequenceType

When it is necessary to refer to a type in an XQuery expression, the syntax shown below is used. This syntax production is called "SequenceType", since it describes the type of an XQuery value, which is a sequence.

[59]	SequenceType	::=	(ItemType OccurrenceIndicator) | "empty"
[60]	ItemType	::=	(("element" | "attribute") ElemOrAttrType?)
			| "node"
			| "processing-instruction"
			| "comment"
			| "text"
			| "document"
			| "item"
			| AtomicType
			| "unknown"
			| "atomic" "value"
[61]	ElemOrAttrType	::=	SchemaType | (QName SchemaContext?)
[62]	SchemaType	::=	QName? "of" "type" QName
[56]	SchemaContext	::=	"in" SchemaGlobalContext ("/" SchemaContextStep)*
[57]	SchemaGlobalContext	::=	QName | ("type" QName)
[58]	SchemaContextStep	::=	QName
[63]	AtomicType	::=	QName
[64]	OccurrenceIndicator	::=	("*" | "+" | "?")?

Here are some examples of SequenceTypes that might be used in XQuery expressions or function parameters:

- `xs:date` refers to the built-in Schema type date
- `attribute?` refers to an optional attribute
- `element` refers to any element
- `element office:letter` refers to an element with a specific name
- `element of type po:address+` refers to one or more elements of the given type
- `node*` refers to a sequence of zero or more nodes of any type
- `item*` refers to a sequence of zero or more nodes or atomic values

During processing of a query, it is sometimes necessary to determine whether a given value matches a type that was declared using the SequenceType syntax. This process is known as **SequenceType matching**. For example, an `instance of` expression returns `true` if a given value matches a given type, or `false` if it does not.

SequenceType matching between a given value and a given SequenceType is performed as follows:

If the SequenceType is `empty`, the match succeeds only if the value is an empty sequence. If the SequenceType is an ItemType with no OccurrenceIndicator, the match succeeds only if the value contains precisely one item and that item matches the ItemType (see below). If the SequenceType contains an ItemType and an OccurrenceIndicator, the match succeeds only if the number of items in the value matches the OccurrenceIndicator and each of these items matches the ItemType. As a consequence of these rules, a value that is an empty sequence matches any SequenceType whose occurrence indicator is `*` or `?`.

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

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

The process of matching a given item against a given ItemType is performed as follows (remember that an item may be a node or an atomic value):

1. The ItemType `item` matches any item. For example, `item` matches the value `1` or the value `<a/>`.
2. The following ItemTypes match atomic values:
    a. `atomic value` matches any atomic value.
    b. A named atomic type matches a value if the dynamic type of the value is the same as the named atomic type or is derived from the named atomic type by restriction. For example, the ItemType `xs:decimal` matches the value `12.34` (a decimal literal); it also matches a value whose dynamic type is `shoesize`, if `shoesize` is a user-defined atomic type derived from `xs:decimal`.
    c. `unknown` matches an atomic value whose most specific type is `xs:anySimpleType`.
3. The following ItemTypes match nodes:
    a. `node` matches any node.
    b. `text` matches any text node.
    c. `processing-instruction` matches any processing instruction node.

d. `comment` matches any comment node.

e. `document` matches any document node.

f. `element` matches an element node. Optionally, `element` may be followed by ElemOrAttrType,which places further constraints on the node (see below).

g. `attribute` matches an attribute node. Optionally, `attribute` may be followed by ElemOrAttrType,which places further constraints on the node (see below).

An **ElemOrAttrType** may be used to place a constraint on the name or type of an element or attribute, as follows:

1. One form of ElemOrAttrType, denoted by the phrase "`of type`", specifies the **required type** of the element or attribute node. The **required type** must be the QName of a simple or complex type that is found in the **in-scope schema definitions**. The match is successful only if the given element or attribute has a type annotation that is the same as the **required type** or is known (in the **in-scope schema definitions**) to be derived from the **required type**. For example, `element of type Employee` matches an element node that has been validated and has received the type annotation `Employee`. Similarly, `attribute of type xs:integer` matches an attribute node that has been validated and has received the type annotation `xs:integer`.

2. Another form of ElemOrAttrType is simply a QName, which is interpreted as the **required name** of the element or attribute. The QName must be an element or attribute name that is found in the **in-scope schema definitions**. The match is successful only if the given element or attribute has the **required name** and also conforms to the schema definition for the **required name**. This can be verified in either of the following ways:

   a. If the schema definition for the **required name** has a named type, the given element or attribute must have a type annotation that is the same as that named type or is known (in the **in-scope schema definitions**) to be derived from that named type. For example, suppose that a schema declares the element named `location` to have the type `State`. Then the SequenceType `element location` will match a given element only if its name is `location` and its type annotation is `State` or some named type that is derived from `State`.

   b. If the schema definition for the **required name** has an anonymous (unnamed) type definition, the actual content of the given element or attribute must structurally comply with this type definition. For example, suppose that a schema declares the element named `shippingAddress` to have an anonymous complex type consisting of a `street` element followed by a `city` element. Then the SequenceType `element shippingAddress` will match a given element only if its name is `shippingAddress` and its content is a `street` element followed by a `city` element.

3. The above two forms of ElemOrAttrType may be combined to specify both the **required name** and the **required type** of an element or attribute node, as in `element person of type Employee` or `attribute color of type xs:integer`.

In some cases, the **required name** of an element or attribute node may be locally declared (that is, declared inside a complex type in some schema.) In this case, the ElemOrAttrType may specify the SchemaContext in which the **required name** is to be interpreted. For example, `element shippingAddress in invoice/customer` matches an element node that conforms to the schema definition of the element name `shippingAddress`, as it would be interpreted inside a `customer` element that is inside an `invoice` element. The first QName in the SchemaContext must be found in the **in-scope schema definitions**.

[**Ed. Note:** The effect of substitution groups on SequenceType matching is an open issue.]

2.1.3.3 Type Conversions

Some expressions do not require their operands to exactly match the expected type. For example, function parameters and returns expect a value of a particular type, but allow some basic conversions to be performed, such as extraction of atomic values from nodes, promotion of numeric values, and implicit casting of untyped values. The conversion rules for function parameters and returns are discussed in **2.2.4 Function Calls**. Other operators that provide special conversion rules include arithmetic operators, which are discussed in **2.5 Arithmetic Expressions**, and value comparisons, which are discussed in **2.6.1 Value Comparisons**.

Type conversions sometimes depend on a process called **atomization**, which is used when an optional atomic value is expected. When atomization is applied to a given value, the result is either a single atomic value, an empty sequence, or a type exception. Atomization is defined as follows:

- If the value is a single atomic value or an empty sequence, atomization simply returns the value.
- If the value is a single node, the **typed value** of the node is extracted and returned; however, if the typed value is a sequence containing more than one item, a type exception is raised.
- In any other case, atomization raises a **type exception**.

2.2 Primary Expressions

Primary expressions are the basic primitives of the language. They include literals, variables, function calls, and the use of parentheses to control precedence of operators. The use of **qualifiers** in primary expressions is discussed in **2.3.2 Qualifiers**.

[28]	PrimaryExpr	::=	(Literal | FunctionCall | Variable | ParenthesizedExpr) Qualifiers

2.2.1 Literals

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

[52]	Literal	::=	NumericLiteral | StringLiteral
[51]	NumericLiteral	::=	IntegerLiteral | DecimalLiteral | DoubleLiteral
[174]	IntegerLiteral	::=	[0-9]+
[175]	DecimalLiteral	::=	("." [0-9]+) | ([0-9]+ "." [0-9]*)
[176]	DoubleLiteral	::=	(("." [0-9]+) | ([0-9]+ ("." [0-9]*)?)) ([e] | [E]) ([+] | [-])? [0-9]+
[193]	StringLiteral	::=	(["] ((" ") | [^"])* ["]) | (['] ((' ') | [^'])* ['])

The value of a **string literal** is a singleton sequence containing an item whose primitive type is `xs:string` and whose value is the string denoted by the characters between the delimiting quotation marks.

The value of a **numeric literal** containing no `"."` and no `e` or `E` character is a singleton sequence containing an item whose type is `xs:integer` and whose value is obtained by parsing the numeric literal according to the rules of the `xs:integer` datatype. The value of a numeric literal containing `"."` but no `e` or `E` character is a singleton sequence containing an item whose primitive type is `xs:decimal` and whose value is obtained by parsing the numeric literal according to the rules of the `xs:decimal` datatype. The value of a numeric literal containing an `e` or `E` character is a singleton sequence containing an item whose primitive type is `xs:double` and whose value is obtained by parsing the numeric literal according to the rules of the `xs:double` datatype.

Here are some examples of literal expressions:

- `"12.5"` denotes the string containing the characters '1', '2', '.', and '5'.
- `12` denotes the integer value twelve.
- `12.5` denotes the decimal value twelve and one half.

- `125E2` denotes the double value twelve thousand, five hundred.

Values of other XML Schema built-in types can be constructed by calling the constructor for the given type. The constructors for XML Schema built-in types are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators]. For example:

- `xf:true()` and `xf:false()` return the boolean values `true` and `false`, respectively.
- `xf:integer("12")` returns the integer value twelve.
- `xf:date("2001-08-25")` returns an item whose type is `xs:date` and whose value represents the date 25th August 2001.

It is also possible to construct values of other built-in types using the `cast` expression. For example:

- `cast as xs:positiveInteger(12)` returns an item whose primitive value is the decimal value 12.0 and whose type is the built-in derived type `xs:positiveInteger`.

2.2.2 Variables

A **variable** evaluates to the value to which the variable's QName is bound in the **evaluation context**. If the variable's QName is not bound, the value of the variable is the error value. Variables can be bound by clauses in **for expressions** and **quantified expressions**, and by function calls, which bind values to the formal parameters of functions before executing the function body.

[164]	Variable	::=	"$" QName

Ed. Note: In XQuery this production may end up being constrained to an NCName. See issue 207.

2.2.3 Parenthesized Expressions

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.

Parentheses are also used as delimiters in constructing a sequence, as described in **2.4.1 Constructing Sequences**.

2.2.4 Function Calls

A **function call** consists of a QName followed by a parenthesized list of zero or more expressions. The QName must match the name of an **in-scope function** in the **static context** (see **2.1.1 Expression Context**). The expressions inside the parentheses provide the arguments of the function call. The number of arguments must equal the number of formal parameters in the function's signature; otherwise a static error is raised.

[54]	FunctionCall	::=	(QName | "document" | "empty") "(" (Expr ("," Expr)*)? ")"

A function call expression is evaluated as follows:

1. Each argument expression is evaluated, producing an argument value.
2. Each argument value is converted to the declared type of the corresponding function parameter, using the function conversion rules listed below.
3. If the function is a built-in function, it is executed using the converted argument values. The result is a value of the function's declared return type.

4. If the function is a user-defined function, the converted argument values are bound to the formal parameters of the function, and the function body is executed. The value returned by the function body is then converted to the declared return type of the function by applying the function conversion rules.

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

1. If the required type is an AtomicType:

   Atomization is applied to the given value. If the resulting atomic value is of type `xs:anySimpleType`, an attempt is made to cast it to the required type; if the cast fails, the function call returns the error value. If the atomic value has a type that can be promoted to the required type using the promotion rules in **B.1 Type Promotion**, the promotion is done. After applying the above rules, if the resulting value does not conform to the required type, the function call returns the error value.

2. If the required type is a sequence or optional occurrence of AtomicType:

   If the given value contains any nodes, these nodes are replaced by their **typed values**. If the cardinality of the resulting sequence does not match the cardinality of the required type, a type exception is raised. Otherwise, each item is converted to the required AtomicType using the conversion rule for AtomicType.

3. If the required type is any other SequenceType:

   SequenceType Matching is used to determine if the type of the given value matches the required type. If not, the function call returns the error value.

A core library of functions is defined in [XQuery 1.0 and XPath 2.0 Functions and Operators]. Functions in this library may be invoked without a namespace prefix by assigning the default function namespace to the namespace of the function library.

The comma operator in XQuery serves two purposes: it separates the arguments in a function call, and it separates items in an expression that constructs a sequence (see **2.4.1 Constructing Sequences**). To distinguish these two uses, parentheses can be used to delimit the individual arguments of a function call. Here are some examples of function calls in which arguments that are literal sequences are delimited with parentheses:

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

2.2.5 Comments

XQuery comments can be used to provide informative annotation. These comments are lexical constructs only, and do not affect the processing of an expression.

[85] ExprComment ::= "{--" [^}]* "--}"

Comments may be used before and after major tokens within expressions and within element content. See **A.4 Lexical structure** for the exact lexical states where comments are recognized.

Ed. Note: The EBNF here should disallow "--}" within the comment, rather than "}". Also, within an enclosed expression, a comment immediately after the opening "{" will cause the "{{" to be mistaken for an escaped "{". Thus, `<a>{{--comment--}foo}</a>` will currently cause an error, and must be disambiguated with a space between the "{" characters. This should be considered an open issue.

2.3 Path Expressions

A **path expression** locates nodes within a tree, and returns a sequence of distinct nodes in **document order**. A path expression is always evaluated with respect to an **evaluation context**. There are two kinds of path expressions: **relative path expressions** and **rooted path expressions**.

[27]	RelativePathExpr	::=	Expr ("/" | "//") Expr
[26]	RootedPathExpr	::=	("/" Expr?) | ("//" Expr)

A **relative path expression** consists of two expressions, separated by `/` or `//`. This section describes the meaning of `/`; the use of `//` is described in **2.3.4 Abbreviated Syntax**.

We will refer to the expression on the left side of `/` as E1 and the expression on the right side of `/` as E2. The expression E1 is evaluated, and if the result is not a sequence of nodes, the error value is returned. 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 **2.1.1.2 Evaluation Context**. Each evaluation of E2 must result in a sequence of nodes; otherwise, the error value is returned. The sequences of nodes resulting from all the evaluations of E2 are merged, eliminating duplicate node identities and sorting the results in document order.

As an example of a relative 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.

A **rooted path expression** consists of `/` or `//`, followed by an expression. If the rooted path expression begins with `/`, the following expression is optional. This section describes the meaning of `/`; the use of `//` is described in **2.3.4 Abbreviated Syntax**.

A `/` by itself selects the **context document**. A `/` followed by an expression E1 establishes an **inner focus** in which the context node is set equal to the context document, and the context position and context length are set to 1. The expression E1 is then evaluated. If the value of E1 is a sequence of nodes, the result of the rooted path expression is this sequence of nodes, in document order, after elimination of duplicates based on nodeid. If the value of E1 is not a sequence of nodes, the error value is returned.

2.3.1 Steps

[29]	StepExpr	::=	(ForwardStep | ReverseStep) Qualifiers
[46]	ForwardStep	::=	(ForwardAxis NodeTest) | AbbreviatedForwardStep
[47]	ReverseStep	::=	(ReverseAxis NodeTest) | AbbreviatedReverseStep

A **step** is an expression that returns a sequence of nodes, in document order and without duplicates. Steps are often used inside path expressions. A step might be thought of as beginning at the context node, navigating to those nodes that are reachable from the context node via a predefined axis, and selecting some subset of the reachable nodes. A step has three parts:

- an **axis**, which specifies the relationship between the nodes selected by the step and the context node. The axis might be thought of as the "direction of movement" of the step.
- a **node test**, which specifies the node kind and/or name of the nodes selected by the step.

- zero or more **qualifiers**, which further modify the sequence of nodes selected by the step.

In the **abbreviated syntax** for a step, the axis can be omitted and other shorthand notations can be used as described in **2.3.4 Abbreviated Syntax**.

The unabbreviated syntax for an step consists of the axis name and node test separated by a double colon, followed by zero or more **qualifiers**. For example, in `child::para[position()=1]`, `child` is the name of the axis, `para` is the node test and `[position()=1]` is a qualifier.

The node sequence selected by a step is found by generating an initial node sequence from the axis and node test, and then applying each of the qualifiers in turn. The initial node sequence consists of the nodes reachable from the context node via the specified axis that have the node kind and/or name specified by the node test. For example, the step `descendant::para` selects the `para` element descendants of the context node: `descendant` specifies that each node in the initial node sequence must be a descendant of the context node, and `para` specifies that each node in the initial node sequence must be an element named `para`. The available axes are described in **2.3.1.1 Axes**. The available node tests are described in **2.3.1.2 Node Tests**. Qualifiers are described in **2.3.2 Qualifiers**. Examples of steps are provided in **2.3.3 Unabbreviated Syntax** and **2.3.4 Abbreviated Syntax**.

2.3.1.1 Axes

[36]	ForwardAxis	::=	"child" "::"
			| "descendant" "::"
			| "attribute" "::"
			| "self" "::"
			| "descendant-or-self" "::"
[37]	ReverseAxis	::=	"parent" "::"

XQuery supports the following axes:

- the `child` axis contains the children of the context node
- the `descendant` axis contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes
- the `parent` axis contains the parent of the context node, if there is one
- the `attribute` axis contains the attributes of the context node; the axis will be empty unless the context node is an element
- the `self` axis contains just the context node itself
- the `descendant-or-self` axis contains the context node and the descendants of the context node

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.

In XQuery, the `parent` axis is a reverse axis; all other axes are forward axes. Since the `self` axis always contains at most one node, it makes no difference whether it is a forward or reverse axis.

In a sequence of nodes selected by a step, the context positions of the nodes are determined in a way that depends on the axis. If the axis is a forward axis, context positions are assigned to the nodes in document order. If the axis is a reverse axis, context positions are assigned to the nodes in reverse document order. In either case, the first context position is 1.

2.3.1.2 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, processing instruction, or namespace) or on the name of the node.

[38]	NodeTest	::=	KindTest | NameTest
[39]	NameTest	::=	QName | Wildcard
[40]	Wildcard	::=	"*" | ":"? NCName ":" "*" | "*" ":" NCName
[41]	KindTest	::=	ProcessingInstructionTest | CommentTest | TextTest | AnyKindTest
[42]	ProcessingInstructionTest	::=	"processing-instruction" "(" StringLiteral? ")"
[43]	CommentTest	::=	"comment" "(" ")"
[44]	TextTest	::=	"text" "(" ")"
[45]	AnyKindTest	::=	"node" "(" ")"

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

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

A node test that is a QName is true if and only if the **kind** of the node is the principal node kind and the expanded-name of the node is equal to the expanded-name specified by the QName. For example, `child::para` selects the `para` element children of the context node; if the context node has no `para` children, it selects an empty set of nodes. `attribute::href` selects the `href` attribute of the context node; if the context node has no `href` attribute, it selects an empty set of nodes.

A QName in a node test is expanded into an expanded-name using the **in-scope namespaces** in the expression context. An unprefixed QName used as a nametest has the namespaceURI associated with the default element namespace in the expression context. It is an error if the QName has a prefix that does not correspond to any in-scope namespace.

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

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

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

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

2.3.2 Qualifiers

Qualifiers are of two general forms: **predicates**, which are used to filter a node sequence by applying some test, and **dereferences**, which are used to map reference-type nodes into the nodes that they reference.

[50]	Qualifiers	::=	(("[" Expr "]") | ("=>" NameTest))*

2.3.2.1 Predicates

A predicate consists of an expression, called a **predicate expression**, enclosed in square brackets. A predicate serves to filter a node sequence, retaining some nodes and discarding others. For each node in the node sequence to be filtered, the predicate expression is evaluated using an **inner focus** derived from that node, as described in **2.1.1.2 Evaluation Context**. The result of the predicate expression is coerced to a Boolean value, called the **predicate truth value**, as described below. Those nodes 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 an empty sequence, the predicate truth value is `false`.
2. If the value of the predicate expression is an atomic value of a numeric type, the predicate truth value is `true` if and only if the value of the predicate expression is equal to the **context position**.
3. If the value of the predicate expression is an atomic value of type Boolean, the predicate truth value is equal to the value of the predicate expression.
4. If the value of the predicate expression is a sequence that contains at least one node and does not contain any item that is not a node, the predicate truth value is `true`. The predicate truth value in this case does not depend on the content of the node(s). [**Ed. Note:** This rule is pending approval by the XSLT Working Group.]
5. In any other case, a type exception is raised.

Here are some examples of 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 whose name is "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 a "secretary" subelement:
  ```
  child::employee[secretary]
  ```

As noted in **2.2 Primary Expressions**, a predicate can also be used with a primary expression that is not a step, as illustrated in the following example:

- List all the integers from 1 to 100 that are divisible by 5.
  ```
  (1 to 100)[. mod 5 eq 0]
  ```

2.3.2.2 Dereferences

A **dereference** operates on a node sequence, mapping element and/or attribute nodes into the nodes that they reference. Every node in the input sequence of the dereference must be an element or attribute node whose typed value is of type `IDREF` or `IDREFS`; otherwise the error value is returned. The dereference generates a new sequence consisting of the element nodes that are referenced by the `IDREF` values in the input sequence. An element node is referenced by an `IDREF` value if it has an ID-type attribute whose value matches the `IDREF` value, and it is in the same document as the node containing the `IDREF` value. The referenced nodes are filtered by the NameTest that follows the dereference operator, retaining only nodes whose expanded-names match the expanded-name specified in the NameTest. The filtered sequence of element nodes is then sorted into document order and duplicate node-ids are eliminated.

Here are some examples of steps that contain dereferences:

- This example selects the figure that is referenced by the first figure reference that is a child of the context node:
  ```
  child::figref[1]/attribute::refid=>figure
  ```
- This example selects the manager of the manager of the context node, assuming that the context node is an `emp` node and that each `emp` node has a `manager` attribute that references the `emp` node of the employee's manager:
  ```
  attribute::manager=>emp/attribute::manager=>emp
  ```

2.3.3 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 **2.3.4 Abbreviated Syntax**.

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

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

2.3.4 Abbreviated Syntax

[48]	AbbreviatedForwardStep	::=	"." | ("@" NameTest) | NodeTest
[49]	AbbreviatedReverseStep	::=	".."

The abbreviated syntax permits the following abbreviations:

1. The most important abbreviation is that `child::` can be omitted from a step. In effect, `child` is the default axis. For example, a path expression `section/para` is short for `child::section/child::para`.
2. There is also an abbreviation for attributes: `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`.
3. `//` is short for `/descendant-or-self::node()/`. For example, `//para` is short for `/descendant-or-self::node()/child::para` and so will select any `para` element in the document (even a `para` element that is a document element will be selected by `//para` since the document element node is a child of the root node); `div1//para` is short for `div1/descendant-or-self::node()/child::para` and so will select all `para` descendants of `div1` children.

   Note that 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 parents.
4. A step consisting of `.` is short for `self::node()`. This is particularly useful in conjunction with `//`. For example, the path expression `.//para` is short for

`self::node()/descendant-or-self::node()/child::para`

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

5. 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.

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[last()]` selects the last `para` child of the context node
- `*/para` selects all `para` grandchildren of the context node
- `/doc/chapter[5]/section[2]` selects the second `section` of the fifth `chapter` of the `doc`
- `chapter//para` selects the `para` element descendants of the `chapter` element children of the context node
- `//para` selects all the `para` descendants of the document root and thus selects all `para` elements in the same document as the context node
- `//list/member` selects all the `member` elements in the same document as the context node that have an `list` parent
- `.` selects the context node
- `.//para` selects the `para` element descendants of the context node
- `..` selects the parent of the context node
- `../@lang` selects the `lang` attribute of the parent of the context node
- `para[@type="warning"]` selects all `para` children of the context node that have a `type` attribute with value `warning`
- `para[@type="warning"][5]` selects the fifth `para` child of the context node that has a `type`attribute with value `warning`
- `para[5][@type="warning"]` selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning`
- `chapter[title="Introduction"]` selects the `chapter` children of the context node that have one or more `title` children with string-value equal to `Introduction`
- `chapter[title]` selects the `chapter` children of the context node that have one or more `title` children
- `employee[@secretary and @assistant]` selects all the `employee` children of the context node that have both a `secretary` attribute and an `assistant` attribute

- `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 `section` element that is a child of the context node.
- `book/xf:ID(publisher)/name` returns the same result as `xf:ID(book/publisher)/name`.

2.4 Sequence Expressions

XQuery supports operators to construct and combine sequences. A **sequence** is an ordered collection of zero or more items. An **item** may be an atomic value or a node. An item is identical to a sequence of length one 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).

2.4.1 Constructing Sequences

[53]	ParenthesizedExpr	::=	"(" ExprSequence? ")"
[3]	ExprSequence	::=	Expr ("," Expr)*
[17]	RangeExpr	::=	Expr "to" Expr

One way to construct a sequence is with a **ParenthesizedExpr**, which is zero or more expressions separated by the comma (",") operator and delimited by parentheses. A parenthesized expression is evaluated by evaluating each of its constituent expressions and concatenating the resulting sequences, in order, into a single result sequence. A sequence may contain duplicate 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.

Here are some examples of expressions that construct sequences:

- This expression is a sequence of five integers:
  ```
  (10, 1, 2, 3, 4)
  ```
- This expression constructs one sequence from the sequences 10, (1, 2), the empty sequence (), and (3, 4):
  ```
  (10, (1, 2), (), (3, 4))
  ```
  It evaluates to the sequence:
  ```
  (10, 1, 2, 3, 4)
  ```
- This expression contains all `salary` children of the context node followed by all `bonus` children:
  ```
  (salary, bonus)
  ```
- Assuming that `$price` is bound to the value `10.50`, this expression:
  ```
  ($price, $price)
  ```
  evaluates to the sequence
  ```
  (10.50, 10.50)
  ```

A **RangeExpr** can be used to construct a sequence of consecutive integers. The `to` operator takes two operands, both of which have a required type of integer. A sequence is constructed containing the two integer operands and every integer between the two operands. If the first operand is less than the second, the sequence is in increasing order, otherwise it is in decreasing order.

- This example uses a range expression inside a sequence expression:

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

It evaluates to the sequence:

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

2.4.2 Combining Sequences

[20]	UnionExpr	::=	Expr ("union" | "|") Expr
[21]	IntersectExceptExpr	::=	Expr ("intersect" | "except") Expr

XQuery provides several 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 of these operators return their result sequences in document order without duplicates based based on nodeid. If an operand of `union`, `intersect`, or `except` contains an item that is not a node, the error value is returned.

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 `$seq1` is bound to a sequence containing A and B, `$seq2` is also bound to a sequence containing A and B, and `$seq3` is bound to a sequence containing B and C. Then:

- `$seq1 union $seq1` evaluates to a sequence containing A and B.
- `$seq2 union $seq3` evaluates to a sequence containing A, B, and C.
- `$seq1 intersect $seq1` evaluates to a sequence containing A and B.
- `$seq2 intersect $seq3` evaluates to a sequence containing B only.
- `$seq1 except $seq2` evaluates to the empty sequence.
- `$seq2 except $seq3` evaluates to a 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 values or nodes from a sequence.

2.5 Arithmetic Expressions

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

[18]	AdditiveExpr	::=	Expr ("+" | "-") Expr
[19]	MultiplicativeExpr	::=	Expr ("*" | "div" | "mod") Expr
[22]	UnaryExpr	::=	("-" | "+") Expr

The binary subtraction operator must be preceded by white space if it follows an NCName, in order to distinguish it from a hyphen, which is a valid name character. For example, `a-b` will be interpreted as a single token.

An arithmetic expression is evaluated by applying the following rules, in order, until an error is encountered or a value is computed:

1. **Atomization** is applied to each operand, resulting in a single atomic value or an empty sequence for each operand.

2. If either operand is an empty sequence, the result of the operation is an empty sequence.

3. If an operand has the type `xs:anySimpleType`, it is cast to `xs:double`. If the cast fails, the error value is returned.

4. If the two operands have different types, and these types can be promoted to a common type using the promotion rules in **B.1 Type Promotion**, the operands are both promoted to their least common type. For example, if the first operand is of type `hatsize` which is derived from `xs:decimal`, and the second operand is of type `shoesize` which is derived from `xs:integer`, then both operands are promoted to the type `xs:decimal`.

5. If the operand type(s) are valid for the given operator, the operator is applied to the operand(s), resulting in an atomic value or an error (for example, an error might result from dividing by zero.) The combinations of atomic types that are accepted by the various arithmetic operators, and their respective result types, are listed in **B.2 Operator Mapping**. If the operand type(s) are not valid for the given operator, a type exception is raised.

Here are some examples of arithmetic expressions:

- In general, arithmetic operations on numeric values result in numeric values:
  ```
  ($salary + $bonus) div 12
  ```
- 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 - $unit-discount
  ```
- Unary operators have higher precedence than binary operators, subject of course to the use of parentheses:
  ```
  -($bellcost + $whistlecost)
  ```

2.6 Comparison Expressions

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

[13]	ValueComp	::=	Expr ("eq" | "ne" | "lt" | "le" | "gt" | "ge") Expr
[12]	GeneralComp	::=	Expr ("=" | "!=" | "<" S | "<=" | ">" | ">=") Expr
[14]	NodeComp	::=	Expr ("is" | "isnot") Expr
[15]	OrderComp	::=	Expr ("<<" | ">>" | "precedes" | "follows") Expr

The "<" comparison operator must be followed by white space in order to distinguish it from a tag-open character. [**Ed. Note:** This rule may be relaxed, pending resolution of some general grammar issues.]

2.6.1 Value Comparisons

Value comparisons are intended for comparing single values. The result of a value comparison is defined by applying the following rules, in order:

1. **Atomization** is applied to each operand, resulting in a single atomic value or an empty sequence for each operand.

2. If either operand is an empty sequence, the result is an empty sequence.

3. If either operand has the type `xs:anySimpleType`, that operand is cast to a required type, which is determined as follows:

- ❍ If the type of the other operand is numeric, the required type is `xs:double`.
- ❍ If the type of the other operand is `xs:anySimpleType`, the required type is `xs:string`.
- ❍ Otherwise, the required type is the type of the other operand.

If the cast fails, the error value is returned.

4. If the value comparison has two numeric operands of different types, one of the operands is promoted to the type of the other operand, following the promotion rules in **B.1 Type Promotion**. For example, a value of type `xs:integer` can be promoted to `xs:decimal`, and a value of type `xs:decimal` can be promoted to `xs:double`.
5. 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`. **B.2 Operator Mapping** describes which combinations of atomic types are comparable, and how comparisons are performed on values of various types. If the value of the first operand is not comparable with the value of the second operand, a type exception is raised.

Here are some examples of value comparisons:

- The following comparison is true only if `$book1` has a single `author` subelement and its value is "Kennedy":
```
$book1/author eq "Kennedy"
```
- The following comparison is true because the two constructed nodes have the same value, even though they have different identities:
```
<a>5</a> eq <a>5</a>
```

2.6.2 General Comparisons

General comparisons are defined by adding existential semantics to value comparisons. The operands of a general comparison may be sequences of any length. The result of a general comparison is always `true` or `false`.

The general comparison `A = B` is `true` for sequences `A` and `B` if the value comparison `a eq b` is `true` for some item `a` in `A` and some item `b` in `B`. Otherwise, `A = B` is `false`.

Similarly:

- `A != B` is `true` if and only if `a ne b` is `true` for some `a` in `A` and some `b` in `B`.
- `A < B` is `true` if and only if `a lt b` is `true` for some `a` in `A` and some `b` in `B`.
- `A <= B` is `true` if and only if `a le b` is `true` for some `a` in `A` and some `b` in `B`.
- `A > B` is `true` if and only if `a gt b` is `true` for some `a` in `A` and some `b` in `B`.
- `A >= B` is `true` if and only if `a ge b` is `true` for some `a` in `A` and some `b` in `B`.

Here is an example of a general comparison:

- The following comparison is true if the value of any `author` subelement of `$book1` is "Kennedy":
```
$book1/author = "Kennedy"
```

2.6.3 Node Comparisons

The result of a node comparison is defined by applying the following rules, in order:

1. Both operands must be either a single node or an empty sequence; otherwise the error value is returned.
2. If either operand is an empty sequence, the result of the comparison is an empty sequence.
3. A comparison with the `is` operator is `true` if the two operands are nodes that have the same identity; otherwise it is `false`. A comparison with the `isnot` operator is `true` if the two operands are nodes that have different identities; otherwise it is `false`. See [XQuery 1.0 and XPath 2.0 Data Model] for a discussion of node identity.

Use of the `is` operator is illustrated below.

- The following comparison is true only if the left and right sides each evaluate to exactly the same single node:
  `//book[isbn="1558604820"] is //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>`

2.6.4 Order Comparisons

The result of an order comparison is defined by applying the following rules, in order:

1. Both operands must be either a single node or an empty sequence; otherwise the error value is returned.
2. If either operand is an empty sequence, the result of the comparison is an empty sequence.
3. A comparison with the << operator returns `true` if the first operand node is earlier than the second operand node in document order, or `false` if the first operand node is later than the second operand node in document order.
4. A comparison with the >> operator returns `true` if the first operand node is later than the second operand node in document order, or `false` if the first operand node is earlier than the second operand node in document order.
5. A comparison with the `precedes` operator returns `true` if the first operand node is earlier than the second operand node in document order and is not an ancestor of the second node; otherwise it returns `false`.
6. A comparison with the `follows` operator returns `true` if the first operand node is later than the second operand node in document order and is not an descendant of the second node; otherwise it returns `false`.

Here is an example of an order comparison:

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

2.7 Logical Expressions

A **logical expression** is either an **and-expression** or an **or-expression**. In the absence of errors, the value of a logical expression is always one of the boolean values `true` or `false`.

[6]	OrExpr	::=	Expr "or" Expr
[7]	AndExpr	::=	Expr "and" Expr

The first step in evaluating a logical expression is to reduce each of its operands to an **effective boolean value**, which is `true`, `false`, or the error value. The effective boolean value of an operand is defined as follows:

1. If the operand is an empty sequence, its effective boolean value is `false`.

2. If the operand is an atomic value of type `xs:boolean`, the operand serves as its own effective boolean value.
3. If the operand is a sequence that contains at least one node and does not contain any item that is not a node, its effective boolean value is `true`. [**Ed. Note:** This rule is pending approval by the XSLT Working Group.]
4. In any other case, a type exception is raised. In XQuery, a type exception always results in the error value.

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

	EBV_2 = true	EBV_2 = false	EBV_2 = error
EBV_1 = true	true	false	error
EBV_1 = false	false	false	false or error
EBV_1 = error	error	false or error	error

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

	EBV_2 = true	EBV_2 = false	EBV_2 = error
EBV_1 = true	true	true	true or error
EBV_1 = false	true	false	false
EBV_1 = error	true or error	false	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 return the error value if the first expression evaluated contains an error. Similarly, an and-expression can return `false` if the first expression evaluated is false, and it can return the error value if the first expression evaluated contains an error. As a result of these rules, the value of 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 = 1 and 2 = 2
  1 = 1 or 2 = 3
  ```
- The following expression may return either `false` or the error value:
  ```
  1 = 2 and 3 div 0 = 47
  ```
- The following expression may return either `true` or the error value:
  ```
  1 = 1 or 3 div 0 = 47
  ```
- The following expression returns the error value:
  ```
  1 = 1 and 3 div 0 = 47
  ```

In addition to and- and or-expressions, XQuery provides a function named `not` that takes a general sequence as parameter and returns a boolean value. The `not` function reduces its parameter to an effective boolean value using the same rules that are used for the operands of logical expressions. 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 the effective boolean value of its operand is the error value, `not` returns the error value. The `not` function is described in [XQuery 1.0 and XPath 2.0 Functions and Operators].

> **Ed. Note:** Functions named `and3`, `or3`, and `not3` may be provided to implement three-valued logic. See Issue 28.

2.8 Constructors

XQuery provides constructors that can create XML structures within a query. There are constructors for elements, attributes, CDATA sections, processing instructions, and comments. A special form of constructor called a computed element or attribute constructor can be used to create an element or attribute with a computed name.

[25]	Constructor	::=	ElementConstructor
			| XmlComment
			| XmlProcessingInstruction
			| CdataSection
			| ComputedElementConstructor
			| ComputedAttributeConstructor
[65]	ElementConstructor	::=	"<" QName AttributeList ("/>" | (">" ElementContent* "</" QName ">"))
[71]	ElementContent	::=	Char
			| "{{"
			| "}}"
			| ElementConstructor
			| EnclosedExpr
			| CdataSection
			| CharRef
			| PredefinedEntityRef
			| XmlComment
			| XmlProcessingInstruction
[72]	AttributeList	::=	(QName "=" AttributeValue)*
[73]	AttributeValue	::=	(["] (" " | AttributeValueContent)* ["])
			| (['] ("''" | AttributeValueContent)* ['])
[74]	AttributeValueContent	::=	Char
			| CharRef
			| "{{"
			| "}}"
			| EnclosedExpr
			| PredefinedEntityRef
[75]	EnclosedExpr	::=	"{" ExprSequence "}"

2.8.1 Element Constructors

An **element constructor** creates an XML element. If the name, attributes, and content of the element are all constants, the element constructor uses standard XML notation. For example, the following expression creates a `book` element that contains attributes, subelements, and text:

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

In an element constructor, the name used in an end tag must match the name of the corresponding start tag. If **namespace prefixes** are declared in the **query prolog**, the prefixes they declare may be used to create **qualified names** for elements and attributes. It is an error to use a **namespace prefix** that has not been declared.

In an element constructor, curly braces { } delimit **enclosed expressions**, distinguishing them from literal text. Enclosed expressions are evaluated and replaced by their value, whereas material outside curly braces is simply treated as literal text, as illustrated by the following example:

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

In an element constructor, an enclosed expression may evaluate to any sequence of nodes and/or atomic values. Attribute nodes occurring in this sequence become the attributes of the constructed element. The remainder of the sequence becomes the content of the constructed element.

An enclosed expression may also be used to compute the value of an attribute. If the enclosed expression returns a node, the **typed value** of the node is extracted and assigned to the attribute, as illustrated by the following example:

```
<book isbn="{$i/@booknum}" />
```

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, XQuery adopts the same convention as XSLT: Two adjacent curly braces in an XQuery character string are interpreted as a single curly brace character.

2.8.2 Computed Element and Attribute Constructors

An alternative way to create elements and attributes is by using a **computed element constructor** or a **computed attribute constructor**. In these constructors, the keyword `element` or `attribute` is followed by the name of the node to be created and then by its content. The name may be specified either by a QName or by an enclosed expression that returns a QName, and the content is specified by an enclosed expression.

[66]	ComputedElementConstructor	::=	"element" (QName | EnclosedExpr) "{" ExprSequence? "}"
[67]	ComputedAttributeConstructor	::=	"attribute" (QName | EnclosedExpr) "{" ExprSequence? "}"

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 this section:

```
element book
```

```
{
    attribute isbn { "isbn-0060229357" },
    element author
    {
        element first { "Crockett" },
        element last { "Johnson" }
    }
}
```

As the names imply, the primary purpose of computed element and attribute constructors is to allow the name of the constructed 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 sequence of entries in a translation dictionary. Here is an example entry:

```
<entry word="address">
   <variant lang="German">Adresse</variant>
   <variant lang="Italian">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
    {cast as xs:Qname
       (data($dict/entry[word=name($e)]/variant[lang="Italian"]))}
    {$e/node()}
```

The result of this expression is as follows:

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

2.8.3 Data Model Representation

Element and attribute constructors can best be understood by examining how their values are represented in the Data Model. The descriptions in this section apply to both the computed and non-computed forms of element and attribute constructors.

2.8.3.1 Constructed Element Nodes

The result of an element constructor is a new element node, with its own node identity. All the attribute and child nodes of the new element node are also new nodes with their own identities, even though they may be copies of existing nodes. The nodes created by an element constructor have no type annotation. The children of the constructed element node, and its typed value, are defined by the following rules:

1. If the content of the element constructor is a sequence of atomic values, all of which have a type other than `xs:anySimpleType`, then the new element node has a single child, which is a text node containing the lexical representation of all the atomic values, separated by blanks.

   The **typed value** of the element node is the sequence of atomic values from which the node was constructed, with their original types. Since the text node child does not contain enough type information to reconstruct this typed value, an implementation must store the element's typed value separately from the text node. This precludes a "text node only" implementation.

Note:

The costs and benefits of this strategy are an open issue (see Issue 274.)

- Example:

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

  The constructed element node has one child, a text node containing the value "`1 2 3`". The typed value of the constructed element must be represented separately, as a sequence of integers.

- Example:

  ```
  <mixture>{1, "2", "three"}</mixture>
  ```

  The constructed element node has one child, a text node, containing the value "`1 2 three`". The typed value of the constructed element must be represented separately, as an integer and two strings.

2. Otherwise, the children of the new element node are constructed as follows:

   a. First, any atomic values in the content of the new element are converted into text nodes containing their lexical representations.

   b. Next, any adjacent text nodes from the preceding step are coalesced into a single text node by concatenating their contents, with no intervening blanks.

   In this case, the **typed value** of the element node is defined to be the same as its **string value**, as an instance of `xs:anySimpleType`. The **string value** of an element node is defined as the concatenated contents of all its text node descendants, in document order.

   - 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, and a text node containing "` cats.`". The child element node in turn has a single text node child containing the value "`8`".

   - Example:

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

     The constructed `<cat>` element node has two children: a constructed element node for the `<breed>` element, and a constructed element node for the `<color>` element.

[**Ed. Note:** Future versions of this document will include more details about the handling of whitespace in element and attribute constructors. Significant issues remain open in this area, such as Issue 191.]

Note that if an element is constructed by copying another element, the new element node does not inherit the type of the original element node. Note also that it is possible to construct an element that has an attribute named `xsi:type`, but the existence of this attribute does not affect the type of the constructed node.

Newly constructed element nodes have no type annotation. A constructed element node may be given a type annotation by a `validate` expression. If the node is successfully validated against a type in the **in-scope schema definitions**, the `validate` expression returns a copy of the node with a new identity and with a specific named type. A `validate` expression may also have other side effects, such as providing default values for attributes. Under certain circumstances, some type information may be lost when an element is validated (see **2.14 Validate Expressions** for examples).

2.8.3.2 Constructed Attribute Nodes

The result of an attribute constructor is a new attribute node, with its own node identity and with no type annotation. Attribute nodes have no children. The **typed value** of the constructed attribute node is determined by the following rules:

1. If the content of the attribute constructor is a sequence of atomic values, all of which have a type other than `xs:anySimpleType`, then the typed value of the attribute node is that sequence of atomic values. Note that, under certain circumstances, type information may be lost if the attribute undergoes schema validation (see **2.14 Validate Expressions**).

    - Example:

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

      The value of the `size` attribute is the integer `7`.

2. Otherwise, the typed value of the attribute node is defined as follows:

    a. Any atomic values contained within the value of the new attribute are converted into their lexical representations.

    b. Any nodes contained within the value of the new attribute are replaced by their string values.

    c. The resulting strings are concatenated without any added whitespace, and the result is given a type annotation of `xs:anySimpleType`.

    - Example:

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

      The value of the `size` attribute is "`7`", as an instance of `xs:anySimpleType`. Note that the double quotes surrounding `7` are attribute delimiters, and do not indicate that the type of the attribute is `xs:string`. Also note that `7` is not interpreted as an integer constant because it is not inside an expression (enclosed in curly braces).

    - Example:

      ```
      <a b="{47, $emp/salary}" />
      ```

      The `b` attribute is constructed from an integer and a node. Its typed value is an instance of `xs:anySimpleType` constructed by concatenating the lexical representation of the integer with the string value of the node. For example, if the string value of `$emp/salary` is `100`, the typed value of the `b` attribute is "`47100`".

2.8.4 Other Constructors and Comments

The syntax for a **CDATA section constructor**, a **processing instruction constructor**, or an **XML comment constructor** is the same as the syntax of the equivalent XML construct.

[68]	CdataSection	::=	"<![CDATA[" Char* "]]>"
[69]	XmlProcessingInstruction	::=	"<?" PITarget Char* "?>"
[70]	XmlComment	::=	"<!--" Char* "-->"

The following example illustrates constructors for processing instructions, comments, and CDATA sections.

```
<?format role="output" ?>
<!-- Tags are ignored in the following section -->
<![CDATA[
    <address>123 Roosevelt Ave. Flushing, NY 11368</address>
]]>
```

Ed. Note: It is not clear that the Data Model can represent a CDATA section.

Note that an **XML comment** actually constructs an XML comment node. An XQuery comment (see **2.2.5 Comments**) is simply a comment used in documenting a query, and is not evaluated. Consider the following example.

```
{-- This is an XQuery comment --}
<!-- This is an XML comment -->
```

The result of evaluating the above expression is as follows.

```
<!-- This is an XML comment -->
```

2.9 FLWR Expressions

XQuery provides a FLWR expression for iteration and for binding 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 "FLWR", pronounced "flower", stands for the keywords `for`, `let`, `where`, and `return`, the four clauses found in a FLWR expression.

[8]	FLWRExpr	::=	(ForClause | LetClause)+ WhereClause? "return" Expr
[32]	ForClause	::=	"for" Variable "in" Expr ("," Variable "in" Expr)*
[33]	LetClause	::=	"let" Variable ":=" Expr ("," Variable ":=" Expr)*
[34]	WhereClause	::=	"where" Expr

The clauses of a FLWR Expression are interpreted as follows:

1. A `for` clause associates one or more variables with expressions, creating tuples of variable bindings drawn from the Cartesian product of the sequences of values to which the expressions evaluate. The variable binding tuples are generated as an ordered sequence as described below.

2. A `let` clause binds a variable directly to an entire expression. If `for` clauses are present, the variable bindings created by `let` clauses are added to the tuples generated by the `for` clauses. If there are no `for` clauses, the `let` clauses generate one tuple with all variable bindings.

3. A `where` clause can be used 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

[2.7 Logical Expressions](#).

4. The `return` clause contains an expression that is used to construct the result of the FLWR expression. The `return` clause is invoked once for every tuple generated by the `for` and `let` clauses, after eliminating any tuples that do not satisfy the conditions of a `where` clause. The expression in the `return` clause is evaluated once for every invocation, and the result of the FLWR expression is an ordered sequence containing the results of these invocations.

A variable name may not be used before it is bound, nor may it be used in the expression to which it is bound. Any variable bound in a `for` or `let` clause is in scope until the end of the FLWR expression in which it is bound. If the variable name used in the binding was already bound in the current scope, the variable name refers to the newly bound variable until that variable goes out of scope. At this point, the variable name again refers to the variable of the prior binding.

Although `for` and `let` both bind variables, the manner in which variables are bound is quite different. In a `let` clause, the variable is bound directly to the expression, and it is bound to the expression as a whole. Consider the following query:

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

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

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

Now consider a similar query which contains a `for` clause instead of a `let` clause:

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

The variable $s is associated with the expression `(<one/>, <two/>, <three/>)`, from which the variable bindings of $s will be drawn. When only one expression is present, the Cartesian product is equivalent to the sequence of values returned by that expression. In this example, the variable $s is bound three times; first it is bound to `<one/>`, then to `<two/>`, and finally to `<three/>`. One tuple is generated for each of these variable bindings, and the `return` clause is invoked for each tuple, creating the following output:

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

A FLWR Expression may contain multiple `for` clauses. In this case, the tuples of variable bindings are drawn from the Cartesian product of the sequences returned by the expressions in all the `for` clauses. The ordering of the tuples is governed by the ordering of the sequences from which they were formed, working from left to right.

The following expression illustrates how tuples are generated from the Cartesian product of expressions in a `for` clause:

```
for $i in (1, 2),
    $j in (3, 4)
return
    <tuple>
      <i>{ $i }</i>
      <j>{ $j }</j>
    </tuple>
```

Here is the result of the above expression (the order is significant).

```
<tuple>
   <i>1</i>
   <j>3</j>
</tuple>
<tuple>
   <i>1</i>
   <j>4</j>
</tuple>
<tuple>
   <i>2</i>
   <j>3</j>
</tuple>
<tuple>
   <i>2</i>
   <j>4</j>
</tuple>
```

The `unordered` function indicates that the order of a sequence is not significant and can be changed during processing of an expression. Using this function in the expressions of a `for` clause allows an implementation more freedom in optimization, and the resulting queries may be faster. The following query returns the same results as above, but the tuples may occur in any order.

```
for $i in unordered((1, 2)),
    $j in unordered((3, 4))
return
    <tuple>
      <i>{ $i }</i>
      <j>{ $j }</j>
    </tuple>
```

The following example inverts a document hierarchy to transform a bibliography into an author list. The bibliography is a list of books in which each book contains a list of authors. The example is based on the following input:

```
<bib>
  <book>
    <title>TCP/IP Illustrated</title>
    <author>
      W. Stevens
    </author>
    <publisher>Addison-Wesley</publisher>
  </book>
  <book>
    <title>Advanced Programming in the Unix environment</title>
    <author>
```

```
      W. Stevens
    </author>
    <publisher>Addison-Wesley</publisher>
  </book>
</bib>
```

The author list is a list of authors, where each author element contains the name of the author and a list of books written by that author. If an author has written more than one book, that author's name will occur more than once in the bibliography, but we want each author's name to occur only once in the author list. We will remove duplicates using the `distinct-values` function.

The `distinct-values` function takes a sequence of nodes or values as input, and returns a sequence in which duplicates have been removed by value. The order of this sequence is not significant. Two elements are considered to have duplicate values if their names, attributes, and normalized content are equal.

The following expression is used to transform the input bibliography into an author list:

```
<authlist>
 {
   let $input := document("bib.xml")
   for $a in distinct-values($input//author)
   return
     <author>
      {
        <name>
           { $a/text() }
        </name>,
        <books>
         {
           for $b in $input//book
           where $b/author = $a
           return $b/title
         }
        </books>
      }
     </author>
 }
</authlist>
```

Here is the result of the above expression.

```
<authlist>
  <author>
    <name>
      W. Stevens
    </name>
    <books>
      <title>TCP/IP Illustrated</title>
      <title>Advanced Programming in the Unix environment</title>
    </books>
  </author>
</authlist>
```

Note:

Query writers should use caution when combining FLWR expressions with path expressions. It is important to remember that path expressions always return their results in document order, whereas the order of the results of a FLWR expression are determined by the orderings of the sequences in the `for` clause.

2.10 Sorting Expressions

A sorting expression provides a way to control the order of items in a sequence.

[5]	SortExpr	::=	Expr "stable"? "sortby" "(" SortSpecList ")"
[30]	SortSpecList	::=	Expr SortModifier ("," SortSpecList)?
[31]	SortModifier	::=	("ascending" | "descending")? ("empty" "greatest" | "empty" "least")?

The value of the expression on the left side of the `sortby` keyword is called the **input sequence**. The items in the input sequence are called **input items**. The result of the sorting expression is called the **output sequence**. The output sequence contains all the input items, retaining their original identities (if any), but possibly in a different order.

The expressions on the right side of the `sortby` keyword are called **ordering expressions**. For each input item, the ordering expressions are evaluated with an **inner focus** derived from the input item, as described in **2.1.1.2 Evaluation Context**. The input items are then reordered according to the values of their respective ordering expressions. If more than one ordering expression is specified, the leftmost ordering expression controls the primary sort, followed by the remaining ordering expressions from left to right. Each ordering expression can be followed by the keyword `ascending` or `descending`, which specifies the direction of the sort (`ascending` is the default). An ordering expression can also be followed by the optional keywords `empty greatest` or `empty least`, which specify the placement of input items whose ordering expression returns the empty sequence.

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

1. **Atomization** is applied to the result of each ordering expression, resulting in a single atomic value or an empty sequence for each operand ordering expression.
2. If the result of an ordering expression has the type `xs:anySimpleType` (such as character data in a schemaless document), it is cast to the type `xs:string`.
3. Each ordering expression must return values of the same type for all input items, and this type must be a (possibly optional) atomic type for which the `gt` operator is defined--otherwise, the error value is returned.
4. For the purpose of the following rule:
    a. For any ordering expression in which `empty greatest` is specified, an ordering value that is an empty sequence is treated as greater than any non-empty ordering value (that is `( ) gt x` is true for any non-empty `x`.)
    b. For any ordering expression in which `empty least` is specified, an ordering value that is an empty sequence is treated as less than any non-empty ordering value (that is `x gt ( )` is true for any non-empty `x`.)
    c. For any ordering expression in which neither `empty greatest` nor `empty least` is specified, an implementation may consistently treat an ordering value that is an empty sequence either as greater than any non-empty ordering value or as less than any non-empty ordering value.
5. If V1 and V2 are the values of an ordering expression for input items I1 and I2 respectively, then:
    a. If the ordering expression is ascending, and if V2 `gt` V1 is true, then I1 precedes I2 in the output sequence.
    b. If the ordering expression is descending, and if V1 `gt` V2 is true, then I1 precedes I2 in the output

sequence.

c. If neither V1 `gt` V2 nor V2 `gt` V1 is true, and `stable` is specified, then the input order of I1 and I2 is preserved in output sequence.

d. If neither V1 `gt` V2 nor V2 `gt` V1 is true, and `stable` is not specified, then the order of I1 and I2 in the output sequence is implementation-defined.

Here are some examples of ordering expressions:

- This example lists all books with price greater than 100, in order by first author; within each group of books with the same first author, the books are ordered by title.

  ```
  //book[price > 100] sortby (author[1], title)
  ```

- Ordering may be specified at multiple levels of a query result. The following example returns an alphabetic list of publishers. Within each publisher element it returns a list of books, each containing a title and a price, in descending order by price.

  ```
  <publisher_list>
     {for $p in distinct-values(document("bib.xml")//publisher)
      return
         <publisher>
            <name> {$p/text()} </name>
            {for $b in document("bib.xml")//book[publisher = $p]
             return
                <book>
                   {$b/title}
                   {$b/price}
                </book>
             sortby(price descending)
            }
         </publisher>
      sortby(name)
     }
  </publisher_list>
  ```

Note:

Using a `sortby` expression may cause unexpected results when combined with a path expression. Consider the following example:

```
(employees sortby (salary))/name
```

One might expect that this query would return a list of employee names sorted in the order of their respective salaries. However, path expressions always return a node sequence in document order. Therefore, the result of this query is a list of employee names in document order. If the names are desired to be in salary order, the query should be rewritten as follows:

```
for $e in (employees sortby (salary)) return $e/name
```

2.11 Conditional Expressions

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

[11]	IfExpr	::=	"if" "(" Expr ")" "then" Expr "else" Expr

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

The first step in processing a conditional expression is to find the **effective boolean value** of the test expression, as defined in 2.7 Logical Expressions.

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 first ("then") result expression is returned. If the effective boolean value of the test expression is `false`, the value of the second ("else") result expression is returned.

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
  ```

2.12 Quantified Expressions

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

[9]	QuantifiedExpr	::=	("some" | "every") Variable "in" Expr ("," Variable "in" Expr)* "satisfies" Expr

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 values. As in the case of a for-clause in a FLWR-expression, the in-clauses generate tuples of variable bindings, using values drawn from the Cartesian product of the sequences returned by the binding expressions. Conceptually, the test expression is evaluated for each tuple of variable bindings. Results depend on the **effective boolean values** of the test expressions, as defined in 2.7 Logical Expressions. 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 order in which test expressions are evaluated for the various binding tuples is implementation-defined. 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 return an error as soon as it finds one binding tuple for which the test expression returns 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 return an error as soon as it finds one binding tuple for which the test expression returns 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 //part satisfies $part/@discounted
  ```

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

  ```
  some $emp in //employee satisfies ($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 return either `true` or the error value, since its test expression returns `true` for one variable binding and the error value for another:

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

- This quantified expression may return either `false` or the error value, since its test expression returns `false` for one variable binding and the error value for another:

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

2.13 Expressions on Datatypes

SequenceTypes occur explicitly in several kinds of XQuery expressions.

[16]	InstanceofExpr	::=	Expr "instance" "of" SequenceType
[10]	TypeswitchExpr	::=	"typeswitch" "(" Expr ")" ("as" Variable)? CaseClause+ "default" "return" Expr
[35]	CaseClause	::=	"case" SequenceType "return" Expr
[24]	CastExpr	::=	("cast" "as" | "treat" "as" | "assert" "as") SequenceType ParenthesizedExpr

The boolean operator `instance of` returns `true` if the value of its first operand matches the type named in its second operand, according to the rules for **SequenceType Matching**; otherwise it returns `false`. Examples:

- `$x instance of element of type animal`

  This example returns `true` if the value associated with variable `$x` is an element whose content is a value of the type `animal`.

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

  This example returns `false` because the given value is not an integer; instead, it is an element whose value is an integer.

- `<a>5</a> instance of element of type xs:integer`

  This example returns `true` because the given value matches the given type according to the rules for **SequenceType Matching**.

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 operand expression can be followed by an `as` clause that defines a variable, called the **operand variable**, that binds to the value of the operand expression. 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** 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 is not a value of any type named in a `case` clause, the value of the typeswitch expression is the value of the `return` expression in the `default` clause.

The `return` expressions in `case` and `default` clauses typically contain references to the operand variable (defined in the `as` clause). If the operand expression consists of a single variable, it can serve as the operand variable and the `as` clause can be omitted. The `as` clause can also be omitted if the `return` expressions do not contain references to the operand variable.

The following example shows how a typeswitch expression might be used to invoke one of several functions depending on the type of a variable.

```
typeswitch ($animal)
     case element duck return quack($animal)
     case element dog return woof($animal)
     default return "No sound"
```

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 SequenceType, called the **target type**. The `cast` expression first performs **atomization** on its input expression, resulting in a single atomic input value or the empty sequence. If the input value is the empty sequence, the `cast` expression returns an empty sequence. Otherwise, the `cast` expression creates a new instance of the target type that is equal to the input value. The type of the input value is called the **input type**. `cast` is supported only for certain combinations of input type and target type, as listed below:

- `cast` is supported for the combinations of input type and target type listed in [XQuery 1.0 and XPath 2.0 Functions and Operators]. 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 type `xs:decimal`.
- A value of a derived atomic type can always be cast into its base type. For example, if the type `shoesize` is derived by restriction from the type `xs:integer`, a value of type `shoesize` can always be cast into the type `xs:integer`.

For any combination of input type and target type that is not in the above list, a `cast` expression returns the error value.

If the input value is not in the value space of the target type, the error value is returned. A `cast` expression also returns the error value if any facet of the target type is not satisfied. For example, `cast as xs:integer($x)` returns the error value if the value of `$x` is `4.99` because this value cannot be represented with zero fractional digits.

Note:

The working group has declared an intention to support casts to user-defined target types, subject to certain restrictions. Work is in progress to define the semantics of these casts.

In addition to `cast`, XQuery provides type-checking expressions called `treat` and `assert`. Each of these expressions takes two operands: an expression and a **SequenceType**. Unlike `cast`, however, `treat` and `assert` do not change the type or value of their operands. Instead, the purpose of a `treat` or `assert` expression is to ensure that its operand has an expected type. Each of these expressions has semantics that are applied during static type-checking, and additional semantics that are applied during expression evaluation. If static type checking is disabled or not supported, only the run-time semantics of `treat` and `assert` apply. The semantics of the two expressions are as follows:

Semantics of `treat as type1 (expr2)`:

- During static type checking:

  `type1` must be a subtype of the static type of `expr2`, using the definition of subtype in [XQuery 1.0 Formal Semantics]--otherwise, a static error is raised. 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 (at "run-time"):

  If `expr2` matches `type1`, using the SequenceType Matching rules in [id-sequencetype], the `treat` expression returns the value of `expr2`; otherwise, it returns the error value. If the value of `expr2` 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:

  ```
  treat as element of type USAddress ($myaddress)
  ```

  The static type of `$myaddress` may be `element of type Address`, a less specific type than `element of type USAddress`. However, at run-time, the value of `$myaddress` must match the type `element of type USAddress` using SequenceType Matching rules; otherwise the error value is returned.

Semantics of `assert as type1 (expr2)`:

- During static type checking:

  The static type of `expr2` must be a subtype of `type1`, using the definition of subtype in [XQuery 1.0 Formal Semantics]--otherwise, a static error is raised. The static type of the `assert` expression is `type1`. `assert` provides a stronger static type guarantee than `treat`.

- During expression evaluation (at "run-time"):

  If `expr2` matches `type1`, using the **SequenceType Matching** rules in [id-sequencetype], the `assert` expression returns the value of `expr2`; otherwise, it returns the error value. If the value of `expr2` is returned, its identity is preserved. If the `assert` expression has passed static type checking without an error, and `expr2` does not return an error, then the `assert` expression will never return the error value at run-time.

- Example:

  ```
  assert as element of type USAddress ($myaddress)
  ```

  The static type of `$myaddress` must be `element of type USAddress` or one of its proper subtypes, and at run-time, the value of `$myaddress` must match the type `element of type USAddress` using **SequenceType Matching** rules; otherwise the error value is returned.

2.14 Validate Expressions

[23]	ValidateExpr	::=	"validate" SchemaContext? "{" Expr "}"

Ed. Note: Curly braces in this syntax may cause problems if this expression is embedded in XSLT--see Issue 267.)

A `validate` expression validates its argument with respect to the **in-scope schema definitions**, using the schema validation process described in [XML Schema]. The argument of a validate expression may be any sequence of elements. Validation replaces element and attribute nodes with new nodes that have their own identity and that contain type annotations and defaults created by the validation process. If a node that has a parent is validated, the parent of the original node will not be the parent of the validated node.

To allow locally declared elements and attributes to be validated, a schema context may optionally be specified. If no context is specified, all top-level names in the material to be validated are treated as global names.

The following example creates and validates a globally-declared `person` element:

```
validate {
   <person>
     <name>
       <first>Elvira</first>
       <last>Fischbein</last>
     </name>
   </person>
}
```

The `validate` expression invokes the full schema validation process, except that identity constraints, as defined in section 3.11.4 of [XML Schema] Part 1, are not applied. All facets of simple types are checked, and default values are supplied as defined in the XML Schema specification.

Validating an expression is equivalent to the following series of steps:

1. Serialize the value of the expression, using `xsi:type` attributes to indicate the types of elements that have type annotations or that contain a single atomic value of a known type (however, do not generate an `xsi:type` attribute for any element that already has such an attribute).
2. Invoke schema validation on the resulting serialized expression, using the **in-scope schema definitions**.
3. Remove any `xsi:type` attributes that were generated in Step 1.

A validate expression may contain a SchemaContext that is used in validating locally declared elements and attributes. When a schema context is supplied, all element QNames are interpreted as they would be if found in that context in an XML document. If the schema context begins with a QName, the QName is interpreted as the name of a globally declared element; however, if the schema context begins with the keyword `type`, the first QName is interpreted as the name of a globally declared type. The steps inside the schema context trace a path relative to the globally declared element or type, as illustrated in the following examples, which are based on schemas defined in [XML Schema], Part 0:

- Suppose that `$x` is bound to a `shipTo` element. Then `validate in po:purchaseOrder {$x}` validates the value of `$x` in the context of the global element declaration `po:purchaseOrder`.
- Suppose that `$y` is bound to an `productName` element. Then `validate in po:purchaseOrder/items/item {$y}` validates the value of `$y` in the context of an `item` element, inside an `items` element, inside the global element declaration `po:purchaseOrder`.
- Suppose that `$z` is bound to a `zip` element. Then `validate in type po:USAddress {$z}` validates the value of `$z` in the context of the global type declaration `po:USAddress`.

Under certain circumstances, validation of an element or attribute may result in the loss of some type information. The following examples illustrate some of these circumstances:

- `validate { <sizes>{1, 2, 3}</sizes> }`

  The constructed `<sizes>` element node has no type annotation. Its value is a sequence of integers, but "sequence of integers" is not a built-in type that can be represented by an `xsi:type` attribute.

The effective serialized value to be validated is `<sizes>1 2 3</sizes>`. After validation, if no more specific type is found, the value of the `<sizes>` element will be "`1 2 3`", an instance of `xs:anySimpleType`.

- `validate { <animal>{ "A cat", "named Oscar" }</animal> }`

  The constructed `animal` element node has no type annotation. Its value was derived from two strings, but the boundaries between the strings will be lost during validation.

  The effective serialized value to be validated is `<animal>A cat named Oscar</animal>`. If `animal` is a global element declared to have `xs:string` as its type, the validated value of the element will consist of a single string. On the other hand, if the declared type of `animal` is `xs:string*`, the validated value of the element will consist of the following four strings: "`A`", "`cat`", "`named`", and "`Oscar`".

- `validate { <mixture>{ 1, "2", "three" }</mixture> }`

  The constructed `<mixture>` element node has no type annotation. Its typed value consists of an integer and two strings, but there is no possible schema type that can be assigned to the element node that will preserve this type information.

  The effective serialized value to be validated is `<mixture>1 2 three</mixture>`. After validation, the value of the `<mixture>` element might be two integers and a string, or three strings, or one string.

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

  The constructed `size` attribute node has no type annotation. Its value is an integer, but there is no way to represent the type of a serialized attribute.

  The effective serialized value to be validated is `<shoe size="7"/>`. After validation, if no more specific type is found, the value of the `size` attribute will be "`7`", an instance of `xs:anySimpleType`.

3 The Query Prolog

The **Query Prolog** is a series of declarations and definitions that affect query processing. The Query Prolog can be used to define namespaces, import definitions from schemas, and define functions.

[1]	Query	::=	QueryProlog ExprSequence?
[2]	QueryProlog	::=	(NamespaceDecl | DefaultNamespaceDecl | SchemaImport)* FunctionDefn*

We describe the Query Prolog in two sections, one on Namespace Declarations and Schema Imports, and one on Function Definitions.

3.1 Namespace Declarations and Schema Imports

[76]	NamespaceDecl	::=	"namespace" QName "=" StringLiteral
[77]	DefaultNamespaceDecl	::=	"default" ("element" | "function") "namespace" "=" StringLiteral
[80]	SchemaImport	::=	"schema" (StringLiteral | NamespaceDecl | DefaultNamespaceDecl) ("at" StringLiteral)?

Ed. Note: The QName in NamespaceDecl should be a NCName. However, there is currently a technical problem with token ambiguity between QName and NCName.

A **Namespace Declaration** defines a namespace prefix and associates it with a namespace URI, adding the (prefix, URI) pair to the set of in-scope namespaces. The namespace URI must be a valid URI, and may not be an empty string. The namespace declaration is in scope for the rest of the query in which it is declared. Consider the following query:

```
namespace foo = "http://www.foo.com"
<foo:bar> Lentils </foo:bar>
```

In the query result, the newly created node is in the namespace associated with the namespace URI `http://www.foo.com`.

In element constructors, namespace declaration attributes also associate a namespace with a prefix, adding a (prefix, URI) pair to the set of in-scope namespaces. In the data model, a namespace declaration is not an attribute, and it will not be retrieved by queries that return the attributes of an element. Namespace declarations are in scope within their containing element. Nested elements and attributes inherit the in-scope namespaces of their parents. The following query creates the same result as the previous query.

```
<foo:bar xmlns:foo="http://www.foo.com"> Lentils </foo:bar>
```

Because namespace declarations are in-scope within their containing element, they may be used in expressions that occur within an element constructor, as in the following query.

```
<foo:bar xmlns:foo="http://www.foo.com">{ //foo:bing }</foo:bar>
```

Names are compared on the basis of the expanded name (see [XML Names] for this and other namespace terms), not the QName. When element or attribute names are compared, they are considered identical if the local part and namespace URI match. Namespace prefixes are disregarded in name comparisons. This is illustrated by the following example:

```
namespace xx = "http://www.foo.com"

let $i := <foo:bar xmlns:foo = "http://www.foo.com">
              <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://www.foo.com"`. 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> Lentils </foo:bing>
```

It is invalid to redefine namespace prefixes using the **NamespaceDecl** production.

```
{-- Error: attempt to redefine 'xx' in NamespaceDecl --}

namespace xx = "http://www.foo.com"
namespace xx = "http://www.bar.com"

//xx:bing
```

It is also invalid to use a QName with a namespace prefix that has not been declared. The following query is also semantically invalid.

```
{-- Error: use of undeclared namespace prefix --}
//xx:bing
```

Namespace declaration attributes may redefine a namespace prefix within a given scope. The following query is valid.

```
namespace xx = "http://www.fee.com"
```

```
<xx:bar xmlns:xx = "http://www.fie.com">
    <xx:bing xmlns:xx = "http://www.fo.com"> One </xx:bing>
    <xx:bing xmlns:xx = "http://www.fum.com"> Two </xx:bing>
    <xx:bing> Three </xx:bing>
</xx:bar>
```

The result of the above query is as follows.

```
<xx:bar xmlns:xx = "http://www.fie.com">
    <xx:bing xmlns:xx = "http://www.fo.com"> One </xx:bing>
    <xx:bing xmlns:xx = "http://www.fum.com"> Two </xx:bing>
    <xx:bing xmlns:xx = "http://www.fie.com"> Three </xx:bing>
</xx:bar>
```

Default Namespace Declarations can be used to define namespace URIs to be associated with unprefixed names. The following kinds of default namespace declarations are supported:

- `default element namespace` defines a namespace URI that is associated with unprefixed names of elements and types.
- `default function namespace` defines a namespace URI that is associated with unprefixed names of functions.

If no default element namespace is in effect, unqualified names of elements and types are in no namespace. If no default function namespace is in effect, unqualified function names are in no namespace. Unqualified attribute names are always in no namespace, since XQuery provides no way to declare a default namespace for attributes.

The following example illustrates a default element namespace:

```
default element namespace = "http://www.foo.com"
<bar> Lentils </bar>
```

The result of the above query is shown below. Note that the name of the newly created element is in the namespace associated with the namespace URI `http://www.foo.com`, even though no namespace prefix occurs in the query.

```
 <bar xmlns = "http://www.foo.com"> Lentils </bar>
```

A **Schema Import** imports the element and attribute declarations and type definitions from a schema, mapping them into the Query Data Model using rules that will be specified in a future edition of [XQuery 1.0 Formal Semantics]. The URI in a schema import specifies the namespace to be imported, and optionally the location of the schema in which the namespace is defined. Importing a schema has no effect on the in-scope namespaces, since it does not associate a prefix with the namespace. When a schema is imported, the query generally accompanies the schema import with appropriate `namespace` or `default namespace` declarations to make it possible to refer to the names defined in the schema.

The following query searches for table elements in an XHTML document, after declaring the namespace and schema location for XHTML.

```
schema "http://www.w3.org/1999/xhtml"
            at "http:/www.w3.org/1999/xhtml/xhtml.xsd"
namespace xhtml = "http://www.w3.org/1999/xhtml"

document("aspect.xhtml")//xhtml:table
```

A shorthand notation is provided to allow a schema to be imported, and a namespace prefix to be bound to the target namespace of the schema, in a single step. This shorthand notation is illustrated by the following example, which is equivalent to the previous example:

```
schema namespace xhtml="http://www.w3.org/1999/xhtml"
            at "http:/www.w3.org/1999/xhtml/xhtml.xsd"

document("aspect.xhtml")//xhtml:table
```

The shorthand notation includes two URI's: the first one identifies a namespace and the second identifies a schema location. If the given namespace is not the target namespace of the schema at the given location, an error results.

It is an error to import two schemas that both define the same name in the same symbol space and in the same scope. For instance, a query may not import two schemas that provide global element declarations for two elements with the same expanded name.

3.2 Function Definitions

In addition to the built-in functions described in [XQuery 1.0 and XPath 2.0 Functions and Operators], XQuery allows users to define functions of their own. A function definition 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 **2.1.3.2 SequenceType**. A function definition also includes an expression called the **function body** that defines how the result of the function is computed from its parameters.

[78]	FunctionDefn	::=	"define" "function" QName "(" ParamList? ")" ("returns" SequenceType)? EnclosedExpr
[79]	ParamList	::=	Param ("," Param)*
[55]	Param	::=	SequenceType? Variable

The name of a function may be qualified with a namespace. The default namespace for functions is the namespace of the XML Query 1.0 and XPath 2.0 Functions and Operators, so these functions can be used without prefixes. The default namespace for functions may be changed by a default namespace declaration, as in this example:

```
default function namespace = "www.mylib.com"
```

If a function parameter is declared using a name but no type, it accepts arguments of any type. If the `returns` clause is omitted from a function definition, the function may return a value of any type.

The following example illustrates the definition and use of a 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.

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

summary(document("acme_corp.xml")//employee[location = "Denver"])
```

The type of a function parameter or result may be a global type (declared as a top-level typename in some schema) or a local type (declared at some level of nesting inside a schema). Function parameters and results of local type can be declared with the help of a SchemaContext, as in the following example:

```
define function price(element product in type catalog $p)
  returns element USPrice in type catalog/product
{
   $p/USPrice
}
```

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 **2.2.4 Function Calls**

A function may be defined recursively--that is, it may reference its own definition. Mutually recursive functions, whose bodies reference each other, are also allowed. The following example defines a recursive function that computes the maximum depth of an element hierarchy, and calls the function to find the maximum depth of a particular document. In its definition, the user-defined function `depth` calls the built-in functions `empty` and `max`.

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

```
define function depth(element $e) returns xs:integer
{
   {-- An empty element has depth 1 --}
   {-- Otherwise, add 1 to max depth of children --}
   if (empty($e/*)) then 1
   else max(for $c in $e/* return depth($c)) + 1
}

depth(document("partlist.xml"))
```

In XQuery 1.0, user-defined functions may not be overloaded. Only one function definition may have a given name. We consider function overloading to be a useful and important feature that deserves further study in future versions of XQuery. Although XQuery does not allow overloading of user-defined functions, some of the built-in functions in the XQuery core library are overloaded--for example, the `string` function of XPath can convert an instance of almost any type into a string.

Note:

If a future version of XQuery supports function overloading, an ambiguity may arise between a function that takes a node as parameter and a function with the same name that takes an atomic value as parameter (since a function call automatically extracts the atomic value of a node when necessary). The designers of such a future version of XQuery can avoid this ambiguity by writing suitable rules to govern function overloading. Nevertheless, users who are concerned about this possibility may choose to explicitly extract atomic values from nodes when calling functions that expect atomic values.

4 Example Applications

In previous sections, we have focused on explaining the meaning of the syntactic constructs of XQuery. This section contains examples of several important classes of queries that can be expressed using the syntax described in earlier sections. In some cases we describe functions introduced to support specific usage scenarios. In others, we show particular ways to combine operators that have already been introduced. The applications described here include filtering a document to produce a table of contents, joins across multiple data sources, grouping and aggregates, and queries based on sequential relationships in documents.

4.1 Filtering

One of the functions in the XQuery core function library is called `filter`. This function takes a single parameter which can be any expression. The function evaluates its argument and returns a shallow copy of the nodes that are selected by the argument, preserving any relationships that exist among these nodes. For example, suppose that the argument to `filter` is a path expression that selects nodes X, Y, and Z from some document. Suppose that, in the original document, nodes Y and Z are descendants (at any level) of node X. Then the result of `filter` is a copy of node X, with copies of nodes Y and Z as its immediate children. Any other intervening nodes from the original document are not present in the result. The name `filter` suggests a function that operates on a document to extract the parts that are of interest and discard the remainder, while retaining the structure of the original document.

The semantics of `filter` are illustrated by Figure 1. Suppose that the left side of Figure 1 represents a node hierarchy that is bound to the variable `$doc`. The right side of Figure 1 shows the result of the function `filter($doc//(A | B))`. The result contains copies of all nodes of type A and B in the original hierarchy, with their original relationships preserved. Note that the action of the `filter` function may split a node hierarchy into multiple hierarchies (preserving the sequential relationships among the root nodes of the resulting hierarchies.)

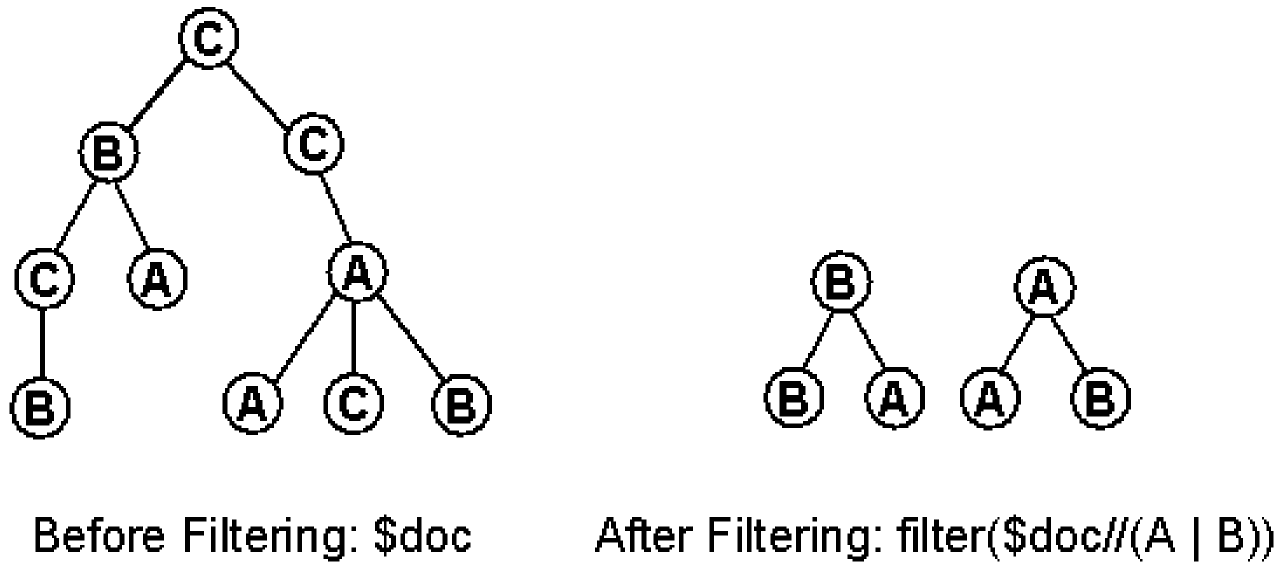

Figure 1: Action of the filter function

The following example illustrates how `filter` might be used to compute a table of contents for a document that contains many levels of nested sections. The query filters the document, retaining only section elements, title elements nested directly inside section elements, and the text of those title elements. Other elements, such as paragraphs and figure titles, are eliminated, leaving only the "skeleton" of the document. The example generates a table of contents for a document named `cookbook.xml`.

```
<toc>
   {
   filter(document("cookbook.xml") //
      (section | section/title | section/title/text()))
   }
</toc>
```

4.2 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 document("catalog.xml")//:item,
         $p in document("parts.xml")//part[partno = $i/partno],
         $s in document("suppliers.xml")//supplier[suppno = $i/suppno]
     return
        <item>
           {
           $p/description,
           $s/suppname,
           $i/price
           }
        </item>
     sortby(description, suppname)
   }
</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 document("suppliers.xml")//supplier
return
   <supplier>
      {
        $s/suppname,
        for $i in document("catalog.xml")//:item
                 [suppno = $s/suppno],
            $p in document("parts.xml")//part
                 [partno = $i/pno]
        return $p/description
        sortby(.)
      }
   </supplier>
sortby(suppname)
```

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 document("suppliers.xml")//supplier
    return
        <supplier>
           {
             $s/suppname,
             for $i in document("catalog.xml")//:item
                     [suppno = $s/suppno],
                 $p in document("parts.xml")//part
                     [partno = $i/partno]
             return
                <part>
                   {
                     $p/description,
                     $i/price
                   }
                </part>
             sortby (description)
           }
        </supplier>
    sortby (suppname)
    ,
    {-- parts that have no supplier --}
    <orphan-parts>
       { for $p in document("parts.xml")//part
         where empty(document("catalog.xml")//:item
               [partno = $p/partno] )
         return $p/description
         sortby (.)
       }
    </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.

4.3 Grouping

Many queries involve forming data into groups and applying some aggregation function such as `count` or `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 distinct-values(document("catalog.xml")//partno)
let $i := document("catalog.xml")//:item[partno = $pn]
where count($i) >= 3
return
   <well-supplied-item>
      {$pn}
      <avgprice> {avg($i/price)} </avgprice>
   </well-supplied-item>
sortby(partno)
```

The `distinct-values` function in this query eliminates duplicate part numbers from the set of all part numbers in the catalog document. The result of `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 `count($i)` and `avg($i/price)`. The query uses an element constructor to enclose each part number and average price in a containing element called `well-supplied-item`.

4.4 Queries on Sequence

XQuery uses the `precedes`, `follows`, `<<`, and `>>` operators to express conditions based on sequence. Although these operators are quite simple, they can be used to express sophisticated queries for XML documents in which sequence is meaningful. The first two queries in this section involve a surgical report that contains `procedure`, `incision`, and `anesthesia` elements. The following query returns a critical sequence that contains all elements and nodes found between the first and second incisions of the first procedure.

```
<critical-sequence>
 {
    let $proc := //procedure[1]
    for $n in $proc//node()
    where $n follows ($proc//incision)[1]
      and $n precedes ($proc//incision)[2]
    return $n
 }
</critical-sequence>
```

The following query reports incisions for which no prior anesthesia was recorded in the surgical report.

```
for $p in //procedure
where some $i in $proc//incision satisfies
        empty($proc//anesthesia[. precedes $i])
return $p
```

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)[. follows $intro][1]
return
    <div>
      {
        $intro,
        if (empty($next-h))
          then //node()[. follows $intro]
          else //node()[. follows $intro and . precedes $next-h]
      }
    </div>
```

Note that the above query also makes explicit the hierarchy that was implicit in the original document.

A Complete BNF

A.1 Grammar

The following grammar uses the same Basic EBNF notation as [XML], except that grammar symbols always have initial capital letters. The EBNF contains only non-terminals, and all terminals are presented in a separate table.

Note:

Note that the Semicolon character is reserved for future use.

NON-TERMINALS

[1]	Query	::=	QueryProlog ExprSequence?
[2]	QueryProlog	::=	(NamespaceDecl | DefaultNamespaceDecl | SchemaImport)* FunctionDefn*
[3]	ExprSequence	::=	Expr (Comma Expr)*
[4]	Expr	::=	SortExpr | OrExpr | AndExpr | FLWRExpr | QuantifiedExpr | TypeswitchExpr | IfExpr | GeneralComp | ValueComp | NodeComp | OrderComp | InstanceofExpr | RangeExpr | AdditiveExpr | MultiplicativeExpr | UnionExpr

			| IntersectExceptExpr | UnaryExpr | ValidateExpr | CastExpr | Constructor | RootedPathExpr | RelativePathExpr | PrimaryExpr | StepExpr
[5]	SortExpr	::=	Expr Stable? Sortby Lpar SortSpecList Rpar
[6]	OrExpr	::=	Expr Or Expr
[7]	AndExpr	::=	Expr And Expr
[8]	FLWRExpr	::=	(ForClause | LetClause)+ WhereClause? Return Expr
[9]	QuantifiedExpr	::=	(Some | Every) Variable In Expr (Comma Variable In Expr)* Satisfies Expr
[10]	TypeswitchExpr	::=	Typeswitch Lpar Expr Rpar (As Variable)? CaseClause+ Default Return Expr
[11]	IfExpr	::=	If Lpar Expr Rpar Then Expr Else Expr
[12]	GeneralComp	::=	Expr (Equals | NotEquals | Lt | LtEquals | Gt | GtEquals) Expr
[13]	ValueComp	::=	Expr (FortranEq | FortranNe | FortranLt | FortranLe | FortranGt | FortranGe) Expr
[14]	NodeComp	::=	Expr (Is | IsNot) Expr
[15]	OrderComp	::=	Expr (LtLt | GtGt | Precedes | Follows) Expr
[16]	InstanceofExpr	::=	Expr Instanceof SequenceType
[17]	RangeExpr	::=	Expr To Expr
[18]	AdditiveExpr	::=	Expr (Plus | Minus) Expr
[19]	MultiplicativeExpr	::=	Expr (Star | Div | Mod) Expr
[20]	UnionExpr	::=	Expr (Union | Vbar) Expr
[21]	IntersectExceptExpr	::=	Expr (Intersect | Except) Expr
[22]	UnaryExpr	::=	(Minus | Plus) Expr
[23]	ValidateExpr	::=	Validate SchemaContext? Lbrace Expr Rbrace
[24]	CastExpr	::=	(CastAs | TreatAs | AssertAs) SequenceType ParenthesizedExpr
[25]	Constructor	::=	ElementConstructor | XmlComment | XmlProcessingInstruction | CdataSection | ComputedElementConstructor | ComputedAttributeConstructor
[26]	RootedPathExpr	::=	(Slash Expr?) | (SlashSlash Expr)
[27]	RelativePathExpr	::=	Expr (Slash | SlashSlash) Expr
[28]	PrimaryExpr	::=	(Literal | FunctionCall | Variable | ParenthesizedExpr) Qualifiers
[29]	StepExpr	::=	(ForwardStep | ReverseStep) Qualifiers
[30]	SortSpecList	::=	Expr SortModifier (Comma SortSpecList)?
[31]	SortModifier	::=	(Ascending | Descending)? (EmptyGreatest | EmptyLeast)?

```
[32]  ForClause                     ::=  For Variable In Expr (Comma Variable In Expr)*
[33]  LetClause                     ::=  Let Variable ColonEquals Expr (Comma Variable ColonEquals Expr)*
[34]  WhereClause                   ::=  Where Expr
[35]  CaseClause                    ::=  Case SequenceType Return Expr
[36]  ForwardAxis                   ::=  AxisChild
                                         | AxisDescendant
                                         | AxisAttribute
                                         | AxisSelf
                                         | AxisDescendantOrSelf
[37]  ReverseAxis                   ::=  AxisParent
[38]  NodeTest                      ::=  KindTest |  NameTest
[39]  NameTest                      ::=  QName |  Wildcard
[40]  Wildcard                      ::=  Star |  NCNameColonStar |  StarColonNCName
[41]  KindTest                      ::=  ProcessingInstructionTest
                                         | CommentTest
                                         | TextTest
                                         | AnyKindTest
[42]  ProcessingInstructionTest     ::=  ProcessingInstructionLpar StringLiteral? Rpar
[43]  CommentTest                   ::=  CommentLpar Rpar
[44]  TextTest                      ::=  TextLpar Rpar
[45]  AnyKindTest                   ::=  NodeLpar Rpar
[46]  ForwardStep                   ::=  (ForwardAxis NodeTest) |  AbbreviatedForwardStep
[47]  ReverseStep                   ::=  (ReverseAxis NodeTest) |  AbbreviatedReverseStep
[48]  AbbreviatedForwardStep        ::=  Dot |  (At NameTest) |  NodeTest
[49]  AbbreviatedReverseStep        ::=  DotDot
[50]  Qualifiers                    ::=  ((Lbrack Expr Rbrack) |  (Arrow NameTest))*
[51]  NumericLiteral                ::=  IntegerLiteral |  DecimalLiteral |  DoubleLiteral
[52]  Literal                       ::=  NumericLiteral |  StringLiteral
[53]  ParenthesizedExpr             ::=  Lpar ExprSequence? Rpar
[54]  FunctionCall                  ::=  (QName |  Document |  Empty) Lpar (Expr (Comma Expr)*)? Rpar
[55]  Param                         ::=  SequenceType? Variable
[56]  SchemaContext                 ::=  In SchemaGlobalContext (Slash SchemaContextStep)*
[57]  SchemaGlobalContext           ::=  QName |  (Type QName)
[58]  SchemaContextStep             ::=  QName
[59]  SequenceType                  ::=  (ItemType OccurrenceIndicator) |  Empty
```

[60]	ItemType	::=	((Element | Attribute) ElemOrAttrType?)
			| Node
			| ProcessingInstruction
			| Comment
			| Text
			| Document
			| Item
			| AtomicType
			| Unknown
			| AtomicValue
[61]	ElemOrAttrType	::=	SchemaType | (QName SchemaContext?)
[62]	SchemaType	::=	QName? OfType QName
[63]	AtomicType	::=	QName
[64]	OccurrenceIndicator	::=	(Star | Plus | QMark)?
[65]	ElementConstructor	::=	StartTagOpen TagQName AttributeList (EmptyTagClose
			| (StartTagClose ElementContent* EndTagOpen TagQName EndTagClose))
[66]	ComputedElementConstructor	::=	Element (QName | EnclosedExpr) Lbrace ExprSequence? Rbrace
[67]	ComputedAttributeConstructor	::=	Attribute (QName | EnclosedExpr) Lbrace ExprSequence? Rbrace
[68]	CdataSection	::=	CdataSectionStart Char* CdataSectionEnd
[69]	XmlProcessingInstruction	::=	ProcessingInstructionStart PITarget Char* ProcessingInstructionEnd
[70]	XmlComment	::=	XmlCommentStart Char* XmlCommentEnd
[71]	ElementContent	::=	Char
			| LCurlyBraceEscape
			| RCurlyBraceEscape
			| ElementConstructor
			| EnclosedExpr
			| CdataSection
			| CharRef
			| PredefinedEntityRef
			| XmlComment
			| XmlProcessingInstruction
[72]	AttributeList	::=	(TagQName ValueIndicator AttributeValue)*
[73]	AttributeValue	::=	(OpenQuot (EscapeQuot | AttributeValueContent)* CloseQuot)
			| (OpenApos (EscapeApos | AttributeValueContent)* CloseApos)
[74]	AttributeValueContent	::=	Char
			| CharRef
			| LCurlyBraceEscape
			| RCurlyBraceEscape
			| EnclosedExpr
			| PredefinedEntityRef
[75]	EnclosedExpr	::=	Lbrace ExprSequence Rbrace
[76]	NamespaceDecl	::=	Namespace QName Equals StringLiteral
[77]	DefaultNamespaceDecl	::=	Default (Element | Function) Namespace Equals StringLiteral

[78]	FunctionDefn	::=	DefineFunction QName Lpar ParamList? Rpar (Returns SequenceType)? EnclosedExpr
[79]	ParamList	::=	Param (Comma Param)*
[80]	SchemaImport	::=	Schema (StringLiteral | NamespaceDecl | DefaultNamespaceDecl) (AtKeyword StringLiteral)?

A.2 Reserved Words

The XQuery grammar currently defines reserved words that may not be used as element or function names without an escaping mechanism (see note on QName escaping). The following defines the set of reserved words.

'or', 'and', 'at', 'div', 'mod', 'in', 'satisfies', 'return', 'then', 'else', 'default', 'namespace', 'to', 'where', 'intersect', 'union', 'except', 'precedes', 'follows', 'as', 'case', 'returns', 'function', 'element', 'item', 'attribute', 'type', 'node', 'empty', 'ref', 'schema', 'is', 'isnot', 'eq', 'ne', 'gt', 'ge', 'lt', 'le', 'for', 'let', 'validate', 'comment', 'document', 'text', 'unknown', 'processing-instruction', 'if', 'typeswitch', 'sortby', 'stable', 'ascending', 'descending'

Ed. Note: See issue xpath-issue-reserved-words.

A.3 Precedence Order

Ed. Note: It is likely that the Working Groups will specify a system of built-in precedence in future drafts, at least for the normative BNF in this appendix. It is recognized that the current table-driven approach contains some bugs.

In the following table, operators with a higher precedence number are more tightly bound than operators with a lower precedence number. Operators listed at the same level are evaluated from left to right.

Precedence Rules for Expr

Precedence#	Productions
1	SortExpr
2	OrExpr
3	AndExpr
4	FLWRExpr, QuantifiedExpr, TypeswitchExpr, IfExpr
5	GeneralComp, ValueComp, NodeComp, OrderComp
6	InstanceofExpr
7	RangeExpr
8	AdditiveExpr
9	MultiplicativeExpr
10	UnionExpr
11	IntersectExceptExpr
12	UnaryExpr
13	ValidateExpr
14	CastExpr
15	Constructor
16	RootedPathExpr
17	RelativePathExpr

18	PrimaryExpr
19	StepExpr

A.4 Lexical structure

A **token** as defined by this specification is a lexical unit specified by the TERMINALS section. A **token symbol** is the symbolic name given to that token. A **token part** is the most atomic part of a token. For instance, AxisDescendantOrSelf has two token parts, "descendant-or-self" and "::".

For readability, Whitespace may be used in expressions even though not explicitly allowed by the grammar: Whitespace may be freely added between tokens and token parts, except in a few cases where whitespace is needed to disambiguate the token:

- A < symbol, when used as a less-than sign in comparisons, must always be followed by whitespace to distinguish it from the opening character in a tag. In tags, whitespace may not occur after the < symbol.
- In XML, "-" is a valid character in an element or attribute name. When used as an operator after the characters of a name, it must be separated from the name, eg by using whitespace or parentheses.

A **lexical state** is a condition that is created by a previous token, or else is the starting state of the lexer. Tokens are often only recognized in a specific lexical state, and a token in turn may cause the lexer to transition to a different state. The details of these state transitions are given in the section on Lexical States.

When tokenizing, the longest possible token is always returned that would be valid in the current syntactic context.. If there is an ambiguity between two tokens that isn't resolved by the longest match rule, the token that has a lower grammar number is more overrides a token with a higher grammar number.

All keywords and tokens are case sensitive.

ExprComment tokens should be ignored by the parser.

TERMINALS

[83]	XmlCommentStart	::=	"<!--"
[84]	XmlCommentEnd	::=	"-->"
[85]	ExprComment	::=	"{--" [^}]* "--}"
[86]	S	::=	WhitespaceChar+
[87]	ProcessingInstructionStart	::=	"<?"
[88]	ProcessingInstructionEnd	::=	"?>"
[89]	AxisChild	::=	"child" "::"
[90]	AxisDescendant	::=	"descendant" "::"
[91]	AxisParent	::=	"parent" "::"
[92]	AxisAttribute	::=	"attribute" "::"
[93]	AxisSelf	::=	"self" "::"
[94]	AxisDescendantOrSelf	::=	"descendant-or-self" "::"
[95]	DefineFunction	::=	"define" "function"
[96]	Or	::=	"or"
[97]	And	::=	"and"
[98]	AtKeyword	::=	"at"
[99]	Div	::=	"div"
[100]	Mod	::=	"mod"

[101]	In	::=	"in"
[102]	Satisfies	::=	"satisfies"
[103]	Return	::=	"return"
[104]	Then	::=	"then"
[105]	Else	::=	"else"
[106]	Default	::=	"default"
[107]	Namespace	::=	"namespace"
[108]	To	::=	"to"
[109]	Where	::=	"where"
[110]	Intersect	::=	"intersect"
[111]	Union	::=	"union"
[112]	Except	::=	"except"
[113]	Precedes	::=	"precedes"
[114]	Follows	::=	"follows"
[115]	As	::=	"as"
[116]	Case	::=	"case"
[117]	Instanceof	::=	"instance" "of"
[118]	Returns	::=	"returns"
[119]	Function	::=	"function"
[120]	Element	::=	"element"
[121]	Item	::=	"item"
[122]	Attribute	::=	"attribute"
[123]	OfType	::=	"of" "type"
[124]	AtomicValue	::=	"atomic" "value"
[125]	Type	::=	"type"
[126]	Node	::=	"node"
[127]	Empty	::=	"empty"
[128]	Ref	::=	"ref"
[129]	Schema	::=	"schema"
[130]	Nmstart	::=	Letter | "_"
[131]	Nmchar	::=	Letter | CombiningChar | Extender | Digit | "." | "-" | "_"
[132]	Star	::=	"*"
[133]	NCNameColonStar	::=	":"? NCName ":" "*"
[134]	StarColonNCName	::=	"*" ":" NCName
[135]	Slash	::=	"/"
[136]	SlashSlash	::=	"//"
[137]	Equals	::=	"="
[138]	Is	::=	"is"
[139]	NotEquals	::=	"!="
[140]	IsNot	::=	"isnot"
[141]	LtEquals	::=	"<="
[142]	LtLt	::=	"<<"
[143]	GtEquals	::=	">="
[144]	GtGt	::=	">>"
[145]	FortranEq	::=	"eq"
[146]	FortranNe	::=	"ne"
[147]	FortranGt	::=	"gt"
[148]	FortranGe	::=	"ge"

[149]	FortranLt	::=	"lt"
[150]	FortranLe	::=	"le"
[151]	ColonEquals	::=	":="
[152]	Lt	::=	"<" S
[153]	Gt	::=	">"
[154]	Minus	::=	"-"
[155]	Plus	::=	"+"
[156]	QMark	::=	"?"
[157]	Arrow	::=	"=>"
[158]	Vbar	::=	"|"
[159]	Lpar	::=	"("
[160]	At	::=	"@"
[161]	Lbrack	::=	"["
[162]	Rpar	::=	")"
[163]	Rbrack	::=	"]"
[164]	Variable	::=	"$" QName
[165]	Some	::=	"some"
[166]	Every	::=	"every"
[167]	For	::=	"for"
[168]	Let	::=	"let"
[169]	CastAs	::=	"cast" "as"
[170]	AssertAs	::=	"assert" "as"
[171]	TreatAs	::=	"treat" "as"
[172]	Validate	::=	"validate"
[173]	Digits	::=	[0-9]+
[174]	IntegerLiteral	::=	Digits
[175]	DecimalLiteral	::=	("." Digits) | (Digits "." [0-9]*)
[176]	DoubleLiteral	::=	(("." Digits) | (Digits ("." [0-9]*)?)) ([e] | [E]) ([+] | [-])? Digits
[177]	Comment	::=	"comment"
[178]	Document	::=	"document"
[179]	Text	::=	"text"
[180]	Unknown	::=	"unknown"
[181]	ProcessingInstruction	::=	"processing-instruction"
[182]	NodeLpar	::=	"node" "("
[183]	CommentLpar	::=	"comment" "("
[184]	TextLpar	::=	"text" "("
[185]	ProcessingInstructionLpar	::=	"processing-instruction" "("
[186]	If	::=	"if"
[187]	Typeswitch	::=	"typeswitch"
[188]	Comma	::=	","
[189]	SemiColon	::=	
[190]	EscapeQuot	::=	" "
[191]	OpenQuot	::=	["]
[192]	CloseQuot	::=	["]
[193]	StringLiteral	::=	(["] ((" ") | [^"])* ["]) | (['] ((' ') | [^'])* ['])
[194]	Dot	::=	"."
[195]	DotDot	::=	".."
[196]	Sortby	::=	"sortby"

[197]	Stable	::=	"stable"
[198]	Ascending	::=	"ascending"
[199]	Descending	::=	"descending"
[200]	EmptyGreatest	::=	"empty" "greatest"
[201]	EmptyLeast	::=	"empty" "least"
[202]	PITarget	::=	NCName
[203]	NCName	::=	Nmstart Nmchar*
[204]	QName	::=	":"? NCName (":" NCName)?
[205]	CdataSectionStart	::=	"<![CDATA["
[206]	CdataSectionEnd	::=	"]]>"
[207]	PredefinedEntityRef	::=	"&" ("lt" | "gt" | "amp" | "quot" | "apos") ";"
[208]	HexDigits	::=	([0-9] | [a-f] | [A-F])+
[209]	CharRef	::=	"&#" (Digits | ("x" HexDigits)) ";"
[210]	StartTagOpen	::=	"<"
[211]	StartTagClose	::=	">"
[212]	EmptyTagClose	::=	"/>"
[213]	EndTagOpen	::=	"</"
[214]	EndTagClose	::=	">"
[215]	ValueIndicator	::=	"="
[216]	TagQName	::=	QName
[217]	Lbrace	::=	"{"
[218]	Rbrace	::=	"}"
[219]	LCurlyBraceEscape	::=	"{{"
[220]	RCurlyBraceEscape	::=	"}}"
[221]	EscapeApos	::=	""""
[222]	OpenApos	::=	[']
[223]	CloseApos	::=	[']
[224]	Char	::=	([#x0009] | [#x000D] | [#x000A] | [#x0020-#xFFFD])
[225]	WhitespaceChar	::=	([#x0009] | [#x000D] | [#x000A] | [#x0020])
[226]	Letter	::=	BaseChar | Ideographic
[227]	BaseChar	::=	([#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6]
			| [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131]
			| [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E]
			| [#x0180-#x01C3] | [#x01CD-#x01F0] | [#x01F4-#x01F5]
			| [#x01FA-#x0217] | [#x0250-#x02A8] | [#x02BB-#x02C1] | [#x0386]
			| [#x0388-#x038A] | [#x038C] | [#x038E-#x03A1] | [#x03A3-#x03CE]
			| [#x03D0-#x03D6] | [#x03DA] | [#x03DC] | [#x03DE] | [#x03E0]
			| [#x03E2-#x03F3] | [#x0401-#x040C] | [#x040E-#x044F]
			| [#x0451-#x045C] | [#x045E-#x0481] | [#x0490-#x04C4]
			| [#x04C7-#x04C8] | [#x04CB-#x04CC] | [#x04D0-#x04EB]
			| [#x04EE-#x04F5] | [#x04F8-#x04F9] | [#x0531-#x0556] | [#x0559]
			| [#x0561-#x0586] | [#x05D0-#x05EA] | [#x05F0-#x05F2]
			| [#x0621-#x063A] | [#x0641-#x064A] | [#x0671-#x06B7]
			| [#x06BA-#x06BE] | [#x06C0-#x06CE] | [#x06D0-#x06D3] | [#x06D5]
			| [#x06E5-#x06E6] | [#x0905-#x0939] | [#x093D] | [#x0958-#x0961]
			| [#x0985-#x098C] | [#x098F-#x0990] | [#x0993-#x09A8]
			| [#x09AA-#x09B0] | [#x09B2] | [#x09B6-#x09B9] | [#x09DC-#x09DD]
			| [#x09DF-#x09E1] | [#x09F0-#x09F1] | [#x0A05-#x0A0A]
			| [#x0A0F-#x0A10] | [#x0A13-#x0A28] | [#x0A2A-#x0A30]

```
                                  | [#x0A32-#x0A33] | [#x0A35-#x0A36] | [#x0A38-#x0A39]
                                  | [#x0A59-#x0A5C] | [#x0A5E] | [#x0A72-#x0A74] | [#x0A85-#x0A8B]
                                  | [#x0A8D] | [#x0A8F-#x0A91] | [#x0A93-#x0AA8] | [#x0AAA-#x0AB0]
                                  | [#x0AB2-#x0AB3] | [#x0AB5-#x0AB9] | [#x0ABD] | [#x0AE0]
                                  | [#x0B05-#x0B0C] | [#x0B0F-#x0B10] | [#x0B13-#x0B28]
                                  | [#x0B2A-#x0B30] | [#x0B32-#x0B33] | [#x0B36-#x0B39] | [#x0B3D]
                                  | [#x0B5C-#x0B5D] | [#x0B5F-#x0B61] | [#x0B85-#x0B8A]
                                  | [#x0B8E-#x0B90] | [#x0B92-#x0B95] | [#x0B99-#x0B9A] | [#x0B9C]
                                  | [#x0B9E-#x0B9F] | [#x0BA3-#x0BA4] | [#x0BA8-#x0BAA]
                                  | [#x0BAE-#x0BB5] | [#x0BB7-#x0BB9] | [#x0C05-#x0C0C]
                                  | [#x0C0E-#x0C10] | [#x0C12-#x0C28] | [#x0C2A-#x0C33]
                                  | [#x0C35-#x0C39] | [#x0C60-#x0C61] | [#x0C85-#x0C8C]
                                  | [#x0C8E-#x0C90] | [#x0C92-#x0CA8] | [#x0CAA-#x0CB3]
                                  | [#x0CB5-#x0CB9] | [#x0CDE] | [#x0CE0-#x0CE1] | [#x0D05-#x0D0C]
                                  | [#x0D0E-#x0D10] | [#x0D12-#x0D28] | [#x0D2A-#x0D39]
                                  | [#x0D60-#x0D61] | [#x0E01-#x0E2E] | [#x0E30] | [#x0E32-#x0E33]
                                  | [#x0E40-#x0E45] | [#x0E81-#x0E82] | [#x0E84] | [#x0E87-#x0E88]
                                  | [#x0E8A] | [#x0E8D] | [#x0E94-#x0E97] | [#x0E99-#x0E9F]
                                  | [#x0EA1-#x0EA3] | [#x0EA5] | [#x0EA7] | [#x0EAA-#x0EAB]
                                  | [#x0EAD-#x0EAE] | [#x0EB0] | [#x0EB2-#x0EB3] | [#x0EBD]
                                  | [#x0EC0-#x0EC4] | [#x0F40-#x0F47] | [#x0F49-#x0F69]
                                  | [#x10A0-#x10C5] | [#x10D0-#x10F6] | [#x1100] | [#x1102-#x1103]
                                  | [#x1105-#x1107] | [#x1109] | [#x110B-#x110C] | [#x110E-#x1112]
                                  | [#x113C] | [#x113E] | [#x1140] | [#x114C] | [#x114E] | [#x1150]
                                  | [#x1154-#x1155] | [#x1159] | [#x115F-#x1161] | [#x1163] | [#x1165]
                                  | [#x1167] | [#x1169] | [#x116D-#x116E] | [#x1172-#x1173] | [#x1175]
                                  | [#x119E] | [#x11A8] | [#x11AB] | [#x11AE-#x11AF] | [#x11B7-#x11B8]
                                  | [#x11BA] | [#x11BC-#x11C2] | [#x11EB] | [#x11F0] | [#x11F9]
                                  | [#x1E00-#x1E9B] | [#x1EA0-#x1EF9] | [#x1F00-#x1F15]
                                  | [#x1F18-#x1F1D] | [#x1F20-#x1F45] | [#x1F48-#x1F4D]
                                  | [#x1F50-#x1F57] | [#x1F59] | [#x1F5B] | [#x1F5D] | [#x1F5F-#x1F7D]
                                  | [#x1F80-#x1FB4] | [#x1FB6-#x1FBC] | [#x1FBE] | [#x1FC2-#x1FC4]
                                  | [#x1FC6-#x1FCC] | [#x1FD0-#x1FD3] | [#x1FD6-#x1FDB]
                                  | [#x1FE0-#x1FEC] | [#x1FF2-#x1FF4] | [#x1FF6-#x1FFC] | [#x2126]
                                  | [#x212A-#x212B] | [#x212E] | [#x2180-#x2182] | [#x3041-#x3094]
                                  | [#x30A1-#x30FA] | [#x3105-#x312C] | [#xAC00-#xD7A3])
[228]  Ideographic           ::=  ([#x4E00-#x9FA5] | [#x3007] | [#x3021-#x3029])
[229]  CombiningChar         ::=  ([#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486]
                                  | [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-#x05BD] | [#x05BF]
                                  | [#x05C1-#x05C2] | [#x05C4] | [#x064B-#x0652] | [#x0670]
                                  | [#x06D6-#x06DC] | [#x06DD-#x06DF] | [#x06E0-#x06E4]
                                  | [#x06E7-#x06E8] | [#x06EA-#x06ED] | [#x0901-#x0903] | [#x093C]
                                  | [#x093E-#x094C] | [#x094D] | [#x0951-#x0954] | [#x0962-#x0963]
                                  | [#x0981-#x0983] | [#x09BC] | [#x09BE] | [#x09BF] | [#x09C0-#x09C4]
                                  | [#x09C7-#x09C8] | [#x09CB-#x09CD] | [#x09D7] | [#x09E2-#x09E3]
                                  | [#x0A02] | [#x0A3C] | [#x0A3E] | [#x0A3F] | [#x0A40-#x0A42]
                                  | [#x0A47-#x0A48] | [#x0A4B-#x0A4D] | [#x0A70-#x0A71]
                                  | [#x0A81-#x0A83] | [#x0ABC] | [#x0ABE-#x0AC5] | [#x0AC7-#x0AC9]
                                  | [#x0ACB-#x0ACD] | [#x0B01-#x0B03] | [#x0B3C] | [#x0B3E-#x0B43]
                                  | [#x0B47-#x0B48] | [#x0B4B-#x0B4D] | [#x0B56-#x0B57]
                                  | [#x0B82-#x0B83] | [#x0BBE-#x0BC2] | [#x0BC6-#x0BC8]
                                  | [#x0BCA-#x0BCD] | [#x0BD7] | [#x0C01-#x0C03] | [#x0C3E-#x0C44]
```

			| [#x0C46-#x0C48] | [#x0C4A-#x0C4D] | [#x0C55-#x0C56] | [#x0C82-#x0C83] | [#x0CBE-#x0CC4] | [#x0CC6-#x0CC8] | [#x0CCA-#x0CCD] | [#x0CD5-#x0CD6] | [#x0D02-#x0D03] | [#x0D3E-#x0D43] | [#x0D46-#x0D48] | [#x0D4A-#x0D4D] | [#x0D57] | [#x0E31] | [#x0E34-#x0E3A] | [#x0E47-#x0E4E] | [#x0EB1] | [#x0EB4-#x0EB9] | [#x0EBB-#x0EBC] | [#x0EC8-#x0ECD] | [#x0F18-#x0F19] | [#x0F35] | [#x0F37] | [#x0F39] | [#x0F3E] | [#x0F3F] | [#x0F71-#x0F84] | [#x0F86-#x0F8B] | [#x0F90-#x0F95] | [#x0F97] | [#x0F99-#x0FAD] | [#x0FB1-#x0FB7] | [#x0FB9] | [#x20D0-#x20DC] | [#x20E1] | [#x302A-#x302F] | [#x3099] | [#x309A])
[230]	Digit	::=	([#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9] | [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F] | [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF] | [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F] | [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29])
[231]	Extender	::=	([#x00B7] | [#x02D0] | [#x02D1] | [#x0387] | [#x0640] | [#x0E46] | [#x0EC6] | [#x3005] | [#x3031-#x3035] | [#x309D-#x309E] | [#x30FC-#x30FE])

A.4.1 Lexical States

XQuery has lexical states that distinguish literal XML from XQuery expressions. In literal XML, any name may be used as an element name or attribute name, even if it is a reserved word in XQuery's expression syntax. To make this possible, XQuery uses separate lexical states for start tags and end tags. Similarly, strings that would be treated as a reserved keyword in XQuery's expression syntax are treated as normal character sequences when they occur in element content, attribute values, processing instructions, XML comments, or CDATA sections. The following example contains the string "FOR" many times, but this string is always interpreted the same way that it would be by a native XML parser, not as an XQuery keyword:

```
<for for="for">
    for
    <!-- for -->
    <?for for="for"?>
    <![CDATA[ for ]]>
</for>
```

An XQuery tokenizer begins in the DEFAULT state, which recognizes XQuery expressions rather than literal XML. In the DEFAULT state, the "<" character, when not followed by whitespace, marks a transition to native XML syntax. When followed by whitespace, it is treated as the less-than sign, and there is no state transition.

Before changing state, the tokenizer pushes the current state to a stack. In the DEFAULT state, the strings"<!--", "<?", and "<![CDATA[" are recognized as tokens which are associated with transitions to the XML_COMMENT, PROCESSING_INSTRUCTION, and CDATA_SECTION states, respectively. In these states, the strings "-->", "?>", and "]]>" are recognized as tokens that mark the end of the relevant XML construct, at which point the tokenizer pops the state.

Element constructors also push the current state, popping it at the conclusion of an end tag. However, several lexical transitions occur between the beginning and end of an element constructor. The character "<", unless followed by whitespace, "!--", "?", or "![CDATA[", is treated as the beginning of a start tag, which is associated with a transition to the START_TAG state. This state allows attributes in the native XML syntax. In the START_TAG state, the string ">" is recognized as a token which is associated with the transition to the ELEMENT_CONTENT state. In the ELEMENT_CONTENT state, the string "</" is interpreted as the beginning of an end tag, which is associated with a transition to the END_TAG state. When the end tag is terminated, the state is popped to the state that was pushed at

the start of the corresponding start tag.

In element content or attribute values, ie in the ELEMENT_CONTENT, QUOT_ATTRIBUTE_CONTENT, or APOS_ATTRIBUTE_CONTENT states, the character "{" marks a transition to the DEFAULT state. Before making this transition, the current state is pushed so that it can be popped when the corresponding "}" is encountered. To allow curly braces to be used as character content, a double left or right curly brace is interpreted as a single curly brace character.

Any lexical pattern that is not recognized under a given state will be an unrecognized token, and thus an error.

An operator that immediately follows a "/" or "//" when used as a root symbol, should not parse. For example, the following expression, which the expression writer may intend to parse as `(/) * foo` (value of the root element multiplied by the value of the foo element) would parse as `(/*) foo`, and thus would be an error:

```
/ * foo
```

This must instead be written as:

```
(/) * foo
```

TRANSITION STATES

tokens	recognize state	next state	action
WhitespaceChar Nmstart NCName Nmchar Digits Letter BaseChar Ideographic CombiningChar Digit Extender HexDigits S			
QName Star NCNameColonStar StarColonNCName TextLpar CommentLpar NodeLpar ProcessingInstructionLpar	QNAME DEFAULT	DEFAULT	
Type Node Empty Ref Schema Attribute Item Element Function Returns Case As Follows Precedes Except Union Intersect Where To Namespace Default Else Then Return Satisfies In Mod Div And Or Comment Document Text Node Sortby Stable Ascending Descending Lpar	DEFAULT	DEFAULT	
At AxisChild AxisDescendant AxisParent AxisAttribute AxisSelf AxisDescendantOrSelf	DEFAULT	QNAME	
XmlCommentStart	DEFAULT ELEMENT_CONTENT	XML_COMMENT	pushState

XmlCommentEnd	XML_COMMENT		popState
ExprComment	DEFAULT ELEMENT_CONTENT		
ProcessingInstructionStart	DEFAULT ELEMENT_CONTENT	PROCESSING_INSTRUCTION	pushState
ProcessingInstructionEnd	PROCESSING_INSTRUCTION		popState
PITarget	PROCESSING_INSTRUCTION		
CdataSectionStart	ELEMENT_CONTENT	CDATA_SECTION	
CdataSectionEnd	ELEMENT_CONTENT CDATA_SECTION	ELEMENT_CONTENT	
PredefinedEntityRef CharRef	ELEMENT_CONTENT QUOT_ATTRIBUTE_CONTENT APOS_ATTRIBUTE_CONTENT		
StartTagOpen	DEFAULT ELEMENT_CONTENT QUOT_ATTRIBUTE_CONTENT APOS_ATTRIBUTE_CONTENT	START_TAG	pushState
StartTagClose	START_TAG	ELEMENT_CONTENT	
EmptyTagClose	START_TAG		popState
EndTagOpen	ELEMENT_CONTENT	END_TAG	
EndTagClose	END_TAG		popState
TagQName	START_TAG END_TAG		
ValueIndicator	START_TAG		
Lbrace	START_TAG END_TAG ELEMENT_CONTENT QUOT_ATTRIBUTE_CONTENT APOS_ATTRIBUTE_CONTENT DEFAULT	DEFAULT	pushState
LCurlyBraceEscape RCurlyBraceEscape	ELEMENT_CONTENT QUOT_ATTRIBUTE_CONTENT APOS_ATTRIBUTE_CONTENT		
Rbrace	DEFAULT		popState
OpenQuot	START_TAG	QUOT_ATTRIBUTE_CONTENT	
EscapeQuot	QUOT_ATTRIBUTE_CONTENT	QUOT_ATTRIBUTE_CONTENT	
CloseQuot	QUOT_ATTRIBUTE_CONTENT	START_TAG	
OpenApos	START_TAG	APOS_ATTRIBUTE_CONTENT	
EscapeApos	APOS_ATTRIBUTE_CONTENT	APOS_ATTRIBUTE_CONTENT	
CloseApos	APOS_ATTRIBUTE_CONTENT	START_TAG	
Char	ELEMENT_CONTENT CDATA_SECTION QUOT_ATTRIBUTE_CONTENT APOS_ATTRIBUTE_CONTENT PROCESSING_INSTRUCTION XML_COMMENT XQUERY_COMMENT		

B Type Promotion and Operator Mapping

B.1 Type Promotion

Under certain circumstances, an atomic value can be promoted from one type to another. The promotion rules listed below are used in basic type conversions (see **2.1.3.3 Type Conversions**) and during processing of arithmetic expressions (see **2.5 Arithmetic Expressions**) and value comparisons (see **2.6.1 Value Comparisons**). The rules may be applied transitively. For example, since `xs:integer` is promotable to `xs:decimal` and `xs:decimal` is promotable to `xs:float`, it follows that `xs:integer` is promotable to `xs:float`, without necessarily passing through the intermediate `xs:decimal` type.

The following type promotions are permitted:

1. A value of type `xs:decimal` can be promoted to the type `xs:float`.
2. A value of type `xs:float` can be promoted to the type `xs:double`.
3. A value of a derived type can be promoted to its base type. As an example of this rule, a value of the derived type `xs:integer` can be promoted to its base type `xs:decimal`.

B.2 Operator Mapping

The tables in this section list the combinations of datatypes for which the various operators of XQuery are defined. For each valid combination of datatypes, the table indicates the function(s) that are used to implement the operator and the datatype of the result. Definitions of the functions can be found in [XQuery 1.0 and XPath 2.0 Functions and Operators]. Note that in some cases the function does not implement the full semantics of the given operator. For a complete description 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.

In the following tables, the term **numeric** refers to the types `xs:integer`, `xs:decimal`, `xs:float`, and `xs:double`. When the result type of an operator is listed as numeric, it means "same as the highest type of any input operand, in promotion order." For example, when invoked with operands of type `xs:integer` and `xs:float`, the binary `+` operator returns a result of type `xs:float`.

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

Binary Operators

Operator	Type(A)	Type(B)	Function	Result type
A + B	numeric	numeric	op:numeric-add(A, B)	numeric
A + B	xs:date	xs:yearMonthDuration	op:add-yearMonthDuration-to-date(A, B)	xs:date
A + B	xs:yearMonthDuration	xs:date	op:add-yearMonthDuration-to-date(B, A)	xs:date
A + B	xs:date	xs:dayTimeDuration	op:add-dayTimeDuration-to-date(A, B)	xs:date
A + B	xs:dayTimeDuration	xs:date	op:add-dayTimeDuration-to-date(B, A)	xs:date
A + B	xs:time	xs:dayTimeDuration	op:add-dayTimeDuration-to-time(A, B)	xs:time
A + B	xs:dayTimeDuration	xs:time	op:add-dayTimeDuration-to-time(B, A)	xs:time
A + B	xs:datetime	xs:yearMonthDuration	op:add-yearMonthDuration-to-dateTime(A, B)	xs:dateTime
A + B	xs:yearMonthDuration	xs:datetime	op:add-yearMonthDuration-to-dateTime(B, A)	xs:dateTime
A + B	xs:datetime	xs:dayTimeDuration	op:add-dayTimeDuration-to-dateTime(A, B)	xs:dateTime
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	(missing)	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	(missing)	xs:dayTimeDuration
A - B	xs:time	xs:dayTimeDuration	op:subtract-dayTimeDuration-from-time(A, B)	xs:time
A - B	xs:datetime	xs:datetime	(missing)	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	xs:decimal	op:multiply-yearMonthDurations(A, B)	xs:yearMonthDuration
A * B	xs:decimal	xs:yearMonthDuration	op:multiply-yearMonthDurations(B, A)	xs:yearMonthDuration
A * B	xs:dayTimeDuration	xs:decimal	op:multiply-dayTimeDurations(A, B)	xs:dayTimeDuration
A * B	xs:decimal	xs:dayTimeDuration	op:multiply-dayTimeDurations(B, A)	xs:dayTimeDuration
A div B	numeric	numeric	op:numeric-divide(A, B)	numeric
A div B	xs:yearMonthDuration	xs:decimal	op:divide-yearMonthDurations(A, B)	xs:yearMonthDuration
A div B	xs:dayTimeDuration	xs:decimal	op:divide-dayTimeDurations(A, B)	xs:dayTimeDuration
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(xf:compare(A, B), 1)	xs:boolean
A eq B	xs:date	xs:date	(missing)	xs:boolean?
A eq B	xs:time	xs:time	(missing)	xs:boolean?
A eq B	xs:dateTime	xs:dateTime	op:datetime-equal(A, B)	xs:boolean?
A eq B	xs:yearMonthDuration	xs:yearMonthDuration	op:yearMonthDuration-equal(A, B)	xs:boolean
A eq B	xs:dayTimeDuration	xs:dayTimeDuration	op:dayTimeDuration-equal(A, B)	xs:boolean
A eq B	Gregorian	Gregorian	(missing)	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:anyURI-equal(A, B)	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	xf:not(op:numeric-equal(A, B))	xs:boolean
A ne B	xs:boolean	xs:boolean	xf:not(op:boolean-equal(A, B))	xs:boolean
A ne B	xs:string	xs:string	xf:not(op:numeric-equal(xf:compare(A, B), 1))	xs:boolean
A ne B	xs:date	xs:date	(missing)	xs:boolean?
A ne B	xs:time	xs:time	(missing)	xs:boolean?
A ne B	xs:dateTime	xs:dateTime	xf:not3(op:datetime-equal(A, B))	xs:boolean?
A ne B	xs:yearMonthDuration	xs:yearMonthDuration	xf:not(op:yearMonthDuration-equal(A, B))	xs:boolean
A ne B	xs:dayTimeDuration	xs:dayTimeDuration	xf:not(op:dayTimeDuration-equal(A, B)	xs:boolean
A ne B	Gregorian	Gregorian	(missing)	xs:boolean
A ne B	xs:hexBinary	xs:hexBinary	xf:not(op:hex-binary-equal(A, B))	xs:boolean
A ne B	xs:base64Binary	xs:base64Binary	xf:not(op:base64-binary-equal(A, B))	xs:boolean
A ne B	xs:anyURI	xs:anyURI	xf:not(op:anyURI-equal(A, B))	xs:boolean
A ne B	xs:QName	xs:QName	xf:not(op:QName-equal(A, B))	xs:boolean

A ne B	xs:NOTATION	xs:NOTATION	xs: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(xf:compare(A, B), 0)	xs:boolean
A gt B	xs:date	xs:date	(missing)	xs:boolean?
A gt B	xs:time	xs:time	(missing)	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 lt B	numeric	numeric	op:numeric-less-than(A, B)	xs:boolean
A lt B	xs:boolean	xs:boolean	op:boolean-less-then(A, B)	xs:boolean
A lt B	xs:string	xs:string	op:numeric-less-than(xf:compare(A, B), 0)	xs:boolean
A lt B	xs:date	xs:date	(missing)	xs:boolean?
A lt B	xs:time	xs:time	(missing)	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 ge B	numeric	numeric	op:numeric-less-than(B, A)	xs:boolean
A ge B	xs:string	xs:string	op:numeric-greater-than(xf:compare(A, B), -1)	xs:boolean
A ge B	xs:date	xs:date	(missing)	xs:boolean?
A ge B	xs:time	xs:time	(missing)	xs:boolean?
A ge B	xs:dateTime	xs:dateTime	op:datetime-less-than(B, A)	xs:boolean?
A ge B	xs:yearMonthDuration	xs:yearMonthDuration	op:yearMonthDuration-less-than(B, A)	xs:boolean
A ge B	xs:dayTimeDuration	xs:dayTimeDuration	op:dayTimeDuration-less-than(B, A)	xs:boolean
A le B	numeric	numeric	op:numeric-greater-than(B, A)	xs:boolean
A le B	xs:string	xs:string	op:numeric-less-than(xf:compare(A, B), 1)	xs:boolean
A le B	xs:date	xs:date	(missing)	xs:boolean?
A le B	xs:time	xs:time	(missing)	xs:boolean?
A le B	xs:dateTime	xs:dateTime	op:datetime-greater-than(B, A)	xs:boolean?
A le B	xs:yearMonthDuration	xs:yearMonthDuration	op:yearMonthDuration-greater-than(B, A)	xs:boolean
A le B	xs:dayTimeDuration	xs:dayTimeDuration	op:dayTimeDuration-greater-than(B, A)	xs:boolean
A is B	node	node	op:node-equal(A, B)	xs:boolean
A isnot B	node	node	xf:not(op:node-equal(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 precedes B	node	node	op:node-precedes(A, B)	xs:boolean
A follows B	node	node	op:node-follows(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:decimal	xs:decimal	op:to(A, B)	xs:integer+
A , B	item*	item*	op:concatenate(A, B)	item*

Unary Operators

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

C References

C.1 Normative References

XML

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

XML Names

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

XML Schema

World Wide Web Consortium. *XML Schema, Parts 0, 1, and 2*. W3C Recommendation, 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/ , http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/ , and http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

XQuery 1.0 and XPath 2.0 Data Model

World Wide Web Consortium. *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft, 30 April 2002. See http://www.w3.org/TR/query-datamodel/.

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 Working Draft, 30 April 2002. See http://www.w3.org/TR/xquery-operators/

XQuery 1.0 Formal Semantics

World Wide Web Consortium. *XQuery 1.0 Formal Semantics*. W3C Working Draft, 26 March 2002. See http://www.w3.org/TR/query-semantics/. [**Ed. Note:** As of the date of this publication, the XQuery Formal Semantics has not incorporated recent language changes; it will be made consistent with this document in its next edition.]

C.2 Non-normative References

XML Query 1.0 Requirements

World Wide Web Consortium. *XML Query 1.0 Requirements*. W3C Working Draft, 15 Feb 2001. See http://www.w3.org/TR/xmlquery-req.

XML Query Use Cases

World Wide Web Consortium. *XML Query Use Cases*. W3C Working Draft, 30 April 2002. See http://www.w3.org/TR/xmlquery-use-cases.

XPath 2.0

World Wide Web Consortium. *XML Path Language (XPath) Version 2.0*. W3C Working Draft, April 30, 2002. See http://www.w3.org/TR/xpath20/

XQueryX 1.0

World Wide Web Consortium. *XQueryX, Version 1.0*. W3C Working Draft, 7 June 2001. See http://www.w3.org/TR/xqueryx [**Ed. Note:** As of the date of this publication, XQueryX has not incorporated recent language changes; it will be made consistent with this document in its next edition.]

XSLT 2.0

World Wide Web Consortium. *XSL Transformations (XSLT) 2.0.* W3C Working Draft. See http://www.w3.org/TR/xslt20

C.3 Background References

Lorel

Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. *The Lorel Query Language for Semistructured Data. International Journal on Digital Libraries*, 1(1):68-88, April 1997. See "http://www-db.stanford.edu/~widom/pubs.html

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.

SQL

International Organization for Standardization (ISO). *Information Technology-Database Language SQL*. Standard No. ISO/IEC 9075:1999. (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.)

XML-QL

Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. *A Query Language for XML*. See http://www.research.att.com/~mff/files/final.html

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

XQL

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

YATL

S. Cluet, S. Jacqmin, and J. Simeon. *The New YATL: Design and Specifications*. Technical Report, INRIA, 1999.

C.4 Informative Material

Character Model

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

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.

Unicode

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

XML Infoset

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

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

XSLT 1.0

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

D Glossary

Ed. Note: To be done.

E XPath 2.0 and XQuery 1.0 Issues (Non-Normative)

Values for **Status** has the following meaning:

resolved: a decision has been finalized and the document updated to reflect the decision.

decided: recommendations and decision(s) has been made by one or more of the following: a task-force, XPath WG, or XQuery WG.

draft: a proposal has been developed for possible future inclusion in a published document.

active: issue is actively being discussed.

unassigned: discussion of issue deferred.

subsumed: issue has been subsumed by another issue.

(parameters used: kwSort: cluster, kwFull: brief, kwDate: 00000000).

Num Cl	Cluster	Status	Locus	Description	Originator
293		active	xquery	Cdata and CharRef Semantics	Michael Rys
294		active	xquery	Should all elements and attributes have type annotations?	Philip Wadler
134	(de)reference-expr	decided	xquery	Dereference Operator and Links	Jonathan Robie
92	(in)equality-operators	active	xpath	Deep equality	Jonathan
96	(in)equality-operators	active	xpath	Normalized Equality	Mary Fernandez
91	(in)equality-operators	decided	xpath	9. String comparisons:	XQuery
28	3-value-logic	active	xpath	Do we need and3(), or3(), not3() built in?	XSL WG
215	3-value-logic	active	xpath	Should we have a 3-valued form of quantifiers?	Andrew Eisenberg

132		attribute accessors	draft	xfo	Attribute Name, Attribute Content	Don Chamberlin
113		axes	draft	xpath	What should the mechanism be for axis subsetting?	jmarsh@microsoft.com
177	TF	choice-context	active	xpath	Consistency of functions that take boolean formal argument	XPath-TF
256	D	collections	active	xpath	What is return type of colections?	XPath TF
257	D	collections	active	xpath	Does collection() always return same result?	XPath TF
169		conformance	unassigned	xquery	Missing conformance section	Jonathan
238		consistency	active	xpath	Consistency: tradeoff between interoperability and efficiency	Phil Wadler
239		consistency	active	xpath	Consistency: bracketing of nested expressions	Phil Wadler
240		consistency	active	xpath	Consistency: parenthesizing test expressions	Phil Wadler
241		consistency	active	xpath	Consistency: keywords	Phil Wadler
252		consistency	active	xquery	"sort by" rather than "sortby" for consistency?	Phil Wadler
286		constructor-expr	active	xquery	Element Construction vs Streaming	Michael Rys
288		constructor-expr	active	xquery	Element Constructor Attribute Order	Michael Rys
289		constructor-expr	active	xquery	Attribute Value Construction from Elements	Michael Rys
290		constructor-expr	active	xquery	Element Attribute Constructor Name Type	Michael Rys
291		constructor-expr	active	xquery	Element Construction anySimpleType Sequence Content	Michael Rys
292		constructor-expr	active	xquery	Element Construction Sequence vs Multi-expr	Michael Rys
147		constructor-expr	decided	xquery	Empty Attributes	XQuery
145		constructor-expr	unassigned	xquery	Copy and Reference Semantics	XQuery Editors
146		constructor-expr	unassigned	xquery		Jonathan
143		constructor-syntax	unassigned	algebra	XML Constructor	Jonathan Robie
69		context	active	xquery	Mapping Input Context	Jerome Simeon
230	D	context	active	xpath	Context document adequate for multiple docuemnts?	Michael Rys
258	D	documents	active	xpath	Identity of Document Nodes	Jonathan Robie
170	D	documents	unassigned	xquery	Document collections	XQuery WG
125		editorial	decided	xpath	Should we say something about the abstractness of XPath?	Editor
126		editorial	decided	xpath	Should we re-explain path expressions?	mhkay@iclway.co.uk

97		error	active	xpath	How should the error object be supported?	Editor
98		error	active	xpath	General discussion of errors	XML Query WG
99		error	unassigned	xquery	TRY/CATCH and error()	XQuery Editors
160		execution-model	active	xquery	Naive Implementation Strategy	Marton Nagy
226		existential expressions	active	xpath	Existential Expressions	Mike Kay
272		external-functions	active	xpath	External Functions	Michael Kay
273		external-objects	active	xpath	External Objects	Michael Kay
231		fallback	decided	xpath	data(SimpleValue) is error or no-op?	XSL WG
163		filter	unassigned	algebra	Typing of Filter	Jerome Simeon
216		focus	active	xpath	Description of focus is very procedural	Michael Rys
217		focus	active	xpath	Context document in focus	Michael Rys
32		for-expr	decided	xpath	Do we really require for at the XPath expression level?	K Karun
265		FTTF-xml:lang	active	xpath-fulltext	How do we determine the xml:lang for a node if it inherits xml:lang from a higher-level node?	FTTF
266		FTTF-xml:lang	active	xpath-fulltext	Do we support the sublanguage portion of xml:lang?	FTTF
105		function-app	active	xpath	Implicit current node for functions?	XQuery Editors
102		function-app	decided	xpath	Functions on Empty Sequences	XQuery
103		function-app	decided	xpath	Functions on Sequences	XQuery
107		function-app	decided	xpath	Path iteration	XQuery
108		function-app	decided	xpath	Aggregate functions on empty sequences.	XQuery
180		functions	active	xpath	Functions expecting complex-valued arguments	XPath-TF
124		functions	unassigned	xquery	External Functions	XQuery Editors
157		functions	unassigned	xquery	Function Libraries	XQuery Editors
223		functions external	active	xquery	We need a way to declare external functions	Michael Rys
3		grammar	active	xpath	What should be the precedence of a RangeExpr be?	Editor
254		grammar	active	xpath	Should the lexical details be in document?	XPath TF
168		groupby	unassigned	xquery	GROUPBY	XML Query
59	T	INSTANCEOF-expr	active	xpath	We need precise semantics for instanceof.	mhkay@iclway.co.uk
151		literal-XML	unassigned	xquery	Cutting and pasting XML into XQuery	XQuery Editors

166		miscellaneous	unassigned	xquery	Excluding Undesired Elements	Don Chamberlin
74		module-semantics	active	xquery	Module syntax	XQuery Editors
75		module-semantics	unassigned	xquery	Importing Modules	XQuery Editors
79		module-syntax	unassigned	xquery	Encoding	Jonathan Robie
228		namespace functions	active	xpath	Should we keep the default function namespace, and the xf: namespace?	Scott Boag
219		namespaces	active	xquery	Context: namespaces	Michael Rys
222		namespaces	active	xquery	Allow redefinition of namespace prefixes?	Michael Rys
247		namespaces	active	xpath	What does default namespace(s) affect?	XPath TF
202		namespaces	decided	xquery	In-scope namespaces and bindings	XPath Editors
86		node-equality	decided	xpath	Set operations based on value	XQuery Editors
85		node-equality	draft	xfo	Identity-based equality operator	Algebra Editors
186		node order	active	xpath	Ordering of result of union, intersect, and except operators	XPath Editors
184	T	node-types	active	xpath	Need ability to test for Comments, PIs	F&O TF
19		nulls-empty	decided	xpath	How should null be treated in the data model?	XSL WG
187		operators	active	xfo	Operations supported on date/time types	XPath Editors
189		operators	active	xpath	Supported combinations of types for comparison operators	XPath Editors
182		operators	draft	xpath	Mapping XPath Operators to F&O Functions	XPath TF
210		order sequences	active	xpath	Order of sequences	Phil Wadler
234		order sequences	active	xpath	Who defines sorting order of ()?	Phil Wadler
5		reserved-words	active	xpath	Reserved words	XQuery
8		reserved-words	decided	xpath	Case-Sensitivity in Keywords	XML Query
123		serialization	active	xquery	Linearization/Serialization	XQuery
244		serialization	active	xquery	CDATA sections and serialization	Michael Rys
36		SOME-expr	decided	xpath	Quantifiers with multiple bindings?	Don Chamberlin
194		sort	active	xquery	Support for stable and unstable sort?	XPath Editors
195		sort	active	xquery	Semantics of sorting heterogeneous sequences	XPath Editors
220		sort	active	xpath	Should there be a way to explicitly sort in document order?	Michael Rys

242		sort	active	xquery	Sortby on partially ordered values?	Michael Rys
243		sort	active	xpath	Provide an example of sorting "disappearing"	Michael Rys
251		sort	active	xquery	Sorting "input to loop", not the result	Phil Wadler
155		sort	unassigned	xquery	Sorting by Non-exposed Data	Michael Rys
109		syntax	active	xpath	Need text to disambiguate Lexical Structure?	mhkay@iclway.co.uk
212		syntax	active	xpath	Is "datatype" a suitable production name?	Phil Wadler
229		syntax	active	xpath	Do we need both << and precedes?	XQuery/XSL WGs
233		syntax	active	xquery	Simpler FLWR syntax?	Phil Wadler
235		syntax	active	xpath	Need parenthesis in conditional expression?	Phil Wadler
236		syntax	active	xpath	SimpleType preceded by a keyword?	Phil Wadler
237		syntax	active	xpath	Need parenthesis in type switch expression?	Phil Wadler
246		syntax	active	xquery	Nested XQuery comments allowed?	Michael Rys
267		syntax	active	xpath	Syntax problems with "validate"	XPath TF
287		syntax	active	xpath	Functional Status of Comma	Michael Rys
110		syntax	decided	xpath	Should we use is and isnot instead of == and !==.	mhkay@iclway.co.uk
144		syntax	draft	xquery	Escaping Quotes and Apostrophes	XML Query
112		syntax	unassigned	xquery	Leading Minus	XML Query
209		syntax attribute values	decided	xquery	Syntax for attribute values - more than one?	Phil Wadler
208		syntax curly brace	active	xquery	Multiple curly braces allowed?	Phil Wadler
245		syntax curly brace	active	xquery	Are {} in text evaluated?	Michael Rys
227		syntax dereference	active	xpath	Syntax for dereference?	Don Chambrelin
232		syntax operators date	active	xpath	Use "+" and "-" on dates and durations?	Phil Wadler
283		syntax-productions	active	xpath	Is the DocumentElement syntax production necessary?	Jonathan Robie
213		syntax quotes	active	xpath	How to get quotes etc in string literals?	Andrew Eisenberg
261		syntax quotes	active	xquery	Quoting rules in nested expressions in attribute value construction	Michael Rys
214		syntax strings in attributes	active	xquery	Strings in attributes	Andrew Eisenberg
183	T	text-nodes	active	xpath	Text nodes - lexical structure and typed form	XPath TF

192	T	type constructed element	decided	xquery	Type of a newly constructed element	XPath Editors
172	T	typed-value/data()	active	xpath	Some functions taking node sequences and implicitly map?	XPath TF
80		typed-value/data()	decided	xpath	Accessing Element Data	Mary Fernandez
81	T	typed-value/data()	decided	datamodel	What should the typed value of a complex type be?	jmarsh@microsoft.com
54		type-errors	decided	xquery	Queries with Invalid Content	XQuery Editors
174	T	type exception	decided	xpath	Support for UnknownSimpleType	XPath TF f2f
185		type exception	decided	xpath	Always explicit cast?	Phil Wadler
200	T	types	active	xpath	Semantics of "only"	XPath Editors
206	T	types	active	xpath	Typing support in XPath	Mike Kay
224	T	types	active	xquery	Why do we want to allow optional returns and DataType?	Michael Rys
196	T	types	decided	xpath	Concrete syntax for datatype declarations	XPath Editors
197	T	types	decided	xpath	Need "attribute of type"?	XPath Editors
198	T	types	decided	xpath	Syntax for named typing	XPath Editors
199	T	types	decided	xpath	Support for locally declared types?	XPath Editors
205	TF	types	decided	xquery	Default function parameter type	XPath Editors
211	T	types	decided	xpath	Treat and structural vs named typing	Phil Wadler
40	T	type-semantics	active	xquery	Correspondence of Types	Jerome Simeon
46	T	type-semantics	active	xquery	typeof() function	Jonathan
48	T	type-semantics	active	algebra	CASE not a subtype	XML Query
255	T	type-semantics	active	xpath	What are the operators on a derived type?	Don Chamberlin
260	T	type-semantics	active	xpath	Implicit data() on instanceof?	Michael Rys
262	T	type-semantics	active	xpath	Definition of data() on elements with known complex type	XPath TF
268	T	type-semantics	active	xpath	xsd:string and anySimpleType'd data behave the same?	XQuery WG
269	T	type-semantics	active	xpath	Can we get rid of the unknown keyword	XQuery WG
270	T	type-semantics	active	xpath	Typing, coercions, and conformance	XQuery WG
271	T	type-semantics	active	xpath	Type compatibility of heterogeneous union types	XQuery WG
274	T	type-semantics	active	xpath	Text-node-only implementation possible?	XSL WG

275	T	type-semantics	active	xpath	Should validation also check identity constraints?	Jonathan Robie
276	T	type-semantics	active	xpath	Does validation introduce new xsi:type attributes?	Jonathan Robie
277	T	type-semantics	active	xpath	Should validate introduce xsi:type?	Jonathan Robie
278	T	type-semantics	active	xpath	Does //@xsi:type match an xsi:type attribute?	Jonathan Robie
279	T	type-semantics	active	xpath	Should there be a lightweight cast?	Jonathan Robie
280	T	type-semantics	active	xpath	Do we need to distinguish atomic simple types and list simple types?	Jonathan Robie
281	T	type-semantics	active	xpath	Representing the type name of untyped character data	Jonathan Robie
282	T	type-semantics	active	xpath	xs:numeric type	Jonathan Robie
284	T	type-semantics	active	xpath	Static Named Typing	Jonathan Robie
285	T	type-semantics	active	xpath	Is the QName denoting type information optional or required?	Phil Wadler
41	T	type-semantics	decided	xquery	Static type-checking vs. Schema validation	Mary Fernandez
42	T	type-semantics	decided	xquery	Implementation of and conformance levels for static type checking	Don
43	T	type-semantics	decided	xquery	Defining Behavior for Well Formed, DTD, and Schema Documents	Don Chamberlin
44	T	type-semantics	decided	xquery	Support for schema-less and incompletely validated documents	Don Chamberlin/Mary Fernandez
45	T	type-semantics	decided	xquery	Names in Type Definitions	Don
47	T	type-semantics	decided	xquery	Subtype Substitutability	XQuery Editors
253	TF	type-semantics	decided	xpath	CAST as simple type of a node type	Michael Rys
259	T	type-semantics	decided	xpath	Arithmetic operation rules for unknown simpletype	Michael Rys
263	T	type-semantics	decided	xpath	Definition of data() on elements with unknown type, whose content may be simple or complex	XPath TF
51	TF	type-semantics	draft	xquery	Function Resolution	XQuery Editors
56	T	type-syntax	active	xquery	Human-Readable Syntax for Types	Algebra Editors
57	T	type-syntax	active	xquery	Inline XML Schema Declarations	Don Chamberlin
164		updates	unassigned	xquery	Updates	XQuery Editors
207		variables	active	xpath	Variable names: QNames or NCnames?	Ashok Malhotra

225		variables	active	xpath	Variable redefinition allowed?	Mike Kay
250		variables	active	xquery	Declaring Variables in Prolog	Jonathan Robie
264		variables	active	xpath	Shadowing of Variables	Michael Kay
191		whitespace	active	xquery	Whitespace handling in element constructors	XPath Editors
218		wildcards	active	xpath	What wildcards; namespaceprefix?	Michael Rys
193	T	xml non representable	active	xquery	Construction of non representable XML	XPath Editors
130		xquery-alignment	active	algebra	Algebra Mapping	XQuery Editors
152		xqueryx	active	xqueryx	XML-based Syntax	XML Query WG
153		xqueryx	active	xquery	Escape between syntaxes	Jerome Simeon

3. xpath-issue-rangeexpr-precidence: What should be the precedence of a RangeExpr be?

Locus: **xpath** Cluster: **grammar** Status: **active**
Originator: **Editor**

Description

What should be the precedence of a RangeExpr? The XQuery grammar has it in a bit different place than is listed here.

5. xpath-issue-reserved-words: Reserved words

Locus: **xpath** Cluster: **reserved-words** Status: **active**
Originator: **XQuery**

Description

XPath 1.0 has no reserved words. The current XQuery spec attempts to avoid reserved words but the result is that the XQuery grammar relies heavily on lexing tricks that make it difficult to document and difficult to extend. There is currently a substantial feeling in the XQuery group that the language should have some reserved words, which would be an incompatible change from XPath 1.0.

XSL WG Position: Exceptionally strong feeling that requiring element names that match a reserved word to be escaped is utterly unacceptable. If reserved words are required they must start with an escape character so they cannot conflict with element names. Attempt to maintain grammar and revisit this issue as necessary. We recognize that there is a problem but solutions are all distasteful.

Interactions and Input

Cf. 5. Reserved Words:

Subsumes: [link to subsumed issue: #xquery-xpath-issue-reserved-words] .

Cf. Keywords in XQuery

Subsumes: [link to subsumed issue: #xquery-reserved-keywords] .

8. xquery-case-sensitive-keywords: Case-Sensitivity in Keywords

Locus: **xpath** Cluster: **reserved-words** Status: **decided**
Originator: **XML Query**

Description

Should keywords in XQuery be case-sensitive?

Actual Resolution

Decision by: **xsl** on 2001-12-07 ([link to member only information])(joint meeting)

Decision by: **xquery** on 2001-12-07 ([link to member only information])(joint meeting)

Keywords are case-sensitive and it was also decided to use only lower-case keywords in the current XQuery / XPath WD. Thtopchange the text, the grammar, and the examples.

19. xpath-issue-null-value: How should null be treated in the data model?

Locus: **xpath** Cluster: **nulls-empty** Status: **decided**
Originator: **XSL WG**

Description

Sequences are defined to not be able to contain sequences. Yet, it is probably that we need to be able to represent null in sequences in order to maintain cardinality for table processing and the like.

Proposal: Nulls are a special value or an empty sequence: It is a special value when it is a member of a sequence. It is an empty sequence for type checking and iteration. This may require tagging empty sequences that were returned as null with a flag. The XSL WG has not reached resolution on this proposal.

Conflicts with : [link to subsumed issue: null-missing-data] .

Actual Resolution

Decision by: **xpath-tf** on 2001-12-18 ([link to member only information])

Decision by: **xquery** on 2001-12-19 ([link to member only information]) confiriming that no action is required by the XQuery WG

XSLT WG agrees that empty sequence should be used to represent missing data (this is the status quo).

28. xpath-issue-not-logic: Do we need and3(), or3(), not3() built in?

Locus: **xpath** Cluster: **3-value-logic** Status: **active**

Originator: **XSL WG**

Description

Should the 3-valued logic functions be part of the XPath function library? Note that they can be implemented as user functions (not as efficient as a "native" implementation). We may want to include the user definition as an example.

32. xpath-issue-is-for-required: Do we really require `for` at the XPath expression level?

Locus: **xpath** Cluster: **for-expr** Status: **decided**
Originator: **K Karun**

Description

Is `for` really a proper construct for a simple expression language? Will the definition of `for` in XPath cause problems for XQuery?

Interactions and Input

[link to member only information] Michael Kay:

Actual Resolution

Decision by: **xpath-tf** on 2001-11-27 ([link to member only information])

Decision by: **xsl** on 2001-11-29 ([link to member only information])

Adoption of FOR expression in XPath 2.0.

Scott Boag will provide 'tips & traps' text on potential semantic problems of mixing FOR, '/' and SORT.

Text is in Working Draft 2001-11-28 sec 2.9, paragraph 2.

36. xquery-quantifier-multiple-variables: Quantifiers with multiple bindings?

Locus: **xpath** Cluster: **SOME-expr** Status: **decided**
Originator: **Don Chamberlin**

Description

We have considered forms of quantified expressions that bind several variables at once, as in SOME $x IN expr1, $y IN expr2. Are such quantifiers desireable? If so, what are their semantics, and what use cases support them? Note that there is no additional expressive power over the current single-variable syntax, this is purely a question of convenience.

Actual Resolution

Decision by: **xpath-tf** on 2001-12-18 ([link to member only information])

Allow them; especially in view of consistency with eg "for" that already allows them. The semantics should be the same.

40. xquery-type-correspondence: Correspondence of Types

Issue Class: **T** Locus: **xquery** Cluster: **type-semantics** Status: **active**
Originator: **Jerome Simeon**

Description

Section 2.9, on functions, portrays XQuery as a statically typed language, but the mechanisms by which static typing is established are still being developed by the XML Query Algebra editorial team. A complete accounting for type requires that the XML Query Algebra conform completely to the XML Schema type system, and that many open issues be resolved.

The semantics of XQuery are defined in terms of the operators of the XML Query Algebra. The mapping of XQuery operators into Algebra operators is still being designed, and may result in some changes to XQuery and/or the Algebra. The type system of XQuery is the type system of XML Schema. Work is in progress to ensure that the type systems of XQuery, the XML Query Algebra, and XML Schema are completely aligned. The details of the operators supported on simple XML Schema datatypes will be defined by a joint XSLT/Schema/Query task force.

41. xquery-static-versus-validate: Static type-checking vs. Schema validation

Issue Class: **T** Locus: **xquery** Cluster: **type-semantics** Status: **decided**
Originator: **Mary Fernandez**

Description

Static type checking and schema validation are not equivalent, but we might want to do both in a query. For example, we might want to assert statically that an expression has a particular type and also validate dynamically the value of an expression w.r.t a particular schema.

The differences between static type checking and schema validation must be enumerated clearly (the XSFD people should help us with this).

This item is a duplicate of the Formal Semantics issue [link to member only information] .

Actual Resolution

Addressed by the "Named Typing" proposal.

42. xquery-type-conformance: Implementation of and conformance levels for static type checking

Issue Class: **T** Locus: **xquery** Cluster: **type-semantics** Status: **decided**
Originator: **Don**

Description

Static type checking may be difficult and/or expensive to implement. Some discussion of algorithmic issues of type checking are needed. In addition, we may want to define "conformance levels" for XQuery, in which some processors (or some processing modes) are more permissive about types. This would allow XQuery implementations that do not understand all of Schema, and it would allow customers some control over the cost/benefit tradeoff of type checking.

This item is a duplicate of the Formal Semantics issue [link to member only information] .

Actual Resolution

Addressed by the "Named Typing" proposal.

43. xquery-define-schema-variants: Defining Behavior for Well Formed, DTD, and Schema Documents

Issue Class: **T** Locus: **xquery** Cluster: **type-semantics** Status: **decided**
Originator: **Don Chamberlin**

Description

We should specify the behavior of XQuery for well formed XML, XML validated by a schema, and XML validated by a DTD.

Actual Resolution

Addressed by the "Named Typing" proposal.

44. xquery-schemaless: Support for schema-less and incompletely validated documents

Issue Class: **T** Locus: **xquery** Cluster: **type-semantics** Status: **decided**
Originator: **Don Chamberlin/Mary Fernandez**

Description

This is related to [link to resolved issue: #xquery-schema-import] . We do not specify what is the effect of type checking a query that is applied to a document without a DTD or Schema. In general, a schema-less document has type xs:AnyType and type checking can proceed under that assumption. A related issue is what is the effect of type checking a query that is applied to an incompletely validated document. As above, we can make *no* assumptions about the static type of an incompletely validated document and must assume its static type is xs:AnyType.

This item is a duplicate of the Formal Semantics issue [link to member only information] .

Actual Resolution

Addressed by the "Named Typing" proposal.

45. xquery-names-type-definition: Names in Type Definitions

Issue Class: **T** Locus: **xquery** Cluster: **type-semantics** Status: **decided**
Originator: **Don**

Description

Does the definition of a type include both element-names and element-contents (as in the Formal Semantics document), or only element-contents (as in XML Schema)?

Actual Resolution

Addressed by the "Named Typing" proposal.

46. xquery-typeof: typeof() function

Issue Class: **T** Locus: **xquery** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan**

Description

Do we need a function that returns the type name of its operand? If so, what should it return if the operand is an element with a given xsi:type - the element name? the name of the type denoted by xsi:type? Specification of this function requires more work on types in XQuery.

47. xquery-subtype-substitutability: Subtype Substitutability

Issue Class: **T** Locus: **xquery** Cluster: **type-semantics** Status: **decided**
Originator: **XQuery Editors**

Description

Should XQuery 1.0 support subtype substitutability for function parameters?

If subtype substitutability is not supported in XQuery Version 1, the motivation for TYPESWITCH is weakened and the decision to support TYPESWITCH should be revisited.

Actual Resolution

Addressed by the "Named Typing" proposal.

48. xquery-typeswitch-case-not-subtype: CASE not a subtype

Issue Class: **T** Locus: **algebra** Cluster: **type-semantics** Status: **active**
Originator: **XML Query**

Description

If the types in the CASE branches are not subtypes of the TYPESWITCH, is this an error, or are these branches simply never executed? If the latter, should we require a warning?

51. xquery-function-resolution: Function Resolution

Issue Class: **TF** Locus: **xquery** Cluster: **type-semantics** Status: **draft**
Originator: **XQuery Editors**

Description

More detailed rules need to be developed for function resolution. What kinds of function overloading are allowed? A promotion hierarchy of basic types needs to be specified. The issue of polymorphic functions with dynamic dispatch needs to be studied. Can overloaded functions be defined such that the parameter-type of one function is a subtype of the parameter-type of another function? If so, what are the constraints on the return-types of these functions? Is function selection based on the static type of the argument or on the dynamic type of the argument (dynamic dispatch, performed at execution time)? If XQuery supports dynamic dispatch, is it based on all the arguments of a function or on only one distinguished argument?

Observation: This is a very complex area of language design. If it proves too difficult to solve in the available time, it may be wise to take a simple approach such as avoiding dynamic dispatch in Version 1 of XML Query.

Proposed Resolution

The XML Query Formal Semantics does not support overloading or dynamic dispatch. We will attempt to simplify XML Query Level 1 by omitting these, unless it becomes clear that they are needed. We realize that this might happen.

54. xquery-invalid-content: Queries with Invalid Content

Locus: **xquery** Cluster: **type-errors** Status: **decided**
Originator: **XQuery Editors**

Description

Is it an error for a query to specify content that may not appear, according to the schema definition? Consider the following query:

```
invoice//nose
```

If the schema does not allow a nose to appear on an invoice, is this query an error, or will it simply return an empty list?

Actual Resolution

Addressed by the "Named Typing" proposal.

56. xquery-type-syntax: Human-Readable Syntax for Types

Issue Class: **T** Locus: **xquery** Cluster: **type-syntax** Status: **active**
Originator: **Algebra Editors**

Description

The Algebra has a syntax for declaring types. Up to now, XQuery uses XML Schema for declaring types. Is this sufficient? Some important questions:

1. Are type names sufficient, or does XQuery really need its own syntax for declaring types?
2. Would Normalized Universal Names (derived from MSL) be sufficient for type names?
3. How will type names be bound to definitions?

57. xquery-inline-xml-schema: Inline XML Schema Declarations

Issue Class: **T** Locus: **xquery** Cluster: **type-syntax** Status: **active**
Originator: **Don Chamberlin**

Description

Do we need to allow inline XML schema declarations in the prolog of a query? The following example shows one potential syntax for this. It extends the namespace declaration to allow literal XML Schema text to occur instead of a URI in a namespace declaration. The implementation would then assign an internal URI to the namespace.

```
                 NAMESPACE fid =
                 "http://www.example.com/fiddlefaddle.xsd"

  NAMESPACE loc =  [[
      <xsd:schema xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema">
      <xsd:simpleType name="myInteger">
      <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="10000"/>
      <xsd:maxInclusive value="99999"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:schema>
  ]]


  FUNCTION string-to-myInteger ($s STRING) RETURNS loc:myInteger
  {
   If the facets of loc:myInteger are not satisfied,
   this function raises an error.

    LET $t := round(number($s))
    RETURN TREAT AS loc:myInteger($t)
  }
  string-to-myInteger("1023")
```

59. xpath-issue-instanceof-symantics: We need precise semantics for instanceof.

Issue Class: **T** Locus: **xpath** Cluster: **INSTANCEOF-expr** Status: **active**
Assigned to: **xquery** Originator: **mhkay@iclway.co.uk**

Description

The semantics of the instanceof operator are not defined over all operands, e.g. sets of several nodes, sequences of several values, etc. What should the symantics be for these types? Should these semantics be defined in the F&O document?

69. xquery-mapping-input-context: Mapping Input Context

Locus: **xquery** Cluster: **context** Status: **active**
Originator: **Jerome Simeon**

Description

Do we need a way to specify the nodes in the input context? Many queries do not specifically state the input to which they will be applied. This allows the same query, for instance, to be applied to a number of databases. It may be helpful for the mapping to introduce a global variable, eg $input, to represent the input nodes for a query. This variable might even be useful in the syntax of XQuery itself.

74. xquery-module-syntax: Module syntax

Locus: **xquery** Cluster: **module-semantics** Status: **active**
Originator: **XQuery Editors**

Description

The definition and syntax of a query module are still under discussion in the working group. The specifications in this section are pending approval by the working group.

Future versions of the language may support other forms of query modules, such as update statements and view definitions.

75. xquery-import: Importing Modules

Locus: **xquery** Cluster: **module-semantics** Status: **unassigned**
Originator: **XQuery Editors**

Description

The means by which a query module gains access to the functions defined an an external function library remains to be defined.

Should xmlns only be respected for construction, Xquery expressions but not functions, or also functions?

79. xquery-encoding: Encoding

Locus: **xquery** Cluster: **module-syntax** Status: **unassigned**
Originator: **Jonathan Robie**

Description

Does XQuery need a way to specify the encoding of a query? For instance, should the prolog allow statements like the following?

```
ENCODING utf-16
```

80. xquery-data-function: Accessing Element Data

Locus: **xpath** Cluster: **typed-value/data()** Status: **decided**
Originator: **Mary Fernandez**

Description

There is no operator to access the typed constant content of an element. In the Algebra, the data() operator does this. Should XQuery do the same?

Interactions and Input

Cf. Some functions taking node sequences and implicitly map?

Actual Resolution

Decision by: **xpath-tf** on 2001-10-23 ([link to member only information])

Decision by: **xquery** on 2001-10-31 ([link to member only information])

Decision by: **xsl** on 2001-12-13 ([link to member only information]) It was noted that there are other issues on data() that may lead to further discussions.

Consensus is to define data() on only singleton node and empty sequence. This decision will be reflected in next WD.

81. xpath-issue-complex-type-value: What should the typed value of a complex type be?

Issue Class: **T** Locus: **datamodel** Cluster: **typed-value/data()** Status: **decided**
Originator: **jmarsh@microsoft.com**

Description

Section 1 - 4th paragraph says the typed-value of an element with complex type is the same as its string-value. This is not currently consistent with the data model. The data model doc says, in section 4.2, "If the element has a complex type, the typed-value is the empty sequence. For an element in a well-formed document with no associated schema, the element's typed-value is the empty sequence."

> **Ed. Note:** Question: I had a note from this, "insert from minutes", but I only see "Need to track "what does data()" return as an issue. Typed value? Something else?" as a response to this issue, and the formulation of issue text for the Data Model. Should this issue be removed from this document? -sb

Proposed Resolution

```
Element & attribute nodes
~~~~~~~~~~~~~~~~~~~~~~~~~~
Given $e is a single element or attribute node:

1. If $e is of unknown type and has simple content (i.e., $e does not contain 
complex
   content and $e does not have a known type), then
   data($e) evaluates to the simple content of the node and has 
UnknownSimpleType.

 Example:
    let $e := <fact about="what i saw"> I saw 8 cats.</ fact>
    return data($e)
 ==> "I saw 8 cats" : UnknownSimpletype

    data($e/@about)
 ==> "what i saw" : UnknownSimpleType

2. Else if $e is of a known simple type or has a simple content type, then
   data($e) evalutes to the simple value of the node and
   has the node's known simple type.

 Example:
   let $e := <weight units="{ 'lbs.' }">{ 42 }</weight>
   return data($e)
 ==> 42 : xs:integer

   data($e/@units)
 ==> "lbs." : xs:string


3. Else if $e is of an unknown (complex) type, then data($e) evaluates
   to string($e) (text node aggregation) and has UnknownSimpleType.

  let $e := <fact>The cat weighs <weight>12</weight> pounds</fact>
  return data($e)
 ==> "The cat weighs 12 pounds" : UnknownSimpleType
```

```
4. Else if $e is of a known complex type with complex or mixed content, then
   there are two options:

   4a. data($e) evaluates to an error
   4b. data($e) is defined as in (3) above.

 Example:
   Assume we have schema type:
   <element name="fact" mixed="true">
     <element name="weight" type="xs:integer"/>
   </element>

 and $e is validated w.r.t. the above type:

 let $e := validate as element fact (<fact>The cat weighs <weight>12</weight>
pounds</fact>)
 return data($e)

   4a. raises error
   4b. "The cat weighs 12 pounds" : UnknownSimpleType

Text nodes
~~~~~~~~~~~
5. Given $t is a single text node, then data($t) evaluates to string($t)
(content of
  text node) and has UnknownSimpleType.

 Example:

   let $e := <fact>The cat weighs 12 pounds</fact>,
       $t := $e/text()
   return data($t)

 ==> "The cat weighs 12 pounds" : UnknownSimpleType

 NB: Open issue as to whether data() should be defined on node sequences.
 NB: Current definition of data() applied to a text node is the empty sequence.

Comment, PI, Namespace nodes
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6. Given $c is a comment, processing instruction, or namespace node, then
data($c)
   evaluates to the empty sequence.

 Example:

   let $c := <!-- a comment -->
   return data($c)
 ==> () : empty
```

Actual Resolution

Decision by: **xpath-tf** on 2002-03-19 ([link to member only information])

Decision by: **xquery** on 2002-03-20 ([link to member only information])

Decision by: **xsl** on 2002-03-28 ([link to member only information])

Mary's proposal was adopted.

In particular, we chose option (4a) instead of (4b), because the current semantics applies data() implicitly in the basic conversion rules (e.g., when implementing equality and inequality operators), and therefore (4a) will raise an error if a user attempts to extract the atomic data from an element that has *known* complex type.

85. xquery-equality-identity: Identity-based equality operator

Locus: **xfo** Cluster: **node-equality** Status: **draft**
Originator: **Algebra Editors**

Description

Do we need an identity-based equality operator? Please justify your answer with sample queries. Note that XPath gets along without it.

Proposed Resolution

The '==' operator is already defined in the F&O document and used in XQuery use cases.

86. xquery-set-operators-on-values: Set operations based on value

Locus: **xpath** Cluster: **node-equality** Status: **decided**
Originator: **XQuery Editors**

Description

The definitions of UNION, INTERSECT, and EXCEPT for simple values are still under discussion. It is not clear that these operators should apply to simple values, because simple values do not have a concept of node identity. If these operators are defined for simple values, perhaps they should have a lower precedence than arithmetic operators.

Proposed Resolution

Text is in Working Draft 2001-11-28 sec 2.4.

Proposed Resolution

Decision by: **xsl** on 2002-01-22 ([link to member only information])

The XSL WG felt that these operators should operate on nodes only and not be overloaded to work on "values" as well. Functions should be used for the "analogous" functionality on values. Action to Ashok to write a proposal. It was noted that one might well want to to a "value" union operating on nodes (where the typed value of each is extracted).

Actual Resolution

Decision by: **xsl** on 2002-01-22 ([link to member only information])

Decision by: **xquery** on 2002-02-28 ([link to member only information])Xquery acceptance of XSL proposal

These operators operate on nodes only and are not overloaded to work on "values" as well. Functions are to be used for the "analogous" functionality on values.

91. xquery-xpath-string-compare: 9. String comparisons:

Locus: **xpath** Cluster: **(in)equality-operators** Status: **decided**
Originator: **XQuery**

Description

In XPath 1.0, string1 = string2 is evaluated by string comparison, but string1 > string2 is evaluated by attempting to convert both strings to floating-point numbers and comparing the numbers. This seems inconsistent. Both of these comparions should be carried out as string comparisons using the default collation.

In XPath 1.0, comparison of a string with a number is carried out by attempting to convert the string to a number. For example, 47 = "47" is True. This seems to conflict with strong typing. One might expect 47 = "47" to be a type error. Status: Open, pending discussion.

Proposed Resolution

Addressed by XPath 2.0 basic and standard-fallback conversions.

Actual Resolution

Decision by: **xquery** on 2002-03-27 ([link to member only information]) XML Query WG has adopted a solution to Issues 259 and 91 as per Proposal 1 in Michael Rys' proposal. See #unknown-simple-type-arithmetic

Decision by: **xsl** on 2002-03-28 ([link to member only information])

92. xquery-deep-equality: Deep equality

Locus: **xpath** Cluster: **(in)equality-operators** Status: **active**
Originator: **Jonathan**

Description

In XPath, <book><title> Mark Twain </title></book> and <book><author> Mark Twain </author></book> are treated as equal in comparisons. Is this acceptable for us? Do we need another notion of deep equality? If so, what are the compatibility issues with XPath?

96. xquery-normalized-equality: Normalized Equality

Locus: **xpath** Cluster: **(in)equality-operators** Status: **active**
Originator: **Mary Fernandez**

Description

When elements are compared, are comments and PIs considered in the comparison? How is whitespace handled? Do we need to allow more than one way to handle these in comparisons?

97. xpath-issue-error-object: How should the error object be supported?

Locus: **xpath** Cluster: **error** Status: **active**
Originator: **Editor**

Description

How should the error object be supported? How does the error object relate to exceptions?

98. xquery-general-errors: General discussion of errors

Locus: **xpath** Cluster: **error** Status: **active**
Originator: **XML Query WG**

Description

This document does not have a general discussion of errors, when they are raised, and how they are processed. This is needed.

99. xquery-try-catch-error: TRY/CATCH and error()

Locus: **xquery** Cluster: **error** Status: **unassigned**
Assigned to: **Dana** Originator: **XQuery Editors**

Description

We believe the following approach to error handling would be very useful - (1) introduce TRY <expression> CATCH <expression>, similar to try/catch in OO languages. Instead of having "throw" to throw objects, use error(<expression>), bind the result of the expression to the variable $err, and allow $err to be used in the CATCH clause.

Proposed Resolution

Dana Florescu has been assigned the task of writing a proposal for this.

102. null-issue-functions-on-empty: Functions on Empty Sequences

Locus: **xpath** Cluster: **function-app** Status: **decided**
Originator: **XQuery**

Description

XQuery-Null-Issue-7: Question: What should happen if a function expecting one element is invoked on an empty sequence?

Actual Resolution

It's an error.

Text is in Working Draft 2001-11-28 sec 2.1.2, paragraph 6, item 1.3.

103. null-functions-on-sequences-a: Functions on Sequences

Locus: **xpath** Cluster: **function-app** Status: **decided**
Originator: **XQuery**

Description

What should happen if a function expecting one element is invoked on a sequence of more than one element?

Actual Resolution

It's an error.

Text is in Working Draft 2001-11-28 sec 2.1.2, paragraph 6, item 1.1.

105. xquery-implicit-current-node: Implicit current node for functions?

Locus: **xpath** Cluster: **function-app** Status: **active**
Originator: **XQuery Editors**

Description

Many XPath functions and node-tests implicitly operate on the current node if no argument is specified. Should this apply to all functions? All unary functions? Only certain functions?

107. null-functions-on-sequences-b: Path iteration

Locus: **xpath** Cluster: **function-app** Status: **decided**
Originator: **XQuery**

Description

How should the current node be passed to a function in a path-step?

Actual Resolution

Using dot notation (".").

Text is in Working Draft 2001-11-28 sec 2.1.1.2, paragraph 4.

108. null-aggregates-on-empty: Aggregate functions on empty sequences.

Locus: **xpath** Cluster: **function-app** Status: **decided**
Originator: **XQuery**

Description

What should the the result of various aggregate functions when applied to the empty sequence?

Interactions and Input

[link to member only information] :

null-aggregates-on-empty: Aggregate functions on empty sequences [link to member only information] Status: The XPath TF recommends that we reopen this issue. We will do a straw poll with the WG on this recommendation. The Chair views the public pushback we have received on this topic as sufficient reason for re-opening this issue:
http://lists.w3.org/Archives/Public/www-xpath-comments/2002JanMar/0013.html
http://lists.w3.org/Archives/Public/www-xpath-comments/2001OctDec/0010.html (point 8)

Paul Cotton - Do we re-open this issue? Straw poll taken - Yes: 10 No: 0 Abstain: 8

The XPath TF should take this matter back under review.

Action A-87-06 - Paul Cotton to convey to XPath TF that issue is re-opened and the WG would like to receive some proposed changes.

Proposed Resolution

As in SQL, count(()) returns 0, and all other aggregate functions return the empty sequence. This rule is familiar to SQL users and minimizes the complexity of SQL-based XQuery implementations.

Actual Resolution

Decision by: **xpath-tf** on 2002-01-29 ([link to member only information])

Decision by: **xquery** on 2002-03-06 ([link to member only information])

Decision by: **xsl** on 2002-04-04 ([link to member only information])

Change semantics of sum() on () to 0.

109. xpath-issue-lexical-disambiguating-text: Need text to disambiguate Lexical Structure?

Locus: **xpath** Cluster: **syntax** Status: **active**
Originator: **mhkay@iclway.co.uk**

Description

There seems to be a lot of material missing here, for example the rules for disambiguating operator names from NCNames, the two uses of "*", etc. The statement "the longest possible token" means the longest sequence that would form a token in the token-space of the grammar, not the longest that would be valid in the current syntactic context.

110. xpath-issue-is-isnot-syntax: Should we use `is` and `isnot` instead of `==` and `!==`.

Locus: **xpath** Cluster: **syntax** Status: **decided**
Originator: **mhkay@iclway.co.uk**

Description

Should we use `is` and `isnot` instead of `==` and `!==`. Using keywords would be clearer, and less prone to error. On the other hand, introduction of more keywords could cause more controversy.

Actual Resolution

At Reston F2F, we agreed to infix operators 'value=', 'value<', 'node=', 'node<'. Next WD should reflect this decision.

Text is in Working Draft 2001-11-28 sec 2.6.2.

Actual Resolution

Decision by: **xsl** on 2002-02-28 ([link to member only information])(joint meeting)

Decision by: **xquery** on 2002-02-28 ([link to member only information])(joint meeting)

Decided to change to "is" and "isnot"

112. xquery-leading-minus: Leading Minus

Locus: **xquery** Cluster: **syntax** Status: **unassigned**
Originator: **XML Query**

Description

Should leading minus/plus be treated as unary operators or as part of a numeric literal? Or should both be supported?

113. xpath-issue-axis-subsetting: What should the mechanism be for axis subsetting?

Locus: **xpath** Cluster: **axes** Status: **draft**
Originator: **jmarsh@microsoft.com**

Description

Why is axis availability part of the run-time context? Myaxis::foo should be a syntax error. This seems like a really clunky mechanism for subsetting. The alternative, as Mike Kay presented, is to provide a set of booleans telling if the given axis is available.

Proposed Resolution

Proposed resolution: Axis subsetting should be done at the grammar level, similar to how XSLT pattern matching is specified.

123. null-serialization: Linearization/Serialization

Locus: **xquery** Cluster: **serialization** Status: **active**
Originator: **XQuery**

Description

When an empty element is linearized, should it get xsi:nil="True"? Option 8A: Yes. Option 8B: No. Option 8C: Only if it is declared to have required content and to be "nilable".

124. xquery-external-functions: External Functions

Locus: **xquery** Cluster: **functions** Status: **unassigned**
Originator: **XQuery Editors**

Description

An extensibility mechanism needs to be defined that permits XQuery to access a library of functions written in some other programming language such as Java.

Some sources of information: the definition of external functions in SQL, the implementation of external functions in Kweelt.

125. xpath-issue-xpath-abstract: Should we say something about the abstractness of XPath?

Locus: **xpath** Cluster: **editorial** Status: **decided**
Originator: **Editor**

Description

Should we say something about the fact that XPath is meant to be a abstract specification? I am concerned about people getting the impression that XPath can be used by itself. Maybe it can be, and maybe this is OK.

Interactions and Input

[link to member only information] Paul Cotton:

[link to member only information] Michael Rys:

Proposed Resolution

Decision by: **xquery** on 2002-01-16 ([link to member only information])

Action to Michael Rys to propose alternative text to resolve issue.

Proposed Resolution

Decision by: **xquery** on 2002-01-23 ([link to member only information])

XQuery WG approves wording in: [link to member only information] with a change so that "cannot" is changed to "should not" and Paul Cotton to convey changes to the XPath TF and XSL WG. Note: This text should be added to a new Conformance section in the XPath 2.0 specification.

Thus the proposed text is:

```
XPath is intended primarily as a component that can be used by other
specifications. Therefore, XPath relies on specifications that use XPath
(such as [XPointer] and [XSLT]) to specify criteria for conformance of
implementations of XPath and does not define any conformance criteria
for independent implementations of XPath. Specifications that set
conformance criteria for their use of XPath should not change the syntactic
or semantic definitions of XPath as given in this specification.
```

Proposed Resolution

Decision by: **xpath-tf** on 2002-03-05 ([link to member only information])

There is great concern in the XSL WG about the proposed last sentence. An alternative one was accepted by the XPath TF for approval by the WGs:

Specifications that set conformance criteria for their use of XPath must not change the syntactic or semantic definitions of XPath as given in this specification; except by subsetting and/or extensions in compatible ways.

Actual Resolution

Decision by: **xquery** on 2002-03-13 ([link to member only information])Accepting the XPath TF proposal

Decision by: **xsl** on 2002-04-04 ([link to member only information])

126. xpath-issue-re-explain-paths: Should we re-explain path expressions?

Locus: **xpath** Cluster: **editorial** Status: **decided**
Originator: **mhkay@iclway.co.uk**

Description

The division of path expressions into relative and absolute isn't reflected in the syntax. I think it's best to define the semantics as follows: / is an abbreviation for root(), /expr is an abbreviation for root()/expr, and // is an abbreviation for "/descendant-or-self::node()/". If the first step is an AxisStepExpr, add "./" in front of the expression. All unabbreviated paths now take the form OtherStepExpr ('/' StepExpr)* and we only need to define the semantics of this kind of path expression.

Actual Resolution

Decision by: **xquery** on 2002-02-06 ([link to member only information])

The current WD text reflects suggestions.

130. algebra-mapping: Algebra Mapping

Locus: **algebra** Cluster: **xquery-alignment** Status: **active**
Assigned to: **Jerome** Originator: **XQuery Editors**

Description

The algebra mapping is incomplete and out of date.

Proposed Resolution

Jerome has created a new version of the mapping, with help from Mary, Dana and Mugur.

132. xquery-attribute-name-content: Attribute Name, Attribute Content

Locus: **xfo** Cluster: **attribute accessors** Status: **draft**
Originator: **Don Chamberlin**

Description

We need functions to return the name of an attribute and the content of an attribute.

Proposed Resolution

My proposal is to create a separate section in the F&O document listing the accessors in the data model that are supported. The semantics would still be specified in the data model document. Similarly, we would add a section that listed the functions in the evaluation context that are supported. The semantics for these functions would be in the XPath document as decided this morning.

134. xquery-dereference-links: Dereference Operator and Links

Locus: **xquery** Cluster: **(de)reference-expr** Status: **decided**
Originator: **Jonathan Robie**

Description

Does the dereference operator work on links, such as XLink or HTML href? Should we also support KEY/KEYREF? In general, our handling of references needs a lot of work.

In general, how are the semantics of the dereference operator defined?

Proposed Resolution

Text is in Working Draft 2001-11-28 sec 2.3.

Actual Resolution

Decision by: **xsl** on 2002-01-22 ([link to member only information])

Decision by: **xquery** on 2002-02-28 ([link to member only information])

Acceptance of text in December 2001 published Working Draft.

143. xquery-literal-xml-constructor: XML Constructor

Locus: **algebra** Cluster: **constructor-syntax** Status: **unassigned**
Originator: **Jonathan Robie**

Description

Is there a need for a constructor that creates an instance of the XML Query Data Model from a string that contains XML text?

Proposed Resolution

Close the issue, pending convincing use cases. When do we need to be able to create a string of XML text, then convert it into an instance of the XML data model? Can't there always be an intervening parser or other tool to create the XML data model instance?

144. xquery-escaping-quotes-and-apostrophes: Escaping Quotes and Apostrophes

Locus: **xquery** Cluster: **syntax** Status: **draft**
Originator: **XML Query**

Description

In attribute constructors and string constructors, XQuery uses quotes or apostrophes as delimiters. How are these characters escaped when they occur within strings that are created by one of these constructors?

Proposed Resolution

I propose that we use double-delimiters within a string literal, e.g. 'I don''t', and unlike most of my syntactic ideas, this proposal seemed to recieve general support.

145. xquery-copy-reference: Copy and Reference Semantics

Locus: **xquery** Cluster: **constructor-expr** Status: **unassigned**
Originator: **XQuery Editors**

Description

Copy and reference semantics must be defined properly for updates to work. This must be coordinated with the algebra team.

146. xquery-document-construction:

Locus: **xquery** Cluster: **constructor-expr** Status: **unassigned**
Originator: **Jonathan**

Description

Can a query construct an entire XML document? What restrictions are there, eg on the internal subset, processing instructions or comments on the root level?

147. null-empty-attributes: Empty Attributes

Locus: **xquery** Cluster: **constructor-expr** Status: **decided**
Originator: **XQuery**

Description

What does <a b={expr}> mean if expr returns an empty sequence? Option 10A: Attribute is not created. Option 10B: Run-time error. Option 10C: The attribute is created, and its value is the empty sequence. Option 10D: The attribute is created and its value is the empty sequence if this is allowed by the attribute type; otherwise it's a run-time error. Option 10E: The attribute is created and its value is the empty sequence. However, an error may occur when the attribute is serialized. Option 10F: If the attribute-type is a list of scalar types that allows the empty list, an attribute is created with an empty sequence as its value; if the attribute is optional and scalar, then the attribute is not created. If the type system does not allow the attribute to be an empty list or absent, a type error is raised.

Actual Resolution

Decision by: **xquery** on 2001-12-07 ([link to member only information])

Adopt 10C.

151. xquery-cut-and-paste-xml: Cutting and pasting XML into XQuery

Locus: **xquery** Cluster: **literal-XML** Status: **unassigned**
Originator: **XQuery Editors**

Description

A variety of XML constructs can not be cut and paste into XQuery, including the internal subset, entities, notations, etc. Should we attempt to ameliorate this?

152. xquery-abql: XML-based Syntax

Locus: **xqueryx** Cluster: **xqueryx** Status: **active**
Originator: **XML Query WG**

Description

XQuery needs an XML representation that reflects the structure of an XQuery query. Drafts of such a representation have been prepared, but it is not yet ready for publication.

153. xquery-escape-to-abql: Escape between syntaxes

Locus: **xquery** Cluster: **xqueryx** Status: **active**
Originator: **Jerome Simeon**

Description

Is there a need to be able to escape to ABQL in XQueryX?

The text is confused. ABQL was the old name of XQueryX. The real issue is should be whether XQueryX and XQuery can escape to each other.

155. xquery-phantom-sortby: Sorting by Non-exposed Data

Locus: **xquery** Cluster: **sort** Status: **unassigned**
Originator: **Michael Rys**

Description

Should we make it easier to sort by data that is not exposed in the result? Although the current language allows this, it is difficult to define complex sort orders in which some items are not exposed in the result and others are computed in the expression that is sorted. Is there a more convenient syntax for this that would be useful in XQuery?

Proposed Resolution

```
Here is an example:

FOR $e IN //employee
WHERE ...
RETURN
  <newe>$e/name</newe>
SORTBY
  -- Now I would like to sort by salary but cannot.

Instead, the query would have to be written as

FOR $e IN (FOR $x IN //employee RETURN $x SORTBY employee/salary/data())
WHERE ...
```

```
RETURN
  <newe>$e/name</newe>

Which seems awkward.

The question is of course how we can integrate the SORTBY in an other
way.

For example, could we say

FOR $e IN //employee
WHERE ...
SORTBY $e/salary/data()
RETURN
   <newe>$e/name</newe>

?

Best regards
Michael
```

157. xquery-function-library: Function Libraries

Locus: **xquery** Cluster: **functions** Status: **unassigned**
Originator: **XQuery Editors**

Description

XQuery needs a mechanism to allow function definitions to be shared by multiple queries. The XQuery grammar allows function definitions to occur without a query expression.

We must provide a way for queries to access functions in libraries. For instance, we might add an IMPORT statement to XQuery, with the URI of the functions to be imported. It must be possible to specify either that (1) local definitions replace the imported definitions, or (2) imported definitions replace the local ones.

160. xquery-naive-implementation: Naive Implementation Strategy

Locus: **xquery** Cluster: **execution-model** Status: **active**
Originator: **Marton Nagy**

Description

Marton Nagy has suggested that it would be helpful to describe a naive implementation strategy for XQuery.

A naive XQuery implementation might parse the query, map it to Algebra syntax, and pass it to an Algebra implementation to request type checking from the algebra, returning an error if there were static type errors. A naive implementation might then request query execution from the algebra, get the results from the algebra and return it to the user.

Alternatively, the implementation might have its own algebra for execution, or it might generate statements in a specific implementation language such as XPath or SQL.We expect a wide variety of implementation approaches to be used in practice.

163. xquery-filter-typing: Typing of Filter

Locus: **algebra** Cluster: **filter** Status: **unassigned**
Assigned to: **Jerome Simeon** Originator: **Jerome Simeon**

Description

The current mapping of filter to the algebra does not preserve much useful type information. Can the mapping be improved? Is there another approach to filter that would yield better type information?

164. xquery-updates: Updates

Locus: **xquery** Cluster: **updates** Status: **unassigned**
Assigned to: **Jonathan** Originator: **XQuery Editors**

Description

We believe that a syntax for update would be extremely useful, allowing inserts, updates, and deletion. This might best be added as a non-normative appendix to the syntax proposal, since the algebra is not designed for defining this portion of the language.

166. xquery-exclude-undesireables: Excluding Undesired Elements

Locus: **xquery** Cluster: **miscellaneous** Status: **unassigned**
Originator: **Don Chamberlin**

Description

How do we exclude undesired elements from the results of joins?

Interactions and Input

[link to member only information] :

This need came out of a thread exploring data integration scenarios, starting with http://lists.w3.org/Archives/Member/w3c-archive/2000Dec/0132.html.

168. xquery-groupby: GROUPBY

Locus: **xquery** Cluster: **groupby** Status: **unassigned**
Originator: **XML Query**

Description

Does XQuery need an explicit GROUPBY expression? This would not add expressive power, but would be convenient, and may be easier to optimize.

169. xquery-conformance: Missing conformance section

Locus: **xquery** Cluster: **conformance** Status: **unassigned**
Originator: **Jonathan**

Description

The final XQuery recommendation must have a conformance section. The XML Query Working Group has not yet decided what should go into this section.

170. xquery-collections: Document collections

Issue Class: **D** Locus: **xquery** Cluster: **documents** Status: **unassigned**
Originator: **XQuery WG**

Description

How are collections of documents named and addressed in queries? How are they associated with files in directory structures, or resources in various systems? How does one address documents within collections?

172. functions-with-implicit-mapping: Some functions taking node sequences and implicitly map?

Issue Class: **T** Locus: **xpath** Cluster: **typed-value/data()** Status: **active**
Originator: **XPath TF**

Description

Should small set of functions on nodes, i.e., data(), name, uri-namespace(), be defined on node sequences and have an implicit mapping semantics?, e.g., data(a/b)

Interactions and Input

Cf. Accessing Element Data

174. fallback-if-untyped-marker: Support for UnknownSimpleType

Issue Class: **T** Locus: **xpath** Cluster: **type exception** Status: **decided**
Originator: **XPath TF f2f**

Description

Do we still need fallbacks if we have a marker for untyped data (="PCDATA")?

Interactions and Input

[link to member only information] Mary F. Fernandez:

Cf. Always explicit cast?

Proposed Resolution

Decision by: **xsl** on 2002-01-22 ([link to member only information]) Acceptance of text in December 2001 published Working Draft.

Text is in Working Draft 2001-11-28 sec 2.5.

Proposed Resolution

```
I discussed this issue with Mary on the phone today, and we have
developed the following modified version of her original proposal. The
key difference is this:

1. In the data model, the type name 'anySimpleType' is used everywhere
that 'unknownSimpleType' was used in her original proposal. This
avoids the problems discussed in:
   http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Mar/0173.html

2. In the SequenceType production, the keyword 'unknown' matches
values whose most specific type is 'anySimpleType'.

Jonathan and Mary

=================

This message proposes, in summary, that XQuery/XPath must have a type
for Text Nodes, denoted by the reserved word 'text', and a type for
Atomic Values with unknown simple type, denoted by the type name
'anySimpleType' in the Data Model. In the SequenceType production, the
keyword 'unknown' matches values whose most specific type is
'anySimpleType'.

Rationale:

XQuery/XPath supports expressions that yield two *distinct* types of
values:

   1. Text nodes, which have identity and contain character data
      Example:
        let $e := <fact>I saw 8 cats</fact>
        return $e/text()
        This expression yields a Text Node (that has identity!) with type
        'text'.

   2. Atomic values with unknown simple type, which no not have identity
      and contain character data

      Example 1:
        let $e := <fact about="what I saw">I saw 8 cats</fact>
        return data($e)

        This expression yields an Atomic Value (that does not have
        identity!)  with type 'anySimpleType' Example 2: data($e/@about)

        This expression also yields an Atomic Value with anySimpleType.

In a typed language, every value must have an associated type.
Neither value above currently has a type in XQuery/XPath.  In
addition, Text Nodes and Atomic Values with unknown simple type do not
denote the same set of values, therefore, they must have distinct
types.

Both these types may occur wherever a type expression is permitted,
e.g., in the SequenceType production. However, note that the type name
'anySimpleType' matches all simple types, since they are all derived
from 'anySimpleType', whereas the keyword 'unknown' matches only
values whose most specific type is 'anySimpleType'.

The formal semantics already constructs core expressions that require
both 'text' and 'unknown'.  for example, here's the rule that converts
a node to an optional atomic value of type xsd:integer:

   typeswitch (Expr) as $v
   case xs:integer? return $v
   case unknown return cast as xs:integer ($v)
   case node return
     typeswitch (data($v)) as $w
     ...

We use the type 'anySimpleType' for simple types because this type
explicity captures the notion that it is a simple type for which there
is no more specific known type. To simplify processing, we map all
PSVI representations of character data with unknown type onto this one
type.

Since our language needs a way to match only values whose most
specific type is 'anySimpleType', we add a keyword to the language,
rather than invent a new type for XML Schema. The functionality we
need is part of our language, not part of the XML Schema type system
that underlies our type system.

The necessity of associating a type to text nodes was first discussed
```

```
in Aug 2001:
   http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2001Aug/0209.html
```

Actual Resolution

Decision by: **xquery** on 2002-03-27 ([link to member only information]) XML Query WG has resolved issue 174 and Strawman Issue 17 by adopting Proposal #1 Jonathan Robie's proposal above.

Decision by: **xsl** on 2002-03-28 ([link to member only information])

177. functions-with-boolean-argument: Consistency of functions that take boolean formal argument

Issue Class: **TF** Locus: **xpath** Cluster: **choice-context** Status: **active**
Originator: **XPath-TF**

Description

The proposal by Michael Rys in [link to member only information: Proposal to close issue expr[nodes]] treats not()/not3() differently from all other functions that take a boolean formal argument.

Interactions and Input

Cf. Implicit conversion of node-sets to boolean for comparisons.

180. functions-complex-valued-arguments: Functions expecting complex-valued arguments

Locus: **xpath** Cluster: **functions** Status: **active**
Originator: **XPath-TF**

Description

Semantics of function application for functions that expect complex-valued arguments is not documented in current WDs.

182. operator-mapping: Mapping XPath Operators to F&O Functions

Locus: **xpath** Cluster: **operators** Status: **draft**
Originator: **XPath TF**

Description

The mapping from operators (such as "+", "*") to the operator functions needs to be defined. A second question is if the text should be in the F&O or XPath/XQuery document.

Proposed Resolution

This belongs in the XPath document. Ashok will send a draft to Don, who will incorporate this in XPath post December 2001 publication.

Proposed Resolution

See [link to member only information] .

Actual Resolution

Decision by: **xpath-tf** on 2002-01-22 ([link to member only information])Accepting Don's Proposal

Decision by: **xsl** on 2002-01-22 ([link to member only information])(whole group at XPath TF telcon)

Decision by: **xquery** on 2002-01-23 ([link to member only information])Accepting Don's Proposal

Decision by: **xquery** on 2002-02-06 ([link to member only information])

183. text-nodes-lexical-typed: Text nodes - lexical structure and typed form

Issue Class: **T** Locus: **xpath** Cluster: **text-nodes** Status: **active**
Originator: **XPath TF**

Description

Lexical structure and typed form of text nodes discussed in datamodel. No analogous discussion in the XPath document on that there are two ways to access information, as typed content or as nodes. This also needs to be discussed in conjunction with construction.

184. kind-tests: Need ability to test for Comments, PIs

Issue Class: **T** Locus: **xpath** Cluster: **node-types** Status: **active**
Assigned to: **xquery** Originator: **F&O TF**

Description

Need to extend languages to test for instance-of comments, pis, etc.

185. always-explicit-cast: Always explicit cast?

Locus: **xpath** Cluster: **type exception** Status: **decided**
Originator: **Phil Wadler**

Description

Should we do explicit casts rather than type conversion, like for example Python? This would result in e.g. forceing the user to tag a string everytime...

Interactions and Input

Cf. Support for UnknownSimpleType

Actual Resolution

Addressed by the "Named Typing" proposal.

186. order-union-intersect-except: Ordering of result of union, intersect, and except operators

Locus: **xpath** Cluster: **node order** Status: **active**
Originator: **XPath Editors**

Description

What should be the ordering of the results of the union, intersect, and except operators?

Issue is for material in sec 2.4. in Working Draft 2001-11-28.

Proposed Resolution

Union, intersect, and except on node sequences are defined to return their results in document order.

187. date-time-operators: Operations supported on date/time types

Locus: **xfo** Cluster: **operators** Status: **active**
Originator: **XPath Editors**

Description

What arithmetic operations should be supported on date/time types?

Issue is for material in sec 2.5. in Working Draft 2001-11-28.

Actual Resolution

Decision by: **xquery** on 2002-02-06 ([link to member only information])

Locus should be Functiond and Operators

189. operators-consistent-f_o: Supported combinations of types for comparison operators

Locus: **xpath** Cluster: **operators** Status: **active**
Originator: **XPath Editors**

Description

Make sure that the XPath/XQuery document is consistent with the F&O document with respect to the supported combinations of types for comparison operators.

Issue is for material in sec 2.6. in Working Draft 2001-11-28.

191. whitespace-in-element-constructors: Whitespace handling in element constructors

Locus: **xquery** Cluster: **whitespace** Status: **active**
Originator: **XPath Editors**

Description

How is whitespace handled in element constructors?

Issue is for material in sec 2.8. in Working Draft 2001-11-28.

Interactions and Input

[link to member only information] Michael Kay:

```
The XQuery WG requested information from XSL WG
as to how we currently deal with whitespace, in the hope that we can provide
an off-the-shelf solution to the problem.

In response to the action, here's a description of what XSLT does.

The stylesheet is an XML document. In constructing its infoset, all
processing instructions, comments, and whitespace-only text nodes are
discarded. (To be absolutely precise, PIs and comments are discarded; then
adjacent text nodes are merged; then whitespace-only text nodes are removed
from the tree). Whitespace text nodes are retained however, in two
circumstances: (a) if the whitespace text node is a child of an <xsl:text>
element, and (b) if an ancestor element specifies xml:space="preserve".

Certain elements in the stylesheet (for example, xsl:element) contain a
"content constructor". A content constructor is a sequence of XSLT
instructions, literal result elements, and literal text nodes. In evaluating
a content constructor, XSLT instructions do whatever the semantics of the
particular instruction say, while literal result elements and literal text
nodes are copied to the result tree.

The effect of this is that a whitespace-only text node in the stylesheet is
copied to the result tree only if either (a) it appears immediately inside
<xsl:text>, or (b) it is within the scope of an xml:space="preserve"
attribute.

Whitespace that is adjacent to non-white text in a literal text node is
copied to the result tree.
```

The effect of these rules is as follows:

```
=======================================
<a>  </a>
generates an empty element:
<a/>
=======================================
<a xml-space="preserve">  </a>
generates:
<a xml-space="preserve">  </a>
=======================================
<a><xsl:text>  </xsl:text></a>
generates:
<a>  </a>
=======================================
<a>
  <b/>
<a>
generates:
<a><b/></a>
=======================================
<a>Some text
  <b/>
</a>
generates:
<a>Some text
  <b/></a>
=======================================
```

There are other complications with whitespace. Whitespace in the result tree can come from the source document as well as from the stylesheet; XSLT provides control over whether whitespace-only text nodes in the source document are significant or not. Whitespace can also be generated in the output during serialization, if the xsl:output option indent="yes" is specified.

Also of course the XSLT rules apply in addition to the XML rules. XML for example normalizes line endings and normalizes whitespace (but not

character references) in attribute values. This happens outside XSLT's control. Whitespace character references such as   are treated differently from literal whitespace by the XML processor, but are treated identically to literal whitespace by the XSLT processor.

It's fair to say that these rules create a fair bit of confusion. It usually doesn't matter for generating HTML, because whitespace in HTML is rarely significant. For generating text files, it can be quite tricky. However, the rules are well-defined and a user who understands the rules can always get the required output.

What should XQuery do? I'd suggest mimicking these rules as closely as

```
possible, if only because users then only have to learn one set of confusing
rules rather than two. I can't think of any obvious improvements that would
make the system less confusing. Where the user wants to explicitly output
whitespace, of course, <a>{'  '}</a> provides a suitable alternative to
XSLT's <xsl:text> instruction.

This analogy would suggest that <a>  {'x'}  </a> should output <a>x</a>,
while <a>z {'x'} y</a> should output <a>z x y</a>: that is, the characters
between <a> and "{" are ignored if they consist entirely of whitespace, but
are all significant if any of them is non-whitespace. <a> </a> should output
<a/>, as should <a> </a>. This is only a suggestion, of course, the
decision is entirely for XQuery to make.

Mike Kay
```

192. type-of-constructed-element: Type of a newly constructed element

Issue Class: **T** Locus: **xquery** Cluster: **type constructed element** Status: **decided**
Originator: **XPath Editors**

Description

What is the type of a newly constructed element?

Issue is for material in sec 2.8. in Working Draft 2001-11-28.

Actual Resolution

Addressed by the "Named Typing" proposal.

193. non-valid-xml-construction: Construction of non representable XML

Issue Class: **T** Locus: **xquery** Cluster: **xml non representable** Status: **active**
Originator: **XPath Editors**

Description

An element constructor may contain adjacent simple values as the content of the element. This cannot be represented in XML. What should we do about this?

Issue is for material in sec 2.8. in Working Draft 2001-11-28.

194. sort-stable-unstable: Support for stable and unstable sort?

Locus: **xquery** Cluster: **sort** Status: **active**
Originator: **XPath Editors**

Description

Should the language support both stable and unstable sort? Using what syntax?

Issue is for material in sec 2.10. in Working Draft 2001-11-28.

195. sort-heterogenous-sequence: Semantics of sorting heterogeneous sequences

Locus: **xquery** Cluster: **sort** Status: **active**
Originator: **XPath Editors**

Description

Is it possible to sort a heterogeneous sequence? What syntax and semantics are used?

Issue is for material in sec 2.10. in Working Draft 2001-11-28.

196. syntaqx-datatype-declarations: Concrete syntax for datatype declarations

Issue Class: **T** Locus: **xpath** Cluster: **types** Status: **decided**
Assigned to: **xquery** Originator: **XPath Editors**

Description

What should be the concrete syntax for datatypes in function signatures and expressions? Should this syntax support node-or-value? Processing-instructions? Comments? Note that some functions in the F&O document (such as "exists") require "node-or-value" parameters.

Issue is for material in sec 2.13. in Working Draft 2001-11-28.

Actual Resolution

Addressed by the "Named Typing" proposal.

197. type-on-attributes: Need "attribute of type"?

Issue Class: **T** Locus: **xpath** Cluster: **types** Status: **decided**
Assigned to: **xquery** Originator: **XPath Editors**

Description

The current Datatype production supports declarations of the form "element of type <var>". Do we also need declarations of the form "attribute of type <var>" or "node of type <var>"?

Issue is for material in sec 2.13. in Working Draft 2001-11-28.

Actual Resolution

Addressed by the "Named Typing" proposal.

198. named-typing-support: Syntax for named typing

Issue Class: **T** Locus: **xpath** Cluster: **types** Status: **decided**
Assigned to: **xquery** Originator: **XPath Editors**

Description

Does the Datatype declaration need a way to specify both the QName of an element or attribute and its type, eg element person of type plumber?

Issue is for material in sec 2.13. in Working Draft 2001-11-28.

Actual Resolution

Addressed by the "Named Typing" proposal.

199. support-locally-declared-types: Support for locally declared types?

Issue Class: **T** Locus: **xpath** Cluster: **types** Status: **decided**
Assigned to: **xquery** Originator: **XPath Editors**

Description

Should XQuery support the import of locally declared types from schemas?

Issue is for material in sec 2.13. in Working Draft 2001-11-28.

Actual Resolution

Addressed by the "Named Typing" proposal.

200. semantics-of-only: Semantics of "only"

Issue Class: **T** Locus: **xpath** Cluster: **types** Status: **active**
Assigned to: **xquery** Originator: **XPath Editors**

Description

The keyword "only" remains in the syntax of "instance of", but the meaning of this keyword is not clear. Do we need this? If so, what are its semantics?

Issue is for material in sec 2.13. in Working Draft 2001-11-28.

Further input from Phil Wadler: Suggest we remove "only" from the text, and turn it into an issue. (As noted in the text, "only" makes sense only if we have named types, which is another issue.) If we do have "only", it should be used consistently in both "instance of" and in the "case" of a type switch. [!]

202. namespace-binding: In-scope namespaces and bindings

Locus: **xquery** Cluster: **namespaces** Status: **decided**
Originator: **XPath Editors**

Description

How do namespace declarations in the prolog and in namespace declaration attributes affect the in-scope namespaces of the context?

Issue is for material in sec 3. in Working Draft 2001-11-28.

Actual Resolution

Decision by: **xpath-tf** on 2002-01-15 ([link to member only information])

Decision by: **xquery** on 2002-01-16 ([link to member only information])

Decision by: **xsl** on 2002-01-22 ([link to member only information])

[Decided modulo suggested changes to text by Mike K] Action Item on Jonathan to incorporate these changes.

205. default-function-parameter-type: Default function parameter type

Issue Class: **TF** Locus: **xquery** Cluster: **types** Status: **decided**
Originator: **XPath Editors**

Description

If a function parameter does not specify a type, can both nodes and values be passed as arguments? If a function return does not specify a type, can both nodes and values be returned?

Issue is for material in sec 3. in Working Draft 2001-11-28.

Proposed Resolution

The default type of a function parameter when no type is specified in the function signature is changed from "any node" to "any type". The default return type of a function when no return type is specified is changed from "any sequence of nodes" to "any type".

Actual Resolution

Decision by: **xpath-tf** on 2002-01-15 ([link to member only information])

Decision by: **xquery** on 2002-01-16 ([link to member only information])

Decision by: **xsl** on 2002-01-22 ([link to member only information])

[Decided modulo suggested changes to text] Action Item on Jonathan to incorporate these changes. "any sequence" or "any type"

206. xpath-datatype-support: Typing support in XPath

Issue Class: **T** Locus: **xpath** Cluster: **types** Status: **active**
Originator: **Mike Kay**

Description

Which of these type productions (CAST, TREAT, ASSERT, TYPESWITCH...) belong in XPath? (ie common to XQuery and XPath)

207. variable-names: Variable names: QNames or NCnames?

Locus: **xpath** Cluster: **variables** Status: **active**
Originator: **Ashok Malhotra**

Description

Should variable names be QNames or NCNames? Note that XPath 1.0 uses QNames, and it may be helpful to disambiguate variable names from different modules.

208. multiple-curly-braces: Multiple curly braces allowed?

Locus: **xquery** Cluster: **syntax curly brace** Status: **active**
Assigned to: **xquery** Originator: **Phil Wadler**

Description

Can you have multiple matching curly braces inside an attribute value? How is it evaluated? What is the associated typing? Note that the BNF permits this.

209. syntax-attribute-values: Syntax for attribute values - more than one?

Locus: **xquery** Cluster: **syntax attribute values** Status: **decided**
Assigned to: **xquery** Originator: **Phil Wadler**

Description

Do we want to continue to support both of the following forms for computed attribute values? If not, is it possible to decide which to remove?

```
<pic size={}>
 {
 }
</pic>


<pic size="{}">
```

```
      {
      }
     </pic>
```

Actual Resolution

Decision by: **xquery** on 2002-01-16 ([link to member only information])

```
Jonathan: Computed attribute values - two ways to do this.  Propose to
elimiate no quoted representation.
Michael: a = {3} - it is clear that the type is integer...

PaulC:  Is there anyone who thinks the semantics are different.  Michael
is
not sure.

PaulC: Anyone think we should retain non quoted syntax:  no objections,
resolved.
```

210. order-sequences: Order of sequences

Locus: **xpath** Cluster: **order sequences** Status: **active**
Originator: **Phil Wadler**

Description

2.10, * Points 1 and 2 (Draft 2001-11-28). We could specify that lexical order is used when the ordering expression yields a sequence of values. This could be quite useful, e.g., if we represent Section 1.2.3 by (1,2,3) and Section 1.4 by (1,4), then lexical ordering does the right thing. This also implies that () should sort before everything else, as that is what happens with lexical order.

211. treat-choice: Treat and structural vs named typing

Issue Class: **T** Locus: **xpath** Cluster: **types** Status: **decided**
Assigned to: **xquery** Originator: **Phil Wadler**

Description

2.13.2 (Draft 2001-11-28) The behaviour of treat is profoundly affected by the choice between structural and named typing. If we use structural typing, then treat changes the type but not the value; if we use named typing, then treat must return a new value containing the appropriate type names. (Similarly for type switch.) The current wording is vague on this point. Should we be less vague and more clear about what the two different choices involve?

Actual Resolution

Addressed by the "Named Typing" proposal.

212. datatype-production-name: Is "datatype" a suitable production name?

Locus: **xpath** Cluster: **syntax** Status: **active**
Assigned to: **xquery** Originator: **Phil Wadler**

Description

Is Datatype the right name for production [54] (Draft 2001-11-28)? It may be confused with XML Schema Type 2.

213. syntax-special-characters: How to get quotes etc in string literals?

Locus: **xpath** Cluster: **syntax quotes** Status: **active**
Originator: **Andrew Eisenberg**

Description

How do you get strings such as single and double quote to be allowed in character strings?

```
      <name>Ben &amp; Jerry&apos;s</name>
```

eg should doubling a quote escape it?

It would seem that we need to allow these CharRef's in string literals as well.

Interactions and Input

Cf. Strings in attributes

214. strings-in-attributes: Strings in attributes

Locus: **xquery** Cluster: **syntax strings in attributes** Status: **active**
Assigned to: **xquery** Originator: **Andrew Eisenberg**

Description

Behavior of strings between quotes in attributes is different from that in string literals - should they be unified?

Interactions and Input

Cf. How to get quotes etc in string literals?

215. quantified-expressions-2-or-3-valued: Should we have a 3-valued form of quantifiers?

Locus: **xpath** Cluster: **3-value-logic** Status: **active**
Originator: **Andrew Eisenberg**

Description

Should we have a 3-valued form of quantifiers?

QuantifiedExpr is defined as being two-valued, while SQL's is 3-valued. SQL would evaluate 5 >ALL (6, null, 7) as UNKNOWN. XQuery would return FALSE where SQL returns UNKNOWN.

216. focus-description: Description of focus is very procedural

Locus: **xpath** Cluster: **focus** Status: **active**
Originator: **Michael Rys**

Description

The description of the focus is very procedural. Given that we define a declarative language this worries me quite a bit. Can this be written in a more declarative manner?

217. focus-context-document: Context document in focus

Locus: **xpath** Cluster: **focus** Status: **active**
Originator: **Michael Rys**

Description

The current definition of context documents may be difficult to implement in an efficient manner. Should this definition be simplified? Is this definition needed at all?

218. wildcards-for-what: What wildcards; namespaceprefix?

Locus: **xpath** Cluster: **wildcards** Status: **active**
Assigned to: **xquery** Originator: **Michael Rys**

Description

Do we need wildcards for only elements and attributes whose namespace URI is null? Are there other wildcards that should be supported?

219. namespaces-context: Context: namespaces

Locus: **xquery** Cluster: **namespaces** Status: **active**
Originator: **Michael Rys**

Description

XQuery must specify how namespaces are handled in the context.

Note that:

1. An unprefixed element name used as a nametest in a query has the namespaceURI associated with the default element namespace from the context.

Note: we may decide that this is the same as the default namespace of the in-scope namespaces, but this is not certain.

2. An unprefixed attribute has a null namespaceURI

220. document-order-explict-sort: Should there be a way to explicitly sort in document order?

Locus: **xpath** Cluster: **sort** Status: **active**
Originator: **Michael Rys**

Description

Should there be a way to explicitly sort in document order?

222. namespace-redefinition: Allow redefinition of namespace prefixes?

Locus: **xquery** Cluster: **namespaces** Status: **active**
Originator: **Michael Rys**

Description

Should we allow redefinition of namespace prefixes using NAMESPACE declarations in order to allow query system defaults that can be overridden by user? What are the interoperability issues of this?

223. external-function-definition: We need a way to declare external functions

Locus: **xquery** Cluster: **functions external** Status: **active**
Originator: **Michael Rys**

Description

Should we allow external functions to be defined using a syntax like this?

```
[71a] ExternalFnDef ::= {"define" S "external" S "function"}
          QName "(" ParamList? ")" ("returns" Datatype)?
```

224. function-return-datatype: Why do we want to allow optional returns and DataType?

Issue Class: **T** Locus: **xquery** Cluster: **types** Status: **active**
Originator: **Michael Rys**

Description

Would we prefer a syntax that requires explicit declaration of a general type when a function is to be loosely typed, rather than use the current syntax in which the type is omitted when the function is untyped.

225. variable-definition-error: Variable redefinition allowed?

Locus: **xpath** Cluster: **variables** Status: **active**
Originator: **Mike Kay**

Description

The binding of a variable in an expression always overrides any in-scope binding of a variable with the same name". In XSLT, it is an error to declare a (local) variable if another local variable with the same name is already in scope. This rule has proved useful in catching many user errors, particularly where users misunderstand the nature of XPath variables and think that a second declaration behaves like an assignment statement. I would prefer it to be an error to declare a variable if a variable of that name is already in scope. Also, the scope of variables has not been defined with very great precision, it's not explicitly clear what

```
for $i ........return for $i in // X, $j in $i/z return ....
```

means: which $i does this refer to?

Interactions and Input

Cf. Shadowing of Variables

226. testing-existential-expressions: Existential Expressions

Locus: **xpath** Cluster: **existential expressions** Status: **active**
Originator: **Mike Kay**

Description

Do we want to require implementations to test all pairs in order to determine if there is an incomparable value? Note that there are three choices: return true if any pair that satisfies the condition is encountered, return error on incomparable, or permit implementations to define what to do if both a true comparison and an incomparable pair are encountered. Should static and dynamic semantics handle this differently?

227. syntax-dereference: Syntax for dereference?

Locus: **xpath** Cluster: **syntax dereference** Status: **active**
Originator: **Don Chambrelin**

Description

What syntax should be used for dereference? In addition to the => syntax described in the document, here are two alternative approaches: (a) Treat dereferencing as an axis; (b) Simply use id(.) in a general step, with no special syntax.

228. default-namespace-functions: Should we keep the default function namespace, and the xf: namespace?

Locus: **xpath** Cluster: **namespace functions** Status: **active**
Originator: **Scott Boag**

Description

The question was raised of whether the Query WG and F&O TF have 1) a good rationale for putting built-in functions in a namespace, and 2) a consistent story about how this will relate to default namespace declarations and user-defined functions.

It would seem odd to have to have all user defined functions put in the FandO namespace if a namespace is not declared for it. If you do not do that, then user defined functions that don't require a prefix will not match the QName.

And, if there is not a good rational that provides user benifits, then it seems like we are going through a lot of additional complexity for little or no benefit.

229. more-than-one-precedes: Do we need both << and precedes?

Locus: **xpath** Cluster: **syntax** Status: **active**
Originator: **XQuery/XSL WGs**

Description

Do we need both << and precedes?

230. context-multiple-documents: Context document adequate for multiple docuemnts?

Issue Class: **D** Locus: **xpath** Cluster: **context** Status: **active**
Originator: **Michael Rys**

Description

Is "/" and the context document adequate for multiple documents?

Interactions and Input

Cf. Mapping Input Context

231. fallback-data-function: data(SimpleValue) is error or no-op?

Locus: **xpath** Cluster: **fallback** Status: **decided**
Originator: **XSL WG**

Description

What is the result of applying the data() function on a SimpleValue? Is it an error, is it a no-op, or is it a type exception with a fallback?

Actual Resolution

Decision by: **xpath-tf** on 2001-12-18 ([link to member only information])

Decision by: **xquery** on 2001-12-19 ([link to member only information])

data() applied to simple value is identity function (no-op). Rationale: does not require special 'fallback' behavior and if we resolve Issue 0172 so that data is defined on node sequences, then for consistency, it should be defined on heterogeneous sequences. Still waiting on general proposal for Issue 172.

232. operators-on-date: Use "+" and "-" on dates and durations?

Locus: **xpath** Cluster: **syntax operators date** Status: **active**
Originator: **Phil Wadler**

Description

Should we use the operators + - on dates and durations, when addition of dates is not associative or commutative?

233. syntax-flwr: Simpler FLWR syntax?

Locus: **xquery** Cluster: **syntax** Status: **active**
Originator: **Phil Wadler**

Description

Now that we have keywords, should we return to the simpler syntax?

```
  FLWRExpr ::= (ForClause | LetClause | WhereClause)* "return" Expr
```

234. order-sequences-who-defines: Who defines sorting order of ()?

Locus: **xpath** Cluster: **order sequences** Status: **active**
Originator: **Phil Wadler**

Description

2.10, + Point 5 (Draft 2001-11-28). If we don't adopt Plil W's proposal in Point 1, then there should be an Issue as to whether the behaviour should be specified by the user, or the vendor. One option is to let the user specify that () sorts high, low, or either, in the latter case the vendor decides.

Interactions and Input

Cf. Order of sequences

235. parenthesis-conditional-expression: Need parenthesis in conditional expression?

Locus: **xpath** Cluster: **syntax** Status: **active**
Originator: **Phil Wadler**

Description

Now that we have keywords, do we need parentheses in the conditional expression?

236. keyword-before-simpletype: SimpleType preceded by a keyword?

Locus: **xpath** Cluster: **syntax** Status: **active**
Assigned to: **xquery** Originator: **Phil Wadler**

Description

Issue as to whether SimpleType (was BuiltInType) might be preceded by a keyword. Now that we have "element of type" it seems natural to use "type" to preced SimpleType, possible generalizing to also allow a complex type.

237. parenthesis-type-switch-expression: Need parenthesis in type switch expression?

Locus: **xpath** Cluster: **syntax** Status: **active**
Originator: **Phil Wadler**

Description

Now that we have keywords, do we need parentheses in the type switch expression?

238. consistency-tradeoffs: Consistency: tradeoff between interoperability and efficiency

Locus: **xpath** Cluster: **consistency** Status: **active**
Originator: **Phil Wadler**

Description

Consistent tradeoff between interoperability and efficiency. There are a number of places where XQuery must choose between pinning down precise behaviour or offering flexibility to the implementor. These include whether order in "for" expressions is significant, whether sorting is stable, whether order is significant for duplicate elimination, whether order is significant when finding the union, merge, except of sequences of values, and the like. We should have a consistent policy for making these choices.

239. consistency-bracketing: Consistency: bracketing of nested expressions

Locus: **xpath** Cluster: **consistency** Status: **active**
Originator: **Phil Wadler**

Description

Consistent bracketing of nested expressions. Some nested expressions are surrounded by parentheses (treat, cast, assert), some nested expressions are surrounded by braces (element construction, function body), and some nested expressions are not bracketed (for expression, conditional expression). We should have a consistent policy for bracketing of nested expressions.

240. consistency-parenthesizing-test-expressions: Consistency: parenthesizing test expressions

Locus: **xpath** Cluster: **consistency** Status: **active**
Originator: **Phil Wadler**

Description

Consistent parenthesizing of test expressions. Some test expressions are surrounded by parentheses (conditional, type switch) and some are not (where). We should have a consistent policy for parenthesizing of test expressions.

241. consistency-keywords: Consistency: keywords

Locus: **xpath** Cluster: **consistency** Status: **active**
Originator: **Phil Wadler**

Description

Consistent keywords. Some keywords concatenate two words "sortby", "typeswitch", while others are multiple words "element" "of" "type", "cast" "as", "instance" "of". We should have a consistent policy for keywords.

242. sortby-partial-order: Sortby on partially ordered values?

Locus: **xquery** Cluster: **sort** Status: **active**
Originator: **Michael Rys**

Description

I was unable to find the semantics of what happens of sortby tries to order based on a base type that only has partial order (e.g., Duration). Can we explicitly disallow this?

243. sort-disappearing: Provide an example of sorting "disappearing"

Locus: **xpath** Cluster: **sort** Status: **active**
Originator: **Michael Rys**

Description

Provide an example of

```
   (employee sortby data(salary))/name
```

244. cdata-serialization: CDATA sections and serialization

Locus: **xquery** Cluster: **serialization** Status: **active**
Originator: **Michael Rys**

Description

What are the semantics of CDATA sections in XQuery? Are they preserved in the data model for serialization?

245. curly-braces-in-text: Are {} in text evaluated?

Locus: **xquery** Cluster: **syntax curly brace** Status: **active**
Originator: **Michael Rys**

Description

In attribute definitions, "`{a}`" will be interpreted as expression a returning the value for the attribute. Is this also the case for texts in general (inside an element construction)?

246. nested-comments: Nested XQuery comments allowed?

Locus: **xquery** Cluster: **syntax** Status: **active**
Originator: **Michael Rys**

Description

Nested XQuery comments allowed?

247. namespace-default-affecting: What does default namespace(s) affect?

Locus: **xpath** Cluster: **namespaces** Status: **active**
Originator: **XPath TF**

Description

What is effect of default namespace declarations on unprefixed QNames that may occur in element constructors, attribute constructors, and name tests (or anywhere else).

In XPath 1.0, in-scope namespace decls effect prefixed QNames, but default namespace decl does not effect unprefixed names in a name test. In XSLT 2.0, we introduced multiple default namespace decls :one for constructed elements, one for names in xpath expressions.

250. declaring-variables-in-prolog: Declaring Variables in Prolog

Locus: **xquery** Cluster: **variables** Status: **active**
Originator: **Jonathan Robie**

Description

Functions in a function library often need to access the same variables. For instance, a function library that manipulates a grammar may need to use the following variables in many functions:

```
$grammar := document("xpath-grammar.xml")/g:grammar
$target := "xquery"
```

One possible syntax for this is:

```
'define' ''variable' varname ':=' expr
```

The above production would occur only in the prolog.

251. sort-by: Sorting "input to loop", not the result

Locus: **xquery** Cluster: **sort** Status: **active**
Originator: **Phil Wadler**

Description

Consider this query:

```
   for $x in /books/book, $y in /reviews/review
   where $x/isbn = $y/isbn
   return <newbook>{ $x/title, $x/author, $y/reviewer }</newbook>
   sortby isbn
```

This is an error, since isbn doesn't appear in newbook. (The static type system would catch this error.) What you have to write is

```
   for $z in
     for $x in book, $y in review
     where $x/isbn = $y/isbn
     return <dummy>{ $x/title, $x/author, $y/reviewer, $x/isbn }</dummy>
     sortby isbn
   return <newbook>{ $z/title, $z/author, $z/reviewer }</newbook>
```

This is painful.

I think that XQuery should support this syntax, or something similar:

```
   for $x in book, $y in review
   where $x/isbn = $y/isbn
   sortby $x/isbn
   return <newbook>{ $x/title, $x/author, $y/reviewer }</newbook>
```

Our plate is full with more important matters just now, but I hope we could fix this before we finalize XQuery.

252. sortby-name: "sort by" rather than "sortby" for consistency?

Locus: **xquery** Cluster: **consistency** Status: **active**
Originator: **Phil Wadler**

Description

By the way, why is it "sortby"? Since we changed "instanceof" to "instance of", shouldn't we change "sortby" to "sort by"?

253. cast-simple-type-of-node: CAST as simple type of a node type

Issue Class: **TF** Locus: **xpath** Cluster: **type-semantics** Status: **decided**
Originator: **Michael Rys**

Description

Given the expression:

```
cast as xsd:decimal(//element_string [3]) div cast as
xsd:decimal("+1.25000000")
```

and a type of ELEMENT element_decimal [xsd:string] for the element_string nodes, our current cast rules disallow the cast of the element node to a value, since the above expression would be translated into:

```
data(cast as xsd:decimal(//element_string [3])) div cast as
xsd:decimal("+1.25000000")
```

instead of (the user expected):

```
cast as xsd:decimal(data(//element_string [3])) div cast as
xsd:decimal("+1.25000000")
```

Could we change the casting rules to allow the above?

We could either add a rule that pushes data() into cast expressions or have a special rule for casting to simple value types that implies applying data() in the casted expression (similar to arithmetic expressions).

Actual Resolution

Addressed by the "Named Typing" proposal.

254. lexical-details-in-doc: Should the lexical details be in document?

Locus: **xpath** Cluster: **grammar** Status: **active**
Originator: **XPath TF**

Description

Current grammar documents lexical states and maps directly to tool we use. An alternative is to use an alternative grammar and document lexical states with the tool separately.

255. operators-derived-type: What are the operators on a derived type?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Don Chamberlin**

Description

User defines a derived type, what operators are inherited by that type. SQL has a concept of distinct type. A distinct type does not inherit the operastors on the base type. XQuery in F&O does the opposite. All operations on the base type are inherited.

Interactions and Input

[link to member only information] Don Chamberlin:

[link to member only information] Don Chamberlin:

256. return-type-collection: What is return type of colections?

Issue Class: **D** Locus: **xpath** Cluster: **collections** Status: **active**
Originator: **XPath TF**

Description

MR: Why does collection() return item*. I want document*.

JR: You can specify what you want. You have to accept a URI and what you return is implementation dependent.

SCA: Strawpoll. 5 item*, 3 document*. No one cannot live with item*

Can an implementation provide a more specific return type than item* and still be a conforming implementation?

Some discussion. No consensus.

257. collection-always-return-same: Does collection() always return same result?

Issue Class: **D** Locus: **xpath** Cluster: **collections** Status: **active**
Originator: **XPath TF**

Description

Does collection() always return same result for the same URI? The same within the scope of a query/transformation? Are the nodes in the sequence identical?

258. document-nodes-identity: Identity of Document Nodes

Issue Class: **D** Locus: **xpath** Cluster: **documents** Status: **active**
Originator: **Jonathan Robie**

Description

Consider the following query:

```
        if (document("foo.com") == document("foo.com"))
                then <yep/>
                else <nope/>
```

I would like the following output:

```
        <yep/>
```

I think we can achieve this if we say that the URI of a resource is used as its identity. However, one resource can be identified by more than one URI. Suppose that "foo.com/here/there/hi.xml" and "file://c:/temp/limerick-tei.xml" refer to the same resource. What does the following return?

```
if (document("foo.com/here/there/hi.xml") ==
document("file://c:/temp/limerick-tei.xml"))
        then <yep/>
        else <nope/>
```

Should we simply use the URI of the parameter to establish identity and say that the two do not match? Should we make the result implementation-dependent?

259. unknown-simple-type-arithmetic: Arithmetic operation rules for unknown simpletype

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **decided**
Originator: **Michael Rys**

Description

Currently, the arithmetic operation rules in [1] say:

The operands of an arithmetic expression do not have required types. An arithmetic expression is evaluated by applying the following rules, in order, until an error is raised or a value is computed:

1. If any operand is a sequence of length greater than one, a type exception is raised.

2. If any operand is an empty sequence, the result is an empty sequence.

3. If any operand is a node, its typed-value accessor is applied. If the node has no typed-value accessor, or if the typed-value accessor returns a sequence of more than one value, an error is raised. If the typed-value accessor returns the empty sequence, the result of the expression is the empty sequence.

4. If any operand is an untyped simple value (such as character data in a schemaless document), it is cast to the type suggested by its lexical form. For example, the untyped value 12 is cast to the type xs:integer.

5. If the arithmetic expression has two numeric operands of different types, one of the operands is promoted to the type of the other operand, following the promotion rules in [XQuery 1.0 and XPath 2.0 Functions and Operators]. For example, a value of type xs:integer can be promoted to xs:decimal, and a value of type xs:decimal can be promoted to xs:double.

6. If the operand type(s) are valid for the given operator, the operator is applied to the operand(s), resulting in either a simple value or an error (for example, an error might result from dividing by zero.) The combinations of simple types that are accepted by the various arithmetic operators, and their respective result types, are listed in [XQuery 1.0 and XPath 2.0 Functions and Operators]. If the operand type(s) are not valid for the given operator, a type exception is raised.

Rule 4 cannot be efficiently implemented and basically would defer any algebrization of the operation to the runtime when the value is actually available to make this decision. Not even XPath 1.0 had such an impossible rule. (see also Mary's reply in [2])

Instead, I would like to align this with the rules of comparison operators in the following way:

The untyped simple value is cast to the type of the other operand.

If both operands are untyped, they are cast to double (in analogy to the XPath 1.0 number conversion).

E.g.,

```
unknownSimpleType + xsd:decimal will become cast as xsd:decimal
(unknownSimpleType) + xsd:decimal.
unknownSimpleType + unknownSimpleType will be cast as xsd:double
(unknownSimpleType) + cast as xsd:double
(unknownSimpleType).
```

Another proposal was provided by Michael Kay in [3] that however relays on the not yet introduced type numeric [4], thus I think we should follow the rule above (which is basically the same as we do for comparisons).

[1] http://www.w3.org/TR/xquery/#id-arithmetic

[2] [link to member only information]

[3] [link to member only information]

[4] [link to member only information]

Proposed Resolution

I have been tasked with rewording the sections 2.5 and 2.6.1 in [1] with two proposed solutions for implicit treatment of unknown simple types (UST) with arithmetic and comparison operators based on my earlier proposal in [2] and a subsequent simplification discussed at today's XPath TF call. The rewording should also refer to the basic conversion rules that are defined in [1] Section 2.1 as:

The basic conversion rules are as follows:

If the required type is a simple type or an optional occurrence of a simple type:

If the given value is a sequence of more than one item, a type exception is invoked. If the given value is a single node, its typed value is extracted by calling its typed value accessor. If the node has no typed value accessor or if its typed value is a sequence containing more than one item, a type exception is invoked.

If the (given or extracted) value has an unknown simple type (as in the case of character data in a schemaless document), an attempt is made to cast it to the required simple type. If the cast fails, a type exception is invoked.

If the (given or extracted) value does not conform to the required type, a type exception is raised. This includes the case in which the (given or extracted) value is an empty sequence and the required type is not an optional occurrence.

If the required type is a sequence of a simple type:

If the given value is a sequence containing one or more nodes, each such node is replaced by its typed value, resulting in a sequence of simple values. Each of these simple values then is converted to the required type by applying the rules described above for converting values to a required simple type.

If the required type is a sequence of nodes:

If the given value contains any item that is not a node, a type exception is invoked.

In addition, I assume that numeric and numeric types are well-defined
terms referring to the set of types double, float, decimal and all their
subtypes (derived by restriction).

****
Proposal I: Rewrite to treat UST as double if operating with numeric
types
****
2.5 Arithmetic Expressions
<leave as is until "For example, a-b will be interpreted as a single
token."/>

An arithmetic expression is evaluated by applying the following rules
that are based on the basic conversion rules, in order, until an error
is raised or a value is computed:

The required types of the operands of an arithmetic expression do not
have required types.

1. If any operand is a sequence of length greater than one, a type
exception is raised (applying the basic conversion rule to each
argument).

2. If any operand is an empty sequence, the result is an empty
sequence(applying the basic conversion rule to each argument).

3. If any operand is a node, its typed-value accessor is applied. If the
node has no typed-value accessor, or if the typed-value accessor returns
a sequence of more than one value, an error is raised. If the
typed-value accessor returns the empty sequence, the result of the
expression is the empty sequence. (applying the basic conversion rule to
each argument).

4. If any or both operands are an untyped simple value (such as
character data in a schemaless document), the basic conversion rule is
applied and the required type is xs:double. Thus an untyped simple value
is cast to type xs:double.

5. If the arithmetic expression has two numeric operands of different
types, one of the operands is promoted to the type of the other operand,
following the promotion rules in [XQuery 1.0 and XPath 2.0 Functions and
Operators]. For example, a value of type xs:integer can be promoted to
xs:decimal, and a value of type xs:decimal can be promoted to xs:double.
This is an extension to the basic conversion rule.

6. If the operand type(s) are valid for the given operator, the operator
is applied to the operand(s), resulting in either a simple value or an
error (for example, an error might result from dividing by zero.) The
combinations of simple types that are accepted by the various arithmetic
operators, and their respective result types, are listed in [XQuery 1.0

and XPath 2.0 Functions and Operators]. If the operand type(s) are not valid for the given operator, a type exception is raised.

Ed. Note: A future version of this document will provide a mapping from the arithmetic operators to the functions that implement these operators for various datatypes. See issue 182.

Here are some examples of arithmetic expressions:

In general, arithmetic operations on numeric values result in numeric values:

<rest stays the same/>

2.6.1 Value Comparisons

Value comparisons are intended for comparing single values. The result of a value comparison is defined by applying the following rules, based on the basic conversion rules, in order:

1. If either operand is an empty sequence, the result is an empty sequence.

2. If either operand is a sequence containing more than one item, an error is raised.

3. If either operand is a node, its typed value is extracted. If the typed value is an empty sequence, the result of the comparison is an empty sequence. If the typed value is a sequence containing more than one item, an error is raised.

4. If one of the operands is an untyped simple value (such as character data in a schemaless document) and the other operand is a typed simple value, the required type for casting the untyped simple value is determined by the type of the other operands as follows:
   - if the type is of the other operand is numeric the required type becomes xs:double.
   - if the type of the other operand is unknown, then the required type for both becomes xs:string.
   - otherwise the type of the other operand becomes the required type.

5. 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. [XQuery 1.0 and XPath 2.0 Functions and Operators] describes which combinations of simple types are comparable, and how comparisons are performed on values of various types. If the value of the first operand is not comparable with the value of the second operand, a type exception is invoked.

Ed. Note: A future version of this document will provide a mapping from the comparison operators to the functions that implement these operators for various datatypes. See issue 182.

Here is an example of a value comparison:

The following comparison is true only if $book1 has a single author subelement and its value is "Kennedy":

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

```
********************************************
Proposal II: Rewrite to treat UST as the other numeric type
********************************************
```

2.5 Arithmetic Expressions
Change Proposal I Rule 4 to:

```
4. If of the operands is an untyped simple value (such as character data
in a schemaless document), the basic conversion rule is applied and the
required type is set based on the type of the other operand:
  - the required type for both operands becomes xs:double if the type of
the other operand is unknown
  - otherwise the required type becomes the type of the other operand.
Thus an untyped simple value added to a float is cast to type xs:float.
```

2.6.1 Value Comparisons
Change Proposal I Rule 4 to:

```
4. If one of the operands is an untyped simple value (such as character
data in a schemaless document) and the other operand is a typed simple
value, the required type for casting the untyped simple value is
determined by the type of the other operands as follows:
   - if the type of the other operand is unknown, then the required type
for both becomes xs:string.
   - otherwise the type of the other operand becomes the required type.
```

```
********
Analysis
********
```

The advantage of proposal I is that it is closer to the known XPath behaviour (although as Peter pointed out not the same), it is simpler to explain and implement and allows expressions where the UST value is a double and an operation combines it with a type based on decimal that would lead to runtime errors in case II. Based on extensive discussions with my constituency, users and implementers alike, this seemed to be the preferred solution.

The disadvantage is a slight loss of type precision on the result that can be regained by explicitly casting the result or operands to the desired type.

Best regards
Michael

```
[1] http://www.w3.org/TR/xquery/
[2] http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Mar/0037.html
```

Actual Resolution

Decision by: **xquery** on 2002-03-27 ([link to member only information]) XML Query WG has adopted a solution to Issues 259 and 91 as per Proposal 1 in Michael Rys' proposal.

Decision by: **xsl** on 2002-03-28 ([link to member only information])

260. instanceof-implicit-data: Implicit data() on instanceof?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Michael Rys**

Description

Does expr instance of type define an expected type with type so that data() is implied explicitly on:

```
let $e := <e xsi:type=?xsd:integer?/>
return
$e instanceof xsd:integer
```

is that

```
let $e := <e xsi:type=?xsd:integer?/>
return
  data($e) instanceof xsd:integer
```

and thus true or false?

This issue is somewhat related to CAST (Issue 253), but since we perform a type test here, I would argue that we should not.

Interactions and Input

Cf. CAST as simple type of a node type

261. attribute-constructor-quoting-rules: Quoting rules in nested expressions in attribute value construction

Locus: **xquery** Cluster: **syntax quotes** Status: **active**
Originator: **Michael Rys**

Description

We currently say that in a constant string in XQuery, I can do any of the following three to escape quotes (please correct if I am wrong):

A. XQuery rule (works both in XQuery only and XQuery in XML embedding): quotes and double quotes are doubled if they are enclosed inside their own kind.

E.g.,

```
Lexical                Value in datamodel
'"'                    "
''''                   '
""""                   "
'""'                   ""
```

B. XML rules (work only in XQuery in XML embedding since XML parser will work:

B1. Use ' when string delimiter is " and viceversa. This of course does not preserve data fidelity, so if the difference between them is important, you will need to use B2.

B2. Entitize

I assume that I cannot do just B, but that I have to use B to get A. E.g., <a foo="&quot;"/> in an XML document is not a valid XQuery and I have to write <a foo="&quot;&quot;"/>

The problem that I have is that I do not know what the rules are for nesting quotes inside XQuery expressions:

What are the rules for:

```
<foo bar="{fnc1(<baz goo="{fnc2("string arg")}"/>)}"/>
```

Do I have to double and then triple? Does {} mean that the outside "" become inconsequential?

262. data-on-elements-with-known-complex-type: Definition of data() on elements with known complex type

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **XPath TF**

Description

An alternative definition of data() on elements with known complex type:

If $e is an element of an known (complex) type, then data($e) evaluates to the concatenation of $e's text nodes and has UnknownSimpleType.

This definition is appropriate when explicit casts are necessary to convert UnknownSimpleType to other atomic types, i.e., when the basic conversion rules *DO NOT* implicitly convert UnknownSimpleType to other atomic types.

The benefit of this alternative is that a user can always safely apply data() to any node and is guaranteed never to get an error.

263. data-on-elemets-with-unknown-type: Definition of data() on elements with unknown type, whose content may be simple or complex

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **decided**
Originator: **XPath TF**

Description

The the data() function is only defined on element and attributes with a simple type. The definition of data() on other kinds of nodes or on element and attributes with unknown type is still an open issue.

The corresponding issue in the Formal Semantics document is Issue-108.

Actual Resolution

Decision by: **xpath-tf** on 2002-04-11 ([link to member only information])

xf:data(): decided that it should return the error object when applied to document, PI, comment and namespace nodes.

This change should be reflected in the F&O, XPath and Data Model document.

264. variables-shadowing: Shadowing of Variables

Locus: **xpath** Cluster: **variables** Status: **active**
Originator: **Michael Kay**

Description

XSLT does not allow a local variable to be declared if there is already a local variable in scope with the same name. For example, the following is disallowed:

```
<xsl:variable name="i" select="0"/>
<xsl:for-each select="item">
  <xsl:variable name="i" select="$i + 1"/>
  <a><xsl:value-of select="$i"/></a>
</xsl:for-each>
```

This restriction has proved with experience to be a good thing, because users writing a construct such as the one above have almost invariably misunderstood the way variables work in a declarative language; if the construct were allowed, it would not give them the results they expect.

The equivalent XQuery expression is currently allowed:

```
let $i := 0
for $x in item
   let $i := $i + 1
   return <a>{$i}</a>
```

XPath 2.0 has no "let" expression, but it can include variable declarations, and these are allowed to shadow each other.

For XSLT users, it seems inconsistent that there should be different rules in XSLT and in XPath. The XSLT 1.0 rule, which makes variable shadowing an error, has proved useful in practice and we recommend that it should be adopted also for XPath 2.0 and XQuery 1.0.

Interactions and Input

Cf. Variable redefinition allowed?

265. lang-inheritance: How do we determine the xml:lang for a node if it inherits xml:lang from a higher-level node?

Locus: **xpath-fulltext** Cluster: **FTTF-xml:lang** Status: **active**
Originator: **FTTF**

Description

How do we determine the xml:lang for a node if it inherits xml:lang from a higher-level node?

266. lang-sublanguage-support: Do we support the sublanguage portion of xml:lang?

Locus: **xpath-fulltext** Cluster: **FTTF-xml:lang** Status: **active**
Originator: **FTTF**

Description

Do we support the sublanguage portion of xml:lang? If so, how?

267. validate-syntax-problem: Syntax problems with "validate"

Locus: **xpath** Cluster: **syntax** Status: **active**
Originator: **XPath TF**

Description

If "validate" is in XPath the "{}" in the syntax conflicts with the current use in XPath. It needs to change to use "()" instead.

268. string-anySImpleTyped-data-behaviour: xsd:string and anySimpleType'd data behave the same?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **XQuery WG**

Description

Should we simplify the relationship between xsd:string and the anySimpleType'd data to have them both behave the same operationally by providing implicit coercions and being type compatible?

Alternate, in some peoples view mutually exclusive, issue:

Should we simplify the relationship between xsd:string and the anySimpleType'd data to have them both behave the same operationally by disallowing implicit coercions on anySimpleType and being type compatible?

269. unknown-keyword-needed: Can we get rid of the unknown keyword

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **XQuery WG**

Description

Can we get rid of the unknown keyword?

Interactions and Input

[link to member only information] Michael Rys:

270. typing-coercions-conformance: Typing, coercions, and conformance

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **XQuery WG**

Description

One difficulty in designing XQuery is that it must meet the needs of two quite different constituencies:

Convenient: Quickly write queries to explore a large body of data. Works best with implicit coercions and dynamic typing.

Reliable: Solidly engineer queries for web services. Works best with explicit coercions and static typing.

We've already agreed to have two conformance levels, one with dynamic typing and one with static typing. Perhaps the dynamic typing level should have implicit coercions and the explicit typing level should have explicit coercions.

271. heterogenous-union-types-compatibility: Type compatibility of heterogeneous union types

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **XQuery WG**

Description

Currently the formal semantics maps arithmetic and comparison operations that have union types as (at least of one of) their argument types to a runtime type switch. This has serious unwanted consequences for "algebraization" of queries and optimization and leads to some surprising static typing results.

Should we make non-promotable union types on such operations a type error instead?

272. external-functions: External Functions

Locus: **xpath** Cluster: **external-functions** Status: **active**
Originator: **Michael Kay**

Description

The ability to call external functions is an established feature of XSLT 1.0, and is retained in XSLT 2.0. The facility is widely used, and some significant libraries of external functions have been developed by third parties. We made an attempt to standardize language bindings for external functions in the XSLT 1.1 working draft, but this proved highly controversial and has been dropped. The facility remains, however, even though the binding mechanisms remain implementation-defined.

The XPath 2.0 specification continues to tolerate external functions, though it doesn't really acknowledge their existence properly. All we say is that the repertoire of functions that can be called is part of the context.

The issue is: should the function function-available() function be transferred from XSLT to XPath?

This function tests whether a named function is available, and returns true if it is, and false if it isn't. A typical call is:

```
if  (function-available('my:debug')) then my:debug('Hi!') else ()
```

This has two implications:

(a) a call on my:debug must not be a static error if the function is not available

(b) the names of functions (and the in-scope namespaces needed to resolve their QNames) must be available at run-time.

Logically the function-available() function has no dependencies on XSLT so it should be transferred to XPath.

273. external-objects: External Objects

Locus: **xpath** Cluster: **external-objects** Status: **active**
Originator: **Michael Kay**

Description

Part of the strength of external functions is that they can return objects that are outside the scope of the XPath type system. For example, a function library for performing SQL database access may have a function sql:connect() that returns an object representing a database connection. This object may be passed as an argument to calls on other functions in the SQL function library.

The way this is handled in XPath 1.0 is that the XPath specification defines four data-types, and explicitly leaves the host language free to define additional data types. We could probably live with a similar solution for XPath 2.0, but it's a fudge, and I think we ought to try and do better.

Note that the only things you can do with an external object (under the XSLT 1.0 rules) are to assign it to a variable, or pass it as the argument to another function. In practice I think implementations also allow external objects to have additional behavior, for example they might allow conversion to a string when used in a context where a string is required. I think we should leave such behavior implementation-defined rather than saying that it is always an error.

The question arises as to where external objects should fit into the type hierarchy. Should they be a top-level thing at the same level as "sequence", or should they be one level down, along with "node" and "atomic value"? I think it makes most sense to preserve the principle "everything is a sequence", which means that there are now three kinds of item: nodes, atomic values, and external objects.

Handling this rigorously sounds like quite a pervasive change to the spec, but I don't think it's as bad as it seems. I don't think we should add any language features to support external objects, with the possible exception of a keyword "external" in the type syntax so that one can test for it using "instance of". Functions and operators that work on any sequence (for example, count) should treat an external object like any other item in the sequence. Operations that expect nodes will fail if presented with an external object; operations that expect atomic values will also fail, except that implementations may define fallback conversions from external objects to atomic values.

274. text-only-implementations: Text-node-only implementation possible?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **XSL WG**

Description

The current proposed text for XQuery element constructors precludes a "text node only" implementation. Since XSL should have semantics aligned with XQuery a decision needs to be made by the XSL WG as to if this restriction is acceptable to construction in XSLT.

275. xquery-validate-identity: Should validation also check identity constraints?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan Robie**

Description

Should validation also check identity constraints? Note that valid identity constraints of partial documents does not imply validity of the document as a whole. The current document says that identity constraints are not checked.

276. xquery-superfluous-xsi:type: Does validation introduce new xsi:type attributes?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan Robie**

Description

Does validation introduce new xsi:type attributes, or should xsi:type attributes introduced for validation be removed?

Note: The XPath telcon suggested that we eliminate this issue and simply say that any attributes introduced for the purpose of validation are removed as part of the validate semantics.

277. xquery-add-xsi:type: Should validate introduce xsi:type?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan Robie**

Description

Should validate introduce xsi:type to reflect type annotations that are not explicitly stated using an xsi:type in the source document?

The current draft introduces xsi:type to reflect inferred or dynamically computed types.

278. xpath-is-xsi:type-attribute: Does //@xsi:type match an xsi:type attribute?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan Robie**

Description

Does //@xsi:type match an xsi:type attribute? Compare to xml:lang, xml:space, xmlns, which are not represented as attributes in the XML Information Set. Note, however, that xsi:type is represented as an attribute in the Infoset. Should we treat it like other special XML attributes in the Data Model?

279. xpath-lightweight-cast: Should there be a lightweight cast?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan Robie**

Description

Should there be any provision for a lightweight cast that does not observe facets? Phil Wadler has suggested that 'validate' be used whenever full schema validation is desired, and 'cast' be used as a lightweight validation, which can be used for either simple or complex types, but which does not supply defaults, enforce facets, or check integrity constraints. It may be easier to optimize through cast than through validate, but supporting both constructs may confuse users due to their similarity. He suggests the following syntax for these expressions:

```
('cast'|'validate') as SequenceType ()
('cast'|'validate') 'in' ContextExpr { Expr }
```

280. xpath-list-simple-type: Do we need to distinguish atomic simple types and list simple types?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan Robie**

Description

XML Schema supports both atomic simple types and list simple types. Do we need to distinguish them in our grammar?

281. xpath-unknown-simple-type: Representing the type name of untyped character data

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan Robie**

Description

We need a way to represent the type name of untyped character data. In XML Schema, there is no one type that corresponds to this - in strict mode, elements whose content is untyped character data will have the type 'anyType' and attributes whose content is untyped character data will have the type 'anySimpleType', but this interacts in somewhat complex ways with lax and skip validation.

For now, we assume that character data may be associated with a SimpleType whose name is 'xs:unknownSimpleType'. This type does not currently exist in XML Schema, so we must coordinate with them on this issue.

If Schema does not provide a single type to represent untyped character data, we will provide a keyword for it in the ItemType production.

282. xpath-xs:numeric-type: xs:numeric type

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan Robie**

Description

For our Formal Semantics, it would be very helpful to have a simple type to represent all numerics. For now, we are using the SimpleType name "xs:numeric". In XML Schema terms, this would be a union type. We must coordinate with XML Schema to see if they can support this. If not, we will provide a keyword for numerics in the ItemType production.

283. xpath-document-element-production: Is the DocumentElement syntax production necessary?

Locus: **xpath** Cluster: **syntax-productions** Status: **active**
Originator: **Jonathan Robie**

Description

Is DocumentElement necessary?

Proposed Resolution

No.

284. xquery-static-named-typing: Static Named Typing

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Jonathan Robie**

Description

The typing must be covered by a static interpretation. If the Formal Semantics encounters difficulties in producing the static semantics, we may need to change the dynamic semantics.

285. type-information-QName: Is the QName denoting type information optional or required?

Issue Class: **T** Locus: **xpath** Cluster: **type-semantics** Status: **active**
Originator: **Phil Wadler**

Description

Is the QName that denotes type information in an element or attribute optional or required?

286. element-construction-vs-streaming: Element Construction vs Streaming

Locus: **xquery** Cluster: **constructor-expr** Status: **active**
Originator: **Michael Rys**

Description

The element construction rules in 2.8 make efficient streaming of element construction difficult/impossible. For example, we execute the following expression on a stream and serialize the result in a streaming processing (such as XSLT like applications):

<foo>{"foo", "bar", if (expr) then "baz" else <baz/>}</foo>

If expr is true, then this is serialized:

<foo>foo bar baz</foo>

If expr is false, then this is serialized: <foo>foobar<baz/></foo>

The implementation must cache up the "foo" and "bar" strings, just in case a sub-element node is constructed. If not, then I must insert a space between "foo" and "bar". This seems to contradict one of our explicit use scenarios in the XML Query Requirements (section 2.5).

Interactions and Input

Cf. Functional Status of Comma

Cf. Element Constructor Attribute Order

287. functional-status-of-comma: Functional Status of Comma

Locus: **xpath** Cluster: **syntax** Status: **active**
Originator: **Michael Rys**

Description

Is , a dyadic operator or (, , ,) a polyargument operator? The element construction rules seem to indicate the later, but the document is not clear. If , is a dyadic node-concatenation operation, then it would be more streamable than as an argument separator for the element constructor.

Interactions and Input

Cf. Element Construction vs Streaming

Cf. Element Constructor Attribute Order

288. element-constructor-attribute-order: Element Constructor Attribute Order

Locus: **xquery** Cluster: **constructor-expr** Status: **active**
Originator: **Michael Rys**

Description

Unlike XSLT, attributes seem to be allowed anywhere which makes streaming implementations impossible. XSLT has the following rule:

"The following are all errors:

* Adding an attribute to an element after children have been added to it; implementations may either signal the error or ignore the attribute."

Example:

```
let $e := expr
return <a>{<b/>, ($e/@* | $e/*)</a>
```

XSLT would consider this an error. XQuery does not. This makes streaming impossible (see #element-construction-vs-streaming). Note that while the above query can easily be rewritten, there are cases (e.g. function calls) where the rewrite is not possible.

Interactions and Input

Cf. Element Construction vs Streaming

Cf. Functional Status of Comma

Proposed Resolution

XQuery should allow implementations to disallow attribute nodes that are specified after the first child node in the sequence. It should not allow disregarding them.

289. attribute-value-construction-from-elements: Attribute Value Construction from Elements

Locus: **xquery** Cluster: **constructor-expr** Status: **active**
Originator: **Michael Rys**

Description

What does XQuery do if assigning a list of element nodes to an attribute if the type of the attribute is not a list type?

Example:

```
<book isbn="{$i/booknum}" />:
```

This query implicitly gets rewritten to: <book isbn="{data($i/booknum)}" />

What happens if there are more than one booknum elements and ISBN is untyped or not a list type?

290. element-attribute-constructor-name-type: Element Attribute Constructor Name Type

Locus: **xquery** Cluster: **constructor-expr** Status: **active**
Originator: **Michael Rys**

Description

Does the name expression on dynamically computed names in element/attribute constructors be of type QName without implicit cast from string, QName with implicit cast from string, or string?

Proposed Resolution

If the name is constructed by an expression, the expected type is xs:QName. Xs:string is in general implicitly cast to xs:QName (and xs:anyURI).

291. element-construction-anySimpleType-sequence-content: Element Construction anySimpleType Sequence Content

Locus: **xquery** Cluster: **constructor-expr** Status: **active**
Originator: **Michael Rys**

Description

The current element construction treats sequences of anySimpleType values different from other simple typed values. This seems inconsistent. The document should make it clear why there is a difference. If there is a reason, is the reason good enough?

292. element-construction-sequence-vs-multi-expr: Element Construction Sequence vs Multi-expr

Locus: **xquery** Cluster: **constructor-expr** Status: **active**
Originator: **Michael Rys**

Description

Which element construction rules cover

```
<e>{1, 2}</e>
vs
<e>{1}{2}</e>
```

293. cdata-charref-semantics: Cdata and CharRef Semantics

Locus: **xquery** Cluster: Status: **active**
Originator: **Michael Rys**

Description

The data model cannot represent CDATA or CharRef, since the Information Set looses this information.

Proposed Resolution

The XQuery document should make it clear that:

1. CDATA sections and CharRefs inside XQueries that are not embedded inside XML (which is what the XQuery document only talks about), are syntactic helps to write queries that otherwise would need entitization (in the case if CDATA sections) or a unicode input device (CharRefs).

2. Implementations can chose to use this information as serialization hints to preserve the CDATA and entitization.

294. type-annotations-on-all-elems-attrs: Should all elements and attributes have type annotations?

Locus: **xquery** Cluster: Status: **active**
Originator: **Philip Wadler**

Description

Should all elements and attributes have type annotations? In particular, should a newly-constructed element or attribute have a generic type annotation such as "anyType", rather than the current proposal in which it has no type annotation at all? (This is largely a cosmetic issue, about whether "anyType" should be denoted by an explicit annotation or by the absence of an annotation.)

F Revision Log (Non-Normative)

F.1 10 Apr 2002

- Grammar of path expressions has been reorganized to conform more closely to description in Formal Semantics.
- Material has been added on Named Typing, including detailed syntax for SequenceType declaration, explanation of static and dynamic type-checking, and new rules for type-matching in function calls.
- Explanatory material has been added on typed values, document order, and the error value.
- Some terminology has been changed for consistency with other documents; for example, "simple value" changed to "atomic value".
- Some changes have been made to rules for handling untyped data in arithmetic and comparison expressions.
- A more detailed specification is provided for the static and dynamic semantics of `treat` and `assert`.
- A new section has been added on `validate` expressions.
- More details have been provided on the semantics of element and attribute constructors, including their data model representations.
- A new section has been added on Input Functions, describing the input(), collection(), and document() functions.
- An Operator Mapping Table has been added (Appendix B.)
- Quantified expressions now permit multiple variable bindings.
- The semantics of `sortby` have changed: all ordering keys must now have the same type, and explicit control is provided over the placement of empty sort keys.
- The semantics of `and`, `or`, and quantified expressions have changed so that an implementation is not forced to evaluate all the operand expressions if it encounters an error or a conclusive result.
- Union, Intersect, and Except expressions now apply to nodes only, not to atomic values.
- The definitions of the precedes and follows operators have been made independent of the preceding and following axes, since these axes are not supported in XQuery.
- The operators `==` and `!==` have been changed to `is` and `isnot`.
- Quotes are now required around attribute values in element constructors, even if the value is computed by an enclosed expression.
- It has been clarified that the result of a dereference operator is a node list in document order without duplicates.
- It has been clarified that the default element namespace applies to types as well as to elements.

F.2 20 Dec 2001

- Document had been completely rewritten during the joint work on XPath (see text in Intro).
- The following issues have been resolved:

  Issue-0028: 3-value-logic

  > Should not(()) == true or () or not3 function be defined?

  Issue-0032: for-expr

  > Do we really require `for` at the XPath expression level?

  Issue-0037: datatypes

  > Datatypes... XQuery suggests that XPath 2.0 should have a richer set of datatypes based on the built-in types of XML Schema and the notion of an ordered sequence.

  Issue-0068: context

  > Do we need a Context Item to track the "current item" in a for itteration, and, if so, how should it work? How should "." work in relation to it?

  Issue-0070: context

  > Does the context include a default namespace declaration?

  Issue-0071: context

  > Do we need a datatype names binding in the context?

  Issue-0072: context

  > How should qname-to-collation bindings be handled?

  Issue-0088: (in)equality-operators

  > Implied existential quantifiers

  Issue-0102: null-issue-functions-on-empty

  > Functions on Empty Sequences... What should happen if a function expecting one element is invoked on an empty sequence?

  Issue-0103: function-app

  > Functions on Sequences... What should happen if a function expecting one element is invoked on a sequence of more than one element?

  Issue-0107: null-functions-on-sequences-b

  > Path iteration... How should the current node be passed to a function in a path-step?

  Issue-0110: syntax

  > Should we use is and isnot instead of == and !==.

  Issue-0178: type-conversions

  > Semantics of positional predicates in XPath

  Issue-0179: type-conversion

  > Function call rules needed