Introduction to XQuery

Overview

XQuery is an XML query language. It became a World Wide Web Consortium Recommendation in January 2007, having spent a few years prior in the status of a Candidate Recommendation.

Basic Principles

XQuery authors list the following principles beind XQuery design:

Compositionality: different expressions can be combined together.

- XQuery is a functional language.
- XQuery consists of several kinds of *expresions*.
- Results of XQery of expressions can be used as *operands* to other expressions.
- Expressions *return values* depending only on operands.
- Expressions have **no** *sideeffects*.

Closure: XQuery is closed under the XQuery data model.

- XQuery expressions are *transforms* on **XQuery data model**.
- Operands and results of XQuery expressions are proper **instances** of XQuery data model.

Schema conformance: support for XML Schema.

- XQuery fully supports XML Schema type system.
- *Primitive types.*
- *Inheritance mechanism.*
- XQuery attempts to modularize XML Schema conformance.

XPath compatibility: XPath is part of XQuery.

- XPath expressions are XQuery expressions.

- trade-offs between XML Schema and XPath resolved in favor of XML Schema.

Simplicity: I am still thinking about this one...

- Huh?
- Is XQuery really simple?

Completeness: XQuery expresses all "desirable" queries.

- Analogous to relational completeness.
- Expressive power: first-order predicate calculus over XML data + ...
- ... recursive functions[1]

Generality: XQuery is usable everywhere.

- Use with diverse applications.
- Use with diverse schema description languages (DTD, Relax-NG, XML Schema).

Conciseness: XQuery is short (?)

- implicit typecasting with numeric operations.
- comparisions implicitly include existential quantification.

Static analysis: Two stages of query execution

- *Query analyisis*: parsing, transformation, optimization.
- *Query evaluation*: execution.
- checks performed at query analyisis stage.

XQuery Data Model

Recall, that in the realm of relational databases, SQL operates on a specially defined data model. SQL statements take as input a list of tables and produce a table. Each table is viewed as a bag of tuples. Each tuple is a sequence of typed (name,value) pairs. SQL statements that return tables may be used as operands in other SQL statements where tables are desired input.

XQuery expressions operate with objects defined by XQuery Data Model. XQuery data model represents the XML data as follows:

- **XQuery data model values** are *sequences of zero or more* **items**.
- **Items** can be
  - *atomic*
  - *nodes*
- **Atomic** values have types (see XML Schema).

[1] Recall that the expressive power of relational algebra/SQL proper does not include transitive closure operation.

- **Nodes**: seven kinds (see XPath):

Node kind
document
element
text
attribute
comment
processing instruction
namespace

- **Nodes** have **identity** (see OO design).
- **Document order** is defined on nodes belonging to the same document. Document order = order of DFS traversal.
- **Nodes** are connected to other nodes. Each node may have **children**, **parents** and **siblings** (other children of their parents). The transitive closure of children are **descendants**. The transitive closure of parents are **ancestors**.
- **Document nodes** represent entire documents.
- **Document nodes** have a single "child", **root** node, which is an *element node.*
- **Sequences** are ordered collections of items. (*bears repeating*).
- **Sequences** are flat: (0,1,2,3) and (0, (1, (2,3))) is the same sequence.
- **Namespaces**. XQuery supports namespaces, i.e., *designations of "origin" of XML elements.* Namespace designations can be used for
  - elements and attributes from an input document.
  - elements and attributes from the query results.
  - functions.
  - variables.
  - types.

XQuery Constructs

XQuery is a functional language. XQuery expressions operate on specified XQuery Data Model instances, and produce new XQuery Data Model instances (sequences of items) as a result. Different types of XQuery expressions are briefly described in this section.

Comments

As an homage to the internet culture, XQuery comments are enclosed in `"(:"`, `":)"` smileys.

```
(: this is a comment :)
```

Smiley comments are not reproduced in the results of XQuery statements.

XML-style comments are allowed as well, but they **are** reproduced in the output of the XQuery expression.

```
(: This comment stays with XQuery expression :)
<!-- but this one will be found in the result -->
(: This is because XML-style comments are part of XML syntax :)
```

is actually an XQuery expression that outputs

```
<!-- but this one will be found in the result -->
```

Literals

Literals are expressions whose value is the literal itself.

Numeric literals:

- integer: $1, +1, -1$
- decimal: $1.01, -2.432,$
- floating point: $2e13, 2.3e-4, -1.2E3$

String literals: "string", 'string', "Hello, 'world'!", 'Hello, "world"!'

Strings may contain predefined *entity references* < (<), > (>), & (&), &quote; (''), ' (').

Sequences

A sequence expression is a sequence of comma-separated expressions enclosed in parentheses:

```
( Expression [, Expression [, Expression ...  ]  ])
```

([...] means *optional*).

For example,

```
('a', 1, 'sequence', "123")
```

is a sequence consisting of four literals. Any of the expressions described below can be put in a sequence.

Variables

Variables are *names* (sequences of characters that satisfy XML Recommendation's `Name` production) prefaced with a $ sign.

Examples: `$a, $b4, $prod, $myVariable123, $_not_quite_here`.

- Variables get their values from assignment expressions.
- Except for their use in iterator (`for`) expressions variables **cannot change** their values.
- **The scope of all variables is local: their values are defined only within a single XQuery expression they are a part of.**

Function calls

The syntax of function calls is straightforward:

    functionName(operands)

There are two types of functions:

- Built-in functions
- User-defined functions

Input functions

Recall that SQL queries operate on relational databases: collections of tables stored by an RDBMS. A single SQL statement uses a FROM clause to identify the subset of the tables involved in the specific SQL query.

Similarly, XQuery queries are processed over collections of XML documents. Each document is represented by a document node, and each document node has a URI associated with it.

Input functions identify data to be queried by XQuery expressions. There are two input functions in XQuery.

- `doc(<URI>)` returns the document node of the document with the specified URI.
- `collection(<URI>)` returns a collection: a sequence of nodes assicuated with the specified URI.

For example

- `doc("hello.xml")` returns the document node for the `hello.xml` XML document stored in the database (or located in the current directory).
- `doc("htpp://www.csc.calpoly.edu/ dekhtyar/hello.xml")` returns the document element for the `hello.xml` file located at the given URL.
- `collection("http://www.csc.calpoly.edu/ dekhtyar/")` returns the sequence of document elements for all XML documents located at the given URL.

Arithmetic operators

XQuery supports the following operators:

Operator	meaning
$+$	addition
$-$	subtraction
$*$	multiplication
`div`	division (any numeric operands)
`idiv`	integer division
`mod`	modulo division

Atomization

Atomization is the process by which XQuery automatically ***uses the content of the node in expressions that use the node itself***.

For example, in the expression

```
2 + <i>{2}</i>
```

the second operand is *atomized* and evaluated to 2, thus the expression returns 4.

(*see below on the meaning of* { } *in this expression.*)

Comparison expressions

XQuery handles comparisons differently than most other languages.

There are two families of comparison operators: general comparison operators and value comparison operators. These operators are pairs: each general comparison operator has a matching value comparison operator. The operators are shown in the table below:

General Comparison Operator	Value Comparison Operator	meaning
=	`eq`	equals to
!=	`ne`	not equals to
<	`lt`	less than
<=	`le`	less than or equal to
>	`gt`	greater than
>=	`ge`	greater than or equal to

In addition, a special operator, `is` is reserved for node comparison.

Value Comparison operators

Value comparison operators work in the same way as comparisons in SQL or most programming languages.

The operands to value comparison operators must be single expressions (not sequences). Examples:

Expression	Value
`3 gt 2`	`true`
`<a>4</a> eq <b>4</b>`	`true`
`<a>04</a> le <b>6</b>`	`false` (`<a>04</a>` is untyped)
`(1,3) eq (2,3)`	type error

General comparison expressions

General comparison expressions work siimilarly to value comparison expressions when the operands are one-item sequences. However, when the operands are multi-item sequences, general comparison expressions have different semantics.

An expression

```
(ExpA1,...,ExpAk) Op (ExpB1,...,ExpBm)
```

is evaluated to

```
(ExpA1 Op ExpB1) or ... or (ExpA1 and ExpBm)
                 or ... or (ExpAk Op ExpB1) or ... or (ExpAk Op ExpBm)
```

That is, a general comparison expression evaluates to `true` **iff** there exists a pair of items from the operan sequences such that their comparison using the comparison operator evaluates to `true`.

For example, $(2,3) > (2,5)$ evaluates to the value of the following expression: `(2 > 2) or (2 > 5) or (3 > 2) or (3 > 5)`. Because, $3 > 5$ evaluates to `true`, so does `(2,3)>(2,5)`.

Be careful when using general comparisons in your XQuery expressions. If the comparison operands are sequences, the results may be unexpected.

Node comparison expressions

Node comparison expression has the following syntax:

```
<NodeExpression1> is <NodeExpression2>
```

This expression evaluates to `true` **iff** both `NodeExpression1` and `NodeExpression2` evaluate to the same node (the comparison is based on **node identity**, not *node content*).

Logical Expressions

Logical Expressions have the following syntax:

```
Expression1 and Expression2

Expression1 or Expression2
```

Note that negation is also possible, but in XPath (and from it, in XQuery), not() is a built-in function, i.e, the negation of an expression `Expression` is `not(Expression)`.

Conditional expressions

Conditional expressions have the syntax:

```
if (<TestExpression>)
   then <TestTrueExpression>
  [else <TestFalseExpression>]
```

The semantics of this expression is similar to the semantics of the ? expression from C: first `TestExpression` is evaluated. If its value is `true`, then the conditional expression is evaluated to the value of `TestTrueExpression`. If `else` clause is present, and `TestExpresion` evaluates to false, then the conditional expression evaluates to the result of `TestFalseExpressions`.

For example

```
if ($n gt 5) then $n else 5
```

will return the value of `$n` if it is greater than 5, otherwise, it will return 5.

Conditional expressions (as expected) can be nested.

Path Expressions

XPath is a proper subset of XQuery. Any XPath expression is a proper XQuery expression. In XQuery, path expressions are combined with input functions, to specify initial context for the path expressions. The value of the path expression is the sequence of items returned by the XPath evaluation of the path expression on the context supplied by the input function.

Evaluate expression

The syntax of the evaluate expression is

```
{ Expression }
```

The evaluate expression **forces an immediate evaluation of its operand**.

In many XQuery contexts, e.g., `$n = 2+2` the operands of the expression, such as $2 + 2$ are evaluated immediately. However, in some other contexts, in particular, in XML content, these expressions are treated as untyped content, and are not evaluated. E.g.,

```
<a>2+3</a>
```

evaluates to `<a>2+3</a>`, and

```
<a>$n</a>
```

evaluates to `<a>$n</a>`.

If we want to force the evaluation of the expressions $2 + 3$ or `$n`, we can use the immediate evaluate expression.

```
<a>{2+3}</a>
```

evaluates to `<a>5</a>`.

Constructors

XQuery provides mechanisms for explicitly constructing XML elements, XML attributes and XML documents. This is done through the means of constructors.

Elements can be constructed in two ways: by building an XML fragment, or by using an explicit element constructor. Explicit element constructor has the following syntax:

```
element Name { Expression }
```

This evaluates to

```
<Name>{Expression}</Name>
```

or

```
<Name Attributes>{Expression - Attributes}</Name>
```

if `Expression` includes attribute constructors for node `<Name>`.

The following three element constructors evaluate to the same XML element:

```
element Pair {
      element X {25},
      element Y {30}
              }
```

```
<Pair>
    <X>{30 - 5}</X>
    <Y>{20 + 10}</Y>
</Pair>
```

```
<Pair>
    <X>25</X>
    <Y>30</Y>
</Pair>
```

The last expression is also the value of all three of these constructors.

Document constructor. A special document node constructor has the syntax:

```
document { Expression }
```

The result of this expression is a document node. Not the difference between the following two constructors:

```
document {<a> <b>1</b><c>2</c></a>}

element{<a> <b>1</b><c>2</c></a>}
```

The first expression returns the document node, whose child document root node is the same as the result of the second expression.

Attribute constructor. Attributes can be constructed within the XML element constructors:

```
<a id="1">Hello, World!</a>
```

returns an element node for `<a>` which has an attribute node for the name-value pair `(id, "1")` associated with it.

There is also an explicit attribute constructor:

```
attrbite Name { Expression }
```

The following expression

```
attribute id {"1"}
```

produces the `id="1"` attrbite node.

FLOWR Expressions

The **heart** of XQuery are FLOWR (flower) expressions. FLOWR stands for

```
for
let
order by
where
return
```

In a nutshell, FLOWR expressions are analogous to the SELECT-FROM-WHERE-ORDER BY statements in SQL.

We discuss the meaning of each clause in this expression in turn.

return clause

`return` clause is the only mandatory clause of a FLOWR statement. It syntax is

```
return Expresson
```

If no other clause appears in the FLOWR expression, then the result of `return Expression` is the value of the `Expression`.

However, when other clauses are present, evaluation of `return Expression` is changed.

For example, consider the following XML document `test.xml`.

```
<test>
  <a id="1">
      <text>Hello, world!</text>
      <number>2</number>
  </a>
  <a id ="2"> <number>3</number></a>
  <b><c>1</c>
     <d>2</d></b>
</test>
```

The following XQuery expression

```
return doc("test.xml")/descendant::number
```

evaluates to the sequence of two XML element nodes:

```
<number>2</number>
<number>3</number>
```

let clause

The `let` clause assigns a value to a local variable. The format of the `let` clause is

```
let Variable := Expression
```

There may be multiple `let` clauses in a single XQuery FLOWR statement, i.e., multiple variables can be defined.

Consider the following query:

```
let $anode := doc("test.xml")/descendant::number
return <nnn>{$anode}</nnn>
```

This query returns the following result on the `test.xml` input:

```
<nnn><number>2</number><number>3</number></nnn>
```

Note: each `let` clause in the query must define a different variable.

for clause

The `for` clause provides a convenient iterator over a sequence of items. The syntax of the `for` clause is

```
for Variable in Expression
   RestOfFLOWR
```

The `for` clause is evaluated as follows. `Expression` is evaluated. If it is evaluated to a sequence, `Variable` recieves the value of the first item in the sequence. The expression `RestOfFLOWR` is then evaluated, with `Variable` set to this value. When the evaluation of `RestofFLOWR` stops, `Variable` is assigned the value of the next item in the sequence returned by `Expression` and `RestOfFLOWR` is evaluated again.

This corresponds to the work of traditional iterators or `foreach Var in Set` statements in various programming languages (e.g., Perl).

`RestOfFLOWR` may contain any other FLOWR clauses, including arbitarily nested `let` and `for` clauses.

In contrast to the previous query example, consider the following query:

```
for $anode in doc("test.xml")/descendant::number
return <nnn>{$anode}</nnn>
```

This query returns the following result on the `test.xml` input:

```
<nnn><number>2</number></nnn>
<nnn><number>3</number></nnn>
```

Here, `doc("test.xml")/descendant::number` evaluates to the same sequence of two `<number>` nodes. But whereas `let $anode :=` clause assigned to `$anode` the entire sequence, the `for $anode in` clause starts an iterator over the contents of the returned sequence. On step 1, `$anode` evaluates to the first element of the sequence, a node representing `<number>2</number>`. The `return` clause is then evaluated, and it outputs `<nnn><number>2</number></nnn>`. This is not all, as there are still more items to be iterated over. On the second iteration, `$anode` evaluates to `<number>3</number>`, and the `return` clause will add the line `<nnn><number>3</number></nnn>` to the output.

Positional variables The `for` clause provides another convenient feature: the ability to instantiate not only the ith element of the sequence, but also, to intantiate the position of the current element in the sequence. The syntax is:

```
for PosVariable at Variable in Expression
```

On the first iteration, PosVariable instantiates to 1, on the second - to 2, etc...

```
for $i at $n in doc("test.xml")//number
return $i+$n
```

returns two numbers: 3 (1+2) and 5 (2+3).

where clause

The where clause is the analog of SQL's WHERE clause in the SELECT (as well as UPDATE and DELETE) statements. It format is

```
where BooleanExpression
```

The semantics of the where clause is straightforward: on each iteration of the (possibly nested) for iterator(s), the value of the expression in the where is evaluated. If the expression is evaluated to true, then the evaluation proceeds to the return clause. If the expression evaluates to false, then the evaluation proceeds to the next value of the innermost iterator.

order by clause

Syntax:

```
order by <Expression>[, Expression [, ... [, Expression]]]
```

Determines how to order output.

```
for $i in doc("test.xml")//number
order by $i descending
return $i
```

will return

```
<number>3</number>
<number>2</number>
```

Order is defined on numbers, as well as on strings.