XML Technologies

XPath, Namespaces

Christian Grün
Database & Information Systems Group
Universität Konstanz

Querying XML

- one big advantage of XML is that it brings along its own *query languages*
- querying XML involves...
  - *navigating* to nodes in a document, or a collection of documents
  - testing *properties* of these nodes
  - *filtering* nodes by certain conditions
  - *passing on* results to other operations

Today...

- ...we'll start off with **XPath**
- currently, three versions are available: 1.0, 2.0, and (surprise) 3.0
- we will focus on XPath 1.0 (contains most important concepts)

XPath

- XPath is a *declarative* language (✎ can someone explain this term?)
- it is *tree-aware*: provides constructs for traversing to children and parents
- it serves as a *subset* for many other X languages, including…
  - **XQuery**: *the* XML query language
  - **XSLT**: a language for *transforming* XML into other formats
  - **XML Schema**: for *validating* the document structure
  - **Schematron**: for defining more complex *validation rules*
  - **XForms**: for building client-side XML applications
- *syntactic sugar* makes querying sufficiently sweet (or cryptic):

```
$v/e[(*|@*)!='']
```

Data Types

- XPath 1.0 only supports some basic data types: **strings**, **numbers**, **booleans**, and **nodes**:
  - strings are enclosed in *simple* or *double* quotes: `''`, `"Guðmundsdóttir"`, ...
  - numbers may be integers (`1`, `2`, ...) or floating-point (`1.5`, `NaN`, `3.1415926535897932`
  - booleans can be generated via the functions `false()` and `true()`
- XPath 1.0 has no means to *create* new XML nodes
- instead, data is passed on to the XPath processor and will be bound to the *initial context item* (`.`) or to *variables* (`$a`, `$園`, ...)
- input can also be supplied by other *languages* using XPath (such as XQuery):

```
doc('addressbook.xml')//address[name = 'John']
<line id='0'>Hello</line>/@id
```

Path Traversal

A formal introduction:

- a path traversal starts from a sequence of *context nodes* (the *context set*)
- it provides a concise navigation syntax: cs_0/step
- cs denotes the context node sequence, from which a navigation in direction step is taken
- XPath expressions may consist of *multiple steps* $step_i$, $i \geq 1$
- $step_1$ starts from cs_0 and yields a sequence of new nodes cs_1
- cs_1 is then used as the *new context node sequence* for $step_2$, and so on
- the general syntax is as follows:
  $cs_0/step_1/step_2/... \equiv ((cs_0/step_1)/step_2)/...$

Location Steps

- this is the complete syntax of a single location step:
  `axis::test[predicate₁]...[predicateₙ]`
  - `axis` represents the *direction of navigation* taken from the context nodes. Examples: `child`, `parent`
  - `test` filters the navigated nodes by their node kind (text, comment, element, etc.) or name. Examples: `text()`, `address`
  - `predicate` is an arbitrary XPath expression, which further filters the sequence of nodes we navigated to. Examples: `[name = 'John']`, `[1]`
- syntactically, a single step is already a valid XPath expression
- this may lead to surprising error messages:
  `hello` (≡ `child::hello`) ⇨ *No context item defined!*

Axes

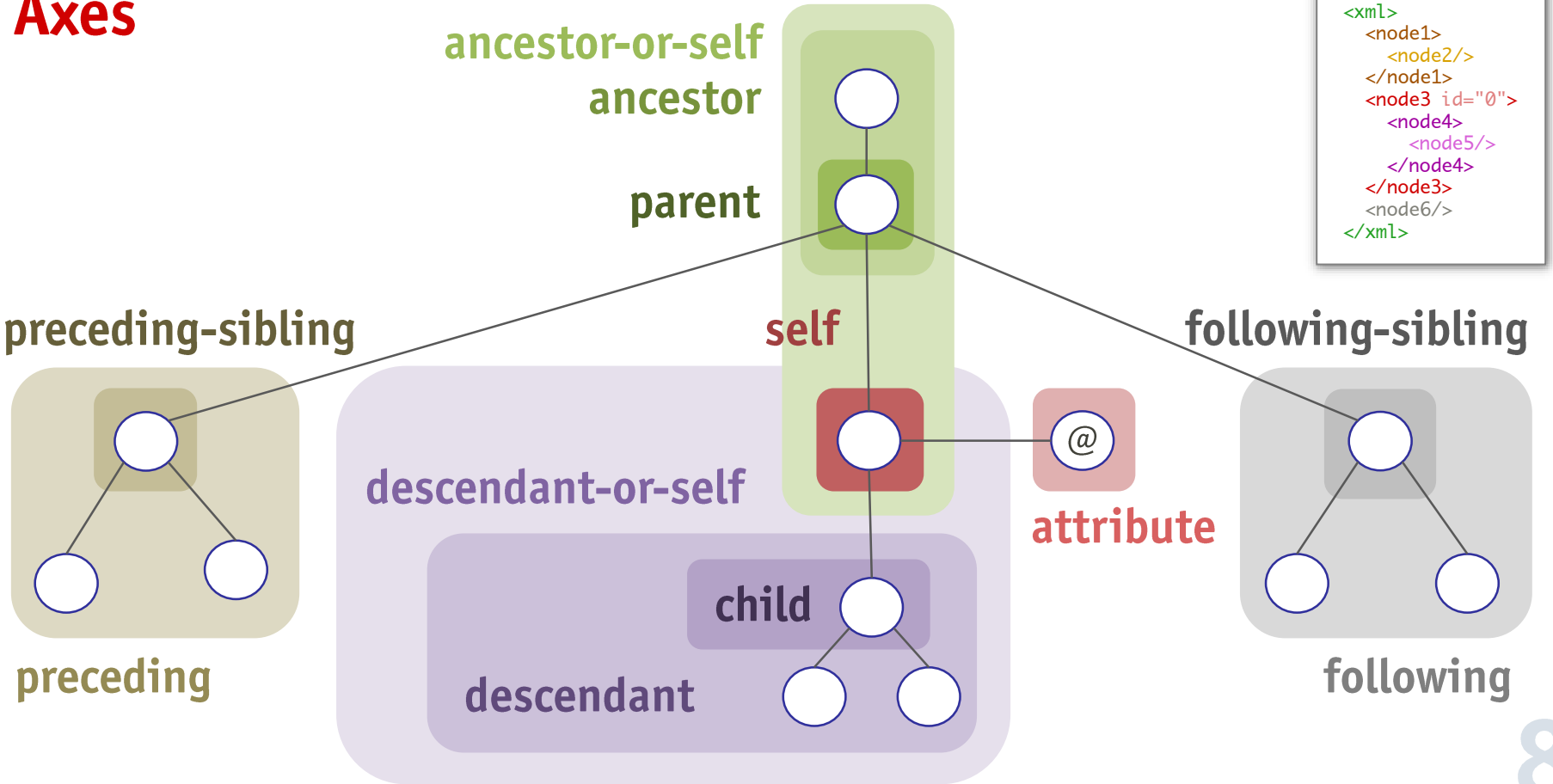
- XPath defines a family of 13 axes (i.e., directions), allowing for a flexible navigation within the node hierarchy of an XML tree
- axes can be divided into *forward* and *reverse* axes
- we will ignore the *namespace* axis (discarded in XPath 2.0 and later)

Examples

- the XML document shown on the right will serve as input for some upcoming queries

```
<xml>
  <node1>
    <node2/>
  </node1>
  <node3 id="0">
    <node4>
      <node5/>
    </node4>
  </node3>
  <node6/>
</xml>
```

Axes




```
<xml>
  <node1>
    <node2/>
  </node1>
  <node3 id="0">
    <node4>
      <node5/>
    </node4>
  </node3>
  <node6/>
</xml>
```

Axes: Examples

- in the queries below, * is used as node test; it accepts all elements
- node3 will be our initial context node

What are the resulting nodes?

a) `child::*`
b) `descendant::*`
c) `descendant::*/child::*`
d) `parent::*`
e) `parent::*/child::*`
f) `child::*/child::*/child::*`
g) `following::*/preceding::*`

```
<xml>
  <node1>
    <node2/>
  </node1>
  <node3 id="0">
    <node4>
      <node5/>
    </node4>
  </node3>
  <node6/>
</xml>
```

Axes: Observations

- take care: in many cases, the *inverse* XPath axis does not yield the original node set
- once a result set gets *empty* while traversing the tree, all subsequent result sets will be empty
- if nodes are *serialized*, all its descendants and attributes will be output
- this does *not* imply that all of the nodes have been *selected* by an XPath expression!
- example: `descendant::*` (starting from node3) will be serialized as `<node4><node5/></node4><node5/>`

```
<xml>
  <node1>
    <node2/>
  </node1>
  <node3 id="0">
    <node4>
      <node5/>
    </node4>
  </node3>
  <node6/>
</xml>
```

Node Tests

- the node test is applied to all nodes of the intermediary axis result set
- nodes are *filtered*, based on their node kind and (if applicable) name

Node tests

`node()`	lets all nodes pass
`element()`	preserves elements
`attribute()`	chooses attributes
`text()`	restricts results to text nodes
`comment()`	selects comment nodes
`document-node()`	accepts document nodes
`processing-instruction()`	returns processing instructions

Node Tests

- if a node test is a *name test*, only element or attribute nodes with the given name will be accepted
- name tests are only defined for *elements* and *attributes*

Name tests

- `name` returns nodes matching *name*
- `*` wildcard: returns all nodes

Example

✎ Formulate an XPath expression that returns all *id* attributes from an XML input like the following one: `<A id="0"><B id="1"/></A>`

Abbreviations

- various shortcut notations exist to keep your queries short
- the most important ones are listed below:

Shortcut	Full Writing	Examples
`test`	child::*test*	`node()`, ปาฏิหาริย์, `*`
`@test`	attribute::*test*	`@id`, `@*`
`.`	self::node()	`.`
`..`	parent::node()	`..`
`//`	/descendant-or-self::node()/	`//text()`, `//what//ever`

- the following notations are allowed at the beginning of an expression:

`step`	./*step*	`navigate/from/context`
`/`	root(.)/	`/navigate/from/root`

Path Semantics

Document Order

- order is essential in XML, because documents may contain *full-texts*
- this is different in JSON: maps may be *arbitrarily swapped*
- the *document order* is the natural order in which a document is parsed
- if different documents are processed by a single query (which may easily happen with XQuery), the order is *implementation defined*

XPath steps may yield results in reverse document order or create duplicates...

In the resulting node set:

a) nodes are returned in *document order*, and

b) *duplicate nodes* are removed

Examples

- What are the results of the following queries?

1. `<root><a/><a/></root>/child::a/parent::root`
   a) `<root/><a/><a/><root>`
   b) `<root/><a/><a/><root>`, `<root/><a/><a/><root>`
2. `<root><a/><b/><c/></root>/child::c/preceding::*`
   a) `<a/><b/>` (note: `preceding` is a *reverse* axis)
   b) `<b/><a/>`
3.
```
<root>
  <a>text</a>
  <b/>
</root>/descendant-or-self::*/child::node()
```
   a) `<a>text</a>`, `<b/>`, `text`
   b) `<a>text</a>`, `text`, `<b/>`

Path Semantics

Some thoughts on performance:

- algorithmic complexity for sorting: $O(n \log n)$
- recovery of document order can be tricky (✎ Why? think about DOM...)
- nodes must be *cached* in order to be sorted

Ways out:

- a clever implementation will *try to avoid* sorting whenever possible
- document order is preserved by many simple path traversals; examples:
  `/simple/child/steps`
  `/descendant::step`

✎ What about `//descendant::step/followed-by-child-step`?

Predicates

- the third (optional) part of a step are *predicates* `[p1]...[pn]`, which further restrict the number of results
- before evaluating the predicates, the current node will be **bound** to the context item (accessible via '`.`')
- a predicate may be *any* XPath expression, comprising location steps, comparisons, numbers, etc.
- predicates may be evaluated as **numbers** or **booleans**:
  - the *boolean value* of the predicate expression will decide if the current context node will be added to the final result set
  - if the result is a number, it will be compared with the *context position* (see later)

Predicates

Boolean value: boolean(*v*)

Data type	Expressions *e*	boolean(*e*)
booleans	`false()`; `true()`	false; true
strings	`""`; `"नमस्ते"`	false; true
nodes	`()`; `<a/>`; `(<a/>,<b/>)`	false; true; true

Examples

- `<XML><A/><B>HI!<B/></XML>/*[text()]` yields `<B>HI!</B>`

What are the results of the following query? (...think first ;)

```
<A>dog</A>/.[text()]
<A>dog</A>/text()[.]
<A>dog</A>/text()['cat']
```

Comparisons

- usually, predicates contain more complex expression than *primitive values or location steps*
- XPath provides six operators to compare values:
  `<`, `<=`, `=`, `>=`, `>`, `!=`
- boolean operations are possible via two boolean operators and additional functions:
  `and`, `or`, `not()`, `boolean()`, `true()`, `false()`
- operators and functions are *fully composable* (✎ What is that?)

Example

- `<HIT>X</HIT>/.[text() and text()='X' and not(@*)]`

More examples

- select all elements which have attributes:
  `<a><b><HIT id="2"/></b></a>/descendant-or-self::*[@*]`
- select all leaf elements:
  `<a><b><HIT/></b></a>/descendant-or-self::*[not(node())]`
- select all elements which have “grandparents”:
  `<a><b><HIT/></b></a>//*[parent::*/parent::*]`

Predicates are not only allowed after location paths (in the examples, I have adopted the *comma syntax* from XQuery):

- pass on a string if it is non-empty: `""[.]` vs. `"non-empty"[.]`
- find substrings: `("life","death")[contains(.,'e')]`
- do comparisons: `("A","B","C")[. != "A"]`

Atomization

- *atomization* is applied to an item whenever its *atomic value* is required
- if a value is already atomic (i.e., a boolean, string or number), nothing needs to be done
- if a value is a *node*, its atomized value will be retrieved by:
  - *concatenating* the strings of all descendant text nodes (elements, documents), or
  - returning its *value* (attributes, processing instructions, comments, texts)
- the `data()` function can be used to enforce atomization

✎ What results do you expect?

- `'BIG' = 'BIG'`
- `<a><c>B</c>IG</a> = 'BIG'`
- `<a><c>BIG</c><e>BIG</e></a> = 'BIG'`

Positional Predicates

- predicates are *matched against* all context nodes of a location step
- the *context position* references the currently processed node (counting starts with 1)
- if the predicate yields a single number, it will be *compared with* the context position (resulting in *true* or *false*)
- the context position can be explicitly requested via `position()`:
  $(x_1, x_2, \ldots, x_n)[\texttt{position()} = i] \Rightarrow x_i$
- if *i* evaluates to a number, the following two expressions are equivalent:
  $e[\texttt{position()} = i] \equiv e[i]$
- the position of the last context item is available via `last()`:
  $(x_1, x_2, \ldots, x_n)[\texttt{last()}] \Rightarrow x_n$

Evaluation of Predicates

- important: predicates bind stronger than slashes!

✎ Can you see why the following queries return different results?

`descendant::*/x[2]` vs.
`(descendant::*/x)[2]`

Context node:

```
<a>
  <a><x a="1"/></a>
  <a><x a="2"/><x a="3"/></a>
</a>
```

- the abbreviation `//` is even more deceptive, as it *hides* the first location step
- the following rewritings are valid...

```
  //x
≡ /descendant-or-self::node()/x
≡ /descendant::x
```

...provided that no positional predicate is used:

`//x[i]` ≢ `/descendant::x[i]` !

Evaluation of Predicates

- important: predicates are evaluated from *left to right*!
- a node is *ignored* by the query processor as soon as it has been filtered out by a predicate

✎ Which results do you expect?

`("A","B","C")[1][2]` vs. `("A","B","C")[2][1]`

- the order of predicates *does not matter* if none of the predicates yields a number
- a query optimizer may decide to evaluate *cheaper* predicates first

Example

- `$input[`*expensive*`][text()]`

⇨ text node check can be evaluated first if *expensive* is known not to yield a number

Combining Node Sets

- node sets (e.g., the results of XPath location steps) may be combined via `union` (shortcut: `|`), `intersect` and `except`.
- well, to be honest: in pure XPath 1.0, only `|` exists
- equal to location paths, *duplicates* will be removed and return results will be returned in *document order*

Examples

- child elements and attributes: `*|@*`
- descendant elements, excluding children: `.//* except *`
- context node and all its siblings:
  `. | preceding-sibling::node() | following-sibling::node()`

XPath: Conclusion

- XPath is a powerful and intuitive language due to its concise syntax
- it is ideally embeddable in larger language constructs

XPath 2.0

...introduces extensions that are heavily used in XQuery:

- *loops*: `for $a in ... return ...`; *conditional* and *quantified* expressions
- strict *type system* (not fully compatible with XPath 1.0)
- provides appr. 100 functions in the function namespace (`fn:`)

XPath 3.0

- higher order features (look forward to Leo's talk!)

Namespaces: Introduction

- a namespace is a *container* for a *set of identifiers*
- used for *encapsulating* related data structures, identifiers, symbols, etc.
- needs to be specified when addressing a resource

Examples

- Java uses *packages* to group classes:
  `package de.unikn.inf.dbis.xml12.Example;`
- C# has similar semantics, *packages* have been renamed to *namespaces*:
  `using de.unikn.inf.DBIS.Xml12.Example;`
- some APIs also treat file paths as namespaces:
  `/home/user/data`

XML Namespaces

- in many use cases, XML from different sources are wrapped together
- XPath expressions get *ambiguous* if the same element names are used

Example

- the XML depicted on the right contains two name elements
- the location path //name yields results from different contexts

```
<xml>
  <shop>
    <item type="book">
      <name>Iceland Unleashed</name>
    </item>
  </shop>
  <addressbook>
    <address type="client">
      <name>John Hanson</name>
    </address>
  </addressbook>
</xml>
```

Solution

- introduce namespaces for shop and addressbook elements

Definition

- a namespace is identified by a URI reference
- a URI is a *string of characters*, which is used to *identify* a resource (✎ What is a URL?)

```
<xml xmlns:shop="http://wow.com/shop"
     xmlns:addr="http://wow.com/addr">
  <shop:shop>
    <shop:item type="book">
      <shop:name>Iceland Unleashed</addr:name>
    </shop:item>
  </shop:shop>
  <addr:addressbook>
    <addr:address type="client">
      <addr:name>John Hanson</addr:name>
    </addr:address>
  </addr:addressbook>
</xml>
```

Namespace declaration

- declarations look like attributes
  ⇨ backward compatibility
- a declaration consists of `xmlns:` and a *namespace prefix* (`shop`, `addr`) and the namespace URI
- namespace is assigned by adding the prefix to *elements* and *attributes*

Default namespace

- a default namespace is specified *without prefix*
- all *descendant-or-self* elements without prefix belong to it
- namespaces can be *overwritten* with another default namespace
- attributes without namespace prefix are in *no* namespace
- with XML 1.1, namespaces can be *undeclared*;
  if an empty URI is specified, the prefix will no longer be in scope:
  `<prefix:name xmlns:prefix=""/>`

```
<xml>
  <shop xmlns="http://wow.com/shop">
    <item type="book">
      <name>Iceland Unleashed</name>
    </item>
  </shop>
  <addressbook xmlns="http://wow.com/addr">
    <address type="client">
      <name>John Hanson</name>
    </address>
  </addressbook>
</xml>
```

Querying (spoiler alert: includes XQuery)

- namespaces can be declared in the *prolog* (header) of a query:
```
declare namespace ns1 = 'namespace-uri-1';
declare namespace ns2 = 'namespace-uri-2';
//ns1:name, //ns2:name
```
- a default element namespace can be declared for location paths and new elements:
```
declare default element namespace 'namespace-uri';
//name, <adopting-default-namespace/>
```
- namespaces can be directly embedded in paths:
```
//Q{namespace-uri}name
```
- wildcards can be used to accept all namespaces:
```
//*:name
```

Take care...

- prefixes are not globally valid, as namespaces can be locally overwritten
- an element is referenced by its *namespace URI*, not its *prefix*!
- this often leads to surprises if a prefix is overwritten, wrongly declared, or not declared at all

Check yourself...

✎ Which are the resulting nodes?

- `//a:*` (if prefix a is bound to "A")
- `//b:*` (if prefix b is bound to "A")
- `//c:*` (if prefix c is bound to "B")

```
<confuzzlement>
  <a:x xmlns:a="A">
    <b:y xmlns:b="A"/>
    <a:z xmlns:a="B"/>
  </a>
</confuzzlement>
```