Chapter 1 - Introduction to XQuery

Example: catalog.xml

```xml
<catalog>
  <product dept="WMN">
    <number>557</number>
    <name language="en">Fleece Pullover</name>
    <colorChoices>navy black</colorChoices>
  </product>
  <product dept="ACC">
    <number>563</number>
    <name language="en">Floppy Sun Hat</name>
  </product>
  <product dept="ACC">
    <number>443</number>
    <name language="en">Deluxe Travel Bag</name>
  </product>
  <product dept="MEN">
    <number>784</number>
    <name language="en">Cotton Dress Shirt</name>
    <colorChoices>white gray</colorChoices>
    <desc>Our <i>favorite</i> shirt!</desc>
  </product>
</catalog>
```

- `doc("catalog.xml")/catalog/product` will select all the product elements from the catalog.xml document.
- The asterisk (*) can be used as a wildcard to indicate any element name. `doc("catalog.xml")/product` will return any product children of the outermost element, regardless of the outermost element's name.
- Alternatively, you can use a double slash (//) to return product elements that appear anywhere in the catalog document, as in: `doc("catalog.xml")/product`.
- `doc("catalog.xml")/catalog/product[@dept = "ACC"]` selects only those product elements whose dept attribute value is ACC. The @ sign is used to indicate that dept is an attribute as opposed to a child element.
FLWOR Expressions

FLWOR (pronounced “flower”) stands for “for, let, where, order by, return,” the most common keywords used in the expression. FLWORs, unlike path expressions, allow you to manipulate, transform, and sort your results.

Example: Simple FLWOR

Query

```
for $prod in doc("catalog.xml")/catalog/product
where $prod/@dept = "ACC"
order by $prod/name
return $prod/name
```

Results

```
<name language="en">Deluxe Travel Bag</name>
<name language="en">Floppy Sun Hat</name>
```

We can use “let” to bind the value of a variable:

```
let $name := $prod/name
```

Adding XML elements and attributes

Any content in an element constructor that is not inside curly braces appears in the results as is. For example:

```
<h1>There are {count(doc("catalog.xml")//product)} products.</h1>
```

will return the result:

```
<h1>There are 4 products.</h1>
```

Elements can be constructed in a FLWOR return clause:

Query

```
<ul>
  for $prod in doc("catalog.xml")/catalog/product
  where $prod/@dept='ACC'
  order by $prod/name
  return <li>{$prod/name}</li>
</ul>
```

Results

```
<ul>
  <li><name language="en">Deluxe Travel Bag</name></li>
  <li><name language="en">Floppy Sun Hat</name></li>
</ul>
```

We can use the “data” function to extract values from XML tags:

```
return <li>(data($prod/name))</li>
```
Perform Joins

We can join a document with another, for example this “order.xml” file:

```xml
<order num="50299432" date="2015-09-15" cust="0221A">
  <item dept="WOM" num="557" quantity="1" color="navy"/>
  <item dept="ACC" num="563" quantity="1"/>
  <item dept="ACC" num="443" quantity="2"/>
  <item dept="MEN" num="784" quantity="1" color="white"/>
  <item dept="MEN" num="784" quantity="1" color="gray"/>
  <item dept="WOM" num="557" quantity="1" color="black"/>
</order>
```

We explicitly join catalog.xml and order.xml:

**Query**

```xml
for $item in doc("order.xml")/item
let $name := doc("catalog.xml")/product[number - $item/@num]/name
return <item num=""${$item/@num}""
  name=""{$name}""
  quan=""{$item/@quantity}""/>
```

**Results**

```xml
<item num="557" name="Fleece Pullover" quan="1"/>
<item num="563" name="Floppy Sun Hat" quan="1"/>
<item num="443" name="Deluxe Travel Bag" quan="2"/>
<item num="784" name="Cotton Dress Shirt" quan="1"/>
<item num="784" name="Cotton Dress Shirt" quan="1"/>
<item num="557" name="Fleece Pullover" quan="1"/>
```

Aggregating and Grouping

“Group by” operations are built-in in FLWOR expressions:

**Query**

```xml
xquery version "3.0";
for $i in doc("order.xml")/item
let $d := $i/@dept
group by $d
order by $d
return <department name=""{$d}"" totQuantity=""{sum($i/@quantity)}""/>
```

**Results**

```xml
<department name="ACC" totQuantity="3"/>
<department name="MEN" totQuantity="2"/>
<department name="WOM" totQuantity="2"/>
Chapter 2 - XQuery Foundations

XQuery and XPath

XPath started out as a language for selecting elements and attributes from an XML document while traversing its hierarchy and filtering out unwanted content. XQuery and XPath overlap to a very large degree. They have the same data model and the same set of built-in functions and operators. **XPath is essentially a subset of XQuery.** XQuery has a number of features that are not included in XPath, such as FLWORs, user-defined functions, and XML constructors. This is because these features are not relevant to selecting, but instead have to do with structuring, sorting query results or more complex programming. The two languages are consistent in that any expression that is valid in both languages evaluates to the same value using both languages.

Processing Queries

As input, a query supports XML documents, collections of documents, or other data presented with an XML frontend. In addition to XML input, it is also possible to query JSON documents and simple text files.

A query is made up of three parts: a version declaration, a prolog, and a body, in that order.

- The optional version declaration says what version of XQuery you are using, for example 3.1. If the version declaration does not appear, the processor makes an assumption about the version based on which version it supports.
- The optional query prolog contains various declarations that are used in evaluating the query. This includes namespace declarations, variable declarations, user-defined functions, and other settings. These declarations are discussed in relevant sections throughout the book and summarized in Chapter 12.
- The query body contains one or more expressions, separated by commas, that indicate what the query should return.

Example: A Query with a prolog

**Query**

```xml
xquery version "3.1";

declare namespace html = "http://www.w3.org/1999/xhtml";
declare variable $orderTitle := "Order Report";

<h1>{$orderTitle}</h1>,
for $item in doc("order.xml")/item
order by $item/@num
return <p>{data($item/@num)}</p>
```

**Results**

```xml
<h1>Order Report</h1>
<p>443</p>
<p>557</p>
```
A query is not evaluated in a vacuum. The query context consists of a collection of information that affects the evaluation of the query. Some of these values can be set by the processor outside the scope of the query, while others are set in the query prolog. The context may include such values as:

- The context item, which determines the context for path expressions in the query, i.e., what input documents are being queried
- Current date and time, and the implicit time zone
- Names and values of variables that are bound outside the query or in the prolog
- External function libraries built into your processor

The query processor returns a sequence of values as the results. The results are often XML elements (or entire documents), but a query could also return a result that is not XML, for example a string or an array of integers.

Writing the results to a physical XML document is known as serialization. In your query you can specify that you want the output serialized as XML, HTML, XHTML, text, or JSON.

The XQuery Data Model (XDM)

- **Node**
  An XML construct such as an element or attribute. An XML document is made up of an hierarchy of nodes.
  There are two kinds of values for a node: string and typed.
  - All nodes have a string value. The string value of an element node is its character data content and that of all its descendant elements concatenated together. If an element has no content, its string value is a zero-length string. The string value of an attribute node is simply the attribute value. The string value of a node can be accessed using the string function.
  - Element and attribute nodes also both have a typed value that takes into account their type, if any. An element or attribute might have a particular type if it has been validated with a schema. The typed value of a node can be accessed using the data function.

- **Atomic value**
  A simple data value with no markup associated with it.
  Atomic values can be extracted from element or attribute nodes using the string and data functions. They can also be created from literals in queries.

- **Function**
  Starting in version 3.0, a function is a full-fledged item in the data model. Maps and arrays are subtypes of functions. These more advanced use cases are described in Chapters 23 and 24.
- **Item**
  A generic term that refers to either a node, atomic value, or function.
- **Sequence**
  An ordered list of zero, one, or more items.
  - The most common way that sequences are created is that they are returned from expressions or functions that return sequences. For example:
    
    ```
    doc("catalog.xml")/catalog/product
    ```
  - A sequence can also be created explicitly using a sequence constructor. The syntax of a sequence constructor is a series of values, delimited by commas, surrounded by parentheses. For example, the expression `(1, 2, 3)` creates a sequence consisting of those three atomic values.
  - A sequence with only one item is known as a singleton sequence. There is no difference between a singleton sequence and the item it contains.
  - A sequence with zero items is known as the empty sequence (different from `""` or a zero value).
  - Sequences cannot be nested within other sequences; there is only one level of items.
# Chapter 3 - Expressions: XQuery building blocks

## Categories of Expressions

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Operators or keywords</th>
<th>Chapter/Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>The basics: literals, variables, function calls, and parenthesized expressions</td>
<td></td>
<td>Chapter 3</td>
</tr>
<tr>
<td>Comparison</td>
<td>Comparison based on value, node identity, or document order</td>
<td>=, !=, &lt;, &lt;=, &gt;, &gt;=, eq, ne, lt, le, gt, ge, is, &lt;, &gt;&gt;</td>
<td>“Comparison Expressions”</td>
</tr>
<tr>
<td>String concatenation</td>
<td>Concatenating two strings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String construction</td>
<td>Interspersing strings with expressions</td>
<td>`{ }`</td>
<td>“String Constructors”</td>
</tr>
<tr>
<td>Conditional</td>
<td>If-then-else expressions</td>
<td>if, then, else</td>
<td>“Conditional (if-then-else) Expressions”</td>
</tr>
<tr>
<td>Switch</td>
<td>Switch expressions</td>
<td>switch, case</td>
<td>“Switch Expressions”</td>
</tr>
<tr>
<td>Logical</td>
<td>Boolean and/or operators</td>
<td>or, and</td>
<td>“Logical (and/or) Expressions”</td>
</tr>
<tr>
<td>Path</td>
<td>Selecting nodes from XML documents</td>
<td>/, //, . . . , child: : , etc.</td>
<td>Chapter 4</td>
</tr>
<tr>
<td>Simple map</td>
<td>Iterating through items</td>
<td>!</td>
<td>“The Simple Map Operator”</td>
</tr>
<tr>
<td>Constructor</td>
<td>Adding XML to the results</td>
<td>&lt;, &gt;, element, attribute</td>
<td>Chapter 5</td>
</tr>
<tr>
<td>FLWOR</td>
<td>Controlling the selection and processing of nodes</td>
<td>for, let, where, order by, group by, count, return</td>
<td>“FLWOR Expressions”</td>
</tr>
</tbody>
</table>
### General Comparisons

<table>
<thead>
<tr>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>doc(&quot;catalog.xml&quot;)/catalog/product[2]/name = 'Floppy Sun Hat'</code></td>
<td>true</td>
</tr>
<tr>
<td><code>doc(&quot;catalog.xml&quot;)/catalog/product[4]/number &lt; 500</code></td>
<td>false</td>
</tr>
<tr>
<td><code>1 &gt; 2</code></td>
<td>false</td>
</tr>
<tr>
<td><code>() = (1, 2)</code></td>
<td>false</td>
</tr>
<tr>
<td><code>(2, 5) &gt; (1, 3)</code></td>
<td>true</td>
</tr>
<tr>
<td><code>1 = &quot;2&quot;</code></td>
<td>Error XPTY0004</td>
</tr>
<tr>
<td><code>(1, &quot;a&quot;) = (2, &quot;b&quot;)</code></td>
<td>Error XPTY0004</td>
</tr>
</tbody>
</table>

General comparisons can operate on sequences of more than one item, as well as empty sequences. If one or both of the operands is a sequence of more than one item, the expression evaluates to true if the corresponding value comparison is true for any combination of two items from the two sequences.

### Value Comparisons

Value comparisons differ fundamentally from general comparisons in that they can only operate on single atomic values.

<table>
<thead>
<tr>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(1, 2) eq (1, 2)</code></td>
<td>Error XPTY0004</td>
</tr>
</tbody>
</table>
Node Comparisons

Another type of comparison is the node comparison. To determine whether two operands are actually the same node, you can use the is operator. Each of the operands must be a single node or the empty sequence. If one of the operands is the empty sequence, the result is the empty sequence.

declare variable $n1 := doc("catalog.xml")/catalog/product[2];
declare variable $n2 := doc("catalog.xml")/catalog/product[3];

<table>
<thead>
<tr>
<th>Example</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n1 is $n2</td>
<td>false</td>
</tr>
<tr>
<td>$n1 is $n1</td>
<td>true</td>
</tr>
<tr>
<td>$n1 is doc(&quot;catalog.xml&quot;)//product[number = 563]</td>
<td>true</td>
</tr>
<tr>
<td>$n1/@dept is $n2/@dept</td>
<td>false</td>
</tr>
</tbody>
</table>

If-then-else expressions

Syntax:

--- if ( <expr> ) then <expr> else <expr> ---

Switch expressions

Syntax:

- switch ( <expr> )

  case <expr> return <expr>

- default return <expr>

Example:

switch ($department)
  case "ACC" return "Accessories"
  case "MEN"
  case "WMN" return "Clothing"
  default return "Other"
Chapter 4 - Navigating XML by using Paths

Path expressions

A path expression is made up of one or more steps that are separated by a slash (/) or double slashes (//). Each is evaluated on the current "context node" and changes it.

<table>
<thead>
<tr>
<th>Example</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc(&quot;catalog.xml&quot;)/catalog</td>
<td>The catalog element that is the outermost element of the document</td>
</tr>
<tr>
<td>doc(&quot;catalog.xml&quot;)//product</td>
<td>All product elements anywhere in the document</td>
</tr>
<tr>
<td>doc(&quot;catalog.xml&quot;)//product/@dept</td>
<td>All dept attributes of product elements in the document</td>
</tr>
<tr>
<td>doc(&quot;catalog.xml&quot;)/catalog/*</td>
<td>All child elements of catalog</td>
</tr>
<tr>
<td>doc(&quot;catalog.xml&quot;)/catalog/*/number</td>
<td>All number elements that are grandchildren of catalog</td>
</tr>
</tbody>
</table>

Steps in a path can simply be primary expressions like function calls (doc("catalog.xml")) or variable references ($catalog). Any expression that returns nodes can be on the lefthand side of the slash operator.

Another kind of step is the axis step, which allows you to navigate around the XML node hierarchy. There are two kinds of axis steps:

- Forward step
  This step selects descendants or nodes appearing after the context node (or the context node itself).
- Reverse step
  This step selects ancestors or nodes appearing before the context node (or the context node itself).

Axis step syntax:

```
<axis-name> :: <name-test> [ <expr> ]
```

In addition to having an axis, each axis step has a node test. The node test indicates which of the nodes (by name or node kind) to select, along the specified axis. You can select by node name and by node kind.
Node name test syntax:

\[
\text{<prefix> : } \quad \text{Q<namespace-name>}
\]

Node kind test syntax:

\[
\text{node()}
\]

\[
\text{text()}
\]

\[
\text{comment()}
\]

\[
\text{document-node()}
\]

\[
\text{processing-instruction(<name>)}
\]

\[
\text{<element-attribute-test>}
\]

Some axes and steps can be abbreviated:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>self::node()</td>
</tr>
<tr>
<td>..</td>
<td>parent::node()</td>
</tr>
<tr>
<td>@</td>
<td>attribute::</td>
</tr>
<tr>
<td>//</td>
<td>/descendant-or-self::node()</td>
</tr>
</tbody>
</table>

In addition to axis steps, other expressions can be used as steps, for example:

<table>
<thead>
<tr>
<th>Example</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>product/(number</td>
<td>name)</td>
</tr>
<tr>
<td>product/* except number</td>
<td>All children of product except number. See “Combining Results” for more information on the</td>
</tr>
<tr>
<td>product/(if (desc) then desc else name)</td>
<td>For each product element, the desc child if it exists; otherwise, the name child.</td>
</tr>
<tr>
<td>product/substring(name, 1, 30)</td>
<td>A sequence of xs:string values that are substrings of product names.</td>
</tr>
</tbody>
</table>

The last step (and only the last step) in a path may return atomic values rather than nodes.
Predicates

Predicates are used in a path expression to filter the results to contain only items that meet specific criteria. For example:

<table>
<thead>
<tr>
<th>Example</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>product[name = &quot;Floppy Sun Hat&quot;]</td>
<td>All product elements that have a name child whose value is equal to Floppy Sun Hat</td>
</tr>
<tr>
<td>product[number &lt; 500]</td>
<td>All product elements that have a number child whose value is less than 500</td>
</tr>
<tr>
<td>product[@dept = &quot;ACC&quot;]</td>
<td>All product elements that have a dept attribute whose value is ACC</td>
</tr>
<tr>
<td>product[desc]</td>
<td>All product elements that have at least one desc child</td>
</tr>
<tr>
<td>product[@dept]</td>
<td>All product elements that have a dept attribute</td>
</tr>
<tr>
<td>product[@dept]/number</td>
<td>All number children of product elements that have a dept attribute</td>
</tr>
</tbody>
</table>

Another use of predicates is to specify the numeric position of an item within the sequence of items currently being processed. These are sometimes called, predictably, positional predicates. For example, if you want the fourth product in the catalog, you can specify: `doc("catalog.xml")/catalog/product[4]`

Examples of position in predicates with and without positional predicates:

<table>
<thead>
<tr>
<th>Example</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>product[position() = 2]</td>
<td>The second product child of catalog</td>
</tr>
<tr>
<td>product[position() &gt; 1]</td>
<td>All product children of catalog after the first one</td>
</tr>
<tr>
<td>product[last()-1]</td>
<td>The second to last product child of catalog</td>
</tr>
<tr>
<td>product[last()]</td>
<td>The last product child of catalog</td>
</tr>
<tr>
<td>*[2]</td>
<td>The second child of catalog, regardless of name</td>
</tr>
<tr>
<td>product[3]/*[2]</td>
<td>The second child of the third product child of catalog</td>
</tr>
</tbody>
</table>

Multiple predicates can be chained together to filter items based on more than one constraint. For example: `doc("catalog.xml")/catalog/product[number < 500][@dept = "ACC"]`
Chapter 5 - Adding elements and attributes to results

You can insert your own XML elements and attributes into the query results by using XML constructors. There are two kinds of XML constructors: direct constructors, which use familiar XML-like syntax, and computed constructors, that allow you to generate dynamically the XML names used in the results.

Direct constructors

A direct element constructor is a constructor of the first kind; it specifies an XML element (optionally with attributes) by using XML-like syntax.
Example:

Query
<html>
  <h1>Product Catalog</h1>
  <ul>
    for $prod in doc("catalog.xml")/catalog/product
    return <li>number: [data($prod/number)], name: [data($prod/name)]</li>
  </ul>
</html>

Results
<html>
  <h1>Product Catalog</h1>
  <ul>
    <li>number: 557, name: Fleece Pullover</li>
    <li>number: 563, name: Floppy Sun Hat</li>
    <li>number: 443, name: Deluxe Travel Bag</li>
    <li>number: 784, name: Cotton Dress Shirt</li>
  </ul>
</html>

Computed constructors

A computed element constructor uses the keyword element, followed by a name and some content in curly braces.

\[
\text{element} \rightarrow \langle \text{element-name} \rangle \{ \langle \text{name - expr} \rangle \{ \langle \text{content - expr} \rangle \} \}
\]

A computed attribute constructor has syntax identical to a computed element constructor, except that it uses the keyword attribute.

\[
\text{attribute} \rightarrow \langle \text{attribute-name} \rangle \{ \langle \text{name - expr} \rangle \{ \langle \text{value - expr} \rangle \} \}
\]

Example:

Query
for $dept in distinct-values(doc("catalog.xml")/catalog/product/@dept)
return element {$dept}
  {doc("catalog.xml")/catalog/product[@dept=$dept]/name}
Results

<WMN>
    <name language="en">Fleece Pullover</name>
</WMN>

<ACC>
    <name language="en">Floppy Sun Hat</name>
    <name language="en">Deluxe Travel Bag</name>
</ACC>

<MEN>
    <name language="en">Cotton Dress Shirt</name>
</MEN>
Chapter 6 - Selecting and joining using FLWORs

The overall syntax of FLOWRs is:

```
<for-clause> <let-clause> <window-clause> return <expr> →
```

**For clause:**

```
for $var-name [as <sequence-type> allowing empty at $var-name] in <expr> →
```

In `<expr>` we can also use range expressions such as 1 to 3.

**Let clause:**

```
let $var-name := <expr> →
```

where clause:

```
where <expr> →
```

The return clause consists of the return keyword followed by the single expression that is to be returned. It is evaluated once for each iteration, assuming the where clause evaluated to true.

**Quantified expressions**

A quantified expression determines whether some or all of the items in a sequence meet a particular condition. Examples:

```xml
some $dept in doc("catalog.xml")//product/@dept satisfies ($dept = "ACC")
```

Or

```xml
every $dept in doc("catalog.xml")//product/@dept satisfies ($dept = "ACC")
```
Syntax:

```
  some $\langle \text{var-name} \rangle$ in $\langle \text{expr} \rangle$ satisfies $\langle \text{expr} \rangle$
```

where every $\langle \text{sequence-type} \rangle$
Chapter 7 - Sorting and Grouping

Sorting in XQuery

Path expressions, which are most often used to select elements and attributes from input documents, always return items in document order. FLWORs by default return results based on the order of the sequence specified in the for clause, which is also often document order if a path expression was used.

You can sort data in an order other than document order by using the order by clause of the FLWOR. Therefore, in some cases it is necessary to use a FLWOR where it would not otherwise be necessary. The other alternative is to use the sort function.

The order by clause:

```
order by <expr> <order-modifier>
```

Order modifiers:

- ascending
- descending
- empty greatest
- empty least
- collation "<collation-name>"

A built-in sort function is available in version 3.1 that will sort a sequence of items. The one-argument version simply sorts the items based on their typed values. For example:

```
sort(doc("catalog.xml")//product/number)
```

will return the number elements, sorted by their contents. A second argument can be used to provide an optional collation, and a third argument can be used to provide function that generates the sort key for each item. For example:

```
sort(doc("catalog.xml")//product, (), function($product) { $product/number })
```

Grouping

Example of grouping with FLWOR:

```
xquery version "3.0";
for $item in doc("order.xml")/item
let $d := $item/@dept
group by $d
order by $d
return <department code="{$d}">
    for $i in $item
        order by $i/@num
        return $i</department>
```

Adding the group by clause changes the iteration of the FLWOR expression. Instead of iterating over the items specified in the for clause (the item elements), it is now iterating over groups of those items, so it evaluates the return clause once for every group.
The group by clause also changes the variables that were defined before it. Before the group by clause, the $item variable is bound to one item element at a time. After the group by clause, the $item variable is bound to a sequence of one or more item elements, whichever ones are in the current group.

Syntax:

```
<expr> as <sequence-type>:
```

Together with group by we can use a few aggregating functions, such as count, sum, min, max, avg.

Example:

```xml
xquery version "3.0";
for $item in doc("order.xml")/item
  group by $d := $item/@dept
  order by $d
  return
    <department code="$d"
      numItems="count($item)"
      distinctItemNums="count(distinct-values($item/@num))"
      totQuant="sum($item/@quantity)">
```

Chapter 8 - Functions

Functions are a useful feature of XQuery that allow a wide array of built-in functionality, as well as the ability to modularize and reuse parts of queries. There are two kinds of functions: built-in functions and user-defined functions.

User defined functions

Functions are defined using function declarations, which can appear either in the query prolog or in an external library.

Example of function declaration in the prolog:

```query
declare function local:discountPrice(  
    $price as xs:decimal?,  
    $discount as xs:decimal?,  
    $maxDiscountPct as xs:integer?) as xs:decimal?
{
    let $maxDiscount := ($price * $maxDiscountPct) div 100
    let $actualDiscount := min( ($maxDiscount, $discount) )
    return ($price - $actualDiscount)
};

let $prod := doc("prices.xml")/prod[1]
return local:discountPrice($prod/price, $prod/discount, 15)
```

Declaration syntax:

```
--- declare ---
| public | function - <function-name>(<param-list>) |
| private | <annotation> |

--- as <sequence-type> ---

--- external ---

; ---
```

Parameter list:

```
--- $<param-name> ---

--- as <sequence-type> ---
```
Chapter 10 - Namespaces and XQuery

There are three ways that namespaces are bound to prefixes in XQuery queries:

- Some namespaces are predeclared; no explicit namespace declaration is necessary to associate a prefix with the namespace.
- Namespace declarations can appear in the query prolog.
- Namespace declarations can appear in XML constructors

**Predeclared namespaces**

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Namespace</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml</td>
<td><a href="http://www.w3.org/XML/1998/namespace">http://www.w3.org/XML/1998/namespace</a></td>
<td>XML attributes such as xml:lang and xml:space</td>
</tr>
<tr>
<td>xs</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
<td>XML Schema built-in types and their constructors</td>
</tr>
<tr>
<td>xsi</td>
<td><a href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a></td>
<td>XML Schema instance attributes such as xsi:type and xsi:nil</td>
</tr>
<tr>
<td>fn</td>
<td><a href="http://www.w3.org/2005/xpath-functions">http://www.w3.org/2005/xpath-functions</a></td>
<td>XPath Functions Namespace: the default namespace of most built-in functions</td>
</tr>
<tr>
<td>local</td>
<td><a href="http://www.w3.org/2005/xquery-local-functions">http://www.w3.org/2005/xquery-local-functions</a></td>
<td>Functions declared in a main module that are not in a specific namespace</td>
</tr>
</tbody>
</table>

**Prolog namespace declaration**

Syntax:

```xml
declare namespace <prefix> = "<namespace-name>" ;
```

Example:

```xml
declare namespace rep = "http://datypic.com/report";
declare namespace prod = "http://datypic.com/prod";
<rep:report> { 
  doc("cat_ns.xml")//prod:product
} </rep:report>
```

You can also declare two kinds of default namespaces (for elements and functions):

```xml
declare default [element] namespace "<namespace-name>" ;
```

**Namespace declarations in direct element constructors**

Example:
Namespace Declarations in Computed Constructors

--- namespace [ <prefix> ] { - <namespace-uri-expr> - } ---

Example:

```xml
<rep:report xmlns="http://datypic.com/cat"
            xmlns:prod="http://datypic.com/prod"
            xmlns:rep="http://datypic.com/report">
  doc("cat_ns.xml")/catalog/prod:product
</rep:report>
```

```xquery
xquery version "3.0";
declare namespace rep="http://datypic.com/report";
declare default element namespace "http://datypic.com/prod";
declare namespace cat="http://datypic.com/cat";

element rep:report {
  namespace [""] ("http://datypic.com/cat"),
  namespace prod ("http://datypic.com/prod"),
  doc("cat_ns.xml")/cat:catalog/product
}
```
Chapter 11 - A closer look at types

XQuery is a strongly typed language, meaning that each function and operator is expecting its arguments or operands to be of a particular type. Data is only typed when schemas are used, otherwise it is “untyped” and the processor will automatically assume its type during operations.

Built-in types

Atomic types:

List types: a list type represents a list of possibly multiple atomic values of a particular type, known as its item type. There are three list types built into the type system: xs:IDREFS, xs:NMTOKENS, and
xs:ENTITIES. They are defined as list types with item types xs:IDREF, xs:NM_TOKEN, and xs:ENTITY, respectively.

**Union** types: a union type allows a value to be a choice among several different types, known as its member types. There is one union type built into the type system, xs:numeric, which is defined as the union of the three primitive numeric types: xs:double, xs:float, and xs:decimal.

Element, attributes and atomic values may be typed in XQuery, usually as a result of schema validation.

In many situations, Automatic type conversion is used to obtain the correct type from another one (including "untyped").

**Atomization** occurs when a function or operator expects an atomic value and receives a node instead. For example:

```xml
<e1>3</e1> + 5
```

returns the value 8 because the value 3 is extracted from the e1 element during atomization.

**Effective Boolean Value:**

It is often useful to treat a sequence as a Boolean value. For example, if you want to determine whether your catalog element contains any products whose price is less than 20, you might use the expression:

```xml
if (doc("prices.xml")//prod[price < 20])
then <bargain-bin>...</bargain-bin>
else ()
```

<table>
<thead>
<tr>
<th>Example</th>
<th>Effective boolean value</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>false</td>
</tr>
<tr>
<td>false()</td>
<td>false</td>
</tr>
<tr>
<td>true()</td>
<td>true</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;false&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;x&quot;</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td>xs:float(&quot;NaN&quot;)</td>
<td>false</td>
</tr>
<tr>
<td>(false() or false())</td>
<td>false</td>
</tr>
<tr>
<td>doc(&quot;prices.xml&quot;)/*</td>
<td>true</td>
</tr>
<tr>
<td>&lt;a&gt;false&lt;/a&gt;</td>
<td>true</td>
</tr>
<tr>
<td>&lt;a&gt;[xs:boolean(&quot;false&quot;)]&lt;/a&gt;</td>
<td>true</td>
</tr>
<tr>
<td>(false(), false(), false())</td>
<td>Error FORG0006</td>
</tr>
<tr>
<td>1, 2, 3</td>
<td>Error FORG0006</td>
</tr>
<tr>
<td>xs:date(&quot;2015-01-15&quot;)</td>
<td>Error FORG0006</td>
</tr>
<tr>
<td>[true()]</td>
<td>Error FORG0006</td>
</tr>
<tr>
<td>data( [true()] )</td>
<td>true</td>
</tr>
</tbody>
</table>
Sequence Types
A sequence type is used in a query to specify the expected type of a sequence of zero, one, or more items.

Constructors and Casting
There are two mechanisms in XQuery for explicitly changing values from one type to another: constructors and casting.

 Constructors are functions used to construct atomic values with given types. For example, the constructor `xs:date("2015-05-03")` constructs an atomic value whose type is `xs:date`. The signature of this `xs:date` constructor function is:
`xs:date($arg as xs:anyAtomicType?) as xs:date`?
There is a constructor function for each of the built-in simple types (both primitive and derived). The qualified name of the constructor is the same as the qualified name of the type.

 Casting is the process of changing a value from one type to another. The cast expression can be used to cast a value to another type. It has the same meaning as the constructor expression; it is simply a different syntax. The only difference is that it can be used with a type name that is in no namespace. For example:
`$myNum cast as xs:integer` casts the value of $myNum to the type `xs:integer`. It is equivalent to `xs:integer($myNum)`.

```
<expr> cast as <type-name>
```
Chapter 24 - Maps, Arrays and JSON

Maps

A map is a collection of key/value pairs that can be constructed, manipulated, and queried in XQuery 3.1. Each key/value pair is known as an entry. Within a map, each key is unique and the order of the entries has no particular significance.

Map constructor:

```xquery
xquery version "3.1";
map {
    "ACC": "Accessories",
    "WMN": "Women's",
    "MEN": "Men's"
}
```

Map can be used as functions for lookups:

```xquery
xquery version "3.1";
declare variable $deptnames := map {
    "ACC": "Accessories",
    "WMN": "Women's",
    "MEN": "Men's"
};
for $prod in doc("catalog.xml")/product
return <product num="${$prod/number}"
    dept-name="{$deptnames($prod/@dept)}"/>
```

Results

```xml
<product num="557" dept-name="Women's"/>
<product num="563" dept-name="Accessories"/>
<product num="443" dept-name="Accessories"/>
<product num="784" dept-name="Men's"/>
```

Arrays

An array is simply an ordered list of values. The values in an array are called its members, and they can be retrieved based on their position number.

Arrays can be constructed with two constructors: `[ "a", "b", "c" ]` or `array { "a", "b", "c" }`.

Arrays are different from sequences because they can be nested.

JSON

JSON can be remapped to XQuery constructs:

<table>
<thead>
<tr>
<th>JSON construct</th>
<th>XQuery equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Map, with an entry for each name/value pair</td>
</tr>
</tbody>
</table>
JSON can be parsed with the `json-doc` function. You can serialize the results of your query as a JSON document by choosing the serialization method `json`.

**Query**

```xml
xml version "3.1";
declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";
declare option output:method "json";
declare option output:indent "yes";
map {
    "number": 557,
    "props":
    <props>
      <length>31</length>
    </props>
}
```

**Results**

```json
{
    "number": 557,
    "props":
    "<props>
    <length>31</length>
    
    </props>"
}
```
Reading XQuery

Chapter 2

XPath: traversing its hierarchy and filtering out unwanted content. XQuery: overlap with XPath in a large degree.

Nodes

Identity and name

Identity is unique to each node and is assigned by the query processor.

It is also possible to retrieve a unique identifier for a node using the `generate-id` function.

String and typed values of nodes

- `string()`: access the string value of a node. The string value of an element node is its character data content and that of all its descendant elements concatenated together.
- `data()`: The typed value of a node can be accessed. If the type is not declared in a schema, the value is considered to be untyped.

Atomic values

Atomic values VS Nodes

All functions and operators that expect to have atomic values as their operands also accept nodes. The atomic value is automatically extracted from the node in a process known as `atomization`.

Atomic values *don't have identities.*

Sequences

`ordered` collections of items.

sequence constructor: `(, , ,)`

- Sequences *do not have names*, although they may be bound to a named variable.
- A sequence with only one item is known as a singleton sequence. There is no difference between a singleton sequence and the item it contains. Any of the functions or operators that can operate on sequences can also operate on individual items, which are treated as singleton sequences.
- Sequences cannot be nested within other sequences; there is only one level of items.
  - `(10, (20, 30), 40)` is equivalent to: `(10, 20, 30, 40)`
- Sequences have no identity.

Types

XQuery is a strongly typed language. There are a number of type conversions that happen automatically, following specified rules.

Automatic conversion works on untyped values. If it is declared to be one type, you need explicitly convert the value's type.
Chapter 3 Expressions

Every expression evaluates to a sequence, which may be a single item (for example, an atomic value or node), the empty sequence, or multiple items.

The inputs and outputs of expressions are sequences.

Whitespace

- use whitespace to separate keywords
- extra whitespace is acceptable.
- you are not required to use whitespace as a separator when using non-word symbols such as =

Variables

- identified by names that are preceded by a dollar sign $.
- a variable is bound to a particular value. That value may be any sequence, including a single item such as a node or atomic value, the empty sequence, or multiple items.

Once the variable is bound to a value, its value does not change. You cannot bind a new value to the variable as you can in most procedural languages. Instead, you must use a new variable.
Comments

- delimited by (`:` and `:`).
- These comments are ignored during processing.
- XQuery comments can contain any text, including XML markup.
- XQuery comments can appear anywhere insignificant whitespace is allowed in a query.
- XQuery comments can be nested within other XQuery comments.
- XML comments, delimited by `. Unlike XQuery comments, these comments appear in the result document.

Precedence and Parentheses

Any `and` operators are evaluated before `or` operators: `true()` and `true()` or `false()` and `false()` is the same as: `(true() and true()) or (false() and false())`

Comparison

General Comparisons

- used for comparing atomic values or nodes that contain atomic values.
- If either operand is the empty sequence, the expression evaluates to false.
- When comparing two values, their types are taken into account.

General comparisons on multi-item sequences

True: if the corresponding value comparison is true for any combination of two items from the two sequences.

Value Comparisons

- only operate on single atomic values.
  - If either operand is a sequence of more than one item, type error XPTY0004 is raised.
  - Unlike general comparisons, if either operand is the empty sequence, the empty sequence is returned.
  - The two operands must have comparable types
    - Untyped values are always treated like strings by value comparisons. This means that if you have two untyped elements that contain numbers, they will be compared as strings unless you explicitly cast them to numbers. The operators:
      - `eq` (equal to),
      - `ne` (not equal to),
      - `lt` (less than),
      - `le` (less than or equal to),
      - `gt` (greater than)
      - `ge` (greater than or equal to)

GC VS VC

General comparison operators are easier to use than value comparison operators in predicates when children are untyped or repeating.
- The down side of general comparison operators is that they also make it less likely that the processor will catch any mistakes you make.

Node Comparison

- the operator: `is`, `deep-equal`
  - `is` compares the nodes based on their identity rather than by any value they may contain.
Chapter 4 Path

- order: Path expressions return nodes in document order.
- format: (context item)/relative path. If there is no context item, this means that the path expression will be evaluated relative to the current context node, which must have been previously determined outside the expression.
- process: The context item changes with each step. A step returns a sequence that serve as the context items for evaluating the next step.

/:
- lefthand: The expression must evaluate to zero or more nodes, not atomic values or other items.
- righthand: can only evaluate to atomic values if it's the last step in a path.

Abbreviated Syntax

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>//</td>
<td>/descendant-or-self::node()</td>
<td>It can even include the direct children to represent the context node itself in predicates and in paths. You can also use the period as an argument to functions, e.g. [starts-with(., &quot;T&quot;)].</td>
</tr>
<tr>
<td>.</td>
<td>self::node()</td>
<td></td>
</tr>
<tr>
<td>..</td>
<td>parent::node()</td>
<td></td>
</tr>
<tr>
<td>@</td>
<td>attribute::</td>
<td></td>
</tr>
</tbody>
</table>

The | operator is a union operator

Last step

The last step (and only the last step) in a path may return atomic values rather than nodes.

Error XPTY0019 is raised if a step that is not the last returns atomic values. For example:

doc("catalog.xml")//product/substring(name, 1, 30)/replace(., ' ', '-')

will raise an error because the substring step returns atomic values, and it is not the last step.

Predicates

[] : used in a path expression to filter the results.

the value in it:

- anything other than a number: its **effective boolean value** is determined.
- a number: it is interpreted as the **position**.

Positions

- starts from 1.
- If you specify a number that is greater than the number of items in the context sequence, it does not raise an error; it simply does not return any nodes.

With positional predicates, the position is the position within the current sequence of items being processed, not the position of an element relative to its parent's children.

- doc("catalog.xml")/catalog/product/name[1]: refers to the first name child of each product
- doc("catalog.xml")/catalog//name[1]: same as the above
Chapter 6 FLWORs

Path expressions are useful for queries where no new elements and attributes are being constructed and the results don’t need to be sorted. A path expression can be preferable to a FLWOR because it is more compact and some implementations will be able to evaluate it faster.

FLOWER:

- for: sets up an iteration,
- let: binds a variable,
- where: filters out elements,
- order by: sort,
- return

functionalities:

- joining data from multiple sources,
- constructing new elements and attributes,
- evaluating functions on intermediate values,
- sorting results.

For

for clause is similar to loops in procedural languages such as C.

However, the iterations are considered to be in no particular order. They do not necessarily occur sequentially, one after the other. You cannot continuously append to the end of a string variable with each iteration.

Range Expression

- 1 to 3 evaluates to a sequence of integers (1, 2, 3).
- can be included within parenthesized expressions, as in (1 to 3, 6, 8 to 10).
- reverse(1 to 3) gives the descending sequences.
- (1 to 100)[. mod 2 = 0] increment by some value other than 1

If the first integer is greater than the second, as in 3 to 1, or if either operand is the empty sequence, the expression evaluates to the empty sequence.

Multiple for clauses

- the order is important.
- especially useful for joining data.
- two forms equivalent:
  - for $i$ in (1, 2) for $j$ in ("a", "b")
  - for $i$ in (1, 2), $j$ in ("a", "b")

Let

- like for, let is also for binding, but binding the whole.
- can be multiple
- Each of the let and for clauses may reference a variable bound in any previous clause.
- usages:
  - bind a variable to a value
  - to perform several functions or operations in order.
Quantified Expressions

A quantified expression is made of several parts:

- A quantifier (the keyword *some* or *every*)
- *in* clauses that bind variables to sequences
- *satisfies* clause that contains the test expression

format: (some/every) var-name (as type) in expr satisfies expr

Join

Outer joins

you can use allowing empty : e.g.

```xml
for $prod in doc("catalog.xml")//product
for $price allowing empty
    in doc("prices.xml")//prices/priceList/prod[@num = $prod/number]
return <product number="{$prod/number}" price="{$price/price}"/>
```
Chapter 7 Sorting and grouping

Sorting

order

format order by <exp1> (<order-modifier1>), <exp2> (<order-modifier2>) (, ...)

order is made up of: one or more ordering specifications, separated by commas, each of which consists of an expression: The expression can only return one value for each item. .../@* will raise an error. You can sort based on almost any expression, as long as it only returns a single item. order by (if ($item/@color) then $item/@color else "unknown")

- optional modifier:
  - ascending (default), descending
  - empty sequence
    - empty greatest: the empty sequence is greater than NaN, and NaN is greater than all other values.
    - empty least: the empty sequence is less than NaN, and NaN is less than all other values. not suitable to zero-length strings, which are always sorted before other strings.

- can be specified as the default behavior: declare default order empty greatest;

- collation

Collation

All the values returned by a single ordering specification expression must have comparable types.

Untyped values are treated like strings.

stable keyword

use the order of input sequence to return the values in the same order for the query

sort function

three arguments:

- to be sorted items
- collation
- provide comparing functions sort(doc("catalog.xml")//product, (), function($product) {
  $product/number })

Document order

If you have a sequence of nodes that are not in document order, but you want them to be, you can simply use the expression:

mySequence/.

The / operator means that it is a path expression, which always returns nodes in document order.

when using the resulting sequence in another expression, the results may be re-sorted to document order.

let $sortedProds := for $prod in doc("catalog.xml")//product
  order by $prod/number
**Grouping**

**group by clause**

format: group by <var-name1> (as <sequence-type1>):=<expr1> (collation1) (, <var-name2> (as <sequence-type2>):=<expr2> (collation2) ... )

The grouping variable name is required, and must be bound to a value. You can include multiple grouping specifications, separated by commas.

**Aggregating values**

- **in-line functions**: count, sum, min, max, avg
- **user-defined functions**: e.g. declare namespace functx = "http://www.functx.com";
  
declare function functx:min-non-empty-string
  ($strings as xs:string*) as xs:string? {
    min($strings[. != ''])
  };
  
```xml
8 of 10
```
Chapter 10 Namespace

the use of URIs for namespaces serves simply as a name. No parser or internet visit happens.

Namespace Declarations and Scope

The scope of a namespace declaration is the element on which it appears and any attributes or descendants of that element.

Override:

- If a namespace declaration appears in the scope of another namespace declaration with the same prefix, it overrides it. Not recommended.
- It is also possible to override the default namespace, which can be useful when a document consists of several subtrees in different namespaces.

Undeclaring:

from Namespaces 1.1, "" undeclares a namespace. e.g. xmlns:prod="", then prod cannot be used. xmlns="" This undeclares the default namespace.

Namespace Declarations in Queries

There are three ways that namespaces are bound to prefixes in XQuery queries:

- Some namespaces are predeclared; no explicit namespace declaration is necessary to associate a prefix with the namespace.
- Namespace declarations can appear in the query prolog.
- Namespace declarations can appear in XML constructors:

```
<rep:report xmlns:default-url
xmlns:prod=prod-url
xmlns:rep=report-url> { 
  doc("cat_ns.xml")/catalog/prod:product
} </rep:report>
```

Results

Prolog

e.g.

- declare namespace prod = "http://datypic.com/prod";
- declare default element namespace "http://datypic.com/cat";
- declare default function namespace "http://datypic.com/funclib";

Chapter 11 Types

XQuery is a strongly typed language, meaning that each function and operator is expecting its arguments or operands to be of a particular type.