Part VIII

XPath—Navigating XML Documents

Outline of this part

1 XPath—Navigational access to XML documents
- Context
- Location steps
- Navigation axes
- Examples

2 XPath Semantics
- Document order & duplicates
- Predicates
- Atomization
- Positional access

XPath—Navigational access to XML documents

- In a sense, the **traversal** or **navigation** of trees of XML nodes lies at the core of every XML query language.
- To this end, XQuery *embeds* **XPath** as its tree navigation sub-language:
  - Every XPath expression also is a correct XQuery expression.
  - XPath 2.0: W3C `http://www.w3.org/TR/xpath20/`.
- Since navigation expressions extract (potentially huge volumes of) nodes from input XML documents, the efficient implementation of the sub-language XPath is a prime concern when it comes to the construction of XQuery processors.

Context node

- In XPath, a path traversal starts off from a **sequence of context nodes**.
  - XPath navigation syntax is simple:

An XPath step

$$cs_0 / step$$

- cs_0 denotes the context node sequence, from which a navigation in direction *step* is taken.

  - It is a common error in XQuery expressions to try and start an XPath traversal *without* the context node sequence being actually defined.

Multiple steps

- An XPath navigation may consist of **multiple steps** $step_i$, $i \geqslant 1$ taken in succession.
- Step $step_1$ starts off from the context node sequence cs_0 and arrives at a sequence of new nodes cs_1.
- cs_1 is then used as the **new context node sequence** for $step_2$, and so on.

Multi-step XPath path

$$cs_0/step_1/step_2/\cdots$$
$$\equiv$$
$$(\underbrace{(cs_0/step_1)}_{cs_1}/step_2)/\cdots$$

XPath location steps

XPath step

Step syntax:

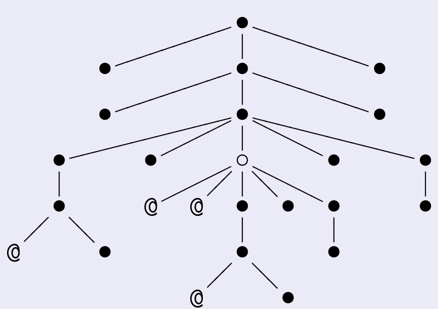
$$ax::nt\,[p_1]\cdots[p_n]$$

- A **step** (or **location step**) $step_i$ specifies
  1. the **axis** ax, *i.e.*, the direction of navigation taken from the context nodes,
  2. a **node test** nt, which can be used to navigate to nodes of certain kind (*e.g.*, only attribute nodes) or name,
  3. optional **predicates** p_i which further filter the sequence of nodes we navigated to.

XPath axes

XPath defines a family of **12 axes** allowing for flexible navigation within the node hierarchy of an XML tree.

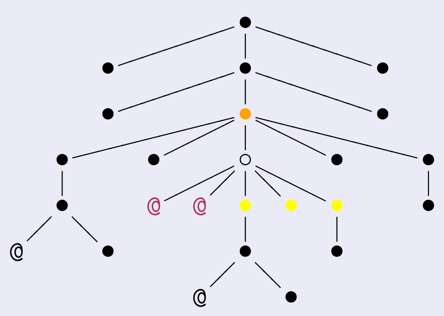
XPath axes semantics, ∘ marks the context node



- @ marks attribute nodes, • represents any other node kind (inner • nodes are element nodes).

XPath axes: `child`, `parent`, `attribute`

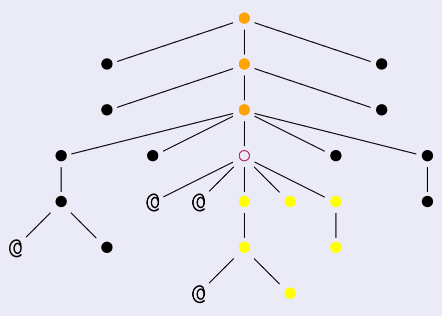
XPath axes: `child`, `parent`, `attribute`



- **Note:** the `child` axis does *not* navigate to the attribute nodes below ∘. The only way to access attributes is to use the `attribute` axis explicitly.

XPath axes: descendant, ancestor, self

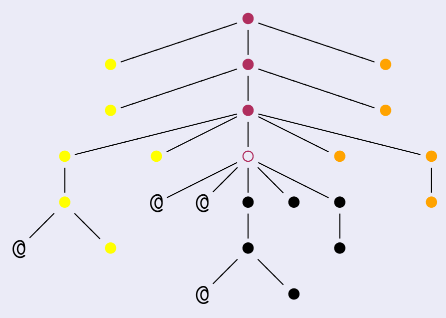
XPath axes: descendant, ancestor, self



- In a sense, descendant and ancestor represent the transitive closures of child and parent, respectively.

XPath axes: preceding, following, ancestor-or-self

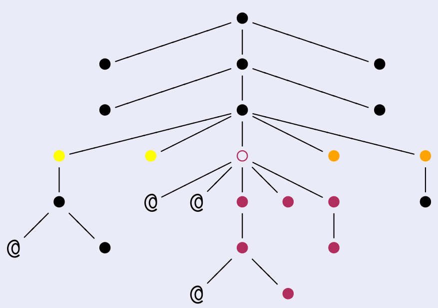
XPath axes: preceding, following, ancestor-or-self



- **Note:** In the serialized XML document, nodes in the `preceding` (`following`) axis appear completely before (after) the context node.

XPath axes: preceding-sibling, following-sibling, descendant-or-self

XPath axes: preceding-sibling, following-sibling, descendant-or-self



XPath axes: Examples (1)

In these first examples, there is a single initial context node, *i.e.*, a context node sequence of length 1: the root element `a`.

- Here, we set the node test *nt* to simply `node()` which means to *not* filter any nodes selected by the axis.

XPath example

```
(<a b="0">
   <c d="1"><e>f</e></c>
   <g><h/></g>
 </a>)/child::node()
```

$\Rightarrow$

```
(<c d="1"><e>f</e></c>,
 <g><h/></g>)
```

XPath axes: Examples (2)

XPath example

```
(<a b="0">
   <c d="1"><e>f</e></c>
   <g><h/></g>
 </a>)/attribute::node()     ⇒     attribute b { "0" }
```

XPath example

```
                                   (<c d="1"><e>f</e></c>,
(<a b="0">                          <e>f</e>,
   <c d="1"><e>f</e></c>            text { "f" },
   <g><h/></g>               ⇒      <g><h/></g>,
 </a>)/descendant::node()           <h/>
                                   )
```

XPath axes: Examples (3)

XPath example

```
(<a b="0">
   <c d="1"><e>f</e></c>
   <g><h/></g>
 </a>)/child::node()/child::node()
```

$\Rightarrow$

```
(<e>f</e>,
 <h/>
)
```

Notes:

- If an extracted node has no suitable XML representation by itself, XQuery serializes the result using the XQuery node constructor syntax, *e.g.*,

  `attribute b { "0" }` or `text { "f" }` .

- Nodes are serialized showing their content. This does *not* imply that all of the content nodes have been selected by the XPath expression!

XPath results: Order & duplicates

XPath Semantics

The result node sequence of any XPath navigation is returned in **document order** with **no duplicate nodes** (remember: node identity).

Examples:

Duplicate nodes are removed in XPath results ...

```
(<a b="0">
   <c d="1"><e>f</e></c>
   <g><h/></g>
 </a>)/child::node()/parent::node()
```

⇒

```
<a>
  ...
</a>
```

```
(<a><b/><c/><d/>
 </a>)/child::node()/following-sibling::node()
```

⇒

```
(<c/>,
 <d/>
)
```

XPath: Results in document order

XPath: context node sequence of length > 1

```
(<a><b/><c/></a>,
 <d><e/><f/></d>)/child::node()   ⇒   (<b/>,<c/>,<e/>,<f/>)
```

Note:

- The XPath document order semantics require `<b/>` to occur before `<c/>` and `<e/>` to occur before `<f/>`.
  - The result `(<e/>,<f/>,<b/>,<c/>)` would have been OK as well.
  - In contrast, the result `(<b/>,<e/>,<c/>,<f/>)` is *inconsistent* with respect to the order of nodes from *separate* trees!

XPath: Node test

Once an XPath step arrives at a sequence of nodes, we may apply a **node test** to filter nodes based on **kind** and **name**.

XPath node test

Kind Test	**Semantics**
`node()`	let any node pass
`text()`	preserve text nodes only
`comment()`	preserve comment nodes only
`processing-instruction()`	preserve processing instructions
`processing-instruction(`p`)`	preserve processing instructions of the form <?p . . . ?>
`document-node()`	preserve the (invisible) document root node

XPath: Name test

A node test may also be a **name test**, preserving only those element or attribute nodes with matching names.

XPath name test

Name Test	**Semantics**
name	preserve element nodes with tag *name* only (for `attribute` axis: preserve attributes)
*	preserve element nodes with arbitrary tag names (for `attribute` axis: preserve attributes)

Note: In general we will have $cs/ax\texttt{::*} \subseteq cs/ax\texttt{::node()}$.

XPath: Node test example

✎ Retrieve all attributes named `id` from this XML tree:

```
<a id="0">
  <b><c id="1"><d id="2"/></c>
     <c id="3"/>
  </b>
  <e di="X" id="4">f</e>
</a>
```

A solution

XPath: Node test example

Collect and concatenate all text nodes of a tree

```
string-join(<a><b>A<c>B</c></b>
                  <d>C</d>
               </a>/descendant-or-self::node()/child::text()
              , "")
```

- The XQuery **builtin function** `string-join` has signature

```
string-join(string*, string) as string .
```

Equivalent: compute the *string value* of node a

```
string(<a><b>A<c>B</c></b>
          <d>C</d>                ⇒   "ABC"
       </a>)
```

XPath: Ensuring order is not for free

The strict XPath requirement to construct a result in document order may imply **sorting effort** depending on the actual XPath implementation strategy used by the processor.

```
(<x>
  <x><y id="0"/></x>
  <y id="1"/>                          ⇒   (<y id="0"/>,
 </x>)/descendant-or-self::x/child::y        <y id="1"/>)
```

- In many implementations, the `descendant-or-self::x` step will yield the context node sequence (`<x>···</x>`,`<x>···</x>`) for the `child::y` step.
- Such implementations thus will typically extract `<y id="1"/>` before `<y id="0"/>` from the input document.

XPath: Predicates

The optional third component of a step formulates a list of **predicates** $[p_1] \cdots [p_n]$ against the nodes selected by an axis.

XPath predicate evaluation

Predicates have higher precedence than the XPath step operator /, *i.e.:*

$$cs/step[p_1][p_2] \quad \equiv \quad cs/((step[p_1])[p_2])$$

The p_i are evaluated left-to-right for each node in turn. In p_i, the **current context node**[23] is available as '.'.

[23] *Context item*, actually: predicates may be applied to sequences of arbitrary items.

XPath: Predicates

An XPath predicate p_i may be *any* XQuery expression evaluating to some value v. To finally evaluate the predicate, XQuery computes the **effective Boolean value** $ebv(v)$.

Effective Boolean value

Value v[24]	$ebv(v)$
`()`	`false()`
`0`, `NaN`	`false()`
`""`	`false()`
`false()`	`false()`
x	`true()`
$(x_1, x_2, \ldots, x_n)$	`true()`

[24] Item $x \notin \{$`0`, `""`, `NaN`, `false()`$\}$, items x_i arbitrary. Builtin function `boolean(item*) as boolean` also computes the effective Boolean value.

XPath: Predicate example

Select all elements with an id attribute

```
(<a id="0">
   <b><c id="1"/></b>
   <b><c><b/></c></b>
   <d id="2">e</d>
 </a>)/descendant-or-self::*[./attribute::id]
```

⇒

```
(<a id="0">
   ...
 </a>,
 <c id="1"/>,
 <d id="2">e</d>
)
```

Select all elements with a b grandchild element

```
(<a id="0">
   <b><c id="1"/></b>
   <b><c><b/></c></b>
   <d id="2">e</d>
 </a>)/descendant-or-self::*[./child::*/child::b]
```

⇒

```
<b>
  <c><b/></c>
</b>
```

- **Note: Existential semantics** of path predicates.

XPath: Predicate example

✎ How to select all leaf elements of a tree?

You may use the builtin function `not (item*) as boolean` which computes the inverted effective Boolean value, *i.e.*,

$$\texttt{not}(v) \equiv \neg\,\texttt{boolean}(v).$$

A solution

XPath: Predicates and atomization

In XQuery, if any item x—atomic value or node—is used in a context where a value is required, **atomization** is applied to convert x into an atomic value.

- Nodes in value contexts commonly appear in XPath predicates. Consider:

Value comparison in a predicate

```
(<a>
   <b>42</b>
   <c><d>42</d></c>
   <e>43</e>
</a>)/descendant-or-self::*[. eq 42]
```

$\Rightarrow$

```
(<b>42</b>,
 <c><d>42</d></c>,
 <d>42</d>
)
```

Atomization

Atomization

Atomization turns a sequence $(x_1, \ldots, x_n)$ of items into a sequence of atomic values $(v_1, \ldots, v_n)$:

1. If x_i is an atomic value, $v_i \equiv x_i$,
2. if x_i is a node, v_i is the *typed value*[25] of x_i.

- The XQuery builtin function `data(item*) as anyAtomicType*` may be used to perform atomization explicitly (rarely necessary).

[25] Remember: the *typed value* is equal to the *string value* if x_i has not been validated. In this case, v_i has type `untypedAtomic`.

XPath: Predicates and atomization

Atomization (and casting) made explicit

```
<a>
  <b>42</b>
  <c><d>42</d></c>
  <e>43</e>
</a>/descendant-or-self::*[data(.) cast as double
                                   eq
                                 42 cast as double]
```

- **Note:** the value comparison operator `eq` is witness to the value context in which '`.`' is used in this query.
- For the context item `<c><d>42</d></c>` (a non-validated node), `data(.)` returns `"42"` of type `untypedAtomic`.

Atomization and subtree traversals

Since atomization of nodes is pervasive in XQuery expression evaluation, *e.g.*, during evaluation of

- arithmetic and comparison expressions,
- function call and return,
- explicit sorting (`order by`),

efficient **subtree traversals** are of prime importance for any implementation of the language:

Applying `data()` to a node and its subtree:

```
data(<a>
       <b>fo</b>o<c>
       <d>b</d><e>ar</e>
     </c>
     </a>)
```

≡ data(tree)

XPath: Positional access

Inside a predicate `[`p`]` the current context item is '`.`'.

- An expression may also access the **position** of '`.`' in the context sequence via `position()`. The first item is located at position 1.
- Furthermore, the position of the **last** context item is available via `last()`.

Positional access

$$(x_1, x_2, \ldots, x_n)\texttt{[position() eq } i\texttt{]} \Rightarrow x_i$$
$$(x_1, x_2, \ldots, x_n)\texttt{[position() eq last()]} \Rightarrow x_n$$

A predicate of the form `[position() eq `i`]` with i being any XQuery expression of numeric type, may be abbreviated by `[`i`]`.

XPath: Positional access example

✎ Predicates [·] bind stronger than /

Given the XML tree below as context *cs*, what is the result of evaluating

```
(cs/descendant-or-self::node()/child::x)[2]
```

vs.

```
cs/descendant-or-self::node()/child::x[2]  ?
```

```
<a>
  <b><x id="1"/></b>
  <d><x id="2"/><x id="3"/></d>
</a>
```

Solution

XPath: Positional access example

Predicates are evaluated after step and node test

Given the XML tree below as context *cs*, what is the result of evaluating

cs`/descendant::*[2]`

vs.

cs`/descendant::x[2]` ?

```
<a>
  <b><x id="1"/></b>
  <d><x id="2"/><x id="3"/></d>
</a>
```

Solution

XPath: Predicate evaluation order

Remember: predicates are evaluated left to right

$$
\begin{array}{rcl}
\texttt{(1,2,3,4)[. gt 2][2]} & \equiv & \texttt{((1,2,3,4)[. gt 2])[2]} \\
 & \not\equiv & \texttt{(1,2,3,4)[2][. gt 2]}
\end{array}
$$

XPath: The context item '.'

As a useful generalization, XPath makes the **current context item** '.' available in each **step** (not only in predicates).

XPath steps (/) and the context item

In the expression

$$cs/e$$

expression e will be evaluated with '.' set to each item in the context sequence cs (in order). The resulting sequence is returned.[26][27]

[26] Remember: if e returns nodes (e has type `node*`), the resulting sequence is sorted in document order with duplicates removed.

[27] Compare this with the expression `map` $(\lambda\,.\,\rightarrow e)\ cs$ in functional programming languages.

XPath: Using the context item

Accessing '.'

```
(<a>1</a>,<b>2</b>,<c>3</c>)/(. + 42)  ⇒ (43.0,44.0,45.0)
(<a>1</a>,<b>2</b>,<c>3</c>)/name(.)    ⇒ ("a","b","c")
(<a>1</a>,<b>2</b>,<c>3</c>)/position() ⇒ (1,2,3)
(<a><b/></a>)/(./child::b, .)           ⇒ (<a><b/></a>,<b/>)
```

✎ Evaluate the following

1. *cs*`/descendant-or-self::node()/count(./descendant::node())`
2. *cs*`/descendant-or-self::node()/count(./ancestor::*)`

with *cs* ≡

```
<a>
  <b c="0"/>
  <d><e>f</e></d>
</a>
```

.

Combining node sequences

Node sequence combinations

Sequences of nodes (*e.g.*, the results of XPath location step) may be combined via

```
|   union
intersect
except   .
```
[28]

These operators **remove duplicate nodes** based on identity and return their result in **document order.**

- **Note:** Introduced in the XPath context because a number of useful navigation idioms are based on these operators.

---

[28] | and `union` are synonyms

Navigation idioms (1)

Selecting all x children and attributes of context node

```
cs/(./child::x | ./attribute::x)
```

Select all siblings of context node

```
cs/(./preceding-sibling::node() | ./following-sibling::node())
```

or

```
cs/(./parent::node()/child::node() except .)
```

Select context node + all its siblings

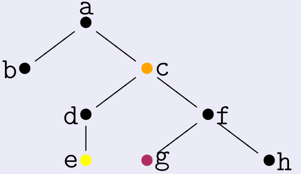
$$cs\texttt{/(./parent::node()/child::node()}\underbrace{\texttt{| .}}_{*}\texttt{)}$$

Why is (∗) required?

Navigation idioms (2)

First common ancestor (fca)

Compute the first common ancestor (fca) of two contexts, cs_0 and cs_1, in the same tree:



$(cs_0$`/ancestor::* intersect` cs_1`/ancestor::*)[last()]`

✎ What is going on here?

And: Will this work for non-singleton $cs_{0,1}$?

XPath: Simulate `intersect` and `except`

In earlier versions of XPath (1.0), the following expressions could simulate `intersect` and `except` of two node sequences $cs_{0,1}$:[29]

Simulate `intersect` and `except`

$$cs_0 \ \texttt{intersect} \ cs_1 \equiv$$
$$cs_0 \texttt{[count(.} \mid cs_1\texttt{) eq count(}cs_1\texttt{)]}$$

$$cs_0 \ \texttt{except} \ cs_1 \equiv$$
$$cs_0 \texttt{[count(.} \mid cs_1\texttt{) ne count(}cs_1\texttt{)]}$$

[29] XQuery builtin operators `eq` and `ne` compare two single items for equality and inequality, respectively.

XPath: Abbreviations

Since XPath expressions are pervasive in XQuery, query authors commonly use the succinct **abbreviated XPath syntax** to specify location steps.

Abbreviated XPath syntax

Abbrevation	Expansion
nt	`child::`*nt*
`@`	`attribute::`
`..`	`parent::node()`
`//`	`/descendant-or-self::node()/`
`/`[30]	`root(.)`
step[30]	`./`*step*

[30] At the beginning of a path expression.

XPath: Abbreviations

XPath abbrevation examples

Abbreviation	Expansion
`a/b/c`	`./child::a/child::b/child::c`
`a//@id`	`./child::a/descendant-or-self::node()/attribute::id`
`//a`	`root(.)/descendant-or-self::node()/child::a`
`a/text()`	`./child::a/child::text()`

✎ XPath abbreviation quiz

What is the expansion (and semantics) of

`a/(*|@*)` and `a[*]` ?

XPath: Abbreviations

NB: Use of these abbreviations may lead to confusion and surprises!

Abbrevations + predicates = confusion

$$cs\texttt{//c[1]} \quad \not\equiv \quad cs\texttt{/descendant-or-self::c[1]}$$

Evaluate both path expressions against

cs =
```
<a>
  <b><c id="0"/><c id="1"/></b>
  <d><c id="2"/></d>
  <c id="3"/>
</a>
```

More XPath weirdness

cs/(/)//(/)　　`parent::text()`　　`attribute::comment()`