XQuery

© Leonidas Fegaras
University of Texas at Arlington

The Data Model

- Literal
  - string, integer, decimal, double, date, etc
  - e.g., "a string", 10, 3.5, 10.5e3
- A sequence
  - is an ordered list of items (nodes or atomic values)
  - can contain heterogeneous values
    - e.g., <"a","b","c"/>
  - empty sequence: ()
  - there is no such thing as a nested sequence
    - e.g., (<1,("a","b","c")> is equivalent to (1,"a","b","c")
  - a value is also a singleton sequence
- A node
  - may be element, text, attribute, documents, etc
  - has identity
  - follows a document order

Expressions

- Comma is sequence concatenation:
  - e.g., 1,2,3 is equivalent to (1,2,3)
- Element construction: <tag> ... </tag>
  - e.g., <person><name>John Smith</name><phone>s1234</phone></person>
  - may include attribute bindings in the start tag
    - e.g., <person src="123456">...</person>
  - the content between the start and end tags (as well as the attribute values)
    - in construction mode
      - to switch to computation mode, must use {}
    - e.g., <a x="q" y="n"[1+2]">[2+3]+4+1</a> is equivalent to
    - <a x="q" y="3">5+4+1</a>
- Alternative syntax for element construction:
  - element { tagname } { content }
  - attribute { attribute-name } { value }
  - inside an element construction
    - where tagname and attribute-name are expressions that return strings

• Influenced by SQL
• 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
  - XQuery types are based on XML Schema
• Has formal semantics based on the XML abstract data model
  - items: value or ordered tree
  - ordered sequence of items
  - literals: int, real, double, string
XPath Expressions

- Starts from a root: `doc("URL")`
- `doc("bib.xml")/book[author/lastname="Smith"]/title`
- `doc("book.xml")/chapter[10]/figure/caption="XML"]`
- 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 result, 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

Expressions

- Arithmetic operators: `+ - * div mod`
  - If an operand is `0`, return `0`
  - If an operand is a non-numeric value, coerce it to double
    - eg. string "3.5" is coerced to the double 3.5
  - If an operand is an element construction with text content, coerce the text to double
    - eg. `<a>3</a>` is coerced to the double 3.0
  - If the operand types are different, coerce them to the most general
    - eg. in `3+5.6` the integer 3 is coerced to the double 3.0
  - Otherwise, it is an error
    - `(1,2) + 3` error!
- Examples:
  - `<a>3</a> + <b>2` = 5.0

Atomization

- How do we extract a value from a node?
  - //gradstudent[gpa > 3.5]/name
- arithmetic expressions/comparisons extract the value of the node before the apply the operation using `fn:data`
  - eg: `fn:data(<a>2</a>) = "2"
- strings are cast to numerical values
  - eg: "3" < 4 is true
- Effective boolean value:
  - `()` `true()` are all false
  - `false()` is true
  - any nonzero number or nonempty string is true
  - any nonempty sequence where the first item is a node is true
  - otherwise, error

Other Expressions

- General comparisons: `= < > <= >= !=`
  - Similar atomization as arithmetic operations
  - ... but operands may be sequences
  - Existential semantics
    - eg. `(1,1,2) < (2,3)` is true because `2 < 3`
    - eg. `(1,1,2) = (2,3)` is true because `2 = 2`
- Value comparisons: `eq lt gt le ge ne`
  - Exactly the same atomization as arithmetic operations
    - Operands must be atomic values
  - Order comparisons: `is << >>`
    - For element constructions only
    - Based on document order
- Boolean operators: and, or, `fn:not(...)`
  - They retrieve the effective boolean value of operands
  - Return `fn:true()` or `fn:false()`
### Other Expressions

- Set operators: union, intersect, except
  - For element sequences only
  - Based on document order equality
- Full-text search: contains
  - contains(/book/title, "XML")
- if-then-else
  - if (Sa/gpa > 3.5) then Sa/name else ()
- Aggregation: count, sum, avg, min, max
  - avg(/book/title="XML"]/price)

### FLWOR Expressions

- Similar to select-from-where queries in OQL
  - for $b in doc("bib.xml")/book
    - where $b/author/firstname = "John Smith"
    - and $b/year > 2000
    - return $b/title
  - Syntax: ([ ] means optional)
    - for $v 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 let $y:=2 return $x+$y
  - Existential/universal quantification
    - some $v in e satisfies e
    - every $v in e satisfies e

### Semantics of FLWOR Expressions

- for $x 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 (v1,v2,...,vn), then
    - variable $x is bound to v1 first; if pred is true, then evaluate the body
    - variable $x is bound to v2 next; if pred is true, then evaluate the body, etc
    - ...; finally, variable $x is bound to vn; 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
    - returns: (11,12,13,14)
- for $x at $i in e [ where pred ] return body
  - $i is bound to the index starting from 1
  - eg. for $a at $i in (5,6,7) return ($a,$i)
  - returns: (5,1,6,2,7,3)
- let $x:=e return body
  - if the expression $e returns the sequence of values (v1,v2,...,vn), 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)
What about this?

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

• Will return:
```xml
<book>$b/title,$b/price</book>
<book>$b/title,$b/price</book>
</books>
```

What about this?

```xml
<books>
  for $b in doc('books.xml')/book
  where $b/author/first_name = 'John'
  and $b/author/last_name = 'Smith'
  return <book>
    $b/title,
    $b/price
  </book>
</books>
```

• It is actually more accurate for multiple authors:
```xml
<book>
  <author>John/first_name</author>
  <last_name>Smith</last_name>
  <author>Mary/first_name</author>
  <last_name>Minty</last_name>
  <author>Travolta/first_name</author>
  <last_name>Travolta</last_name>
</book>
```

Equivalent Query

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

Join

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

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

Dependent Join

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

Using ‘let’

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

What about this?

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

- It will return only one student:

```xml
<best_students>
  <student>
    <name>John Smith</name><name>Mary Jones</name>...
    <gpa>3.6</gpa><gpa>4.0</gpa>
  </student>
</best_students>
```
Existential Quantification

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

- which is equivalent to:

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

Universal Quantification

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

- which is equivalent to:

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

Nested XQueries

- Group book titles by author:

```xml
$result|
  for $a in distinct-values(doc('bib.xml')/bib
  /book[author='Wesley']/author)
  return <author>{$a}
    doc('bib.xml')/bib/book[author>{$a}/title
  </author>
</result>
```

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

More Nested XQueries

```xml
$prices|
  for $a in doc('www.amazon.com')/book
  return <book>
    {$a/title, $a/price }
  for $b in doc('www.bn.com')/book
    where $b/@isbn=$a/@isbn
    and $b/price < $a/price
    return $b/price }
</book>
</prices>
```
Functions

```plaintext
declare function best ($x$) {
    max(doc('bids.xml')//bid[@item=$x']/price)
};
declare function get_best ($x$) {
    for $i$ in doc('item.xml')//item
    where $i$/name = $x$
    return <item>$i$, best($i$/@id)$</item>
};
get_best('bicycle')
```

- May have a type signature
  ```plaintext
declare function best ($x$ as xs:string) as xs:element [...]```
- A function may be recursive
  ```plaintext
eg, compute the total cost of a part that contains subparts```
- There is also a `declare variable $x := e;'` syntax

Copying XML Data

```plaintext
declare function local:copy($element) {
    element (name($element))
    $element/@*,
    for $child in $element/node()
    return
    if ($child/name())
    then local:copy($child)
    else $child
}
```

Type Checking and Casting

- Check if the expression result has a given type
  ```plaintext
  $a/gpa/data() instanceof xs:int```
- Can the expression be casted to the given type?
  ```plaintext
  $a/gpa/data() castable as xs:int```
- Cast the expression to the given type
  ```plaintext
  $a/gpa/data() cast as xs:int```
- Typeswitch: a case statement to associate branches to types, based on the type of an expression

Updates

- Updates are XQuery expressions that return ()
- There are some restrictions about where an update can occur
- Examples:

  ```plaintext
  replace $d//graduate[name/lastname="Smith"]/gpa with 3.7
  insert <zip>12345</zip>
  into $d//graduate[name/lastname="Smith"]/address
  for $e in $d//employee
  return replace $e/salary with $e/salary*1.5```

Web Databases and XML  L7: XQuery  25
Web Databases and XML  L7: XQuery  26
Web Databases and XML  L7: XQuery  27
Web Databases and XML  L7: XQuery  28