Part X

XQuery—Querying XML Documents

Outline of this part

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 (W3C `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[DB], Ioana Manolescu[DB], Phil Wadler[PL], Mary Fernández[DB+PL], Don Chamberlin[DB,34], Jérôme Siméon[DB], Michael Rys[DB], and many others.

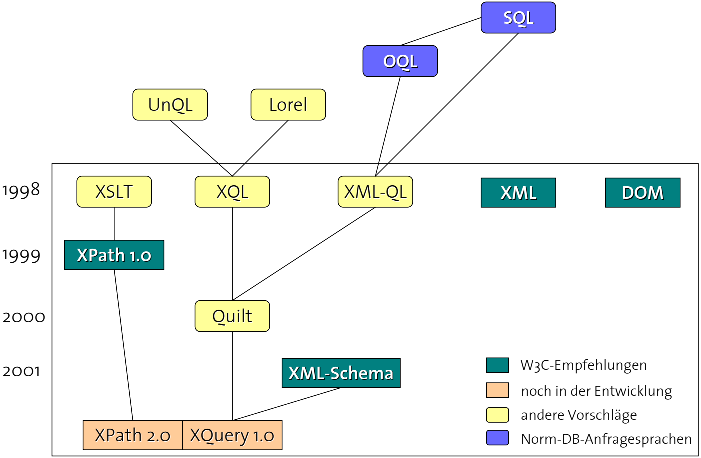
[34] Don is the "father" of SQL.

1/2 Programming Language, 1/2 Query Language

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



[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

$$\theta \in \{\texttt{=}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>=}, \texttt{>}\}$$

yields `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)   ⇒  true()
(1,2,3) = 1         ⇒  true()
() = 0              ⇒  false()
2 <= 1              ⇒  false()
(1,2,3) != 3        ⇒  true()
(1,2) != (1,2)      ⇒  true()
not((1,2) = (1,2))  ⇒  false()
```

Value comparisons

The six **value comparison operators** `eq`, `ne`, `lt`, `le`, `ge`, `gt` compare *single items by value* (atomization!):

```
2 gt 1.0                  ⇒  true()
<x>42</x> eq <y>42</y>    ⇒  true()
(0,1) eq 0                ⇒  ↯       (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)  ⇒  true()[35]
    ("2",1) = 1  ⇒  true() or ↯   (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

`<x>42</x> eq <x>42</x>`	⇒	`true()`
`<x>42</x> is <x>42</x>`	⇒	`false()`
`root(`e_1`) is root(`e_2`)`	⇒	nodes $e_{1,2}$ in same tree?
`let $a := <x><y/></x>` `return $a << $a/y`	⇒	`true()`

Working with sequences

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

Common sequence operations

Function	**Example**		
`count`	`count((0,4,2))`	$\Rightarrow$	`3`
`max`	`max((0,4,2))`	$\Rightarrow$	`4`
`subsequence`	`subsequence((1,3,5,7),2,3)`	$\Rightarrow$	`(3,5,7)`
`empty`	`empty((0,4,2))`	$\Rightarrow$	`false()`
`exists`	`exists((0,4,2))`	$\Rightarrow$	`true()`
`distinct-values`	`distinct-values((4,4,2,4))`	$\Rightarrow$	`(4,2)`
`to`	`(1 to 10)[. mod 2 eq 1]`	$\Rightarrow$	`(1,3,5,7,9)`

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

`<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.

FLWOR: Iteration via `for···in`

Explicit iteration

Explicit iteration is expressed using the `for···in` construct:[36]

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

If e_1 evaluates to the sequence $(x_1, \ldots, x_n)$, the loop body e_2 is evaluated n times with varable $\$v$ 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

```
for $x in (3,2,1)
return ($x,"*")          ⇒   (3,"*",2,"*",1,"*")
```

```
for $x in (3,2,1)
return $x,"*"            ⇒   (3,2,1,"*")
```

```
for $x in (3,2,1)                               (3,"a",3,"b",
return for $y in ("a","b")               ⇒        2,"a",2,"b",
       return ($x,$y)                             1,"a",1,"b")
```

FLWOR: Abbreviations

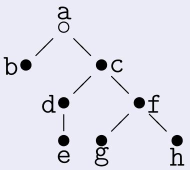
$$
\begin{array}{l}
\texttt{for } \$v_1 \texttt{ in } e_1 \\
\texttt{return} \\
\quad \texttt{for } \$v_2 \texttt{ in } e_2 \\
\quad \texttt{return } e_3
\end{array}
\equiv
\begin{array}{l}
\texttt{for } \$v_1 \texttt{ in } e_1 \\
\texttt{for } \$v_2 \texttt{ in } e_2 \\
\texttt{return } e_3
\end{array}
\equiv
\begin{array}{l}
\texttt{for } \$v_1 \texttt{ in } e_1\texttt{,} \\
\quad\;\; \$v_2 \texttt{ in } e_2 \\
\texttt{return } e_3
\end{array}
$$

FLWOR: Iteration via for · · · in

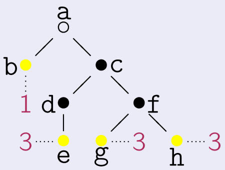
✎ Purpose of this query *Q*?

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

A sample *cs*



"Annotated" sample *cs*



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 `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`:

```
let $v := e1
return e2
```

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

- `for` and `let` clauses may be freely intermixed.

FLWOR: Variable Binding via let··· :=

Iteration *vs.* variable binding

```
for $x in (3,2,1)
return ($x,"*")
```
⇒ (3,"*",2,"*",1,"*")

```
let $x := (3,2,1)
return ($x,"*")
```
⇒ (3,2,1,"*")

"Every other item" revisited (flip back two slides)

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

```
let $seq := e
return for $i at $p in $seq
       return if ($p mod 2)
              then $seq[$p]
              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:

```
for $v in e1            for $v in e1
where p           ≡     return if (p)
return e2                      then e2
                               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 e1
order by e3 [ascending|descending] [empty greatest|least]
return e2
```

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                       ⇒  (5,3,1,4,2)
return $x
```

All bound variables in scope in order by

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

Reordering as in SQL's ORDER BY

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

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

FLWOR semantics: tuple space

- 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

```
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

$x	$p	$y	where	order by	return
10	1	100	false	4	(1,10)
9	2	81	false	3	(2,9)
8	3	64	false	2	(3,8)
7	4	49	false	1	(4,7)
6	5	36	true	0	(5,6)
5	6	25	true	-1	(6,5)
4	7	16	true	-2	(7,4)
3	8	9	true	-3	(8,3)
2	9	4	true	-4	(9,2)
1	10	1	true	-5	(10,1)

FLWOR semantics: tuple space (2)

2. Filtering: `where` clause (`$y <= 42`)

$x	$p	$y	where	order by	return
10	1	100	false	4	(1,10)
9	2	81	false	3	(2,9)
8	3	64	false	2	(3,8)
7	4	49	false	1	(4,7)
6	5	36	true	0	(5,6)
5	6	25	true	-1	(6,5)
4	7	16	true	-2	(7,4)
3	8	9	true	-3	(8,3)
2	9	4	true	-4	(9,2)
1	10	1	true	-5	(10,1)

FLWOR semantics: tuple space (3)

3. Reordering: `order by` clause

$x	$p	$y	where	order by	return
1	10	1	true	-5	(10,1)
2	9	4	true	-4	(9,2)
3	8	9	true	-3	(8,3)
4	7	16	true	-2	(7,4)
5	6	25	true	-1	(6,5)
6	5	36	true	0	(5,6)

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

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

1. Populate variable bindings only

\$x	\$p	\$y			
10	1	100			
9	2	81			
8	3	64			
7	4	49			
6	5	36			
5	6	25			
4	7	16			
3	8	9			
2	9	4			
1	10	1			

FLWOR: populate tuple space lazily (2)

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

$x	$p	$y	where		
10	1	100	false		
9	2	81	false		
8	3	64	false		
7	4	49	false		
6	5	36	true		
5	6	25	true		
4	7	16	true		
3	8	9	true		
2	9	4	true		
1	10	1	true		

3. Prune tuples

$x	$p	$y	where		
6	5	36	true		
5	6	25	true		
4	7	16	true		
3	8	9	true		
2	9	4	true		
1	10	1	true		

FLWOR: populate tuple space lazily (3)

4. Evaluate: `order by` clause

`$x`	`$p`	`$y`		`order by`	
6	5	36		0	
5	6	25		-1	
4	7	16		-2	
3	8	9		-3	
2	9	4		-4	
1	10	1		-5	

5. Normalize `order by` column, evaluate `return` clause

`$x`	`$p`	`$y`		`position()`	`return`
6	5	36		6	`(5,6)`
5	6	25		5	`(6,5)`
4	7	16		4	`(7,4)`
3	8	9		3	`(8,3)`
2	9	4		2	`(9,2)`
1	10	1		1	`(10,1)`

Variable bindings: Variables are not variable!

"Imperative" XQuery

Evaluate the expression

```
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

```
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 → 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"

`<x><![CDATA[foo & bar]]></x>` $\underset{\text{XQuery}}{\equiv}$ `<x>foo & bar</x>`

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

Computed element content

`<x>4{ max((1,2,0)) }</x>` $\Rightarrow$ `<x>42</x>`

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

Computed element constructors

Definition

In a **computed element constructor**

$$\texttt{element}\ \{e_1\}\ \{e_2\}$$

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

```
element { string-join(("foo","bar"),"-") } { 40+2 }
⇒   <foo-bar>42</foo-bar>
```

Computed element constructors

An application of computed element constructors: i18n

Consider a dictionary in XML format (bound to variable `$dict`) with entries like

```
<entry word="address">
  <variant lang="de">Adresse</variant>
  <variant lang="it">indirizzo</variant>
</entry>
```

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

```
element
  { $dict/entry[@word=name($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

```
<x a="{(4,2)}"/>    ⇒   <x a="4 2"/>
<x a="{{" b='}}'/>  ⇒   <x b="}" a="{"/>
<x a="'" b='"'/>    ⇒   <x a="'" b="&quot;"/>
```

- 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 $\{e\}$

```
if (empty(e))
then ()
else text { string-join(for $i in data(e)
                        return string($i),
                        "␣") }
```

Text node construction examples

```
text { (1,2,3) }  ≡  text { "1 2 3" }
```

```
let $n := <x>
            <y/><z/>
          </x>//name(.)
return <t>{ text {$n} }</t>
```

⇒ `<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:

Remember the (invisible) document root node!

xy.xml

```
<x>
  <y/>
</x>
```

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

The context node for the first expression above is the document node for document `xy.xml`.

- A document node may be constructed via `document` $\{e\}$.

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*`:
  0. **Consecutive literal characters** yield a single text node containing these characters.
  1. Expression enclosed in $\{\cdots\}$ are evaluated.
  2. **Adjacent atomic values** are cast to type `string` and collected in a single text node with intervening `"␣"`.
  3. A **node** is **copied** into the content **together with its content**. *All* copied nodes receive a **new identity.**
  4. Then, **adjacent text nodes** are merged by concatenating their content. Text nodes with content `""` are dropped.

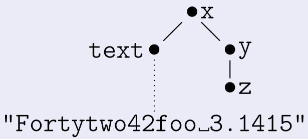
Example: processing element content

✎ Evaluate the expression below

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

Solution:

The constructed node is



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

```
let $id := "id"
return
  element x {
    attribute {$id}  {0},
    attribute {"id"} {1} }
```

⇒ ↯ (dynamic error)

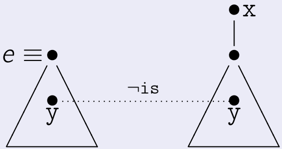
```
<x>foo{ attribute id {0} }</x>
```

⇒ ↯ (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




```
let $e := <a><b/><c><y>foo</y></c></a>
let $x := element x { $e }                                ⇒ false()
return exactly-one($e//y) is exactly-one($x//y)
```

- Node constructors have **side effects**.

Observing node identity

```
let $x := <x/>
return $x is $x ⇒ true()
```

```
let $d := doc(uri)
return $d is $d ⇒ true()
```

Construction establishes document order

✎ Result of the following query?

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

Solution

Construction: pair join partners

A join query

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

Grouping (attempt #1)

A grouping query

```
let $a := <a><b><c>0</c></b>
             <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
return <group>{
            $c, <mem>{ for $z in $x//z[@id eq $c]
                       return $z/text() }</mem>
       }</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>                ← empty group!
<group><c>2</c><mem>two</mem></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

```
let $a := ··· unchanged ···
let $x := ··· unchanged ···
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(.)
```

$\Rightarrow$ ("a/b/c/d","a/b/d")

May not place user-def'd functions in the XQuery builtin function namespace (predefined prefix `fn`).

$\Rightarrow$ 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:

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

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

```
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

User-defined functions: recursion examples

✎ Purpose of function hmm and output of this query?

```
declare function local:hmm($e as node()) as xs:integer
{ if (fn:empty($e/*)) then 1
  else fn:max(for $c in $e/*
              return local:hmm($c)) + 1
};

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

Rename 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

```
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.)