Module 6

XQuery
XML queries

- An XQuery basic structure:
  - a prolog + an expression

- Role of the prolog:
  - Populate the context where the expression is compiled and evaluated

- Prologue contains:
  - namespace definitions
  - schema imports
  - default element and function namespace
  - function definitions
  - collations declarations
  - function library imports
  - global and external variables definitions
  - etc
XQuery expressions


Expressions can be nested with full generality!

Functional programming heritage (ML, Haskell, Lisp)
Constants

XQuery grammar has built-in support for:

- Strings: “125.0” or ‘125.0’
- Integers: 150
- Decimal: 125.0
- Double: 125.e2

- 19 other atomic types available via XML Schema
- Values can be constructed
  - with constructors in F&O doc: \texttt{fn:true()}, \texttt{fn:date(“2002-5-20”)}
  - by casting
  - by schema validation
Variables

- $ + Qname (e.g. $x, $ns:foo)
- bound, not assigned
- **XQuery does not allow variable assignment**

created by **let, for, some/every, typeswitch**
expressions, function parameters

example:

```plaintext
let $x := ( 1, 2, 3 )
return count($x)
```

above scoping ends at conclusion of **return** expression
A built-in function sampler

- fn:document(xs:anyURI) => document?
- fn:empty(item*) => boolean
- fn:index-of(item*, item) => xs:unsignedInt?
- fn:distinct-values(item*) => item*
- fn:distinct-nodes(node*) => node*
- fn:union(node*, node*) => node*
- fn:except(node*, node*) => node*
- fn:string-length(xs:string?) => xs:integer?
- fn:contains(xs:string, xs:string) => xs:boolean
- fn:true() => xs:boolean
- fn:date(xs:string) => xs:date
- fn:add-date(xs:date, xs:duration) => xs:date

See Functions and Operators W3C specification
Atomization

- `fn:data(item*)` -> `xs:anyAtomicType*`
- Extracting the “value” of a node, or returning the atomic value
  - `fn:data(<a>001</a>)`
    - ("001", `xs:untypedAtomic`)
  - `fn:data(validate {<a xsi:type="xs:integer">001</a>})`
    - (1, `xs:integer`)
- Implicitly applied:
  - Arithmetic expressions
  - Comparison expressions
  - Function calls and returns
  - Cast expressions
  - Constructor expressions for various kinds of nodes
  - `order by` clauses in FLWOR expressions
Constructing sequences

(1, 2, 2, 3, 3, <a/>, <b/>)

- "\\" is the sequence concatenation operator
- Nested sequences are flattened:
  
  \[(1, 2, 2, (3, 3)) \Rightarrow (1, 2, 2, 3, 3)\]

- Range expressions: (1 to 3) => (1, 2, 3)
Combining sequences

- Union, Intersect, Except
- Work only for sequences of nodes, not atomic values
- Eliminate duplicates and reorder to document order

\[ x := <a />, y := <b />, z := <c /> \]

\[(x, y) \cup (y, z) \Rightarrow (<a />, <b />, <c />) \]

- F&O specification provides other functions & operators; eg. \texttt{fn:distinct-values()} and \texttt{fn:distinct-nodes()} particularly useful
Arithmetic expressions

1 + 4          $a \ div \ 5
5 \ div \ 6      $b \ mod \ 10
1 - (4 \ * \ 8.5) -55.5
<a>42</a> + 1    <a>baz</a> + 1
validate {<a xsi:type="xs:integer">42</a> } + 1
validate {<a xsi:type="xs:string">42</a> } + 1

Apply the following rules:

- **atomize** all operands. if either operand is (), => ()
- if an operand is untyped, cast to **xs:double** (if unable, => **error**)
- if the operand types differ but can be promoted to common type, do so (e.g.: **xs:integer** can be promoted to **xs:double**)
- if operator is consistent w/ types, apply it; result is either atomic value or **error**
- if type is not consistent, throw type exception
Logical expressions

expr1 \textbf{and} expr2

expr1 \textbf{or} expr2 \quad \textbf{fn:not}() \textbf{as a function}

\begin{itemize}
  \item return \textbf{true}, \textbf{false}
  \item \textbf{Different from SQL}
    \begin{itemize}
      \item \textbf{two} value logic, \textbf{not} \textbf{three} value logic
    \end{itemize}
  \item \textbf{Different from imperative languages}
    \begin{itemize}
      \item \textbf{and}, \textbf{or} are commutative in Xquery, but not in Java.
      \item if \((($x \text{ castable as} \ xs:\text{integer}) \text{ and } ((($x \text{ cast as x}s:\text{integer}) \text{ eq } 2)) \) …..
    \end{itemize}
  \item \textbf{Non-deterministic}
    \begin{itemize}
      \item false and error => false \textbf{or} error ! (non-deterministically)
    \end{itemize}
\end{itemize}

\textbf{Rules}:

\begin{itemize}
  \item first compute the \textit{Boolean Effective Value (BEV)} for each operand:
    \begin{itemize}
      \item if (), \textbf{“”}, NaN, 0, then return \textbf{false}
      \item if the operand is of type \textit{xs:boolean}, return it;
      \item If operand is a sequence with first item a node, return \textbf{true}
      \item else raises an error
    \end{itemize}
  \item then use standard two value Boolean logic on the two BEV's as appropriate
\end{itemize}
## Comparisons

<table>
<thead>
<tr>
<th>Value</th>
<th>for comparing single values</th>
<th>eq, ne, lt, le, gt, ge</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Existential quantification + automatic type coercion</td>
<td>=, !=, &lt;=, &lt;, &gt;, &gt;=</td>
</tr>
<tr>
<td>Node</td>
<td>for testing identity of single nodes</td>
<td>is, isnot</td>
</tr>
<tr>
<td>Order</td>
<td>testing relative position of one node vs. another (in document order)</td>
<td>&lt;&lt;, &gt;&gt;</td>
</tr>
</tbody>
</table>
Value and general comparisons

- `<a>42</a>` eq “42” true
- `<a>42</a>` eq 42 error
- `<a>42</a>` eq “42.0” false
- `<a>42</a>` eq 42.0 error
- `<a>42</a>` = 42 true
- `<a>42</a>` = 42.0 true
- `<a>42</a>` eq `<b>42</b>` true
- `<a>42</a>` eq `<b>42</b>` false
- `<a>baz</a>` eq 42 error
- `()` eq 42 ()
- `()` = 42 false
- `<a>42</a>, <b>43</b>) = 42.0 true
- `<a>42</a>, <b>43</b>) = “42” true
- `ns:shoesize(5)` eq `ns:hatsize(5)` true
- `(1,2) = (2,3)` true
Algebraic properties of comparisons

- General comparisons not reflexive, transitive
  - $(1,3) = (1,2)$ (but also $!=, <, >, <=, >=$ !!!)
  - Reasons
    - implicit existential quantification, dynamic casts
- Negation rule does not hold
  - fn:not($x = $y) is not equivalent to $x != $y
- General comparison not transitive, not reflexive
- Value comparisons are almost transitive
  - Exception:
    - xs:decimal due to the loss of precision

Impact on grouping, hashing, indexing, caching !!!
XPath expressions

- An expression that defines the set of nodes where the navigation starts + a series of selection steps that explain how to navigate into the XML tree

- A step:
  - `axis :: nodeTest`

- Axis control the navigation direction in the tree
  - `attribute, child, descendant, descendant-or-self, parent, self`
  - The other Xpath 1.0 axes (`following, following-sibling, preceding, preceding-sibling, ancestor, ancestor-or-self`) are optional in XQuery

- Node test by:
  - **Name** (e.g. publisher, myNS:publisher, *: publisher, myNS:* , *:* )
  - **Kind of item** (e.g. node(), comment(), text() )
  - **Type test** (e.g. element(ns:PO, ns:PoType), attribute(*, xs:integer) )
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
    
    \[ \text{\$x/child::person} \rightarrow \text{\$x/person} \]

- **Short-hands for common axes**
  - Descendent-or-self
    
    \[ \text{\$x/descendant-or-self::*//child::comment()} \rightarrow \text{\$x//comment()} \]

  - Parent
    
    \[ \text{\$x/parent::*} \rightarrow \text{\$x/..} \]

  - Attribute
    
    \[ \text{\$x/attribute::year} \rightarrow \text{\$x/@year} \]

  - Self
    
    \[ \text{\$x/self::*} \rightarrow \text{\$x/.} \]
Xpath filter predicates

- Syntax:
  
  expression1 [ expression2 ]

- [ ] is an overloaded operator

- Filtering by position (if numeric value):
  
  /book[3]
  /book[3]/author[1]
  /book[3]/author[1 to 2]

- Filtering by predicate:
  
  //book [author/firstname = "ronald"]
  //book [@price <25]
  //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:**
  
  ```
  let variable := expression1
  return expression2
  ```

- **Example:**
  
  ```
  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*
**FLW(O)R expressions**

- Syntactic sugar that combines FOR, LET, IF

- **Example**
  ```plaintext
  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:**
  
  ```
  for $x$ in //bib/book
  let $y := $x/author
  where $x/title=“Ulysses”$
  return count($y)
  ```

- **Equivalent to:**
  
  ```
  for $x$ in //bib/book
  return (let $y := $x/author
          return
          if ($x/title=“Ulysses” )
          then count($y)
          else ()
          )
  ```
More FLWR expression examples

- **Selections**

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

- **Joins**

  ```xml
  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

Syntax

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

```xml
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>
```
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
<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>
```
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”
Nested scopes

declare namespace ns="uri1"

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

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

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

- **In-place XQuery functions**
  
  ```xml
  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
Boundary-space Declaration
Default Collation Declaration
Base URI Declaration
Construction Declaration
Ordering Mode Declaration
Empty Order Declaration
Copy-Namespace Declaration
Schema Import
Module Import
Namespace Declaration
Default Namespace Declaration
Variable Declaration
Function Declaration
Library modules (example)

**Library module**

```xml
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**

```xml
import module namespace ns="moduleURI";
ns:add(2, ns:zero)
```
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, MXQuery, Zorba
- XQuery editors, debuggers
  - StylusStudio, oXygen