Part X

XQuery—Querying XML Documents
Outline of this part

1. XQuery—Declarative querying over XML documents
   - Introduction
   - Preliminaries

2. Iteration (FLWORs)
   - For loop
   - Examples
   - Variable bindings
   - where clause
   - FLWOR Semantics
   - Variable bindings
   - Constructing XML Fragments
   - User-Defined Functions
XQuery—Introduction

- XQuery is a truly **declarative** language specifically designed for the purpose of querying XML data.
- As such, XML assumes the role that SQL occupies in the context of relational databases.
- XQuery exhibits properties known from database (DB) languages as well as from (functional) programming (PL) languages.
- The language is designed and formally specified by the W3C XQuery Working Group ([http://www.w3.org/XML/XQuery/](http://www.w3.org/XML/XQuery/)).
  - The first working draft documents date back to February 2001. The XQuery specification has become a W3C Recommendation in January 2007.
  - Members of the working group include Dana Florescu\textsuperscript{DB}, Ioana Manolescu\textsuperscript{DB}, Phil Wadler\textsuperscript{PL}, Mary Fernández\textsuperscript{DB+PL}, Don Chamberlin\textsuperscript{DB}, Jérôme Siméon\textsuperscript{DB}, Michael Rys\textsuperscript{DB}, and many others.

\textsuperscript{34}Don is the “father” of SQL.
XQuery is a hybrid exhibiting features commonly found in *programming* as well as *database query* languages:

- **Programming language** features:
  - explicit iteration and variable bindings (*for*···*in*, *let*···*in*)
  - recursive, user-defined functions
  - regular expressions, strong [static] typing
  - ordered sequences (much like lists or arrays)

- **Database query language** features:
  - filtering
  - grouping, joins } expressed via nested *for* loops
History of XQuery

The diagram illustrates the history of XML query languages and standards:

- 1998: XSLT, XQL, XML-QL
- 1999: XPath 1.0, Quilt
- 2000: XML, DOM
- 2001: XPath 2.0, XQuery 1.0, XML-Schema

Other query languages and standards:
- SQL
- OQL
- UnQL
- Lorel

The timeline shows the development of XML query languages and standards from 1998 to 2001, with XPath 1.0 and 2.0, XQuery 1.0, and XML-Schema becoming standardized.

[illustration © C. Türker]
**XQuery—Preliminaries**

- **Remember:** XPath is part of XQuery (as a sublanguage).
- Some constructs that have not previously been discussed, yet are not within the core of our focus on XQuery include:
  - **Comparisons:** any XQuery expression evaluates to a sequence of items. Consequently, many XQuery concepts are prepared to accept sequences (as opposed to single items).

### General Comparisons

The **general comparison** $e_1 \theta e_2$ with

$$\theta \in \{=, \neq, <, \leq, >, \geq\}$$

yields $\text{true}()$ if any of the items in the sequences $e_{1,2}$ compare true (existential semantics).
Comparisons

General comparison examples

\[(1,2,3) > (2,4,5) \Rightarrow \text{true()}\]
\[(1,2,3) = 1 \Rightarrow \text{true()}\]
\[(\) = 0 \Rightarrow \text{false()}\]
\[2 \leq 1 \Rightarrow \text{false()}\]
\[(1,2,3) \neq 3 \Rightarrow \text{true()}\]
\[(1,2) \neq (1,2) \Rightarrow \text{true()}\]
\[\text{not}((1,2) = (1,2)) \Rightarrow \text{false()}\]

Value comparisons

The six \textbf{value comparison operators} \textit{eq}, \textit{ne}, \textit{lt}, \textit{le}, \textit{ge}, \textit{gt} compare \textit{single items by value} (atomization!):

\[2 \text{ gt } 1.0 \Rightarrow \text{true()}\]
\[<x>42</x> \text{ eq } <y>42</y> \Rightarrow \text{true()}\]
\[(0,1) \text{ eq } 0 \Rightarrow \text{ ? } \quad \text{(type error)}\]
More on comparisons . . .

**Note:** The existential semantics of the general comparison operators may lead to unexpected behavior:

### Surprises

\[(1,2,3) = (1,3) \Rightarrow \text{true()}^{35}\]
\[\text{("2",1) = 1} \Rightarrow \text{true()} \text{ or } \\not\approx \text{ (impl. dependent)}\]
# Node comparisons

## Node comparison

... based on *identity* and *document order*:

- $e_1$ is $e_2$ nodes $e_{1,2}$ identical?
- $e_1 << e_2$ node $e_1$ before $e_2$?
- $e_1 >> e_2$ node $e_1$ after $e_2$?

## Node comparison examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;x&gt;42&lt;/x&gt;</code> eq <code>&lt;x&gt;42&lt;/x&gt;</code></td>
<td>true()</td>
</tr>
<tr>
<td><code>&lt;x&gt;42&lt;/x&gt;</code> is <code>&lt;x&gt;42&lt;/x&gt;</code></td>
<td>false()</td>
</tr>
<tr>
<td><code>root(e_1)</code> is <code>root(e_2)</code></td>
<td>nodes $e_{1,2}$ in same tree?</td>
</tr>
<tr>
<td><code>let $a := &lt;x&gt;&lt;y/&gt;&lt;/x&gt;</code> return $a &lt;&lt; $a/y`</td>
<td>true()</td>
</tr>
</tbody>
</table>
Working with sequences

XQuery comes with an extensive **library of builtin functions** to perform common computations over sequences:

### Common sequence operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td><code>count((0,4,2))</code></td>
<td>3</td>
</tr>
<tr>
<td>max</td>
<td><code>max((0,4,2))</code></td>
<td>4</td>
</tr>
<tr>
<td>subsequence</td>
<td><code>subsequence((1,3,5,7),2,3)</code></td>
<td>(3,5,7)</td>
</tr>
<tr>
<td>empty</td>
<td><code>empty((0,4,2))</code></td>
<td><code>false()</code></td>
</tr>
<tr>
<td>exists</td>
<td><code>exists((0,4,2))</code></td>
<td><code>true()</code></td>
</tr>
<tr>
<td>distinct-values</td>
<td><code>distinct-values((4,4,2,4))</code></td>
<td>(4,2)</td>
</tr>
<tr>
<td>to</td>
<td><code>(1 to 10)[. mod 2 eq 1]</code></td>
<td>(1,3,5,7,9)</td>
</tr>
</tbody>
</table>

See [W3C](http://www.w3.org/TR/xpath-functions/).
Arithmetics

Only a few words on arithmetics—XQuery meets the common expectation here. Points to note:

1. Infix operators: +, -, *, div, idiv (integer division),
2. operators first atomize their operands, then perform promotion to a common numeric type,
3. if at least one operand is (), the result is ()

Examples and pitfalls

```xml
<x>1</x> + 41 ⇒ 42.0
(()) * 42 ⇒ ()
(1,2) - (2,3) ⇒ ↗ (type error)
x - 42 ⇒ ./child::x-42 (use x - 42)
x/y ⇒ ./child::x/child::y (use x div y)
```
XQuery Iteration: FLWORs

- Remember that XPath steps perform **implicit iteration**: in \(cs/e\), evaluation of \(e\) is iterated with ‘.’ bound to each item in \(cs\) in turn.
- XPath subexpressions aside, **iteration in XQuery is explicit** via the **FLWOR** (‘“flower”’) construct.
  
  - The versatile FLWOR is used to express
    
    - nested iteration,
    - joins between sequences (of nodes),
    - groupings,
    - orderings beyond document order, *etc.*

  - In a sense, FLWOR assumes the role of the **SELECT–FROM–WHERE** block in SQL.
Explicit iteration

Explicit iteration is expressed using the `for···in` construct:\(^3^6\)

\[
\text{for } v \ [\text{at } p] \ \text{in } e_1 \\
\text{return } e_2
\]

If \( e_1 \) evaluates to the sequence \((x_1, \ldots, x_n)\), the loop body \( e_2 \) is evaluated \( n \) times with variable \( v \) bound to each \( x_i \) [and \( p \) bound to \( i \)] in order. The results of these evaluations are concatenated to form a single sequence.
Iteration

Iteration examples

\[
\text{for } x \in (3,2,1) \quad \Rightarrow \quad (3, "\,*", 2, "\,*", 1, "\,*")
\]
\[
\text{return } (x, "\,*")
\]
\[
\Rightarrow \quad (3,2,1,"\,*")
\]

\[
\text{for } x \in (3,2,1) \quad \Rightarrow \quad (3,2,1,"\,*")
\]
\[
\text{return } x, "\,*"
\]
\[
\Rightarrow \quad (3,2,1,"\,*")
\]

\[
\text{for } x \in (3,2,1) \quad \Rightarrow \quad (3,"a",3,"b",
\]
\[
\text{return } \text{for } y \in ("a","b") \quad \Rightarrow \quad 2,"a",2,"b",
\]
\[
\text{return } (x,y)
\]
\[
\Rightarrow \quad 1,"a",1,"b"
\]

FLWOR: Abbreviations

\[
\text{for } v_1 \in e_1 \quad \Rightarrow \quad \text{for } v_1 \in e_1, \quad \text{for } v_1 \in e_1,
\]
\[
\text{return } \quad \text{return } e_3 \quad \text{return } e_3
\]
\[
\text{for } v_2 \in e_2 \quad \Rightarrow \quad \text{for } v_2 \in e_2 \quad \text{for } v_2 \in e_2
\]
\[
\text{return } e_3
\]
FLWOR: Iteration via `for`···`in`

**Purpose of this query Q?**

```xml
max( for $i$ in $cs$/descendant-or-self::*[not(*)]
    return count($i/ancestor::*))
```

A sample $cs$

```
      a
     /|
   b  c
   /|
 d  f
 /|
 e  g  h
```

“Annotated” sample $cs$

```
      a
     /|
   b  c
   /|
 d  f
 /|
 e  g  h
```

**Answer**
**FLWOR: Iteration via for...in**

Return every other item in sequence

These queries both return the items at odd positions in the input sequence $e$:

```
for $i$ in (1 to count($e$))[. mod 2 eq 1]
return $e[$i$]
```

```
for $i$ at $p$ in $e$
return if ($p$ mod 2)
  then $e[$p$]
  else ()
```

Remember: $ebv(0) = false()$. 
FLWOR: Variable Binding via `let· · · :=`

Note that in the examples on the last slide, expression $e$ is re-evaluated $\text{count}(e)/2$ times although $e$ is constant in the loop.

**Variable bindings**

The result of evaluating an expression $e$ may be bound to a variable $\$v$ via `let`:

\[
\begin{align*}
\text{let } \$v & := e_1 \\
\text{return } e_2
\end{align*}
\]

evaluates $e_2$ with free occurrences of $\$v$ replaced by $e$.

- `for` and `let` clauses may be freely intermixed.
FLWOR: Variable Binding via \texttt{let}· · · :=

<table>
<thead>
<tr>
<th>Iteration vs. variable binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $x$ in (3,2,1) \Rightarrow (3,&quot;<em>&quot;,2,&quot;</em>&quot;,1,&quot;*&quot;)</td>
</tr>
<tr>
<td>let $x := (3,2,1)$ \Rightarrow (3,2,1,&quot;*&quot;)</td>
</tr>
</tbody>
</table>

“Every other item” revisited (flip back two slides)

The following hoists the constant $e$ out of the loop body:

\[
\text{let } \$seq := e \\
\text{return for } \$i \text{ at } \$p \text{ in } \$seq \\
\text{return if } (\$p \text{ mod } 2) \\
\text{then } \$seq[\$p] \\
\text{else ()}
\]
Adding a where clause

Inside loop bodies, the idiom if (p) then e else () is so common that FLWOR comes with a SQL-like where clause to address this.

A where clause

If ebv(p) evaluates to false() under the current variable bindings, the current iteration does not contribute to the result:

\[
\text{for } \$v \text{ in } e_1 \text{ where } p \text{ return } e_2 \]

\[
\equiv
\text{for } \$v \text{ in } e_1 \text{ return if } (p) \text{ then } e_2 \text{ else } ()
\]
Explicit vs. implicit iteration

XPath: implicit iteration

a[@b = "foo"]/c[2]/d[@e = 42]

Equivalent nested FLWOR blocks

for $a in a
where $a/@b = "foo"
return for $c at $p in $a/c
where $p = 2
return for $d in $c/d
where $d/@e = 42
return $d

**NB.** Unlike the XPath step operator `/`, `for` does not change the context item `'.'`. 
FLWOR: Reorder iteration result via order by

In a FLWOR block for $v$ in $e_1$ return $e_2$, the order of $e_1$ determines the order of the resulting sequence.

Reordering via order by

In the FLWOR block

```
for $v$ in $e_1$
    order by $e_3$ [ascending|descending] [empty greatest|least]
return $e_2$
```

the value (atomization!) of $e_3$ determines the order in which the bindings of $v$ are used to evaluate $e_2$. 
FLWOR: Reordering examples

An order by “no-op”: reordering by sequence order

for $x$ at $p$ in (5,3,1,4,2)  
order by $p$  
return $x$  
⇒ (5,3,1,4,2)

All bound variables in scope in order by

for $x$ at $p$ in (5,3,1,4,2)  
order by $p + x$  
return $x$  
⇒ (1,3,5,2,4)

Reordering as in SQL’s ORDER BY

for $x$ in (5,3,1,4,2)  
order by $x$  
return $x$  
⇒ (1,2,3,4,5)
FLWOR: Reordering examples

Value-based reordering of an XPath step result

This query reorders the result of the XPath location step descendant::b based on (string) value. Which result is to be expected?

```
let $a := <a>
  <b id="0">42</b>
  <b id="1">5</b>
  <b id="2"/>
  <b id="3">3</b>
  <b id="4">1</b>
</a>
for $b in $a/descendant::b
order by $b/text() empty greatest
return $b/@id
```

Answer
In the W3C XQuery specification, the interaction of the five clauses of a FLWOR (for–let–where–order by–return) block is formally explained by means of a tuple space:

- **Size** of tuple space ≡ number of iterations performed by FLWOR block.
- The fields of the tuples represent, for each iteration,
  1. for/let variable bindings,
  2. the outcome of the where clause,
  3. the value of the reordering criterion, and
  4. the value returned by the return clause.

Let us exemplify this here because our own relational compilation scheme for FLWOR blocks resembles the tuple space idea.
FLWOR semantics: tuple space (1)

Sample FLWOR block

```plaintext
for $x$ at $p$ in reverse(1 to 10)
let $y := x * x$
where $y <= 42$
order by 5 - $p$
return ($p,$x)
```

1 Complete tuple space

<table>
<thead>
<tr>
<th>$x$</th>
<th>$p$</th>
<th>$y$</th>
<th>where</th>
<th>order by</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>100</td>
<td>false</td>
<td>4</td>
<td>(1,10)</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>81</td>
<td>false</td>
<td>3</td>
<td>(2,9)</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>64</td>
<td>false</td>
<td>2</td>
<td>(3,8)</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>49</td>
<td>false</td>
<td>1</td>
<td>(4,7)</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>36</td>
<td>true</td>
<td>0</td>
<td>(5,6)</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>25</td>
<td>true</td>
<td>-1</td>
<td>(6,5)</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>16</td>
<td>true</td>
<td>-2</td>
<td>(7,4)</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
<td>true</td>
<td>-3</td>
<td>(8,3)</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>4</td>
<td>true</td>
<td>-4</td>
<td>(9,2)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>true</td>
<td>-5</td>
<td>(10,1)</td>
</tr>
</tbody>
</table>
Filtering: where clause ($y \leq 42$)

<table>
<thead>
<tr>
<th>$x$</th>
<th>$p$</th>
<th>$y$</th>
<th>where</th>
<th>order by</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>100</td>
<td>false</td>
<td>4</td>
<td>(1,10)</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>81</td>
<td>false</td>
<td>3</td>
<td>(2,9)</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>64</td>
<td>false</td>
<td>2</td>
<td>(3,8)</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>49</td>
<td>false</td>
<td>1</td>
<td>(4,7)</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>36</td>
<td>true</td>
<td>0</td>
<td>(5,6)</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>25</td>
<td>true</td>
<td>-1</td>
<td>(6,5)</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>16</td>
<td>true</td>
<td>-2</td>
<td>(7,4)</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
<td>true</td>
<td>-3</td>
<td>(8,3)</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>4</td>
<td>true</td>
<td>-4</td>
<td>(9,2)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>true</td>
<td>-5</td>
<td>(10,1)</td>
</tr>
</tbody>
</table>
FLWOR semantics: tuple space (3)

3. Reordering: order by clause

<table>
<thead>
<tr>
<th>$x$</th>
<th>$p$</th>
<th>$y$</th>
<th>where</th>
<th>order by</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>true</td>
<td>-5</td>
<td>(10, 1)</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>4</td>
<td>true</td>
<td>-4</td>
<td>(9, 2)</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
<td>true</td>
<td>-3</td>
<td>(8, 3)</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>16</td>
<td>true</td>
<td>-2</td>
<td>(7, 4)</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>25</td>
<td>true</td>
<td>-1</td>
<td>(6, 5)</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>36</td>
<td>true</td>
<td>0</td>
<td>(5, 6)</td>
</tr>
</tbody>
</table>

4. To emit the final result, scan the tuple space in the order specified by the order by column, and concatenate the return column entries:

\[(10,1,9,2,8,3,7,4,6,5,5,6)\].

Observation: some values have been computed, but never used ...
FLWOR: populate tuple space lazily (1)

Sample FLWOR block

```plaintext
for $x$ at $p$ in reverse(1 to 10)
let $y := x \times x$
where $y \leq 42$
order by 5 - $p$
return ($p, x$)
```

1 Populate variable bindings only

<table>
<thead>
<tr>
<th>$p$</th>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>81</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>
FLWOR: populate tuple space lazily (2)

2 Evaluate: where clause ($y <= 42$)

<table>
<thead>
<tr>
<th>$x$</th>
<th>$p$</th>
<th>$y$</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>100</td>
<td>false</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>81</td>
<td>false</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>64</td>
<td>false</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>49</td>
<td>false</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>36</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>25</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>16</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>true</td>
</tr>
</tbody>
</table>

3 Prune tuples

<table>
<thead>
<tr>
<th>$x$</th>
<th>$p$</th>
<th>$y$</th>
<th>where</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>36</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>25</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>16</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>true</td>
</tr>
</tbody>
</table>
FLWOR: populate tuple space lazily (3)

4 Evaluate: order by clause

<table>
<thead>
<tr>
<th>$x$</th>
<th>$p$</th>
<th>$y$</th>
<th>order by</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>25</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>16</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
<td>-3</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>4</td>
<td>-4</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>-5</td>
</tr>
</tbody>
</table>

5 Normalize order by column, evaluate return clause

<table>
<thead>
<tr>
<th>$x$</th>
<th>$p$</th>
<th>$y$</th>
<th>position()</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>36</td>
<td>6</td>
<td>(5,6)</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>25</td>
<td>5</td>
<td>(6,5)</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>16</td>
<td>4</td>
<td>(7,4)</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
<td>3</td>
<td>(8,3)</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>4</td>
<td>2</td>
<td>(9,2)</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>(10,1)</td>
</tr>
</tbody>
</table>
Variable bindings: Variables are not variable!

"Imperative" XQuery

Evaluate the expression

```xml
let $x :=
  <x><y>12</y>
  <y>10</y>
  <y>7</y>
  <y>13</y>
</x>
let $sum := 0
for $y in $x//y
let $sum := $sum + $y
return $sum
```

Equivalent query

```xml
let $x :=
  <x><y>12</y>
  <y>10</y>
  <y>7</y>
  <y>13</y>
</x>
for $y in $x//y
return 0 + $y
```

- let-bound variables are named values and thus immutable.
- Obtain equivalent query via textual replacement (lhs $\rightarrow$ rhs).\(^{37}\)

\(^{37}\)Not valid if rhs value depends on a node constructor!
Constructing XML fragments

- XQuery expressions may construct nodes with new identity of all 7 node kinds known in XML:
  - document nodes, elements, attributes, text nodes, comments, processing instructions (and namespace nodes).

- Since item sequences are flat, the nested application of node constructors is the only way to hierarchically structure values in XQuery:
  - Nested elements may be used to group or compose data, and, ultimately,
  - XQuery may be used as an XSLT replacement, i.e., as an XML transformation language.
Direct node constructors

XQuery node constructors come in two flavors:

1. **direct** constructors and
2. **computed** constructors.

### Direct constructors

The syntax of **direct constructors** exactly matches the **XML syntax**: any well-formed XML fragment $f$ also is a correct XQuery expression (which, when evaluated, yields $f$).

**Note:** Text content and CDATA sections are both mapped into text nodes by the XQuery data model ("CDATA isn’t remembered.")
Direct element constructors

“CDATA isn’t remembered”

\[
\begin{align*}
\langle x \rangle & \langle ![CDATA[foo & bar]] \rangle \langle /x \rangle \\
\equiv & \langle x \rangle foo & \text{amp; bar} \langle /x \rangle \\
\end{align*}
\]

XQuery

- The **tag name** of a direct constructor is constant, its **content**, however, may be computed by any XQuery expression enclosed in curly braces `{⋯}`.

**Computed element content**

\[
\begin{align*}
\langle x \rangle & 4\{ \text{max}((1,2,0)) \} \langle /x \rangle \\
\Rightarrow & \langle x \rangle 42 \langle /x \rangle \\
\end{align*}
\]

- Double curly braces (``{{ or {}}}``) may be used to create content containing literal curly braces.
Computed element constructors

**Definition**

In a **computed element constructor**

\[
\text{element} \{e_1\} \{e_2\}
\]

expression \(e_1\) (of type string or QName) determines the element name, \(e_2\) determines the sequence of nodes in the element’s content.

**Example: computed element name and content**

\[
\text{element} \{ \text{string-join}(("foo","bar"),"-") \} \{ 40+2 \} \\
\Rightarrow <\text{foo-bar}>42</\text{foo-bar}>
\]
An application of computed element constructors: i18n

Consider a dictionary in XML format (bound to variable $\text{dict}$) with entries like

\[
\text{<entry word="address">}
\text{    <variant lang="de">Adresse</variant>}
\text{    <variant lang="it">indirizzo</variant>}
\text{</entry>}
\]

We can use this dictionary to "translate" the tag name of an XML element $e$ into Italian as follows, preserving its contents:

```xml
  element
    { $\text{dict/entry[@word=name($e)]/variant[@lang="it"]} }
    { $e/@*, $e/node() }
```
Direct and computed attribute constructors

- In **direct attribute constructors**, computed content may be embedded using curly braces.

  **Computed attribute content**

  \[
  \begin{align*}
  &\langle x\ a="\{(4,2)\}\"/\rangle \quad \Rightarrow \quad \langle x\ a="4\ 2"/\rangle \\
  &\langle x\ a="\{\"\ b=\'}\}\"/\rangle \quad \Rightarrow \quad \langle x\ b="\}"\ a="\{"/\rangle \\
  &\langle x\ a="\'\"\ b='\"\"/\rangle \quad \Rightarrow \quad \langle x\ a="\'\"\ b="\\quot;"/\rangle \\
  \end{align*}
  \]

- A **computed attribute constructor** attribute \{e_1\} \{e_2\} allows to construct *parent-less* attributes (impossible in XML) with computed names and content.

  **A computed and re-parented attribute**

  ```
  let $a := attribute \{"a"\} \{ sum((40,2)) \}
  return <x>{ $a }</x>
  ```
Text node constructors

Text node construction

Text nodes may be constructed in one of three ways:

1. Characters in element content,
2. via `<![CDATA[···]]>`, or
3. using the computed text constructor `text {e}`.

Content sequence `e` is atomized to yield a sequence of type `anyAtomicType*`. The atomic values are converted to type `string` and then concatenated with an intervening "\". If `e` is `()`, no text node is constructed—the constructor yields `()`.
Examples: computed text node constructor

Explicit semantics of text node construction text \{e\}

\[
\begin{align*}
\text{if } \text{empty}(e) \\
\text{then } () \\
\text{else text } \{ \text{string-join(for } i \text{ in data}(e) \\
\quad \text{return string}(i), \\
\quad "\_" \}) \}
\end{align*}
\]

Text node construction examples

\[
\begin{align*}
\text{text } \{ (1,2,3) \} & \equiv \text{text } \{ "1 2 3" \} \\
\text{let } n := <x> \\
\quad <y/> <z/> \\
\quad </x> \text{name}(.) \\
\text{return } <t>\{ \text{text } \{n\} \} </t>
\end{align*}
\]

⇒ <t>x y z</t>
XML documents vs. fragments

Unlike XML **fragments**, an XML **document** is rooted in its **document node**. The difference is observable via XPath:

1. The context node for the first expression above is the document node for document `xy.xml`.
2. A document node may be constructed via `document { e }`.

```xml
<xy.xml>
  <x>
    <y/>
  </x>
</xy.xml>
```

- `doc("xy.xml")/*` ⇒ `<x><y/></x>`
- `<x><y/></x>/` ⇒ `<y/>`

Creating a document node

- `(document { <x><y/></x> })/` ⇒ `<x><y/></x>`
Processing element content

- The XQuery element constructor is quite flexible: the content sequence is not restricted and may have type item*.
- Yet, the content of an element needs to be of type node*:
  1. **Consecutive literal characters** yield a single text node containing these characters.
  2. Expression enclosed in `{···}` are evaluated.
  3. **Adjacent atomic values** are cast to type string and collected in a single text node with intervening "".
  4. A node is copied into the content **together with its content**. All copied nodes receive a **new identity**.
  5. Then, **adjacent text nodes** are merged by concatenating their content. Text nodes with content '"" are dropped.
Example: processing element content

Evaluate the expression below

```
count(
    <x>Fortytwo{40 + 2}{ "foo",3.1415,<y><z/></y>,
        ("","!")[1] </x>/node())
```

Solution:

The constructed node is

```
<x>
  text
    y
      z
    "Fortytwo42foo_3.1415"
</x>
```
Well-formed element content

XML fragments constructed by XQuery expressions are subject to the XML rules of **well-formedness**, e.g.,

- no two attributes of the same element may share a name,
- attribute nodes precede any other element content.\(^{38}\)

### Violating the well-formedness rules

```xml
<xml>
  let $id := "id"
  return
  element x {
    attribute ${id} {0},
    attribute {"id"} {1} }
  </x>
</xml>  \(\Rightarrow \) (type error)
```

---

\(^{38}\) The content type needs to be a subtype of attribute(*), (element(*)|text()|⋯)*.
Construction generates new node identities

**element x \{e\}: Deep subtree copy**

\[
\begin{align*}
\text{let } e & := \langle a \rangle \langle b / \rangle \langle c \rangle \langle y \rangle \text{foo} \langle / y \rangle \langle / c \rangle \langle / a \rangle \\
\text{let } x & := \text{element x } \{ \ e \ \} \\
\text{return exactly-one}(e // y) & \text{ is exactly-one}(x // y)
\end{align*}
\]

- Node constructors have **side effects**.

**Observing node identity**

\[
\begin{align*}
\text{let } x & := \langle x / \rangle \\
\text{return } x & \text{ is } x \Rightarrow \text{true()} \\
\text{let } d & := \text{doc}(\text{uri}) \\
\text{return } d & \text{ is } d \Rightarrow \text{true()}
\end{align*}
\]
Construction establishes document order

Result of the following query?

```xml
let $x := <x/>
let $y := <y/>
let $unrelated := ($x, $y)
let $related := <z>{$unrelated}</z>/*
return ($unrelated[1] << $unrelated[2],
```

Solution
### Construction: pair join partners

#### A join query

```
let $a := <a><b><c>0</c><c>1</c><c>2</c></b></a>

let $x := <x><z id="2">two</z><z id="0">zero</z>
               <y><z id="0">zero’</z><z id="3">three</z></y></x>

for $c in $a/b/c
for $z in $x//z[@id eq $c] (: join predicate :)  
return <pair>{ $c,$z/text() } </pair>
```

#### Result

```
<pair>c0 zero</pair>,<pair>c0 zero’</pair>,
<pair>c0 zero</pair>,<pair>c0 zero’</pair>,
<pair>c2 two</pair>
```
Grouping (attempt #1)

A grouping query

```xml
let $a := <a><b><c>0</c><c>1</c><c>2</c><b></b></a>
let $x := <x><z id="2">two</z><z id="0">zero</z>
       <y><z id="0">zero'</z><z id="3">three</z><y></y>
for $c in $a/b/c
return <group>{
    $c, <mem>{
        for $z in $x//z[@id eq $c]
        return $z/text() }
    }</group>
```

- **Aggregate functions** (sum, count, ...) may be applied to group members, *i.e.*, element mem inside each group.
Grouping (attempt #1)

Result (NB: group of <c>0</c> appears twice)

```
<group><c>0</c><mem>zerozero'</mem></group>
<group><c>0</c><mem>zerozero'</mem></group>
<group><c>1</c><mem/></group>
<group><c>2</c><mem>two</mem></group>
```

← empty group!

Remarks:

- The preservation of the empty group for <c>1</c> resembles the effect of a relational **left outer join**.
- The duplicate elimination implicit in $a/b/c$ is based on node identity but we **group by value** (@id eq $c$).
  - ⇒ Such groupings call for value-based duplicate elimination.
Grouping (attempt #2)

**Improved grouping query**

```xml
let $a := \cdots unchanged \cdots$
let $x := \cdots unchanged \cdots$
for $c$ in distinct-values($a/b/c$)
return <group>
  <c>{$c}</c>,
  <mem>{$x//z[@id eq $c]/text() }</mem>
</group>
```

**Note:**
- Need to “rebuild” element c ($c$ bound to values).
- Inner for loop replaced by equivalent XPath expression.
XQuery: user-defined functions

It is typical for non-toy XQuery expressions to contain **user-defined functions** which encapsulate query details.

- User-defined functions may be collected into **modules** and then 'import’ed by a query.
- Function declarations may be directly embedded into the **query prolog** (prepended to query, separated by ’;’).

**Declaration of n-ary function f with body e**

```
declare function f($p_1$ as $t_1$, ..., $p_n$ as $t_n$) as $t_0$ { e }
```

- If $t_i$ is omitted, it defaults to `item()`*.
- The pair $(f, n)$ is required to be unique (overloading).
- Atomization is applied to the $i$-th parameter, if $t_i$ is atomic.
User-defined function examples

Form textual root-to-node paths

declare default function namespace
    "http://www-db.in.tum.de/XQuery/functions";

declare function path($n as node()) as xs:string
{ fn:string-join(for $a in $n/ancestor-or-self::*
    return fn:name($a), "/")
};

let $a := <a><b><c><d/></c><d/></b></a>
return $a//d/path(.)

⇒ ("a/b/c/d","a/b/d")

May not place user-def’d functions in the XQuery builtin function namespace (predefined prefix fn).
⇒ Use explicit prefix for user-def’d or builtin functions.
User-defined function examples

Reverse a sequence

Reversing a sequence does not inspect the sequence’s items in any way:

```xml
declare function reverse($seq)
{
    for $i at $p in $seq
        order by $p descending
        return $i
}
```

```xml
reverse((42,"a",<b/>,doc("foo.xml"))
```

Note:
- The calls $f()$ and $f(()$ invoke different functions.
Uder-defined functions: recursion

Trees are the prototypical **recursive** data structure in Computer Science and it is natural to describe computations over trees in a recursive fashion.\(^{39}\)

### Simulate XPath ancestor via parent axis

```xml
declare function ancestors($n as node()?) as node()*
{
  if (fn:empty($n)) then ()
  else (ancestors($n/..), $n/..)
}
```

#### Questions

1. Will the result be in document order and duplicate free?
2. What if we declare the parameter type as `node()*`?

---

\(^{39}\)This is a general and powerful principle in programming: *derive a function's implementation from the shape of the data it operates over.*
Answers

1. Yes.

2. Allowing $n$ as node() may violate this property: Consider ¡a¿¡b¿¡c¿¡d¿¡/¿¡/c¿¡/b¿¡e/¿¡/a¿ with context node sequence (d,e) as argument to ancestors(). We get:

- ancestors((d,e)) ⇒ (ancestors((a,c)), a, c)
- ancestors((a,c)) ⇒ (ancestors((b)), b)
- ancestors((b)) ⇒ (ancestors((a)), a)
- ancestors((a)) ⇒ (ancestors(()), ())
- ancestors(() ⇒ ()

Overall result: (a,b,a,c)

User-defined functions: recursion examples

<table>
<thead>
<tr>
<th>Purpose of function <code>hmm</code> and output of this query?</th>
</tr>
</thead>
<tbody>
<tr>
<td>declare function local:hmm($e as node()) as xs:integer</td>
</tr>
<tr>
<td>{ if (fn:empty($e/*)) then 1</td>
</tr>
<tr>
<td>else fn:max(for $c in $e/*/</td>
</tr>
<tr>
<td>return local:hmm($c)) + 1</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

```xml
local:hmm(<a><b/>
   <b><c><d/foo</d><e/></c></b>
</a>)
```

**Good style:**

- Use predefined namespace `local` for user-def’d functions.
- `hmm` has a more efficient equivalent (*cf.* a previous slide 411), exploiting the recursion “built into” axes descendant and ancestor.
User-defined functions: “rename” attribute

Renew attribute \$from to \$to

```
declare function local:xlate($n as node(),
                            $from as xs:string,
                            $to     as xs:string)

{ typeswitch ($n)
  case $e as element() return
     let $a := ($e/@*)[name(.) eq $from]
     return element
       { node-name($e) }
       { $e/(@* except $a),
          if ($a) then attribute { $to } { data($a) }
          else (),
          for $c in $e/node() return local:xlate($c, $from, $to) }
  default return $n
};
```
User-defined functions: “rename” attribute

Invoke `xlate`

```xml
local:xlate(<x id="0" foo="!"> foo 
  <y zoo="1">bar</y> 
</x>, "foo", "bar")

⇓

<x id="0" bar="!"> foo 
  <y zoo="1">bar</y> 
</x>
```

- **NB:** This constructs an entirely new tree.
- In XQuery 1.0, there is currently no way to modify the properties or content of a node.
- XQuery Update will fill in this gap (work in progress at W3C).

**N.B.: XSLT** (see above) has been designed to support **XML transformations** like the one exemplified here.
XQuery: the missing pieces

- This chapter did not cover XQuery exhaustively. As we go on, we might fill in missing pieces (e.g., typeswitch, validate).
- This course will not cover the following XQuery aspects:
  - (namespaces),
  - modules (declaration and import),
  - collations (string equality and comparison).

Reminder: [W3C XQuery specification](http://www.w3.org/TR/xquery/)
(Has become a W3C Recommendation in January 2007.)