XQuery Notes

Jef Wijsen

October 22, 2017

1 XPath Expressions

XQuery supports XPath expressions. Every XQuery implementation must support the following axes: `self`, `child`, `descendant`, `descendant-or-self`, `attribute`, and `parent`. The following axes are optional: `ancestor`, `ancestor-or-self`, `following`, `following-sibling`, `preceding`, and `preceding-sibling`. XQuery does not support the namespace axis.

For example,

```
doc("catalog.xml")//car[color="blue"]/price
```

returns

```
<price unit="BEF">115000</price>
```

The function call `doc("catalog.xml")` opens `catalog.xml` and returns its root node. The file `catalog.xml` is the one used in the *XSLT Notes*:

```
<?xml version="1.0"?>
<!--The name of this file is catalog.xml-->
<catalog>
   <car>       <model>Renault CLIO</model>
               <color>blue</color>
               <price unit="BEF">115000</price></car>
   <bike>      <height>56</height>
               <price unit="EUR">500</price></bike>
   <go-kart>   <price unit="BEF">3000</price></go-kart>
   <car>       <model>Peugeot Partner</model>
               <color>red</color>
               <price unit="EUR">12000</price></car>
</catalog>
```

2 Constructing Sequences and Elements

A *sequence* is an ordered collection of zero or more items. An *item* is either an *atomic value* (a string, a number, a date, a boolean, an URI) or a single node. A *node* conforms to one of the seven node kinds (see *XSLT Notes*); a node may have other nodes as children. There is no distinction between a single item and a sequence of length one containing that item. A *typed value* is a sequence of zero or more atomic values. XQuery supports operators to construct sequences:

```
<seq-expr>
The expression (3,4,5,6,7)
   results in {(3,4,5,6,7)}
```

```
The range expression 3 to 7
     also generates {3 to 7}
There is no notion of nested sequences: (3 to 4,((5,6),7))
                          also generates {(3 to 4,((5,6),7))}
Sequences are not limited to atomic values:
                            (<i>{{(1,<two/>,3)}}</i>,4 to 7)
                 generates {(<i> {(1,<two/>,3)} </i>,4 to 7)}
</seq-expr>
```

The *direct element constructor* `<seq-expr>...</seq-expr>` creates a `seq-expr` element node. In a direct element constructor, single curly braces { and } delimit enclosed expressions, distinguishing them from literal text. Enclosed expressions are evaluated and replaced by their value, whereas material outside curly braces is simply treated as literal text. Doubled curly braces {{ and }} represent the characters { and } (see line 9). Here is the result of the above query:

```
<seq-expr>
The expression (3,4,5,6,7)
   results in 3 4 5 6 7
The range expression 3 to 7
     also generates 3 4 5 6 7
There is no notion of nested sequences: (3 to 4,((5,6),7))
                          also generates 3 4 5 6 7
Sequences are not limited to atomic values:
                            (<i>{(1,<two/>,3)}</i>,4 to 7)
                 generates <i>1<two/>3</i>4 5 6 7
</seq-expr>
```

In the following example, the *sequence constructor* `( , , )` creates a sequence of three element nodes. The curly braces enclosing `$bef` within the *direct element constructor* `<price unit="BEF">...</price>` are used to indicate that `$bef` has to be evaluated rather than treated as text.

```
let $bef:=1000000
return
(
<price unit="BEF"> { $bef } </price>,
<price unit="CHF"> { round($bef div 26.0019) } </price>,
element price { attribute unit {"EUR"}, round($bef div 40.3399) }
)
```

The result is:

```
<price unit="BEF">1000000</price>
<price unit="CHF">38459</price>
<price unit="EUR">24789</price>
```

Computed element constructors can compute element names:

```
let $bef:=1000000, $p:="prix"
return
<convertisseur>
{ element {$p} { attribute unit {"BEF"}, $bef } }
{ element {$p} { attribute unit {"CHF"}, round($bef div 26.0019) } }
{ element {$p} { attribute unit {"EUR"}, round($bef div 40.3399) } }
</convertisseur>
```

The result is:

```
<convertisseur>
   <prix unit="BEF">1000000</prix>
   <prix unit="CHF">38459</prix>
   <prix unit="EUR">24789</prix>
</convertisseur>
```

3 Comparison Operators

The *value comparison operators* `eq`, `ne`, `lt`, `le`, `gt`, `ge` are used to compare two single non-node values. They raise an error if either operand is a sequence of length greater than one.

The *general comparison operators* `=`, `!=`, `<`, `<=`, `>`, `>=` can deal with operands that are sequences, providing implicit "existential" semantics for both operands. That is, if θ denotes a general comparison operator, then $(a_1, \ldots, a_n)\ \theta\ (b_1, \ldots, b_m) \Leftrightarrow \exists a_i \exists b_j : a_i\ \theta\ b_j$. Thus, two sequences of atomic values are equal under "=" if they have an atomic value in common.

There are two *node comparison operators* in XQuery: `is` and `is not`. These operators compare the identity of two nodes. Two nodes may have a different identity even though their names and values are the same. The *order comparison operators* `<<` and `>>` test whether the first operand node precedes/follows the second one in document order.

If an operator or function requires an atomic value but finds something else instead, then the atomic value may be obtained by a process called *atomization*. For example, `<i>3</i>+<j>5</j>` evaluates to `8`; the elements `<i>3</i>` and `<i>5</i>` are automatically atomized into `3` and `5`. A sequence is atomized by atomizing each individual item of the sequence.

For example, the query

```
<comparison>
   (3)=3 is {(3)=3}.
   (3,4,5)=(5,6,7) is {(3,4,5)=(5,6,7)}.
   ()=() is {()=()}!
   (<i>3</i>)=(3) is {(<i>3</i>)=(3)} due to atomization.
</comparison>
```

yields the following result:

```
<comparison>
   (3)=3 is true.
   (3,4,5)=(5,6,7) is true.
   ()=() is false!
   (<i>3</i>)=(3) is true due to atomization.
</comparison>
```

4 Quantified and Conditional Expressions

The query

```
<quantified>
{every $x in 1 to 10 satisfies
       (some $y in 0 to 5, $z in 1 to 5 satisfies $x = $y + $z)}
</quantified>
```

yields the following result:

```
<quantified>
   true
</quantified>
```

The `then` branch in the following query will be executed if the XPath expression returns the empty sequence; otherwise the `else` branch is chosen.

```
<cond>{
   if   ( empty(doc("catalog.xml")//price[@unit="BEF"]) )
   then "all prices converted"
   else "still some old prices"
}</cond>
```

5 FLWOR Expressions

5.1 Syntax

The syntax of FLWOR expressions is given by:

(`ForClause` | `LetClause`)$^+$ `WhereClause`? `OrderByClause`? "`return`" `ExprSingle`

The symbol `ExprSingle` includes quantified expressions, conditional expressions, and FLWOR expressions.

```
for    $i in 1 to 3
for    $j in 1 to $i            ⇝
return <row>{$i}:{$j}</row>
```

```
<row>1:1</row>
<row>2:1</row>
<row>2:2</row>
<row>3:1</row>
<row>3:2</row>
<row>3:3</row>
```

Compare the preceding query with the following one.

```
for    $i in 1 to 9
let    $j := 1 to $i            ⇝
return <row>{$i}:{$j}</row>
```

```
<row>1:1</row>
<row>2:1 2</row>
<row>3:1 2 3</row>
<row>4:1 2 3 4</row>
<row>5:1 2 3 4 5</row>
<row>6:1 2 3 4 5 6</row>
<row>7:1 2 3 4 5 6 7</row>
<row>8:1 2 3 4 5 6 7 8</row>
<row>9:1 2 3 4 5 6 7 8 9</row>
```

5.2 Comparison with SQL

Figure 1 shows that the XQuery FLWOR expression seems to be inspired by SQL. The XQuery query of Figure 1 does not yield a well-formed XML document:

```
<name>An</name>
<company>ULB</company>
<name>Ed</name>
<company>UMH</company>
```

The following query returns a well-formed XML document:

```
<?xml version="1.0" standalone="no" ?>
<!--The name of this file is emp.xml-->
<employees>
  <emp>
     <name>Ed</name>
     <sal>100</sal>
     <company>UMH</company>
  </emp>
  <emp>
     <name>Tim</name>
     <sal>50</sal>
     <company>UCL</company>
  </emp>
  <emp>
     <name>An</name>
     <sal>75</sal>
     <company>ULB</company>
  </emp>
</employees>
```

EMP	Name	Sal	Company
	Ed	100	UMH
	Tim	50	UCL
	An	75	ULB

```
for $a in doc("emp.xml")/*/emp
where number($a/sal) gt 60
order by number($a/sal)
return ($a/name, $a/company)
```

```
FROM     EMP a
WHERE    a.Sal > 60
ORDER BY a.Sal
SELECT   a.Name, a.Company
```

Figure 1: Comparison between XQuery and SQL.

```
<answer>{
for $a in doc("emp.xml")/*/emp
where number($a/sal) gt 60
order by number($a/sal)
return <emp>{ $a/name,
              $a/company
       }</emp>
}</answer>
```

$\rightsquigarrow$

```
<answer>
  <emp>
    <name>An</name>
    <company>ULB</company>
  </emp>
  <emp>
    <name>Ed</name>
    <company>UMH</company>
  </emp>
 </answer>
```

For all products that are red or blue, get the Euro price and the color.

```
<blue-red-products>{
for    $product in doc("catalog.xml")/catalog/*
where  $product/color=("blue","red")
return element { name($product) }
               {    <price unit="EUR">{
                            if ( $product/price/@unit="EUR" )
                            then $product/price/text()
                            else round($product/price div 40.3399)
                    }</price>,
                    $product/color
               }
}</blue-red-products>
```

The function `name()` returns the name of an element node. The result of this query:

```
<blue-red-products>
   <car>   <price unit="EUR">2851</price>
           <color>blue</color></car>
   <car>   <price unit="EUR">12000</price>
           <color>red</color></car>
</blue-red-products>
```

5.3 Join Queries

The file `cy.xml` stores locations of companies.

```
<?xml version="1.0" standalone="no"?>
<!--The name of this file is cy.xml-->
<companies>
   <cy>   <cname>UMH</cname>
          <loc>Mons</loc>
          <loc>Charleroi</loc></cy>
   <cy>   <cname>UCL</cname>
          <loc>Louvain</loc></cy>
   <cy>   <cname>ULB</cname>
          <loc>Bruxelles</loc></cy>
</companies>
```

Get names of employees who work for a company located in Mons or Brussels; order employees by increasing salary.

```
for $emp in doc("emp.xml")//emp
for $cy in doc("cy.xml")//cy
where $emp/company = $cy/cname                    ⇝   <name>An</name>
and $cy/loc = ("Mons","Bruxelles")                    <name>Ed</name>
order by number($emp/sal)
return $emp/name
```

The first `for` clause binds the variable `$emp` to the *binding sequence* `doc("emp.xml")//emp`. The second `for` clause binds the variable `$cy` to the binding sequence `doc("cy.xml")//emp`. The two `for` clauses produce a stream of tuples; each tuple is a combination of values in the respective binding sequences. Thus, the *tuple stream* is the Cartesian product of the binding sequences.

See Section 3 for the comparison $cy/loc = ("Mons","Bruxelles"). Would the comparison `$emp/company = $cy/cname` need to be changed if an employee can work for more than one company?

Same query, using more XPath style:

```
for $emp in doc("emp.xml")//emp
for $cy in doc("cy.xml")//cy[cname=$emp/company]
where $cy/loc="Mons" or $cy/loc="Bruxelles"
order by number($emp/sal)
return $emp/name
```

5.4 Aggregate Queries

Who earns the highest salary?

```
let $emp := doc("emp.xml")//emp
for $a in $emp
where $a/sal = max($emp/sal)
return $a/name
```

⇝ `<name>Ed</name>`

6 Functions

What is the most expensive product?

```
declare namespace my="my.uri";
declare function my:toBEF ($p as element(price)) as element(price)
{
if ($p/@unit="BEF") then $p
else if ($p/@unit="EUR") then <price unit="BEF">{40.3399*$p}</price>
else error("Unknown Unit")
};

<most-expensive-product>{
   let $max := max(doc("catalog.xml")/catalog/*/my:toBEF(price))
   return doc("catalog.xml")/catalog/*[my:toBEF(price) = $max]
}</most-expensive-product>
```

The result of this query is:

```
<most-expensive-product>
  <car>
    <model>Peugeot Partner</model>
    <color>red</color>
    <price unit="EUR">12000</price>
  </car>
</most-expensive-product>
```

Note:

- The default namespace for function names is the namespace of the XQuery core function library, which is bound to the prefix `fn`. So `max` is shorthand for `fn:max`. The namespace for the function `toBEF` is bound to the prefix `my`.
- The *sequence type* `element(price)` refers to an element named `price`. The use of the term "sequence type" (rather than, e.g., "value type") comes from the fact that every XQuery value *is* actually a sequence: a single value is a sequence of length one.
- `max()` is defined for (sequences of) atomic values, not for elements. *Atomization* extracts atomic values from elements. The following query, e.g., yields `true`:

  ```
  3 eq sum((<a>1</a>,<b>2</b>))
  ```

7 Elaborated Example

The elaborated example of the *XSLT Notes* (Section 4), stated in XQuery.

```
declare namespace my="my.uri";
declare function my:toBEF ($p as element(price)) as element(price)
{
if ($p/@unit="BEF")
then $p
else if ($p/@unit="EUR")
     then <price unit="BEF">{round(40.3399*$p)}</price>
     else <price>?</price>
};

<HTML>
<BODY>
<H1>Cars</H1>
<TABLE BORDER="3">
<TR><TH>Car Model</TH><TH>Price</TH></TR>
{
for $car in doc("catalog.xml")//car
return
<TR><TD> { $car/model/text() } </TD>
    <TD> { ($car/my:toBEF(price))/text() } </TD>
</TR>
}
</TABLE>
<H1>Bikes</H1>
<TABLE BORDER="3">
<TR><TH>Frame Height</TH><TH>Price</TH></TR>
{
for $bike in doc("catalog.xml")//bike
return
<TR><TD> { $bike/height/text() } </TD>
    <TD> { ($bike/my:toBEF(price))/text() } </TD>
</TR>
}
</TABLE>
</BODY>
</HTML>
```

Here is the result:

```
<HTML>
   <BODY>
      <H1>Cars</H1>
      <TABLE BORDER="3">
         <TR>
            <TH>Car Model</TH>
            <TH>Price</TH>
         </TR>
         <TR>
            <TD>Renault CLIO</TD>
            <TD>115000</TD>
```

```
         </TR>
         <TR>
            <TD>Peugeot Partner</TD>
            <TD>484079</TD>
         </TR>
      </TABLE>
      <H1>Bikes</H1>
      <TABLE BORDER="3">
         <TR>
            <TH>Frame Height</TH>
            <TH>Price</TH>
         </TR>
         <TR>
            <TD>56</TD>
            <TD>20170</TD>
         </TR>
      </TABLE>
   </BODY>
</HTML>
```