XML Path Language (XPath) 2.0

W3C Working Draft 02 May 2003

This version:  
http://www.w3.org/TR/2003/WD-xpath20-20030502

Latest version:  
http://www.w3.org/TR/xpath20/

Previous versions:  
http://www.w3.org/TR/2002/WD-xpath20-20021115/  
http://www.w3.org/TR/2002/WD-xpath20-20020816/  
http://www.w3.org/TR/2002/WD-xpath20-20020430/  
http://www.w3.org/TR/2001/WD-xpath20-20011220/  

Editors:  
Anders Berglund (XSL WG), IBM Research <alrb@us.ibm.com>  
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>  
Michael Kay (XSL WG), Software AG <Michael.Kay@softwareag.com>  
Jonathan Robie (XML Query WG), DataDirect Technologies <jonathan.robie@datadirect-technologies.com>  
Jérôme Siméon (XML Query WG), Bell Labs, Lucent Technologies <simeon@research.bell-labs.com>

Copyright © 2003 W3C® (MIT, ERCIM, Keio). All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

Abstract

XPath is a language for addressing parts of an XML document.

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/.

This document is the result of joint work by the XSL and XML Query 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 contains a new syntax for identifying types, and new features that allow path expressions to select element and attribute nodes based on their types as well as their names. It changes some of the details of the semantics of arithmetic and comparison operators, including making the value comparison operators transitive. It includes some grammar changes, including changes in the comment delimiters and the syntax of the cast expression. A detailed list of changes can be found in H Revision Log.

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, in particular comments on Issue 510.
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/).

XPath 2.0 has been defined jointly by the XML Query Working Group and the XSL Working Group (both
part of the XML Activity).

Patent disclosures relevant to this specification may be found on the XSL Working Group’s patent
disclosure page at http://www.w3.org/Style/XSL/Disclosures.

Table of Contents

1 Introduction
2 Basics
   2.1 Expression Context
      2.1.1 Static Context
      2.1.2 Evaluation Context
   2.2 Input Functions
   2.3 Expression Processing
      2.3.1 Document Order
      2.3.2 Typed Value and String Value
   2.4 Types
      2.4.1 Predefined Types
      2.4.2 Type Checking
      2.4.3 SequenceType
         2.4.3.1 SequenceType Matching
      2.4.4 Type Conversions
         2.4.4.1 Atomization
         2.4.4.2 Effective Boolean Value
   2.5 Error Handling
      2.5.1 Kinds of Errors
      2.5.2 Handling Dynamic Errors
      2.5.3 Errors and Optimization
   2.6 Optional Features
      2.6.1 Basic XPath
      2.6.2 Static Typing Feature
3 Expressions
   3.1 Primary Expressions
      3.1.1 Literals
      3.1.2 Variables
      3.1.3 Parenthesized Expressions
      3.1.4 Function Calls
      3.1.5 XPath Comments
   3.2 Path Expressions
      3.2.1 Steps
         3.2.1.1 Axes
         3.2.1.2 Node Tests
      3.2.2 Predicates
      3.2.3 Unabbreviated Syntax
      3.2.4 Abbreviated Syntax
   3.3 Sequence Expressions
      3.3.1 Constructing Sequences
      3.3.2 Combining Sequences
   3.4 Arithmetic Expressions
   3.5 Comparison Expressions
      3.5.1 Value Comparisons
      3.5.2 General Comparisons
      3.5.3 Node Comparisons
1 Introduction

The primary purpose of XPath is to address parts of an [XML] document. XPath uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. This logical structure is known as the data model, and is described in the [XQuery 1.0 and XPath 2.0 Data Model] document. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document.

XPath is designed to be embedded in a host language such as [XSLT 2.0] or [XQuery]. XPath has a natural subset that can be used for matching (testing whether or not a node matches a pattern); this use of XPath is described in [XSLT 2.0].

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

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

- The XPath data model defines the information in an XML document that is available to an XPath processor. The data model is defined in [XQuery 1.0 and XPath 2.0 Data Model].
The static and dynamic semantics of XPath are formally defined in [XQuery 1.0 Formal Semantics]. This document is useful for implementors and others who require a rigorous definition of XPath.

The type system of XPath is based on [XML Schema].

The default library of functions and operators supported by XPath is defined in [XQuery 1.0 and XPath 2.0 Functions and Operators].

This document specifies a grammar for XPath, using the same Basic EBNF notation used in [XML], except that grammar symbols always have initial capital letters. Unless otherwise noted (see A.2 Lexical structure), whitespace is not significant in the grammar. Grammar productions are introduced together with the features that they describe, and a complete grammar is also presented in the appendix [A XPath Grammar].

In the grammar productions in this document, nonterminal symbols are underlined and literal text is enclosed in double quotes. Certain productions (including the productions that define DecimalLiteral, DoubleLiteral, and StringLiteral) employ a regular-expression notation. The following example production describes the syntax of a function call:

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

The production should be read as follows: A function call consists of a QName followed by an open-parenthesis. The open-parenthesis is followed by an optional argument list. The argument list (if present) consists of one or more expressions, separated by commas. The optional argument list is followed by a close-parenthesis. The symbol ExprSingle denotes an expression that does not contain any top-level commas (since top-level commas in a function call are used to separate the function arguments).

Certain aspects of language processing are described in this specification as implementation-defined or implementation-dependent. These terms are defined as follows:

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

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

<table>
<thead>
<tr>
<th>Editorial note</th>
</tr>
</thead>
<tbody>
<tr>
<td>A future version of this document will include links between terms (in bold font) and their definitions.</td>
</tr>
</tbody>
</table>

## 2 Basics

The basic building block of XPath 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. XPath is a functional language which allows various kinds of expressions to be nested with full generality. (However, unlike a pure functional language, it does not allow variable substitutability if the variable definition contains construction of new nodes.) XPath is also a strongly-typed language in which the operands of various expressions, operators, and functions must conform to the expected types.

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

The value of an expression is always a sequence, which 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]. Each node has a unique node identity. 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.
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 `fn:` is considered to be bound to the namespace of XPath/XQuery functions, `http://www.w3.org/2003/05/xpath-functions` (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` are used without a namespace prefix. Also, this document assumes that the default function namespace is set to the namespace of XPath/XQuery functions, so function names appearing without a namespace prefix can be assumed to be in this namespace.

### 2.1 Expression Context

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

#### 2.1.1 Static Context

The **static context** of an expression is the information that is available during static analysis of the expression, prior to its evaluation. This information can be used to decide whether the expression contains a static error.

Any component of the static context that is not assigned a default value in the XPath specification, and is not assigned a value by the host language, may be assigned an implementation-defined initial value. If processing of an expression relies on some component of the static context that has not been assigned a value, a static error is raised.

Static context consists of the following components:

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

- **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 element namespace.** 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 function namespace.** This is a namespace URI. This namespace URI is used for any unprefixed QName appearing as the function name in a function call.

- **In-scope schema definitions.** This is a generic term for all the element, attribute, and type definitions that are in scope during processing of an expression. It includes the following three parts:

  - **In-scope type definitions.** The in-scope type definitions always include the built-in types of [XML Schema] and the predefined types in the namespace `http://www.w3.org/2003/05/xpath-datatypes`. Additional type definitions may be added to the in-scope type definitions by the host language.

    XML Schema distinguishes **named types**, which are given a QName by the schema designer, must be declared at the top level of a schema, and are uniquely identified by their QName, from **anonymous types**, which are not given a name by the schema designer, must be local, and are identified in an implementation-dependent way. Both named types and anonymous types can be present in the in-scope type definitions.

  - **In-scope element declarations.** Each element declaration is identified either by a QName (for a top-level element) or by an implementation-defined element identifier (for a local element). Element declarations may be provided by the language environment. An element declaration includes information about the **substitution groups** to which this element
**In-scope attribute declarations.** Each attribute declaration is identified either by a QName (for a top-level attribute) or by an implementation-defined attribute identifier (for a local attribute). Attribute declarations may be provided by the language environment.

- **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 expression. The QName represents the name of the variable, and the type represents its static data type.

  The static types of in-scope variables may be derived from static analysis of the expressions in which the variables are bound, or provided by the external environment.

- **In-scope functions.** This part of the static context defines the set of functions that are available to be called from within an expression. Each function is uniquely identified by its QName and its arity (number of parameters). The static context maps the QName and arity into a function signature and a function definition. The function signature specifies the static types of the function parameters and the function result.

  For each atomic type in the **in-scope type definitions**, there is a constructor function in the **in-scope functions**. Constructor functions are discussed in [3.10.4 Constructor Functions](#).

- **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 when necessary in the resolution of relative URIs (for example, by the `fn:resolve-uri` function.) The **base URI** is always provided by the external environment.

### 2.1.2 Evaluation Context

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

Any component of the evaluation context that is not assigned a default value in the XPath specification, and is not assigned a value by the host language, may be assigned an implementation-defined initial value. If processing of an expression relies on some component of the evaluation context that has not been assigned a value, a dynamic error is raised.

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

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

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

- **The context item** is the item currently being processed. An item is either an atomic value or a node. When the context item is a node, it can also be referred to as the **context node**. The context item is returned by the expression `"."`. When an expression `E1/E2` or `E1[E2]` is evaluated, each item in the
sequence obtained by evaluating \( E_1 \) becomes the context item in the inner focus for an evaluation of \( E_2 \).

- 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 `fn:position()`. When an expression \( E_1/E_2 \) or \( E_1[E_2] \) is evaluated, the context position in the inner focus for an evaluation of \( E_2 \) is the position of the context item in the sequence obtained by evaluating \( E_1 \). 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 \( E_1/E_2 \) or \( E_1[E_2] \) is evaluated, the context size in the inner focus for an evaluation of \( E_2 \) is the number of items in the sequence obtained by evaluating \( E_1 \).

- **Dynamic variables.** This is a set of (QName, value) pairs. It contains the same QNames as the in-scope variables in the static context for the expression. Each variable name is associated with a typed value. The dynamic type associated with the value of 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 typed value of a variable may be set by execution of an expression that binds a value to the variable, or by the external environment.

- **Current date and time.** This information represents an implementation-dependent point in time during processing of a query or transformation. It can be retrieved by the `fn:current-date`, `fn:current-time`, and `fn:current-dateTime` functions. If invoked multiple times during the execution of a query or transformation, these functions always return the same result.

- **Implicit timezone.** This is the timezone to be used when a date, time, or dateTime value that does not have a timezone is used in a comparison or in any other operation. This value is an instance of `xdt:dayTimeDuration` that is implementation-defined. See [ISO 8601] for the range of legal values of a timezone.

- **Input sequence.** An input sequence is a 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 by the host language.

### 2.2 Input Functions

XPath has a set of functions that provide access to input data. These functions are of particular importance because they provide a way in which an expression can reference a document or a collection of documents. The input functions are described informally here, 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 defined by the host language.

The input functions supported by XPath are as follows:

- The `fn:input` function, which takes no parameters, returns the input sequence. For example, the expression `fn:input()//@customer` returns all the `customer` elements that are descendants of nodes in the input sequence. If no input sequence has been bound, the `fn:input` function raises a dynamic error.

- The `fn: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 `fn:collection("http://example.org")//@customer` identifies all the `customer` elements that are descendants of nodes found in the collection whose URI is `http://example.org`.

- The `fn:doc` function, when its first argument is a string containing a single URI that refers to an XML document, returns a document node whose content is the Data Model representation of the given document.

If a given input function is invoked repeatedly with the same arguments during the scope of a single query or transformation, each invocation returns the same result.
2.3 Expression Processing

<table>
<thead>
<tr>
<th>Editorial note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some material in this section duplicates material in [XQuery 1.0 and XPath 2.0 Data Model]. Work is in progress to decide where this material will be normatively defined (see Issue 554.)</td>
</tr>
</tbody>
</table>

XPath 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 and atomic values. Before an XPath 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 might be converted to the Data Model by the following steps:

1. The document might be parsed using an XML parser that generates an XML Information Set (see [XML Infoset]).

2. The parsed document might 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).

3. If necessary, the PSVI can be transformed to make it acceptable for processing by a particular system. For example, if the implementation understands only built-in XML Schema types, user-defined typenames in the PSVI might be replaced by their built-in base types or by generic types such as xs:anyType (for elements) and xs:anySimpleType (for attributes).

4. The PSVI can be transformed into the Data Model by a process described in [XQuery 1.0 and XPath 2.0 Data Model]. During this transformation, an error is raised if the PSVI contains an element, attribute, or type whose expanded name matches a name in the in-scope schema definitions (ISSD), but whose definition in the PSVI is not consistent with the corresponding definition in the ISSD. Typenames in the PSVI that are not present in the ISSD are replaced by xs:anyType (for types of elements) or xs:anySimpleType (for types of attributes). At the conclusion of the transformation process, every element name, attribute name, and type name in the resulting Data Model instance matches an entry in the ISSD.

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. XPath 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 (except for the constraint that the result must be consistent with the in-scope schema definitions).

Each element or attribute node in the Data Model has an annotation that indicates its dynamic type. If the Data Model was derived from an input XML document, the dynamic types of the elements and attributes are derived from schema validation. The dynamic type of an element or attribute indicates its range of values—for example, an attribute named version might have the dynamic 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 dynamic type xdt:untypedAtomic.

The value of an element is represented by the children of the element node, which may include text nodes and other element nodes. The dynamic type of an element node indicates how the values in its child text nodes are to be interpreted. An element whose type is unknown (such as might occur in a schemaless document) is annotated with the type xs:anyType.

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

This document provides a description of how each kind of expression is processed. For each expression, the operands and result are instances of the Data Model.

The terms document order, typed value, and string value are described here because they are of particular importance for the processing of expressions.
2.3.1 Document Order

**Document order** defines a total ordering among all the nodes seen by the language processor. Informally, document order corresponds to a pre-order, depth-first, left-to-right traversal of the nodes in the Data Model.

Within a given 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 form of the document (after expansion of entities). Element nodes occur before their children, and the children of an element node occur before its following siblings. 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.

The relative order of nodes in distinct documents is implementation-defined but stable within a given query or transformation. Given two distinct documents A and B, if a node in document A is before a node in document B, then every node in document A is before every node in document B. The relative order among free-floating nodes (those not in a document) is also implementation-defined but stable.

2.3.2 Typed Value and String Value

Nodes have a **typed value** and a **string value** that can be extracted by calling the `fn:data` function and the `fn:string` function, respectively. The typed value of a node is a sequence of atomic values, and the string value of a node is a string. Element and attribute nodes also have a **type annotation**, which is a type identifier that is found in the in-scope type definitions. The type annotation represents the dynamic (run-time) type of the node. XPath does not provide a way to directly access the type annotation of an element or attribute node.

The typed value and string value for each kind of node are defined by the `dm:typed-value` and `dm:string-value` accessors in [XQuery 1.0 and XPath 2.0 Data Model]. The relationship between the typed value and the string value for various kinds of nodes is described and illustrated by examples below.

1. For text, document, comment, processing instruction, and namespace nodes, the typed value of the node is the same as its string value, as an instance of `xdt:untypedAtomic`. (The string value of a document node is formed by concatenating the string values of all its descendant text nodes, in document order.)

2. The typed value of an attribute node with the type annotation `xdt:untypedAtomic` is the same as its string value, as an instance of `xdt:untypedAtomic`. The typed value of an attribute node with any other type annotation is derived from its string value and type annotation in a way that is consistent with schema validation.

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

   Example: A2 is an attribute with type annotation `IDREFS`, which is a list type derived from `IDREF`. Its string value is "bar baz faz". The typed value of A2 is a sequence of three atomic values ("bar", "baz", "faz"), each of type `IDREF`. The typed value of A node is never treated as an instance of a named list type. Instead, if the type annotation of a node is a list type (such as `IDREFS`), its typed value is treated as a sequence of the underlying base type (such as `IDREF`).

3. For an element node, the relationship between typed value and string value depends on the node’s type annotation, as follows:

   1. If the type annotation is `xs:anyType`, or denotes a complex type with mixed content, then the typed value of the node is equal to its string value, as an instance of `xdt:untypedAtomic`.

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

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

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

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

3. If the type annotation denotes a complex type with empty content, then the typed value of the node is the empty sequence.

4. If the type annotation denotes a complex type with non-mixed complex content, then the typed value of the node is undefined. The `fn:data` function raises an error when applied to such a node.

Example: E5 is an element node with the type annotation `weather`, which is a complex type whose content type specifies `elementOnly`. E5 has two child elements named `temperature` and `precipitation`. The typed value of E5 is undefined, and the `fn:data` function applied to E5 raises an error.

### 2.4 Types

XPath is a strongly typed language with a type system based on [XML Schema]. When the type of a value is incompatible with the expected type for the context in which it is used, a type error is raised. A type error may be detected and reported during the analysis phase or during the evaluation phase, as described in 2.4.2 Type Checking.

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

#### 2.4.1 Predefined Types

All the built-in types of [XML Schema] are recognized by XPath. These built-in types are in the namespace `http://www.w3.org/2001/XMLSchema`, which is represented in this document by the prefix `xs`. Some examples of built-in schema types include `xs:integer`, `xs:string`, and `xs:date`.

In addition, XPath recognizes the predefined types listed below. All these predefined types are in the namespace `http://www.w3.org/2003/05/xpath-datatypes`, which is represented in this document by the prefix `xdt`.

1. `xdt:anyAtomicType` is an abstract type that includes all atomic values (and no values that are not atomic). It is a subtype of `xs:anySimpleType`. Which is the base type for all simple types, including atomic, list, and union types. All specific atomic types such as `xs:integer`, `xs:string`, and `xdt:untypedAtomic`, are subtypes of `xdt:anyAtomicType`.

2. `xdt:untypedAtomic` is a specific atomic type used for untyped data, such as text that is not given a specific type by schema validation. It has no subtypes.

3. `xdt:dayTimeDuration` is a subtype of `xs:duration` whose lexical representation contains only day, hour, minute, and second components.

4. `xdt:yearMonthDuration` is a subtype of `xs:duration` whose lexical representation is restricted to contain only year and month components.

For more details about predefined types, see [XQuery 1.0 and XPath 2.0 Functions and Operators].

Additional type definitions may be added by the host language environment via the in-scope type definitions.
2.4.2 Type Checking

XPath defines two phases of processing called the **analysis phase** and the **evaluation phase**.

The **analysis phase** depends on the expression itself and on the **static context**. The analysis phase does not depend on any input data. The purpose of type-checking during the analysis phase is to provide early detection of type errors and to compute the type of a result.

During the analysis phase, 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 `5 + 1.2` is `xs:decimal`. 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 static type checking is in effect and an operand of an expression is found to have a static type that is not appropriate for that operand, a type error is raised. If static type checking raises no errors and assigns a static type `T` to an expression, then execution of the expression on valid input data is guaranteed either to produce a value of type `T` or to raise a dynamic error.

The **evaluation phase** is performed only after successful completion of the **analysis phase**. The evaluation phase depends on input data, on the expression being evaluated, and on the **evaluation context**. During the evaluation phase, a **dynamic type** is associated with each value as it is computed. The dynamic type of a value may be either a structural type (such as "sequence of integers") or a named type. 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 incompatible with the expected type for that operand, a type error is raised.

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

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

2.4.3 SequenceType

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

```
SequenceType
```
QNames appearing in a SequenceType have their prefixes expanded to namespace URIs by means of the **in-scope namespaces** and the **default element namespace**. It is a static error to use a name in a SequenceType if that name is not found in the appropriate part of the **in-scope schema definitions**. If the name is used as an element name, it must appear in the **in-scope element declarations**; if it is used as an attribute name, it must appear in the **in-scope attribute declarations**; and if it is used as a type name, it must appear in the **in-scope type definitions**.

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

- `xs:date` refers to the built-in Schema type `date`
- `attribute()?` refers to an optional attribute
- `element()` refers to any element
- `element(po:shipto, po:address)` refers to an element that has the name `po:shipto` (or is in the substitution group of that element), and has the type annotation `po:address` (or a subtype of that type)
- `element(po:shipto, `*) refers to an element named `po:shipto` (or in the substitution group of `po:shipto`), with no restrictions on its type
- `element(*, po:address)` refers to an element of any name that has the type annotation `po:address` (or a subtype of `po:address`). If the keyword `nillable` were used following `po:address`, that would indicate that the element may have empty content and the attribute `xsi:nil="true"`, even though the declaration of the type `po:address` has required content.
- `node()` refers to a sequence of zero or more nodes of any type
- `item()` refers to a sequence of one or more nodes or atomic values

### 2.4.3.1 SequenceType Matching
During processing of an expression, 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 is consistent with 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

As stated above, an item may be a node or an atomic value. The process of matching a given item against a given ItemType is performed as follows:

1. The ItemType `item()` matches any single item. For example, `item()` matches the atomic value `i` or the element `<a/>`.

2. If an ItemType consists simply of a QName, that QName must be the name of an atomic type that is in the in-scope type definitions; otherwise a static error is raised. An ItemType consisting of the QName of an 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 an atomic type derived by restriction from `xs:decimal`. The named atomic type may be a generic type such as `xdt:anyAtomicType`. (Note that names of non-atomic types such as `xs:IDREFS` are not accepted in this context.)

3. The following ItemTypes (referred to generically as **KindTests**) match nodes:

   1. `node()` matches any node.
   2. `text()` matches any text node.
   3. `processing-instruction()` matches any processing instruction node.
   4. `processing-instruction(\texttt{N})` matches any processing instruction node whose name (called its "PITarget" in XML) is equal to \texttt{N}, where \texttt{N} is a StringLiteral. Example: `processing-instruction("browser")` matches any processing instruction directed to the application named `browser`.
   5. `comment()` matches any comment node.
   6. `document-node()` matches any document node.
   7. `document-node(\texttt{E})` matches any document node whose content consists of exactly one element node that matches \texttt{E}, where \texttt{E} is an **ElementTest** (see below). Example: `document-node(element(book))` matches any document node whose content consists of exactly one element node named `book`, that conforms to the schema declaration for the top-level element `book`.
   8. An **ElementTest** (see below) matches an element node, optionally qualifying the node by its name, its type, or both.
   9. An **AttributeTest** (see below) matches an attribute node, optionally qualifying the node by its
An ElementTest is used to match an element node by its name and/or type. An ElementTest may take one of the following forms:

1. `element()`, or `element(*)`, or `element(*,*)`. All these forms of ElementTest are equivalent, and they all match any single element node, regardless of its name or type.

2. `element(N, T)`, where `N` is a QName and `T` is a QName optionally followed by the keyword `nillable`. In this case, `T` must be the name of a top-level type definition in the in-scope type definitions. The ElementTest matches a given element node if:
   
   1. the name of the given element node is equal to `N` (expanded QNames match), or is equal to the name of any element in a substitution group headed by a top-level element with the name `N`; and:
   
   2. the type annotation of the given element node is `T`, or is a named type that is derived by restriction or extension from `T`. However, this test is not satisfied if the given element node has an attribute `xsi:nil="true"` and `T` does not specify `nillable`.

The following examples illustrate this form of ElementTest, matching an element node whose name is `person` and whose type annotation is `surgeon` (the second example permits the element to have `xsi:nil="true"`):

```
  element(person, surgeon)
  element(person, surgeon nillable)
```

3. `element(N)`, where `N` is a QName. This form is very similar to the previous form, except that the required type, rather than being named explicitly, is taken from the top-level declaration of element `N`. In this case, `N` must be the name of a top-level element declaration in the in-scope element declarations. The ElementTest matches a given element node if:

   1. the name of the given element node is equal to `N` (expanded QNames match), or is equal to the name of any element in a substitution group headed by `N`; and:

   2. the type annotation of the given element node is the same as, or derived by restriction or extension from, the type of the top-level declaration for element `N`. The types to be compared may be either named types (identified by QNames) or anonymous types (identified in an implementation-dependent way). However, this test is not satisfied if the given element node has an attribute `xsi:nil="true"` and the top-level declaration for element `N` does not specify `nillable`.

The following example illustrates this form of ElementTest, matching an element node whose name is `person` and whose type annotation conforms to the top-level `person` element declaration in the in-scope element declarations:

```
  element(person)
```

4. `element(N, *)`, where `N` is a QName. This ElementTest matches a given element node if the name of the node is equal to `N` (expanded QNames match), or is equal to the name of any element in a substitution group headed by a top-level element with the name `N`. The given element node may have any type annotation.

The following example illustrates this form of ElementTest, matching any element node whose name is `person` or is in the `person` substitution group, regardless of its type annotation:

```
  element(person, *)
```

5. `element(*, T)`, where `T` is a QName optionally followed by the keyword `nillable`. In this case, `T` must be the name of a top-level type definition in the in-scope type definitions. The ElementTest
matches a given element node if the node’s type annotation is $T$, or is a named type that is derived by restriction or extension from $T$. However, this test is not satisfied if the given element node has an attribute `xsi:nil="true"` and $T$ does not specify `nillable`.

The following examples illustrate this form of `ElementTest`, matching any element node whose type annotation is `surgeon`, regardless of its name (the second example permits the element to have `xsi:nil="true"`):

```
   element(*, surgeon)
   element(*, surgeon nillable)
```

6. `element(P)`, where $P$ is a valid schema context path beginning with a top-level element name or type name in the `in-scope schema definitions` and ending with an element name. This `ElementTest` matches a given element node if:

   1. the name of the given element node is equal to the last name in the path (expanded QNames match), and:
   
   2. the type annotation of the given element node is the same as the type of the element represented by the schema path $P$.

The following examples illustrate this form of `ElementTest`, matching element nodes whose name is `person`. In the first example, the node must conform to the schema definition of a `person` element in a `staff` element in a `hospital` element. In the second example, the node must conform to the schema definition of a `person` element within the top-level type `schedule`:

```
   element(hospital/staff/person)
   element(type(schedule)/person)
```

An `AttributeTest` is used to match an attribute node by its name and/or type. An `AttributeTest` may take one of the following forms:

1. `attribute()`, or `attribute(@*)`, or `attribute(@*,*)`. All these forms of `AttributeTest` are equivalent, and they all match any single attribute node, regardless of its name or type.

2. `attribute(@N, T)`, where $N$ and $T$ are QNames. In this case, $T$ must be the name of a top-level simple type definition in the `in-scope type definitions`. This `AttributeTest` matches a given attribute node if:

   1. the name of the given attribute node is equal to $N$ (expanded QNames match), and:
   
   2. the type annotation of the given attribute node is $T$, or is a named type that is derived by restriction from $T$.

The following example illustrates this form of `AttributeTest`, matching an attribute node whose name is `price` and whose type annotation is `currency`:

```
   attribute(@price, currency)
```

3. `attribute(@N)`, where $N$ is a QName. This form is very similar to the previous form, except that the required type, rather than being named explicitly, is taken from the top-level attribute declaration with name $N$. In this case, $N$ must be the name of a top-level attribute declaration in the `in-scope attribute declarations`. This `AttributeTest` matches a given attribute node if:

   1. the name of the given attribute node is equal to $N$ (expanded QNames match), and:
   
   2. the type annotation of the given attribute node is the same as, or derived by restriction from, the type of the top-level attribute declaration for $N$. The types to be compared may be either named types (identified by QNames) or anonymous types (identified in an implementation-dependent way).
The following example illustrates this form of AttributeTest, matching an attribute node whose name is `price` and whose type annotation conforms to the schema declaration for a top-level `price` attribute:

```
attribute(@price)
```

4. `attribute(@N, `)`

   where `N` is a QName. This AttributeTest matches a given attribute node if the name of the node is equal to `N` (expanded QNames match). The given attribute node may have any type annotation.

The following example illustrates this form of AttributeTest, matching any attribute node whose name is `price`, regardless of its type annotation:

```
attribute(@price, `)`
```

5. `attribute(@*, `T`)

   where `T` is a QName. In this case, `T` must be the name of a top-level simple type definition in the in-scope type definitions. This AttributeTest matches a given attribute node if the node’s type annotation is `T`, or is a named type that is derived by restriction from `T`.

The following example illustrates this form of AttributeTest, matching any attribute node whose type annotation is `currency`, regardless of its name:

```
attribute(@*, currency)
```

6. `attribute(P, `)`

   where `P` is a valid schema context path beginning with a top-level element name or type name in the in-scope schema definitions, and ending with an attribute name (preceded by `@`). This AttributeTest matches a given attribute node if:

   1. the name of the given attribute node is equal to the last name in the path (expanded QNames match), and:
   2. the type annotation of the given attribute node is the same as the type of the attribute represented by the schema path `P`.

The following examples illustrate this form of AttributeTest, matching attribute nodes whose name is `price`. In the first example, the node must conform to the schema definition of a `price` attribute in a `product` element in a `catalog` element. In the second example, the node must conform to the schema definition of a `plan` attribute within the top-level type `plan`:

```
attribute(catalog/product/@price)
attribute(type(plan)/@price)
```

### 2.4.4 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 automatically perform certain type conversions, 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 3.1.4 Function Calls. Other operators that provide special conversion rules include arithmetic operators, which are discussed in 3.4 Arithmetic Expressions, and value comparisons, which are discussed in 3.5.1 Value Comparisons.

#### 2.4.4.1 Atomization

Type conversions sometimes depend on a process called **atomization**, which is used when a sequence of atomic values is required. The result of atomization is either a sequence of atomic values or a type error. Atomization of a sequence is defined as the result of invoking the `fn:data` function on the sequence, as defined in [XQuery 1.0 and XPath 2.0 Functions and Operators].

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

- If the item is an atomic value, it is returned.
- If the item is a node, it is replaced by its typed value.

Atomization may be used in processing the following types of expressions:

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

2.4.4.2 Effective Boolean Value

Under certain circumstances (listed below), it is necessary to find the effective boolean value of a sequence. This is defined as the result of invoking the fn:boolean function on the sequence, as defined in [XQuery 1.0 and XPath 2.0 Functions and Operators].

The semantics of fn:boolean are repeated here for convenience. fn:boolean returns false if its operand is any of the following:

- An empty sequence.
- The boolean value false.
- A zero-length string ("").
- A numeric value that is equal to zero.
- The double or float value NaN.

Otherwise, fn:boolean returns true.

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

- Logical expressions (and, or)
- The fn:not function
- Certain types of predicates, such as $a$|$b$.
- Conditional expressions (if)
- Quantified expressions (some, every)

Note that the definition of effective boolean value is not used when casting a value to the type xs:boolean.

2.5 Error Handling

2.5.1 Kinds of Errors

As described in 2.4.2 Type Checking, XPath defines an analysis phase, which does not depend on input data, and an evaluation phase, which does depend on input data.

The result of the analysis phase is either success or one or more type errors and/or static errors. Type errors reported by the analysis phase occur when the static type of an expression is not correct for the context in which it appears. Static errors are non-type-related errors such as syntax errors. The means by which errors are reported during the analysis phase is implementation-defined.
The result of the evaluation phase is either a result value, a type error, or a dynamic error. Type errors are raised during the evaluation phase when the dynamic type of an expression is not correct for the context in which it appears. Dynamic errors are non-type-related errors such as numeric overflow. If evaluation of an expression yields a value (that is, it does not raise an error), the value must be the value specified by the dynamic semantics defined in [XQuery 1.0 Formal Semantics].

If an implementation can determine by static analysis that an expression will necessarily raise a dynamic error (for example, because it attempts to construct a decimal value from a constant string that is not in the lexical space of xs:decimal), the implementation is allowed to report this error during the analysis phase (as well as during the evaluation phase).

[XQuery 1.0 Formal Semantics] defines the set of static, dynamic, and type errors. In addition to these errors, an XPath implementation may raise implementation-defined warnings, either during the analysis phase or the evaluation phase. The circumstances in which warnings are raised, and the ways in which warnings are handled, are implementation-defined.

In addition to the errors defined in this specification, an implementation may raise a dynamic error if insufficient resources are available for processing a given expression. For example, an implementation may specify limitations on the maximum numbers or sizes of various objects. These limitations, and the consequences of exceeding them, are implementation-defined.

2.5.2 Handling Dynamic Errors

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

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

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

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

A dynamic error carries an error value, which may be a single item or an empty sequence. For example, an error value might be an integer, a string, a QName, or an element. An implementation may provide a mechanism whereby an application-defined error handler can process error values and produce diagnostics; in the absence of such an error handler, the string-value of the error value may be used directly as an error message.

A dynamic error may be raised by a built-in function or operator. For example, the input function raises an error if the input sequence is not defined in the evaluation context.

An error can be raised explicitly by calling the fn:error function, which only raises an error and never returns a value. The fn:error function takes an optional item as its parameter, which is used as the error value. For example, the following function call raises a dynamic error whose error value is a string:

```
fn:error(fn:concat("Unexpected value ", fn:string($v)))
```

2.5.3 Errors and Optimization

Because different implementations may choose to evaluate or optimize an expression in different ways, the detection and reporting of dynamic errors is implementation dependent.

When an implementation is able to evaluate an expression without evaluating some subexpression, the implementation is never required to evaluate that subexpression solely to determine whether it raises a dynamic error. For example, if a function parameter is never used in the body of the function, an implementation may choose whether to evaluate the expression bound to that parameter in a function call.
In some cases, an optimizer may be able to achieve substantial performance improvements by rearranging an expression so that the underlying operations such as projection, restriction, and sorting are performed in a different order than that specified in [XQuery 1.0 Formal Semantics]. In such cases, dynamic errors may occur that could not have occurred if the expression were evaluated as written. For example, consider the following expression:

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

This expression cannot fail with a casting error if it is evaluated exactly as written. An implementation is permitted, however, to reorder the predicates to achieve better performance (for example, by taking advantage of an index). This reordering could cause the above expression to fail. However, an expression must not be rearranged in a way that causes it to return a non-error result that is different from the result defined by [XQuery 1.0 Formal Semantics].

To avoid unexpected errors caused by reordering of expressions, tests that are designed to prevent dynamic errors should be expressed using conditional expressions, as in the following example:

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

In the case of a conditional expression, the implementation is required not to evaluate the `then` branch if the condition is false, and not to evaluate the `else` branch if the condition is true. Conditional expressions are the only kinds of expressions that provide guaranteed conditions under which a particular subexpression will not be evaluated.

### 2.6 Optional Features

XPath defines a a required level of functionality, called **Basic XPath**, and an optional feature called the **Static Typing Feature**.

#### 2.6.1 Basic XPath

A **Basic XPath** implementation must implement the full XPath language as described in this specification, subject to the following limitations:

1. In a Basic XPath implementation, the **in-scope type definitions** consist only of the built-in types defined in [XML Schema] and the additional predefined types in the `http://www.w3.org/2003/05/xpath-datatypes` namespace.

2. A mapping from a Post-Schema Validation Infoset (PSVI) to the Data Model is specified in [XQuery 1.0 and XPath 2.0 Data Model]. In a Basic XPath implementation, this mapping maps each datatype that is not one of the predefined types listed above into its nearest supertype that belongs to this list. As a result of this mapping, all complex types are mapped into `xs:anyType`. (Of course, mapping from a PSVI is only one way in which a Data Model instance might be constructed—other ways are also possible.)

3. If any **SequenceType** contains a typename that is not one of the predefined types listed above, a Basic XPath implementation raises a static error.

4. If any **SequenceType** contains an **ElementTest** or **AttributeTest** that contains a **TypeName** or a **SchemaContextPath**, a Basic XPath implementation raises a static error.

5. If the processing of an expression depends on the type of some value, and that type is not one of the predefined types listed above, a Basic XPath implementation raises a dynamic error.

6. A Basic XPath implementation is not required to raise type errors during the analysis phase. If an expression contains one or more non-type-related static errors, then a Basic XPath implementation must raise at least one of these static errors during the analysis phase. If the analysis phase is successful but one or more dynamic errors are encountered during the evaluation phase, then a Basic XPath implementation must raise at least one of these dynamic errors.
2.6.2 Static Typing Feature

The **Static Typing Feature** removes the limitation specified by Rule 6 of Basic XPath. An implementation that includes this feature is required to detect type errors during the analysis phase. If an expression contains one or more static errors or type errors, then a Static Typing implementation must raise at least one of these errors during the analysis phase.

3 Expressions

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

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

The highest-level (goal) symbol in the XPath grammar is `XPath`.

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

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

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

3.1 Primary Expressions

**Primary expressions** are the basic primitives of the language. They include literals, variables, function calls, constructors, and the use of parentheses to control precedence of operators.

A **literal** is a direct syntactic representation of an atomic value. XPath supports two kinds of literals: numeric literals and string literals.
The value of a numeric literal containing no "." and no e or E character is an atomic value 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 an atomic value whose 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 an atomic value whose type is `xs:double` and whose value is obtained by parsing the numeric literal according to the rules of the `xs:double` datatype.

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

If a string literal is used in an XPath expression contained within the value of an XML attribute, the characters used to delimit the literal should be different from the characters that are used to delimit the attribute.

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.
- "He said, "I don’t like it."" denotes a string containing two quotation marks and one apostrophe.

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

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]. In general, the name of a constructor function for a given type is the same as the name of the type (including its namespace). For example:

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

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

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

### 3.1.2 Variables

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

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

1. A variable may be added to the in-scope variables by the host language environment.
2. A variable may be bound by an XPath expression. The kinds of expressions that can bind variables are for expressions (**3.7 For Expressions**) and quantified expressions (**3.9 Quantified Expressions**).

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

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

### 3.1.3 Parenthesized Expressions

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

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

### 3.1.4 Function Calls

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

If the expanded QName and number of arguments in a function call do not match the name and arity of an **in-scope function** in the **static context**, an error is raised (the host language environment may define this error as either a static or a dynamic error.)

A function call is evaluated as follows:

1. Each argument expression is evaluated, producing an argument value. The order of argument evaluation is implementation-dependent and a function need not evaluate an argument if the function can evaluate its body without evaluating that argument.
2. Each argument value is converted by applying the function conversion rules listed below.
3. The function is executed using the converted argument values. The result is a value of the function’s declared return type.

The **function conversion rules** are used to convert an argument value to its expected type; that is, to the declared type of the function parameter. The expected type is expressed as a **SequenceType**. The function conversion rules are applied to a given value as follows:
If XPath 1.0 compatibility mode is true, then one of the following conversions is applied:

1. If the expected type is xs:string or xs:string?, then the given value \( v \) is effectively replaced by \( \text{fn:string(fn:subsequence}(V, 1, 1)) \).

2. If the expected type is xs:double or xs:double?, then the given value \( v \) is effectively replaced by \( \text{fn:number(fn:subsequence}(V, 1, 1)) \).

3. If the expected type is a (possibly optional) node or item, then the given value \( v \) is effectively replaced by \( \text{fn:subsequence}(V, 1, 1) \).

4. Otherwise, the given value is unchanged.

If the expected type is a sequence of an atomic type (possibly with an occurrence indicator *, +, or ?), the following conversions are applied:

1. Atomization is applied to the given value, resulting in a sequence of atomic values.

2. Each item in the atomic sequence that is of type xdt:untypedAtomic is cast to the expected atomic type.

3. For each numeric item in the atomic sequence that can be promoted to the expected atomic type using the promotion rules in B.1 Type Promotion, the promotion is done.

If, after the above conversions, the resulting value does not match the expected type according to the rules for SequenceType Matching, a type error is raised. Note that the rules for SequenceType Matching permit a value of a derived type to be substituted for a value of its base type.

A core library of functions is defined in [XQuery 1.0 and XPath 2.0 Functions and Operators]. Additional functions may be provided in the static context.

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

- 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.
- one-argument-function(()) denotes a function call with one argument that is an empty sequence.
- zero-argument-function() denotes a function call with zero arguments.

3.1.5 XPath Comments

XPath comments can be used to provide informative annotation. These comments are lexical constructs only, and do not affect the processing of an expression. Comments are delimited by the symbols (:) and (:). Comments may be nested.

Comments may be used anywhere that ignorable whitespace is allowed. See A.2 Lexical structure for the exact lexical states where comments are recognized.

The following is an example of a comment:
3.2 Path Expressions

A path expression can be used to locate nodes within a tree.

A path expression consists of a series of one or more steps, separated by "/" or "//", and optionally beginning with "/" or "//". An initial "/" or "//" is an abbreviation for one or more initial steps that are implicitly added to the beginning of the path expression, as described below.

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

Each occurrence of // in a path expression is expanded as described in 3.2.4 Abbreviated Syntax, leaving a sequence of steps separated by /. This sequence of steps is then evaluated from left to right. Each operation E1/E2 is evaluated as follows: Expression E1 is evaluated, and if the result is not a sequence of nodes, a dynamic error is raised. Each node resulting from the evaluation of E1 then serves in turn to provide an inner focus for an evaluation of E2, as described in 2.1.2 Evaluation Context. Each evaluation of E2 must result in a sequence of nodes; otherwise, a dynamic error is raised. The sequences of nodes resulting from all the evaluations of E2 are merged, eliminating duplicate nodes based on node identity and sorting the results in document order.

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

A "/" at the beginning of a path expression is an abbreviation for the initial step fn:root(self::node()). The effect of this initial step is to begin the path at the root node of the tree that contains the context node. If the context item is not a node, a type error is raised.

A "//" at the beginning of a path expression is an abbreviation for the initial steps fn:root(self::node())/descendant-or-self::node(). The effect of these initial steps is to establish an initial node sequence that contains all nodes in the same tree as the context node. This node sequence is then filtered by subsequent steps in the path expression. If the context item is not a node, a dynamic error is raised.

3.2.1 Steps

A step generates a sequence of items and then filters the sequence by zero or more predicates. The value of the step consists of those items that satisfy the predicates. Predicates are described in 3.2.2 Predicates. XPath provides two kinds of step, called a filter step and an axis step.

A filter step consists simply of a primary expression followed by zero or more predicates. The result of the filter expression consists of all the items returned by the primary expression for which all the predicates are true. If no predicates are specified, the result is simply the result of the primary expression. This result may contain nodes, atomic values, or any combination of these. The ordering of the items returned by a filter step is the same as their order in the result of the primary expression.

The result of an axis step is always a sequence of zero or more nodes, and these nodes are always returned in document order. An axis step may be either a forward step or a reverse step, followed by zero or more
predicates. An axis step might be thought of as beginning at the context node and navigating to those nodes that are reachable from the context node via a specified axis. Such a step has two parts: an axis, which defines the "direction of movement" for the step, and a node test, which selects nodes based on their kind, name, and/or type.

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

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

3.2.1.1 Axes

[51] ForwardAxis ::= "child" :::
    "descendant" :::
    "attribute" :::
    "self" :::
    "descendant-or-self" :::
    "following-sibling" :::
    "following" :::
    "namespace" :::

[52] ReverseAxis ::= "parent" :::
    "ancestor" :::
    "preceding-sibling" :::
    "preceding" :::
    "ancestor-or-self" :::

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

• the child axis contains the children of the context node

• the descendant axis contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes

• the parent axis contains the parent of the context node, if there is one

• the ancestor axis contains the ancestors of the context node; the ancestors of the context node consist of the parent of context node and the parent’s parent and so on; thus, the ancestor axis will always include the root node, unless the context node is the root node

• the following-sibling axis contains all the following siblings of the context node; if the context node is an attribute node or namespace node, the following-sibling axis is empty

• the preceding-sibling axis contains all the preceding siblings of the context node; if the context node is an attribute node or namespace node, the preceding-sibling axis is empty

• the following axis contains all nodes, in the same tree as the context node, that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes

• the preceding axis contains all nodes, in the same tree as the context node, that are before the context node in document order, excluding any ancestors and excluding attribute nodes and namespace nodes

• the attribute axis contains the attributes of the context node; the axis will be empty unless the context node is an element

• the self axis contains just the context node itself
the descendant-or-self axis contains the context node and the descendants of the context node

the ancestor-or-self axis contains the context node and the ancestors of the context node; thus, the ancestor-or-self axis will always include the root node

the namespace axis contains the namespace nodes of the context node; this axis is empty unless the context node is an element node. The namespace axis is deprecated in XPath 2.0. Whether an implementation supports the namespace axis is implementation-defined. An implementation that does not support the namespace axis must raise a static error if it is used. Applications needing information about the namespaces of an element should use the functions fn:get-in-scope-namespaces and fn:get-namespace-uri-for-prefix defined in [XQuery 1.0 and XPath 2.0 Functions and Operators].

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 XPath, the parent, ancestor, ancestor-or-self, preceding, and preceding-sibling axes are reverse axes; all other axes are forward axes. The ancestor, descendant, following, preceding and self axes partition a document (ignoring attribute and namespace nodes): they do not overlap and together they contain all the nodes in the document.

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.

3.2.1.2 Node Tests

A node test is a condition that must be true for each node selected by a step. The condition may be based on the kind of the node (element, attribute, text, document, comment, processing instruction, or namespace), the name of the node, or (in the case of element and attribute nodes), the type annotation of the 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 consists of a QName is called a name test. A name test is true if and only if the kind of the node is the principal node kind and the expanded-QName of the node is equal to the expanded-QName specified by the name test. For example, child::para selects the para element children of the context node; if the context node has no para children, it selects an empty set of nodes. attribute::abc:href selects the attribute of the context node with the QName abc:href; if the context node has no such attribute, it selects an empty set of nodes.

A QName in a name test is expanded into an expanded-QName using the in-scope namespaces in the expression context. It is a static error if the QName has a prefix that does not correspond to any in-scope namespace. An unprefixed QName, when used as a name test on an axis whose principal node kind is element, has the namespaceURI of the default element namespace in the expression context; otherwise, it has no namespaceURI.

A name test is not satisfied by an element node whose name does not match the QName of the name test, even if it is in a substitution group whose head is the named element.
A node test * is true for any node of the principal node kind. 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 in-scope namespaces in the static context. If the prefix is not found in the in-scope namespaces, a static error is raised. The node test is true for any node of the principal node kind whose expanded-QName has the namespace URI to which the prefix is bound, regardless of the local part of the name.

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

An alternative form of a node test is called a KindTest, which can select nodes based on their kind, name, and type annotation. The syntax and semantics of a KindTest are described in 2.4.3 SequenceType. When a KindTest is used in a node test, only those nodes on the designated axis that match the KindTest are selected. Shown below are several examples of KindTests that might be used in path expressions:

- node() matches any node.
- text() matches any text node.
- comment() matches any comment node.
- element() matches any element node.
- element(person) matches any element node whose name is person (or is in the substitution group headed by person), and whose type annotation conforms to the top-level schema declaration for a person element.
- element(person, *) matches any element node whose name is person (or is in the substitution group headed by person), without any restriction on type annotation.
- element(person, surgeon) matches any element node whose name is person (or is in the substitution group headed by person), and whose type annotation is surgeon.
- element(*, surgeon) matches any element node whose type annotation is surgeon, regardless of its name.
- element(hospital/staff/person) matches any element node whose name and type annotation conform to the schema declaration of a person element in a staff element in a top-level hospital element.
- attribute() matches any attribute node.
- attribute(@price, *) matches any attribute whose name is price, regardless of its type annotation.
- attribute(*, xs:decimal) matches any attribute whose type annotation is xs:decimal, regardless of its name.
- document-node() matches any document node.
- document-node(element(book)) matches any document node whose content consists of a single element node that satisfies the KindTest element(book).

3.2.2 Predicates

A predicate consists of an expression, called a predicate expression, enclosed in square brackets. A predicate serves to filter a sequence, retaining some items and discarding others. For each item in the sequence to be filtered, the predicate expression is evaluated using an inner focus derived from that item, as described in 2.1.2 Evaluation Context. The result of the predicate expression is coerced to a Boolean value, called the predicate truth value, as described below. Those items for which the predicate truth value is true are retained, and those for which the predicate truth value is false are discarded.
The predicate truth value is derived by applying the following rules, in order:

1. If the value of the predicate expression is an atomic value of a numeric type, the predicate truth value is true if the value of the predicate expression is equal to the context position, and is false otherwise.

2. Otherwise, the predicate truth value is the Effective Boolean Value of the predicate expression.

Here are some examples of axis steps that contain predicates:

- This example selects the second chapter element that is a child of the context node:
  
  ```
  child::chapter[2]
  ```

- This example selects all the descendants of the context node 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]
  ```

Here are some examples of filter steps that contain predicates:

- List all the integers from 1 to 100 that are divisible by 5. (See 3.3.1 Constructing Sequences for an explanation of the to operator.)
  
  ```
  {1 to 100}[. mod 5 eq 0]
  ```

- The result of the following expression is the integer 95:
  
  ```
  {99 to 0}[5]
  ```

### 3.2.3 Unabbreviated Syntax

This section provides a number of examples of path expressions in which the axis is explicitly specified in each step. The syntax used in these examples is called the unabbreviated syntax. In many common cases, it is possible to write path expressions more concisely using an abbreviated syntax, as explained in 3.2.4 Abbreviated Syntax.

- child::para selects the para element children of the context node
- child::* selects all element children of the context node
- child::text() selects all text node children of the context node
- child::node() selects all the children of the context node, whatever their node type
- attribute::name selects the name attribute of the context node
- attribute::* selects all the attributes of the context node
- parent::* selects the parent of the context node. If the context node is an attribute node, this expression returns the element node (if any) to which the attribute node is attached.
- descendant::para selects the para element descendants of the context node
- ancestor::div selects all div ancestors of the context node
- ancestor-or-self::div selects the div ancestors of the context node and, if the context node is a div
element, the context node as well

- descendant-or-self::para selects the para element descendants of the context node and, if the context node is a para element, the context node as well
- self::para selects the context node if it is a para element, and otherwise 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 root of the node hierarchy that contains 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 a list parent and that are in the same document as the context node
- child::para[fn:position() = 1] selects the first para child of the context node
- child::para[fn:position() = fn:last()] selects the last para child of the context node
- child::para[fn:position() = fn:last()-1] selects the last but one para child of the context node
- child::para[fn:position() > 1] selects all the para children of the context node other than the first para child of the context node
- following-sibling::chapter[fn:position() = 1] selects the next chapter sibling of the context node
- preceding-sibling::chapter[fn:position() = 1] selects the previous chapter sibling of the context node
- /descendant::figure[fn:position() = 42] selects the forty-second figure element in the document
- /child::doc/child::chapter[fn:position() = 5]/child::section[fn:position() = 2] selects the second section of the fifth chapter of the doc 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'][fn:position() = 5] selects the fifth para child of the context node that has a type attribute with value warning
- child::para[fn:position() = 5][attribute::type="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][fn:position() = fn:last()] selects the last chapter or appendix child of the context node

3.2.4 Abbreviated Syntax

[49] AbbreviatedForwardStep ::= "*" | (*[@NameTest]) | NodeTest
[50] AbbreviatedReverseStep ::= ".."
The abbreviated syntax permits the following abbreviations:

1. The most important abbreviation is that the axis name can be omitted from an **axis step**. If the axis name is omitted from an axis step, the default axis is **child** unless the axis step contains an **AttributeTest**; in that case, the default axis is **attribute**. For example, a path expression `section/para` is an abbreviation for `child::section/child::para`. Similarly, `section/attribute(@id)` is an abbreviation for `child::section/attribute::attribute(@id)`.

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 effectively replaced by `/descendant-or-self::node()/` during processing of a path expression. For example, `//para` is an abbreviation 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 `.` returns the context item. This is particularly useful in conjunction with the `//` operator. For example, the path expression `./para` returns all `para` descendants 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[fn: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 a `list` parent
- `.` selects the context item
- `./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 `book` element that is a child of the context node.

• `book/fn:id(publisher)/name` returns the same result as `fn:id(book/publisher)/name`.

• If $E$ is any expression that returns a sequence of nodes, then the expression $E/.$ returns the same nodes in document order, with duplicates eliminated based on node identity.

### 3.3 Sequence Expressions

XPath supports operators to construct 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).

#### 3.3.1 Constructing Sequences

[16] $Expr ::= ExprSingle ("," ExprSingle)*$

[29] $RangeExpr ::= AdditiveExpr ("to" AdditiveExpr)?$

One way to construct a sequence is by using the comma operator, which evaluates each of its operands and concatenates the resulting values, in order, into a single result sequence. Empty parentheses can be used to denote an empty sequence. In places where the grammar calls for `ExprSingle`, such as the arguments of a function call, any expression that contains a top-level comma operator must be enclosed in parentheses.

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:
This expression contains all salary children of the context node followed by all bonus children:

\( \text{salary, bonus} \)

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. Each of the operands of the `to` operator is converted as though it was an argument of a function with the expected parameter type `xs:integer` (this process raises an error if the operand cannot be converted to a single 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 as one operand in constructing a sequence:

\( 10, 1 \text{ to } 4 \)

It evaluates to the sequence:

10, 1, 2, 3, 4

This example constructs a sequence of length one:

\( 10 \text{ to } 10 \)

It evaluates to a sequence consisting of the single integer 10.

### 3.3.2 Combining Sequences

**UnionExpr**

\[ \text{UnionExpr} ::= \text{IntersectExceptExpr} \ (\text{"∪" | "|"}) \text{IntersectExceptExpr} \star \]

**IntersectExceptExpr**

\[ \text{IntersectExceptExpr} ::= \text{ValueExpr} \ (\text{"∩" | "except"}) \text{ValueExpr} \star \]

**ValueExpr**

\[ \text{ValueExpr} ::= \text{PathExpr} \]

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

Here are some examples of expressions that combine sequences. Assume the existence of three element nodes that we will refer to by symbolic names A, B, and C. Assume that \( \$seq1 \) 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:

- \( \$seq1 \text{ union } \$seq1 \) evaluates to a sequence containing A and B.
- \( \$seq2 \text{ union } \$seq3 \) evaluates to a sequence containing A, B, and C.
- \( \$seq1 \text{ intersect } \$seq1 \) evaluates to a sequence containing A and B.
- \( \$seq2 \text{ intersect } \$seq3 \) 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.

3.4 Arithmetic Expressions

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

The binary subtraction operator must be preceded by whitespace if it could otherwise be interpreted as part of the previous token. For example, a-b will be interpreted as a name, but a - b will be interpreted as an arithmetic operation.

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

1. Atomization is applied to each operand.

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

3. If either operand is now a sequence of length greater than one, then:

   1. If XPath 1.0 compatibility mode is true, any items after the first item in the sequence are discarded.

   2. Otherwise, a type error is raised.

4. If either operand is now of type xdt:untypedAtomic, it is cast to the default type for the given operator. The default type for the idiv operator is xs:integer; the default type for all other arithmetic operators is xs:double. If the cast fails, a type error is raised.

5. If the operand types are now valid for the given operator, the operator is applied to the operands, resulting in an atomic value or a dynamic error (for example, an error might result from dividing by zero.) The combinations of atomic types that are accepted by the various arithmetic operators, and their respective result types, are listed in B.2 Operator Mapping together with the functions in [XQuery 1.0 and XPath 2.0 Functions and Operators] that define the semantics of the operation for each type.

6. If the operand types are not valid for the given operator, and XPath 1.0 compatibility mode is true, and the operator is not idiv, then each operand is further converted according to the rules in 3.1.4 Function Calls as if it were a function argument with the expected type xs:double. The operator is then applied to the operands, resulting in an atomic value or a dynamic error.

7. If the operand types are still not valid for the given operator, a type error is raised.

XPath supports two division operators named div and idiv. The div operator accepts operands of any numeric types. The type of the result of the div operator is the least common type of its operands; however, if both operands are of type xs:integer, div returns a result of type xs:decimal. The idiv operator, on the other hand, requires its operands to be of type xs:integer and returns a result of type xs:integer, rounded toward zero.

Here are some examples of arithmetic expressions:

• The first expression below returns -1.5, and the second expressions returns -1:
-3 div 2
-3 idiv 2

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

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

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

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

- Unary operators have higher precedence than binary operators, subject of course to the use of parentheses:

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

### 3.5 Comparison Expressions

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

```xml
ComparisonExpr ::= RangeExpr { (ValueComp | GeneralComp | NodeComp | OrderComp) RangeExpr }?
```

```xml
ValueComp ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
```

```xml
GeneralComp ::= "=" | "+=" | "+<" | "+<=" | "+>" | "+=>"
```

```xml
NodeComp ::= "is" | "isnot"
```

```xml
OrderComp ::= "<<" | ">>"
```

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

### 3.5.1 Value Comparisons

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. If the result, called an atomized operand, does not contain exactly one atomic value, a type error is raised.

2. Any atomized operand that has the dynamic type `xdt:untypedAtomic` is cast to the type `xs:string`.

3. 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 atomized operand is not comparable with the value of the second atomized operand, a type error 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":

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

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

General comparisons are existentially quantified comparisons that may be applied to operand sequences of any length. The result of a general comparison that does not raise an error is always true or false.

Atomization is applied to each operand of a general comparison. The result of the comparison is true if and only if there is a pair of atomic values, one belonging to the result of atomization of the first operand and the other belonging to the result of atomization of the second operand, that have the required magnitude relationship. Otherwise the result of the general comparison is false. The magnitude relationship between two atomic values is determined as follows:

1. If either atomic value has the dynamic type xdt:untypedAtomic, that value is cast to a required type, which is determined as follows:
   1. If the dynamic type of the other atomic value is a numeric type, the required type is xs:double.
   2. If the dynamic type of the other atomic value is xdt:untypedAtomic, the required type is xs:string.
   3. Otherwise, the required type is the dynamic type of the other atomic value.

2. If XPath 1.0 compatibility mode is true, and at least one of the atomic values has a numeric type, then both atomic values are cast to the type xs:double.

3. After any necessary casting, the atomic values are compared using one of the value comparison operators eq, ne, lt, le, gt, or ge, depending on whether the general comparison operator was =, !=, <, <=, >, or >=. The values have the required magnitude relationship if the result of this value comparison is true.

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

Here are some examples of general comparisons:

- The following comparison is true if the value of any author subelement of $book1 has the string value "Kennedy":

  $book1/author = "Kennedy"

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

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

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

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

1. Each operand must be either a single node or an empty sequence; otherwise a type error is raised.

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:

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

### 3.5.4 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 a type error is raised.

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; otherwise it returns `false`.

4. A comparison with the `>>` operator returns `true` if the first operand node is later than the second operand node in document order; 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:

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

### 3.6 Logical Expressions

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

#### Logical Expressions

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

### Logical Expressions

1. **and-expression**

   ```
   OrExpr ::= AndExpr (*or* AndExpr )*
   
   AndExpr ::= InstanceofExpr (*and* InstanceofExpr )*
   ```

The first step in evaluating a logical expression is to find the effective boolean value of each of its operands (see 2.4.4.2 Effective Boolean Value).

The value of an and-expression is determined by the effective boolean values (EBV’s) of its operands. If an error is raised during computation of one of the effective boolean values, an and-expression may raise a dynamic error, as shown in the following table:

<table>
<thead>
<tr>
<th>AND:</th>
<th>EBV₂ = true</th>
<th>EBV₂ = false</th>
<th>error in EBV₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBV₁ = true</td>
<td>true</td>
<td>false</td>
<td>error</td>
</tr>
</tbody>
</table>
The value of an or-expression is determined by the effective boolean values (EBV’s) of its operands. If an error is raised during computation of one of the effective boolean values, an or-expression may raise a dynamic error, as shown in the following table:

<table>
<thead>
<tr>
<th>OR:</th>
<th>EBV₂ = true</th>
<th>EBV₂ = false</th>
<th>error in EBV₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBV₁ = true</td>
<td>true</td>
<td>true</td>
<td>true or error</td>
</tr>
<tr>
<td>EBV₁ = false</td>
<td>true</td>
<td>false</td>
<td>error</td>
</tr>
<tr>
<td>error in EBV₁</td>
<td>true or error</td>
<td>error</td>
<td>error</td>
</tr>
</tbody>
</table>

The order in which the operands of a logical expression are evaluated is implementation-dependent. The tables above are defined in such a way that an or-expression can return `true` if the first expression evaluated is true, and it can raise an error if evaluation of the first expression raises an error. Similarly, an and-expression can return `false` if the first expression evaluated is false, and it can raise an error if evaluation of the first expression raises an error. As a result of these rules, a logical expression is not deterministic in the presence of errors, as illustrated in the examples below.

Here are some examples of logical expressions:

- The following expressions return `true`:
  
  
  ```
  1 eq 1 and 2 eq 2
  1 eq 1 or 2 eq 3
  ```

- The following expression may return either `false` or raise a dynamic error:
  
  ```
  1 eq 2 and 3 idiv 0 = 1
  ```

- The following expression may return either `true` or raise a dynamic error:
  
  ```
  1 eq 1 or 3 idiv 0 = 1
  ```

- The following expression must raise a dynamic error:
  
  ```
  1 eq 1 and 3 idiv 0 = 1
  ```

In addition to and- and or-expressions, XPath 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. It then returns `true` if the effective boolean value of its parameter is `false`, and `false` if the effective boolean value of its parameter is `true`. If an error is encountered in finding the effective boolean value of its operand, `not` raises a dynamic error. The `not` function is described in [XQuery 1.0 and XPath 2.0 Functions and Operators].

### 3.7 For Expressions

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

ForExpr ::= SimpleForClause "return" ExprSingle  

SimpleForClause ::= "for" *"&" VarName "in" ExprSingle ("," "&" VarName "in" ExprSingle)*
A **for** expression is evaluated as follows:

1. If the **for** expression uses multiple variables, it is first expanded to a set of nested **for** expressions, each of which uses only one variable. For example, the expression `for $x$ in X, $y$ in Y return $x + y` is expanded to `for $x$ in X return for $y$ in Y return $x + y`.

2. In a single-variable **for** expression, the variable is called the **range variable**, the value of the expression that follows the **in** keyword is called the **input sequence**, and the expression that follows the **return** keyword is called the **return expression**. The result of the **for** expression is obtained by evaluating the **return expression** once for each item in the input sequence, with the range variable bound to that item. The resulting sequences are concatenated in the order of the items in the input sequence from which they were derived.

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

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

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

```xml
for $a$ in distinct-values(//author)
  return ($a,
    for $b$ in //book[author = $a$]
      return $b/title)
```

The result of the above expression consists of the following sequence of elements. The ordering of **author** elements in the result is implementation-dependent.

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

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

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

The scope of a variable bound in a **for** expression comprises all subexpressions of the **for** expression that appear after the variable binding. The scope does not include the expression to which the variable is bound. The following example illustrates how a variable binding may reference another variable bound earlier in the same **for** expression:
Note that the focus for evaluation of the `return` clause of a `for` expression is the same as the focus for evaluation of the `for` expression itself. The following example, which attempts to find the total value of a set of order-items, is therefore incorrect:

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

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

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

### 3.8 Conditional Expressions

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

**Conditional Expression**

```[
ifExpr ::= "if" "(" Expr ")" "then" Expr "else" ExprSingle
```

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

The first step in processing a conditional expression is to find the **effective boolean value** of the test expression, as defined in [2.4.4.2 Effective Boolean Value](#).

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

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

Here are some examples of conditional expressions:

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

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

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

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

### 3.9 Quantified Expressions

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

```[
QuantifiedExpr ::= ("some" "$" | "every" "$") VarName "in" ExprSingle | "(" VarName "=" VarName "in"
```
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. 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 2.4.4.2 Effective Boolean Value. The value of the quantified expression is defined by the following rules:

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

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

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

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

Here are some examples of quantified expressions:

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

  ```
  every $part in //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 either return `true` or raise a type error, since its test expression returns `true` for one variable binding and raises a type error for another:

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

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

SequenceTypes are used in instance of, cast, castable, and treat expressions.

3.10.1 Instance Of

\[24\]  

\[
\text{InstanceOfExpr ::= \text{TreatExpr \{ "instance" \ "of" \ SequenceType \}}?}
\]

The boolean operator \text{instance of} returns true if the value of its first operand matches the type named in its second operand, according to the rules for \text{SequenceType Matching}; otherwise it returns false. For example:

- \text{5 instance of xs:integer}
  
  This example returns true because the given value is an instance of the given type.

- \text{5 instance of xs:decimal}
  
  This example returns true because the given value is an integer literal, and \text{xs:integer} is derived by restriction from \text{xs:decimal}.

- \text{. instance of element()}
  
  This example returns true if the context item is an element node.

3.10.2 Cast

\[27\]  

\[
\text{CastExpr ::= \text{ComparisonExpr \{ "cast" \ "as" \ SingleType \}}?}
\]

\[60\]  

\[
\text{SingleType ::= \text{AtomicType ??}}
\]

Occasionally it is necessary to convert a value to a specific datatype. For this purpose, XPath provides a cast expression that creates a new value of a specific type based on an existing value. A cast expression takes two operands: an input expression and a target type. The type of the input expression is called the input type. The target type must be a named atomic type, represented by a QName, optionally followed by the occurrence indicator ? if an empty sequence is permitted. If the target type has no namespace prefix, it is considered to be in the default element namespace. The semantics of the cast expression are as follows:

1. Atomization is performed on the input expression.

2. If the result of atomization is a sequence of more than one atomic value, a type error is raised.

3. If the result of atomization is an empty sequence:
   1. If ? is specified after the target type, the result of the cast expression is an empty sequence.
   2. If ? is not specified after the target type, a type error is raised.

4. If the result of atomization is a single atomic value, the result of the cast expression depends on the input type and the target type. In general, the cast expression attempts to create a new value of the target type based on the input value. Only certain combinations of input type and target type are supported. The rules are listed below. For the purpose of these rules, we use the terms subtype and supertype in the following sense: if type B is derived from type A by restriction, then B is a subtype of A, and A is a supertype of B.

1. 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 built-in schema types. For example, a value of type \text{xs:string} can be cast into the type \text{xs:decimal}. For each of these built-in combinations, the semantics of casting are specified in [XQuery 1.0 and XPath 2.0 Functions and Operators].
2. `cast` is supported if the input type is a derived atomic type and the target type is a supertype of the input type. In this case, the input value is mapped into the value space of the target type, unchanged except for its type. For example, if `shoesize` is derived by restriction from `xs:integer`, a value of type `shoesize` can be cast into the type `xs:integer`.

3. `cast` is supported if the target type is a derived atomic type and the input type is `xs:string` or `xdt:untypedAtomic`. The input value is first converted to a value in the lexical space of the target type by applying the whitespace normalization rules for the target type; a dynamic error is raised if the resulting lexical value does not satisfy the pattern facet of the target type. The lexical value is then converted to the value space of the target type using the schema-defined rules for the target type; a dynamic error is raised if the resulting value does not satisfy all the facets of the target type.

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

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

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

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

### 3.10.3 Castable

[XPath provides a form of Boolean expression that tests whether a given value is castable into a given target type. The expression `V castable as T` returns `true` if the value `V` can be successfully cast into the target type `T` by using a `cast` expression; otherwise it returns `false`. The `castable` predicate can be used to avoid errors at evaluation time. It can also be used to select an appropriate type for processing of a given value, as illustrated in the following example:]

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

### 3.10.4 Constructor Functions

Constructor functions provide an alternative syntax for casting.

For every built-in atomic type `T` that is defined in [XML Schema], as well as the predefined types `xdt:dayTimeDuration`, `xdt:yearMonthDuration`, and `xdt:untypedAtomic`, a built-in constructor function is provided. The signature of the built-in constructor function for type `T` is as follows:

```
T($x as item) as T
```

The constructor function for type `T` accepts any single item (either a node or an atomic value) as input, and returns a value of type `T` (or raises a dynamic error). Its semantics are exactly the same as a `cast` expression with target type `T`. The built-in constructor functions are described in more detail in [XQuery 1.0 and XPath 2.0 Functions and Operators]. The following are examples of built-in constructor functions:
• This example is equivalent to "2000-01-01" cast as xs:date.

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

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

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

• This example returns a dayTimeDuration value equal to 21 days. It is equivalent to "P21D" cast as xdt:dayTimeDuration.

    xdt:dayTimeDuration("P21D")

For each user-defined top-level atomic type $T$ in the in-scope type definitions, a constructor function is effectively defined. Like the built-in constructor functions, the constructor functions for user-defined types have the same name (including namespace or lack of namespace) as the type, accept any item as input, and have semantics identical to a cast expression with the user-defined type as target type. For example, if usa:zipcode is a user-defined top-level atomic type in the in-scope type definitions, then the expression usa:zipcode("12345") is equivalent to the expression "12345" cast as usa:zipcode.

If the argument to any constructor function is a literal value, the result of the function may be computed statically, and an error encountered in this process may be reported as a static error.

### 3.10.5 Treat

$$\text{TreatExpr ::= CastableExpr \ ("treat" \ "as" \ SequenceType) ?}$$

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

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

The semantics of expr1 treat as type1 are as follows:

• During static analysis (if the Static Typing Feature is implemented):

  type1 must be derived by restriction from the static type of expr1 -- otherwise, a type 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 expr1 matches type1, using the SequenceType Matching rules in 2.4.3 SequenceType, the treat expression returns the value of expr1; otherwise, it raises a dynamic error. If the value of expr1 is returned, its identity is preserved. The treat expression ensures that the value of its expression operand conforms to the expected type at run-time.

• Example:

        $myaddress treat as element(*, USAddress)

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

### A XPath Grammar

#### A.1 EBNF
The following grammar uses the same Basic Extended Backus-Naur Form (EBNF) notation as [XML], except that grammar symbols always have initial capital letters. The notation "< ... >" is used to indicate a grouping of terminals that together may help disambiguate the individual symbols. To help readability, this "< ... >" notation is absent in the EBNF in the main body of this document.

Comments on grammar productions are between '/*' and '*/' symbols. A 'pn:' prefix means a 'Parser Note', and are meant as clarifications for parsing rules, and are explained in [A.1.1 Parsing Notes]. A 'ws:' prefix explains the whitespace rules for the production, the details of which are explained in [A.2.1 Whitespace Rules].

**Note:**

The Semicolon character is reserved for future use.

### Named Terminals

1. **ExprComment** ::= "(:* (ExprCommentContent | ExprComment)* ":") /* pn: parens */
2. **ExprCommentContent** ::= Char
3. **IntegerLiteral** ::= Digits
4. **DecimalLiteral** ::= ("*" Digits | Digits "." [0-9]*) /* ws: explicit */
5. **DoubleLiteral** ::= ("*" Digits | Digits ("." [0-9]*)) ("e" | "E") ("+" | "-")? Digits /* ws: explicit */
6. **StringLiteral** ::= ("" {"" ";" | [^"\]]} ";") | ("" {"" ";" | [^"\]]} ";") /* ws: significant */
7. **SchemaGlobalTypeName** ::= "type" "(" QName ")"  
8. **SchemaGlobalContext** ::= QName | SchemaGlobalTypeName  
9. **SchemaContextStep** ::= QName  
10. **Digits** ::= [0-9]+  
11. **NCName** ::= [http://www.w3.org/TR/REC-xml-names/#NT-NCName]  
12. **VarName** ::= QName  
13. **QName** ::= [http://www.w3.org/TR/REC-xml-names/#NT-QName]  
14. **Char** ::= [http://www.w3.org/TR/REC-xml#NT-Char]  

### Non-Terminals

44 of 135 16.06.2003 09:57
XPath ::= Expr
Expr ::= ExprSingle ("*", ExprSingle) |
ExprSingle ::= ForExpr |
ForExpr ::= SimpleForClause "return" ExprSingle |
SimpleForClause ::= <"for" "$"> VarName "in" ExprSingle ("*", "@") VarName "in" ExprSingle |
QuantifiedExpr ::= (<"some" "$"> | <"every" "$">) VarName "in" ExprSingle ("*", "@" VarName "in" ExprSingle) |
IfExpr ::= <"if" "(" Expr ")" "then" Expr "else" ExprSingle |
OrExpr ::= AndExpr ("or" AndExpr) |
AndExpr ::= InstanceofExpr ("and" InstanceofExpr) |
InstanceofExpr ::= TreatExpr (<"instance" "of"> SequenceType) |
TreatExpr ::= CastableExpr (<"treat" "as"> SingleType) |
CastExpr ::= ComparisonExpr (<"cast" "as"> SingleType) |
ComparisonExpr ::= RangeExpr (ValueComp |
RangeExpr ::= AdditiveExpr ("to" AdditiveExpr) |
AdditiveExpr ::= MultiplicativeExpr ("*" |
MultiplicativeExpr ::= UnaryExpr ("*" | 
UnaryExpr ::= ("-" | 
UnionExpr ::= IntersectExceptExpr (<"union" | 
IntersectExceptExpr ::= ValueExpr (<"intersect" | 
ValueExpr ::= PathExpr |
PathExpr ::= StepExpr ("//" RelativePathExpr) |
RelativePathExpr ::= StepExpr ("/*" |
StepExpr ::= AxisStep | FilterStep |
AxisStep ::= (ForwardStep | ReverseStep) Predicates |
FilterStep ::= PrimaryExpr Predicates |
PrimaryExpr ::= Literal | FunctionCall | (<"@" VarName) | ParenthesizedExpr |
Predicates ::= (<"Expr ">) |
GeneralComp ::= 
ValueComp ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge" |
NodeComp ::= "is" | "isnot" |
OrderComp ::= "<" | "="> |
ForwardStep ::= (ForwardAxis NodeTest) | AbbreviatedForwardStep |
ReverseStep ::= (ReverseAxis NodeTest) | AbbreviatedReverseStep |
AbbreviatedForwardStep ::= 
AbbreviatedReverseStep ::= 
ForwardAxis ::= <"child" "> |
ReverseAxis ::= <"parent" "> |

A.1.1 Parsing Notes

1. A look-ahead of one character is required to distinguish function patterns from a QName followed by a comment. For example: `address (: this may be empty :)` may be mistaken for a call to a function named "address" unless this lookahead is employed.

2. Token disambiguation of the overloaded "<" pattern is defined in terms of positional lexical states. The "<" comparison operator can not occur in the same places as a "<" tag open pattern. The "<" comparison operator can only occur in the OPERATOR state, and the "<" tag open pattern can only occur in the DEFAULT state. (These states are only a specification tool, and do not imply an implementation strategy for this same effect.)

A.2 Lexical structure

Legal characters are those allowed in the [XML] recommendation.

When patterns are simple string matches, the strings are embedded directly into the EBNF. In other cases, named terminals are used.

It is up to an implementation to decide on the exact tokenization strategy, which may be different depending on the parser construction. In the EBNF, the notation "< ... >" is used to indicate a grouping of terminals that together may help disambiguate the individual symbols.

This document uses lexical states to assist with terminal symbol recognition. The states specify lexical constraints and transitions based on grammatical positioning. The rules for calculating these states are given in the A.2.2 Lexical Rules section. The specification of these states in this document does not imply any tokenization strategy on the part of implementations.
When tokenizing, the longest possible match that is valid in the current lexical state is preferred.

All keywords are case sensitive.

**A.2.1 Whitespace Rules**

For readability, Whitespace may be used in most expressions even though not explicitly notated in the EBNF. Whitespace may be freely added between terminals, except a few cases where whitespace is needed to disambiguate the token. For instance, 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, e.g. by using whitespace or parentheses.

Special whitespace notation is specified with the EBNF productions, when it is different from the default rules, as follows.

**Whitespace: explicit**

"ws: explicit" means that the EBNF notation must explicitly notate where whitespace is allowed, otherwise whitespace may not be freely used.

**Whitespace: significant**

"ws: significant" means that whitespace is significant as value content.

**A.2.2 Lexical Rules**

<table>
<thead>
<tr>
<th>Editorial note</th>
</tr>
</thead>
<tbody>
<tr>
<td>The role of these states as normative parts of the specification is being investigated. One proposal suggests that only a subset of these states need be normative.</td>
</tr>
</tbody>
</table>

The lexical contexts and transitions between lexical contexts is described in terms of a series of states and transitions between those states.

The tables below define the complete lexical rules for XPath. Each table corresponds to a lexical state in which the tokens listed are recognized only in that state. When a given token is recognized in the given state, the transition to the next state is given. In some cases, a transition will "push" the current state or a specific state onto an abstract stack, and will later restore that state by a "pop" when another lexical event occurs.

The lexical states have in many cases close connection to the parser productions. However, just because a token is recognized in a certain lexical state, does not mean it will be legal in the parser state.

**The DEFAULT State**

This state is for patterns that occur at the beginning of an expression or subexpression.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>DecimalLiteral, &quot;..&quot;, &quot;,&quot;, DoubleLiteral, IntegerLiteral, &lt;NCName &quot;:&quot;, &quot;#&quot;&gt;, &quot;[&quot;, &quot;]&quot;, &quot;,&quot;, &lt;&quot;#&quot; : &quot;NCName&quot;&gt;, &quot;#&quot; StringLiteral</td>
<td>OPERATOR</td>
</tr>
<tr>
<td>&quot;$&quot;, &lt;&quot;for&quot; &quot;$&quot;&gt;, &lt;&quot;some&quot; &quot;$&quot;&gt;, &lt;&quot;every&quot; &quot;$&quot;&gt;</td>
<td>VARNAME</td>
</tr>
<tr>
<td>&lt;&quot;element&quot; &quot;(&quot;&gt;, &lt;&quot;attribute&quot; &quot;(&quot;&gt;, &lt;&quot;comment&quot; &quot;(&quot;&gt;, &lt;&quot;text&quot; &quot;(&quot;&gt;, &lt;&quot;node&quot; &quot;(&quot;&gt;, &lt;&quot;processing-instruction&quot; &quot;(&quot;&gt;, &lt;&quot;document-node&quot; &quot;(&quot;</td>
<td>KINDTEST</td>
</tr>
<tr>
<td>ofType</td>
<td>QNAME</td>
</tr>
<tr>
<td>rbrace</td>
<td>popState()</td>
</tr>
<tr>
<td>&quot;(&quot;</td>
<td>EXPR_COMMENT</td>
</tr>
</tbody>
</table>

pushState(OPERATOR)
**The OPERATOR State**

This state is for patterns that are defined for operators.

```
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;and&quot;, &quot;,&quot;, &quot;div&quot;, &quot;else&quot;, &quot;+&quot;, &quot;except&quot;, &quot;eq&quot;, &quot;ge&quot;, &quot;gt&quot;, &quot;le&quot;, &quot;lt&quot;, &quot;ne&quot;, &quot;&gt;=&quot;</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>&quot;&gt;&gt;&quot;, &quot;&gt;&quot;, &quot;+”, Global, &quot;intersect&quot;, &quot;in&quot;, &quot;isnot&quot;, &quot;;is&quot;, &quot;[&quot;, &quot;;=&quot;, &quot;;=&quot;&quot;, &quot;;&lt;&quot;, &quot;;=&quot;&quot;, &quot;;mod&quot;, &quot;**&quot;, &quot;!=&quot;, &quot;;or&quot;, &quot;+&quot;, QMark, &quot;ret\urn&quot;, &quot;satisfies&quot;, &quot;;//&quot;, &quot;;/&quot;, &quot;then&quot;, &quot;;to&quot;, &quot;union&quot;, &quot;;]&quot;, LbraceExprEnclosure</td>
<td></td>
</tr>
<tr>
<td>&lt;&quot;instance&quot; &quot;of&quot;&gt;, &lt;&quot;castable&quot; &quot;as&quot;&gt;, &lt;&quot;cast&quot; &quot;as&quot;&gt;, &lt;&quot;treat&quot; &quot;as&quot;&gt;, InContext</td>
<td>ITEMTYPE</td>
</tr>
<tr>
<td>OfType</td>
<td>QNAME</td>
</tr>
<tr>
<td>Rbrace</td>
<td>popState()</td>
</tr>
<tr>
<td>&quot;$&quot;, &lt;&quot;for&quot; &quot;$&quot;&gt;, &lt;&quot;some&quot; &quot;$&quot;&gt;, &lt;&quot;every&quot; &quot;$&quot;&gt;</td>
<td>VARNAME</td>
</tr>
<tr>
<td>&quot;:;&quot;</td>
<td>EXPR_COMMENT</td>
</tr>
<tr>
<td>&quot;[&quot;, IntegerLiteral, DecimalLiteral, DoubleLiteral, &quot;]&quot;, StringLiteral, &lt;NCName := &quot;#&quot;&gt;, &lt;&quot;#&quot; : NCName&gt;, &quot;*&quot;, &quot;*&quot;</td>
<td>(maintain state)</td>
</tr>
</tbody>
</table>
```

**The QNAME State**

When a qualified name is expected, and it is required to remove ambiguity from patterns that look like functions, this state is used.

```
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;NCName := &quot;#&quot;&gt;, &lt;&quot;#&quot; &quot;NCName&gt;, &quot;*&quot;, &quot;*&quot;</td>
<td>OPERATOR</td>
</tr>
<tr>
<td>&quot;(:&quot;</td>
<td>EXPR_COMMENT</td>
</tr>
<tr>
<td>pushState()</td>
<td></td>
</tr>
</tbody>
</table>
```

**The ITEMTYPE State**

This state distinguishes tokens that can occur only inside the ItemType production.

```
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;$&quot;</td>
<td>VARNAME</td>
</tr>
<tr>
<td>&lt;&quot;empty&quot; &quot;(&quot; &quot;)&quot;&gt;</td>
<td>OPERATOR</td>
</tr>
<tr>
<td>&quot;(:&quot;</td>
<td>EXPR_COMMENT</td>
</tr>
<tr>
<td>pushState()</td>
<td></td>
</tr>
<tr>
<td>&lt;&quot;element&quot; &quot;(&quot;&gt;, &lt;&quot;attribute&quot; &quot;(&quot;&gt;, &lt;&quot;comment&quot; &quot;(&quot;&gt;, &lt;&quot;text&quot; &quot;(&quot;&gt;, &lt;&quot;node&quot; &quot;(&quot;&gt;, &lt;&quot;processing-instruction&quot; &quot;(&quot;&gt;, &lt;&quot;document-node&quot; &quot;(&quot;</td>
<td>KINDTEST</td>
</tr>
<tr>
<td>pushState(OCCURRENCEINDICATOR)</td>
<td></td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>QName, &lt;&quot;item&quot; &quot;(&quot; &quot;)&quot;&gt;</td>
<td>OCCURRENCEINDICATOR</td>
</tr>
</tbody>
</table>

**The KINDTEST State**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>LbraceExprEnclosure</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>&lt;SchemaGlobalContext &quot;/&quot;, SchemaGlobalTypeName</td>
<td>SCHEMACONTEXTSTEP</td>
</tr>
<tr>
<td>&quot;)&quot;</td>
<td>popState()</td>
</tr>
<tr>
<td>&quot;@&quot;, QName</td>
<td>CLOSEKINDTEST</td>
</tr>
<tr>
<td>&quot;nillable&quot;</td>
<td>(maintain state)</td>
</tr>
</tbody>
</table>

**The CLOSEKINDTEST State**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;)&quot;</td>
<td>popState()</td>
</tr>
<tr>
<td>&quot;,,&quot;</td>
<td>KINDTEST</td>
</tr>
<tr>
<td>LbraceExprEnclosure</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>&quot;nillable&quot;</td>
<td>(maintain state)</td>
</tr>
</tbody>
</table>

**The OCCURRENCEINDICATOR State**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotOccurrenceIndicator</td>
<td>OPERATOR</td>
</tr>
<tr>
<td>input_stream.backup(1)</td>
<td></td>
</tr>
<tr>
<td>&quot;?&quot;, &quot;@&quot;, &quot;+&quot;</td>
<td>OPERATOR</td>
</tr>
</tbody>
</table>

**The SCHEMACONTEXTSTEP State**

This state distinguishes the SchemaContextStep from the SchemaGlobalContext.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;SchemaContextStep &quot;/&quot;, &quot;@&quot;</td>
<td>(maintain state)</td>
</tr>
<tr>
<td>LbraceExprEnclosure</td>
<td>DEFAULT</td>
</tr>
<tr>
<td>QName</td>
<td>CLOSEKINDTEST</td>
</tr>
</tbody>
</table>

**The VARNAME State**

This state differentiates variable names from qualified names. This allows only the pattern of a QName to be recognized when otherwise ambiguities could occur.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>VarName</td>
<td>OPERATOR</td>
</tr>
<tr>
<td>&quot;(&quot;</td>
<td>EXPRCOMMENT</td>
</tr>
<tr>
<td>pushState()</td>
<td></td>
</tr>
</tbody>
</table>
The EXPR_COMMENT State

The "(:" token marks the beginning of an expression Comment, and the ":)" token marks the end. This allows no special interpretation of other characters in this state.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Transition To State</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;:)&quot;</td>
<td>popState()</td>
</tr>
<tr>
<td>&quot;(:&quot;</td>
<td>EXPR_COMMENT</td>
</tr>
<tr>
<td>ExprCommentContent</td>
<td>(maintain state)</td>
</tr>
</tbody>
</table>

A.3 Reserved Function Names

The following is a list of names that may not be used as user function names, in an unprefixed form.

- if
- typeswitch
- node
- comment
- text
- processing-instruction

A.4 Precedence Order

In all cases the grammar defines built-in precedence. In the cases where a number of statements are a choice at the same production level, the expressions are always evaluated from left to right.

B Type Promotion and Operator Mapping

B.1 Type Promotion

Under certain circumstances, an atomic value can be promoted from one type to another. Type promotion is used in function calls (see 3.1.4 Function Calls) and in processing of operators that accept numeric operands (listed in the tables below). The following type promotions are permitted:

1. A value of type xs:float (or any type derived by restriction from xs:float) can be promoted to the type xs:double. The result is the xs:double value that is the same as the original value. This kind of promotion may cause loss of precision.

2. A value of type xs:decimal (or any type derived by restriction from xs:decimal) can be promoted to either of the types xs:float or xs:double. The result is the value of the target type that is closest to the original value.

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

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

- A function that expects a parameter $p$ of type xs:decimal can be invoked with a value of type xs:integer. This is an example of subtype substitution. The value retains its original type. Within the body of the function, $p$ instance of xs:integer returns true.
B.2 Operator Mapping

The tables in this section list the combinations of types for which the various operators of XPath are defined. For each valid combination of types, the table indicates the function(s) that are used to implement the operator and the type 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.

Operators listed in the tables may be validly applied to operands whose types are derived by restriction from the listed operand types. For example, a table entry indicates that the `gt` operator may be applied to two `xs:date` operands, returning `xs:boolean`. Therefore, the `gt` operator may also be applied to two (possibly different) subtypes of `xs:date`, also returning `xs:boolean`.

In the operator tables, the term numeric refers to the types `xs:integer`, `xs:decimal`, `xs:float`, and `xs:double`. An operator whose operands and result are designated as numeric might be thought of as representing four operators, one for each of the numeric types. For example, the numeric `+` operator might be thought of as representing the following four operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>First operand type</th>
<th>Second operand type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td><code>xs:integer</code></td>
<td><code>xs:integer</code></td>
<td><code>xs:integer</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>xs:decimal</code></td>
<td><code>xs:decimal</code></td>
<td><code>xs:decimal</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>xs:float</code></td>
<td><code>xs:float</code></td>
<td><code>xs:float</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>xs:double</code></td>
<td><code>xs:double</code></td>
<td><code>xs:double</code></td>
</tr>
</tbody>
</table>

A numeric operator accepts operands of the four numeric types and any type that is derived by restriction from one of the four numeric types. If the result type of an operator is listed as numeric, it means "the first numeric type, in promotion order, into which all operands can be converted by subtype substitution and promotion." As an example, suppose that the type `hatsize` is derived from `xs:integer` and the type `shoesize` is derived from `xs:float`. Then if the `+` operator is invoked with operands of type `hatsize` and `shoesize`, it returns a result of type `xs:float`. Similarly, if `+` is invoked with two operands of type `hatsize` it returns a result of type `xs:integer`.

In the 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`.):
<table>
<thead>
<tr>
<th>Operator</th>
<th>Type(A)</th>
<th>Type(B)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>numeric</td>
<td>numeric</td>
<td>op:numeric-add(A, B) numeric</td>
</tr>
<tr>
<td>A + B</td>
<td>xs:date</td>
<td>xdt:yearMonthDuration</td>
<td>op:add-yearMonthDuration-to-date(A, B) xs:date</td>
</tr>
<tr>
<td>A + B</td>
<td>xdt:yearMonthDuration</td>
<td>xs:dateTime</td>
<td>op:add-yearMonthDuration-to-dateTime(A, B) xs:dateTime</td>
</tr>
<tr>
<td>A + B</td>
<td>xdt:dayTimeDuration</td>
<td>xs:date</td>
<td>op:add-dayTimeDuration-to-date(B, A) xs:date</td>
</tr>
<tr>
<td>A + B</td>
<td>xs:dateTime</td>
<td>xdt:yearMonthDuration</td>
<td>op:add-dayTimeDuration-to-dateTime(B, A) xs:dateTime</td>
</tr>
<tr>
<td>A + B</td>
<td>xdt:dayTimeDuration</td>
<td>xs:dateTime</td>
<td>op:add-dayTimeDuration-to-dateTime(B, A) xs:dateTime</td>
</tr>
<tr>
<td>A + B</td>
<td>xs:dateTime</td>
<td>xdt:dayTimeDuration</td>
<td>op:add-dayTimeDuration-to-dateTime(B, A) xs:dateTime</td>
</tr>
<tr>
<td>A + B</td>
<td>xdt:yearMonthDuration</td>
<td>xdt:yearMonthDuration</td>
<td>op:add-yearMonthDurations(A, B) xdt:yearMonthDuration</td>
</tr>
<tr>
<td>A * B</td>
<td>numeric</td>
<td>numeric</td>
<td>op:numeric-multiply(A, B) numeric</td>
</tr>
<tr>
<td>A * B</td>
<td>xdt:yearMonthDuration</td>
<td>xdt:yearMonthDuration</td>
<td>op:multiply-yearMonthDuration(A, B) xdt:yearMonthDuration</td>
</tr>
<tr>
<td>A * B</td>
<td>xdt:dayTimeDuration</td>
<td>xdt:dayTimeDuration</td>
<td>op:multiply-dayTimeDuration(A, B) xdt:dayTimeDuration</td>
</tr>
<tr>
<td>A ÷ B</td>
<td>numeric</td>
<td>numeric</td>
<td>op:numeric-divide(A, B) numeric</td>
</tr>
<tr>
<td>A ÷ B</td>
<td>xdt:yearMonthDuration</td>
<td>xdt:yearMonthDuration</td>
<td>op:divide-yearMonthDuration(A, B) xdt:yearMonthDuration</td>
</tr>
<tr>
<td>A ÷ B</td>
<td>xdt:dayTimeDuration</td>
<td>xdt:dayTimeDuration</td>
<td>op:divide-dayTimeDuration(A, B) xdt:dayTimeDuration</td>
</tr>
<tr>
<td>A mod B</td>
<td>numeric</td>
<td>numeric</td>
<td>op:numeric-modulo(A, B) numeric</td>
</tr>
<tr>
<td>A mod B</td>
<td>xdt:yearMonthDuration</td>
<td>xdt:yearMonthDuration</td>
<td>op:modulo-yearMonthDuration(A, B) xdt:yearMonthDuration</td>
</tr>
<tr>
<td>A mod B</td>
<td>xdt:dayTimeDuration</td>
<td>xdt:dayTimeDuration</td>
<td>op:modulo-dayTimeDuration(A, B) xdt:dayTimeDuration</td>
</tr>
<tr>
<td>A % B</td>
<td>numeric</td>
<td>numeric</td>
<td>op:numeric-modulo(A, B) numeric</td>
</tr>
<tr>
<td>A % B</td>
<td>xdt:yearMonthDuration</td>
<td>xdt:yearMonthDuration</td>
<td>op:modulo-yearMonthDuration(A, B) xdt:yearMonthDuration</td>
</tr>
<tr>
<td>A % B</td>
<td>xdt:dayTimeDuration</td>
<td>xdt:dayTimeDuration</td>
<td>op:modulo-dayTimeDuration(A, B) xdt:dayTimeDuration</td>
</tr>
<tr>
<td>A eq B</td>
<td>numeric</td>
<td>numeric</td>
<td>op:numeric-equality(A, B) numeric</td>
</tr>
<tr>
<td>A eq B</td>
<td>xs:integer</td>
<td>xs:integer</td>
<td>fn:not(op:numeric-equality(A, B)) xs:boolean</td>
</tr>
<tr>
<td>A eq B</td>
<td>xs:dateTime</td>
<td>xs:dateTime</td>
<td>fn:yearMonthDuration-equal(A, B) xs:boolean</td>
</tr>
<tr>
<td>A eq B</td>
<td>xs:dayTimeDuration</td>
<td>xs:dayTimeDuration</td>
<td>fn:dayTimeDuration-equal(A, B) xs:boolean</td>
</tr>
<tr>
<td>A eq B</td>
<td>xs:Gregorian</td>
<td>xs:Gregorian</td>
<td>fn:gYear-equal(A, B) xs:boolean</td>
</tr>
<tr>
<td>A eq B</td>
<td>xs:hexBinary</td>
<td>xs:hexBinary</td>
<td>fn:hex-binary-equal(A, B) xs:boolean</td>
</tr>
<tr>
<td>A eq B</td>
<td>xs:base64Binary</td>
<td>xs:base64Binary</td>
<td>fn:base64-binary-equal(A, B) xs:boolean</td>
</tr>
<tr>
<td>A eq B</td>
<td>xs:anyURI</td>
<td>xs:anyURI</td>
<td>fn:anyURI-equal(A, B) xs:boolean</td>
</tr>
<tr>
<td>A eq B</td>
<td>xs:QName</td>
<td>xs:QName</td>
<td>fn:QName-equal(A, B) xs:boolean</td>
</tr>
<tr>
<td>A eq B</td>
<td>xs:NOTATION</td>
<td>xs:NOTATION</td>
<td>fn:NOTATION-equal(A, B) xs:boolean</td>
</tr>
<tr>
<td>A ne B</td>
<td>numeric</td>
<td>numeric</td>
<td>fn:not(op:numeric-equality(A, B)) xs:boolean</td>
</tr>
<tr>
<td>Comparison</td>
<td>Type 1</td>
<td>Type 2</td>
<td>Function</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
<td>--------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:boolean</td>
<td>xs:boolean</td>
<td>fn:not(op:boolean-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:string</td>
<td>xs:string</td>
<td>fn:not(op:numeric-equal(fn:compare(A, B), 1))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:date</td>
<td>xs:date</td>
<td>fn:not(op:date-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:time</td>
<td>xs:time</td>
<td>fn:not(op:time-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:dateTime</td>
<td>xs:dateTime</td>
<td>fn:not(op:dateTime-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:yearMonthDuration</td>
<td>xs:yearMonthDuration</td>
<td>fn:not(op:yearMonthDuration-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:dayTimeDuration</td>
<td>xs:dayTimeDuration</td>
<td>fn:not(op:dayTimeDuration-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>Gregorian</td>
<td>Gregorian</td>
<td>fn:not(op:gYear-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:hexBinary</td>
<td>xs:hexBinary</td>
<td>fn:not(op:hex-binary-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:base64Binary</td>
<td>xs:base64Binary</td>
<td>fn:not(op:base64-binary-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:anyURI</td>
<td>xs:anyURI</td>
<td>fn:not(op:anyURI-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:QName</td>
<td>xs:QName</td>
<td>fn:not(op:QName-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>xs:NOTATION</td>
<td>xs:NOTATION</td>
<td>fn:not(op:NOTATION-equal(A, B))</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>numeric</td>
<td>numeric</td>
<td>fn:not(op:numeric-equal(fn:compare(A, B), 1))</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>numeric</td>
<td>numeric</td>
<td>op:numeric-greater-than(A, B)</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>xs:boolean</td>
<td>xs:boolean</td>
<td>op:boolean-greater-than(A, B)</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>xs:string</td>
<td>xs:string</td>
<td>op:numeric-greater-than(fn:compare(A, B), 0)</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>xs:date</td>
<td>xs:date</td>
<td>op:date-greater-than(A, B)</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>xs:time</td>
<td>xs:time</td>
<td>op:time-greater-than(A, B)</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>xs:dateTime</td>
<td>xs:dateTime</td>
<td>op:datetime-greater-than(A, B)</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>xdt:yearMonthDuration</td>
<td>xdt:yearMonthDuration</td>
<td>op:yearMonthDuration-greater-than(A, B)</td>
</tr>
<tr>
<td>A &gt; B</td>
<td>xdt:dayTimeDuration</td>
<td>xdt:dayTimeDuration</td>
<td>op:dayTimeDuration-greater-than(A, B)</td>
</tr>
<tr>
<td>A ≤ B</td>
<td>numeric</td>
<td>numeric</td>
<td>fn:not(op:numeric-greater-than(A, B))</td>
</tr>
<tr>
<td>A ≤ B</td>
<td>xs:string</td>
<td>xs:string</td>
<td>op:numeric-greater-than(fn:compare(A, B), -1)</td>
</tr>
<tr>
<td>A ≤ B</td>
<td>xs:date</td>
<td>xs:date</td>
<td>fn:not(op:date-less-than(A, B))</td>
</tr>
<tr>
<td>A ≤ B</td>
<td>xs:time</td>
<td>xs:time</td>
<td>fn:not(op:time-less-than(A, B))</td>
</tr>
<tr>
<td>A ≤ B</td>
<td>xs:dateTime</td>
<td>xs:dateTime</td>
<td>fn:not(op:dateTime-less-than(A, B))</td>
</tr>
<tr>
<td>A ≤ B</td>
<td>xdt:yearMonthDuration</td>
<td>xdt:yearMonthDuration</td>
<td>op:yearMonthDuration-less-than(A, B)</td>
</tr>
<tr>
<td>A ≤ B</td>
<td>xdt:dayTimeDuration</td>
<td>xdt:dayTimeDuration</td>
<td>op:dayTimeDuration-less-than(A, B)</td>
</tr>
<tr>
<td>A ≥ B</td>
<td>numeric</td>
<td>numeric</td>
<td>fn:not(op:numeric-less-than(A, B))</td>
</tr>
<tr>
<td>A ≥ B</td>
<td>xs:string</td>
<td>xs:string</td>
<td>fn:not(op:numeric-less-than(fn:compare(A, B), 0))</td>
</tr>
<tr>
<td>A ≥ B</td>
<td>xs:date</td>
<td>xs:date</td>
<td>fn:not(op:date-greater-than(A, B))</td>
</tr>
<tr>
<td>A ≥ B</td>
<td>xs:time</td>
<td>xs:time</td>
<td>fn:not(op:time-greater-than(A, B))</td>
</tr>
<tr>
<td>A ≥ B</td>
<td>xs:dateTime</td>
<td>xs:dateTime</td>
<td>fn:not(op:datetime-greater-than(A, B))</td>
</tr>
<tr>
<td>A ≥ B</td>
<td>xdt:yearMonthDuration</td>
<td>xdt:yearMonthDuration</td>
<td>fn:not(op:yearMonthDuration-less-than(A, B))</td>
</tr>
<tr>
<td>A ≥ B</td>
<td>xdt:dayTimeDuration</td>
<td>xdt:dayTimeDuration</td>
<td>fn:not(op:dayTimeDuration-less-than(A, B))</td>
</tr>
<tr>
<td>A = B</td>
<td>node</td>
<td>node</td>
<td>op:node-equal(A, B)</td>
</tr>
<tr>
<td>A ≠ B</td>
<td>node</td>
<td>node</td>
<td>fn:not(op:node-equal(A, B))</td>
</tr>
<tr>
<td>A &lt;&lt; B</td>
<td>node</td>
<td>node</td>
<td>op:node-before(A, B)</td>
</tr>
<tr>
<td>A &gt;&gt; B</td>
<td>node</td>
<td>node</td>
<td>op:node-after(A, B)</td>
</tr>
<tr>
<td>A ∪ B</td>
<td>node*</td>
<td>node*</td>
<td>op:union(A, B)</td>
</tr>
<tr>
<td>A ∩ B</td>
<td>node*</td>
<td>node*</td>
<td>op:intersection(A, B)</td>
</tr>
</tbody>
</table>
### Unary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand type</th>
<th>Function</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ A</td>
<td>numeric</td>
<td>op:numeric-unary-plus(A)</td>
<td>numeric</td>
</tr>
<tr>
<td>- A</td>
<td>numeric</td>
<td>op:numeric-unary-minus(A)</td>
<td>numeric</td>
</tr>
</tbody>
</table>

### C References

#### C.1 Normative References

**XQuery 1.0 and XPath 2.0 Functions and Operators**

**XQuery 1.0 Formal Semantics**

**XML Schema**

**XQuery 1.0 and XPath 2.0 Data Model**

**XSLT 2.0 and XQuery 1.0 Serialization**

**XML**

**XML Names**

**ISO/IEC 10646**

**ISO/IEC 10646-2000**

**Unicode**

**Unicode3**

#### C.2 Non-normative References

**XPath 2.0 Requirements**

**XQueryX 1.0**
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.

XQuery

XSLT 2.0

ISO 8601

C.3 Background References

XQL

XML-QL

SQL

ODMG

Lorel

YATL

Quilt

XPath 1.0

C.4 Informative Material

RFC2396

Character Model

XML Infoset

XPointer

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

D Glossary
E Conformance

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

F Backwards Compatibility with XPath 1.0 (Non-Normative)

This section provides a summary of the main areas of incompatibility between XPath 2.0 and [XPath 1.0].

The list given here assumes (a) that the source document is processed in the absence of a schema, and (b) that XPath 1.0 compatibility mode is true.

In the description below, the terms node-set and number are used with their XPath 1.0 meanings, that is, to describe expressions which according to the rules of XPath 1.0 would have generated a node-set or a number respectively.

- The rules for comparing a node-set to a boolean have changed. In XPath 1.0, an expression such as \$nodeset=true() was evaluated by converting the node-set to a boolean and comparing the result: so this expression would return true if \$nodeset was non-empty. In XPath 2.0, this expression is handled in the same way as other comparisons between a sequence and a singleton: it is true if \$nodeset contains at least one node whose typed value is true.

  This means that if \$nodeset is empty, the result will be false regardless of the value of the boolean operand, and regardless of which operator is used. If \$nodeset is non-empty, then in the absence of a schema the expression will usually raise a type error, caused by a failure when attempting to cast the untyped value of a node to a boolean. But if a node has the value "0", "1", "true", or "false", evaluation of the expression may succeed.

- Additional numeric types have been introduced, with the effect that arithmetic may now be done as an integer, decimal, or single-precision floating point calculation where previously it was always performed as double-precision floating point. The result of the div operator when dividing two integers is now a value of type decimal rather than double. The expression 10 div 0 raises an error rather than returning positive infinity.

- The rules for converting numbers to strings have changed. These will affect the way numbers are displayed in the output of a stylesheet. The output format depends on the data type of the result: floating point values, for example, will be displayed using scientific notation. The result of a decimal calculation such as 1.5 + 3.5 will be displayed as 5.0, not 5 as previously. The general rule is that the numbers are converted to strings using the rules for casting to the type xs:string.

- The rules for converting strings to numbers have changed. A string that cannot be interpreted as a number now produces an error, whereas in XPath 1.0 it produced the value NaN (not a number). The representation of special values such as Infinity has been aligned with XML Schema. Strings containing a leading plus sign, or numbers in scientific notation, may now be converted to ordinary numeric values, whereas in XPath 1.0 they were converted to NaN.

- Many operations in XPath 2.0 produce an empty sequence as their result when one of the arguments or operands is an empty sequence. With XPath 1.0, the result of such an operation was typically a zero-length string or the numeric value NaN. Examples include the numeric operators and certain functions such as substring. Functions also produce an empty sequence when applied to an argument for which no other value is defined. However, for functions in the core such as name, the XPath 1.0 behavior has been retained; for example, applying the name function to a text node produces a zero-length string, as it did in XPath 1.0.

- In XPath 1.0, the < and > operators, when applied to two strings, attempted to convert both the strings to numbers and then made a numeric comparison between the results. In XPath 2.0, these operators perform a lexicographic comparison using the default collating sequence.
In XPath 1.0, functions and operators that compared strings (for example, the = operator and the contains function) worked on the basis of character-by-character equality of Unicode codepoints, allowing Unicode normalization at the discretion of the implementor. In XPath 2.0, these comparisons are done using the default collating sequence. The working group may define mechanisms allowing codepoint comparison to be selected as the default collating sequence, but there is no such mechanism in the current draft.

In XPath 1.0, it was defined that with an expression of the form A and B, B would not be evaluated if A was false. Similarly in the case of A or B, B would not be evaluated if A was true. This is no longer guaranteed with XPath 2.0: the implementation is free to evaluate the two operands in either order or in parallel. This change has been made to give more scope for optimization in situations where XPath expressions are evaluated against large data collections supported by indexes.

In XPath 1.0, the expression (4 = true()) is evaluated by converting the number 4 to boolean (yielding true). The expression returns true. In XPath 2.0, running in compatibility mode, the same expression is evaluated by converting both operands to double (yielding 4e0 = 1e0). The expression returns false.

Consecutive comparison operators such as A < B < C were supported in XPath 1.0, but are not supported in XPath 2.0.

Constructs such as @text(), @comment(), and @processing-instruction() were legal in XPath 1.0 (they returned an empty sequence) but have become illegal according the the XPath 2.0 grammar.

The namespace axis is deprecated in XPath 2.0. Implementations may support the namespace axis for backward compatibility with XPath 1.0, but they are not required to do so.

Suppose that the context node is an element node derived from the following markup: <background color="red green blue">. In XPath 1.0, the predicate [@color="blue"] would return false. In XPath 2.0, if the color attribute is defined in a schema or DTD to be of type NMTOKENS, the same predicate will return true.

Backward compatibility issues affecting individual functions in the core function library are described in [XQuery 1.0 and XPath 2.0 Functions and Operators].

G 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).

<table>
<thead>
<tr>
<th>Num</th>
<th>Cl</th>
<th>Pr</th>
<th>Cluster</th>
<th>Status</th>
<th>Locus</th>
<th>Description</th>
<th>Responsible</th>
</tr>
</thead>
<tbody>
<tr>
<td>293</td>
<td>1</td>
<td></td>
<td></td>
<td>decided</td>
<td>xquery</td>
<td>Cdata and CharRef Semantics</td>
<td></td>
</tr>
<tr>
<td>444</td>
<td>1</td>
<td></td>
<td></td>
<td>decided</td>
<td>formal-semantics</td>
<td>Support for mixed content in the type system</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>o-1</td>
<td>equality-operators</td>
<td>decided</td>
<td>xpath</td>
<td>Normalized Equality</td>
<td>What does CDATA section constructor construct?</td>
<td></td>
</tr>
<tr>
<td>328</td>
<td>1</td>
<td></td>
<td>cdata section</td>
<td>decided</td>
<td>xquery</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page</td>
<td>Collection/Decision</td>
<td>Active Xpath</td>
<td>Question</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>--------------</td>
<td>----------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>546</td>
<td>conformance-levels</td>
<td></td>
<td>Are there more processing model options that could make sense as consistent features and thus as conformance levels?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>286</td>
<td>constructor-expr</td>
<td></td>
<td>Element Construction vs Streaming</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>290</td>
<td>constructor-expr</td>
<td></td>
<td>Element Attribute Constructor Name Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>529</td>
<td>constructor-expr</td>
<td></td>
<td>Node identity in the formal semantics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>329</td>
<td>constructors</td>
<td></td>
<td>Duplicate attribute constructors</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>258</td>
<td>documents</td>
<td></td>
<td>Identity of Document Nodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>554</td>
<td>editorial</td>
<td></td>
<td>What is the really normative text?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>553</td>
<td>errors</td>
<td></td>
<td>Rules for reporting dynamic errors statically</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>339</td>
<td>errors</td>
<td></td>
<td>Error type for attributes constructed too late</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>340</td>
<td>errors</td>
<td></td>
<td>How to identify errors?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>317</td>
<td>extensions</td>
<td></td>
<td>XQuery Extension Mechanisms</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>538</td>
<td>extensions</td>
<td></td>
<td>Can (and should) keyword extensions be allowed in XQuery by equipping them with a namespace prefix?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>272</td>
<td>external-functions</td>
<td></td>
<td>External Functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>273</td>
<td>external-objects</td>
<td></td>
<td>External Objects</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>555</td>
<td>formal semantics</td>
<td></td>
<td>Formal Semantics of Module Import</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>556</td>
<td>formal semantics</td>
<td></td>
<td>Formal Semantics of Variable Definitions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>557</td>
<td>formal semantics</td>
<td></td>
<td>The content of element types should always allow PI’s and comment node types</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>558</td>
<td>formal semantics</td>
<td></td>
<td>New Sequence Type needs to be fully implemented in Formal Semantics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>559</td>
<td>formal-semantics</td>
<td></td>
<td>XPath/XQuery’s current semantics greatly interferes with optimization</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>335</td>
<td>formal-semantics</td>
<td></td>
<td>How do we determine the xml:lang for a node if it inherits xml:lang from a higher-level node? Do we support the sublanguage portion of xml:lang?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>265</td>
<td>FTTF-xml:lang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>266</td>
<td>FTTF-xml:lang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>o-1 functions</td>
<td></td>
<td>External Functions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>o-1 functions</td>
<td></td>
<td>Function Libraries</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>327</td>
<td>functions</td>
<td></td>
<td>Evaluate unused function parameters?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We need a way to declare external functions.

What is in the default context?

Casting and validation

Is validate working on sequences?

Derivation by extension in XQuery

May the content of a text node be the empty string?

Lexical Representation of Atomic Values

Module syntax

Importing Modules

What happens to imported schemas that are used in function signatures?

Should we keep the default function namespace, and the xf: namespace?

Namespace definitions and in-scope namespaces

Do functions in the null namespace clash with functions in the default namespace?

What does default namespace(s) affect?

Computed namespace-constructor

Semantics of order by

When to process the query prolog

Semantics of Schema Context

Support for lax and strict wildcards

Namespaces in element constructors

Conformance Levels

Static type errors and warnings

Implementation of and conformance levels for static type checking

Semantics of data()

Semantics of element and attribute constructors

Dynamic context for current date and time

Data model syntax and literal values

Semantics of special functions

Non-determinism in the semantics
1 Semantics decided formal-semantics Type equivalence rules
1 Semantics decided formal-semantics Treatment of nilability and xsi:nil
1 Semantics decided formal-semantics Representation of text nodes in formal values
1 Semantics decided formal-semantics Validation of an empty string against a string list
1 Semantics decided formal-semantics Collations in the static environment
1 Semantics decided formal-semantics Raising errors
1 Semantics decided formal-semantics Coercion between untyped and atomic values
1 Semantics decided formal-semantics Semantics of XPath 1.0 compatibility
1 Semantics decided formal-semantics Static context accessible from the dynamic context
1 Semantics decided xpath Semantics of text constructor on empty sequence
1 Semantics? active formal-semantics Namespace resolution
2 serialization decided xquery CDATA sections and serialization
1 sort decided xquery Sorting by Non-exposed Data
3 sort decided xpath Provide an example of sorting "disappearing"
2 sort decided xquery Sorting "input to loop", not the result
1 sort decided xquery Add ‘order by’ clause to FLWR?
1 Static typing active formal-semantics Typing for descendant
1 Static typing decided formal-semantics Typing for the typeswitch default clause
1 Static typing decided formal-semantics Static typing of union
1 Static typing decided formal-semantics Static typing of path expressions in the presence of derivation by extension
1 Static typing decided formal-semantics Static typing for validate
1 Static typing decided formal-semantics Imprecise static type of constructed elements
1 syntax active xquery Use "declare" for declarations
1 syntax active xpath Lexical Rules: states normative?
1 syntax active xquery Location of Comments
1 syntax active xpath Should the quotes in processing-instruction("...") be optional?
1 syntax decided xquery Escaping Quotes and Apostrophes
2 syntax decided xquery Nested XQuery comments allowed?
1 syntax decided xpath Problems with SequenceType
1 syntax decided xpath Alignment between path expressions and sequence types
1 syntax decided xpath Lexical state tables
1 syntax decided xpath Plural datatypes different from Singular *?
input(), collection(), document(); validation semantics

Semantics of function calls and notion of "expected type"

What should be the type of an attribute or element that is well-formed but not validated, or is validated in skip mode?

Defining Behavior for Well Formed, DTD, and Schema Documents

Subtype Substitutability

CASE not a subtype

Should there be a lightweight cast?

How are documents for which validation has failed processed?

Should discard the type of copied subelement

Strict static typing for automatic coercion of untyped atomic to atomic values

Should it be a static error if an expression other than () has type empty?

How can a path expression match elements in the substitution group of a given element?

How can a path expression match nodes of a given type?

Human-Readable Syntax for Types

Schema Types from input documents?

Semantics for anySimpleType and untypedAtomic

Data model to infoset mapping and type information

Semantics of fs:cast-untypedAtomic in backward compatibility mode

How does the static semantics works in the case where the input types are unions?

Are XS element/attribute declarations implicitly added to the schema context?

Constructor functions for unprefixed user defined types

Typing support in XPath

Why do we want to allow optional returns and DataType?

Should XPath have "type binding" in variable?
43. xquery_DEFINE_SCHEMA_VARIANTS: Defining Behavior for Well Formed, DTD, and Schema Documents

Issue Class: T Locus: xquery Cluster: type-semantics Priority: 1 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.

Proposed Resolution

The mapping of a DTD validated or well formed document still needs to be defined in the Data Model.

Actual Resolution
Decision by: **xquery** on 2003-02-26 ([link to member only information])

Closing of the directly related issues: [resolved issue #xquery-abstract-syntax] and [resolved issue #xpath-issue-complex-type-value] is sufficient to close this issue.

Decision by: **xpath-tf** on 2003-03-04 ([link to member only information])

Decision by: **xsl** on 2003-03-07 ([link to member only information])

Decision by: **xquery** on 2003-03-07 ([link to member only information])

Joint meeting

On recommendation of the Processing Model Taskforce the disposition is reaffirmed.

**47. xquery-subtype-substitutability:** Subtype Substitutability

Issue Class: **T** Locus: **xquery** Cluster: **type-semantics** Priority: **2** Status: **decided**

Originator: **XQuery Editors** Subtype Substitutability

**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.

**Interactions and Input**

[link to member only information] Michael Rys:

> I think this is still an open issue given some semantic issues that we found between named subtype substitutability and derivation by extension. I will send mail on this issue this week,

**Actual Resolution**

Addressed by the "Named Typing" proposal.

Decision by: **xpath-tf** on 2002-05-07 ([link to member only information])

Decision by: **xquery** on 2002-05-22 ([link to member only information])

Decision by: **xsl** on 2002-06-27 ([link to member only information])

Accepting text in 2002-04-30 public draft.

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

Issue Class: **T** Locus: **algebra** Cluster: **type-semantics** Priority: **3** Status: **decided**

Originator: **XML Query** CASE not a subtype

**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?

**Actual Resolution**

Decision by: **xquery** on 2003-02-12 ([link to member only information])

Issue 48 closed without changes to the document (no static errors will be raised, implementations can but are not required to raise warnings).
56. **xquery-type-syntax**: Human-Readable Syntax for Types

**Issue Class: T**  **Locus: xquery**  **Cluster: type-syntax**  **Priority: 3**  **Status: decided**  **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?

**Actual Resolution**

Decision by: **xquery** on 2003-01-15 ([link to member only information] )

Joint F2F

Decided: Are type names sufficient - YES

Would NUNS be sufficient for type name - not needed

How will type names be bound to definitions - with imported schemas and namespaces

74. **xquery-module-syntax**: Module syntax

**Locus: xquery**  **Cluster: module-semantics**  **Priority: 0-1**  **Status: decided**  **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.

**Proposed Resolution**

Paulist Modules Proposal

---

This proposal adds modules, external variables, external functions, and global variables to XQuery. It addresses the following open issues:

- 74 Module syntax
- 75 Importing Modules
- 124 External Functions
- 157 Function Libraries
- 223 We need a way to declare external functions
- 250 Declaring Variables in Prolog

This proposal is called the Paulist Modules Proposal, because it is based on a simplification Paul Cotton made to earlier module proposals, which can be found here:
1. Modules and Module Declarations

An XQuery module is a resource that contains a Query. A module may declare a target namespace using a ModuleDecl. The syntax of Query and ModuleDecl is as follows:

```
Query ::= ModuleDecl? QueryProlog ExprSequence?
ModuleDecl ::= "module" ModuleURI
```

If a module contains a ModuleDecl, its functions may be imported by other modules. A module that does not have a ModuleDecl cannot be imported. A ModuleURI may be relative - if so, it is combined with the Base URI to create an absolute URI.

In a module that is imported by another module, the optional ExprSequence must be absent, or a static error is raised.

2. Module Imports

A query may import modules in the QueryProlog using ModuleImports. The syntax of QueryProlog and ModuleImport is as follows:

```
QueryProlog ::= (NamespaceDecl | DefaultNamespaceDecl | SchemaImport | ModuleImport | VarDn)*
FunctionDn*
ModuleImport ::= "import" "module" ModuleURI ("at" ImplURI)?
```

Functions and global variables defined in a module may be used in any module that imports the defining module. Importing a module does not provide access to the imported schemas or namespaces of that module. Importing a module does not provide access to the functions imported by that module.

An implementation may locate a module in any convenient manner, provided the ModuleURI of the module it locates matches the ModuleURI of the ModuleImport. The optional ImplURI specifies the resource containing the implementation of the module. It is treated like a schema location hint in XML Schema - an implementation may ignore it completely.

Two modules may import each other. For instance, if module A imports module B, then module B may import module A.

3. Variable declarations and definitions

A variable may be declared or defined using a VarDn in the QueryProlog. The syntax of VarDn is as follows:

```
VarDn ::= "define" "global"? "external"? "variable" "$" VarName ("as" SequenceType)? ("=" Expr)?
```

A variable may have no forward references to functions or variables. Functions and variables imported into a module are considered prior to a global variable if they are imported before the global variable is defined. A variable may not be used before it is defined.

If the "global" keyword is present, then the variable will be imported whenever the module that contains it is imported.

If the "external" keyword is present, the variable is [external]; if it is absent, the variable is [internal].

An [external] variable definition must include only the variable’s name and optional sequence type, or a static error is raised. During dynamic evaluation, an [external] variable must be bound to a value in the dynamic context, or an error is raised.

An [internal] variable definition must include the variable’s name,
optional sequence type, and a required expression, or a static error is raised.

It is a type error, if the value of a variable does not match the declared sequence type of the variable.

4. External functions

An external function definition allows XQuery to call functions written in other languages. The syntax of a FunctionDefn is as follows:

```
FunctionDefn ::= <"define" "function"> "external"? <QName "(" ParamList? "(" | ("(" "as"> SequenceType)) EnclosedExpr
```

If the "external" keyword is present, the function is [external]; if it is absent, the function is [internal].

If a function is [internal], its implementation is given as an XQuery expression in the module's implementation. If a function is [external], its implementation is given by the query environment. XQuery does not specify how external functions are implemented or accessed, how actual arguments are passed to external functions, or how return values are passed to the query environment - all that is implementation defined.

Questions in proposal that need resolution:

In today's telcon, we identified three central questions that need to be resolved for the modules proposal. Once we resolve these questions, it will be easy to rewrite the Paulist module accordingly.

Question 1: What is the relationship between the Module URI and the namespace of the functions it contains?

Possible answers:
1. A. There is no relationship. A module is identified by a URI, and may contain functions in any namespace.
2. B. The module URI is the namespace of functions and global variables defined in that module. It serves as a default namespace for both functions and global variables, but the prolog can change these defaults.
3. C. The module URI is the namespace of functions and global variables exported by the module. A module may contain functions and global variables defined in other namespaces or in no namespace, but they are not exported when the module is imported into another module.

2. Should there an optional Location URI?

2. A. A module has an optional Location URI, which provides the location of a resource. This allows the name of a resource to be decoupled from its location, while allowing relative or absolute locations to be specified explicitly.
2. B. A module is identified only by the Module URI, which may be relative or absolute. Catalogs or other similar devices are used to map the Module URI onto a location.

There are two variations on 2. B:
2. B. i: The Module URI may be relative.
2. B. ii: The Module URI must be absolute.

Relative Module URIs are not useful if we choose 2. A.

Jonathan

Note: Michael Kay suggested Option A and Option B. They are defined as:

Option A: (C3, L1)
Option B: (C1, L2)

Actual Resolution

Decision by: xquery on 2003-02-26 ([link to member only information] )
Decided to adopt 1B and 2A.

75. **xquery-import**: Importing Modules

**Locus:** xquery  **Cluster:** module-semantics  **Priority:** o-1  **Status:** decided  
**Originator:** XQuery Editors  **Importing Modules**

**Description**

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

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

**Actual Resolution**

Decision by: xquery on 2003-02-26 ([link to member only information] )

Decided to adopt 1B and 2A in proposal for issue #xquery-module-syntax.

79. **xquery-encoding**: Encoding

**Locus:** xquery  **Cluster:** module-syntax  **Priority:** o-1  **Status:** decided  
**Originator:** Jonathan Robie  **Encoding**

**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
```

**Actual Resolution**

Decision by: xquery on 2003-01-29 ([link to member only information] )

The text of a query is in Unicode. Serialization is done in Unicode, but APIs can set other encodings for serialization.

96. **xquery-normalized-equality**: Normalized Equality

**Locus:** xpath  **Cluster:** (in)equality-operators  **Priority:** o-1  **Status:** decided  
**Originator:** Mary Fernandez  **Normalized Equality**

**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?

**Actual Resolution**

Decision by: xquery on 2002-09-04 ([link to member only information] )

Decision by: xsl on 2002-10-03 ([link to member only information] ) The XSL WG is not agreeable to blessing this.

"When elements are compared, are comments and PIs considered in the comparison?": NO

"How is whitespace handled?: This depends on the whitespace in the instance as per the data model."
"Do we need to allow more than one way to handle these in comparisons?": NO, not in XPath 2.0/XQuery 1.0.

124. xquery-external-functions: External Functions

Locus: xquery Cluster: functions Priority: o-1 Status: decided
Originator: XQuery Editors External Functions

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.

Actual Resolution

Decision by: xquery on 2003-02-26 ([link to member only information] )

Decided to adopt 1B and 2A in proposal for issue #xquery-module-syntax.

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

Locus: xquery Cluster: syntax Priority: o-1 Status: decided
Originator: XML Query Escaping Quotes and Apostrophes

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?

Interactions and Input

[link to member only information] Michael Rys:

I asked a member of my team to check for a resolution that he can live with. He said he has not found it and currently works on constructing examples that shows the open issue.

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.

Actual Resolution

Decision by: xpath-tf on 2002-04-30 ([link to member only information] )

Decision by: xquery on 2002-05-22 ([link to member only information] )

Decision by: xsl on 2002-06-27 ([link to member only information] )

Syntax in current draft has this, but explanatory text need to be added.

152. xquery-abql: XML-based Syntax

Locus: xqueryx Cluster: xqueryx Priority: o-1 Status: active
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.

155. xquery-phantom-sortby: Sorting by Non-exposed Data

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?

Interactions and Input

Proposed Resolution

Here is an example:

```xml
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
```

Actual Resolution

Decision by: xquery on 2002-09-18 ([link to member only information] )

Decision by: xsl on 2002-10-10 ([link to member only information] ) Confirming that there are no concerns for the resolution of this XQuery only issue.

Resolved by the adoption of the proposal to add "orderby" to the FLWR expression and to drop the previous "sortby" syntax/

157. xquery-function-library: Function Libraries
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.

Actual Resolution

Decision by: xquery on 2003-02-26 ([link to member only information] )

Decided to adopt 1B and 2A in proposal for issue #xquery-module-syntax.

191. whitespace-in-element-constructors: Whitespace handling in element constructors

Locus: xquery Cluster: whitespace Priority: o-1 Status: decided
Originator: XPath EditorsWhitespace handling in element constructors

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:

```
&lt;a&gt;&lt;/a&gt;
generates an empty element:
&lt;a/&gt;
```
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 \u0020 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> </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

Actual Resolution

Decision by: xquery on 2002-09-11 ([link to member only information] )

Decided: Resolve this by adopting the whitespace proposal.

206. xpath-datatype-support: Typing support in XPath

Issue Class: T Locus: xpath Cluster: types Priority: 2 Status: decided
Originator: Mike KayTyping support in XPath

Description

Which of these type productions (CAST, TREAT, ASSERT, TYPESWITCH...) belong in XPath? (ie common to XQuery and XPath)

The list is now (2002-10): CAST, TREAT, VALIDATE, INSTANCE OF, types on variable bindings,
223. **external-function-definition**: We need a way to declare external functions

**Locus**: xquery  
**Cluster**: functions external  
**Priority**: o-1  
**Status**: decided  
**Originator**: Michael Rys

We need a way to declare external functions

**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)?
```

**Interactions and Input**

**Actual Resolution**

Decision by: xquery on 2003-02-26 ([link to member only information] )

Decided to adopt 1B and 2A in proposal for issue #xquery-module-syntax.

224. **function-return-datatype**: Why do we want to allow optional returns and DataType?

**Issue Class**: T  
**Locus**: xquery  
**Cluster**: types  
**Priority**: 2  
**Status**: decided  
**Originator**: Michael Rys

Why do we want to allow optional returns and DataType?

**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.

**Actual Resolution**

Decision by: xquery on 2003-02-12 ([link to member only information] )

Decided that the default is item*.

228. **default-namespace-functions**: Should we keep the default function namespace, and the xf: namespace?

**Locus**: xpath  
**Cluster**: namespace functions  
**Priority**: o-2  
**Status**: decided  
**Originator**: Scott Boag

Should we keep the default function namespace, and the xf: namespace?

**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.

Interactions and Input

[link to member only information] Kristoffer Rose:

Actual Resolution

Decision by: xquery on 2003-01-22 ([link to member only information] )Unqualified, user-defined functions are placed in the default namespace for the scope of the query *only*.

Decision by: xsl on 2003-01-16 ([link to member only information] )

This issue is identical to XSLT issue 155. Decision:

- no controls on default function namespace
- XSLT functions are in F&O namespace
- User defined function must be in a namespace

243. sort-disappearing: Provide an example of sorting "disappearing"

Locus: xpath Cluster: sort Priority: 3 Status: decided
Originator: Michael RysProvide an example of sorting "disappearing"

Description

Provide an example of

(employee sortby data(salary))/name

Actual Resolution

Decision by: xquery on 2002-09-11 ([link to member only information] )
Decision by: xsl on 2002-10-10 ([link to member only information] )

Decided to leave this to editorial discretion.

244. cdata-serialization: CDATA sections and serialization

Locus: xquery Cluster: serialization Priority: 2 Status: decided
Originator: Michael RysCDATA sections and serialization

Description

What are the semantics of CDATA sections in XQuery? Are they preserved in the data model for serialization?

Actual Resolution
Decision by: xquery on 2002-12-11 ([link to member only information])

Decided to close, with document changes as described by Mike Kay in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Dec/0095.html, including editorial comments.

246. nested-comments: Nested XQuery comments allowed?

Locus: xquery Cluster: syntax Priority: 2 Status: decided
Originator: Michael Rys Nested XQuery comments allowed?

Description

Nested XQuery comments allowed?

Actual Resolution

Decision by: xsl on 2003-03-07 ([link to member only information]) Joint meeting
Decision by: xquery on 2003-03-07 ([link to member only information]) Joint meeting

Permit nested comments. Use "(:" and ":)" syntax.

247. namespace-default-affecting: What does default namespace(s) affect?

Locus: xpath Cluster: namespaces Priority: 2 Status: decided
Originator: XPath TF What does default namespace(s) affect?

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.

Actual Resolution

Decision by: xquery on 2002-09-11 ([link to member only information])

Decided to accept status quo:
- 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.

250. declaring-variables-in-prolog: Declaring Variables in Prolog

Locus: xquery Cluster: variables Priority: 2 Status: decided
Originator: Jonathan Robie Declaring Variables in Prolog

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:
One possible syntax for this is:

```
'define' ''variable' varname ':=' expr
```

The above production would occur only in the prolog.

**Actual Resolution**

Decision by: **xquery** on 2003-02-26 ([link to member only information](#))

Decided to adopt 1B and 2A in proposal for issue #xquery-module-syntax.

251. **sort-by**: Sorting "input to loop", not the result

**Locus:** xquery  **Cluster:** sort  **Priority:** 2  **Status:** decided

**Originator:** Phil Wadler  

**Description**

Consider this query:

```
for $x in /books/book, $y in /reviews/review
  where $x/isbn = $y/isbn
  return <newbook躅 $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躅 $x/title, $x/author, $y/reviewer, $x/isbn </dummy>
sortby isbn
return <newbook躅 $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躅 $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.

**Actual Resolution**

Decision by: **xquery** on 2002-09-18 ([link to member only information](#))

Decision by: **xsl** on 2002-10-10 ([link to member only information](#))  

Confirming that there are no concerns for the resolution of this XQuery only issue.

Resolved by the adoption of the proposal to add "orderby" to the FLWR expression and to drop the previous "sortby" syntax.

257. **collection-always-return-same**: Does collection() always return
same result?

Issue Class: D Locus: xpath Cluster: collections Priority: 1 Status: decided
Originator: XPath TF Does collection() always return same result?

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?

Actual Resolution

Decision by: xpath-tf on 2002-10-16 ([link to member only information] )
Decision by: xquery on 2002-10-23 ([link to member only information] )
Decision by: xsl on 2002-10-31 ([link to member only information] )

It should return the same answer every time. Applies also to input(). We should use the same language as we use for document() and for current-dateTime().

258. document-nodes-identity: Identity of Document Nodes

Issue Class: D Locus: xpath Cluster: documents Priority: 2 Status: decided
Originator: Jonathan Robie Identity of Document Nodes

Description

Consider the following query:

```xml
define ($root := document("foo.com"))
if ($root == document("foo.com"))
  then <yep/>
  else <nope/>
```

I would like the following output:

```xml
<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?

```xml
define ($root := document("foo.com/here/there/hi.xml"))
define ($limerick := document("file://c:/temp/limerick-tei.xml"))
if ($root == $limerick)
  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?

Actual Resolution

Decision by: xpath-tf on 2002-10-16 ([link to member only information] )
Decision by: xquery on 2002-10-23 ([link to member only information] )
Decision by: xsl on 2002-10-31 ([link to member only information] )

This is already covered by the existing spec, but we may want to review the language as part of the actions for issue 257.
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  Priority: 2  Status: decided  
Originator: FTTF

How do we determine the xml:lang for a node if it inherits xml:lang from a higher-level node?

**Description**

How do we determine the xml:lang for a node if it inherits xml:lang from a higher-level node?

**Actual Resolution**

Decision by: xquery on 2003-02-12 ([link to member only information] )

Decided to close issue with no change to documents, since user-written expressions or functions can determine the value of xml:lang that a node inherits.

266. **lang-sublanguage-support**: Do we support the sublanguage portion of xml:lang?

Locus: xpath-fulltext  Cluster: FTTF-xml:lang  Priority: 2  Status: decided  
Originator: FTTF

Do we support the sublanguage portion of xml:lang? If so, how?

**Actual Resolution**

Decision by: xquery on 2003-02-26 ([link to member only information] )

A user-written expression of function has to determine the value of xml:lang and then do the appropriate analysis of the value. No change to document.

272. **external-functions**: External Functions

Locus: xpath  Cluster: external-functions  Priority: 1  Status: decided  
Originator: Michael Kay

External Functions

**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.

Actual Resolution

Decision by: xpath-tf on 2002-12-10 ([link to member only information] )

Proposed to resolve by stating that the XPath specification to state that at the discretion of the host language, a call to a function that is not in the static context may generate a dynamic error rather than a static error.

Decision by: xquery on 2003-02-26 ([link to member only information] )

This issue is also impacted by the following decision:

Decided to adopt 1B and 2A in proposal for issue #xquery-module-syntax.

273. external-objects: External Objects

Locus: xpath Cluster: external-objects Priority: 1 Status: decided
Originator: Michael Kay External Objects

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.

Interactions and Input

Proposed Resolution
Technically, I believe we could close the issue with no change to the documents, on the basis that the
definition for the static context (in both XPath and XQuery) states: "Additional type definitions may
be added to the in-scope type definitions by the language environment." I think an implementation
could take this as sufficient authority to extend the type hierarchy with additional types, including
types needed to represent objects returned by extension functions (such as "an SQL database
connection").

Editorially, I think it would be a good idea if we state this explicitly in a Note, as follows:

NOTE: an implementation may allow function calls in a Query/Expression to bind to functions
written in languages that use a different type system (these are known as "extension functions" in
XSLT, "external functions" in XQuery). In this case, the way in which the arguments and results of
such functions are mapped between the two type systems is implementation-defined. An
implementation may define new types anywhere within the type hierarchy of the [Data Model] that
are designed to facilitate interworking with other languages, for example a type that encapsulates an
object returned by an external function; or it may provide mechanisms for the user to define such
types.

Actual Resolution

Decision by: xpath-tf on 2003-03-11 ([link to member only information] )

Decision by: xquery on 2003-03-19 ([link to member only information] )

Proposal accepted.

279. xpath-lightweight-cast: Should there be a lightweight cast?

Issue Class: T Locus: xpath Cluster: type-semantics Priority: 1 Status: decided
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:

\[
('cast'|'validate') \text{ as } \text{SequenceType} ()
\]

Actual Resolution

Decision by: xquery on 2002-09-11 ([link to member only information] )

Decision by: xsl on 2002-10-10 ([link to member only information] )

Decided to reject this proposal. Cast checks all facets.

286. element-construction-vs-streaming: Element Construction vs Streaming

Locus: xquery Cluster: constructor-expr Priority: 1 Status: decided
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**

**Actual Resolution**

Decision by: xquery on 2002-09-04 ([link to member only information] )

This issue has been addressed by the recent changes to the element constructor.

290. **element-attribute-constructor-name-type**: Element Attribute Constructor Name Type

**Locus:** xquery **Cluster:** constructor-expr **Priority:** 1 **Status:** decided

**Originator:** Michael Rys

**Element Attribute Constructor Name Type**

**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).

**Proposed Resolution**

1. In Section 3.7.1 ("Direct Element Constructors"), modify the paragraph below the first example as follows:

"Unqualified element names used in a direct element constructor are implicitly qualified by the default namespace for element names. In a direct element constructor, the name used in the end tag must exactly match the name used in the corresponding start tag (including its prefix or absence of a prefix)."

2. In Section 3.7.2.1 ("Computed Element Constructors"), add a new first paragraph as follows:

"The name expression of a computed element constructor is processed as follows:

1. If the name expression returns a QName, that QName is used as the name of the constructed element. If the QName returned by the name expression is unqualified, the name of the constructed element is in default namespace for element names.

2. If the name expression returns a string, that string is implicitly cast to a QName by using the fn:QName-in-context function with its $use-default parameter set to True. The resulting QName is used as the name of the constructed element.

3. If the name expression does not return a QName or a string, a dynamic error is raised."
(3) In Section 3.7.2.2 ("Computed Attribute Constructors"), add a new first paragraph as follows:

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

1. If the name expression returns a QName, that QName is used as the name of the constructed attribute. If the QName returned by the name expression is unqualified, the name of the constructed attribute is in no namespace.

2. If the name expression returns a string, that string is implicitly cast to a QName by using the fn:QName-in-context function with its $use-default parameter set to True. The resulting QName is used as the name of the constructed attribute.

3. If the name expression does not return a QName or a string, a dynamic error is raised."

Actual Resolution

Decision by: x-editors on 2003-02-05 ([link to member only information] )

Decision by: xquery on 2003-02-12 ([link to member only information] )

Recommend Don’s proposal as the closure of this issue.

293. cdata-charref-semantics: Cdata and CharRef Semantics

Locus: xquery Cluster: Priority: Status: decided
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.

Actual Resolution

Decision by: xquery on 2002-12-11 ([link to member only information] )

Decided to close, with document changes as described by Mike Kay in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Dec/0095.html, including editorial comments.

295. lexical-representation-of-atomic-values: Lexical Representation of Atomic Values

Locus: xquery Cluster: lexical-representation Priority: Status: decided
Originator: xquery/xsl f2f

Description

The wording in section 2.8.3 needs to be aligned with Data model and F&O; 'lexical representation’ needs to be defined differently and be consistent in XQuery, Datamodel, F&O. It probably needs to
be string value (canonical value of integer in W3C Schema has a ",." so not appropriate [or we need to have our own definition of the canonical value...]).

**Actual Resolution**

Decision by: **xsl** on 2003-03-06 ([link to member only information] )Joint meeting

Decision by: **xquery** on 2003-03-06 ([link to member only information] )Joint meeting

3.7.2.4 "text node constructors", that’s the one that should be aligned to "string-value".

### 297. `binding`: Should XPath have "type binding" in variable?

**Issue Class:** T  **Locus:** xpath  **Cluster:** types  **Priority:** 1  **Status:** decided

**Originator:** XQuery Editors

**Should XPath have "type binding" in variable?**

**Description**

The 2002-06-24 grammar introduces the "binding" production. "assert" is removed from both XQuery and XPath. "binding" currently requires reserved words. Should "binding" be in XPath?

Several reasons NOT to include it in XPath:

- grammar

- semantics don’t match function call semantics

- rarely useful

- reads oddly for range variables if (some integer $x in (1,2,"fred") satisfies $x=1)

- biggest of all, users are asking us to simplify. We will get a very adverse reaction if we make "for" expressions more complicated.

But it is worrying to have such a big split between XPath and XQuery.

**Actual Resolution**

Decision by: **xpath-tf** on 2002-11-12 ([link to member only information] )

Decision by: **xsl** on 2002-11-14 ([link to member only information] )

Decision by: **xsl** on 2003-01-09 ([link to member only information] )

Closed based on the XSLT recommendation to not include type declarations in XPath.

### 306. `PSVI-Data-Model-mapping-normative`: PSVI to Data Model mapping part of normative text?

**Issue Class:** T  **Locus:** xpath  **Cluster:** types  **Priority:** 1  **Status:** decided

**Originator:** Don Chamberlin

**PSVI to Data Model mapping part of normative text?**

**Description**

Should the mapping from PSVI to Query Data Model be part of the normative language specification? (The mapping is affected by the presence or absence of the Schema Import Feature).

**Actual Resolution**

Decision by: **xquery** on 2003-02-26 ([link to member only information] )
307. **schema-types-from-input-documents**: Schema Types from input documents?

**Description**

During the analysis phase, in-scope schema definitions are derived from schemas named in Schema Import clauses.

Should there be additional in-scope schema definitions that are part of the static context? Should there be another set of in-scope schema definitions that are part of the dynamic context? What would that do to static typing? Would this interfere with interoperability?

In particular:

1. Should an environment be allowed to statically predefine schema definitions for a query? This would allow queries on fixed collections or known message types to provide strong type information without forcing users to explicitly import the corresponding schemas.

2. How does type information found dynamically in queried documents affect the query environment? Does data() dynamically use the type information found in the instances that are queried, even when the types have not been declared in imported schemas? Note that Basic XQuery does not work properly if this is not true, since it must be able to discover the types of elements without importing their definition.

May an instance contain types that have not been imported into the static environment? If we say it may not, then schema imports are needed to query any document that contains types that are not predefined in XQuery.

May documents that are queried have different definitions for the same names? Note that solutions that dynamically load type information from a document into the in-scope schema definitions may try to introduce all such definitions into the static environment, which should cause an error according to our current rules.

**Proposed Resolution**

If we want Basic XQuery to be able to query documents that have types that are not predefined, then data() must be able to use type information from instances. But if we want to be able to query validated documents that have different schemas this information must not be added to the static environment (or else we must find a way to add more than one schema definition for a given name to the static environment, or say that the static environment is dynamic, or....).

My tentative conclusions:

1. The in-scope schema definitions should be determined statically.

2. data() and many operators must be able to utilize the type information found in a document.

3. As a matter of convenience, it should be possible to import the schemas associated with a single document or with all documents in a collection. For instance, a syntax like the following may be useful:

```xml
import schema from collection
"jdbc:datadirect:sqlserver://localhost:1433;database='airports'"
```

or
4. Implementations should be allowed to implicitly import schemas into the environment before executing a query, using any of the approved ways above.

**Proposed Resolution**

Proposal (3): Types in Queries, Types in Documents

This proposal distinguishes the relationship between types known statically in the query and types that are discovered dynamically in documents during query processing. This version of the proposal follows the results of a telcon on the subject, as recorded in:

Re: Issue 307 Telcon: Friday, 15 Nov, 12:00 EST From: Jonathan Robie (jonathan.robie@datadirect-technologies.com) Date: Fri, Nov 15 2002
http://lists.w3.org/Archives/Member/w3c-query-editors/2002Nov/0148.html

This proposal addresses the following issue:

Issue 307 schema-types-from-input-documents: Schema Types from input documents?

And here is the proposal itself:

1. Static Types in Queries

Implementations are allowed to implicitly import schemas into the environment before executing a query. The in-scope schema definitions are determined statically, containing the built-in type definitions, any schemas imported explicitly into the query, and any schemas imported implicitly from the query environment.

In a query, it is a static error to use a name in a SequenceType production if the name is not found in the in-scope schema definitions.

2. Types in Input Documents

An input document may be accessed via an input function or via an externally defined variable that is bound to the input document by the query environment. An input document may refer to type names that have not been imported into the in-scope schema definitions, but it must not contain types that have the same names as types in the static environment, but conflicting definitions. Input functions and the query environment must promote unknown type names to the nearest type name in the in-scope schema definitions that is known to be substitutable for the unknown type name. If the unknown type name is derived from a known type by extension, list, or union, then the type name is changed to xs:anySimpleType if it is a simple type, or to xs:anyType if it is a complex type. This loses type information, but ensures that all type names in input documents are also in the in-scope schema definitions.

Note: This approach guarantees that the built-in types and the static in-scope-schema definitions are the only types found in an input document. As a result, there is no need to extend the known types when a new document is read.

Note: One ramification of this approach is that types derived by extension are not substitutable for their base types unless the schemas defining these types are imported into the query. We consider this a fair tradeoff for type safety. If people want to exploit the full type hierarchy, they need to import the schema that defines it.

Note: Mapping all the types when an input function is called is obviously an inefficient implementation strategy. Many implementations will take a different approach, but this is a simple way to specify the desired behavior.

**Actual Resolution**

Decision by: xquery on 2002-12-04 ([link to member only information]) Proposal (3) accepted.
308. type-soundness: Type Soundness

Issue Class: T Locus: xpath Cluster: types Priority: 1 Status: decided
Originator: Don Chamberlin Type Soundness

Description

The Static Typing Feature is claimed to have a property called "type soundness." However, Dana says she has counterexamples to the claim of type soundness as described in this section (Section 2.4.2.3). (At least sequence type matching breaks TYPE SOUNDNESS so what else breaks it?) We need to examine the claim and make sure it is stated correctly.

Actual Resolution

Decision by: xpath-tf on 2003-03-25 ([link to member only information] )

Decision by: xsl on 2002-11-14 ([link to member only information] )

Decision by: xsl on 2003-01-09 ([link to member only information] )

Closed due to the definition of type errors in language and FS books.

310. type-annotations-child-of-element: Are the children of a newly constructed element typed?

Issue Class: T Locus: xquery Cluster: types Priority: 1 Status: decided
Originator: Dana FAre the children of a newly constructed element typed?

Description

What happens with the type annotation of the children of a newly constructed element? Type annotation kept or given xs:anyType.

Actual Resolution

Decision by: xquery on 2002-12-18 ([link to member only information] )

Decision to accept the proposal of Phil Wadler in
http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Dec/0378.html.

Decision by: xquery on 2003-01-29 ([link to member only information] )

Decided to reopen the issue.

Decision by: xquery on 2003-03-06 ([link to member only information] )

Decided NOT to reopen the issue.

311. whitespace-attribute-constructors: Whitespace and Attribute Constructors

Locus: xquery Cluster: whitespace Priority: 1 Status: decided
Originator: Whitespace and Attribute Constructors

Description

Whitespace in an XSLT attribute value template is significant insofar as it survives the XML rules for normalization of attributes. This means that CRLF combinations and tabs are normalized to x20
spaces, unless they are written as XML character references. I think it’s an open question whether XQuery wants to emulate the XML attribute normalization rules.

In XSLT,

```
<foo bar=""/>
```

will produce the result `<foo bar=" x "/>`

while `<foo bar=""/>` produces an element containing an attribute whose value is a single CR character.

If XQuery wants to reproduce this behavior exactly, then it’s going to have to reproduce the XML treatment of whitespace as well as the XSLT treatment.

But whether it does so or not, I think Mary is right that

```
<foo bar=" {'x'} "/>
```

should produce `<foo bar=" x "/>` and not `<foo bar="x"/>`. There is a difference between element and attribute constructors: attributes have quotes around them, and this sets a different expectation.

**Actual Resolution**

Decision by: xquery on 2002-12-04 ([link to member only information] )

Confirmed that Mary’s proposal for whitespace, that was accepted 2002-11-27, closes this issue.

Decision by: xquery on 2002-12-11 ([link to member only information] )

Resolved by the recent adoption of the revised text on attribute constructors.

**316. cardinality-typed-comparison:** Is anySimpleType = anySimpleType*?

**Issue Class:** T  **Locus:** xpath  **Cluster:** types  **Priority:** 1  **Status:** decided  
**Originator:** Michael Rys

**Description**

Given the following function:

```
define function foo(xs:anySimpleType $x) returns xs:anySimpleType {$x}
```

and the untyped data

```
<a a="1"/><a a="2"/><a a="3"/>
```

should the following be a (static or dynamic) type error or should it work?

```
foo(/a/@a)
```

If it should work, what should the following query return?

```
count(data(<e>1 2</e>))
count(data(<a b="1" 2"/>/@b))
```

Should it be 1 or 2?
Proposed Resolution

anySimpleType is the base type of all the *primitive* XML Schema datatypes.

List and union types are *generated* (my word, Schema uses the word *derived*) from primitive types and thus, not included in anySimpleType. See the extract from the XML Schema datatypes spec below.

"[Definition:] There exists a conceptual datatype, whose name is anySimpleType, that is the simple version of the ur-type definition from [XML Schema Part 1: Structures]. anySimpleType can be considered as the *base type* of all *primitive* types. The *value space* of anySimpleType can be considered to be the *union* of the *value space*s of all *primitive* datatypes."

Thus, the function call above should result in a (static or dynamic, depending on typing conformance) type error.

The count expressions would always return 1.

Actual Resolution

Decision by: xpath-tf on 2002-12-17 ([link to member only information] )

Decided that anySimpleType is a list one. This needs to be carried out clearly in the documents.

Decision by: xsl on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: xquery on 2003-03-07 ([link to member only information] )Joint meeting

To solve issues 316 and 524 we need to do eliminate three constructor functions from F&O, and integrating with the changes described in the message: http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2003Mar/0021.html.

This was agreed.

317. extension-mechanism: XQuery Extension Mechanisms

Locus: xquery Cluster: extensions Priority: 1 Status: decided
Originator: Andrew EisenbergXQuery Extension Mechanisms

Description

Issue: XQuery has not determined what extension mechanisms might be supported. XQuery may support the following:

- implementation-defined extensions to the XQuery grammar
- an XQuery-flagger that identifies extensions to XQuery 1.0
- implementation-defined extensions to the functions that are available to
- an XQuery author. The author would be required to explicitly declare the
- namespaces of any extension functions that are used

Interactions and Input

Proposed Resolution

This proposal adds the following to the XQuery language document (in section 2.5 Errors and Conformance):

2.5.5 Extensions

Conforming XQuery implementations are permitted to make two different kinds of extensions to the specifications: grammar extensions and semantic extensions. There are two types of grammar extensions: pragmas and must-understand extensions. While an XQuery implementation may support some...
or all of these types of extensions, this does not negate the requirement to support the XQuery functionality defined in this specification.

2.5.5.1 Pragmas
A pragma may be used to provide additional information to an XQuery implementation.

```
Pragma ::= "(::" "pragma" PragmaQName PragmaContents ")"
PragmaQName ::= ExplicitQName
PragmaContents ::= .*
```

An ExplicitQName requires that QName contain a Prefix. Pragmas may be used anywhere that ignorable whitespace is allowed, and within element content. See A.1 Lexical structure for the exact lexical states where pragmas are recognized. A pragma is identified by its PragmaQName.

If an implementation does not support a pragma, then that pragma shall be ignored. If an implementation does support a pragma and the implementation determines that the PragmaContents are invalid, then a static error is raised. Otherwise, the effect of the pragma on the result of the Query is implementation-defined.

The following example shows how a pragma might be used:

```
declare namespace exq = "http://example.org/XQueryImplementation"
(count(input()//author (:: pragma:timeout 1000 ::))
```

An implementation that supports the exq:timeout pragma might raise a dynamic error if it is unable to count the authors within 1000 seconds. An implementation that did not support this pragma would execute as long as necessary to count the authors.

2.5.5.2 Must-Understand Extensions
An implementation may extend the XQuery grammar by supporting must-understand extensions.

```
MustUnderstandExtension ::= "(::" "extension" ExtensionQName ExtensionContents ")"
ExtensionQName ::= ExplicitQName
ExtensionContents ::= .*
```

A MustUnderstandExtension may be used anywhere that ignorable whitespace is allowed, and within element content. See A.1 Lexical structure for the exact lexical states where pragmas are recognized. A must-understand extension is identified by its ExtensionQName.

If an implementation does not support a must-understand extension, then a static error is raised. If an implementation does support an must-understand extension and the implementation determines that the ExtensionContents are invalid, then a static error is raised. Otherwise, the effect of the must-understand extension on the result of the Query being executed is implementation-defined.

The following example shows how a must-understand extension might be used:

```
declare namespace exq = "http://example.org/XQueryImplementation"
for $e in doc("employees.xml")//employee
  order by $e/lastname (:: extension exq:RightToLeft ::)
  return $e
```

An implementation that supports the exq:RightToLeft must-understand extension might order the last names by examining characters from right to left instead of left to right. An implementation that did not support this must-understand extension would raise a static error.

2.5.5.3 Semantic Extensions
An implementation may extend XQuery by supporting semantic extensions. A semantic extension allows a conforming Query to be processed in a non-conforming way. The way in which such semantic extensions are enabled...
is implementation-defined.

The effect of a semantic extension on the result of a Query is implementation-defined.

The following example shows how a command line might be used to enable a semantic extension:

```xquery
xquery q12.xquery
98
xquery q12.xquery -EmptyIdentity=on
100
```

The execution of the query contained in q12.xquery might treat an empty sequence as 0 when it is being used in addition and subtraction and treat it as 1 when it is being used in multiplication and division.

2.5.5.4 XQuery Flagger

An XQuery Flagger is a facility that is provided by an implementation that is able to identify queries that contain extensions other than pragmas. If an implementation supports extensions other than pragmas, then an XQuery Flagger must be provided. The XQuery Flagger is disabled by default; the mechanism by which the XQuery Flagger is enabled is implementation-defined.

When enabled, the XQuery Flagger will raise a static error for an otherwise valid Query that contains either must-understand extensions or semantic extensions.

An XQuery Flagger is provided to assist programmers in producing queries that are portable among multiple conforming XQuery implementations.

The following examples show how an XQuery Flagger might be used:

```xquery
xquery q10.xquery
<employee> ... </employee>
xquery q10.xquery -Flagger=on
[static error] A must-understand extension is being used: exq:RightToLeft
xquery q12.xquery -EmptyIdentity=on
100
xquery q12.xquery -EmptyIdentity=on -Flagger=on
[static error] A semantic extension is being used: EmptyIdentity
```

Updates to proposal:

- Usage of (:: ::) syntax.
- must-understand extension.

Actual Resolution

Decision by: xquery on 2003-03-12 ([link to member only information] )

#1 PRAGMAS: Adopted Unanimously.

#2 MUST-UNDERSTAND EXTENSIONS: Adopted.

#3 SEMANTIC EXTENSIONS: Rejected.

#4 FLAGGER: Adopted.

318. sort-in-FLWR: Add ’order by’ clause to FLWR?

Locus: xquery Cluster: sort Priority: 1 Status: decided
Originator: Jonathan RobieAdd ’order by’ clause to FLWR?

Description

Four issues were raised in a proposal to restructure sorting in XQuery [1]:

89 of 135
1. Should we add an 'order by' clause to FLWR?

The following syntax has been proposed:

```plaintext
FLWRexpr ::= (ForClause | LetClause)+ SortClause? WhereClause?
"return" Expr
OrderClause ::= "order by" stable? SortSpecList
SortSpecList ::= Expr SortModifier ("," SortSpecList)?
SortModifier ::= ("ascending" | "descending")? ("empty" "greatest" | "empty" "least")?
```

The OrderClause sorts the tuple stream based on the conditions specified in the SortSpecList.

In the status quo, 'sortby' is a standalone postfix expression. FLWR is used to iterate, and 'sortby' is used to sort. This causes certain difficulties, because iteration and sorting are not distinct, unrelated operations - in general, the order in which the output sequence is ordered determines the best order to choose when iterating over the input.

When sorting data created with element constructors, it can sometimes be quite tricky to determine the original source of data in a constructed element. The more complex the expressions that construct the element, the more tricky this becomes. It is also tricky to iterate in an order determined by data that is not returned in a generated sequence.

If we add this clause, should a FLWOR expression that contains an OrderClause but no ForClause result in a semantic error, since there is no tuple stream to sort?

2. If we add an 'order by' clause, should we keep the sortby() expression, or remove it from our language?

Keeping it is convenient for some expressions. Removing it leaves us with a simpler language, and does not require us to explain to our users why we have two ways of doing the same thing.

3. How should we formalize 'order by' - or should we formalize it in Version 1.0?

The most straightforward way to formalize 'order by' is to use tuples, which do not exist in our Data Model, and these would cause significant change to our Formal Semantics. However, the semantics of 'order by' are straightforward.

Our options seem to be:

3.a. Ensure that we know how it would be formalized using tuples, but postpone including this in the Formal Semantics until after Version 1.

3.b. Refuse to add the feature unless it can be formalized with our current Data Model.

3.c. Restrict the feature to functionality easily formalized with our current Data Model.

I believe there was significant enthusiasm for 3.a. in today’s telcon.

4. Is the 'order by' clause part of the XPath spec, or is it only in XQuery?

Jonathan


Interactions and Input

Actual Resolution

Decision by: xpath-tf on 2002-10-16 ([link to member only information] )

'order by' has been added.

319. namespace-definitions-and-validate: Namespace definitions and
in-scope namespaces

Locus: xquery Cluster: namespaces Priority: 1 Status: active
Originator: XPath TF Namespace definitions and in-scope namespaces

Description

Addition of namespaces in the query prolog to the in-scope namespaces; how is this information carried through to provide input to validation?

Additional aspects are:

(a) the whole question of how the namesapce context is affected by namespace declarations in element constructors

(b) the general notion (in XQuery, specifically), that the static context can vary for different parts of a query

(c) what information gets through to act as input to validation

320. types-conformancelevels-same-results: Should different conformance levels give the same result for the same query and data?

Issue Class: T Locus: xquery Cluster: types Priority: 1 Status: decided
Originator: Anders Berglund Should different conformance levels give the same result for the same query and data?

Description

Are different conformance levels going to give the same results? Is it possible to get different results for the same query? The principle should be that you either get the same result or a failure. Currently this isn’t the case.

Interactions and Input

[link to member only information] Anders Berglund:

[link to member only information] Kristoffer Rose:

Actual Resolution

Decision by: proc-mod-tf on 2003-02-11 ([link to member only information] )

Decision by: xquery on 2003-02-19 ([link to member only information] )

Given that types add semantics, there is no way to achieve the proposed principle between basic XQuery and schema import without adding additional specification in the query.

It is decided not to add any such functionality and close this issue without action.

321. validate-lax-strict: Is validate strict or lax?

Locus: xquery Cluster: validate Priority: 1 Status: decided
Originator: XSL/XQuery joint Is validate strict or lax?

Description

Does "validate" do strict or lax validation?

Actual Resolution
322. **validate-lax-strict-override**: "validate" strict/lax override?

**Locus:** xquery  **Cluster:** validate  **Priority:** 1  **Status:** decided

**Originator:** XSL/XQuery joint "validate” strict/lax override?

**Description**

Should the user should be able to override which type (strict/lax) of validation "validate" does?

**Actual Resolution**

Decision by: xquery on 2002-12-18 ([link to member only information] )

Decision to accept the proposal of Phil Wadler in http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Dec/0378.html.

325. **element-def-not-in-in-scope**: Refering to element that is not in the in-scope schema def.

**Issue Class:** T  **Locus:** xpath  **Cluster:** types  **Priority:** 1  **Status:** decided

**Originator:** XQuery Refering to element that is not in the in-scope schema def.

**Description**

In recent discussions on the type system and when reviewing the documents I noticed that many people interpret

```
<element foo>
```

differently than the spec indicates it needs to be interpreted. This seems to indicate that we have a problem.

The semantics that people expect seems to be:

- `element foo` is the same as `element foo of type xs:anyType` and matches all elements with the given name (regardless where or whether the element was declared in the schema).
- `attribute foo` is the same as `attribute foo of type xs:anySimpleType` and matches all attributes with the given name regardless where or whether it was declared in the schema).

However our spec [1] says:

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:

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.

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.

The constraint that an element must have a required name is considered to be satisfied if the element has been validated and found to be a member of a substitution group whose head element has the required name. Substitution groups are described in [XML Schema].

which seems to indicate that this is not the case.

I would like to open an issue on this. I think even if people do not want to use schema types, they may still want to restrict the argument type of a function to an element with a given name.

In addition, many people are surprised to see the SchemaContext in the SequenceType production for other usages than for validate in ...

While there is a use case, I wonder whether this needs to be a required feature for XQuery V1 given the leap in complexity. Maybe it would be useful to solve the above problem by saying that you have to use

```
  element foo in /
```

to force the name to be in the in-scope schema definition? This would mean that / becomes allowed in the SchemaGlobalContext production.

**Actual Resolution**

Decision by: xpath-tf on 2002-12-17 ([link to member only information])

Decision by: xquery on 2002-12-19 ([link to member only information]) Joint F2F.

Decision by: xsl on 2002-12-19 ([link to member only information]) Joint F2F.

In sequence type it is an error, it is valid in a path expression.

**327. functions-parameter-evaluation: Evaluate unused function parameters?**

Locus: xpath Cluster: functions Priority: 1 Status: decided
Originator: XSL/XQuery joint Evaluate unused function parameters?

Description

Is an implementation obligated to evaluate all its parameters (and raise any errors doing so) even when they’re not needed in the function body?

**Actual Resolution**

Decision by: xpath-tf on 2002-10-29 ([link to member only information]) as part of Agendum 1.

There is no need to evaluate "non needed" expressions.

Decision by: xquery on 2002-10-30 ([link to member only information]) accepting text for next publication, but keep issue active.

Decision by: xsl on 2002-10-31 ([link to member only information]) accepting text for next publication, but keep issue active.

Decision by: xpath-tf on 2003-03-18 ([link to member only information])
No adverse public comments received. Close issue with no change.

328. **cdata-section**: What does CDATA section constructor construct?

**Locus:** xquery  **Cluster:** cdata section  **Priority:** 1  **Status:** decided

**Originator:** Michael Kay

What does CDATA section constructor construct?

**Description**

Section 3.7.5 describes CDATA section constructor, but these cannot be represented in the data model.

**Actual Resolution**

Decision by: xquery on 2002-12-11 ([link to member only information])

Decided to close, with document changes as described by Mike Kay in [http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Dec/0095.html](http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Dec/0095.html), including editorial comments.

329. **duplicate-attribute-constructors**: Duplicate attribute constructors

**Locus:** xquery  **Cluster:** constructors  **Priority:** 1  **Status:** decided

**Originator:** Michael Kay

Duplicate attribute constructors

**Description**

If there are multiple constructors for the same attribute on an element; which one is taken or is it an error?

**Actual Resolution**

Decision by: xquery on 2002-12-11 ([link to member only information])

Decided to make it an error for an element constructor to specify two attributes with the same name. Error to be documented in XQuery Section 3.7.4.1.

334. **xquery-failed-validation**: How are documents for which validation has failed processed?

**Issue Class:** T  **Locus:** xpath  **Cluster:** type-semantics  **Priority:** 1  **Status:** decided

**Originator:** Jonathan Robie

How are documents for which validation has failed processed?

**Description**

What is the result of a failed validation? Can you inspect the result to detect this? Do you just get anySimpleType and anyType?

**Interactions and Input**

**Actual Resolution**

Decision by: xpath-tf on 2003-03-04 ([link to member only information])

Decision by: xsl on 2003-03-07 ([link to member only information]) Joint meeting

Decision by: xquery on 2003-03-07 ([link to member only information]) Joint meeting

Access to documents that fail validation is not supported.
335. **semantics-interfering-with-optimization**: XPath/XQuery’s current semantics greatly interferes with optimization

**Locus:** xpath  **Cluster:** formal-semantics  **Priority:** 1  **Status:** decided  
**Originator:** Michael Rys

XPath/XQuery’s current semantics greatly interferes with optimization

**Description**

Current semantics basically defines the semantics by mapping FLWRs and path expression to go top-down. Errors are normative. Problem is that we can not apply many optimizations. Simple query rewrites such as pushing or pulling filters, etc. Like to be able to push predicates down and evaluate them. Potentially the predicate might raise an error you would not have gotten if you processed top down. Want to allow implementations to do bottom up evaluations.

**Suggested resolution:**

The formal semantics defines dynamic evaluation in terms of a naive, top-down reduction of a core expression to a data-model value. Implementations may choose alternative evaluation strategies, which, for example, may reduce a core expression bottom-up. If an evaluation of a core expression yields a value (i.e., it does not raise an error), the value must be the same value as would be produced by the dynamic semantics defined in this document. The evaluation of a core expression may raise an error that may not be raised by the dynamic semantics as defined in this document.

**Actual Resolution**

Decision by: xpath-tf on 2002-10-29 ([link to member only information]) as part of Agendum 1.

Expressions, with the exception of “if”, may be reordered and thus some errors may be raised that using another evaluation strategy may not have occurred.

Decision by: xquery on 2002-10-30 ([link to member only information]) accepting text for next publication, but keep issue active.

Decision by: xs1 on 2002-10-31 ([link to member only information]) accepting text for next publication, but keep issue active.

Decision by: xpath-tf on 2003-03-18 ([link to member only information])

No adverse public comments received. Close issue with no change.

338. **whitespace-character-reference**: Handling of whitespace and character references

**Locus:** xquery  **Cluster:** whitespace  **Priority:** 1  **Status:** decided  
**Originator:** Michael Kay

Handling of whitespace and character references

**Description**

The effect of CharRef in Element or Attribute content is underspecified. For example, if the value of the CharRef is whitespace, does it behave like whitespace in Element or Attribute content, or does it behave like ordinary characters? Either way, it is hard to make it behave exactly like the XML construct that it mimics: in XML, a character reference such as suppresses certain effects such as whitespace normalization. It is not clear whether such normalization happens in XQuery.

**Interactions and Input**

**Actual Resolution**

Decision by: xquery on 2002-12-04 ([link to member only information])

Confirmed that Mary’s proposal for whitespace, that was accepted 2002-11-27, closes this issue.
Resolved by the recent adoption of the revised text on attribute constructors.

339. **element-attribute-construction-order**: Error type for attributes constructed too late

Locus: **xquery** Cluster: **errors** Priority: 2 Status: **decided**
Originator: **Don Chamberlin**

**Description**

What kind of error should be raised by an element constructor in which an attribute is encountered after other element content?

**Actual Resolution**

This a type error. Type error to be documented in XQuery Section 3.7.2.

340. **errors-unique-identification**: How to identify errors?

Locus: **xpath** Cluster: **errors** Priority: 1 Status: **decided**
Originator: **XPath TF**

**Description**

How should we allocate codes or identifiers to errors defined in the spec? We should not use "explanatory sentences" as these are not appropriate for I18N reasons.

**Actual Resolution**

(Not really an issue); Ashok and Norm will investigate whether we can editorially markup errors consistently.

Issue should remain open pending proposal on: ACTION XPATH-091-11 Mary adds herself to XPATH-091-11.

1. All occurrences of static and dynamic errors should be identified in the F&O, DM, language, and FS specs using distinguished error markup.

2. Error markup should include unique alpha-numeric identifier for each error which permits the source to refer to unique error definitions and to include reverse indices on errors.

3. In first cut, alpha-numeric error codes will not be revealed in printed documents, but we can reveal them at a later time if we decide to make error codes part of conformance requirements.

4. Until we consider an API for conveying errors to the evaluation environment, we recommend *not* requiring implementations to return specific error codes (and possibly additional context info such as location of error in query body or name of built-in function, etc.)
341. **SequenceType-problems**: Problems with SequenceType

**Locus:** xpath **Cluster:** syntax **Priority:** 1 **Status:** decided
Originator: Michael Kay Problems with SequenceType

**Description**

Using the "instance of SequenceType" construct within a predicate of a path expression, or in an XSLT match pattern, is very unwieldy: the verbose English-like syntax of a SequenceType does not go well with the terse syntax of path expressions, leading to inelegant constructs like

```
select="//*[.instance of element of type address]"
```

**Interactions and Input**

[link to member only information] Michael Kay:

**Proposed Resolution**

1. **Motivation**

Many people have asked for the ability to test for elements of a given type in path expressions, and to be able to write path expressions that would retrieve nodes with types that match the declared types of function parameters. For instance, consider the following query in the syntax of our current Working Draft:

```
define function name($e as element person) returns element name
  {
    $e/name
  }
for $p in input()//person
  return name($p)
```

This query can fail at runtime, because the path expression matches any element named person, and the function parameter requires a valid element conforming to a globally declared element named person.

This proposal adds ElementType and AttributeType nodetests to path expressions, and aligns the syntax of SequenceType with the syntax of type tests done in path expressions. For instance, in the following syntax, the KindTest "element(person)" matches only person elements that are valid instances of the globally declared person element, and the same test is used for the function’s parameter type:

```
define function name($e as element(person)) returns element name
  {
    $e/name
  }
for $p in input()//element(person)
  return name($p)
```

Using the same syntax and semantics for these tests in both path expressions and SequenceType ensures that they are well aligned, and reduces the number of expressions users of the language must learn. In addition, it allows us to fix some problems with SequenceType that have made the grammar complex to maintain.

ElementType and AttributeType nodetests also allow the names of types to be used to select nodes, eg:

```
define function name($e as element(person, surgeon)) returns element name
  {
    $e/name
  }
for $p in input()//element(person, surgeon)
  return name($p)
```
Or:

```xml
for $date in input()//element(*, xs:date)
  return <date>{ $date }</date>
```

Note that the above queries provide enough type information to be
statically valid without ‘treat as’. If a NameTest were used instead
of the ElementType nodetest, the above query would not be statically
valid, and the query writer would need to use treat as to make it so.

Using this proposal, the most convenient way to match locally declared
types in a path expression is to use the globally declared types in
which they are contained to establish a type context; eg, the
following query from use case STRONG is statically valid as modified
below:

```xml
define function names-match(
  $s as element(ipo:purchaseOrder/ipo:shipTo),
  $b as element(ipo:purchaseOrder/ipo:billTo)
  as xs:boolean
{
  $s/ipo:name = $b/ipo:name
})
```

```xml
for $p in document("ipo.xml")//element(ipo:purchaseOrder)
  where not( names-match( $p/ipo:shipTo, $p/ipo:billTo ) )
  return $p
```

ElementType and AttributeType nodetests can also be used in match
patterns on XSLT stylesheets, eg:

```xml
<xsl:template match="element(person)">
  <xsl:template match="attribute(*,xs:date)">
```

For more examples of ElementType and AttributeType nodetests in match
patterns, see [1].

This proposal steals liberally from proposals by Phil Wadler, Jeni
Tennison, Michael Rys, and Mike Kay. Don Chamberlin gave detailed
review and pointed out important problems and their solutions. Scott
Boag tested the grammar.

========================================================================
2. Examples of ElementTypeTest and AttributeTypeTest

Informally, the principles for matching an ElementTypeTest or
AttributeTypeTest are as follows:

- `element()` matches elements, `attribute()` matches attributes.

- names in ElementTypeTest and AttributeTypeTest test the name of a
  node. If the name of an element identifies a Corresponding Element
  Declaration which is the head of a substitution group, the name of
  an ElementTypeTest also matches other names in the substitution
  group.

- types in ElementTypeTest and AttributeTypeTest match type
  annotations. If no type is supplied, there must be a Corresponding
  Element Declaration or a Corresponding Attribute Declaration, which
  is used to nominate a type.

The following examples illustrate these semantics by stating what
nodes an ElementTypeTest would match.

```xml
element()
element(*)
element(*,*)
```

Matches any element. These three forms are equivalent.

```xml
element( person )
```

Matches any element named ‘person’ with the type corresponding to
the globally declared ‘person’ element; if this element is
nullable, also matches any empty element named person containing an
xsi:nil attribute with the value ‘true’. If the globally declared
‘person’ element is the head of a substitution group, also matches
element names from the substitution group. If there is no globally
declared ‘person’ element, a type error is raised.

```xml
element( person, * )
```
Matches any element named person or any element whose name is in the substitution group of the globally declared element person, if one exists. This is equivalent to the earlier ~person syntax, which we feel is no longer needed.

element( person, personType nillable )

Matches any element named ‘person’ with the type ‘personType’, including any empty element named person containing an xsi:nil attribute with the value ‘true’. If the globally declared ‘person’ element is the head of a substitution group, also matches element names from the substitution group.

element( *, xs:integer nillable )

Matches any element of type xs:integer, including any such element with xsi:nil="true" and no content.

element( foo/bar/person )

Matches any element named ‘person’ with the type of the corresponding element declaration.

element( foo/bar/person, personType )

Syntax Error.

attribute()

Matches any attribute.

attribute( foo/bar/person/@id )

Matches any attribute named ‘id’ with the type of the corresponding attribute declaration.

element( @*, xs:integer )

Matches any attribute of type xs:integer.

attribute( @price, xs:integer )

Matches any attribute named price of type xs:integer.

========================================================================

3. Syntax

Tokens
Scott currently has the following as tokens, and is working on making them productions:

```
[22]  SchemaGlobalContext ::= QName | ("type" "(" QName ")")
[23]  SchemaContextStep ::= QName

Productions
[112]  SequenceType ::= (SequenceSingleType OccurrenceIndicator?) | <"empty" "(* *)">
[113]  SequenceSingleType ::= AtomicType | ItemType
[114]  AtomicType ::= QName
[115]  ItemType ::= KindTest | <"item" "(" *)">
[116]  KindTest ::= DocumentTest | ElementTypeTest | AttributeTypeTest | ProcessingInstructionTest | CommentTest | TextTest | AnyKindTest

[117]  DocumentTest ::= <"document" "(" *)">
[118]  ElementTypeTest ::= <"element" "(" ((SchemaContextPath LocalName) | (NodeName ("," TypeName "nillable"?)?) ? ")")
[119]  AttributeTypeTest ::= <"attribute" "(" ((SchemaContextPath "@" LocalName) | ("@" NodeName ("," TypeName)?)) ? ")")

[120]  SchemaContextPath ::= <SchemaGlobalContext "/"><SchemaContextStep "/">*
[121]  LocalName ::= QName
[122]  OccurrenceIndicator ::= *** | *+ | **+
```
4. Semantics

A ContextName identifies a valid path from a globally declared element declaration or type definition in the in-scope schema definitions. For instance, the following indicates a shipTo element on a purchaseOrder:

    purchaseOrder/shipTo

If a ContextName begins with type(t), then the path starts with a global named type in the in-scope schema definitions. For instance, the following indicates a shipTo element in a complex type called purchaseOrderType:

    type(purchaseOrderType)/shipTo

The rules for matching an ElementTypeTest depend on the Corresponding Element Declaration, which is determined as follows:

An ElementName is the first argument of an ElementTypeTest.

If the ElementName has the form NodeName, and it matches the name of a globally declared element declaration in the in-scope schema definitions, it is the "Corresponding Element Declaration". If the ElementName has the form SchemaContextPath LocalName, then the element declaration which would correspond to an element named LocalName in the given context is the "Corresponding Element Declaration".

The rules for matching an AttributeTypeTest depend on the Corresponding Attribute Declaration,

An AttributeName is the first argument of an AttributeTypeTest.

If the AttributeName has the form NodeName, and it matches the name of a globally declared element declaration in the in-scope schema definitions, it is the "Corresponding Element Declaration". If the AttributeName has the form SchemaContextPath LocalName, then the element declaration which would correspond to an element named LocalName in the given context is the "Corresponding Attribute Declaration".

An ElementTypeTest matches a node if the node is an element, the element's name matches the ElementName of the ElementTypeTest, and the element's type matches the type and nillability of the ElementTypeTest.

An ElementName is matched as follows:

If the ElementName is absent or "+", it matches an element of any name.

If the ElementName has the form NodeName, it matches an element if the expanded-QNames of the element and the ElementName match. If the ElementName has the form SchemaContextPath LocalName, it matches an element if the expanded-QNames of the element and the LocalName match.

If the Corresponding Element Declaration is the head of a substitution group, then the ElementName also matches an element if the expanded-QNames of the element and any element in the substitution group match.

Otherwise, the ElementName does not match the element's name.

An element’s TypeName is matched as follows:

If the TypeName is "+", it matches an element of any type.

If the TypeName is absent, then the default TypeName is selected, which is the type of the Corresponding Element Declaration. The default nillability is also selected, which is the nillability of the Corresponding Element Declaration.

If the expanded-QName of the TypeName matches the element’s type or any of its subtypes, then the TypeName matches the element’s type if the type nillability is also satisfied.

If the element’s type name is an anonymous local type, the TypeName must be a default type chosen from the Corresponding Element Declaration, and they match if the element’s type is the same as or derived by restriction or extension from the type of the TypeName, provided the type nillability is also satisfied.

An element’s type nillability is not satisfied when the type is not
nillable, and the element's dm:nil property is 'true'; otherwise, it is satisfied.

The semantics of AttributeTypeTest are parallel, substituting the word 'attribute' for 'element' in the above description and eliminating substitution groups and nillable, which do not apply to attributes. An AttributeTypeTest matches a node if the node is an attribute, the attribute's name matches the AttributeName of the AttributeTypeTest, and the attribute's type matches the type and nillability of the AttributeTypeTest.

An AttributeName is matched as follows:

If the AttributeName is absent or "***", it matches any attribute.

If the AttributeName has the form NodeName, it matches an element if the expanded-QNames of the element and the AttributeName match. If the AttributeName has the form SchemaContextPath LocalName, it matches an element if the expanded-QNames of the element and the LocalName match.

Otherwise, the AttributeName does not match the attribute's name.

An attribute's TypeName is matched as follows:

If the TypeName is "***", it matches an attribute of any type.

If the TypeName is absent, then the default TypeName is selected, which is the type of the corresponding attribute declaration.

If the expanded-QName of the TypeName matches the attribute's type or any of its subtypes, then the TypeName matches the attribute.

If the attribute's type name is an anonymous local type, the TypeName must be a default type chosen from the Corresponding Attribute Declaration, and they match if the attribute's type is the same as or derived by restriction or extension from the type of the TypeName.

6. attribute() and default axes

To allow attribute() to be used in path expressions, abbreviated steps need to be modified as follows:

Current text:

>     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.

Proposed text:

>     The most important abbreviation is that child:: can be omitted
>     from a step, or attribute:: can be omitted from a step that
>     contains an AttributeTypeTest. In most cases, child is the
>     default axis. For example, a path expression section/para is
>     short for child::section/child::para. However, if the step
>     contains an AttributeType, the default axis is attribute::. For
>     example, child::section/attribute( id ) is short for
>     child::section/attribute::attribute( id ).

7. Static Semantics

To be supplied.

8. Issues

Issue:

Should a WildCard be allowed for the names of attributes and elements? For instance, one might imagine wanting to select only HTML elements of type xs:string:

element(html:* , xs:string)
Actual Resolution

Decision by:xpath-tf on 2003-03-04 ([link to member only information])

Decision by: xsl on 2003-03-06 ([link to member only information]) Joint meeting

Decision by: xquery on 2003-03-06 ([link to member only information]) Joint meeting

Proposal accepted resolving issue.

343. namespaces-functions: Do functions in the null namespace clash with functions in the default namespace?

Locus: xpath Cluster: namespaces Priority: 1 Status: active
Originator: XPath TF Do functions in the null namespace clash with functions in the default namespace?

Description

If you define the DFN (default function namespace) to be the null namespace, which you must do if you are going to call functions in this namespace, can you still use unprefixed names to refer to functions in the built-in namespace?

Interactions and Input

Actual Resolution

Decision by: xquery on 2003-01-22 ([link to member only information]) A static error is raised when there is a clash between a user-defined function in the default namespace and any other function of the same name in the default namespace.

437. FS-Issue-0094: Static type errors and warnings

Locus: formal-semantics Cluster: Semantics Priority: 1 Status: decided
Originator: Don Chamberlin Static type errors and warnings

Description

Static type errors and warnings are not specified. We need to enumerate in both the [XPath/XQuery] and formal semantics documents what kinds of static type errors and warnings are produced by the type system. See also [resolved issue #FS-Issue-0090].

Actual Resolution

Decision by: fs-editors on 2003-01-12 ([link to member only information])

Dec. FS working draft describes the semantics of errors.

441. FS-Issue-0098: Implementation of and conformance levels for static type checking

Locus: formal-semantics Cluster: Semantics Priority: 1 Status: decided
Originator: Don Chamberlin Implementation of and conformance levels for static type checking

Description

This issue is related to [resolved issue #FS-Issue-0059] 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 [XPath/XQuery], in which some processors
(or some processing modes) are more permissive about types. This would allow [XPath/XQuery] implementations that do not understand all of Schema, and it would allow customers some control over the cost/benefit tradeoff of type checking.

**Actual Resolution**

Decision by: **xquery** on 2003-02-26 ([link to member only information] )

Close issue 441 with no further work.

443. **FS-Issue-0100**: Namespace resolution

**Locus**: formal-semantics  **Cluster**: Semantics?  **Priority**: 1  **Status**: active

**Originator**: FS Editors Namespace resolution

**Description**

The way (when? where?) namespace prefixes are resolved is still an open issue.

444. **FS-Issue-0101**: Support for mixed content in the type system

**Locus**: formal-semantics  **Cluster**: Priority: 1  **Status**: decided

**Originator**: FS Editors Support for mixed content in the type system

**Description**

Support for mixed content in the type system is an open issue. This reopens issue [resolved issue #FS-Issue-0016]. Dealing with mixed content with interleaving raises complexity issue. See also #FS-Issue-0103.

**Actual Resolution**

Decision by: **xsl** on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: **xquery** on 2003-03-07 ([link to member only information] )Joint meeting

The formal semantics is a specification of the behavior of an XQuery processor, not a recipe for implementation. The uses of interleaving in the specification are sufficiently restricted that a practical implementation can be written.

446. **FS-Issue-0103**: Complexity of interleaving

**Locus**: formal-semantics  **Cluster**: Typing  **Priority**: 1  **Status**: decided

**Originator**: FS Editors Complexity of interleaving

**Description**

The current type system allows interleaving is allowed on arbitrary types. Interleaving is an expensive operation and it is not clear how to define subtyping for it. Should we restrict use of interleaving on (optional) atomic types? Should this restriction reflects the one in XML schema? Related to [resolved issue #FS-Issue-0077].

**Actual Resolution**

Decision by: **xsl** on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: **xquery** on 2003-03-07 ([link to member only information] )Joint meeting

The formal semantics is a specification of the behavior of an XQuery processor, not a recipe for implementation. The uses of interleaving in the specification are sufficiently restricted that a practical
implementation can be written.

449. **FS-Issue-0106**: Constraint on attribute and element content models

**Locus:** formal-semantics  **Cluster:** Typing  **Priority:** 1  **Status:** decided  
**Originator:** Jerome  
**Constraint on attribute and element content models**

**Description**

The [XPath/XQuery] type system allows more content model than what XML Schema allows. For instance, the current type grammar allows the following types:

```
  element d { (attribute a | element b, attribute c)* }  
  attribute a { element b }  
```

Section #FS-sec_types indicates corresponding constraints on the [XPath/XQuery] type system to avoid that problem. The status of these constraints is unclear. When are they enforced and checked?

**Actual Resolution**

Decision by: xsl on 2003-03-07 ([link to member only information])  
Decision by: xquery on 2003-03-07 ([link to member only information])  
Closed following the semantics of element constructors agreed at the December f2f meeting.

450. **FS-Issue-0107**: Semantics of data()

**Locus:** formal-semantics  **Cluster:** Semantics  **Priority:** 1  **Status:** decided  
**Originator:** FS Editors  
**Semantics of data()**

**Description**

What is the semantics of data() applied to anything else than an element or attribute node?

**Proposed Resolution**

XQuery 1.0 and XPath 2.0 Data Model  
W3C Working Draft 17 January 2003  
http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2003Jan/0251.html

4.1.5 string-value Accessor

```
4.1.6 typed-value Accessor
```

1. REPLACE entire section WITH:

```
Every node has a string value; the way in which the string value is computed is different for each kind of node and is specified in the sections on nodes below.
```

2. REPLACE:

```
Otherwise, xs:anyType for elements or xs:anySimpleType for attributes.
WITH:
Otherwise, xs:anyType for elements or xdt:untypedAtomic for attributes.
```

* If the item is an element node with type xs:anyType, then its typed value is equal to its string value, as an instance of xdt:untypedAtomic.

WITH:
* If the node is an element node with type xs:anyType, then its typed value is equal to its string value, as an instance of xdt:untypedAtomic.

3. REPLACE:

* The typed value of an element node with a simple type is derived from its string value and type in a way that is consistent with XML Schema validation.

WITH:

* If the node is an element node with a simple type or with a complex type of simple content, then its typed value is derived from its string value and type in a way that is consistent with XML Schema validation.

4. ADD:

* If the item is an element node with complex type of empty content, then its typed value is derived from its string value and type in a way that is consistent with XML Schema validation.

* If the node is an element node with a complex type of mixed content, then its typed value is its string value as an instance of xdt:untypedAtomic.

5. REPLACE:

* If the item is an element node with complex content, dm:typed-value returns the empty sequence.

WITH:

* If the item is an element node with complex type of complex content, then its typed value is the empty sequence.

*************************************************

XQuery 1.0 and XPath 2.0 Functions and Operators
W3C Working Draft: 10 December 2002
http://lists.w3.org/Archives/Member/w3c-archive/2002Dec/att-0035/02-xquery-operators.html

2.4 fn:data
-----------

1. REWRITE entire section as described in:
   http://lists.w3.org/Archives/Member/w3c-xsl-query/2002Nov/0391.html

This text incorporates changes for nodes with complex type of mixed content:

fn:data($srcval as item*) as atomic value*

The fn:data function takes a sequence of items and returns a sequence of atomic values.

The result of fn:data is the sequence of atomic values produced by applying the following rules to each item in $srcval:

If the item is an atomic value, it is returned.

If the item is a node, fn:data returns the typed value of the node, as defined by the accessor function dm:typed-value defined for that kind of node in [XQuery 1.0 and XPath 2.0 Data Model].

If the item is a comment, document, namespace, processing-instruction, or text node, then its typed value is equal to its string value, as an instance of xdt:untypedAtomic.

If the item is an attribute node with type annotation xs:atomicSimpleType, then its typed value is equal to its string value, as an instance of xdt:untypedAtomic. The typed value of any other attribute node is derived from its string value and type annotation in a way that is consistent with XML Schema validation, as described in [XQuery 1.0 and XPath 2.0 Data Model].

If the item is an element node whose type annotation denotes a complex type of empty content, then its typed value is equal to the empty sequence.

If the item is an element node whose type annotation is xs:anyType or denotes a complex type of mixed content, then its typed value is equal to its string value, as an instance of xs:untypedAtomic.

If the item is an element node whose type annotation denotes a simple type or a complex type of simple content, then its typed
value is derived from its string value and type in a way that is consistent with XML Schema validation, as described in [XQuery 1.0 and XPath 2.0 Data Model].

If the item is an element node whose type annotation denotes a complex type with complex content (i.e., a type that permits subelements), fn:data raises a type error.

[ Ed note: The typed-value accessor returns an empty sequence when applied to an element node with complex type of complete content, whereas fn:data() raises a type error, because type errors are a characteristic of the language, not the data model. ]

1. ADD definition of fn:data() as given above following:

The typed value and string value for each kind of node are defined by the dm:typed-value and dm:string-value accessors in [XQuery 1.0 and XPath 2.0 Data Model].

The normative definitions are in the [XQuery 1.0 and XPath 2.0 Data Model] and is repeated here for convenience:

1. This section is entirely rewritten to align with new semantics.

Introduction

The fn:data function converts a sequence of items to a sequence of atomic values.

Notation

Inferring the type for the fn:data function is done by applying the fn:data function as a Filter, using the same approach as for the XPath steps.

\[
\text{statEnv} |- \text{fn:data on Type}_1 : \text{Type}_2
\]

Static Type Analysis

The general rule for the function fn:data is to apply the filter \([\text{fn:data on}]\) to the prime type of its argument type, then apply the quantifier to the result:

\[
\begin{align*}
\text{statEnv} |- \text{Expr} : \text{Type} \\
\text{statEnv} |- [\text{fn:data on}] \text{prime(Expr)} : \text{Type}_1 \\
\end{align*}
\]

\[
\text{statEnv} |- \text{fn:data(Expr)} : \text{Type}_1 \circ \text{quantifier(Expr)}
\]

The rules for the \([\text{fn:data on}]\) filter always take a prime type (i.e., a union of item types) as an argument.

When applied to none, \([\text{fn:data on}]\) yields none.

When applied to the union of two types, \([\text{fn:data on}]\) is applied to each of the two types. The resulting types are combined into a factored type. This rule is necessary because \([\text{fn:data on}]\) may return a sequence.
of atomic types.

\[
\begin{align*}
\text{statEnv} & \vdash \text{[fn:data on]} \text{Type1 : Type3} \\
\text{statEnv} & \vdash \text{[fn:data on]} \text{Type2 : Type4}
\end{align*}
\]

When applied to an atomic type, the \text{[fn:data on]} filter simply returns the atomic type:

\[
\begin{align*}
\text{Type} & \vdash \text{xdt:anyAtomicType} \\
\text{statEnv} & \vdash \text{[fn:data on]} \text{Type} : \text{Type}
\end{align*}
\]

When applied to comment, processing instruction, text, and document node types, the \text{[fn:data on]} filter returns \text{xdt:untypedAtomic}.

\[
\begin{align*}
\text{Type} & \vdash \text{comment | processing-instruction | text | document} \\
\text{statEnv} & \vdash \text{[fn:data on]} \text{Type} : \text{xdt:untypedAtomic}
\end{align*}
\]

When applied to attribute node types with type annotation \text{xs:anySimpleType} or element node types with type annotation \text{xs:anyType}, the \text{[fn:data on]} filter returns \text{xdt:untypedAtomic}.

\[
\begin{align*}
\text{statEnv} & \vdash \text{AttributeType static lookup (of type xs:anySimpleType)} \\
\text{statEnv} & \vdash \text{[fn:data on]} \text{AttributeType} : \text{xdt:untypedAtomic} \\
\text{statEnv} & \vdash \text{ElementType static lookup (of type xs:anyType)} \\
\text{statEnv} & \vdash \text{[fn:data on]} \text{AttributeType} : \text{xdt:untypedAtomic} \\
\text{statEnv} & \vdash \text{ElementType static lookup (of type TypeName)} \\
\text{not(TypeName = xs:anySimpleType)} \\
\text{statEnv} & \vdash \text{(of type TypeName) expands to Type} \\
\text{statEnv} & \vdash \text{[fn:data on]} \text{AttributeType} : \text{Type} \\
\text{statEnv} & \vdash \text{ElementType static lookup TypeReference TypeReference expands to Type Type <= (Type1, Type2)} \\
\text{Type1 <= AttributeType*} \\
\text{Type2 <= xs:anySimpleType}
\end{align*}
\]

When applied to an element type whose type annotation denotes a simple type or a complex type of simple content or a complex type of empty content, the \text{[fn:data on]} filter returns the element’s simple type.

\[
\begin{align*}
\text{statEnv} & \vdash \text{ElementType static lookup TypeReference} \\
\text{statEnv} & \vdash \text{TypeReference expands to Type Type <= (Type1, Type2)} \\
\text{Type1 <= AttributeType*} \\
\text{Type2 <= xs:anySimpleType}
\end{align*}
\]

When applied to an element type whose type annotation denotes a complex type of mixed content, the \text{[fn:data on]} filter returns \text{xdt:untypedAtomic}.

\[
\begin{align*}
\text{statEnv} & \vdash \text{ElementType static lookup (of type TypeName)} \\
\text{statEnv.typeDefn(TypeName) => define type TypeName Derivation? Mixed ( Type1? )} \\
\text{statEnv} & \vdash \text{[fn:data on]} \text{ElementType} : \text{xdt:untypedAtomic}
\end{align*}
\]

The \text{[fn:data on]} filter is not defined on any element type whose type annotation denotes a complex type of complex content and therefore raises a static error.

Dynamic Evaluation

The dynamic semantics of \text{fn:data()} are specified in [2.4 fn:data] of [XQuery 1.0 and XPath 2.0 Functions and Operators].
Proposal accepted.

452. **FS-Issue-0109**: Semantics of order by

*Locus:* formal-semantics  *Cluster:* Semantics  *Priority:* 1  *Status:* active  
*Originator:* Jerome  

**Description**

The precise semantics of order by is still an open issue.

453. **FS-Issue-0110**: Semantics of element and attribute constructors

*Locus:* formal-semantics  *Cluster:* Semantics  *Priority:* 1  *Status:* decided  
*Originator:* Jerome  

**Description**

The precise semantics of element constructors is still an open issue.

**Actual Resolution**

Decision by: xsl on 2003-02-27 ([link to member only information] ) Joint meeting

Decision by: xquery on 2003-03-07 ([link to member only information] ) Joint meeting

Closed following the semantics of element constructors agreed at the December f2f meeting.

455. **FS-Issue-0112**: Typing for the "typeswitch" default clause

*Locus:* formal-semantics  *Cluster:* Static typing  *Priority:* 1  *Status:* decided  
*Originator:* Jerome  

**Description**

There is an asymmetry in the typing for the default clause in typeswitch vs. the other case clauses. This results in a less precise type when the default clause can be applied.

It would be nicer to be able to have the type be more precise, like for the other case clauses.

The technical problem is the need for some form of negation. I think one could define a "non-common-primes" function that would do the trick, but I leave that as open for now until further review of the new typeswitch section is made.

**Actual Resolution**

Decision by: fs-editors on 2003-02-13 ([link to member only information] )

Decision by: xquery on 2003-02-19 ([link to member only information] )

Decision by: xsl on 2003-02-27 ([link to member only information] )

Accepting proposal in #FS-Issue-0173 resolves this issue.

Decision by: xsl on 2003-03-07 ([link to member only information] ) Joint meeting

Decision by: xquery on 2003-03-07 ([link to member only information] ) Joint meeting

Reaffirming decision.
457. **FS-Issue-0114**: Dynamic context for current date and time

Locus: formal-semantics Cluster: Semantics Priority: 1 Status: decided
Originator: Jerome

Dynamic context for current date and time

**Description**

The following components dynamic contexts have no formal representation yet: current date and time.

Related question: where are these context components used?

**Actual Resolution**

Decision by: xsl on 2003-03-07 (link to member only information) Joint meeting

Decision by: xquery on 2003-03-07 (link to member only information) Joint meeting

Add current date and time in the Dynamic Environment.

458. **FS-Issue-0115**: What is in the default context?

Locus: formal-semantics Cluster: Language Priority: 1 Status: active
Originator: Jerome

What is in the default context?

**Description**

What do the default namespace and type environments contain? I believe at least the default namespace environment should contain the "xs", "fn" and "op" prefixes, as well as the default namespaces bound to the empty namespace. Should the default type environment contain wildcard types?

459. **FS-Issue-0116**: Serialization

Locus: formal-semantics Cluster: Language Priority: 1 Status: decided
Originator: Jerome

Serialization

**Description**

Serialization of data model instances, and XQuery results is still an open issue.

**Actual Resolution**

Decision by: xsl on 2003-03-07 (link to member only information) Joint meeting

Decision by: xquery on 2003-03-07 (link to member only information) Joint meeting

Serialization is implemented in the data model.

461. **FS-Issue-0118**: Data model syntax and literal values

Locus: formal-semantics Cluster: Semantics Priority: 1 Status: decided
Originator: Phil Wadler

Data model syntax and literal values

**Description**

Phil suggests the data model should support primitive literals in their lexical form, in which case no explicit dynamic semantic rule would be necessary.
More generally, should the data model support a constructor syntax?

**Actual Resolution**

Decision by: xsl on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: xquery on 2003-03-07 ([link to member only information] )Joint meeting

The corresponding syntax is going to be implemented in the formal semantics.

472. **FS-Issue-0129**: Static typing of union

**Locus:** formal-semantics  **Cluster:** Static typing  **Priority:** 1  **Status:** decided

**Originator:** Michael Rys  **Static typing of union**

**Description**

What should be the semantics of arithmetics expressions over unions. Right now, it would raise a dynamic error. Do we want to raise a static error?

Should operators and functions consistenly with respect to typing?

With the current semantics in Section 4.5 expr1 + expr2 raises a static type error if (e.g.) expr1 has type string and expr2 has type integer. It raises only a dynamic error, if expr1 has type (string | integer) and expr2 has type integer, and expr1 actually evaluates to a string. An alternative would be that this raises also a static error, because it cannot be guaranteed to succeed on all instances.

**Proposed Resolution**

For user-defined functions there is no issue and the current static typing rules work fine.

There is an issue about static typing for built-in overloaded functions (arithmetic and comparison operators). In that case, the following algorithm is proposed:

If one or both of the types of the operands are a union, static typing is performed for each pair of item types in the union. If any of the cases raises a static type error, the whole static typing raises a static type error. Otherwise each resulting static type is unioned to form the final result type.

Here are examples for the expression (E1 + E2) in the case of various input types:

<table>
<thead>
<tr>
<th>E1</th>
<th>E2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>float</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>(float</td>
<td>int)</td>
<td>(float</td>
</tr>
<tr>
<td>(float</td>
<td>int)</td>
<td>(int</td>
</tr>
</tbody>
</table>

those cases did not change. In case there is a union:

```
E1 : (float | int) and E2 : int
then E1 + E2 : (float | int)
```

since both cases (int + int) and (float | int) could occur.

```
E1 : (float | int) and E2 : (int | string)
```

raises a static error since (int + string) raises a static error.

**Actual Resolution**

Decision by: xquery on 2003-02-12 ([link to member only information] )
Proposal accepted.

473. **FS-Issue-0130**: When to process the query prolog

**Locus**: formal-semantics  
**Cluster**: Semantics  
**Priority**: 1  
**Status**: active  
**Originator**: Jerome

**Description**

The query prolog needs to be processed before the normalization phase. This is not reflected yet in the processing model.

475. **FS-Issue-0132**: Typing for descendant

**Locus**: formal-semantics  
**Cluster**: Static typing  
**Priority**: 1  
**Status**: active  
**Originator**: Peter Fankhauser

**Description**

The current static typing for descendant is still under review and the inferences rules in that version are probably containing bugs.

478. **FS-Issue-0135**: Semantics of special functions

**Locus**: formal-semantics  
**Cluster**: Semantics  
**Priority**: 1  
**Status**: decided  
**Originator**: Michael Kay

**Description**

The current semantics does not completely cover built-in functions. Some functions used in the Formal semantics, or some functions from the XQuery 1.0 and XPath 2.0 Functions and Operators document need additional semantics specification.

**Actual Resolution**

Decision by: xsl on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: xquery on 2003-03-07 ([link to member only information] )Joint meeting

Ashok has posted a list of functions for which we need special typing rules:  
http://lists.w3.org/Archives/Member/w3c-query-editors/2003Feb/0027.html

It is now editorial just to implement those typing rules in the formal semantics document.

479. **FS-Issue-0136**: Non-determinism in the semantics

**Locus**: formal-semantics  
**Cluster**: Semantics  
**Priority**: 1  
**Status**: decided  
**Originator**: Mary Fernandez

**Description**

Some operations, such as logical operations and quantified operations are not deterministics ("early-out" semantics allowing not to evaluate all of the expressions). The formal semantics cannot capture the non-determinism in those operations.

**Actual Resolution**

Decision by: fs-editors on 2003-01-12 ([link to member only information] )
Dec. FS working draft describes non deterministic semantics.

481. **FS-Issue-0138**: Semantics of Schema Context

**Locus**: formal-semantics  **Cluster**: Semantics  **Priority**: 1  **Status**: active
**Originator**: Jerome Simeon  **Semantics of Schema Context**

**Description**

The semantics of Schema Context types in the SequenceType production is still an open issue.

482. **FS-Issue-0139**: Type equivalence rules

**Locus**: formal-semantics  **Cluster**: Semantics  **Priority**: 1  **Status**: decided
**Originator**: Michael Rys  **Type equivalence rules**

**Description**

Should we add back equivalence rules for types (e.g., $T_1^++ = T_1^*$ or $(T_1 | T_1) = T_1$). They are useful in practical implementations (e.g., to print an inferred type or reduce complexity of inferred types), and could be added in an appendix.

**Actual Resolution**

Decision by: xquery on 2003-02-12 ([link to member only information] )

Decided to add back equivalence rules.

484. **FS-Issue-0141**: Treatment of nillability and xsi:nil

**Locus**: formal-semantics  **Cluster**: Semantics  **Priority**: 1  **Status**: decided
**Originator**: Michael Rys  **Treatment of nillability and xsi:nil**

**Description**

Nillability on an element declaration indicates that content of the corresponding element can be empty. The current data model preserves the xsi:nil attribute and this is used in the semantics at the type level. An alternative design would be to remove the xsi:nil attribute and add a ‘nillable’ marker in the schema component in the data model. This might be helping when we perform updates.

**Actual Resolution**

Decision by: xpath-tf on 2002-12-17 ([link to member only information] )

Guided by Jerome the following was decided:

Assume: an element of type T may be nillable. Another element of type T may not be nillable.

There is a Bug in language book re. type matching. Element with nil='true’ does not have content according to the given type.

Question 1 - Yes 10 Abstain 1, Recorded No from Vassilis

Question 4 - Yes (unanimous) Do not know Vasillis votes.

Question 3 - nil marker in datamodel and keep xsi:nil (both) Votes:1, 0, 8 MRys abstains as he wants update case clarified.

Question 2 - Not relevant due to Yes answer to 4.

Decision by: xquery on 2002-12-19 ([link to member only information] )Joint F2F
486. **FS-Issue-0143**: Support for PI, comment and namespace nodes

Locus: *formal-semantics*  Cluster: *Typing*  Priority: 1  Status: *active*
Originator: *FS Editors*

**Support for PI, comment and namespace nodes**

**Description**

The [XPath/XQuery] type system does not currently support PI nodes, comment nodes and namespace nodes.

487. **FS-Issue-0144**: Representation of text nodes in formal values

Locus: *formal-semantics*  Cluster: *Semantics*  Priority: 1  Status: *decided*
Originator: *Jonathan Robie*

**Representation of text nodes in formal values**

**Description**

Formal Values described in section [#FS-sec_values](http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Dec/0352.html) represents either text nodes for well-formed documents or values for validated documents. Do we need to support a dual representation with both text nodes and values in the formal semantics?

**Actual Resolution**

Decision by: *xsl* on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: *xquery* on 2003-03-07 ([link to member only information] )Joint meeting

The formal semantics will keep the current formalism for values which uses either atomic values or text, but not both. The current version is easier to read and understand.

488. **FS-Issue-0145**: Static typing of path expressions in the presence of derivation by extension

Locus: *formal-semantics*  Cluster: *Static typing*  Priority: 1  Status: *decided*
Originator: *Jerome Simeon*

**Static typing of path expressions in the presence of derivation by extension**

**Description**

The current static type analysis rules for Step expressions is broken in the presence of derivation by extension. This bug is impacting section [#FS-sec_steps](http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Dec/0352.html).

**Actual Resolution**

Decision by: *fs-editors* on 2003-01-12 ([link to member only information] )

Dec. FS working draft fixes the static semantics of path expressions in the presence of derivation by extension.

491. **FS-Issue-0148**: Validation of an empty string against a string list

Locus: *formal-semantics*  Cluster: *Semantics*  Priority: 1  Status: *decided*
Originator: *Jerome Simeon*

**Validation of an empty string against a string list**

**Description**
The formal semantics assumes that the result of validating an element with empty content or with an empty text node against a list of strings is the empty sequence, not the empty string. Is that consistent with XML Schema?

Proposed Resolution

I think XML Schema is clear that validating and empty element validated against a list type yields an empty element in the PSVI and therefore an empty element in the data model. In the formal semantics that corresponds to validating the content of the element to result in an empty sequence.

I think this issue should be closed and that the current Formal Semantics documents is correct here.

Actual Resolution

Decision by: xquery on 2003-02-12 ([link to member only information] )

JS’s proposal ACCEPTED. No changes to the document.

492. FS-Issue-0149: Derivation by extension in XQuery

Locus: formal-semantics Cluster: Language Priority: 1 Status: decided
Originator: Phil WadlerDerivation by extension in XQuery

Description

If type u is derived from type t by extension, then the formal semantics document specifies that type u may appear wherever type t is expected. It is not clear what the XQuery document says on this point.

Proposed Resolution

The relevant text in the XQuery document can be found in Section 2.4.2.1, "SequenceType Matching", under the description of "ElemOrAttrType", Rules 1 and 2.

Rule 1 describes matching by type-name. It says "The match is successful only if the given element or attribute has a type annotation that is the same as the required type or ... derived from the required type."

Rule 2 describes matching by element-name. It says "The match is successful only if the name of the given element or attribute is equal to the required name or ... derived from the required name, and if the element or attribute has been validated."

Rules 1 and 2 do not distinguish between derivation by extension or by restriction, so by implication they apply to both kinds of derivation. I do not believe that Rules 1 and 2 were intended to apply to derivation by list or union. I believe this should be clarified by changing "derived" to "derived by restriction or extension" in Rules 1 and 2.

I believe that this change will close Issue 492.

Actual Resolution

Decision by: xquery on 2003-01-29 ([link to member only information] )

Decided to accept proposal and thus resolving issue.

493. FS-Issue-0150: May the content of a text node be the empty string?

Locus: formal-semantics Cluster: Language Priority: 1 Status: decided
Originator: Phil WadlerMay the content of a text node be the empty string?

Description
May the content of a text node be the empty string? None of the formal semantics, the datamodel, or the XQuery document addresses this point.

**Actual Resolution**

Decision by: **xquery** on 2003-02-26 ([link to member only information] )

Add a constraint that a text node must consist of at least one character information items to the Data Model.

**496. FS-Issue-0153**: Support for lax and strict wildcards

**Locus**: formal-semantics **Cluster**: Semantics **Priority**: 1 **Status**: active

**Originator**: FS Editors **Support for lax and strict wildcards**

**Description**

The Formal Semantics does not currently model lax and strict wildcards. The mapping in Section 7, only describes how XML Schema wildcards with the 'skip validation' semantics are imported into the XQuery Type System.

**499. FS-Issue-0156**: Casting and validation

**Locus**: formal-semantics **Cluster**: Language **Priority**: 1 **Status**: active

**Originator**: Jerome Simeon **Casting and validation**

**Description**

Validation from text to simple type performs an operation similar to casting. Should validation of simple type values and ‘cast as’ in [XPath/XQuery] be aligned?

**501. FS-Issue-0158**: Support for XML Schema groups

**Locus**: formal-semantics **Cluster**: Typing **Priority**: 1 **Status**: decided

**Originator**: Jerome Simeon **Support for XML Schema groups**

**Description**

How to support XML Schema groups during the schema import phase is not clear. If the mapping is based on the XML Schema syntax, then it should be handled during the mapping phase. Should we have support for XML Schema groups in the XQuery type system?

**Actual Resolution**

Decision by: **xsl** on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: **xquery** on 2003-03-07 ([link to member only information] )Joint meeting

XML Schema groups are mapped in the type system. There is no need for a syntax for schema group in the type system.

**503. FS-Issue-0160**: Collations in the static environment

**Locus**: formal-semantics **Cluster**: Semantics **Priority**: 1 **Status**: decided

**Originator**: Jerome Simeon **Collations in the static environment**

**Description**

The Formal Semantics does not represent collations in the static environment. Should it?
Actual Resolution

Decision by: xsl on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: xquery on 2003-03-07 ([link to member only information] )Joint meeting

Add collations in the Dynamic Environment.

508. **FS-Issue-0165**: Namespaces in element constructors

**Locus**: formal-semantics  **Cluster**: Semantics  **Priority**: 1  **Status**: active
**Originator**: Denise Draper Namespaces in element constructors

**Description**

We do not supply either namespaces or schema-components to the constructor. We cannot do these things because of the bottom-up nature of element construction: we do not, in general, know either the namespaces in scope or the validation-associated schema type until this element has been “seated” in some containing element (and so on recursively).

509. **FS-Issue-0166**: Static typing for validate

**Locus**: formal-semantics  **Cluster**: Static typing  **Priority**: 1  **Status**: decided
**Originator**: Jerome Simeon Static typing for validate

**Description**

Although validate should always return a typed value, there is no way to do static analysis for it, since the type against which it is validated cannot be known at static type (it depends on the result of evaluation for the input expression).

**Actual Resolution**

Decision by: xpath-tf on 2003-03-04 ([link to member only information] )

Decision by: xsl on 2003-03-06 ([link to member only information] )Joint meeting

Decision by: xquery on 2003-03-06 ([link to member only information] )Joint meeting

Ignore content’s type. Assign static type using element name and validation context.

May raise type error if element content will NEVER validate successfully.

```xml
<fixbrain>{ validate { input()/fixsink/person } }</fixbrain>
```

510. **FS-Issue-0167**: Is validate working on sequences?

**Locus**: formal-semantics  **Cluster**: Language  **Priority**: 1  **Status**: active
**Originator**: Jerome Simeon validate working on sequences?

**Description**

We have not yet reached consensus on the semantics of validate for all sequences. In particular:

1. Can validate be applied to a sequence, or only to a single item?

2. Can validate be applied to any item? Some believe that it should apply only to elements and document nodes - the two kinds of nodes for which it is well defined - and raise an error if other items are encountered. Others believe that validate should validate elements and document nodes, returning other items without change. Should an attribute node be validated? Note that XML Schema does not
define stand-alone validation for attribute nodes.

We need to carefully consider the usage scenarios in which validation might be applied to sequences or items that are not elements or document nodes to determine which behavior is most desirable.

Should validate work on sequences of nodes, or only on a single node?

Proposed Resolution

At today’s Query telcon, we discussed the semantics of the validate expression. We need to decide whether it applies to a sequence, or only to a single node. We also need to define its semantics for various kinds of nodes (and for atomic values, if they occur in a validated sequence).

Most people agreed that the validate operator, applied to an attribute node, should raise an error. This is because we rely on the XML Schema specification for the semantics of validation, and XML Schema does not specify how to validate a stand-alone attribute node.

For text, comment, and PI nodes and atomic values, some people believe that validate should raise an error, and others believe that validate should simply return the original node or value unchanged. We did not reach a consensus on this.

If validate is defined to operate on a sequence, the sequence may contain some items for which validate is well-defined and others for which it is not well-defined. In order for users to invoke validate on a sequence, they will first need to filter the sequence to keep only items for which validate is well-defined. If users need to do this work anyway, they might as well invoke validate expressions only on the individual items for which validation is well-defined. This was considered a strong argument for not allowing sequences to be passed to a validate expression.

At the end of the discussion, a consensus seemed to be emerging around the following points:

(1) The operand of validate must be a single element node or a single document node (anything else raises a type error).

(2) The semantics of validate on an element node are as defined in Section 3.13 ("Validate Expressions") of the current language document.

(3) The semantics of validate on a document node are as defined in Karun’s proposal, http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2003Feb/0015.html, approved by the Query working group on 2/12/03.

512. FS-Issue-0169: Conformance Levels

Locus: formal-semantics Cluster: Semantics Priority: 1 Status: active
Originator: Jerome Simeon

Description

[XPath/XQuery] supports several conformance levels. Whether the formal semantics need to distinguish those conformance levels is an open issue. If yes, how to distinguish those conformance levels in the formal semantics is an open issue.

513. FS-Issue-0170: Imprecise static type of constructed elements

Locus: formal-semantics Cluster: Static typing Priority: 1 Status: decided
Originator: Mary Fernandez

Description

Implementation of Alternative 1 means that the static type for constructed elements and attributes is very imprecise. E.g., the type of {(1,2,3)} is element a { xs: anyType }. See remark by Denise in section on element constructors for possible fix.
Actual Resolution

Decision by: xpath-tf on 2003-03-04 ([link to member only information] )

Decision by: xsl on 2003-03-06 ([link to member only information] ) Joint meeting

Decision by: xquery on 2003-03-06 ([link to member only information] ) Joint meeting

Applies to literal element constructor and computed element constructor with literal name

Raise type error if element content will NEVER validate successfully.

If static typing is enabled, it is a static type error if the type of the content of the element is NOT a subtype of type declared for the element. Untyped content treated "liberally".

Parallels semantics of static typing rules for function calls.

| type plumber { element name of type xs:string } |
| type surgeon { element name of type xs:string } |
| element fixsink { element person of type plumber } |
| element fixbrain { element person of type surgeon } |

<fixbrain>{ input()/fixsink/person }</fixbrain>

514. FS-Issue-0171: Raising errors

Locus: formal-semantics Cluster: Semantics Priority: 1 Status: decided
Originator: Mary Fernandez Raising errors

Description

The semantics of raising errors in [XPath/XQuery] is not formally specified.

Actual Resolution

Decision by: fs-editors on 2003-01-12 ([link to member only information] )

Dec. FS working draft describes the semantics of errors.

516. FS-Issue-0173: Typeswitch and type substitutability

Locus: formal-semantics Cluster: Typing Priority: 1 Status: decided
Originator: XSL Working Group Typeswitch and type substitutability

Description

It seems that some examples of typeswitch with xs:anySimpleType might break type substitutability.

Proposed Resolution

The required changes include the new (simpler!) normalization and typing rules below, plus:

1. A new formal-semantics function fs:cast-untypedAtomic, which takes two arguments and:
   a. If the first argument is the empty sequence, returns the empty sequence.
   b. If first argument is untypedAtomic, and
       b1. If the second argument is untypedAtomic, returns the first argument cast to string;
       b2. Else if the second argument is numeric, returns the first
argument cast to double;

b3. Otherwise, returns the first argument cast to the type of the "second" argument.

c. Otherwise, returns the first argument, unchanged.

Thus fs:cast-untypedAtomic encapsulates the implicit coercion semantics of the arithmetic and general comparison operators.

The "generic" signature of fs:cast-untypedAtomic is:

\[
\text{fs:cast-untypedAtomic} (\$v1 \text{ as xdt:anyAtomicType?}, \$v2 \text{ as xdt:anyAtomicType?}) \text{ as xdt:anyAtomicType?}
\]

The specific type rules for fs:cast-untypedAtomic are below.

2. New signatures for the overloaded (op:numeric-OP) operators such that they are all defined on the empty sequence. If either argument is the empty sequence, these operators return the empty sequence.

E.g., here's the new signature for op:numeric-add (similar signatures for the other op:numeric-op functions):

\[
\text{op:numeric-add} (\text{$\text{operand1 as numeric?}$, $\text{operand2 as numeric?}$}) \text{ as numeric?}
\]

New normalization rules:

Given the changes above, here are the normalization rules for the plus and the general less-than operators. (The general rules for any arithmetic or general comparison operator are similar -- it's just easier to explain by example).

Plus:

\[
[\text{Expr1} + \text{Expr2}]_\text{Expr} ==
\]

let $e1 := \text{fn:data} ( [\text{Expr1}]_\text{Expr} ) \text{ return}
let $e2 := \text{fn:data} ( [\text{Expr2}]_\text{Expr} ) \text{ return}
let $v1 := \text{fs:cast-untypedAtomic} ($e1, 1.0e0) \text{ return}
let $v2 := \text{fs:cast-untypedAtomic} ($e2, 1.0e0) \text{ return}
\text{op:numeric-add}($v1, $v2)

Note that arithmetic operators always cast an untypedAtomic argument to a double -- the call to fs:cast-untypedAtomic above performs this cast.

Less-than:

\[
[\text{Expr1} < \text{Expr2}]_\text{Expr} ==
\]

some $v1 in \text{fn:ndata} ( [\text{Expr1}]_\text{Expr} ) \text{ satisfies}
some $v2 in \text{fn:ndata} ( [\text{Expr2}]_\text{Expr} ) \text{ satisfies}
let $v1' := \text{fs:cast-untypedAtomic} ($v1, $v2) \text{ return}
let $v2' := \text{fs:cast-untypedAtomic} ($v2, $v1) \text{ return}
\text{op:less-than}($v1', $v2')

Note that general comparison operators always cast an untypedAtomic argument to the type of the other argument -- the call to fs:cast-untypedAtomic performs this cast.

Static type rules for fs:cast-untypedAtomic:

Rule (a) above:

\[
\text{statEnv} \vdash \text{Expr1} : ()
\]

\[
\text{statEnv} \vdash \text{fs:cast-untypedAtomic(Expr1, Expr2)} : ()
\]

Rule (b1) above:

\[
\text{statEnv} \vdash \text{Expr1 : Type1} \quad \text{statEnv} \vdash \text{Type1 <: xdt:untypedAtomic}
\]

\[
\text{statEnv} \vdash \text{Expr2 : Type2} \quad \text{statEnv} \vdash \text{Type2 <: xdt:untypedAtomic}
\]

\[
\text{statEnv} \vdash \text{fs:cast-untypedAtomic(Expr1, Expr2)} : \text{xs:string}
\]

Rule (b2) above:

\[
\text{statEnv} \vdash \text{Expr1 : Type1} \quad \text{statEnv} \vdash \text{Type1 <: xdt:untypedAtomic}
\]

\[
\text{statEnv} \vdash \text{Expr2 : Type2} \quad \text{statEnv} \vdash \text{Type2 <: (xs:decimal | xs:float | xs:double)}}
\]

\[
\text{statEnv} \vdash \text{fs:cast-untypedAtomic(Expr1, Expr2)} : \text{xs:double}
\]

Rule (b3) above:
statEnv |- Expr1 : Type1  
Type1 <: xdt:untypedAtomic
statEnv |- Expr2 : Type2
statEnv |- not(Type2 <: xdt:untypedAtomic | xs:decimal | xs:float | xs:double)
statEnv |- fs:cast-untypedAtomic(Expr1, Expr2) : Type2

Rule (c) above:
statEnv |- Expr1 : Type1
not(Type1 <: xdt:untypedAtomic)
-----------------------------
fs:cast-untyped-atomic(Expr1, Expr2) : Type1

Static type rules for typeswitch:
---------------------------------------------------
The static rules for typeswitch are now very simple. We type each case expression and take the union of these types:
statEnv |- Expr0 : Type0
statEnv |- Expr0 [case] case Variable1 as SequenceType1 return
statEnv |- Expr1 : Type1
statEnv |- Expr0 [case] case Variable1 as SequenceType1 return
statEnv |- Expr1 : Type1
statEnv |- Expr0 [case] case Variable1 as SequenceType1 return
statEnv |- Expr1 : Type1
statEnv |- Expr0 [case] case Variable1 as SequenceType1 return
statEnv |- Expr1 : Type1
statEnv |- Expr0 [case] case Variable1 as SequenceType1 return
statEnv |- Expr1 : Type1
statEnv |- Expr0 [case] default Variable1+1 return Exprn+1 : Typen+1
statEnv |- (typeswitch (Expr0)
  case Variable1 as SequenceType1 return Expr1
  ...
  case Variable1 as SequenceType1 return Exprn
  default Variable1+1 return Exprn+1)
  : Type1 | ... | Typen+1

The type of the case clauses is inferred based on the CaseType, independently of the typeswitch expression type.

CaseType = [ SequenceType ]_sequencetype
statEnv + varType(Variable : CaseType) |- Expr : Type1
---------------------------------------------
statEnv |- Type0 [case] case Variable as SequenceType return Expr : Type1

The type of the default branch is inferred based on the typeswitch expression type.
statEnv + varType(Variable : Type0) |- Expr : Type1
---------------------------------------------
statEnv |- Type0 [case] default Variable return Expr : Type1

Actual Resolution

Decision by: fs-editors on 2003-02-13 ([link to member only information] )
Decision by: xquery on 2003-02-19 ([link to member only information] )
Decision by: xsl on 2003-02-27 ([link to member only information] )
Proposal accepted resolving this issue.

519. FS-Issue-0176: Type of document node

Locus: formal-semantics Cluster: Typing Priority: 1 Status: decided
Originator: Jerome Simeon
Type of document node

Description

What should the type of a document node be? Should a document node be described by a type name?

Actual Resolution
520. **FS-Issue-0177**: Coercion between untyped and atomic values

**Locus**: formal-semantics  **Cluster**: Semantics  **Priority**: 1  **Status**: decided

**Originator**: Jerome Simeon  

**Description**

Function calls, arithmetics expressions, etc. should cast untyped data to values. This is not currently formally specified.

**Actual Resolution**

Decision by: fs-editors on 2003-01-12 ([link to member only information] )

Dec. FS working draft describes coercion between untyped and atomic values in function calls.

521. **FS-Issue-0178**: Semantics of XPath 1.0 compatibility

**Locus**: formal-semantics  **Cluster**: Semantics  **Priority**: 1  **Status**: decided

**Originator**: Jerome Simeon  

**Description**

The semantics is not specified in the case the XPath 1.0 compatibility flag is on.

**Actual Resolution**

Decision by: fs-editors on 2003-01-12 ([link to member only information] )

Dec. FS working draft describes xpath 1.0 backward compatibility semantics.

523. **input-and-schema-validation**: input(), collection(), document(); validation semantics

**Locus**: xpath  **Cluster**: type-semantics  **Priority**: 1  **Status**: active

**Originator**: XPath TF  

**Description**

What are the semantics of the functions input(), collection(), document(), and the implicit context that binds variables to documents with respect to schema validation.

**Interactions and Input**

524. **schema-plural-data-types**: Plural datatypes different from Singular *?

**Locus**: xpath  **Cluster**: type semantics  **Priority**: 1  **Status**: decided

**Originator**: XPath TF  

Plural datatypes different from Singular *?
Description

Is there any semantic difference between eg NMTOKEN* and NMTOKENS? Statically? Dynamically?

Actual Resolution

Decision by: xsl on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: xquery on 2003-03-07 ([link to member only information] )Joint meeting

To solve issues 316 and 524 we need to do eliminate three constructor functions from F&O, and integrating with the changes described in the message: http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2003Mar/0021.html.

This was agreed.

525. FS-Issue-0180: Static context accessible from the dynamic context

Locus: formal-semantics Cluster: Semantics Priority: 1 Status: decided
Originator: Jerome SimeonStatic context accessible from the dynamic context

Description

The formal semantics states that the static context is available from the dynamic context at run time. This should only be the case for part of the static context. Moreover certain parts of the dynamic context must be updated appropriately from during the dynamic evaluation (e.g., namespaces).

Proposed Resolution

My recollection is that Mary implemented some changes in the internal FS document. I think we never reported that change to the working group.

I'm not sure how to deal with this one. Maybe Mary can explain the change and the issue be closed?

Actual Resolution

Decision by: fs-editors on 2003-04-07 ([link to member only information] )

The new formal semantics working draft clarifies which part of the static context is available from the dynamic context.

526. static-semantics-anySimpleType-untypedAtomic: Semantics for anySimpleType and untypedAtomic

Locus: xpath Cluster: types Priority: 1 Status: active
Originator: XPath TFSemantics for anySimpleType and untypedAtomic

Description

1. If you statically infer xs:anySimpleType, can you error if the expected type is a subtype or do you handle it as xdt:untypedAtomic at runtime?

2. If you statically infer xdt:untypedAtomic, can you recast into expected type statically?

Actual Resolution

Decision by: xquery on 2003-04-09 ([link to member only information] )

The two questions are answered in the current draft of the formal semantics.
527. **static-typing-index-expressions**: Static typing of XPath index expressions

Locus: [formal-semantics] Cluster: types Priority: 1 Status: decided
Originator: Algebra Editors

Static typing of XPath index expressions

Description

Typing for XPath predicate expressions of the form E1[Integer] is not precise (often of the form ItemType* instead of ItemType or ItemType?). Should we have specific typing rules for XPath predicate expressions of the form E1[Integer], or is the use of 'type relief' functions, as described in: [http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Nov/0161.html](http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2002Nov/0161.html) sufficient?

Actual Resolution

Decision by: [xquery](http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2003Jan/0026.html) on 2003-01-22 (link to member only information)

The general static type of the general expression:

\[ \text{Expr1}[\text{Expr2}] \]

Is (assuming normalization has taken place)

\[ \text{TYPE}(\text{Expr1}[\text{Expr2}]) = \text{IF TYPE(Expr2) <: numeric THEN TYPE(Expr1)} \]

Decided that if Expr2 is a literal number, then do cardinality reduction, otherwise don’t.

528. **text-constructor-empty-sequence**: Semantics of text constructor on empty sequence

Locus: [xpath] Cluster: semantics Priority: 1 Status: decided
Originator: XPath TF

Semantics of text constructor on empty sequence: should it return empty sequence or raise an error?

Actual Resolution

Decision by: [xquery](http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2003Jan/0026.html) on 2003-02-26 (link to member only information)

Change the language document, adding that a text node constructor on empty sequence returns the empty sequence.

529. **node-identity-modeling**: Node identity in the formal semantics

Locus: [xquery] Cluster: constructor-expr Priority: 1 Status: decided
Originator: Phil Wadler

Node identity in the formal semantics

Description

The fs currently does not model node identity. The position of the formal semantics editors is that a formal semantics need not model everything. So it is not absolutely required that we model node identity, though it would be desirable for us to do so, not least because a clear description of node identity may help clarify the semantics of update. The formal semantics should include a mode of node identity if time permits.

Actual Resolution
530. **default-validation-mode**: Default default validation mode

**Description**

The adoption of implicit validation of element constructors specified that syntax be provided to allow query authors to specify the validation mode (skip, lax, or strict) for a given constructor, as well as syntax to allow query authors to specify in the query prolog the default validation for all constructors in the query. If neither syntax is used, there must be a "default default" mode. That mode can be specified by the XQuery specification itself, or it can be made implementation-defined to account for varying implementation requirements. Specifying the default mode in the XQuery specification would seem to improve interoperability, but there are strong user requirements that are best satisfied by allowing the XQuery implementation determine the default mode based on customer needs and other factors. In order to improve interoperability, XQuery authors can specify their default mode in their XQuery prologs.

**Actual Resolution**

Decision by: **xquery** on 2003-03-06 ([link to member only information] )

This issue is an example of a more generic issue: how should the default value be determined for the parts of the static context and evaluation context that are not explicitly assigned values in the query prolog? I believe that this generic issue should be addressed by a well-defined and uniform policy that is documented in the language books. I propose the following resolution:

(1) Section 2.1.1, "Static Context": Add the following sentence just before the bullet-list of context items (XQuery only): "Any part of the Static Context that is not assigned a value in the Query Prolog may be assigned a default value by the XQuery implementation."

(2) Section 2.1.2, "Evaluation Context": Add the following sentence just before the bullet-list of context items: "Any part of the Evaluation Context may be assigned an initial value by the XPath/XQuery implementation."

Proposal accepted.

531. **type-copied-subelement**: Should discard the type of copied subelement

**Description**

Currently element construction untypes all simple content (untypedAtomic) but preserves the type of subnodes.

This is seen as inconsistent by some and thus an issue.

The proposed resolution would be to erase the type for any node under the new element (make it element/attribute of type xs:anyType/xdt:untypedAtomic or potentially (and preferably) untypedComplex for elements).
Decision by: xsl on 2003-03-07 ([link to member only information] )Joint meeting

Decision by: xquery on 2003-03-07 ([link to member only information] )Joint meeting

Element construction should behave like skip validation and erase the type of all subnodes.

532. **sequence-type-path-alignment**: Alignment between path expressions and sequence types

Locus: xpath Cluster: syntax Priority: 1 Status: decided
Originator: Phil Wadler Alignment between path expressions and sequence types

Description

Should there be better alignment between path expressions and sequence types?

Actual Resolution

Decision by: xpath-tf on 2003-03-04 ([link to member only information] )

Decision by: xsl on 2003-03-06 ([link to member only information] )Joint meeting

Decision by: xquery on 2003-03-06 ([link to member only information] )Joint meeting

Proposal in issue #SequenceType-problems accepted resolving issue.

533. **automatic-coercion-strict-or-not**: Strict static typing for automatic coercion of untyped atomic to atomic values

Locus: formal-semantics Cluster: type-semantics Priority: 1 Status: decided
Originator: Mary Fernandez Strict static typing for automatic coercion of untyped atomic to atomic values

Description

Should static typing for automatic coercion of untyped atomic to atomic values be strict or not? For instance:

Should the following function call:

```xml
define function f($x as xs:integer) ....
let $x := <a>1</a> return f($x/node())
```

be allowed or not?

Actual Resolution

Decision by: xpath-tf on 2003-03-04 ([link to member only information] )

Decision by: xsl on 2003-03-06 ([link to member only information] )Joint meeting

Decision by: xquery on 2003-03-06 ([link to member only information] )Joint meeting

In principle, xdt:untypedAtomic is typed liberally rather than conservatively. That is, it is made statically well-typed whenever possible.

Thus, addition of xdt:untypedAtomic to xs:integer is well-typed and has type xs:double. Comparing xdt:untypedAtomic to any other type is well-typed.

This resolution applies only to the general comparison (=, <), not to value comparison (eq, lt), so that
the latter retains transitivity.

534. function-calls-and-expected-type: Semantics of function calls and notion of "expected type"

Locus: formal-semantics Cluster: type-semantics Priority: 1 Status: active
Originator: Kristoffer Rose

Semantics of function calls and notion of "expected type"

Description

While the "expected type" is well defined for user and non-overloaded standard functions, it is not so for the overloaded built-in functions. For general [XQuery/XPath] use (without XPath 1.0 compatibility flag) we must ensure that each overloaded argument is either always atomic or non-atomic so it is unambiguous for every function argument which of the two normalization choices to pick. With XPath 1.0 compatibility flag some other solution must be found as several functions are now overloaded with, e.g., numeric and non-numeric types allowed for the same argument.

535. lexical-state-tables: Lexical state tables

Locus: xpath Cluster: syntax Priority: 1 Status: decided
Originator: Scott Boag

Lexical state tables

Description

Should the state tables be removed or made non normative?

Actual Resolution

Decision by: xsl on 2003-03-07 ([link to member only information] )Joint meeting
Decision by: xquery on 2003-03-07 ([link to member only information] )Joint meeting

Decided to keep them normative.

536. datamodel-infoset-types: Data model to infoset mapping and type information

Locus: xpath Cluster: types Priority: 1 Status: active
Originator: Michael Rys

Data model to infoset mapping and type information

Description

When mapping from a data model to the infoset (as it is used in the definition of validate), we need to lose any type properties that may be part of the data model. In particular, this means that the infoset will not populate the attribute’s type property. We need to make sure that the documents reflect this.

537. imported-schemas-used-function-signatures: What happens to imported schemas that are used in function signatures?

Locus: xquery Cluster: modules Priority: 1 Status: decided
Originator: Michael Rys

What happens to imported schemas that are used in function signatures?

Description

If a function module imports a schema and uses it in the function signature, is the import of the function module implicitly importing the schema components needed by the function signatures?

Example:
The module says:

```xquery
import schema namespace foo = "abc"
namespace f = "xyz"
define function f:foo($x as foo:bar) as xs:int { ... }
```

Does

```xquery
import module namespace f = "xyz"
```

import the schema and one only needs to bind a prefix to the schema namespace or does the schema have to be explicitly imported?

**Actual Resolution**

Decision by: xquery on 2003-03-06 ([link to member only information] )

The adopted module proposal implicitly resolves this issue. An example showing this clearly needs to be added.

538. **keyword-extensions:** Can (and should) keyword extensions be allowed in XQuery by equipping them with a namespace prefix?

*Locus:* xquery  
*Cluster:* extensions  
*Priority:* I  
*Status:* decided  
*Originator:* Peter Fankhauser

**Description**

Can (and should) keyword extensions be allowed in XQuery by equipping them with a namespace prefix?

**Example:**

```xquery
declare namespace my = http://example.com/myextensions
for $e in document("employees.xml")//employee
order by $e/lastname my:fastorder
return $e
```

The advantage of this is that namespaces are a well-established concept for extending vocabularies, and different prefixes may be used for multiple extensions (my:fastorder vs. real:fastorder). The possible disadvantage may be that namespaces are regarded as too clumsy for this purpose.

Note that using namespaces for keyword extensions does not appear to provide a generic mechanism to make the XQuery grammar extensible (one cannot simply allow a sequence of Qnames at arbitrary places without running into ambiguities). An implementation providing some extensions still needs to modify the XQuery grammar accordingly. However this also holds for the other two options, unless "x-" is added to the reserved keywords.

**Interactions and Input**

**Actual Resolution**

Decision by: xquery on 2003-03-12 ([link to member only information] )

The decision for #extension-mechanism also covers this issue.

539. **cast-untypedAtomic-backward-compatibility-mode:** Semantics of fs:cast-untypedAtomic in backward compatibility mode
What should be the semantics of `fs:cast-untypedAtomic` in the XPath 1.0 backward compatibility mode?

540. static-semantics-input-unions: How does the static semantics work in the case where the input types are unions?

How does the static semantics works in the case where the input types are unions? (union propagation) For instance, what is the static type obtained for an input `(untypedAtomic | int) + (float)` [the expected result is `(double | float)`].

541. static-type-empty-error: Should it be a static error if an expression other than () has type empty?

Motivation: This catches common errors such as $x/e$ when there is no `e` element.

Interactions and Input

Actual Resolution

Decision by: xpath-tf on 2003-03-04 ([link to member only information])

Decision by: xsl on 2003-03-06 ([link to member only information]) Joint meeting

Decision by: xquery on 2003-03-06 ([link to member only information]) Joint meeting

Yes this should be a static error.

542. type-of-well-formed: What should be the type of an attribute or element that is well-formed but not validated, or is validated in skip mode?

Currently, attributes are given type `xs:anySimpleType` and elements are given type `xs:anyType`. In order to do a better job of static typing for well-formed data, it might be better to give attributes type `xdt:untypedAtomic` and to give elements type `xdt:anyUntyped` (where the latter is a newly minted type).
543. xpath-matching-substitution-groups: How can a path expression match elements in the substitution group of a given element?

Locus: xpath Cluster: type-semantics Priority: 1 Status: decided
Originator: Jonathan Robie

How can a path expression match elements in the substitution group of a given element?

Description

How can a path expression match elements in the substitution group of a given element?

Actual Resolution

Decision by: xpath-tf on 2003-03-04 ([link to member only information] )
Decision by: xsl on 2003-03-06 ([link to member only information] ) Joint meeting
Decision by: xquery on 2003-03-06 ([link to member only information] ) Joint meeting
Proposal in issue #SequenceType-problems accepted resolving issue.

544. xpath-matching-types: How can a path expression match nodes of a given type?

Locus: xpath Cluster: type-semantics Priority: 1 Status: decided
Originator: Jonathan Robie

How can a path expression match nodes of a given type?

Description

How can a path expression match nodes of a given type?

Actual Resolution

Decision by: xpath-tf on 2003-03-04 ([link to member only information] )
Decision by: xsl on 2003-03-06 ([link to member only information] ) Joint meeting
Decision by: xquery on 2003-03-06 ([link to member only information] ) Joint meeting
Proposal in issue #SequenceType-problems accepted resolving issue.

545. implicitly-element-attribute: Are XS element/attribute declarations implicitly added to the schema context?

Locus: xpath Cluster: types Priority: 1 Status: active
Originator: Processing Model TF

Are XS element/attribute declarations implicitly added to the schema context?

Description

Are XS element/attribute declarations implicitly added to the schema context?

546. conformance-levels: Are there more processing model options that could make sense as consistent features and thus as conformance levels?

Locus: xpath Cluster: conformance-levels Priority: 1 Status: active
Originator: Processing Model TF

Are there more processing model options that could make sense as consistent features and thus as conformance levels?
Description

Are there more processing model options that could make sense as consistent features and thus as conformance levels? What types can these "predefine" in the static context? Do these options have the desired/expected interoperability characteristics?

547. **grammar-declare**: Use "declare" for declarations

Locus: qquery Cluster: syntax Priority: 1 Status: active
Originator: Michael RysUse "declare" for declarations

Description

G3. Section 4.3 Validation Declaration

```xml
[118] ValidationDecl ::= <"validation" "lax"> | <"validation" "strict"> |
                    <"validation" "skip">
```

Add declare keyword in order to align with general declaration syntax

Proposal:

```xml
<"declare" "validation" (*lax* | *strict* | *skip*)>
```

G4. Section 4.4 Xmlspace Declaration

```xml
[104] XMLSpaceDecl ::= <"declare" "xmlspace"> *=* (*preserve* | *strip*)
```

drop "=" in order to align with general declaration syntax (this is an option and not a value like in namespace declarations)

Proposal:

```xml
<"declare" "xmlspace"> (*preserve* | *strip*)
```

G5. Section 4.5 Default Collation

```xml
[105] DefaultCollationDecl ::= <"default" "collation" *="> StringLiteral
```

drop "=" and add "declare" in order to align with general declaration syntax.

Proposal:

```xml
<"declare" "default" "collation"> StringLiteral
```

548. **lexical-rules**: Lexical Rules: states normative?

Locus: xpath Cluster: syntax Priority: 1 Status: active
Originator: Michael RysLexical Rules: states normative?

Description

G6. Section A.1.2 Lexical Rules
Current wording: "The lexical contexts and transitions between lexical contexts is described in terms of a series of states and transitions between those states."

Make all states non-normative except those that are needed for XML construction, string literal and XQuery context.

549. **namespace-constructor**: Computed namespace-constructor

Locus: `xquery` Cluster: `namespaces` Priority: 1 Status: decided
Originator: XQuery WG Computed namespace-constructor

Description

Is the following construction is a valid XQuery expression or not:

```
element foo:bar {
  attribute a1 {"foo"},
  attribute xmlns:foo {"urn1"},
  element e1 {"goo"}
}
```

The XML Query WG has decided to forbid the above construct by defining an explicit constructor for namespaces in computed constructor syntax, and to disallow the use of computed attribute constructors for declaring namespaces. The WG also believes that there should be a constructor for comments.

The WG also agreed that the namespace constructor should have a syntax parallel to that of attribute constructors, be restricted to constant namespace prefixes and URLs, and should appear first in the children of an element constructor, as in the following example:

```
element foo:bar {
  namespace test { "urn1" },
  attribute a1 {"foo"},
  element e1 {"goo"}
}
```

The above example is precisely equivalent to:

```
<foo:bar xmlns:test="urn1">{
  attribute a1 {"foo"},
  element e1 {"goo"}
}</foo:bar>
```

Proposed Resolution

The proposal itself is quite simple:

Syntax:

```
ComputedElementConstructor ::= (<"element" QName "{">
  | (<"element" "{"> Expr "}" "{")) NamespaceCList? ExprSequence?
"

NamespaceCList ::= ComputedNamespaceConstructor ("," ComputedNamespaceConstructor)*",

ComputedNamespaceConstructor ::= (<"namespace" NCName "{"> StringLiteral
"
```

A NamespaceCList may appear as the first expressions in the ExprSequence of an element constructor, using either computed or XML element constructor syntax. It is syntactically impossible for ComputedNamespaceConstructors to appear in any other context.

Semantics:
If two ComputedNamespaceConstructors in the same element declare the same prefix, a type error is raised. A namespace declared using a ComputedNamespaceconstructor is in scope within the constructed element, including the name of the element itself.

If the name of an attribute in a computed attribute constructor begins with the string "xmlns", a type error is raised.

In an ElementConstructor, if an attribute in an AttributeList begins with the string "xmlns", then it is treated as a namespace declaration, and is equivalent to a computed namespace constructor that declares the same prefix and URI.

Actual Resolution

Decision by: xquery on 2003-03-12 ([link to member only information] )

Adoption of the proposal.

550. comments-location: Location of Comments

Locus: xquery Cluster: syntax Priority: 1 Status: active
Originator: Don Chamberlin Location of Comments

Description

XQuery 1.0 states "Comments may be used anywhere that ignorable whitespace is allowed, and within element content." I am trying to understand what that means and whether it is still true.

What does this mean?

\[<a>A (:big:) cat</a>\]

Does (:big:) get recognized as a comment and removed from the element content, and then the two surrounding pieces of text get coalesced into a single text node with the content "A cat"?

Or does the tag get a text node containing the string "A (:big:) cat"? If this is not the case, then how would I construct an element that actually has this content?

Have we thought carefully about whether we want comments to be recognized inside tag content? In most other ways, element-content is treated much like a string. I believe that comments are not recognized inside strings, right? Is this a strong argument for not recognizing comments inside element content? Is there any argument for recognizing them?

551. constructor-functions-user-types: Constructor functions for unprefixed user defined types

Locus: xpath Cluster: types Priority: 1 Status: active
Originator: XSL WG Constructor functions for unprefixed user defined types

Description

We state that there is (implicitly) a constructor function in the static context for each user defined type. How do you refer to such a constructor function for a non-prefixed type?

552. processing-instruction-quotes: Should the quotes in processing-instruction("...") be optional?

Locus: xpath Cluster: syntax Priority: 1 Status: active
Originator: XSL WG Should the quotes in processing-instruction("...") be optional?

Description
Do we want to make the quotes in processing-instruction() optional? It is always a string literal and we have introduced element() and attribute() that already do not look like functions...

553. **dynamic-errors-reported-statically**: Rules for reporting dynamic errors statically

Locus: **xpath** Cluster: **errors** Priority: 1 Status: **active**
Originator: **XSL WG** Rules for reporting dynamic errors statically

**Description**

Under what circumstances can a dynamic error detected early be reported statically, for example if it occurs in code that might not be executed. A host language may determine this, e.g. an `<xsl:if>` construct.

**Interactions and Input**

554. **what-is-normative**: What is the really normative text?

Locus: **xpath** Cluster: **editorial** Priority: 1 Status: **active**
Originator: **XPath TF** What is the really normative text?

**Description**

Some normative material in XPath/XQuery specifications occurs in more than one location. The specifications need to be clear which of the replicated material is the unique normative definition to ensure ease of implementation and avoid conflicts.

555. **module-import-fs**: Formal Semantics of Module Import

Locus: **formal-semantics** Cluster: **formal semantics** Priority: 1 Status: **active**
Originator: **fs editors** Formal Semantics of Module Import

**Description**

The semantics of Module Import is not formally specified.

556. **variable-definitions-fs**: Formal Semantics of Variable Definitions

Locus: **formal-semantics** Cluster: **formal semantics** Priority: 1 Status: **active**
Originator: **fs editors** Formal Semantics of Variable Definitions

**Description**

The semantics of Variables Definitions in the XQuery prolog is not formally specified.

557. **validation-declaration-fs**: Formal semantics of Validation Declaration

Locus: **formal-semantics** Cluster: **formal semantics** Priority: 1 Status: **active**
Originator: **fs editors** Formal semantics of Validation Declaration

**Description**

The semantics of Validation Declaration in the XQuery prolog is not formally specified.

558. **FS-allow-PIs-comments**: The content of element types should
always allow PI’s and comment node types

**Description**

The content of element types should always allow PI’s and comment node types. This is not currently taken into account in the FS document. The best way to deal with that problem is still an open issue. Two possible options are to add PI and comment types during XML Schema import into the XQuery type system (Section 8). Another option is to add PI and comment types during type expansion (judgment expands_to in 7.2.7 Type expansion).

**559. FS-support-new-sequence-type: New Sequence Type needs to be fully implemented in Formal Semantics**

**Description**

The formal semantics of the new form of sequence types is incomplete and is still under work.

**H Revision Log (Non-Normative)**

**H.1 18 April 2003**

- Extensive changes have been made in the syntax for specifying a type, for example in a function signature or in a cast expression.
- New options have been added to KindTest in a path expression, enabling elements and attributes to be selected by their type as well as by their name.
- A "validation mode" (strict, lax, or skip) has been added to the validate expression.
- The definition of the value comparison operators has changed to make these operators transitive. The relationship between the value comparison and general comparison operators has been affected by this change.
- The syntax of the "cast" and "treat" expressions has changed so that the operator and target type follow the operand expression, as in 5 cast as decimal.
- Certain changes have been made to the grammar. An expression that may include a concatenation operator is now called Expr (formerly ExprSequence). An expression that may not contain a concatenation operator is now called ExprSingle (formerly Expr). Some changes have also been made in the precedence of operators (see grammar for details).
- Comment delimiters have been changed from {-- --} to (: :).
- A new predefined namespace prefix, xdt, has been created for predefined types, and two new types, xdt:untypedAtomic and xdt:anyAtomicType, have been introduced.
- Changes have been made in the definition of typed value for certain kinds of nodes.
- Certain changes have been made in the conversion rules for operands of arithmetic operators (for example, idiv casts its operands to integer rather than double). Also, the result of the div operator on two integers is decimal rather than double.
- Implementations are now allowed to provide default initial values for all components of the static context and evaluation context.
• The former section 2.5 on the scope of variable bindings has been replaced by more specific material distributed throughout the document.