Net-Based Applications

Chapter 8: XPath

Holger Schwarz
Universität Stuttgart

Winter Term 2016/2017

Overview

- Motivation and introduction
- Data model
- Atomic values and simple expressions
- Path Expressions, node tests and predicates
- Examples
- Functions

Motivation

Xpath provides a common syntax and semantics to address a part of an XML document

- Name comes from use of a path notation as in URLs navigating through the hierarchical structure of an XML document
- Operates on the abstract, logical structure of an XML document
- Allows the selection of nodes or atomic values from the document tree
- XPath uses
  - a compact, path-based syntax
  - no XML element-based syntax
- Usage:
  - XSL transformation (like XSLT)
  - XML query languages (e. g. XQuery, see next chapter)
  - XPointer – an addition to URIs

XML Standardization

- Standard documents:
  - XPath 1.0: W3C Recommendation. November 16, 1999
    - Latest version: http://www.w3.org/TR/xpath
    - Standard is also a good reference!
  - XPath 2.0: W3C Recommendation, January 23, 2007
    - Latest version: http://www.w3.org/TR/xpath20
    - Relies on
      XQuery 1.0 and XPath 2.0 Data Model (XDM)
      XQuery 1.0 and XPath 2.0 Functions and Operators
  - XPath 3.0: W3C Recommendation, April 8, 2014
    - XQuery and XPath Data Model 3.0
    - XPath and XQuery Functions and Operators 3.0
    - XML Path Language (XPath) 3.0

Example

```
<lecture>
 <Chapter1>
  XML-Daten...
 </Chapter1>
 <Chapter2>
  DTDs...
 </Chapter2>
 <Chapter3>
  <Chapter3a>
   Syntax...
  </Chapter3a>
 </Chapter3>
</lecture>
```

- XPath-Expressions:
  - /lecture/Chapter1
  - /lecture/Chapter3/Chapter3a
- similar to file systems, but
  - addresses node sets (rather than a single file or directory)
  - handles attributes in the XML document
  - has functions and predicates
  - expressions can have different result types
  - ...
- two syntaxes:
  - verbose syntax (easier to understand)
  - abbreviated syntax (easier to write)

Overview

- Motivation and introduction
- Data model
- Atomic values and simple expressions
- Path Expressions, node tests and predicates
- Examples
- Functions

Sequences

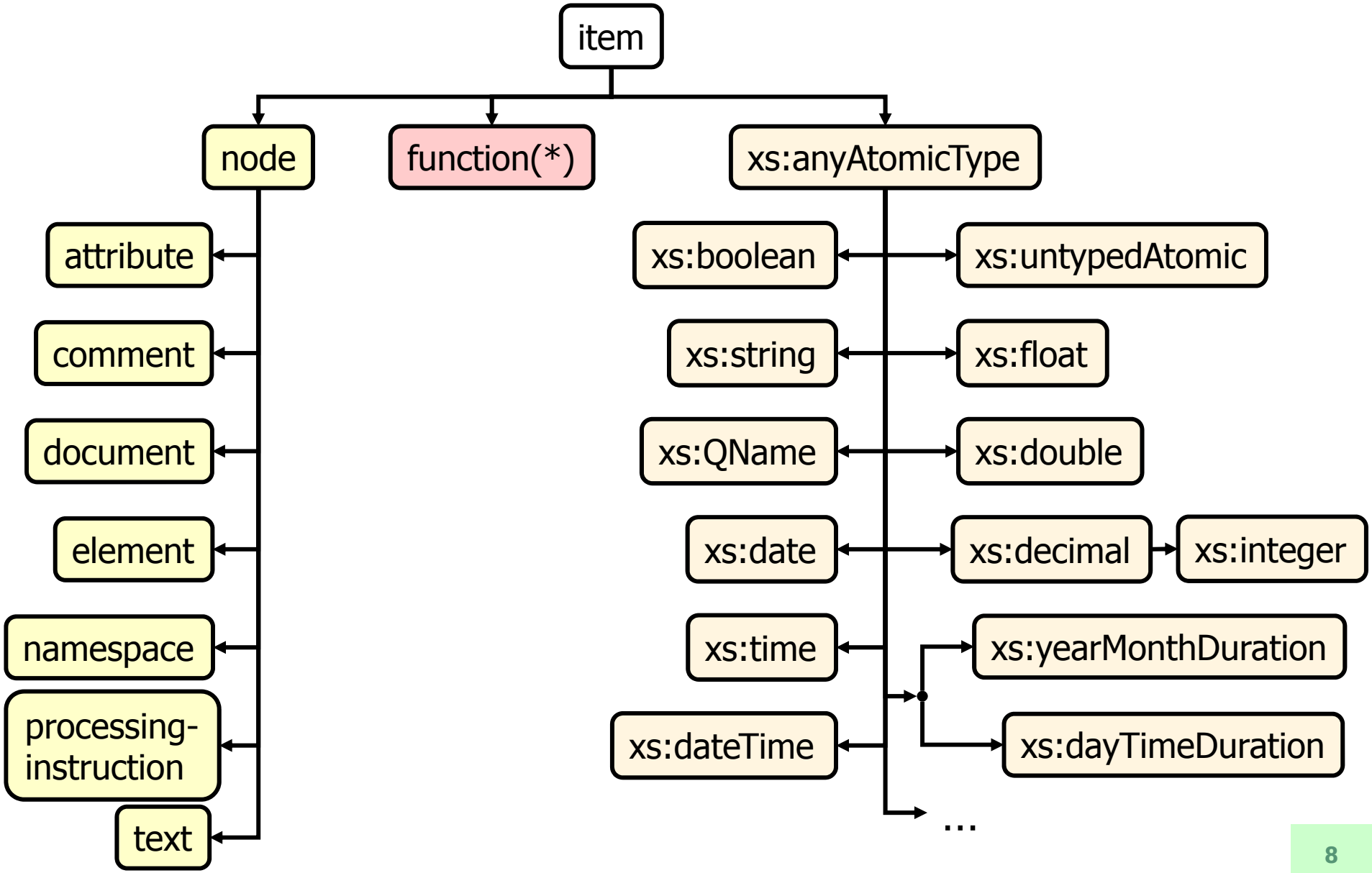
A sequence is an ordered collection of zero or more items

- Every instance of the data model is a sequence
- A sequence containing only one item is identical to the item
- A sequence cannot be a member of a sequence, i.e., sequences are always flattened

equivalent sequences	
(1, 2, 3, 4)	(1, 2, (3, 4))
(1, (), <B/>)	(1, <B/>)
(<A/>)	<A/>
()	(((), ())

different sequences	
(1, 2, 3, <A/>)	(1, 2, 3, 2, <A/>)
(1, 2, 3, 4)	("1", "2", "3", "4")
(1, 2)	(2, 1)

Excerpt from Type Hierarchy



Nodes and Tries

Every node is one of the seven kinds of nodes : document, element, attribute, text, namespace, processing instruction, comment

- Nodes contain (depending on their kind): name, typed-value, string-value, attributes, children, …
- Each node has its own identity
- Document nodes may contain more than one element child

Nodes with their parents and children build a tree with the root node being the topmost node of a tree

- The data model can handle sequences of trees
- A tree whose root node is a Document Node is referred to as a document
- A tree whose root node is not a Document Node is referred to as a fragment

Document Order

- Total order of nodes in a tree according to the following rules:
  1. The root node is the first node.
  2. Every node occurs before all of its children and descendants.
  3. Namespace Nodes immediately follow the Element Node with which they are associated. The relative order of Namespace Nodes is stable but implementation-dependent.
  4. Attribute Nodes immediately follow the Namespace Nodes of the element with which they are associated. If there are no Namespace Nodes associated with a given element, then the Attribute Nodes associated with that element immediately follow the element. The relative order of Attribute Nodes is stable but implementation-dependent.
  5. The relative order of siblings is the order in which they occur in the children property of their parent node.
  6. Children and descendants occur before following siblings.

Overview

- Motivation and introduction
- Data model
- Atomic values and simple expressions
- Path Expressions, node tests and predicates
- Examples
- Functions

Atomic Values and Simple Expressions

- constants and literals:

  Strings: 'Hotel Neptun', "12.34", "Minibar"
  Integers: 123, -24, 0, +7
  Decimals: -24.0, 123.45, +.23
  Doubles: -123.5e3, 200e6

- Constructors for all atomic types :

  xs:date(„2009-01-27"), xs:integer("12345"), xs:float(1), …

- casting:

  "0815" cast as xs:integer

- arithmetic: +, -, *, DIV, MOD
  (only on nodes and sequences with length 1)

  ($price – 10) DIV 100

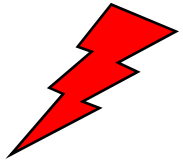
- logical expressions: AND, OR, function not()

Value Comparisons

- Compare atomic values of different types:
  EQ (equal), NE (not equal), LT (less than), LE (less or equal), GT (greater than), GE (greater or equal)
- Examples:

```
2 gt 1        => true        // compared as xs:integer
2 gt 1.0      => true        // compared as xs:decimal
2 eq 1E0      => false       // compared as xs:double

2 eq "1"                     // incompatible types
2 eq (2, 2)                  // (2, 2) is no atomic value
() eq ()                     // missing atomic value in empty sequence
```



General Comparison

- Compare sequences of values:
  =, !=, <, <=, >, >=
- Expression evaluates to true, if
  - there exists an element a in the first operand and an element b in the second operand, and
  - the comparison between a and b evaluates to true
- Examples:

```
(1, 2, 3) < (4, 5, 6)        => true
(1, 2, 3) > (4, 5, 6)        => false
(1, 2, 3) = (4, 5, 6)        => false
(1, 2, 3) != (1, 2, 3)       => true
```

Node Comparison

- Compare single nodes:
  is: true if the two nodes are the same node (by identity)
  <<: true if the left node appears before the right node in document order
  >>: vice versa
- Examples:
  let $a := <x><y/></x>

```
<x/> is <x/>          => false     // not the same node
$a/y  is  $a/y        => true      // the same node
$a  <<  $a/y          => true
$a  >>  $a/y          => false

<text>happy</text> = <text>happy</text>      => true
<text>happy</text> is <text>happy</text>     => false
```

Overview

- Motivation and introduction
- Data model
- Atomic values and simple expressions
- Path Expressions, node tests and predicates
- Examples
- Functions

Context

- XPath expressions are evaluated with respect to a (dynamic) expression context

  The context of an expression consists of
  - Context item: the item currently being processed (context node in case it is a node)
  - Context size: the number of items in the sequence of items currently being processed
  - Context position: the position of the context item within the sequence of items currently being processed (≤ context size, first context position is 1)
  - Bindings for variables, functions, namespaces, …
  - …
- Can be established outside the XPath expression as initial context item

Path Expressions

- Step (location step)
  - Consists of three parts:

```
axis::node-test[predicate 1][predicate 2]…
```

  - An axis: Selects a set of nodes (items currently being processed) based on the tree structure of the document
  - A node test: Narrows this set based on node kinds and node names
  - Zero or more predicates: Further filters the result of the node test
    - Predicates are evaluated from left to right.
- Path expression (location path)
  - Sequence of steps separated by "/"
  - Initial "/" starts evaluation at the root node

Location Path

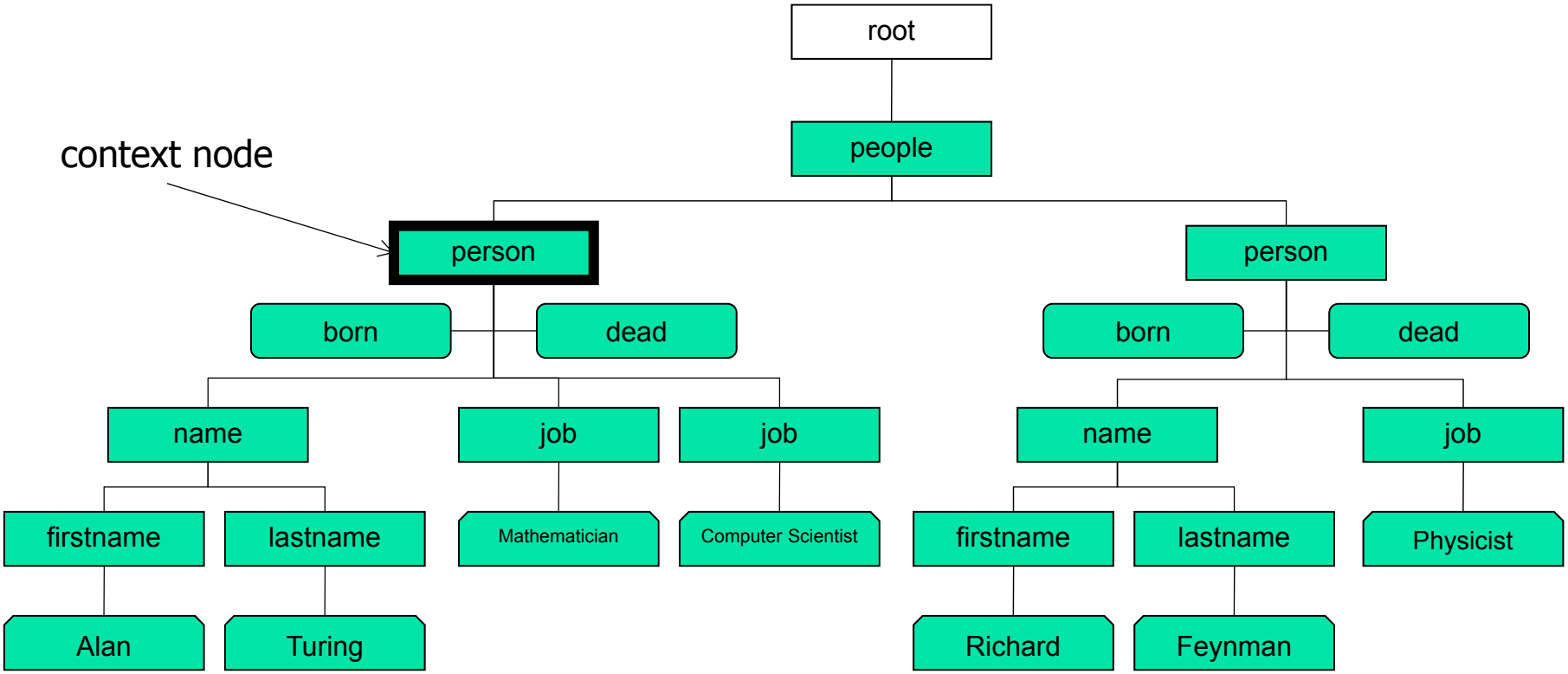
- relative location path
  - sequence of one or more location steps, seperated by "/"
  - composed from left to right
  - relative to the context node
- absolute location path
  - "/" followed by a relative location path
  - is always relative to the root node in the document

Axes

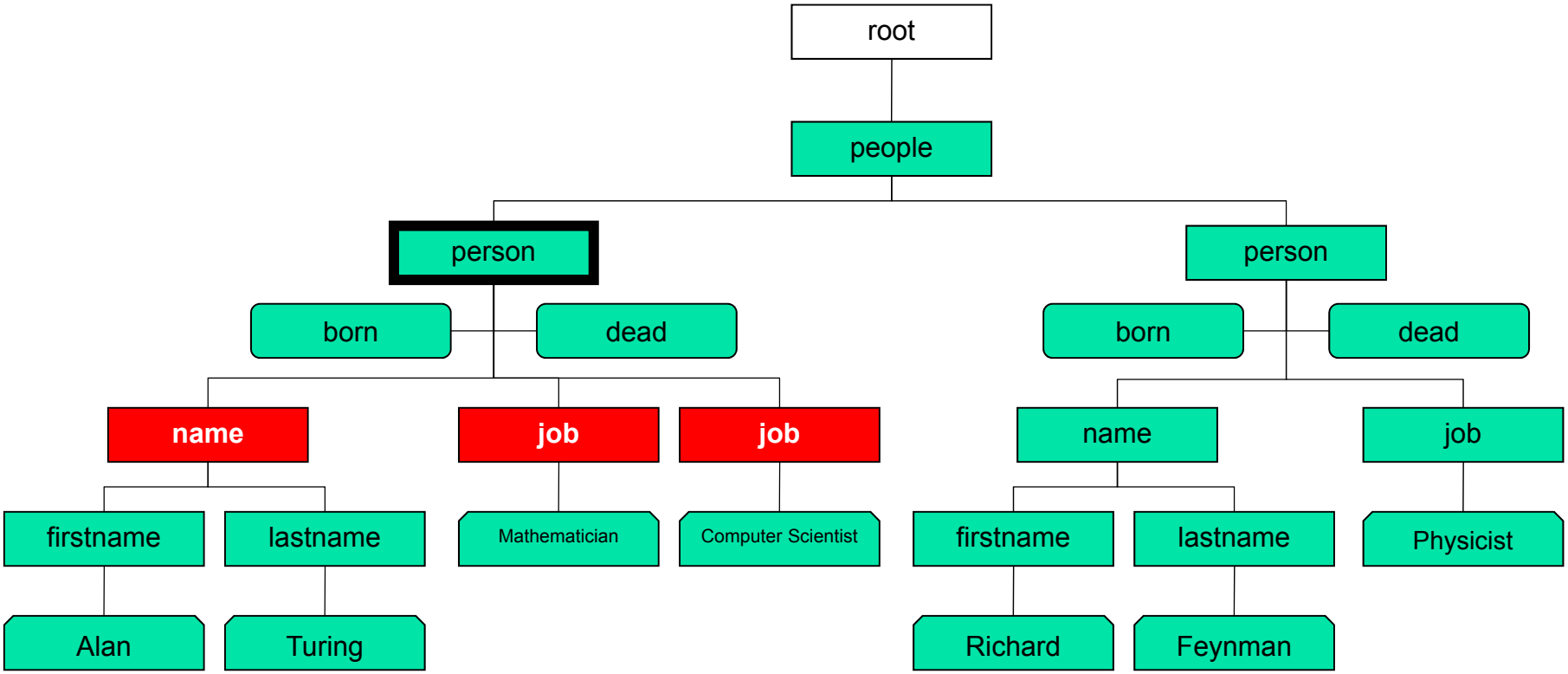
- An axis (plural axes) is a set of nodes relative to a given node
- X::Y means "choose Y from the X axis"
- Example:
  - /child::lecture/child::Chapter2

```
<lecture>
 <Chapter1>
  XML-Daten...
 </Chapter1>
 <Chapter2>
  DTDs...
 </Chapter2>
 <Chapter3>
  <Chapter3a>
   Syntax...
  </Chapter3a>
 </Chapter3>
</lecture>
```

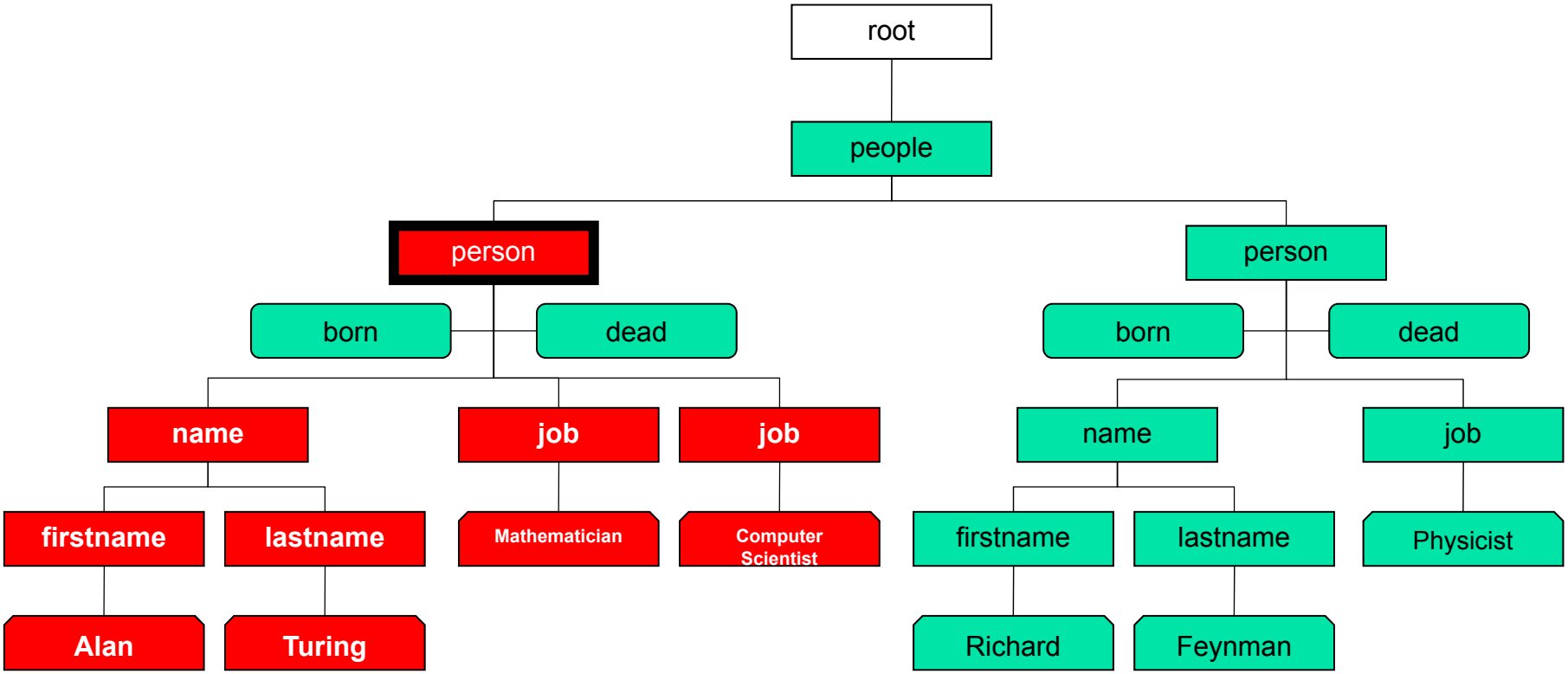
Example


root
people
context node
person
person
born
dead
born
dead
name
job
job
name
job
firstname
lastname
Mathematician
Computer Scientist
firstname
lastname
Physicist
Alan
Turing
Richard
Feynman

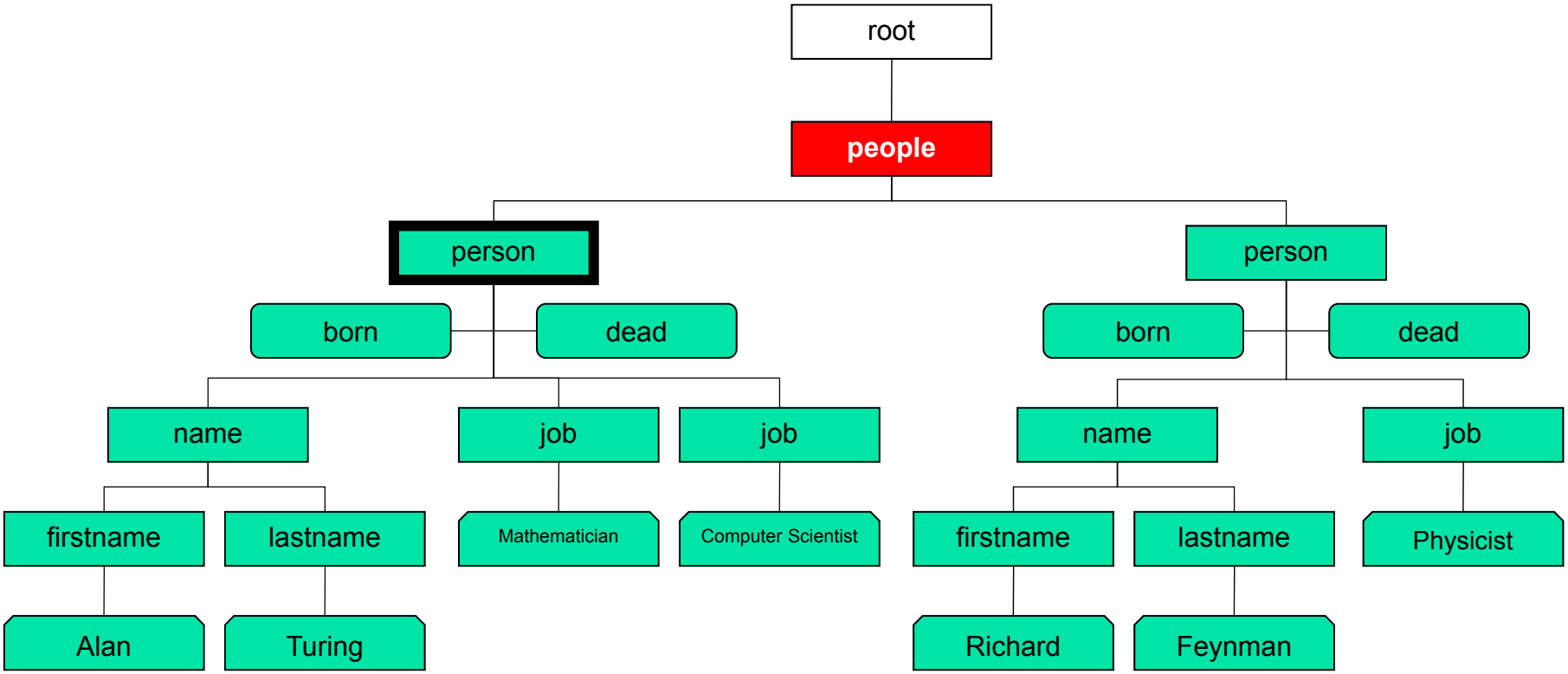
child axis


root
people
person
person
born
dead
born
dead
name
job
job
name
job
firstname
lastname
Mathematician
Computer Scientist
firstname
lastname
Physicist
Alan
Turing
Richard
Feynman

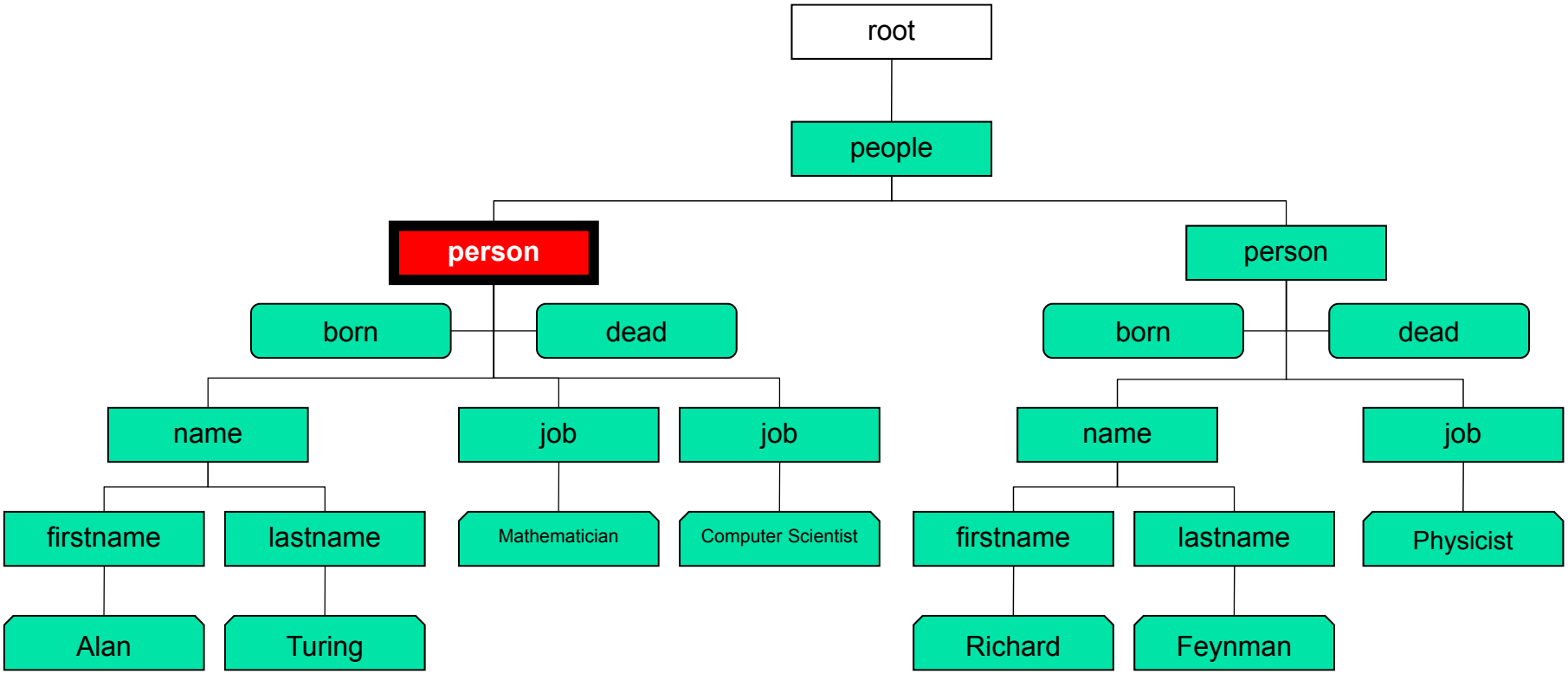
descendant-or-self axis



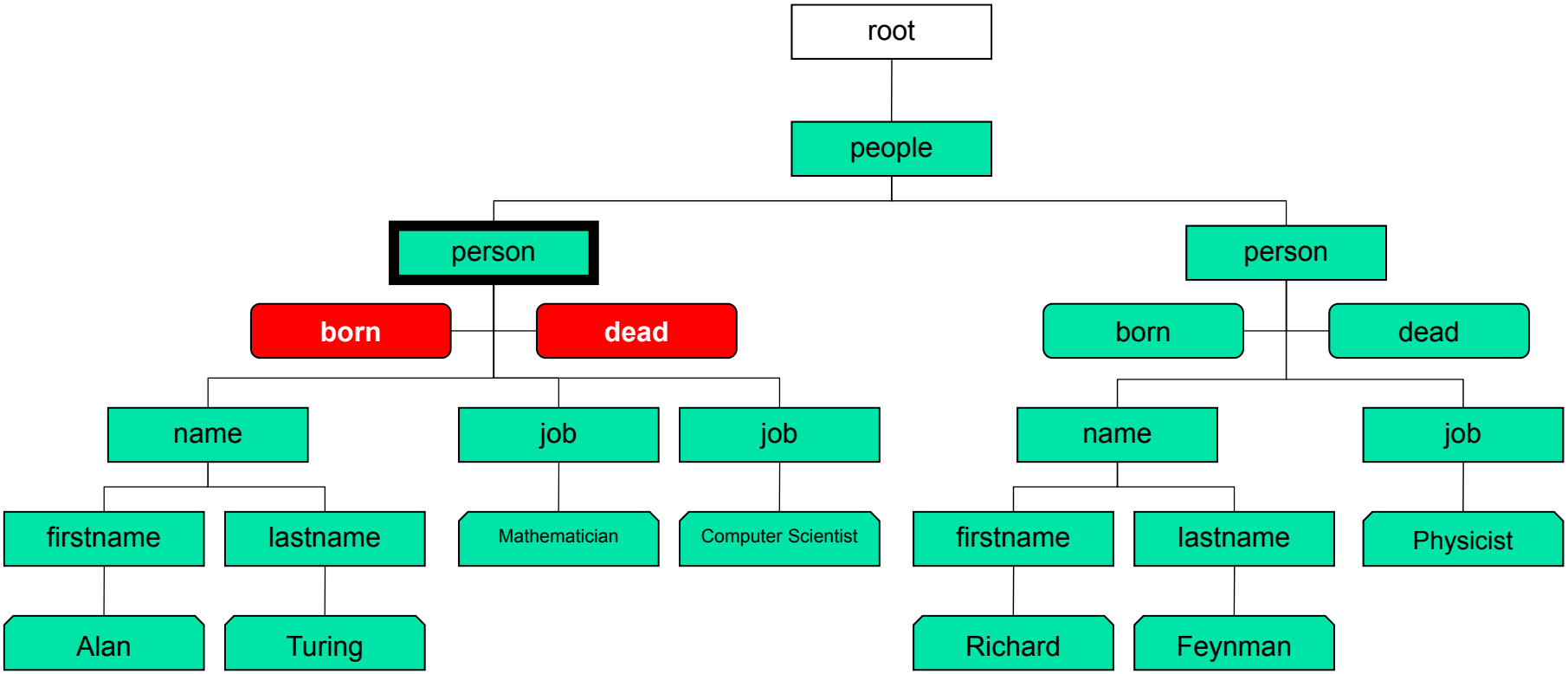
parent axis


root
people
person
person
born
dead
born
dead
name
job
job
name
job
firstname
lastname
Mathematician
Computer Scientist
firstname
lastname
Physicist
Alan
Turing
Richard
Feynman

self axis


root
people
person
person
born
dead
born
dead
name
job
job
name
job
firstname
lastname
Mathematician
Computer Scientist
firstname
lastname
Physicist
Alan
Turing
Richard
Feynman

attribute axis



Available Axes (1)

- child
  - contains the children of the context node
- descendant
  - contains the descendants of the context node
  - i. e. child, child of child, ...
  - never contains attribute or namespace nodes
- parent
  - contains the parent of the context node, if there is one
- ancestor
  - contains the ancestors of the context node
  - i. e. parent, parent of parent, ...
  - will always include the root node, unless the context node is the root node
- following-sibling
  - contains all the following siblings of the context node
  - is empty, if the context node is an attribute node or namespace node

Available Axes (2)

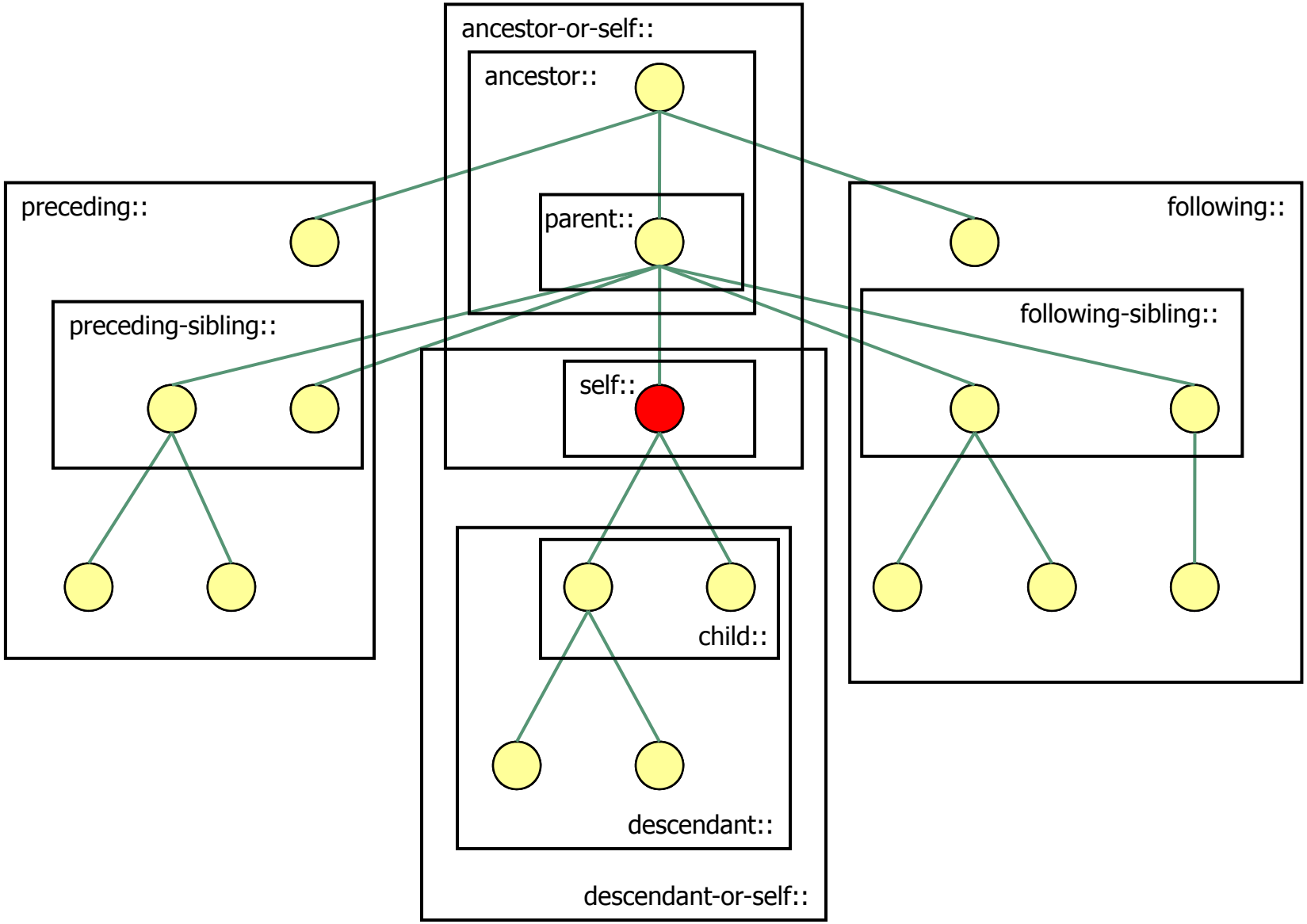
- preceding-sibling
  - contains all the preceding siblings of the context node
  - is empty, if the context node is an attribute node or namespace node
- following
  - contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes
- preceding
  - contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and excluding attribute nodes and namespace nodes
- attribute
  - contains the attributes of the context node
  - is empty unless the context node is an element

Available Axes (3)

- namespace
  - contains the namespace nodes of the context node
  - is empty unless the context node is an element
- self
  - contains just the context node itself
- descendant-or-self
  - contains the context node and the descendants of the context node
- ancestor-or-self
  - contains the context node and the ancestors of the context node
  - will always include the root node

- NOTE: The ancestor, descendant, following, preceding and self axes partition a document (ignoring attribute and namespace nodes): they do not overlap and together they contain all the nodes in the document.

Available Axes (4)



Node Tests

- Narrow result of axis set based on node kinds and node names
- Each axis has a principal node type:
  - For the attribute axis, the principal node type is attribute
  - For the namespace axis, the principal node type is namespace
  - For other axes, the principal node type is element
- Available node tests:
  - QName (e.g. "Chapter3a"): true, if name is equal
  - * : true for any node of the principal node type
  - text() : true for any text node
  - comment() : true for any comment node
  - processing-instruction() : true for any processing instruction
  - node() : true for any node of any type whatsoever

Predicates

- A predicate filters a node-set with respect to an axis to produce a new node-set
- predicate expression is evaluated for each node of the node-set and the result is converted to boolean
  - if this delivers true, the node is included in the new node-set
  - otherwise, it is not included
- converting expression results to boolean
  - if result is a number:
    true, if the number is equal to the context position, false otherwise
    /chapter[5]/section[2] selects the second section of the fifth chapter
  - a node-set is true if and only if it is non-empty
  - a string is true if and only if its length is non-zero
  - an object of a type other than the four basic types is converted to a boolean in a way that is dependent on that type

Overview

- Motivation and introduction
- Data model
- Atomic values and simple expressions
- Path Expressions, node tests and predicates
- Examples
- Functions

Example 1

- Birthday of Feynman

```
<people>
  <person born="1914" dead="1952">
    <name>
        <firstname>Alan</firstname>
        <lastname>Turing</lastname>
    </name>
    <job>mathematician</job>
    <job>computer scientist</job>
  </person>
  <person born="1916" dead="1988">
    <name>
        <firstname>Richard</firstname>
        <lastname>Feynman</lastname>
    </name>
    <job>physicist</job>
  </person>
</people>
```

/child::people /child::person [child::name /child::lastname="Feynman" /attribute::born

Example 2

- First names of persons which grew older than 50 years

```
<people>
  <person born="1914" dead="1952">
      <name>
         <firstname>Alan</firstname>
         <lastname>Turing </lastname>
      </name>
      <job>mathematician</job>
      <job>computer scientist</job>
  </person>
  <person born="1916" dead='1988'>
      <name>
         <firstname>Richard</firstname>
         <lastname>Feynman</lastname>
      </name>
      <job>physicist</job>
  </person>
</people>
```

/child::people /child::person[attribute::dead-attribute::born>50]/child::name/child::firstname

Example 3

- Names of all computer scientists

```
<people>
  <person born="1914" dead="1952">
    <name>
        <firstname>Alan</firstname>
        <lastname>Turing </lastname>
      </name>
      <job>mathematician</job>
    <job>computer scientist</job>
  </person>
  <person born="1916" dead="1988">
      <name>
        <firstname>Richard</firstname>
        <lastname>Feynman</lastname>
      </name>
      <job>physicist</job>
  </person>
</people>
```

/child::people /child::person[child::job="computer scientist"]/child::name

Abbreviated Syntax

Verbose Syntax	Abbreviated Syntax
child::	
/descentant-or-self::node()/	//
self::node()	.
parent::node()	..
attribute::	@

- /child::people/child::person/child::name
  /people/person/name
- /child::people/child::person[attribute::born=1914]
  /people/person[@born=1914]
- self::node()/descendant-or-self::node()/child::name
  .//name
- /descendant-or-self::job[1]
  //job[1]

Not the same!

Overview

- Motivation and introduction
- Data model
- Atomic values and simple expressions
- Path Expressions, node tests and predicates
- Examples
- Functions

Functions (1)

- XPath 2.0/3.0 shares a function catalogue with XQuery 1.0 and XSLT 2.0
- XPath 1.0 uses its own function library
- Function names belong to the namespace http://www.w3.org/2005/xpath-functions
  - Commonly bound to the prefix "fn"
  - "op" prefix is not bound to a namespace, those functions cannot be called by users
- Some functions operate on the context item if invoked without arguments.

Filtering Sequences

- Filter expressions
  - Remove all items not satisfying the filter expressions
  - Similar to XPath's predicates
    - `("a","b","c")[2]` ➔ "b"

      (equivalent to ("a", "b", "c")[fn:position() = 2])
    - `("a","b","c")[position()=2 to 3]` ➔ `("b","c")`
    - `(1 to 100)[. mod 13 = 0][fn:last()]` ➔ 91

Sequences of Atomic Values

- Operators and functions for sequences of atomic values

signature	description
,	`((1,2,3),4)` ➔ `(1,2,3,4)` concatenation of sequences
op:to($firstval as xs:integer, $lastval as xs:integer) as xs:integer*	`4 to 8` ➔ `(4,5,6,7,8)` delivers sequence starting with value $firstval and ending with value $lastval
fn:index-of($seq as xdt:anyAtomicType*, $value as xdt:anyAtomicType[, $collation as xs:string]) as xs:integer*	`fn:index-of((5,7,8,5),5)` ➔ `(1,4)` returns all position where $seq contains $value; $collation defines sort order
fn:distinct-values($seq as xdt:anyAtomicType*[, $collation as xs:string]) as item()	delivers all distinct values of the input; $collation defines sort order

Cardinality of Sequences

function signature	description
fn:zero-or-one($seq as item()*) as item()?	`fn:zero-or-one(())`➔`()` `fn:zero-or-one((1,2))`➔ ⚡ delivers input sequences if it consists of at most one item
fn:one-or-more($seq as item()*) as item()+	delivers input sequence if it consists of at least one item
fn:exactly-one($seq as item()*) as item()	delivers input sequences if it consists of exactly one item

Functions for Analysing Sequences

signature	description
fn:deep-equal($seq1 as item()*, $seq2 as item()*) as xs:boolean	returns true iff sequences are deep-equal
fn:empty($seq as item()*) as xs:boolean	true iff input sequence is empty
fn:exists($seq as item()*) as xs:boolean	true iff input sequence is not empty
fn:count($seq as item()*) as xs:integer	returns the number of items in sequence

Functions for Altering Sequences

signature	description
fn:insert-before($seq as item()*, $position as xs:integer, $seqnew as item()*) as item()*	`fn:insert-before(("A","B","C","D"),2,("A1","A2"))` ➔`("A","A1","A2","B","C","D")` inserts sequence $seqnew before position $position in sequence $seq
fn:remove($seq as item()*, $position as xs:integer) as item()*	`fn:remove((9,8,7,6),3)`➔`(9,8,6)` delete position $position of $seq; $seq is returned if position does not exist
fn:reverse($seq as item()*) as item()*	`fn:reverse((1,2,3,4))`➔`(4,3,2,1)` reverses order of sequence elements
fn:subsequence($seq as item()*, $start as xs:double[, $length as xs:double]) as item()*	`fn:subsequence(("A","B","C","D","E"),3,2)`➔`("C","D")` returns subsequence starting at position $start and having $length elements; if no length is specified, all elements until the end are returned
fn:distinct-values($seq as xs:anyAtomicType*) as item()	`fn:distinct-values((5,7,8,5))`➔`(5,7,8)` removes duplicates from $seq

Node Sequences

- Operators only for sequences of nodes
  - $node-list1 union $node-list2
  - $node-list1 intersect $node-list2
  - $node-list1 except $node-list2
- Duplicates are removed
- Result is sorted in document order
- Examples

```
<x/> union <x/> ➔ (<x/>,<x/>)
```

```
let $x := <x/> return $x union $x ➔ (<x/>)
```

```
let $a := <a/>, $b := <b/> return ($b, $a) union () ➔ (<a/>, <b/>)
```

Accessors for Nodes

$n	fn:node-name($n)	fn:string($n)	fn:data($n)
element p:a { "foo" }	{http://example.org}a	foo	foo
attribute p:a { "foo" }	{http://example.org}a	foo	foo
text { "foo" }	()	foo	foo
<!-- foo -->	()	foo	foo
<?pi foo ?>	pi	foo	foo
foobar	a	foobar	foobar
<a><!-- foo -->bar	a	bar	bar

- Prefix p is assumed to be bound to http://example.org
- fn:string always converts the result to an xs:string
  - fn:string(<a xsi:type="xs:int">1) + 2 ➔ error
  - fn:data(<a xsi:type="xs:int">1) + 2 ➔ 3

Functions on Nodes

signature	description
fn:root($arg as node()?) as node()?	returns the root of the tree of the node
fn:name($arg as node()?) as xs:string	returns result of fn:node-name($arg) as xs:string
fn:local-name($arg as node()?) as xs:string	equivalent to fn:local-name-fromQName(fn:node-name($arg))
fn:namespace-uri($arg as node()?) as xs:anyURI	equivalent to fn:namespace-uri-from-QName (fn:node-name($arg))

Others

- Type conversions
  - fn:data(arg) performs atomization of arg
    - converts a sequence to a sequence of atomic values
    - nodes are replaced by their typed-value

```
fn:data((1,2,<x>3 4</x>) ➔(1,2,3,4)
```

    - implicit atomization in arithmetic expressions, comparisons, cast, results of functions, etc.
  - fn:boolean(arg) returns the effective boolean value of arg
    - booleans are returned unchanged
    - empty sequence: false
    - sequences starting with a node: true
    - xs:string / xs:anyURI / xs:untypedAtomic: true, iff length is > 0
    - numeric values: false, if 0 or NaN, true otherwise
    - all other cases are errors
- Retrieving documents
  - fn:doc(uri) returns the document node of the document denoted by uri

Literature & Information

[XPa14a] XPath and XQuery Functions and Operators 3.0, W3C Recommendation 08 April 2014

[XPa14b] XQuery and XPath Data Model 3.0, W3C Recommendation 08 April 2014

[XPa14c] XML Path Language (XPath) 3.0, W3C Recommendation 08 April 2014