

Semantic Web

XSL and XQuery

F. Abel, N. Henze, and D. Krause

IVS Semantic Web Group

30.10.2008

1 XSL Extensible Stylesheet Language

- XSLT
 - XML to result tree
 - Understanding A Template
 - Applying Style Procedurally
 - Conditional Processing
 - Numbering
 - Sorting
- XSL-FO
 - Common Formatting Objects
 - Basic properties
 - Inheritance
- Combine XSLT and XSL-FO

2 XQuery

- Expressions
- FLWOR

Why do we need stylesheets?

Note: Slides in this section are based on Paul Grosso, Norman Walsh: "XSL Concepts and Practical Use", available online at <http://nwalsh.com/docs/tutorials/xsl/xsl/slides.html>

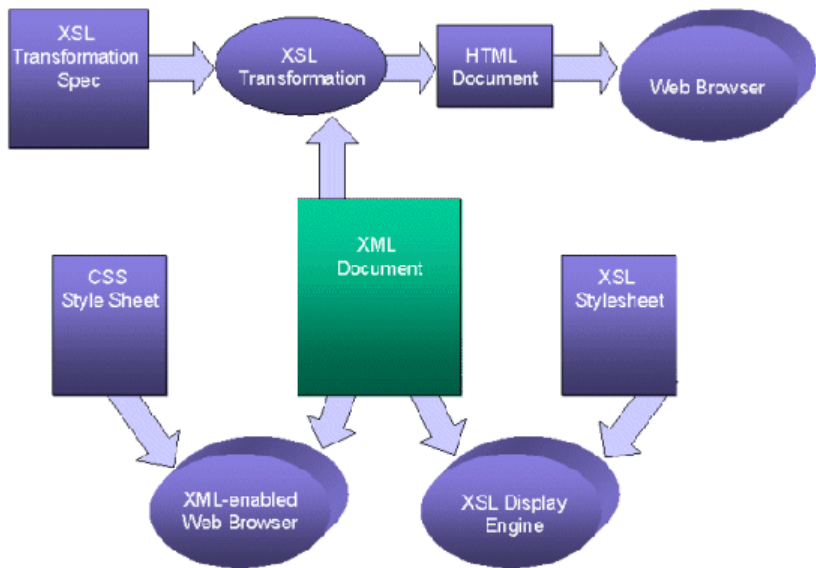
Why do we need stylesheets?

- XML is not a fixed tag set (like HTML)
- XML by itself has no (application) semantics
- A generic XML processor has no idea what is "meant" by the XML
- XML markup does not (usually) include formatting information
- The information in an XML document may not be in the form in which it is desired to present it
 - e.g. order of items
- Therefore there must be something in addition to the XML document that provides information on how to present or otherwise process the XML

Advantages of separating content from style

- Contrary to when style information is hard-coded into the content, separation of style from content allows for the same data to be presented in different ways. This enables:
 - reuse of fragments of data: the same content should look different in different contexts
 - multiple output formats: different media (paper, online), different sizes (manuals, reports), different classes of output devices (workstations, hand-held devices)
 - styles tailored to the reader's preference (e.g., accessibility): print size, color, simplified layout for audio readers
 - standardized styles: corporate stylesheets can be applied to the content at any time
 - freedom from style issues for content authors: technical writers needn't be concerned with layout issues because the correct style can be applied later

Options for displaying XML



What does a Stylesheet do?

- A stylesheet specifies the presentation of XML information using two basic categories of techniques:
 - An optional transformation of the input document into another structure
 - A description of how to present the transformed information (i.e., a specification of what properties to associate to each of the various parts of the transformed information)

- Transformation capabilities include:
 - generation of constant text
 - suppression of content
 - moving text (e.g., exchanging the order of the first and last name)
 - duplicating text (e.g., copying titles to make a table of contents)
 - sorting
 - more complex transformations that "compute" new information in terms of the existing information

- Description of how to present the (possibly transformed) data includes three levels of formatting information:
 - Specification of the general screen or page (or even audio) layout
 - Assignment of the transformed content into basic "content container types" (e.g., lists, subsections, inline text)
 - Specification of formatting properties (spacing, margins, alignment, fonts, etc.) for each resulting "container"

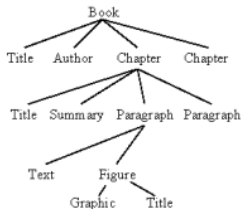
The XSL family (<http://www.w3.org/Style/XSL/>) logically consists of three component languages which are described in three W3C (World Wide Web Consortium) Recommendations:

- **XPATH** (you should know this)
- **XSL Transformations** (XSLT)
- **XSL Formatting Objects**(XSL-FO)

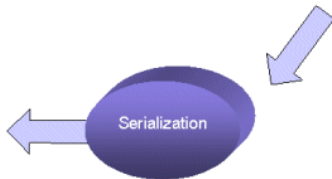
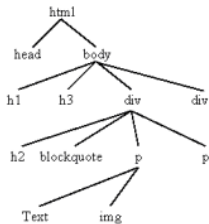
XML to result tree

An XSLT "stylesheet" transforms the input (source) document's tree into a structure called a *result tree* consisting of result objects

XML Source Tree



XHTML Result Tree



```
<html>  
<head>...</head>  
<body>  
<h1></h1>  
<h3></h3>  
.....  
</body>  
</html>
```

- The result tree's structure is that of an XML document, and its objects correspond to elements with attributes
- The result tree's structure and "tag set" can match that of any XML document or doctype. In particular, the result tree could be:
 - result tree is easily written out as an HTML document
 - result tree is easily written out as an XML document in this other doctype (for some further application-specific processing)
 - result tree's structure (and element and attribute names) matches the set of *formatting objects* and *formatting properties* defined by the (non-transformation) part of XSL
- Serialization of the result tree is not necessary for further processing of the result tree.

- An XSL stylesheet basically consists of a set of templates
- Each template "matches" some set of elements in the source tree and then describes the contribution that the matched element makes to the result tree
- Generally, elements in a stylesheet in the "xsl" namespace are part of the XSLT language, and non-xsl elements within a template are what get put into the result tree

Structure of a Stylesheet

- XSLT Stylesheets are XML documents; namespaces are used to identify semantically significant elements
- Most stylesheets are stand-alone documents rooted at `<xsl:stylesheet>` or `<xsl:transform>`. It is possible to have "single template" stylesheet/documents
- `<xsl:stylesheet>` and `<xsl:transform>` are completely synonymous
- Note that it is the mapping from namespace abbreviation to URI that is important, not the literal namespace abbreviation "xsl:" that is used most commonly

Example (A Stylesheet)

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  ...
</xsl:stylesheet>
```

Example (A Transformation Sheet)

```
<eg:transform
  xmlns:eg="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  ...
</eg:transform>
```

Example (Stylesheet in a document)

```
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <head>
    <title>Silly Example</title>
  </head>
  <body>
    <h1>Silly Example</h1>
    <p>
      You'd probably use extension elements, or something
      more interesting in real life:
      3+4 is <xsl:value-of select="3+4"/>.
    </p>
  </body>
</html>
```

Generally, elements in a stylesheet in the "xsl" namespace are part of the XSLT language, and non-xsl elements within a template are what is put into the result tree

Example (An external XSLT Stylesheet)

The XML input file:

```
<?xml version='1.0'?>
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
<para>This is a <emphasis>test</emphasis>.</para>
```

The XSLT style.xsl file:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
  <xsl:template match="para">
    <p><xsl:apply-templates/></p>
  </xsl:template>
  <xsl:template match="emphasis">
    <i><xsl:apply-templates/></i>
  </xsl:template>
</xsl:stylesheet>
```

Example (An external XSLT Stylesheet)

Output:

```
<?xml version="1.0" encoding="utf-8"?>  
<p>This is a <i>test</i>.</p>
```

Most templates have the following form:

```
<xsl:template match="emphasis">
  <i><xsl:apply-templates/></i>
</xsl:template>
```

- The whole `<xsl:template>` element is a *template*
- The *match pattern* determines where this template applies
 - match pattern are XPATH expressions
- Literal result element(s) come from non-XSL namespace(s)
- XSLT elements come from the XSL namespace

Example with nested statements

Have a look at the following XML file:

```
<doc>
<para>This is a <emphasis>test</emphasis>.
<emphasis>Nested <emphasis>emphasis</emphasis></emphasis>.</para>
</doc>
```

A stylesheet for this XML file:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="doc">
  <html><head><title>A Document</title></head>
  <body><xsl:apply-templates/></body></html>
</xsl:template>
<xsl:template match="para">
  <p><xsl:apply-templates/></p>
</xsl:template>
<xsl:template match="emphasis">
  <i><xsl:apply-templates/></i>
</xsl:template>
<xsl:template match="emphasis/emphasis">
  <b><xsl:apply-templates/></b>
</xsl:template> </xsl:stylesheet>
```

```
<html> <head> <title>A Document</title> </head>
<body>
<p>This is a <i>test</i>
<i>Nested <b>emphasis</b></i></p>
</body>
</html>
```

- The problem of multiple patterns that match is handled by conflict resolution:
 - Matching templates from imported modules are not considered if there is a matching template in the current module
 - Matching templates with a lower priority are not considered. The default priority is determined as follows:
 - Unqualified child or attribute names have a priority of 0.
 - Processing-instructions with a target have a priority of 0.
 - A namespace-qualified "*" child or attribute name has a priority of -0.25.
 - An unqualified "*" has a priority of -0.5
 - Any other template has a default priority of 0.5
 - Template priority may be specified explicitly with the *priority* attribute on `<xsl:template>`

Example

- "emphasis", "html:p", and "@foo" have a priority of 0
 - "html:*" has a priority of -0.25
 - "*" has a priority of -0.5
 - "para/emphasis" has a priority of 0.5
 - "emphasis/emphasis" has a priority of 0.5
 - "emphasis[@role]" has a priority of 0.5
-
- It is technically an error if the conflict resolution process yields more than one template, however, XSLT processors may (silently) recover from this error by selecting the template that occurs last in the stylesheet.
 - Effectively, this means that stylesheet template order is the final arbiter.

A series of templates is created, such that each template explicitly selects and processes the necessary elements.

- **<xsl:for-each>**

```
<xsl:for-each select="row">  
  <tr><xsl:apply-templates/></tr>  
</xsl:for-each>
```

xsl:for-each can be nested!

Example (for-each)

```
<?xml version='1.0'?>
<table>  <row><entry>a1</entry><entry>a2</entry></row>
         <row><entry>b1</entry><entry>b2</entry></row>
         <row><entry>c1</entry><entry>c2</entry></row>  </table>
```

foreach.xsl

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>
<xsl:template match="table">
  <table>
    <xsl:for-each select="row">
      <tr>
        <xsl:for-each select="entry">
          <td><xsl:apply-templates/></td>
        </xsl:for-each>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>
```

Example (for-each)

foreach.html

```
<table>
<tr>
<td>a1</td><td>a2</td>
</tr>
<tr>
<td>b1</td><td>b2</td>
</tr>
<tr>
<td>c1</td><td>c2</td>
</tr>
</table>
```

tr: table row, td: table data

■ Literal Result Elements

Any element in a template rule that is not in the XSL (or other extension) namespace is copied literally to the result tree

■ `<xsl:text>`

The content of `<xsl:text>` elements is copied directly to the result tree; whitespace is preserved by default

```
<xsl:text>Literal result text</xsl:text>
```

■ `<xsl:value-of>`

Inserts the value of an expression into the result tree, converting it to a string first if necessary

```
<xsl:value-of select="$count + 1"/>
```

■ `<xsl:copy>` and `<xsl:copy-of>`

Copies the current node or, in the case of `xsl:copy-of`, the selected nodes, into the result tree without first converting them to a string

```
<xsl:copy-of select="title"/>
```

■ `<xsl:attribute>`

Adds the named attribute to the nearest containing element

```
<table>
```

```
  <xsl:if test="@pgwide='1'">
```

```
    <xsl:attribute name="width">100%</xsl:attribute>
```

```
  </xsl:if>
```

```
  ...
```

```
</table>
```

- **<xsl:if>**

Simple conditional (no "else")

```
<xsl:if test="$somecondition">
  <xsl:text>
    this text only gets used if $somecondition is true()
  </xsl:text>
</xsl:if>
```

- **<xsl:choose>**

Select among alternatives with `<xsl:when>` and `<xsl:otherwise>`

```
<xsl:choose>
  <xsl:when test="$count > 2"><xsl:text>, and </xsl:text></xsl:when>
  <xsl:when test="$count > 1"><xsl:text> and </xsl:text></xsl:when>
  <xsl:otherwise><xsl:text> </xsl:text></xsl:otherwise>
</xsl:choose>
```

- **To report errors, use `<xsl:message>`**

```
<xsl:message>
  <xsl:text>Error: no ID found for linkend: </xsl:text>
  <xsl:value-of select="@linkend"/>
  <xsl:text>.</xsl:text>
</xsl:message>
```

- The `<xsl:number>` element performs two functions:
- It evaluates a numeric expression and converts the result into a formatted string:

```
<xsl:number value="3" format="A. "/>
```

```
<xsl:number value="count(listitem)" format="01"/>
```

- It counts elements in the *source* tree and converts the result into a formatted string:

```
<xsl:number count="listitem" format="i. "/>
```

```
<xsl:number count="chapter" from="book" level="any"  
format="1. "/>
```

```
<xsl:number count="h1|h2|h3" level="multiple"  
from="chapter|appendix" format="1."/>
```

- The `<xsl:sort>` element sorts a set of nodes according to the criteria specified:

```
<xsl:apply-templates select="row">  
  <xsl:sort data-type="number" select="entry[2]" />  
</xsl:apply-templates>
```

- It can appear as a child of `<xsl:apply-templates>` or `<xsl:for-each>`. It can also be nested.

Example (Sorting Example)

sort.xml

```
<doc><para>Here's a table of sales:</para><table>
<row><cell>3000</cell><cell>Widgets 'R' Us</cell></row>
<row><cell>10000</cell><cell>Widgets for Dummies</cell></row>
<row><cell>101</cell><cell>101 Uses for a Dead Widget</cell></row>
</table></doc>
```

sort.xsl

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:import href="element.xsl"/>
<xsl:template match="table">
  <table>
    <xsl:apply-templates select="row">
      <xsl:sort data-type="number" select="./cell[1]"/>
    </xsl:apply-templates>
  </table>
</xsl:template>
</xsl:stylesheet>
```

Example (Sorting Example)

sort.html

```
<p>Here's a table of sales:</p>
<table>
<tr>
<td>101</td><td>101 Uses for a Dead Widget</td>
</tr>
<tr>
<td>3000</td><td>Widgets 'R' Us</td>
</tr>
<tr>
<td>10000</td><td>Widgets for Dummies</td>
</tr>
</table>
```

Try it on your own:

`http://xalan.apache.org/index.html`

XSL Formatting Objects (XML-FO)

- Similar to CSS
- XML conform language to describe the visualization of a document
- normally not directly used
 - XML → XSLT → XSL-FO → XSL-FO processor → visualization
- also called XSL!

Formating capabilities

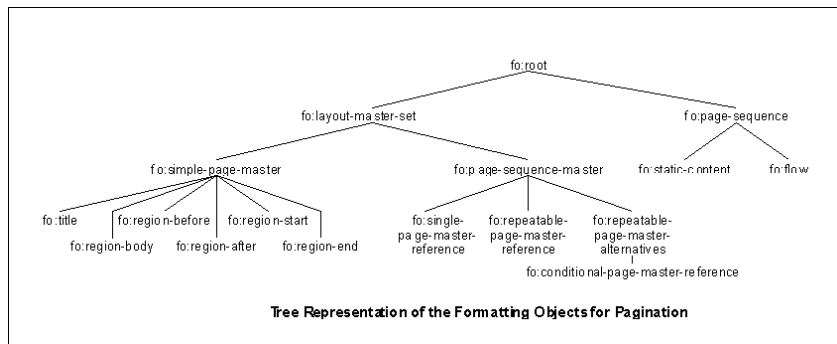
- XSL FO formatting capabilities in XSL 1.0 are approximately the union of: HTML + CSS capabilities
- most high quality print output capabilities including internationalization
- Not included are complex page layouts (e.g., magazine and newspaper layout), complex layout-driven formatting (e.g., copyfitting and complex floats), and looseleaf pagination (change page production)

Example (Sample file)

```
<?xml version="1.0" encoding="utf-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:simple-page-master master-name="only">
      <fo:region-body region-name="xsl-region-body"
        margin="0.7in" padding="6pt" />
      <fo:region-before region-name="xsl-region-before"
        extent="0.7in" />
      <fo:region-after region-name="xsl-region-after"
        extent="0.7in" />
    </fo:simple-page-master>
  </fo:layout-master-set>

  <fo:page-sequence master-reference="only">
    <fo:flow flow-name="xsl-region-body">
      <fo:block>Hello
        <fo:inline font-style="italic">italic</fo:inline>
        World</fo:block>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

- *simple-page-master* – a page master layout
- *page-sequence* – a major part (such as front or body) in which the basic page layout may differ from other parts



Example (Two page layout)

```
<fo:layout-master-set>
  <fo:simple-page-master master-name="even" margin-left="1.5cm" />
  <fo:simple-page-master master-name="odd" margin-left="3.5cm" />

  <fo:page-sequence-master master-name="chapter">
    <fo:repeatable-page-master-alternatives>
      <fo:conditional-page-master-reference
        master-reference="odd" page-position="rest" odd-or-even="odd"/>
      <fo:conditional-page-master-reference
        master-reference="even" page-position="rest" odd-or-even="even"/>
    </fo:repeatable-page-master-alternatives>
  </fo:page-sequence-master>
</fo:layout-master-set>

<fo:page-sequence master-reference="chapter">
  ...
</fo:page-sequence>
```

- *flow* – a chapter- or section-like division within a page-sequence
- *block* – a subsection (or title or block quote, etc.)
- *inline* – e.g., a font change within a subsection
- list FOs – *list-block*, *list-item*, *list-item-label*, *list-item-body*

Example (Lists in XSL-FO)

```
<fo:list-block>
  <fo:list-item>
    <fo:list-item-label>
      <fo:block>&#x2022;</fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>List item contents.</fo:block>
    </fo:list-item-body>
  </fo:list-item>
</fo:list-block>
```

- *wrapper* – a "transparent" object usable as either a block or inline object that has no effect other than to provide a place to hang inheritable properties
- *graphic* – references an external graphic object
- *table FOs* – mostly analogous to the standard (CALS, OASIS, HTML) table models

XSL-FO properties are defined as XML attributes:

eg. `margin-left="1.5cm"`

- font properties
- margin and spacing properties
- border and padding properties
- keeps/breaks
- horizontal alignment/justification
- indentation
- more formatting object specific properties

There are more than 240 properties specified in XSL-FO 1.0!

- Properties are either inheritable or not—inheritance occurs within the result tree
- Inheritable properties may not have any effect on a particular formatting object on which they are set but their values may affect descendant objects
- Properties have some initial (default) value
- Therefore, there is always a value defined for every property on every formatting object (either inherited, defaulted, or assigned), though not every property will be relevant to every formatting object
- The description of each FO says which properties are relevant to it and how

Try it on your own:

<http://xmlgraphics.apache.org/fop/>

Examples:

<http://www.dpawson.co.uk/xsl/examples.html>

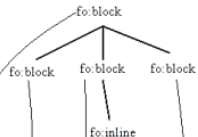
Combine XSLT and XSL-FO

XML Source

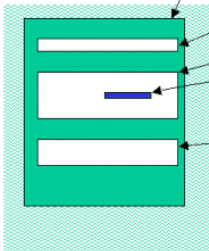
```
<chapter>  
<title>...</title>  
<para>...  
  <emph>...</emph>  
</para>  
<para>...</para>  
</chapter>
```



FO Result Tree



Area Structure



Example

xslfo.xml

```
<chapter>
<title>Chapter title</title>
<section>
<title>First section title</title>
<para>Section one's first paragraph.</para>
<para>Section one's second paragraph.</para>
</section>
<section>
<title>Second section title</title>
<para>Section two's only paragraph.</para>
</section>
</chapter>
```

Example

xslfo.xsl

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">

  <xsl:template match="/">
  <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <fo:layout-master-set>
      <fo:simple-page-master master-name="only">
        <fo:region-body region-name="xsl-region-body"
          margin="0.7in" padding="6pt" />
      </fo:simple-page-master>
    </fo:layout-master-set>
  <fo:page-sequence master-reference="only">>
  <fo:flow flow-name="xsl-region-body">
    <xsl:apply-templates/>
  </fo:flow>
  </fo:page-sequence>
  </fo:root>
</xsl:template>
```

Example

```
<xsl:template match="chapter">
  <fo:block break-before="page">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="chapter/title">
  <fo:block text-align="center" space-after="8pt" space-before="16pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="section">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="section/title">
  <fo:block text-align="center" space-after="6pt"
    space-before="12pt" space-before.precedence="0"
    space-after.precedence="3">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Example

```
<xsl:template match="paragraph[1]" priority="1">
  <fo:block text-indent="0pc" space-after="7pt"
            space-before.minimum="6pt" space-before.optimum="8pt"
            space-before.maximum="10pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

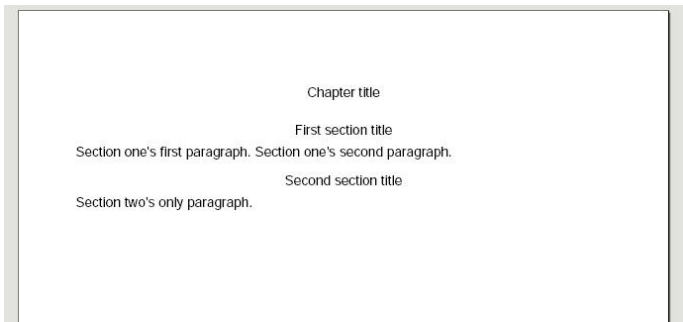
<xsl:template match="paragraph">
  <fo:block text-indent="2pc" space-after="7pt"
            space-before.minimum="6pt" space-before.optimum="8pt"
            space-before.maximum="10pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

</xsl:stylesheet>
```

Generating output:

- `xalan -in xslfo.xml -xsl xslfo.xsl -out xslfo.fo`
- `fop xslfo.fo xslfo.pdf`

xslfo.pdf:



- XML query language
- aims on the database community
 - similar syntax than SQL

Note: Slides in this section are based on

<http://www.brics.dk/~amoeller/XML/querying/>

Path Expressions

XQuery supports XPath expressions.

Example (An XPATH expression in XQuery)

```
document("recipes.xml")//recipe[title="Ricotta Pie"]//ingredient[@amount]
```

Element constructors

Example (Construction of new XML elements)

```
<employee empid="12345">  
  <name>John Doe</name>  
  <job>XML specialist</job>  
  <deptno>187</deptno>  
  <salary>125000</salary>  
</employee>
```

Example (Variables in XQuery)

```
<employee empid="{ $id }">  
  <name>{ $name }</name>  
  { $job }  
  <deptno>{ $deptno }</deptno>  
  <salary>{ $SGMLspecialist+100000 }</salary>  
</employee>
```

Here the variables \$id, \$name, and \$job must be bound to appropriate XML fragments or strings.

FLWOR (For-Let-Where-Order-Return) expressions

- pronounced "flower"
- generalizes SELECT-FROM-HAVING-WHERE-ORDER BY from SQL

Example (Variables in XQuery)

```
for $d in document("depts.xml")//deptno
let $e := document("emps.xml")//employee[deptno = $d]
where count($e) >= 10
order by avg($e/salary) descending
return
  <big-dept>
    { $d,
      <headcount>{count($e)}</headcount>,
      <avgsal>{avg($e/salary)}</avgsal>
    }
  </big-dept>
```

General rules:

- for and let may be used many times in any order
- only one where is allowed
- many different sorting criteria can be specified

Note the difference between for and let:

```
for $x in /company/employee
```

generates a list of bindings of \$x to each employee element in the company, but:

```
let $x := /company/employee
```

generates a single binding of \$x to the list of employee elements in the company

Example (Join two documents)

```
for $p IN document("www.irs.gov/taxpayers.xml")//person
for $n IN document("neighbors.xml")//neighbor[ssn = $p/ssn]
return
  <person>
    <ssn> { $p/ssn } </ssn>
    { $n/name }
    <income> { $p/income } </income>
  </person>
```

List expressions

- constant lists: (7, 9, <thirteen/>)
- integer ranges: i to j
- XPath expressions, like all named children of the context node: name
- concatenation: ,
- set operators: | (or union), intersect, except
 - if lists are used as sets the order will be preserved, duplicates are removed
- functions: remove, index-of, count, avg, max, min, sum, distinct-values ...

Example (lists each publisher and the average price of their books)

```
for $p in distinct-values(document("bib.xml")//publisher)
let $a := avg(document("bib.xml")//book[publisher = $p]/price)
return
  <publisher>
    <name>{ $p/text() }</name>
    <avgprice>{ $a }</avgprice>
  </publisher>
```

Conditional expressions XQuery supports if-then-else constructions

Example (extracts from the holdings of a library the titles and either editors or authors.)

```
for $h in document("library.xml")//holding
return
  <holding>
    { $h/title,
      if ($h/@type = "Journal")
      then $h/editor
      else $h/author
    }
  </holding>
```

Quantified expressions

- `some-in-satisfies` – element satisfies condition if **some** elements in the list satisfy the condition
- `every-in-satisfies` – element satisfies condition if **all** elements in the list satisfy the condition

Example (finds the titles of all books which mention both sailing and windsurfing in the same paragraph)

```
for $b in document("bib.xml")//book
where some $p in $b//paragraph satisfies
  (contains($p,"sailing") AND contains($p,"windsurfing"))
return $b/title
```

Example (finds the titles of all books which mention sailing in every paragraph)

```
for $b in document("bib.xml")//book
where every $p in $b//paragraph satisfies
  contains($p,"sailing")
return $b/title
```

Datatype expressions

- XQuery supports all datatypes from XML Schema, both primitive and complex types
- Constant values can be written:
 - as literals (like `string`, `integer`, `float`)
 - as constructor functions (`true()`, `date("2001-06-07")`)
 - as explicit casts (`verb+cast as xsd:positiveInteger(47)+`)
- Arbitrary XML Schema documents can be imported into a query
- An instance of operator allows runtime validation of any value relative to a datatype or a schema
- A typeswitch operator allows branching based on types