XQuery

© Leonidas Fegaras
University of Texas at Arlington

- Influenced by ODMG OQL
- Based on XPath
- Purely functional language
  - may access elements from documents, may construct new values (elements), but cannot modify data
  - any expression is a query
  - query nesting is allowed at any place and on any level
- Strongly and statically typed
  - both type checking and type inference
- Has formal semantics based on the XML abstract data model
  - item: value or ordered tree
  - ordered sequence of items
  - literal: int, real, double, string

The Data Model

- An atomic value (literal)
  - eg. “a string”, 10, 3.5
  - has a type (based on the XML Schema atomic types)
    - eg. xsd:string, xsd:integer
  - to construct or coerce a value of a specific type:
    - xsd:dateTime("2008-03-25")
- A node
  - may be element, text, attribute, document, etc
  - has a unique identity
  - has a string value
  - may have children, parent, ancestors, descendants, siblings
  - follows a document order
- An item is either an atomic value or a node

Data Model (cont.)

- A sequence
  - is an ordered list of items (nodes or atomic values)
  - can contain heterogeneous values
    - eg. ("a", 1, <a>b</a>)
  - the empty sequence: ()
  - there is no such thing as a nested sequence
    - eg. ((1,"a","b"),"c") is equivalent to (1,"a","b","c")
  - a single item is also a singleton sequence
    - eg. (1) = 1
- XQuery Data Model:
  - every XQuery expression returns a sequence of items
Expressions

- Comma is sequence concatenation: eg. 1, 2, 3 which is equivalent to (1, 2, 3)
- Element construction: `<tag>` ... `<tag>`
  eg. `<person>`<name>John Smith</name><phone>1234</phone>`</person>
  may include attribute bindings in the start tag
  eg. `<person sn="123456">...`<person>
  the content between the start and end tags (as well as the attribute values)
  is in construction mode
  to switch to computation mode, must use `{`
  eq. `<a x="q" y="1+2">{ 2+3 }=4+1</a>` is equivalent to
  `a x="q" y="3">5=4+1</a>`
- Alternative construction:
  - element `{ tagname } { content }
  - attribute `{ attribute-name } { value } ` inside an element construction `where tagname and attribute-name are expressions that return strings`

Logic:

- Arithmetic operators: `+`, `-`, `*`, `div`, `mod`
- Cast values to double, if possible (otherwise, it is an error)
- `a ()` operand results to a `()`
- if the two operands are sequences of n/m elements, then the result is a sequence of n/m elements!
  `(1, 2, 3) + (80, 90) = (81, 91, 82, 92, 83, 93)`
- General comparisons: `<=` `>=` `<>` `=!`
  Existential semantics:
  - for any XQueries xs and ys:
    `xs <= ys` if for all numbers y in is there is at least one x in xs such that `x < y`
  eq. `(1, 2, 3) < (2, 4)`
- Value (strict) comparisons: eq, lt, gt, le, ge, ne
  - each operand must be a single atomic value or a node containing a single
    atomic value
  eg. `person/sex eq 12345`  `-- OK`
  eg. `items/item/price lt 100`  `-- Error!` (multiple prices)
- Checking document order for nodes: `<<` `>>`

XPath Expressions

- They often start from a root: `<document("URL")>`
  `document("book.xml")/chapter[10]/figure[caption="XML"]`
- XPath has been extended with ID dereference:  `<idrefname->`
  `document("movies.xml")/movie[title="Matrix"]/@cast->name`
- An XPath predicate acts as a filter:  `e[p]`
  for each element in the sequence e, if p is true, then propagate the element to the output, otherwise discard it
- Existential semantics of predicates
  - `[A/B < 10]` is true if at least one element returned by A/B is numeric and less than 10
    - note that `[A/B < 10]` is false if A/B returns the empty sequence
- The predicate may be a simple XPath
  - `[A/B]` is true if A/B returns a non-empty sequence

More Operations

- Set operators: union, intersect, except
- String functions: `concat("x","y")`
- Full-text search: contains
  eg. contains(/book/title,"XML")
- if condition then e1 else e2
  - a value is false if it is:
    - the xs:boolean value false
    - the number 0
    - the string ""
    - the empty sequence ()
- Boolean operators: and, or, not(...)
  (lazy evaluation)
- Aggregation: `count`, `sum`, `avg`, `min`, `max`
  eg. `avg(/book/title="XML")/price`
FLWOR Expressions

- Stands for: for-let-where-order-by-return
- They introduce variables: $x$
- Similar to select-from-where queries in OQL
  
  ```xml
  for $b$ in document("bib.xml")/book
  where $b$/author/name = "John Smith"
  and $b$/year > 2000
  return $b/title;
  ```

- Syntax: ([ ] means optional)
  - for $v$ [ at $i$ ] in $e$ [ where $e$ ] [ order by ... ] return $e$
  - let $v := e$ [ where $e$ ] [ order by ... ] return $e$
- Order-by clause
  - order by $e$ [ ascending | descending ] ...
- May include sequences of for/let bindings
  - let $x_1 := 1$ let $x_2 := 2$ return $x_1 + x_2$
- Existential/universal quantification
  - some $v$ in $e$ satisfies $e$
  - every $v$ in $e$ satisfies $e$

Semantics of FLWOR Expressions

- for $x$ [ at $i$ ] in $e$ [ where $pred$ ] return body
  - both $pred$ and body may depend on the value of $x$
  - if the expression $e$ returns the sequence of values ($v_1, v_2, ..., v_n$), then
    - variable $x$ is bound to $v_1$ first; if $pred$ is true, then evaluate the body
    - variable $x$ is bound to $v_2$ next; if $pred$ is true, then evaluate the body, etc
  - ... finally, variable $x$ is bound to $v_n$; if $pred$ is true, then evaluate the body
- all the resulting sequences from evaluating the body are concatenated
- eg, the query: for $a$ in $(1, 2, 3, 4)$ return $a + 10$
  return: $(11, 12, 13, 14)$
- the positional variable $i$ is the position of $x$ in $e$ (starting from 1)
- let $x := e$ return body
  - if the expression $e$ returns the sequence of values ($v_1, v_2, ..., v_n$), then $x$ is
    bound to the entire sequence
  - eg, the query: let $a := (1, 2, 3, 4)$ return $a$, $a$
    returns: $(1, 2, 3, 4, 1, 2, 3, 4)$
  - note that you can't update a value (you can't do let $i := i + 1$)

Example

```xml
<books>
  for $b$ in document("books.xml")/book
  where $b$/author/firstname = 'John'
  and $b$/author/lastname = 'Smith'
  return <book>
    $b/title,
    $b/price
  </book>
</books>
```

- May return:
  <book>
  <book/title>DOM and SAX</title><price>40</price></book>
</books>

What about this?

```xml
<books>
  for $b$ in document("books.xml")/book
  where $b$/author/firstname = 'John'
  and $b$/author/lastname = 'Smith'
  return <book>
    $b/title,
    $b/price
  </book>
</books>
```

- Will return:
  <book>
    <book/title>$b/title</book>
    <book/title>$b/price</book>
  </book>
</books>
Equivalent Query

```xml
<books>
  for $b in document('books.xml')/book
  [author/firstname = 'John' and author/lastname = 'Smith']
  return <book>
    $b/title,
    $b/price
  </book>
</books>
```

What about this?

```xml
<books>
  for $b in document('books.xml')/book
  where $b/author/[firstname = 'John' and lastname = 'Smith']
  return <book>
    $b/title,
    $b/price
  </book>
</books>
```

- It is actually more accurate for multiple authors:
  ```xml
  <book>
    <author><firstname>Mary</firstname></author>
    <author><firstname>Smith</firstname></author>
    <author><firstname>Travolta</firstname></author>
  </book>
  ```

Join

```xml
<bids>
  for $i in document('items.xml')/item
  let $b:=document('bids.xml')/bid[@item=$i/@id]
  order by $i/@id ascending
  return <bid item='{$i/@id}'>
    $i/name,
    <price>(max($b/price))</price>
  </bid>
</bids>
```

- May return:
  ```xml
  <bids>
    <bid item='3'><name>bicycle</name><price>100</price></bid>
    <bid item='5'><name>car</name><price>10000</price></bid>
  </bids>
  ```

Join 2

```xml
<bids>
  for $i in document('items.xml')/item
  for $b in document('bids.xml')/bid[@item=$i/@id]
  order by $i/@id ascending
  return <bid item='{$i/@id}'>
    $i/name,
    $b/price
  </bid>
</bids>
```
**Dependent Join**

```
<best_students>{
  for $d in document('depts.xml')/department[name='cse']
  for $s in $d/gradstudent
  where $s/gpa > 3.5
  return <student>{
    $s/name,
    $s/gpa,
    count($d//gradstudent)
  }<student>
}</best_students>
```

**Using 'let'**

```
<best_students>{
  let $d := document('depts.xml')/department[name='cse']
  for $s in $d/gradstudent
  where $s/gpa > 3.5
  return <student>{
    $s/name,
    $s/gpa,
    count($d//gradstudent)
  }<student>
}</best_students>
```

**What about this?**

```
<best_students>{
  let $d := document('depts.xml')/department[name='cse']
  let $s := $d//gradstudent[gpa > 3.5]
  return <student>{
    $s/name,
    $s/gpa,
  }<student>
}</best_students>
```

- It will return only one student:
  ```
  <best_students>
  <student>
  <name>John Smith</name><name>Mary Jones</name>...
  <gpa>3.6</gpa><gpa>4.0</gpa>
  </student>
  </best_students>
  ```

**Existential Quantification**

```
<result>{
  for $i in document('items.xml')/item
  where some $b in document('bids.xml')/bid[@item=$i/@id]
  satisfies $b/price > 1000
  return <bid>{$i}</bid>
}</result>
```

- which is equivalent to:
  ```
  <result>{
    for $i in document('items.xml')/item
    where document('bids.xml')/bid[@item=$i/@id][price > 1000]
    return <bid>{$i}</bid>
  }</result>
  ```
Universal Quantification

```xml
<result>{
  for $i$ in document('items.xml')/item
  where every $b$ in document('bids.xml')/bid[@item=$i/@id]
  satisfies $b$/price > 1000
  return <bid>{$i}$/bid
}<result>
```

which is equivalent to:

```xml
<result>{
  for $i$ in document('items.xml')/item
  where not(document('bids.xml')/bid[@item=$i/@id]
           [price <= 1000])
  return <bid>{$i}$/bid
}<result>
```

Nested XQueries

- Group book titles by author:
  ```xml
  <result>{
    for $a$ in distinct-values(document('bib.xml')/bib/book[author=$a]/title
      [publisher='Wesley']/author)
    return <author>{$a}
    book[author=$a]/title
  }
  </result>
  ```

- To group-by as in relational databases, distinct-values is typically needed to remove duplicate groups

More Nested XQueries

```xml
<prices>{
  for $a$ in document('www.amazon.com')/book
  return 
  <book>
    {$a/title, $a/price }
  </book>
}<prices>
```

Functions

```xml
declare function best ( $x$ as xsd:string )
  as xsd:integer {
    max(document('bids.xml')/bid[@item=$x]/price)
  }

declare function get_best ( $x$ as element() )
  as element() {
    for $i$ in document('item.xml')/item
    where $i/name = $x
    return <item>{$i, best($i/@id)}</item>
  }
```

get_best('bicycle')

- A function may be recursive
  - eg, compute the total cost of a part that contains subparts