XML Data Management

6. XPath 1.0 Principles

Werner Nutt

XPath Expressions and the XPath Document Model

- XPath expressions are evaluated over documents
- XPath operates on an *abstract document* structure
  (essentially the same as DOM)
- Documents are *trees* with several *types of nodes*, the most important of which are
  - element nodes
  - attribute nodes
  - text nodes

There are other node types (namespaces, comments, etc.), which we ignore in this lecture

The Recipes Example (DTD)

```
<!ELEMENT recipes     (recipe*)>
<!ELEMENT recipe       (title, ingredient+, preparation, nutrition)>
<!ELEMENT title            (#PCDATA)>
<!ELEMENT ingredient (ingredient*, preparation?)>
<!ATTLIST ingredient
        name   CDATA #REQUIRED
        amount CDATA #IMPLIED
        unit   CDATA #IMPLIED>
<!ELEMENT preparation (step+)>
<!ELEMENT step             (#PCDATA)>
<!ELEMENT nutrition      EMPTY>
<!ATTLIST nutrition
        calories CDATA #REQUIRED
        fat         CDATA #REQUIRED>
```

A Recipe Document

```
<recipes>
 <recipe>
  <title>Zuppa Inglese</title>
  <ingredient name="egg yolks" amount="4"/>
  <ingredient name="milk" amount="2.5" unit="cup"/>
  <ingredient name="Savoiardi biscuits" amount="21"/>
  <preparation>
    <step>Warm up the milk in a sauce pan.</step>
    <step>In a large bowl beat the egg yolks with the sugar.</step>
   </preparation>
   <comment>Refrigerate for at least 4 hours.</comment>
   <nutrition calories="612" fat="49"/>
  </recipe>
</recipes>
```

Document Nodes Are Ordered

Document order

- *Element* e1 is “before” e2 if the opening tag of e1 occurs before the opening tag of e2
- *Element* e is “before” its *attributes*

  (order on attributes is implementation dependent, but most often attributes are ordered according to their occurrence)
- The *attributes* of e are “before” the *child* elements of e
- If e1 is “before” e2, then all attributes of e1 are “before” all attributes of e2

Reverse document order is document order backwards

Expressions

There are two kinds of expressions, returning either

- a *set of nodes* (= “node sets”), or
- a *value* (i.e., number, string, boolean)

Mechanism:

- specify node sets
- compute values from node sets
- use values in conditions that further constrain a node set

Example: //recipe/nutrition[@calories > 1000]

The Basic Mechanism for Specifying Node Sets: Location Steps

A location step

- goes in some *direction* (i.e., along some “axis”)
- leads to a node with a certain *property*

Properties can be specified by *node tests* and zero or more *predicates*.

Node tests test for

- *node types*: e.g., element (“**`*`**”), text (“**`text()`**“),
- element or attribute *names*

Syntax: **`<axis>::<test>[<pred1>]...[<predN>]`**

Location Steps: Examples

- `descendant::*`

  all descendant elements

- `following-sibling::ingredient`

  all “ingredient” siblings following in document order

- `following::text()`

  all following text nodes

- `@*`

  all attributes

Location Steps: Examples (cntd.)

- `@amount`

  all “amount” attributes

- `descendant::ingredient[@amount=1.5]`

  all descendant elements with name “ingredient” where the attribute “amount” has the value 1.5

- `descendant::ingredient[position()=2]`

  the second descendant element with name “ingredient”

- `descendant::*[self::ingredient][2]`

  same as above

Steps Can be Combined to Paths

A path has a *starting point*, which can be

- the *root* of the document tree: "/“
- a “*current node*”

Example:

```
/descendant::recipe[1]
/child::ingredient[@unit="tablespoon"]
/@name
```

“ / “ has *two meanings*:

- “*the root*” at the beginning of a path
- step *concatenation*

Semantics of Steps and Paths (1)

- A step leads from a *node* (the “context node”) to a *set of nodes* (which may be empty)

- For any node *n*, and axis α, there is the set of nodes *reachable from n via α*,

denoted as $R_{\alpha}(n)$

(The definition of reachable nodes for an axis α is more or less as one would expect. More later on.)

Semantics of Steps and Paths (2)

Consider a step $S = \alpha$ ::<test>[<pred>]

- If S starts from a node *n* , then it returns the set $S(n)$ of all nodes in $R_\alpha(n)$ that satisfy
  - the test and
  - the predicate
- The set $S(n)$ is the *context* for each node in the set.

Semantics of Steps and Paths (3)

A path P is a *sequence of steps*

$$P = S_1/S_2/\ldots/S_n$$

A path defines a *set of nodes* P(n) as follows:

- If P consists of a *single step* S, then P(n) = S(n)
- If P is a combined path $P = P_0/S$, then

  P(n) = Union of all S(n_0) where n_0 in $P_0(n)$
- The context of a result node is determined

  by the last step

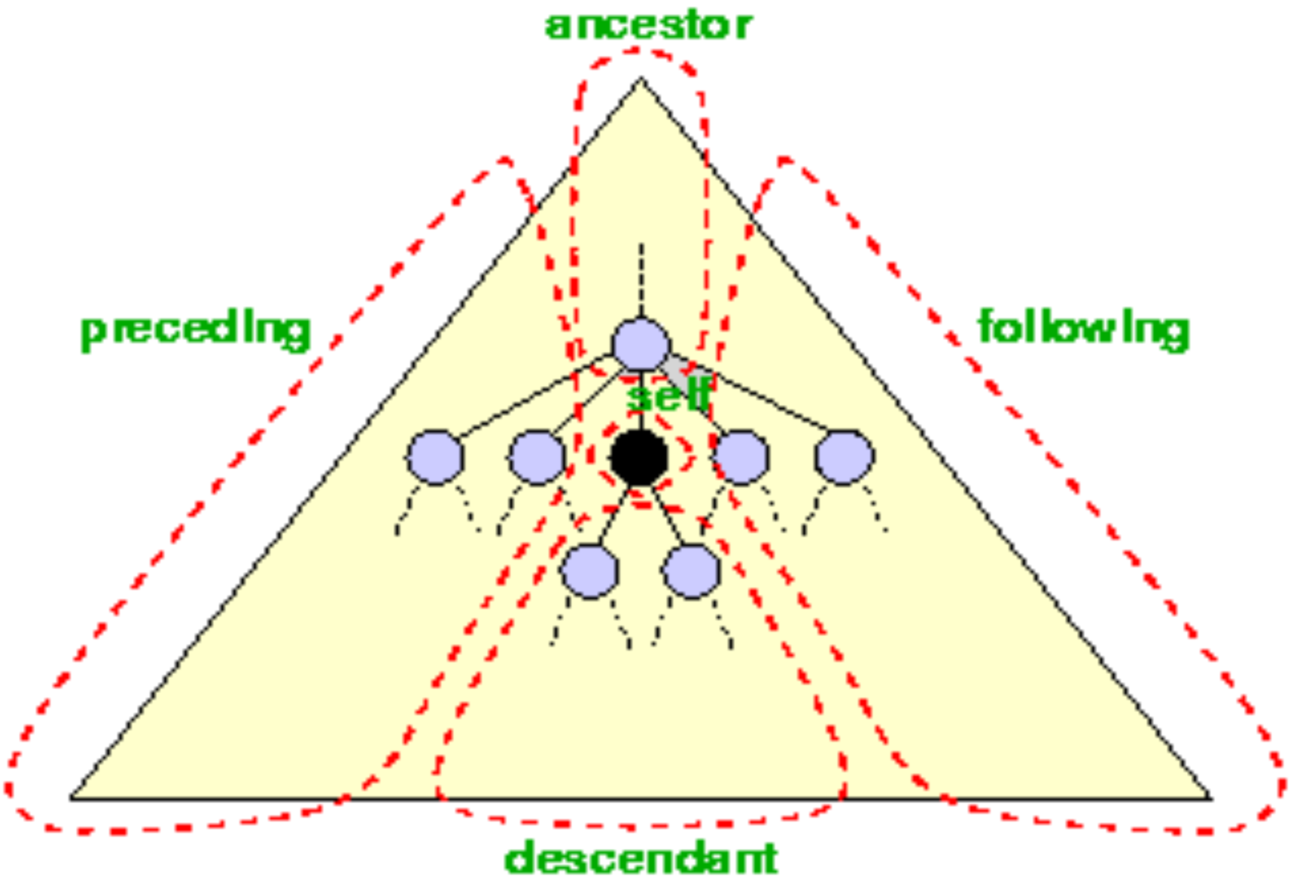
Axes in XPath

• child	the *children* of the context node
• descendant	all *descendants* (children, children’s children, ...)
• parent	the *parent* (empty if at the root)
• ancestor	all *ancestors* from the parent to the root
• self	the *context node* itself
• following-sibling	siblings to the *right*
• preceding-sibling	siblings to the *left*

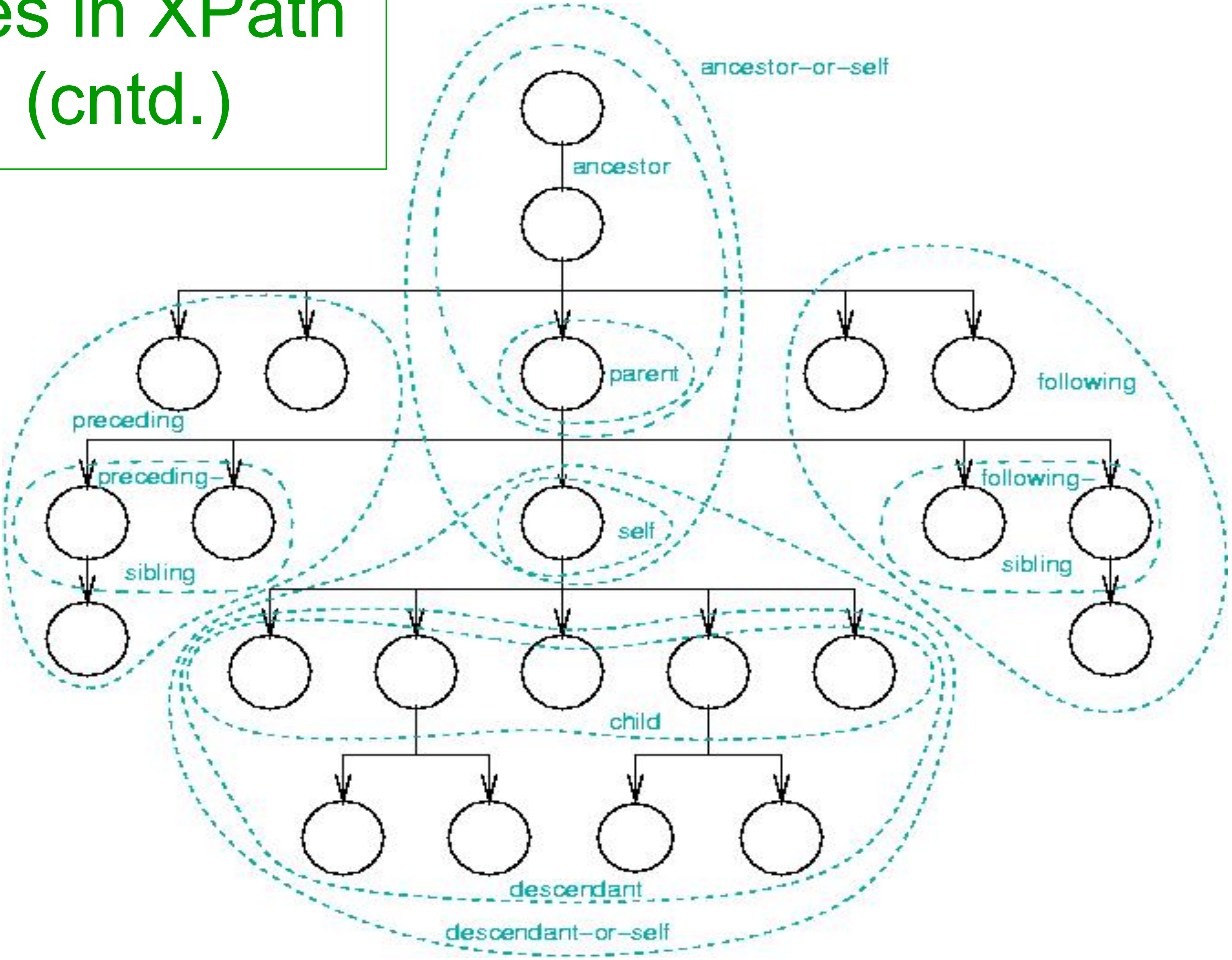
Axes in XPath (cntd.)

- following all *following* nodes in the document, excluding descendants
- preceding all *preceding* nodes in the document, excluding ancestors
- attribute the *attributes* of the context node
- namespace *namespace declarations* in the context node
- descendant-or-self the *union* of descendant and self
- ancestor-or-self the *union* of ancestor and self

Axes in XPath (cntd.)



Axes in XPath (cntd.)



Ordering of Axes

What should be the meaning of

```
/descendant::recipe[last()]
                  /preceding::recipe[2] ?
```

Forward axes: child, descendant, following-sibling, following

Reverse axes: ancestor, preceding-sibling, preceding

- By *default*, the ordering of a node set is *document order*.
- If a node set has been obtained by a step along a reverse axis, it is in reverse document order.

Node Tests

Testing by node type:

- text() character data nodes
- comment() comment nodes
- processing-instruction()
  processing instruction nodes
- node() all nodes (not including attributes and namespace declarations)

Testing by node name (for elements and attributes):

- recipe nodes with the name “recipe”
- * any element (*) or attribute node (@*)

Node Tests (Exercises)

What is the meaning of

- /descendant::text() ?
- /descendant::* ?
- /descendant::node() ?
- /descendant::*/@amount ?
- /descendant::*/@* ?

Shorthands

There are shorthands for moving along the descendant and the child axis

`//`	means	/descendant-or-self::node()/
`/recipes`	means	/child::recipes
`/*`	means	/child::*
`/node()`	means	/child::node()
`.`	means	self::node()
`..`	means	parent::*

Shorthands: Exercises

- What is the difference between //* and //. ?
- What is the result of

  //*[2] ?

  //self::*[2] ?

  //*[2]/self::* ?

- //* means /descendant-or-self::node()/child::*

  Is this different from /descendant::* !

Why?

Predicates

- Predicates are expressions of type boolean,

  although they do not always look like that ...

- A predicate filters a node-set by evaluating the predicate expression on each node in the set with
  - that node as the context node,
  - the size of the node-set as the context size, and
  - the position of the node in the node-set wrt. the axis ordering as the context position.
- Predicates can be combined with and, or, and not()

  Expressions that are not boolean are cast to boolean

Casting of Node Sets (1)

- Casting of node sets to boolean
  - true, if the set is nonempty
  - false otherwise
- Example:

  /descendant::ingredient[@unit]

  means:

  ingredients having a unit attribute

Casting of Node Sets (2)

Casting of nodes to string

- Every text node has a string as its *content*
- Every element node has a *string content*:
  - the strings occurring in the node and its descendants,
  - concatenated in document order
- Every attribute node has a string value

(which may be empty)

Casting of Node Sets (3)

Casting of nodes to number

- Interpret the string value as a number
- If not possible, value is NaN *(= "not a number")*

Casting of node sets to string, number, or boolean

- Take the value of the *first node*

wrt to document order

Casting Between Values

- XPath has explicit casting functions for the value types boolean, string, and number
- Essentially, they work as one would expect.
  - Note: an integer in a predicate is interpreted as referring to the position of the context node
- Examples:
  - string(true) = "true“, string(0) = "0“
  - number(false) = 0, number(true) = 1,
  - number("123") = 123, number("sugar") = NaN

Casting Between Values (cntd.)

- boolean(0) = false, boolean(2) = true, boolean(NaN) = false
- boolean("") = false, boolean("false") = true

What is the meaning of

- /descendant::recipe/title["Ricotta Pie"] ?

And what about

- /descendant::recipe/title[self::*="Ricotta Pie"] ?

Equalities

- Things become more complicated if a node set is involved in an equality:

  <node set> = <value>

  means:

  “*The* <node set> *contains* some *node that has the value* <value> *after casting.*”

- Similarly, the test

  <node set 1> = <node set 2>

  succeeds if

  “There is a node1 *in* <node set 1>, node2 *in* <node set 2> *s.t. the string content of* node1 *and* node2 *is equal”*

Examples

- “Recipes where sugar is *one* of the ingredients”

```
/descendant::recipe[ingredient/@name = "sugar"]
```

- “Recipes with *some* ingredient *other than* sugar”

```
/descendant::recipe[ingredient/@name != "sugar"]
```

- “Recipes where sugar is *the only* ingredient”

```
/descendant::recipe[ not (ingredient/@name != "sugar")]
```

Exercise

What is the meaning of

/descendant-or-self::*
[descendant-or-self::node() = "Zuppa Inglese"] ?

What of

/descendant-or-self::*
[descendant-or-self::* = "Zuppa Inglese"] ?

And what about

/descendant-or-self::node()
[descendant-or-self::node() = "Zuppa Inglese"] ?

Arithmetic Expressions

- XPath has *built-in functions* returning numbers, e.g.

  position() the position of the context node in the current node set

  last() the number of elements in the current node set

- With numbers and numeric functions one can build up *arithmetic expressions*

  2, 2 * 2, last() div 2, last() -1, last() - position()

Arithmetic Expressions

- A predicate that contains only a numeric expression, e.g.,

  [last() -1] ,

  is a *shorthand* for a position predicate, e.g.,

  [position() = last() -1] .
- Otherwise, numbers are cast to boolean.

Exercise: What is the meaning of

//*[2 and ingredient] ?

Aggregation Functions

- The aggregation functions count and sum are applied to node sets and return numbers.
  - The count result is *always* a number.
  - The sum result is *only* a number *if every* node in the argument set can be *cast as a number*.
- Aggregation functions in a predicate
  refer to the current node set.
- Functions min, max, and avg do not exist in XPath
  (but in XQuery)

Examples

What is the meaning of

- sum(/descendant::ingredient
  [@unit="cup" and @amount]/@amount) ?

- //recipe[count(ingredient) > 5] ?

- //recipe[count(.//ingredient) > 5] ?

XPath Expressions: Summary

An expression can be:

- a constant, e.g. "..."
- a function call: *function*(*arguments*)
- a boolean expression: or, and, =, !=, <, >, <=, >=
  (standard precedence, all left associative)
- a numerical expression: +, -, *, div, mod
- a node-set expression: using location paths
  and “ | ” (set union)

XPath Expressions: Summary

- Expressions have a type: *node-set* (set of nodes), *boolean* (true or false), *number* (floating point), or *string* (text)

- Coercion/casting may occur at function arguments and when expressions are used as predicates.

- Functions are evaluated using the context.

Core function library (1)

- Node-set functions:
  - last() returns the context size
  - position() returns the context position
  - count(*node-set*) number of nodes in node-set
  - name(*node-set*) string representation of first node in node-set
  - id(*ID*) returns element with id *ID*
- String functions:
  - string(*value*) type cast to string
  - concat(*string*, *string*, ...) string concatenation

Core function library (2)

- Boolean functions:

– boolean(*value*)	type cast to boolean
– not(*boolean*)	boolean negation
– contains(*string*, *substring*)	substring test
– starts-with(*string*, *prefix*)	prefix test

- Number functions:

– number(*value*)	type cast to number
– sum(*node-set*)	sum of number value of each node in node-set

The Family DTD

```
<!DOCTYPE family [
  <!ELEMENT family  (person)*>
  <!ELEMENT person  (name)>
  <!ELEMENT name    (#PCDATA)>
  <!ATTLIST person
       id        ID       #REQUIRED
       mother    IDREF    #IMPLIED
       father    IDREF    #IMPLIED
       children  IDREFS   #IMPLIED>
]>
```

A Family Document

```
<family>
  <person  id="lisa"  mother="marge" father="homer">
      <name> Lisa Simpson </name>
  </person>
  <person  id="bart"  mother="marge" father="homer">
      <name> Bart Simpson </name>
   </person>
  <person id="marge" children="bart lisa">
      <name> Marge Simpson </name>
  </person>
  <person id="homer" children="bart lisa">
      <name> Homer Simpson </name>
  </person>
</family>
```

Family Exercise

- Return the children of Marge.
- Return the names of the children of Marge.
- Return the father of the children of Marge.

Exercises

- Write XPath queries that ask for the following over the Recipes document:
  - The titles of all recipes, returned as strings.
  - The titles of all recipes that use olive oil.
  - The titles of all recipes that do not use olive oil.
  - The amount of sugar needed for Zuppa Inglese.
  - The recipes that have an ingredient in common with Zuppa Inglese.

Exercises (cntd.)

- The number of recipes in the document.
- The last step in preparing Zuppa Inglese.
- The average fat content per recipe.
- The recipes with less than average fat content.
- The titles of recipes that have no compound ingredients.
- The titles of recipes where all top level ingredients are compound.
- The titles of recipes that have only non-compound ingredients.