More on XQuery (module #7)
Examples of path expressions

- `document("bibliography.xml")/child::bib`
- `$x/child::bib/child::book/attribute::year`
- `$x/parent::*`
- `$x/child::*/descendent::comment()`
- `$x/child::element(*, ns:PoType)`
- `$x/attribute::attribute(*, xs:integer)`
- `$x/ancestors::document(schema-element(ns:PO))`
- `$x/(child::element(*, xs:date) | attribute::attribute(*, xs:date))`
- `$x/f(.)`
Xpath abbreviated syntax

- **Axis can be missing**
  - By default the child axis
    $x/child::person -> $x/person

- **Short-hands for common axes**
  - Descendent-or-self
    $x/descendant-or-self::*//child::comment() -> $x//comment()
  - Parent
    $x/parent::* -> $x/..
  - Attribute
    $x/attribute::year -> $x/@year
  - Self
    $x/self::* -> $x/.
Xpath filter predicates

• Syntax:

   \textit{expression1 [ expression2 ]}

• [ ] is an overloaded operator

• Filtering by position (if numeric value):

   \textit{/book[3]}
   \textit{/book[3]/author[1]}
   \textit{/book[3]/author[1 to 2]}

• Filtering by predicate:

   – \textit{//book [author/firstname = “ronald”]}
   – \textit{//book [@price <25]}
   – \textit{//book [count(author [@gender=“female”] )>0}

• Classical Xpath mistake

   • $x/a/b[1]$ means $x/a/(b[1])$ and not ($x/a/b)[1]$
Conditional expressions

if ( $book/@year < 1980 )
then "oldTitle"
else "newTitle"

- Only one branch allowed to raise execution errors
- Impacts scheduling and parallelization
- Else branch mandatory
Local variable declaration

• Syntax :

```plaintext
let variable := expression1
return expression2
```

• Example :

```plaintext
let $x := document("bib.xml")/bib/book
return count($x)
```

• Semantics :

– bind the `variable` to the result of the `expression1`
– add this binding to the current environment
– evaluate and return `expression2`
Simple iteration expression

• Syntax :

  for variable in expression1
  return expression2

• Example

  for $x$ in document("bib.xml")/bib/book
  return $x/title

• Semantics:
  – bind the variable to each root node of the forest returned by expression1
  – for each such binding evaluate expression2
  – concatenate the resulting sequences
  – nested sequences are automatically flattened
FLW(O)R expressions

- Syntactic sugar that combines FOR, LET, IF

```
FOR var IN expr
  LET var := expr
  WHERE expr
  RETURN expr
```

- Example
  ```
  for $x in //bib/book /* similar to FROM in SQL */
  let $y := $x/author /* no analogy in SQL */
  where $x/title="The politics of experience" /* similar to WHERE in SQL */
  return count($y) /* similar to SELECT in SQL */
  ```
FLWR expression semantics

- FLWR expression:
  
  \[
  \text{for } x \text{ in } //\text{bib/book} \\
  \text{let } y := x/\text{author} \\
  \text{where } x/\text{title} = \text{"Ulysses"} \\
  \text{return } \text{count}(y)
  \]

- Equivalent to:
  
  \[
  \text{for } x \text{ in } //\text{bib/book} \\
  \text{return } \left( \text{let } y := x/\text{author} \\
  \text{return } \\
  \quad \text{if } (x/\text{title} = \text{"Ulysses"} ) \\
  \quad \text{then } \text{count}(y) \\
  \quad \text{else } () \\
  \right)
  \]
More FLWR expression examples

• Selections

  for $b$ in document("bib.xml")//book
  where $b$/publisher = "Springer Verlag" and
      $b$/@year = "1998"
  return $b/title

• Joins

  for $b$ in document("bib.xml")//book,
      $p$ in //publisher
  where $b$/publisher = $p$/name
  return ( $b/title , $p/address)
The “O” in FLW(O)R expressions

- Syntactic sugar that combines FOR, LET, IF

```
FOR var IN expr
LET var := expr
WHERE expr
RETURN expr
```

- Syntax

```plaintext
for $x$ in //bib/book /* similar to FROM in SQL */
let $y := $x/author /* no analogy in SQL */
[stable] order by ( [expr] [empty-handling ? Asc-vs-desc? Collation?] )+ /* similar to ORDER-BY in SQL */
return count($y) /* similar to SELECT in SQL */
```
Node constructors

• Constructing new nodes:
  – elements
  – attributes
  – documents
  – processing instructions
  – comments
  – text

• Side-effect operation
  – Affects *optimization* and *expression rewriting*

• Element constructors create local scopes for namespaces
  – Affects *optimization* and *expression rewriting*
Element constructors

• A special kind of expression that creates (and outputs) new elements
  – Equivalent of a new Object() in Java

• Syntax that mimics exactly the XML syntax
  – `<a b="24">foo bar</a>`

is a normal XQuery expression.

• Fixed content vs. computed content
  – `<a>{some-expression}</a>`
  – `<a> some fixed content {some-expression} some more fixed content</a>`
Computed element constructors

- If even the name of the element is unknown at query time, use the other syntax
  - Non XML, but more general

\[
\text{element \{name-expression\} \{content-expression\}}
\]

`let $x := <a b="1">3</a>`
`return element \{fn:node-name($e)\} \{$e/@*, 2 * fn:data($e)\}`

`<a b="1">6</a>`
Copy during node construction

let $x := <foo><a/></foo>
let $y :=<b> {$x/a}</b>
return ( $x/a is $y/a ) true or false ? False !

• The <a/> inside the <b> is a copy of the original <a/>
• XML is a tree not a graph
• What does it mean to “copy” a tree ?
• What about NS ? What about the types ?
• Ways to control the semantics of copy -- per query
Other node constructors

• Attribute constructors: direct (embedded inside the element tags) and computed
  - `<article date="{fn:getCurrentDate()}"/>
  - attribute “date” {fn:getCurrentDate()}

• Document constructor
  - document {expression}

• Text constructors
  - text {expression}

• Other constructors (comments, PI), but no NS
A more complex example

<?xml version="1.0" encoding="UTF-8"?>

<livres>
  {for $x in fn:doc("input.xml")//book
     where $x/year > 2000 and some $y in $x/author satisfies
     $y/address/country="France"
     return
       <livre annee="{$x/year}">
         <titre>{$x/title/text()}</titre>
         { for $z in $x/( author | editor )
           return
             if(fn:name($z)="editor)
               then <editeur>{$z/*}</editeur>
             else <auteur>{$z/*}</auteur>
           }
       </livre>
  }
</livres>
Sample data

- **parts.xml**: contains many part elements; each part element in turn contains partno and description subelements.
- **suppliers.xml**: contains many supplier elements; each supplier element in turn contains suppno and suppname subelements.
- **catalog.xml**: contains information about the relationships between suppliers and parts. The catalog document contains many item elements, each of which in turn contains partno, suppno, and price subelements.
Joins

<descriptive-catalog>
{for $i in fn:doc("catalog.xml")/items/item,
   $p in fn:doc("parts.xml")/parts/part
   [partno = $i/partno],
   $s in fn:doc("suppliers.xml")/suppliers
   /supplier[suppno = $i/suppno]
   order by $p/description, $s/suppname
return
   <item>{$p/description,
   $s/suppname,$i/price}</item>}
</descriptive-catalog>
for $s in fn:doc("suppliers.xml")/suppliers/supplier
order by $s/suppname
return

<supplier>
  {
    $s/suppname,
    for $i in fn:doc("catalog.xml")/items/item
      [suppno = $s/suppno],
    $p in fn:doc("parts.xml")/parts/part
      [partno = $i/pno]
      order by $p/description
    return $p/description
  }
</supplier>
No explicit Group-By

for $pn in fn:distinct-values(
        fn:doc("catalog.xml")/items/item/partno)
let $i := fn:doc("catalog.xml")/items/item[partno = $pn]
where fn:count($i) >= 3
order by $pn
return
<well-supplied-item>
    <partno> {$p} </partno>
    <avgprice> {fn:avg($i/price)} </avgprice>
</well-supplied-item>
Quantified expressions

• Universal and existential quantifiers

• Second order expressions
  - some variable in expression satisfies expression
  - every variable in expression satisfies expression

• Examples:
  - some $x$ in //book satisfies $x$/price <100
  - every $y$ in //((author | editor) satisfies $y$/address/city = “New York”
Counting while iterating

for $x \texttt{ at } ¥\texttt{i} \texttt{ in } //a/b$
where $x/c = 30$ and $i \text{ mod } 2 \text{ eq } 0$
return <result no="{$i \text{ div } 2}”>{$x} </result>

• Binds $i$ to (1,2,...) while iterating
• $i$ -- count variable
• Once defined, a count variable can be used like any other variables
• Always of type integer
• Allows to number the *input* of the iteration; no way to number the *output*!
Nested scopes

declare namespace ns="uri1"

for $x in fn:doc("uri")/ns:a
where $x/ns:b eq 3
return

  <result xmlns:ns="uri2">
    { for $x in fn:doc("uri")/ns:a
      return $x/ns:b }
  </result>

Local scopes impact optimization and rewriting!
Operators on datatypes

**expression instanceof sequenceType**
- returns true if its first operand is an instance of the type named in its second operand

**expression castable as singleType**
- returns true if first operand can be casted as the given sequence type

**expression cast as singleType**
- used to convert a value from one datatype to another

**expression treat as sequenceType**
- treats an expr as if its datatype is a subtype of its static type (down cast)

**typeswitch**
- case-like branching based on the type of an input expression
Typeswitch

typeswitch($customer/billing-address)
  case $a as element(*, USAddress) return $a/state
  case $a as element(*, CanadaAddress) return $a/province
  case $a as element(*, JapanAddress) return $a/prefecture
  default return "unknown"

• Like a “normal” switch, but based on the type, not on the value
• Allows dynamic dispatch based the runtime structure of the data
• No “normal” (value based) switch in XQuery; use a cascade of conditionals
• Each case described by a SequenceType -- the XQuery syntax for an XML Type (see the XML type system later)
  – Element(), element(ns:foo), element(ns:foo, xs:string), element(*, xs:string)
  – Same for attributes
  – Simple types: xs:integer, xs:date
  – General types: node(), item(), xs:AnyType
  – +, ?, *, empty-sequence()
• The same syntax is used in all places for types (e.g. function signatures, variable type declarations, type operators)
• The same syntax and semantics for describing the filtering criteria in a path expression !
Schema validation

- Explicit syntax
  validate [validation mode] { expression }
- Validation mode: strict or lax
- Semantics:
  - Translate XML Data Model to Infoset
  - Apply XML Schema validation
  - Ignore identity constraints checks
  - Map resulting PSVI to a new XML Data Model instance
- It is not a side-effect operation
Ignoring order

- In the original application XML was totally ordered
  - Xpath 1.0 preserves the document order through implicit expensive sorting operations
- In many cases the order is not semantically meaningful
  - The evaluation can be optimized if the order is not required
- **Ordered** \{ expr \} and **unordered** \{ expr \}
- Affect: path expressions, FLWR without order clause, union, intersect, except
- Leads to non-determinism
- Semantics of expressions is again context sensitive

```xml
let $x$:= (//a)[1] unordered {(//a)[1]/b}
return unordered {$x/b}
```
Functions in XQuery

• In-place XQuery functions

```xquery
declare function ns:foo($x as xs:integer) as element()
{
  <a> {$x+1} </a>
}
```
– Can be recursive and mutually recursive

• External functions

XQuery functions as database views
How to pass “input” data to a query?

• External variables (bound through an external API)
  
  `declare variable $x as xs:integer external`

• Current item (bound through an external API)

• External functions (bound through an external API)
  
  `declare function ora:sql($x as xs:string) as node()* external`

• Specific built-in functions
  
  `fn:doc(uri), fn:collection(uri)`
XQuery prolog

Version Declaration
Module Declaration (: what module is the query in :)

Base URI Declaration
Namespace Declaration
Default Namespace Declaration (: URI sand namespaces handling :)

Schema Import
Module Import
Variable Declaration
Function Declaration (: imports, local declarations :)

Boundary-space Declaration (: controlling the element construction :)
Construction Declaration
Copy-Namespace Declaration

Empty Order Declaration (: controlling the order by :)
Default Collation Declaration

Ordering Mode Declaration (: controlling the order :)

02/05/07
Library modules (example)

**Library module**

module namespace mod="moduleURI";
declare namespace ns="URI1";
define variable $mod:zero as xs:integer
    {0}
define function mod:add($x as xs:integer, $y as xs:integer)
    as xs:integer

{ }
    $x+$y
}

**Importing module**

import module namespace
ns="moduleURI";
ns:add(2, ns:zero)
XQuery type system

- XQuery has a powerful (and complex!) type system
- XQuery types are imported from XML Schemas
- Every XML data model instance has a dynamic type
- Every XQuery expression has a static type
- Pessimistic static type inference
- The goal of the type system is:
  - detect statically errors in the queries
  - infer the type of the result of valid queries
  - ensure statically that the result of a given query is of a given (expected) type if the input dataset is guaranteed to be of a given type
XQuery type system components

• Atomic types
  – xdt:untypedAtomic
  – All 19 primitive XML Schema types
  – All user defined atomic types
• Empty, None
• Type constructors (simplification!)
  – Elements: element name {type}
  – Attributes: attribute name {type}
  – Alternation: type1 | type2
  – Sequence: type1, type2
  – Repetition: type*
  – Interleaved product: type1 & type2

• type₁ intersect type₂ ?
• type₁ subtype of type₂ ?
• type₁ equals type₂ ?
XQuery implementations

• Relational databases
  – Oracle 10g, SQLServer 2005, DB2 Viper
• Middleware
  – Oracle, DataDirect, BEA WebLogic
• DataIntegration
  – BEA AquaLogic
• Commercial XML database
  – MarkLogic
• Open source XML databases
  – BerkeleyDB, eXist, Sedna
• Open source Xquery processor (no persistent store)
  – Saxon, Zorba, Galax
• XQuery editors, debuggers
  – StylusStudio, oXygen
• Help lists  talk@x-query.com