Information Systems

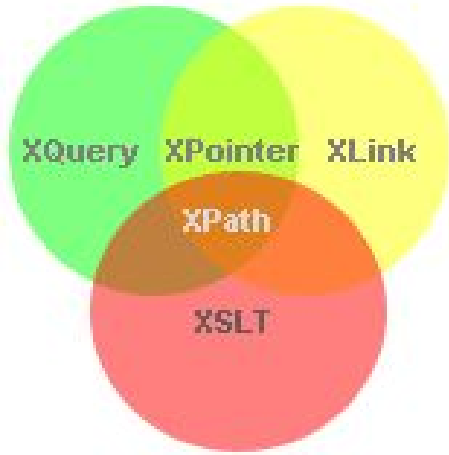
XPath and XQuery

Nikolaj Popov

Research Institute for Symbolic Computation
Johannes Kepler University of Linz, Austria
popov@risc.uni-linz.ac.at

Outline

XPath



What is XPath?

- XPath is a language whose primary purpose is to provide common syntax and functionality to address parts of XML documents.
- XPath uses path expressions to navigate in XML documents.
- XPath contains a library of standard functions.

XPath

- XPath operates on the logical structure of an XML document and uses a syntax that resembles to the path constructions in URIs.
- XPath models an XML document as a tree of nodes (e.g. elements, attributes, namespaces, etc.)
- XPath expressions can compute strings, numbers, sets of nodes from the data of XML documents.

Location Paths

- Location paths are special expressions for selecting a set of nodes.
- A location path consists of location steps composed together from left to right and separated by ’/’.
- An absolute location path is one that starts with a ’/’.
- Relative location paths are defined always with respect to the context node.

Example

The node selection is analogous to the file selection in a Unix-like file system.

```
../reports/*/summary
```

Location Paths

Example

Path Expression	Result
`/bookstore`	Selects the root element `bookstore` Note: If the path starts with a slash (`/`) it always represents an absolute path to an element!
`bookstore/book`	Selects all `book` elements that are children of `bookstore`.

Location Paths

Example

Path Expression	Result
`//book`	Selects all `book` elements no matter where they are in the document.
`bookstore//book`	Selects all `book` elements that are descendant of the `bookstore` element, no matter where they are under the `bookstore` element.
`//@lang`	Selects all attributes that are named `lang`.

Predicates

- Predicates are used to find a specific node or a node that contains a specific value.
- Predicates are always embedded in square brackets.

Example

Path Expression	Result
`/bookstore/book[1]`	Selects the first `book` element that is the child of the `bookstore` element.
`/bookstore/book[last()-1]`	Selects the last but one `book` element that is the child of the `bookstore` element.
`/bookstore/book[position()<3]`	Selects the first two `book` elements that are children of the `bookstore` element.

Predicates

Example

Path Expression	Result
`//title[@lang='eng']`	Selects all the title elements that have an attribute named lang with a value of `'eng'`.
`/bookstore/book[price>35.00]`	Selects all the `book` elements of the `bookstore` element that have a `price` element with a value greater than 35.00.

Selecting Unknown Nodes

XPath wildcards can be used to select unknown XML elements.

Example

Wildcard	Result
`/bookstore/*`	Selects all the child nodes of the bookstore element.
`//*`	Selects all elements in the document.
`//title[@*]`	Selects all title elements which have any attribute.

Selecting Several Paths

By using the | operator in an XPath expression you can select several paths.

Example

Path Expression	Result
`//title | //price`	Selects all the `title` AND `price` elements in the document.

Location Steps

Location steps have the following parts:

- **axis**. It specifies the (in-tree) relationship between the context node and the nodes selected by the location step:

  Available axes: child, descendant, parent, ancestor, self, descendant-or-self, ancestor-or-self. (Used explicitly in "long notation")
- **node test**. Specifies the node type for the nodes selected by the location step (separated by :: from the axis).
- **predicate**. It specifies further expressions with boolean value, to refine the selected node set (enclosed in [ ], described on before).

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<folder>
  <email date='14 Dec 2009'>
    <from>robert@company.com</from>
    <to>oliver@company.com</to>
    <subject>Meeting</subject>
    Could we meet this week to discuss the
    interface problem in the NTL project?  -Rob
  </email>
</folder>
```

- This location path (using the “long notation”) selects all attributes of all the email elements.
  `/child::folder/child::email/attribute::*`
- This selects only the attribute of the first email element in the XML document.
  `/child::folder/child::*[position()=1]/@*`

XPath Expressions

- Simple expressions: numerical and string literals, variable references, function calls.
- The value of a variable `x` can be retrieved by `$x`.
- Basic arithmetic operations are available for numbers.
- More complex expressions are location paths and boolean expressions (e.g. using $<$, $>$, $!=$, $=$ and logical connectives `and`, `or`).

What Is XQuery

- The purpose of XQuery is to provide a language for extracting data from XML documents.
- Queries can operate on more than one documents at once. Subsets of nodes can be selected using XPath expressions.
- The query language is functional (but it also includes universal and existential quantifiers), supports simple and complex data types defined in XML Schema.

Simple Examples

vehicles.xml. A Sample XML Document Containing Vehicle Data:

```
<?xml version="1.0" encoding="utf-8"?>
  <vehicles>
    <vehicle year="2007" make="Opel" model="Astra">
      <mileage>13495</mileage>
      <color>green</color>
      <price>13900</price>
      <options>
        <option>navigation system</option>
        <option>heated seats</option>
      </options>
    </vehicle>
...
```

Simple Examples

vehicles.xml. A Sample XML Document Containing Vehicle Data (cont):

```
    <vehicle year="2008" make="Opel" model="Astra">
     <mileage>07541</mileage>
     <color>white</color>
     <price>13900</price>
     <options>
       <option>spoiler</option>
       <option>Traction Control</option>
     </options>
   </vehicle>
   <vehicle year="2007" make="Opel" model="Astra">
     <mileage>18753</mileage>
     <color>white</color>
     <price>12900</price>
     <options />
   </vehicle>

</vehicles>
```

Simple Examples

- The query that retrieves all of the color elements from the vehicles:

```
xquery version "1.0";
for $c in doc("vehicles.xml")//color
return $c
```

- Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<color>green</color>
<color>white</color>
<color>white</color>
```

- Explanation:
  - `doc("vehicles.xml")` is used to open vehicles.xml file.
  - `doc("vehicles.xml")//color` selects all color elements in the document.
  - `$c` is a variable.

Simple Examples

Using filters:

- Return any vehicle elements that contain a `color` element with a value of `green`:

```
xquery version "1.0";
for $v in doc("vehicles.xml")//vehicle[color='green']
return $v
```

- Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<vehicle year="2007" make="Opel" model="Astra">
<mileage>13495</mileage>
<color>green</color>
<price>13900</price>
<options>
  <option>navigation system</option>
  <option>heated seats</option>
</options>
</vehicle>
```

Simple Examples

Using filters:

- Find all of the vehicles with a color of green or a price less than 14000:

```
xquery version "1.0";
for $v in
  doc("vehicles.xml")//vehicle[color='green' or
                               price<'14000']
return $v
```

Simple Examples

Using filters:

- Find `options` of all the white cars:
  `//vehicle[color='white']/options`

Using wildcards:

- Find the `option` elements that are one layer below `vehicle`:

```
 for $o in vehicles/vehicle/*/option
return $o
```

Simple Examples

Referencing attributes:

- Find all the `year` attributes of `vehicle` elements:
  `//vehicle/@year`
- Return all of the vehicle elements that have year attributes as well:
  `//vehicle[@year]`
- Return all the vehicle elements that have a year attribute with the value 2008:

```
for $v in //vehicle[@year="2008"]
return $v
```

Simple Examples

Processing results:

- Query results can be packaged within other surrounding XML code to create transformed data.
- To incorporating query results into surrounding code, query data is put within curly braces (`{}`).
- Access the content within a node by calling the XQuery `data()` function and supplying it with the node in question.
- Query:

```
for $c in //color
return <p>Vehicle color:  {data($c)}</p>
```

- Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<p>Vehicle color:  green</p>
<p>Vehicle color:  white</p>
<p>Vehicle color:  white</p>
```

FLWOR Expressions

- FLWOR is an acronym for “For, Let, Where, Order by, Return”. Reads as “flower”.
- ```
  xquery version "1.0";

    for $v in //vehicle
    let $y := $v/price
    where $v/mileage > '10000'
    order by $y
    return
      {data($v/@model)} - {data($y)}

  ```
- `for` - (optional) binds a variable to each item returned by the `in` expression
- `let` - (optional)
- `where` - (optional) specifies criteria
- `order by` - (optional) specifies the sort-order of the result
- `return` - specifies what to return in the result
  ```

FLWOR Expressions

- The `for` clause binds a variable to each item returned by the `in` expression.
- The `for` clause results in iteration.
- There can be multiple `for` clauses in the same FLWOR expression.
- To loop a specific number of times in a for clause, you may use the `to` keyword:

```
for $x in (1 to 3)
return <test>$x</test>
```

- Returns

```
<test>1</test>
<test>2</test>
<test>3</test>
```

FLWOR Expressions

- It is allowed with more than one in expression in the `for` clause.
- Use comma to separate each in expression:

```
for $x in (10,20), $y in (100,200)
return <test>x=$x and y=$y</test>
```

- Result:

```
<test>x=10 and y=100</test>
<test>x=10 and y=200</test>
<test>x=20 and y=100</test>
<test>x=20 and y=200</test>
```

FLWOR Expressions

- The `let` clause allows variable assignments and avoids repeating the same expression many times.
- The `let` clause does not result in iteration.

```
let $x := (1 to 5)
return <test>$x</test>
```

- Result:

```
<test>1 2 3 4 5</test>
```

FLWOR Expressions

- The `where` clause is used to specify one or more criteria for the result.
- The `order by` clause is used to specify the sort order of the result.
- The `return` clause specifies what is to be returned.

Basic Syntax Rules

- XQuery is case-sensitive
- XQuery elements, attributes, and variables must be valid XML names
- An XQuery string value can be in single or double quotes
- An XQuery variable is defined with a `$` followed by a name, e.g. `$vehicle`
- XQuery comments are delimited by `(:` and `:)`, e.g.

```
(:  XQuery Comment :)
```

Conditional Expressions

```
xquery version "1.0";
<p>
  {
    for $v in //vehicle
    return if (data($v/options) != "")
    then
      <div>Options for $v:  data($v/options)</div>
    else
      <div>The vehicle $v has no options</div>
  }
</p>
```

XQuery Comparisons

- Two ways of comparing values.
  1. General comparisons: `=, !=, <, <=, >, >=`
  2. Value comparisons: `eq, ne, lt, le, gt, ge`
- Difference between the two comparison methods:
- `//vehicle/@year > '2007'`
  Returns true if any `year` attributes have values greater than '2007'.
- `//vehicle/@year gt '2007'`
  Returns true if there is only one `year` attribute returned by the expression, and its value is greater than '2007'. If more than one `year` is returned, an error occurs.

Summary

- XPath provides a language for addressing parts of XML documents.
- XPath uses path expressions to navigate in XML documents.
- XQuery provides a language for extracting data from XML documents.
- Queries can operate on more than one documents at once and may become inputs for other queries.