XML Technologies

XQuery: More than Just a Query Language!

Christian Grün, Leonard Wörteler
Database & Information Systems Group
Universität Konstanz
XQuery Versions

Version 2.0 ...was skipped (as XPath 2.0 was already out)

Version 3.0
- enhanced FLWOR expressions
- error handling: try/catch
- HOF (higher-order functions):
  - inline functions
  - dynamic function calls
  - named function references
  - partial function applications

Version 3.1
- (immutable) maps: map{...}
- (immutable) arrays: [...]
- lookup operator: ?
- arrow operator: =>
- string constructor: ` `[...]` 
- embedded JSON support (functions, serialization)
**FLWOR Expressions: Tuples**

**Semantics**

- FLWOR iterations can be regarded as *tuple streams*
- A tuple stream is an ordered sequence of zero or more *tuples*
- A tuple is a set of *variables*, each of which is bound to a *value*
- A value (as you know?) is a sequence of zero or more *items*

**Example**

```xml
<xml>
  <a>2</a>
  <b>4</b>
  <c>(2,4)</c>
</xml>
```

- `for $a$ in (2,3)
  for $b$ in (4,5)
  let $c := ($a,$b)$ (: ignored :)
  return $a * b$

- Tuple stream with 4 tuples:
  - $a: 2, b: 4, c: (2,4)$
  - $a: 2, b: 5, c: (2,5)$
  - $a: 3, b: 4, c: (3,4)$
  - $a: 3, b: 5, c: (3,5)$

- Result: 8, 10, 12, 15
FLWOR Expressions: Windows

Semantics

- windows are an essential feature in *streaming languages* (e.g. SPL)
- windows allow you to process *sub-sequences* of iterated tuples
- two types of windows exist:
  - *sliding* windows: may overlap (items of a stream may be contained in *more than one* window)
  - *tumbling* windows: each item will only be passed on once

Examples

.lazy for sliding window $w$ in (1 to 4)
  start at $s$ when true()
  end at $e$ when $e - s = 2$
  return string-join($w$, ' / ')
Result: 1/2/3, 2/3/4, 3/4, 4


.lazy for tumbling window $w$ in (1 to 4)
  start at $s$ when true()
  end at $e$ when $e - s = 2$
  return string-join($w$, ' / ')
Result: 1/2/3, 4
FLWOR Expressions: Windows

Syntax (simplified)

Window ::= for (Tumbling | Sliding)
Tumbling ::= tumbling window $Name in Expr Start End?
Sliding ::= sliding window $Name in Expr Start End
Start ::= start Vars when Expr
End ::= only? end Vars when Expr
Vars ::= ($Name)? (at $Name)? (previous $Name)? (next $Name)?

• if only end is specified, remaining items may be skipped
• the current, previous and next item may be bound to a variable

砭 for tumbling window $w in (1 to 10)
    start $c previous $p next $n when $c mod 2 = 0
    return string-join(($p, $c, $n), '/')

✏ What do you expect as the result?
FLWOR Expressions: Grouping

**Semantics**

- query results are partitioned into groups of tuples whose *grouping keys* are *equivalent*
- grouping keys are the atomized values of *grouping variables* (values must be single items)
- equivalent means: $\text{var}_1 \text{ eq } \text{var}_2$ must yield true
- an expression can be *bound* to the grouping variable

**Example I**

$$\text{for } n \text{ in (66, 99, 66)}$$
$$\text{group by } n$$
$$\text{return } n$$

Result: 66, 99

- grouping variable: $n$
- grouping keys: 66, 99, 66
- equivalent query:
  $$\text{distinct-values}((66, 99, 66))$$
FLWOR Expressions: Grouping

Example II

 우리나라 for $n$ at $p$ in (66,99,66) group by $n$
 return ($n$, $p$)
☞ The result is: 66, 1, 3, 99, 2

Example III

 우리나라 for $n$ in (66,99,66) let $x := 1$
group by $x := 1$
 return count($n$)
☞ What is the result?

Example IV

Based on the Factbook document*:

 우리나라 for $cntr$ in //country
 group by $size :=$xs:integer($cntr/@population$)
 order by $size$ descending
 return element countries { attribute size { $size$ }, $cntr/name
 }
☞ Describe the expected output!

* files.base.org/xml/factbook.xml
FLWOR Expressions: Counting, Empty Sequences

Counting

- the `count` clause enables you to enumerate FLWOR loops
- in contrast to `at` (XQuery 1.0), `count` can occur everywhere

Example

```xml
⊇ for $n at $p in (8,0,8)
  where $n = 8
  count $c
  return $c || '/' || $p
☞ Result: 1/1, 2/3
```

Allowing Empty

- derived from SQL
- if the input is empty, *loops once* and binds *empty sequence* to variable

Example

```xml
⊇ for $n in ()
  return count($n),
  for $n allowing empty in ()
  return count($n)
☞ What is the result?
```
Error Handling

Try/Catch Expression

- essential feature for dealing with *unexpected behavior*
- can be restricted to certain error codes

Generic Case

    try { expr1 }  
    catch * { expr2 }

Catching Specific Errors

    try { 1 + '' }  
    catch err:XPTY0004 {  
        'Typing!'  
    }

Catching Additional Information

    try { file:read-text('x.txt') }  
    catch * {  
        'code:' || $err:code ||  
        ',msg:' || $err:description  
    }
Higher-Order Functions

Functions in XQuery 1.0:

- have a name (are *named*)
- are *statically* declared
- values are sequences of:
  - nodes: `element()`, `text()`, ...
  - atomic values: `xs:string`, ...
- which function is called must be known *beforehand*

Functions in XQuery 3.x:

- can be unnamed (*anonymous*)
- can be declared *on-the-fly*
- values can be *functions*:
  - function that takes a function as argument: `fn:for-each`
  - functions that returns a function: `fn:function-lookup`
- function to be evaluated can be chosen *dynamically*:
  let $f := ... return $f()
Functions Items

In XQuery:

- an item can be a function item
- each function item has
  - a name, possibly absent (€)
  - an arity (number of arguments)
  - a function type, which specifies argument and result types
    e.g. function(xs:integer) as xs:boolean
- types of function items:
  inline functions, named function references, partially applied functions

In other functional languages:

- a very common concept
- also called
  - lambda (-abstraction), λ
  - inline function
  - anonymous function
- present in Java 8, Python, Haskell, Matlab, C#, ...

function(xs:integer) as xs:boolean
Functions Items: Syntax

**Inline function**

```xml
function($arg$ as type, ...) as type {
  ...
}
```

- creates an *unnamed* function item
- functions and variables from the *outer scope* can be used (closure)
- type declarations (*as ...*) are optional (as usual)

**Named function reference**

```xml
function-name#arity

- returns a function item for a *pre-declared* function
- example:
  ```xml
  let $substr := fn:substring#2
  return $substr('World', 2)
  ```
- equivalent to:
  ```xml
  function($string, $start) {
    fn:substring($string, $start)
  }('World', 2)
  ```
Functions Items: Syntax

Partial Function Application

function-name($x_1, \ldots, ?, \ldots, x_n)$

- *partially applies* arguments to a function (item)
- creates a new *function item* with a reduced number of arguments
- example:
  
  ```
  let $\text{chop-first} := \text{fn:substring}(?, 2)
  return $\text{chop-first('World')}$
  ```

  equivalent to:

  ```
  function($\text{string}$) {
    \text{fn:substring($\text{string}$, 2)}
  }('World')
  ```
Function Types

- check function type:
  \( f \) instance of function\((\ast)\),
  \( f \) instance of function\((\ldots) \) as ...
- retrieve name and number of arguments of a function:
  
  for \( f \) in (fn:count\#1, math:pi\#0)
  return (function-name\( (f)\),
         function-arity\( (f)\))
- function types can be arbitrarily nested:
  function(function() as \( \text{xs:integer} \))
  as function() as \( \text{xs:boolean} \)

⚠️ What is the type of \text{true}\#0?
HOF: Applications

Abstraction

- HOFs can be used to extract patterns into library functions: *iteration, recursion schemes*
- HOFs can be used to build data structures: *arrays, trees, ...*

Example: Newton’s Method

- Task: calculate the square root of a number
- Newton’s method:
  - start with an initial guess
  - while the guess is not good enough:
    - come up with a better guess by improving upon the old one
    - repeat
Newton’s Method

declare function local: square-root($n$ as $xs:double$, $eps$ as $xs:double$) {
  local: square-root($n$, 1, $eps$)
}

declare function local: square-root($n$, $guess$, $eps$) {
  let $new := n \text{ div } (guess)$
  let $diff := abs(n - guess \times guess)$
  return if($diff < $eps) then $guess$ else
    let $better-guess := (guess + new) \text{ div } 2$
    return local: square-root($n$, $better-guess$, $eps$)
}

(: call function with input number and newton epsilon :) 
local: square-root(1000, 1)
Newton’s Method II

Now the same for cube roots...

- seems easy:
  - adapt the part saying if the guess is good enough
  - adapt the code that calculates the new guess
- most things stay the same:
  - checking the current guess
  - calculating a new one
  - calling the function recursively
Newton’s Method II

declare function local:cube-root($n as xs:double, $eps as xs:double) {
    local:cube-root($n, 1, $eps)
};

declare function local:cube-root($n, $guess, $eps) {
    let $new := $n div ($guess * $guess)
    let $diff := abs($n - $guess * $guess * $guess)
    return if($diff lt $eps) then $guess else
        let $better(guess) := ($guess + $new) div 2
        return local:cube-root($n, $better(guess), $eps)
};

(: call function with input number and newton epsilon :) local:cube-root(1000, 1)
Newton’s Method III

Abstracting out the pattern

- problem:
  - lots of redundant code
- way out:
  - factor out Newton’s method
  - abstract over the parts of the code that change with function items
- advantages:
  - more concise, more general
  - reusable for similar problems
  - can be put in a library

```
declare function local:newtons-method(
  $guess
  as item()*,
  $good-enough
  as function(item())* as xs:boolean,
  $improve
  as function(item())* as item()*)
) as item()* {
  if($good-enough($guess))
    then $guess
  else let $better := $improve($guess)
          let $average := ($guess + $better) div 2
          return local:newtons-method(
            $average, $good-enough, $improve)
};
```
Newton’s Method III

declare function local:square-root($n as xs:double, $eps as xs:double) {
    local:newtons-method(
        1,
        function($guess) { abs($n - $guess * $guess) lt $eps },
        function($guess) { $n div $guess }
    )
};

declare function local:cube-root($n as xs:double, $eps as xs:double) {
    local:newtons-method(
        1,
        function($guess) { abs($n - $guess * $guess * $guess) lt $eps },
        function($guess) { $n div ($guess * $guess) }
    )
};
Newton’s Method IV

Abstracting again

• redundant code has been eliminated, but...
• there’s an even more general pattern in it:
  • start with an initial value
  • repeatedly update it until a condition holds
  • similar to a while loop

```xml
declare function local:repeat-until($condition as function(item()*) as xs:boolean,
  $current as item()*,
  $update as function(item()*) as item()*)
  as item()*
{
  if($condition($current))
    then $current
   else let $next := $update($current)
      return local:repeat-until($next, $condition, $update)
};
```
Newton’s Method IV

```
declare function local:newtons-method(
  $guess as item()*,
  $good-enough as function(item())* as xs:boolean,
  $improve as function(item())* as item()*)
) as item()*
{
  local:repeat-until(
    $good-enough,
    $guess,
    function($curr) {
      let $better-guess := $improve($guess)
      return ($guess + $better-guess) div 2
    }
  )
};
```
Pre-Defined Functions

Some common programming patterns have been standardized:

**Official functions**
- the XQuery 3.0 specification defines various higher-order functions\(^1\)
- use the \texttt{fn} prefix
- most work on \textit{sequences}
- will be presented next

**BaseX functions**
- separate module for additional higher-order functions\(^2\)
- use the \texttt{hof} prefix
- mostly inspired by the \textit{Haskell Prelude} module
- \textit{sorting, top-k queries, ...}

\(^1\) [http://www.w3.org/TR/xpath-functions-30](http://www.w3.org/TR/xpath-functions-30)
\(^2\) [http://docs.basex.org/wiki/Higher-Order_Functions_Module](http://docs.basex.org/wiki/Higher-Order_Functions_Module)
Pattern I: Examples

Increment integers in a sequence

```
declare function local:inc(  
    $ints as xs:integer*  
) as xs:integer* {  
    for $i in $ints return $i + 1  
};
```

Retriece names of elements

```
declare function fn:names(  
    $elements as element()*  
) as xs:string* {  
    for $e in $elements  
      return name($e)  
};
```

Lower-case sequence of strings

```
declare function local:lower-all(  
    $lines as xs:string*  
) as xs:string* {  
    for $l in $lines  
      return fn:lower-case($l)  
};
```

- Home work... Rewrite with fn:for-each
Pattern I

fn:for-each($seq, $f)

- applies a user-defined function to each item of a sequence
- restricted form of a FLWOR expression
- equivalent to the map operator: $seq ! $f(.)

XQuery implementation

```xquery
declare function local:for-each($seq as item(*), $f as function(item()) as item(*)) as item(*) {
  for $i in $seq
    return $f($i)
};
```

Example

```xquery
fn:for-each(
  (1.25, 19.84),
  floor#1
) ⨷ (1, 19)
```
Pattern II: Examples

Remove non-positive values
declare function local:positive($doubles as xs:double*) as xs:double* {
  for $d in $doubles where $d > 0
  return $d
};

Return strings starting with 'A'
declare function fn:strings-with-a($strings as xs:string*) as xs:string* {
  for $s in $strings where starts-with($s, 'A')
  return $s
};

✎ Home work... Rewrite with fn:filter
Pattern II

**fn:filter($seq, $pred)**

- returns all items of a sequence for which a *predicate function* returns true
- kind of superfluous, as XPath predicates are shorter: $seq[f(.)]

**XQuery implementation**

```xml
declare function local:filter(
    $seq as item()*,
    $f as function(item()) as xs:boolean
) as item()* {
    for $i in $seq
    where $f($i)
    return $i
};
```

**Example**

```xml
fn:filter(  
    1 to 10,  
    function($x) { $x * $x < 10 }  
) ⇢ (1, 2, 3)
```
Pattern III: Examples

Vector addition

```
declare function local:add-vectors(
    $vec1 as xs:double*,
    $vec2 as xs:double*
) as xs:double* {
    for $x at $p in $vec1
    let $y := $vec2[$p]
    return $x + $y
};
```

Enumerate lines in a file

```
declare function fn:enumerate-lines(
    $numbers as xs:integer*,
    $lines as xs:string*
) as xs:string* {
    for $line at $p in $lines
    let $nr := $numbers[$p]
    return concat($nr,' : ', $line)
};
```

✎ Home work... Rewrite with fn:for-each-pair
Pattern III

**fn:**for-each-pair($seq1, $seq2, $f)

- combines the elements of two sequences pairwise with a user-defined function:
  - stops when the shorter list ends
  - also known as zipping function

⚠️ What does this do?

every $bool in
   fn:for-each-pair( $seq, $seq[position()] > 1],
               function($x,$y) { $x < $y } ) satisfies $bool

**XQuery implementation**

declare function local:for-each-pair( $seq1 as item()*,
                                        $seq2 as item()*,
                                        $f as function(item(),item()) as item()
                                  ) as item()* {
            for $i in 
                1 to min(count($seq1),count($seq2))
              return $f($seq1[$i], $seq2[$i])
        };

**Example**

fn:for-each-pair( ('foo','bar'), (2,3), substring#2 ) ↦ ('oo', 'r')
Pattern IV: Examples

Sum up a numbers of a sequence

```xml
declare function local:sum(
   $nums as xs:double*
) as xs:double {
    let $head := head($nums)
    let $tail := tail($nums)
    return if(empty($tail)) then
        $head
    else
        $head + local:sum($tail)
};
```

Concatenate strings

```xml
declare function local:concat(
   $strs as xs:string*
) as xs:string {
    if(empty($strs)) then
        ''
    else
        $strs[1] || local:concat(
            $strs[position() > 1]
        )
    }
```

⚠️ Home work... Rewrite with fn:fold-(left|right)
Pattern IV

fold-right($seq, $start, $f)

- starts with an initial value
- consumes the input sequence right-to-left:
  - the value $start is used as the rightmost value
  - each new item is combined with the accumulator using $f

XQuery implementation

```xquery
declare function local:fold-right(
    $seq as item()*,
    $start as item()*,
    $f as function(item(),item())* as item()*
) as item()* {
    if(empty($seq)) then $start
    else $f(
        head($seq),
        local:fold-right(tail($seq),$start,$f))
};
```

Example

```xquery
fn:fold-right((1 to 5), 1,
    function($i, $res) { $res * $i }
) ...
```

\[ \rightarrow 120 \]
Pattern IV

fold-left($seq, $start, $f)

• starts with an initial value
• consumes the input sequence left-to-right:
  • the value $start is used as the leftmost value
  • each new item is combined with the accumulator using $f
• in comparison to fold-right, the accumulator comes before the new item

XQuery implementation

```
declare function local:fold-left(
  $seq as item()*,
  $start as item()*,
  $f as function(item()*,item()) as item()*)
  as item()* {
  if(empty($seq)) then $start
  else local:fold-left(
      tail($seq),
      $f($start, head($seq)),
      $f)
};
```

Example

```
fn:fold-left((0, 4, 0, 4), 0,
  function($i, $d) { 10 * $i + $d }) ...
```

\[
\Rightarrow 404
\]
Pattern IV

Evaluation order

\[
\text{fn:fold-right}((1,2,3), 42, f) \\
\rightarrow f(1,f(2,f(3,42)))
\]

Pseudo code

\[
\text{res} \leftarrow \text{start} \\
\text{foreach} (\text{x in reverse(seq)} ) \{ \\
\quad \text{res} \leftarrow f(x, \text{res}) \\
\} \\
\text{return res}
\]

Evaluation order

\[
\text{fn:fold-left}((1,2,3), 42, f) \\
\rightarrow f(f(f(42,1),2),3)
\]

Pseudo code

\[
\text{res} \leftarrow \text{start} \\
\text{foreach} (\text{x in seq} ) \{ \\
\quad \text{res} \leftarrow f(\text{res}, x) \\
\} \\
\text{return res}
\]
HOF: Data Structures

With XQuery 1.0

- modelling *structured data*, like trees, is hardly possible with plain sequences
  - remember: sequences are always flattened
- possibly realizable via tricks, e.g. \((1, 3, <sep/>, 9, 6)\)
- XML is great at representing trees, but contents can only be *textual*:
  - when storing, data needs to be *serialized*, and item types need to be remembered
  - when retrieving, data needs to be *parsed* and cast to the original type

```xml
<node id="1">
  <node id="3">
    <node id="5"/>
    <node id="9"/>
  </node>
  <node id="6">
    <node id="8"/>
    <empty/>
  </node>
</node>
```
HOF: Data Structures

With function items

As inline functions capture the variables from their environment, they can be used as simple data containers:

• values are not serialized
• structure can be expressed as control flow inside the function

Example I: pair of sequences

declare namespace pair = 'pair';
declare function pair:new($s1, $s2) {
  function($first as xs:boolean) {
    if($first) then $s1 else $s2
  }
};
declare function pair:first($pair) {
  $pair(true())
};
declare function pair:second($pair) {
  $pair(false())
};
(: retrieve first sequence of pair :) 
pair:first(pair:new(1 to 4, 5 to 9))

See here what’s possible:

http://github.com/LeoWoerteler/xq-modules
HOF: Data Structures

Example II: arrays

declare namespace array = 'array';
declare function array:new($seq) as function(*) {
    function($pos as xs:integer?) {
        if_exists($pos) then $seq[$pos]
        else count($seq)
    }
};
declare function array:length($array as function(*)) as xs:integer {
    $array()
};
declare function array:get($array as function(*), $pos as xs:integer) as item()? {
    $array($pos)
};
(: retrieve first sequence of pair :)
array:get(array:new(1 to 5), 3)
XQuery 3.1: Data Structures

Map

- available in nearly every language
- unordered collection of key/value pairs; keys are distinct

XQuery maps

- maps are function items!
- maps are immutable! 😄 Why?
  Implemented in BaseX as Hash Trie

имер sets can be simulated by storing dummy values

Examples

- empty map: \( \text{map} \{ \} \)
- map with initial entries:
  \( \text{map} \{ \text{'A':(1,2), 456:<a/>} \} \)
- access entry (via function argument):
  let \( \$\text{map} := \text{map} \{ \text{'A':1} \} \)
  return \( \$\text{map('A')} \)
- dynamically create map:
  \( \text{map:merge}((1 \text{ to } 5) ! \text{map} \{ \ldots \}) \)

What are the resulting entries?
XQuery 3.1: Data Structures

Array
- very basic data structure
- ordered collection of values, accessible by index
- arrays can be nested

XQuery arrays
- index starts with 1
- arrays are function items!
- arrays are immutable!
  Implemented in BaseX as Finger Tree

Examples
- empty array: []
- array with initial entries: [ 1, (2,3), (), [4] ]
- access entry (via function argument):
  let $array := [ 1, (2,3) ]
  return $array(2)
- dynamically create array:
  array { 1 to 5 },
  array:join((1 to 5) ! [.] )

What are the resulting entries?
XQuery 3.1: Syntactic Sugar

Lookup operator (?)

- access values of maps and arrays without parentheses

Example

```xquery
let $map := map { 'R':'red', 'G':'green', 'B':'blue' }
return ($map?R, (: "red" :) $map?* ( : all values :) )
```

Result: red, red, green, blue

Context-based lookup

- if no left operator is specified, a value will be looked up in the current context item

Example

```xquery
(map { 'name':'Guðrún', 'city':'Reykjavík' },
    map { 'name':'Hildur', 'city':'Akureyri' })[?name = 'Hildur']?city
```

Result: Akureyri
XQuery 3.1: Syntactic Sugar

Arrow operator (=>)

- applies a value to a function
- value is used as first argument to the function

String constructor (``[...]``)

- inspired by here document literals of the Unix shell/script languages
- generate strings that contain XQuery delimiters

Equivalent examples

- `string-join(tokenize('hello'), '-')`  => `tokenize()`  => `string-join('-')`
- `'hello'`  => `tokenize()`  => `string-join('-')`

Result: h-e-l-l-o

```
''live' & "direct"
```
```
``[\"live\' & \"direct\"]
```

Result: 'live' & "direct"