Chapter 3

XPath
XPath is a language that lets you identify particular parts of XML documents.

XPath interprets XML documents as nodes (with content) organised in a tree structure.

XPath indicates nodes by (relative) position, type, content, and several other criteria.

Basic syntax is somewhat similar to that used for navigating file hierarchies.

XPath 1.0 (1999) and 2.0 (2010) are W3C recommendations.
Some Tools for XPath

- **Saxon** (specifically Saxon-HE which implements XPath 2.0, XQuery 1.0 and XSLT 2.0)
- **eXist-db** (a native XML database supporting XPath 2.0, XQuery 1.0 and XSLT 1.0)
- **XPath Checker** (add-on for Firefox)
- **XPath Expression Testbed** (available online)
Data Model

XPath’s data model has some non-obvious features:

- The tree’s root node is not the same as the document’s root (document) element
- The tree’s root node contains the entire document including the root element (and comments and processing instructions that appear before it)
- XPath’s data model does not include everything in the document: XML declaration and DTD are not addressable
- xmlns attributes are reported as namespace nodes
There are 6 types of *node*:

- *root*
- *element*
- *attribute*
- *text*
- *comment*
- *processing instruction*

Element nodes have an associated set of attribute nodes

Attribute nodes are *not* children of element nodes

The order of child element nodes is *significant*

We will only consider the first 4 types of node
Example (1)

Consider representing information about CDs in XML as follows:

```xml
<CD-library>
  <CD number="724356690424">
    <performance>
      <composer>Frederic Chopin</composer>
      <composition>Waltzes</composition>
      <soloist>Dinu Lipatti</soloist>
      <date>1950</date>
    </performance>
  </CD>
...
Example (2)

...<CD number="419160-2">
   <composer>Johannes Brahms</composer>
   <soloist>Emil Gilels</soloist>
   <performance>
      <composition>Piano Concerto No. 2</composition>
      <orchestra>Berlin Philharmonic</orchestra>
      <conductor>Eugen Jochum</conductor>
      <date>1972</date>
   </performance>
   <performance>
      <composition>Fantasias Op. 116</composition>
      <date>1976</date>
   </performance>
</CD>
...

Example (3)

...  
<CD number="449719-2">  
  <soloist>Martha Argerich</soloist>  
  <orchestra>London Symphony Orchestra</orchestra>  
  <conductor>Claudio Abbado</conductor>  
  <date>1968</date>  
  <performance>  
    <composer>Frederic Chopin</composer>  
    <composition>Piano Concerto No. 1</composition>  
  </performance>  
  <performance>  
    <composer>Franz Liszt</composer>  
    <composition>Piano Concerto No. 1</composition>  
  </performance>  
</CD>  
...
Example (4)

...<CD number="430702-2">
  <composer>Antonin Dvorak</composer>
  <performance>
    <composition>Symphony No. 9</composition>
    <orchestra>Vienna Philharmonic</orchestra>
    <conductor>Kirill Kondrashin</conductor>
    <date>1980</date>
  </performance>
  <performance>
    <composition>American Suite</composition>
    <orchestra>Royal Philharmonic</orchestra>
    <conductor>Antal Dorati</conductor>
    <date>1984</date>
  </performance>
</CD>
</CD-library>
Example — Tree Structure
Location Path

- The most useful XPath expression is a location path: e.g., /CD-library/CD/performance
- A location path consists of at least one location step: e.g., CD-library, CD and performance are location steps
- A location step takes as input a set of nodes, also called the context (to be defined more precisely later)
- The location step expression is applied to this node set and results in an output node set
- This output node set is used as input for the next location step
Location Path (2)

- There are two different kinds of location paths:
  - *Absolute* location paths
  - *Relative* location paths

- An absolute location path
  - starts with `/`
  - is followed by a relative location path
  - is evaluated at the root (context) node of a document
  - e.g., `/CD-library/CD/performance`

- A relative location path
  - is a sequence of location steps
  - each separated by `/`
  - evaluated with respect to some other context nodes
  - e.g., `CD/performance`
Evaluation of absolute location path
Evaluation of absolute location path

/
Evaluation of absolute location path

/CD-library
Evaluation of absolute location path

/CD-library/CD
Evaluation of absolute location path

/CD-library/CD/performance
Location Step

- In general, a location step in turn consists of a
  - (navigation) axis
  - node test
  - predicate(s)

- Syntax is $axis :: node\ test\ [\ predicate\ ] ... [\ predicate\ ]$
- e.g., `child::CD[composer=’Johannes Brahms’]`
  - child is the axis
  - CD is the node test
  - composer=’Johannes Brahms’ is the predicate

- A location step is applied to each node in the context (i.e., each node becomes the context node)
- All resulting nodes are added to the output set of this location step
Evaluation of predicate

/child::CD-library/child::CD
Evaluation of predicate

/child::*:CD-library/child::*:CD[composer='Johannes Brahms']
A axes specifies what nodes, relative to the current context node, to consider.

There are 13 different axes (some can be abbreviated):

- self, abbreviated by .
- child, abbreviated by empty axis
- parent, abbreviated by..
- descendant-or-self, abbreviated by empty location step
- descendant, ancestor, ancestor-or-self
- following, following-sibling, preceding, preceding-sibling
- attribute, abbreviated by @
- namespace
The following slides show (graphical) examples of the axes, assuming the node in bold typeface is the context node.
Self-Axis

- The self-axis just contains the context node itself
Child-Axis

- The child-axis contains the children (direct descendants) of the context node
Parent-Axis

- The parent-axis contains the parent (direct ancestor) of the context node.
Descendant-Axis

- The descendant-axis contains all direct and indirect descendants of the context node
Descendant-Or-Self-Axis

- The descendant-or-self-axis contains all direct and indirect descendants of the context node + the context node itself
Ancestor-Axis

- The ancestor-axis contains all direct and indirect ancestors of the context node
Ancestor-Or-Self-Axis

- The ancestor-or-self-axis contains all direct and indirect ancestors of the context node + the context node itself
Following-Axis

- The following-axis contains all nodes that begin after the context node ends
Preceding-Axis

- The preceding-axis contains all nodes that end before the context node begins
Following-Sibling-Axes

- The following-sibling-axis contains all following nodes that have the same parent as the context node.
The preceding-sibling-axis contains all preceding nodes that have the same parent as the context node.
Partitioning

The axes self, ancestor, descendant, following and preceding partition a document into five disjoint subtrees:
Attribute-Axis

- Attributes are handled in a special way in XPath.
- The attribute-axis contains all the attribute nodes of the context node.
- This axis is empty if the context node is not an element node.
- Does not contain `xmlns` attributes used to declare namespaces.
Namespace-Axis

- The namespace-axis contains all namespaces in scope of the context node.
- This axis is empty if the context node is not an element node.
Node Tests

- Once the correct relative position of a node has been identified the type of a node can be tested.
- A *node test* further refines the nodes selected by the location step.
- A double colon `::` separates the axis from the node test.
- There are seven different kinds of node tests:

  ```
  name
  prefix::*
  node()
  text()
  comment()
  processing-instruction()
  *
  ```
The *name* node test selects all elements with a matching name
- e.g., if our context is a set of 4 CD elements and the location step uses the *child* axis, then we get element nodes with different names
- we can use the *name* node test to return, e.g., only *soloist* elements

Along the attribute-axis it matches all attributes with the same name
Prefix:* 

- Along an element axis, all nodes whose namespace URIs are the same as the prefix are matched.
- This node test is also available for attribute nodes.
Comment, Text, Processing-Instruction

- `comment()` matches all comment nodes
- `text()` matches all text nodes
- `processing-instruction()` matches all processing instructions
Node and *

- `node()` selects all nodes, regardless of type: attribute, namespace, element, text, comment, processing instruction, and root.
- `*` selects all element nodes, regardless of name.
  - If the axis is the attribute axis, then it selects all attribute nodes.
  - If the axis is the namespace axis, then it selects all namespace nodes.
### Key for full CD library example

<table>
<thead>
<tr>
<th>Element name</th>
<th>Abbreviation</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td></td>
<td>black</td>
</tr>
<tr>
<td>library</td>
<td>L</td>
<td>white</td>
</tr>
<tr>
<td>cd</td>
<td>C</td>
<td>grey</td>
</tr>
<tr>
<td>performance</td>
<td>p</td>
<td>pink</td>
</tr>
<tr>
<td>composer</td>
<td>c</td>
<td>blue</td>
</tr>
<tr>
<td>composition</td>
<td></td>
<td>green</td>
</tr>
<tr>
<td>soloist</td>
<td>s</td>
<td>yellow</td>
</tr>
<tr>
<td>conductor</td>
<td>t</td>
<td>red</td>
</tr>
<tr>
<td>orchestra</td>
<td>o</td>
<td>brown</td>
</tr>
<tr>
<td>date</td>
<td>d</td>
<td>orange</td>
</tr>
</tbody>
</table>
Full CD library example
Example using * and node()

/CD-library/CD/*/node()
Example showing difference between * and node()

/CD-library/CD/*/(*
Example using descendant

//composer or /descendant-or-self::node()/composer
Another example using descendant

//performance/composer or
/descendant-or-self::performance/child::composer
Predicates

- A node set can be reduced further with *predicates*
- While each location step must have an axis and a node test (which may be empty), a predicate is optional
- A predicate contains a Boolean expression which is tested for each node in the resulting node set
- A predicate is enclosed in square brackets `[ ]`
XPath supports a full complement of relational operators, including =, >, <, >=, <=, !=

XPath also provides Boolean and and or operators to combine expressions logically

In some cases a predicate may not be a Boolean; then XPath will convert it to one implicitly (if that is possible):
- an empty node set is interpreted as false
- a non-empty node set is interpreted as true
Example using a predicate

//performance[composer]
Another example using a predicate

//CD[performance/orchestra]
Example using multiple predicates

//performance[conductor][date]
Functions

- XPath provides many functions that may be useful in predicates.
- Each XPath function takes as input or returns one of these four types:
  - node set
  - string
  - Boolean
  - number
More about Context

- Each location step and predicate is evaluated with respect to a given *context*
- A specific context is defined as \(<N_1, N_2, \ldots N_m, N_c>\) with
  - a *context list* \(<N_1, N_2, \ldots N_m>\) of nodes in the tree
  - a *context node* \(N_c\) belonging to the list
- The *context length* \(m\) is the size of the context list
- The *context node position* \(c \in [1, m]\) gives the position of the context node in the list
More about XPath Evaluation

- Each step $s_i$ is interpreted with respect to a context; its result is a node list.
- A step $s_i$ is evaluated with respect to the context of step $s_{i-1}$.
- More precisely:
  - for $i = 1$ (first step)
    - if the path is absolute, the context is the root of the XML tree;
    - else (relative paths) the context is defined by the environment;
  - For $i > 1$
    - if $\mathcal{N} = \langle N_1, N_2, \ldots N_m \rangle$ is the result of step $s_{i-1}$,
      step $s_i$ is successively evaluated with respect to the context $(\mathcal{N}, N_j)$,
      for each $j \in [1, m]$.
- The result of the path expression is the node list obtained after evaluating the last step.
Node-set Functions

- **Node-set functions** operate on or return information about node sets

Examples:

- `position()`: returns a number equal to the position of the current node in the context list
  - `[position()=i]` can be abbreviated as `[i]`
- `last()`: returns the size (i.e. the number of nodes in) the context list
- `count(set)`: returns the size of the argument node set
- `id()`: returns a node set containing all elements in the document with any of the specified IDs
Example about context

- The expression //CD/performance[2] returns the second performance of each CD, not the second of all performances.
- The result of the step CD is the list of the 4 CD nodes.
- The step performance[2] is evaluated once for each of 4 CD nodes in the context.
Example about context (2)

- The result is the list comprising the second performance element child of each CD:

1. `<performance>
   <composition>Fantasias Op. 116</composition>
   <date>1976</date>
</performance>

2. `<performance>
   <composer>Franz Liszt</composer>
   <composition>Piano Concerto No. 1</composition>
</performance>

3. `<performance>
   <composition>American Suite</composition>
   <orchestra>Royal Philharmonic</orchestra>
   <conductor>Antal Dorati</conductor>
   <date>1984</date>
</performance>`
More about the position() function

- `position()` is a function that returns the position of the current node in the context node set.
- For most axes it counts forward from the context node.
- For the “backward” axes it counts backwards from the context node.
- The “backward” axes are: ancestor, ancestor-or-self, preceding, and preceding-sibling.
Examples using position()

- So, to get the CD immediately before the one that was composed by Dvorak:
  \[//\text{CD[composer='Antonin Dvorak']}/\text{preceding::CD[1]}\]
- This selects the third CD
- To get the last CD (without having to know how many there are), use
  \[//\text{CD[position()=last()]}\]
Example using a different axis

//date/following-sibling::* returns the following:

1. <performance>
   <composer>Frederic Chopin</composer>
   <composition>Piano Concerto No. 1</composition>
</performance>

2. <performance>
   <composer>Franz Liszt</composer>
   <composition>Piano Concerto No. 1</composition>
</performance>

only one date element in the document has any following siblings
Examples using count

- Assume we want the CDs containing only one orchestra element
- \(/ \text{CD}[\text{count(orchestra)=1}]\) returns only one CD, where the orchestra is “London Symphony Orchestra”
- This is because we are counting the orchestra children of \(\text{CD}\) elements
- But orchestras are also represented below performance elements
Examples using count

- Assume we want the CDs containing only one orchesta element
- \(//CD[count(orchestra)=1]\) returns only one CD, where the orchestra is “London Symphony Orchestra”
- This is because we are counting the orchestra children of CD elements
- But orchestras are also represented below performance elements
- What about \(//CD[count(//orchestra)=1]\)?
  - But \(//orchestra\) is an absolute expression evaluated at the root
  - So the answer to \(count(//orchestra)\) is 4, not 1
Examples using count

- Assume we want the CDs containing only one orchestra element.
- \(/\text{CD}[\text{count(orchestra)}=1]\) returns only one CD, where the orchestra is “London Symphony Orchestra”.
- This is because we are counting the orchestra \textit{children} of \textit{CD} elements.
- But orchestras are also represented below performance elements.
- What about \(/\text{CD}[\text{count(//orchestra)}=1]\)?
  - But //orchestra is an absolute expression evaluated at the root.
  - So the answer to \text{count}(//orchestra) is 4, not 1.
- What we need is \(/\text{CD}[\text{count(.//orchestra)}=1]\), where “.” represents the current context node.
  - This gives us the CDs with the “Berlin Philharmonic” and “London Symphony Orchestra”.
String Functions

- **String functions** include basic string operations
- Examples:
  - `string-length()`: returns the length of a string
  - `concat()`: concatenates its arguments in order from left to right and returns the combined string
  - `contains(s1, s2)`: returns true if `s2` is a substring of `s1`
  - `normalize-space()`: strips all leading and trailing whitespace from its argument
Boolean Functions

 octave functions always return a Boolean with the value true or false:

- `true()`: simply returns true (makes up for the lack of Boolean literals in XPath)
- `false()`: returns false
- `not()`: inverts its argument (i.e., true becomes false and vice versa)
Boolean Functions

Boolean functions always return a Boolean with the value true or false:

- true(): simply returns true (makes up for the lack of Boolean literals in XPath)
- false(): returns false
- not(): inverts its argument (i.e., true becomes false and vice versa)

Examples:

- //performance[orchestra][not(conductor)] returns performance elements which have an orchestra child but no conductor child
- //CD[not(.//soloist)] returns CDs containing no soloists
**Boolean Functions (2)**

- `boolean()`: converts its argument to a Boolean and returns the result
  - Numbers are false if they are zero or NaN (not a number)
  - Node sets are false if they are empty
  - Strings are false if they have zero length
Number Functions

- **Number functions** include a few simple numeric functions

**Examples:**
- `sum(set)`: converts each node in a node set to a number and returns the sum of these numbers
- `round()`, `floor()`, `ceiling()`: round numbers to integer values
XPath is used to navigate through elements and attributes in an XML document.

XPath is a major element in many W3C standards: XQuery, XSLT, XLink, XPointer.

It is also used to navigate XML trees represented in Java or JavaScript, e.g.

So an understanding of XPath is fundamental to much advanced XML usage.