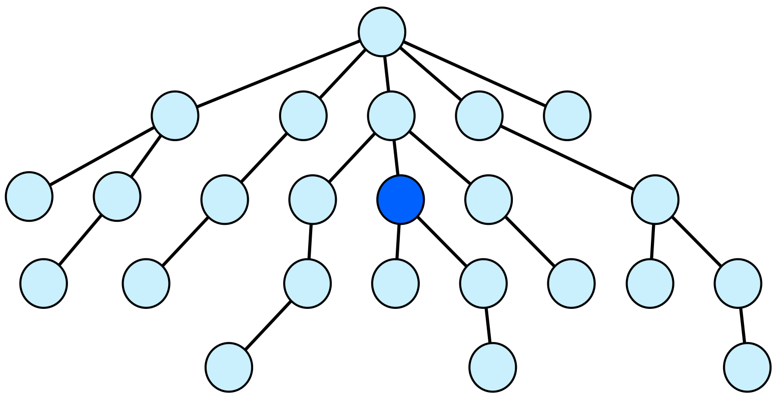
Querying and Storing XML

Week 2
XPath & XQuery
January 22-25, 2013

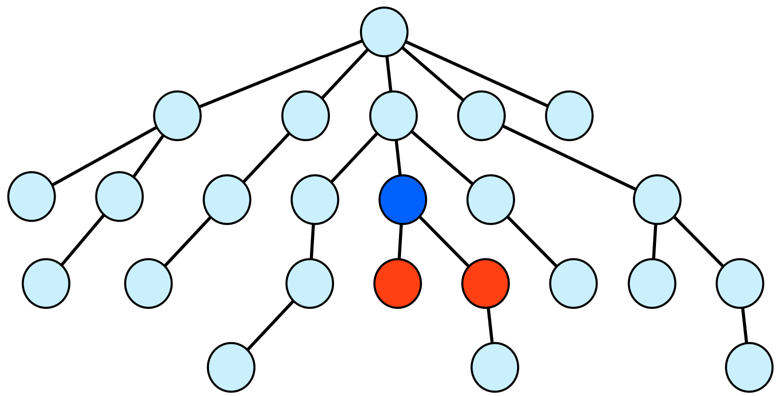
XPath

- A language of “path expressions”
- Loosely related to “file paths”
  - root (/)
  - sequential composition (p/q)
  - wildcards (*)
  - axis steps (child, parent, descendant, etc.)
- also: filters, text nodes, label tests
  - plus positional & string functions
- Used for navigation
  - component of XSLT, XQuery, etc.

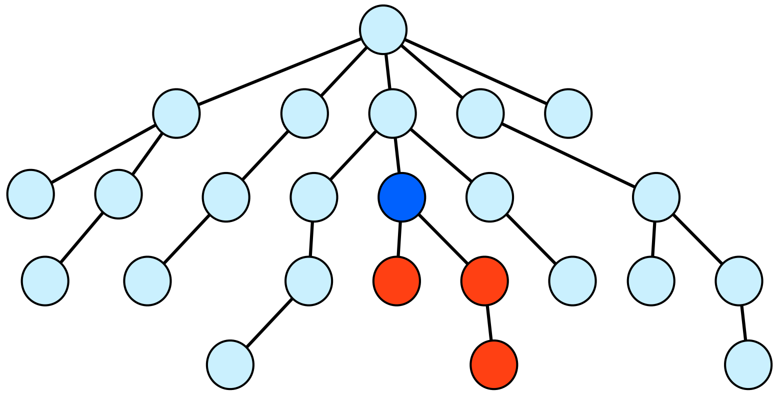
Context node (starting point)



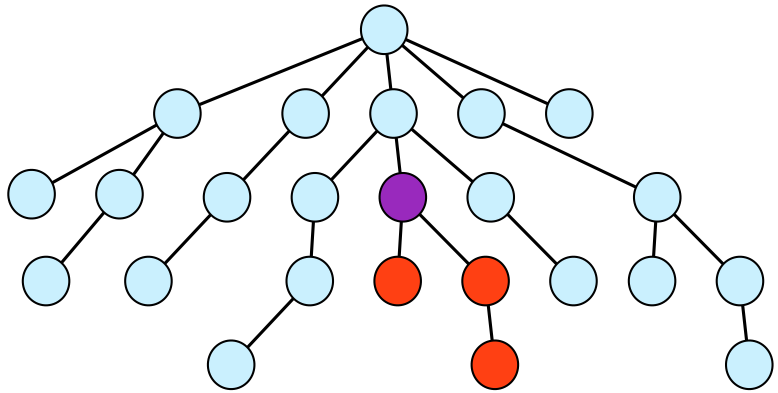
Child



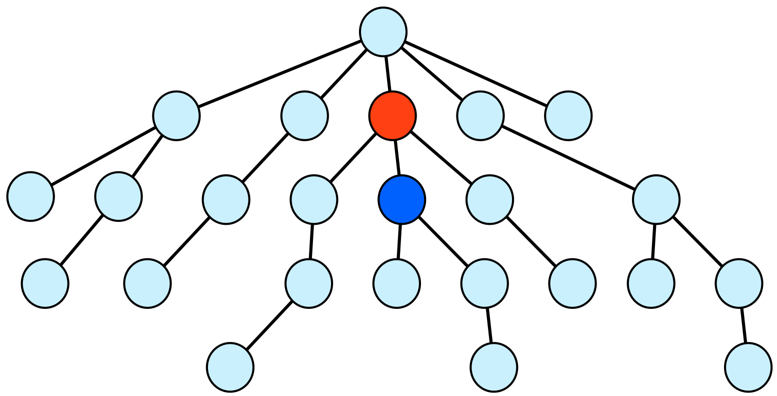
Descendant



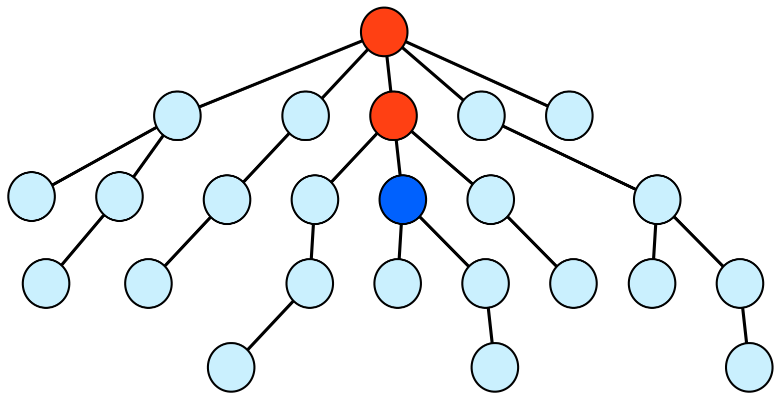
Descendant-or-self



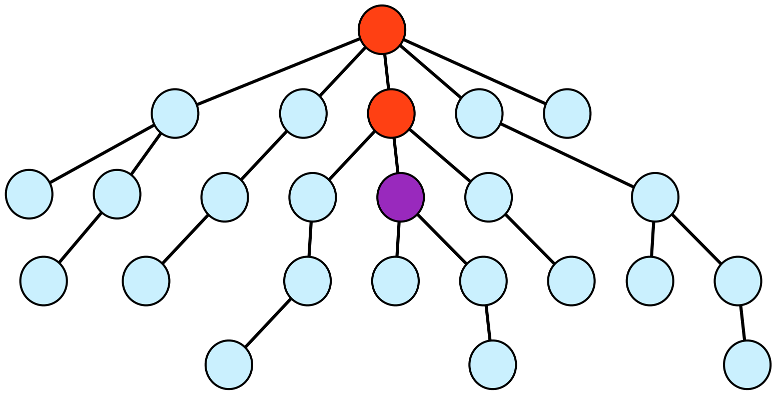
Parent



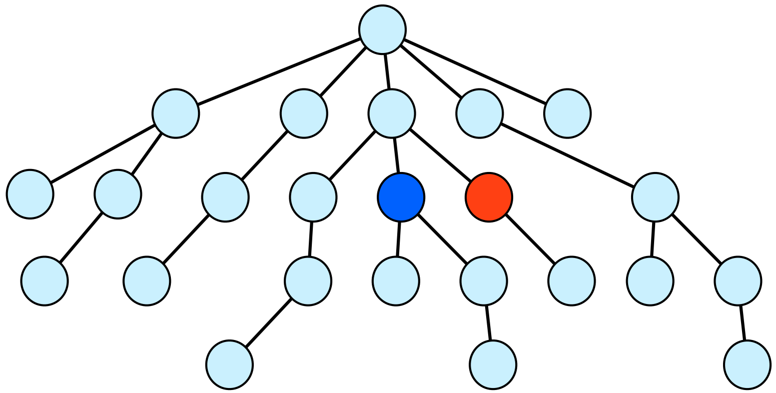
Ancestor



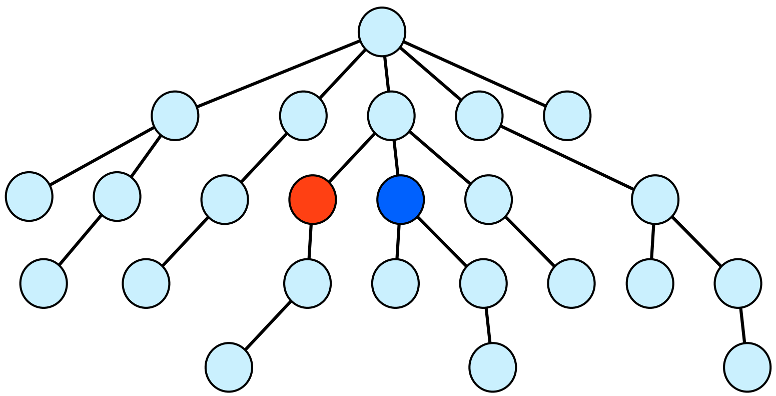
Ancestor-or-self



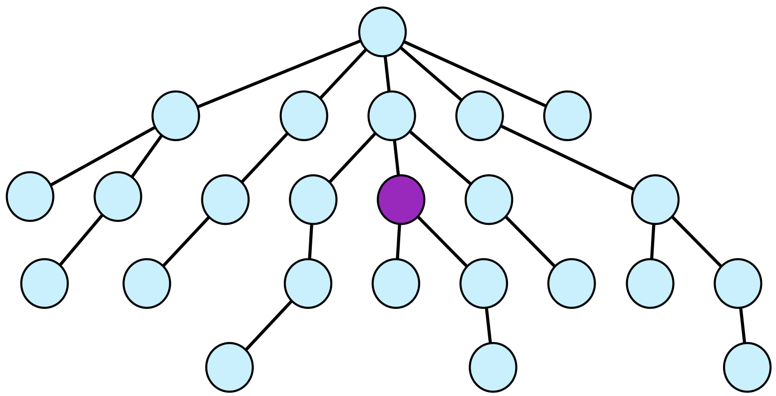
Following-sibling



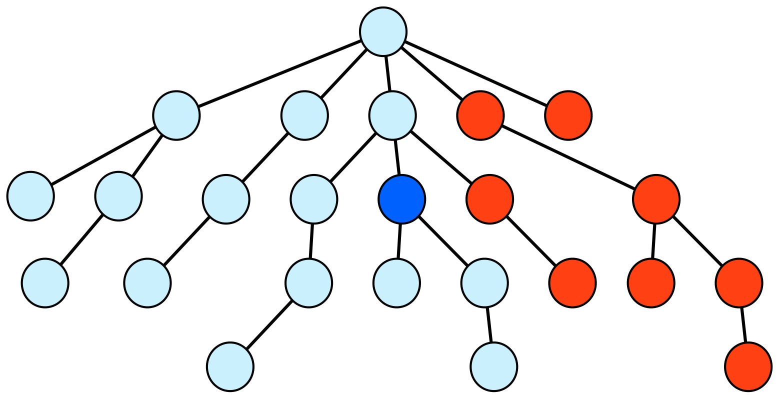
Preceding-sibling



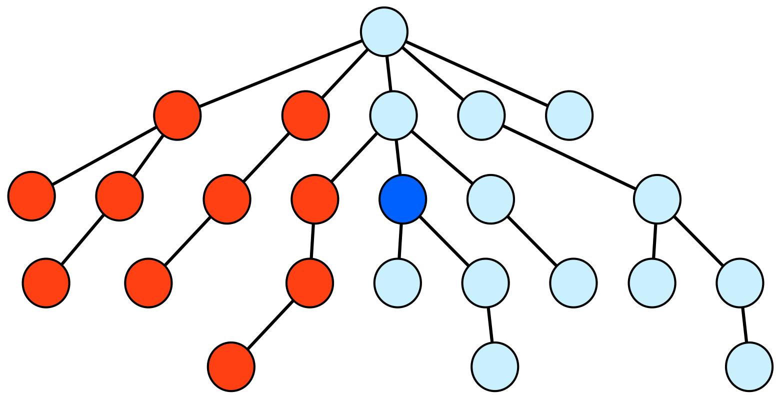
Self



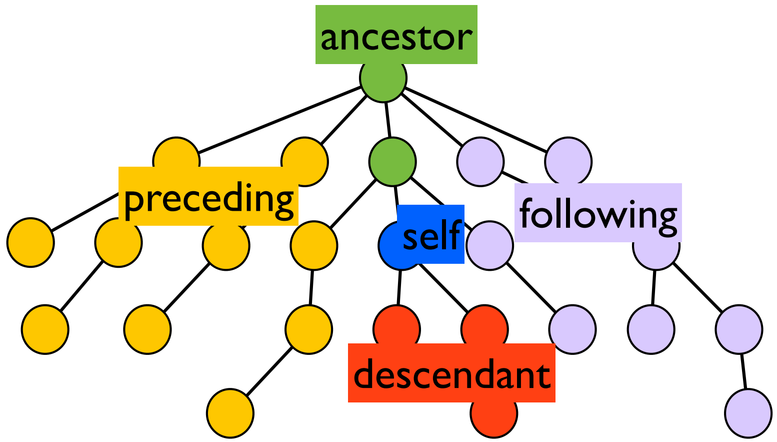
Following



Preceding



Partition



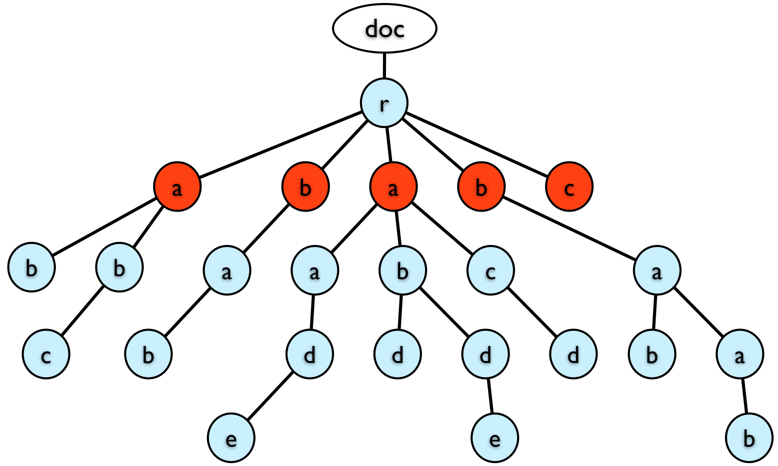
Syntax

- axis ::= child | descendant | descendant-or-self
  | parent | ancestor | ancestor-or-self
  | preceding-sibling | following-sibling
  | self | preceding | following
- test ::= * | text() | node() | a | b | @a | ...
- p ::= ax::tst[q] | p/p′
- q ::= p | q and q′ | q or q′ | not(q) | ...
- ap ::= /p

Abbreviations

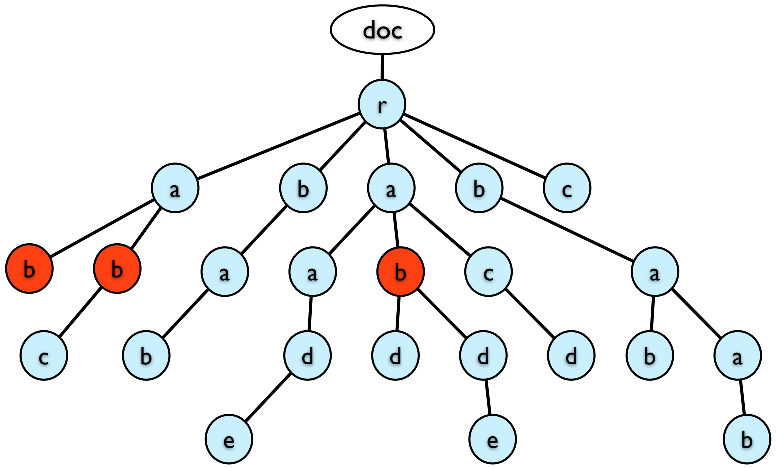
- /a = child::a
- //a = descendant-or-self::*/child::a
- . = self::*
- .. = parent::*
- Starting path with "/" means "start from document root"
  - this is a special node above the root element

Sequential composition



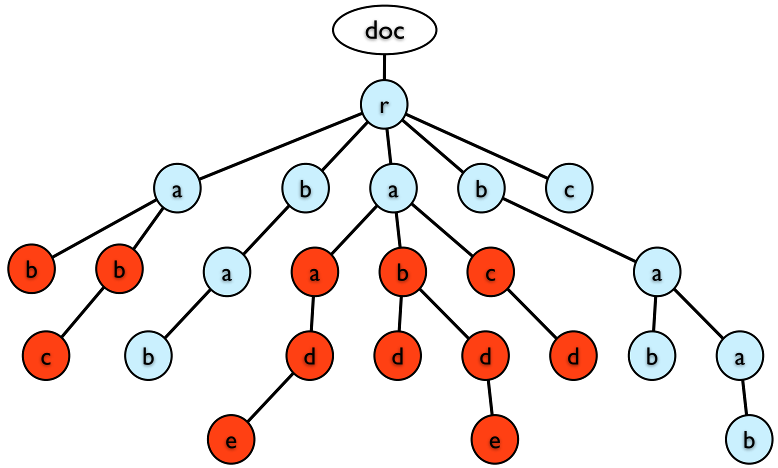
`/child::*/child::*`

Node label tests



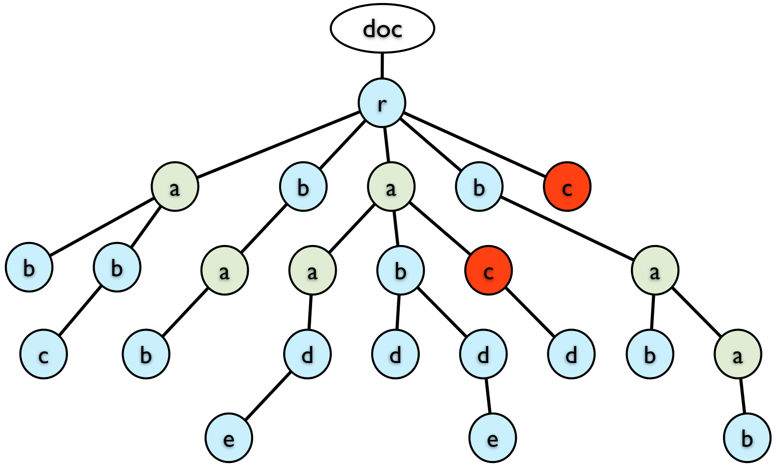
`/r/a//b`

Node label tests



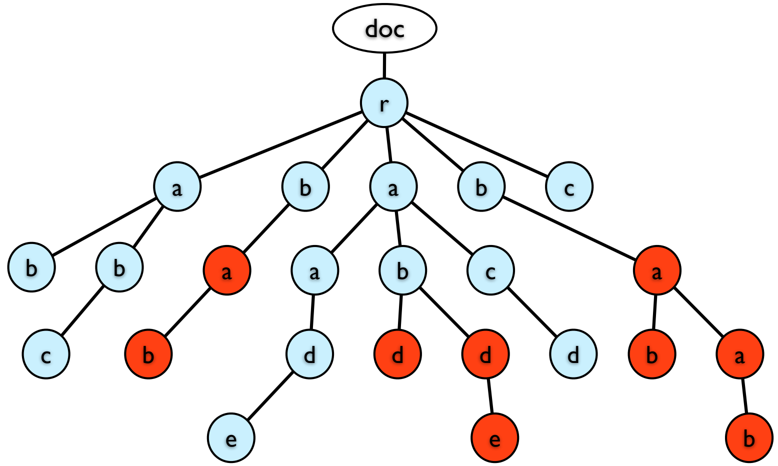
`/r/a//*`

Sequential composition



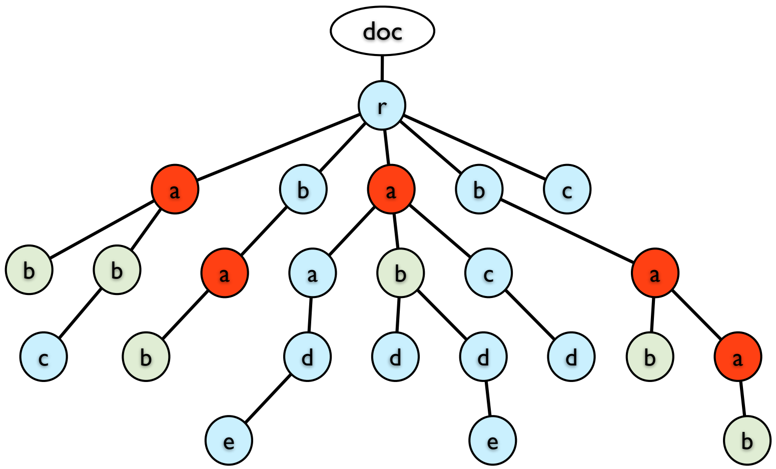
//a/following-sibling::c

Sequential composition



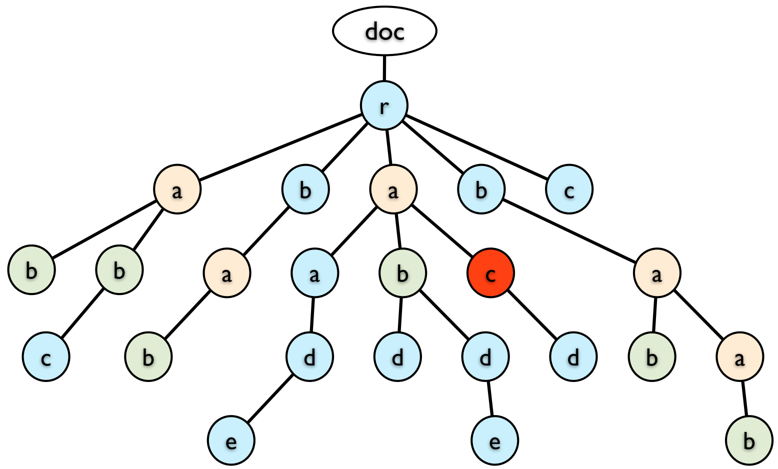
//a/following-sibling::b//*

Filters



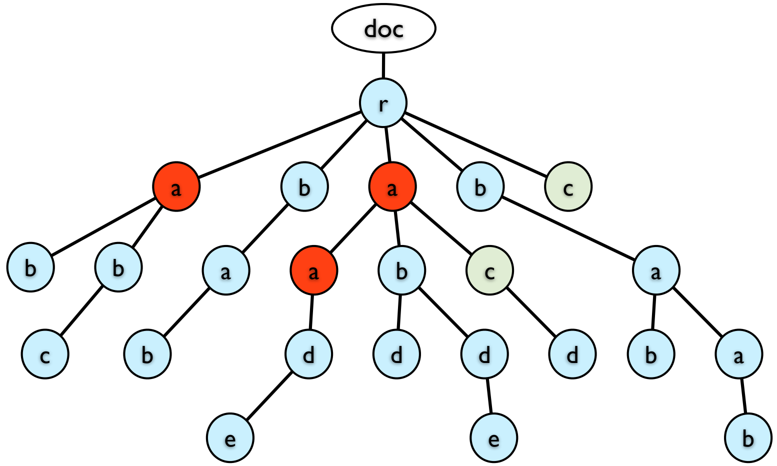
//a[b]

Filters



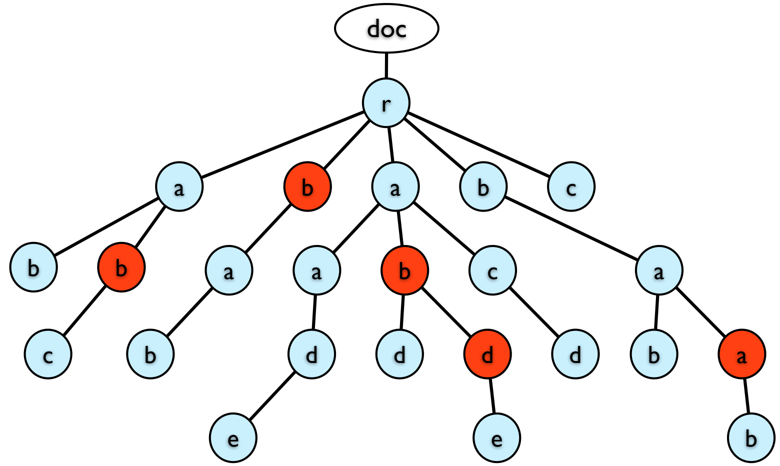
//a[b]/c

Filters



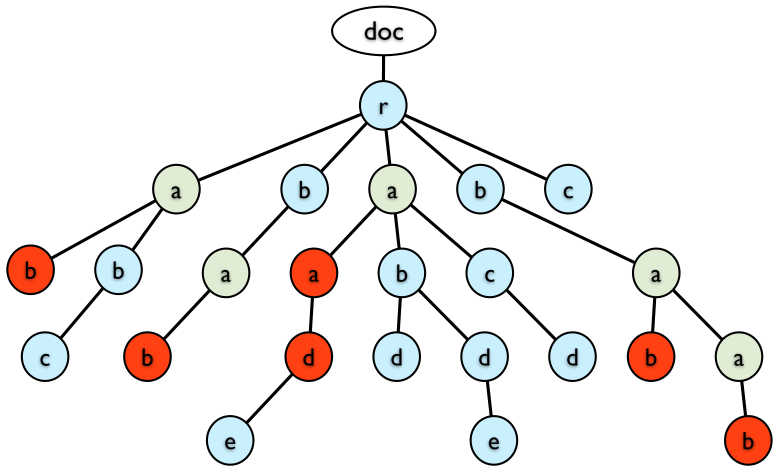
`//a[following-sibling::c]`

Positional tests



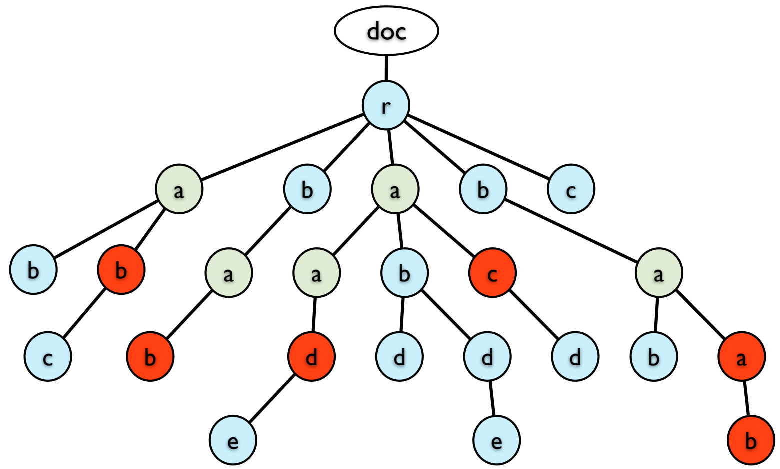
`//*[position()=2]` (or just `//a[2]`)

Positional tests



`//a/*[first()]`

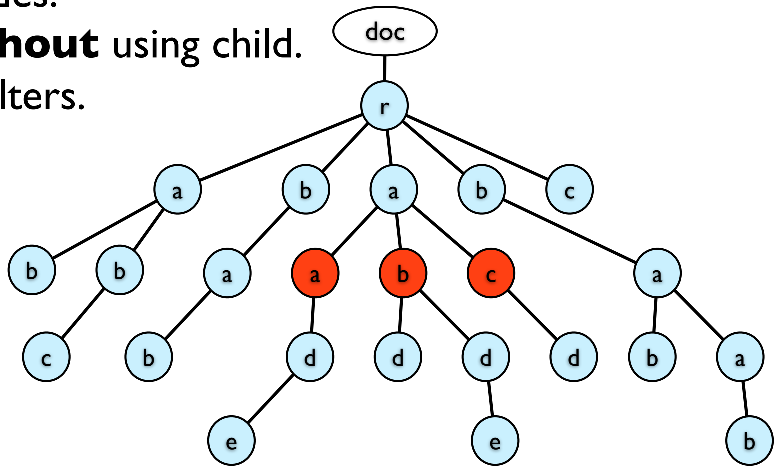
Positional tests



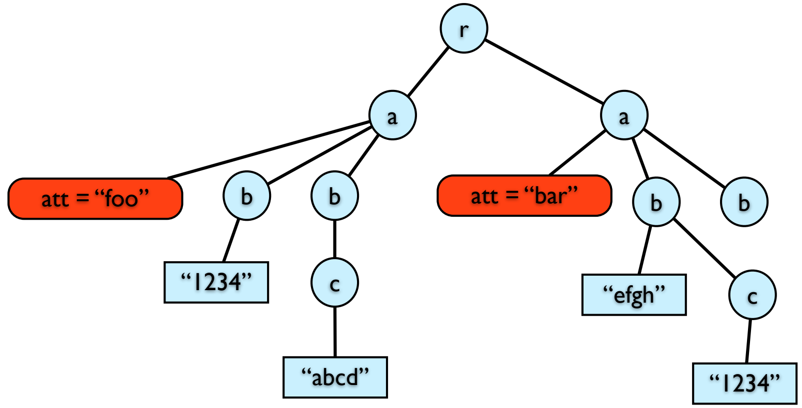
`//a/*[last()]`

Quiz

1. Write XPath to select red nodes.
2. **Without** using child.
3. Or filters.

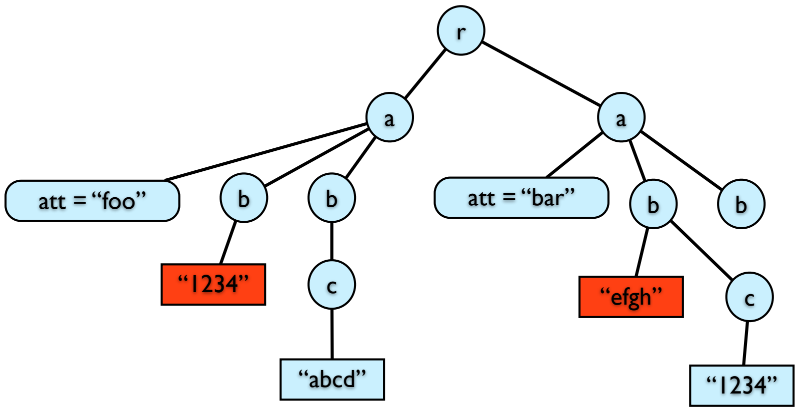


Attributes & Text



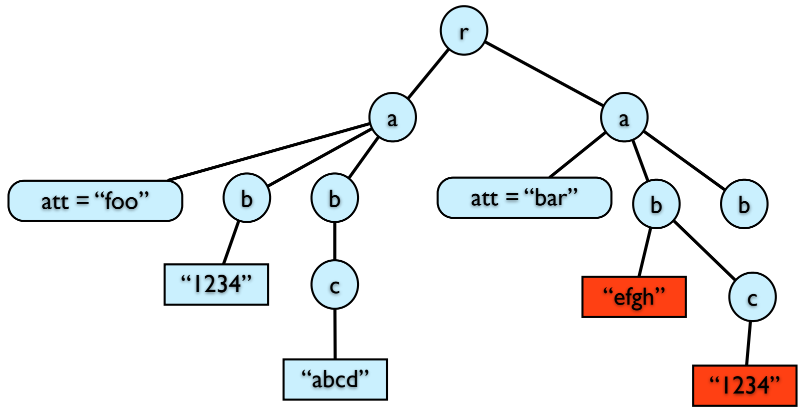
`//a/@att`

Attributes & Text



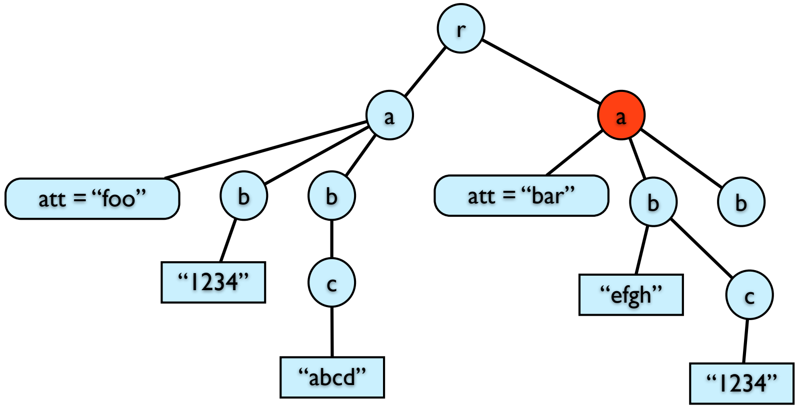
`//a/b/text()`

Equality



`//a[@att="bar"]//*/text()`

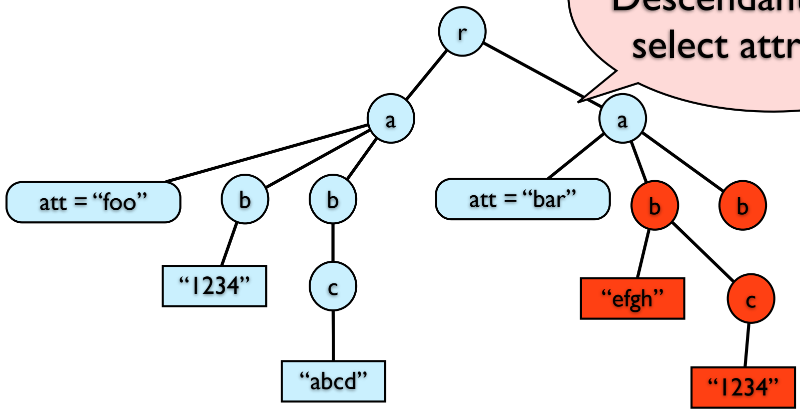
Equality



`//a[@att="bar"]`

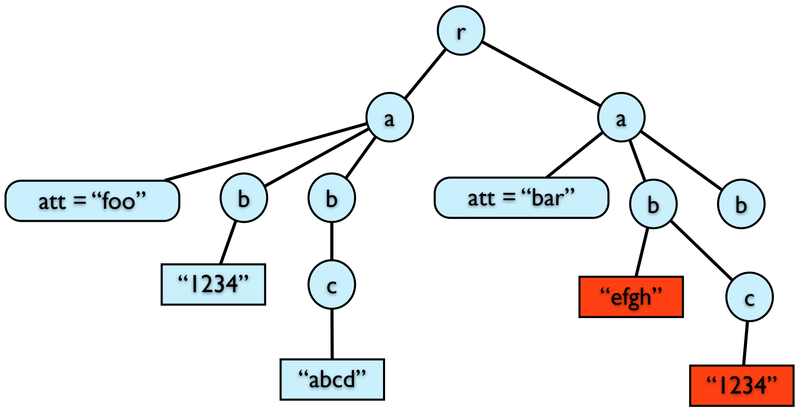
Equality

Child, Descendant do not select attributes!



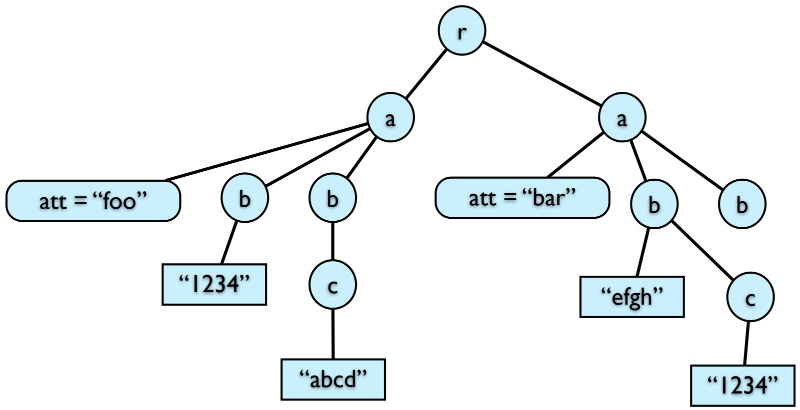
`//a[@att="bar"]//*`

Equality



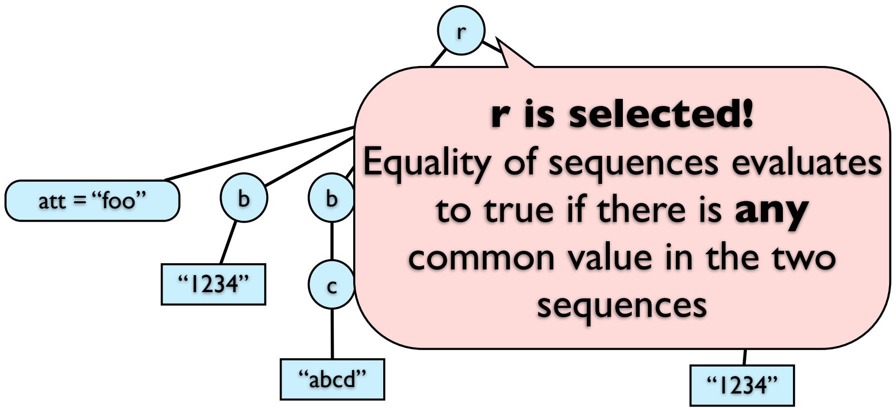
`//a[@att="bar"]//text()`

Equality quiz



`/r[a/b/text() = a/b/c/text()]`

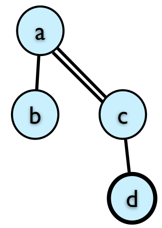
Equality quiz



`/r[a/b/text() = a/b/c/text()]`

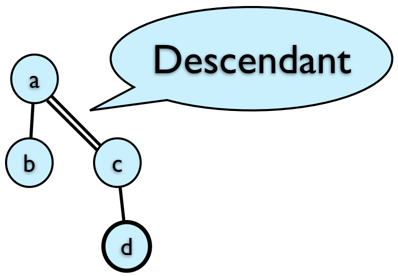
Tree patterns

- A graphical notation for (downward) XPath queries/filters

`a[b]//c/d` ≅ 

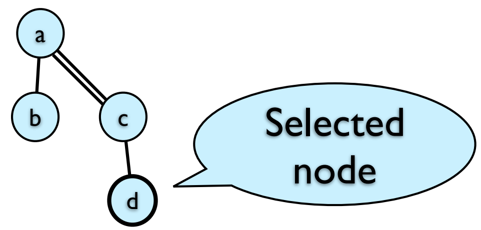
Tree patterns

- A graphical notation for (downward) XPath queries/filters

`a[b]//c/d` ≅ 

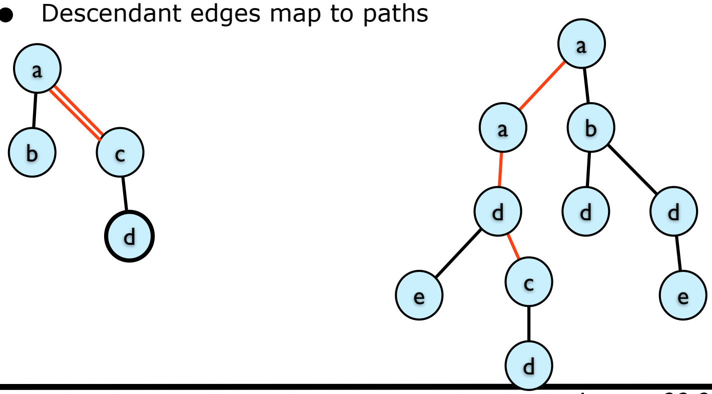
Tree patterns

- A graphical notation for (downward) XPath queries/filters

`a[b]//c/d` ≅ 

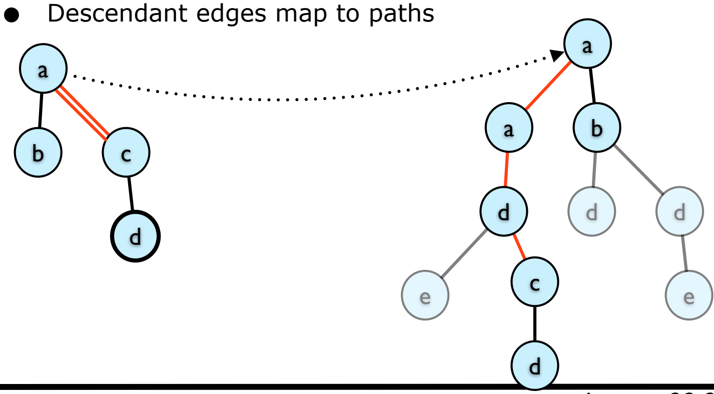
Tree pattern matching

- A function *h: P -> T* such that:
  - Child edges map to edges
  - Descendant edges map to paths



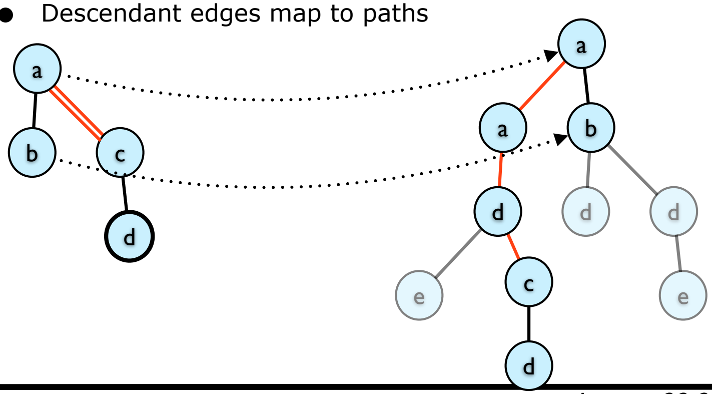
Tree pattern matching

- A function *h: P -> T* such that:
  - Child edges map to edges
  - Descendant edges map to paths



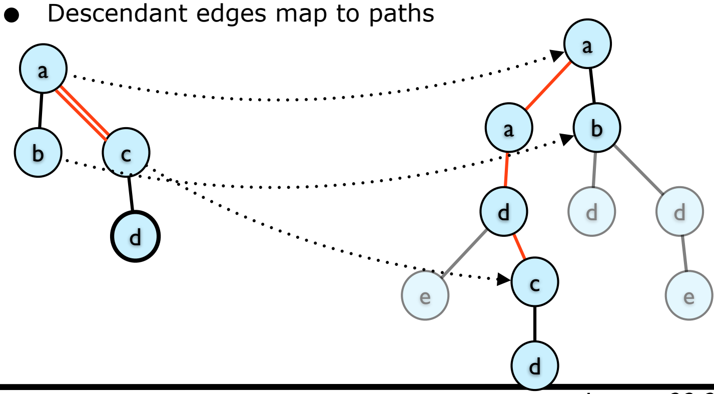
Tree pattern matching

- A function *h: P -> T* such that:
  - Child edges map to edges
  - Descendant edges map to paths



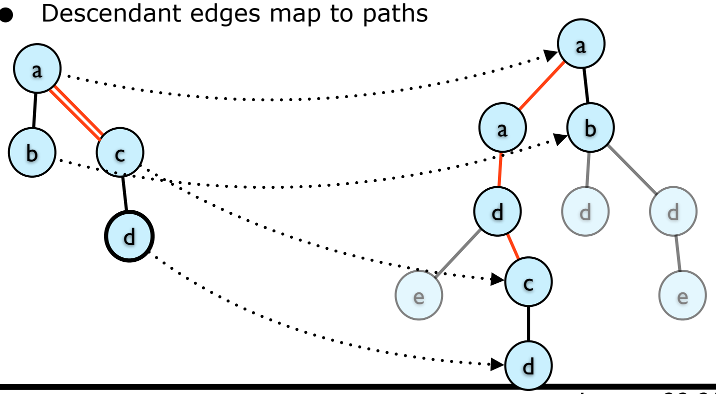
Tree pattern matching

- A function *h: P -> T* such that:
  - Child edges map to edges
  - Descendant edges map to paths



Tree pattern matching

- A function *h: P -> T* such that:
  - Child edges map to edges
  - Descendant edges map to paths



Semantics of XPath

- Represent tree as T = (V,E,λ,<)
  - Σ is set of possible node labels
  - $E \subseteq V \times V$ is parent/child edge relation
  - $\lambda : V \rightarrow \Sigma$ gives node labels
  - $< \subseteq V \times V$ linearly orders children of each node
- For simplicity, will ignore text nodes, attributes
  - but in general these need to be modeled too!

Semantics of XPath: steps

- $Ax[self](T) = \{(x,x) \mid x \in V\}$
- $Ax[child](T) = E$
- $Ax[descendant](T) = E^+$
- $Ax[descendant\text{-}or\text{-}self](T) = E^*$
- Ax[parent](T) = {(y,x) | (x,y) in E}
- ...
- $Ax[following\text{-}sibling](T) = \{(x,y) \mid x < y\}$

Semantics of XPath: tests, paths & filters

- $Test[*](T) = \{x \mid x \in V\}$
- $Test[a](T) = \{x \mid x \in V, \lambda(x) = a\}$
- $Path[ax::test](T) = \{(x,y) \in Ax[ax](T) \mid y \in Test[test](T)\}$
- $Path[p/p'](T) = \{(x,z) \mid (x,y) \in Path[p](T), (y,z) \in Path[p'](T)\}$
- $Path[p[q]](T) = \{(x,y) \mid (x,y) \in Path[p](T), y \in Filt[q](T)\}$
- $Filt[p](T) = \{x \mid \exists y.\ (x,y) \in Path[p](T)\}$
- $Filt[q\ and\ q'](T) = Filt[q](T) \cap Filt[q'](T)$
- $Filt[q\ and\ q'](T) = Filt[q](T) \cup Filt[q'](T)$
- $Filt(not(q))(T) = \{x \in V \mid x \notin Filt[q](T)\}$

Next time

- XQuery
  - Putting XPath to work
  - Iteration, binding, sequences, and XML construction expressions
  - Recursive functions

XQuery

What can XPath **not** do (well)?

- Construct new XML documents
- Combine information from different parts of document
  - Joins
- Abstraction over parts of query
  - Function definitions/recursion

XQuery - a query language for XML

- Goals:
  - "SQL-like" query language for XML
  - Support query optimization
  - Support data types/XML Schema (will cover next week)
- Design:
  - Purely functional (more or less)
  - Every expression evaluates to a value (= sequence of XML trees or primitive values)
  - Extends XPath 2.0 with comprehensions, functions

A first example

A first example

```
for $x in document(“books.xml”)/books/book
where $x/author=”Abiteboul”
return <result>
        <title>{$x/title/text()}</title>
        <year>{$x/year/text()}</year></result>
```


A first example

```
for $x in document(“books.xml”)/books/book
where $x/author=”Abiteboul”
return <result>
        <title>{$x/titl
        <year>{$x/ye
```

document() loads data from file/by name

A first example

```
for $x in document(“books.xml”)/books/book
where $x/author=”Abiteboul”
return <result>
        <title>{$x/title/text()}</title>
        <year>{$x/year/text()}</year></result>
...
<result><title>Data on the Web</title>
       <year>2000</year></result>
<result><title>Web Data Management</title>
       <year>2011</year></result>
```

Atomic values

- Integers 1,2,3
- Strings 'abcd', "abcd"
- Dates / times
- Other basic types from XML Schema (will cover these later)

Values

- Atomic constants (last slide)
- XML trees
  - <elt att1=v1 ... attn=vn>...value seq...</elt>
- Value sequences are sequences of atomic/tree values
  - (), $(v_1, v_2, \ldots, v_n)$
  - cannot be nested, i.e., $((v_1, v_2), v_3) = (v_1, v_2, v_3)$
  - however, v_1 could be an element with another sequence as content
- Formally:

  v ::= c | <elt att=v ... att=v>{vs}</elt>

  vs ::= () | $(v_1, \ldots, v_n)$

Variables

- In XQuery, variables always start with $
  - $x, $y, $z, $i
- This is common in other W3C standards with human-readable syntax
- A variable denotes a value sequence (more or less)

XML constructors

- XML values can be embedded in XQuery directly

```
<element att1="v1" ...>...</element>
```

- Can “antiquote” to embed XQuery expressions in elements

```
<element>{$x/a/b}</element>
```

- Can explicitly construct elements (with arbitrary names, attributes

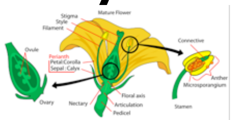
```
element $foo { attribute {$bar} {$baz},
               text {$some_text}}
```

Building sequences

- Empty sequence: ()
  - like empty list in other languages
- Sequence concatenation: (e_1,e_2)
  - evaluates e_1, e_2 to value sequences vs_1,vs_2
  - concatenates vs_1 and vs_2
- Examples: (expression equivalence)
  - (1,2,()) ≡ (1,2) ≡ ((),1,2) ≡ (1,(),2)
  - (1,(2,3)) ≡ (1,2,3) ≡ ((1,2),3)
  - ((1,2), (3,4)) ≡ (1,2,3,4)

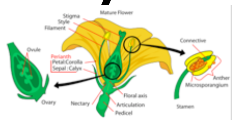
Reminder:
Next review assignment due:
Monday (Jan 28) 4pm
Electronic handin only!

Anatomy of a query: FLWOR



`for $x in ...xpath...`	iterates over items in sequence
`let $y := ...expression...`	binds variable to expression
`where ...condition...`	filters results based on boolean test
`order by ...ordering...`	orders results by key value
`return ...expression...`	constructs return values

Anatomy of a query: FLWOR



`for $x in ...xpath...`	iterates over items in sequence
`let $y := ...expr`	...to
`where ...conditi`	...ed on
`order by ...ordering...`	...results by key value
`return ...expression...`	constructs return values

Essentially list comprehensions (see also Haskell, Python, ...)

For / comprehension

```
for $x in ...xpath...
```

- Evaluates xpath to a sequence
  - actually can be any expression
- Generates one binding of $x for each element
- Evaluate rest of query once for each $x-binding
- Concatenate results in order

Let binding

```
let $y := ...expression...
```

- Evaluates expression to value
- Binds $x to the value
- Evaluates rest of query with new binding

Where clause

```
where ...condition...
```

- Evaluates condition expression to (Boolean) value
- If true, continue evaluating query
- If false, rest of query evaluates to ()
  - i.e., filters out results that don't satisfy condition

Order by

```
order by ...ordering...
```

- Orders results of rest of query by key
- Key specification is defined in terms of values available so far
- can specify increasing or decreasing
  - many other options

Return

`return ...`*`expression`*`...`

- Ends current iteration of query and generates result for it
  - unless filtered out by where-clause earlier
- Evaluates expression under current bindings

Let vs. for

- Both bind variables

```
let $x := (1,2,3)
let $y := ("a","b")
return ($x,$y)
...
```

(1,2,3,"a","b")

Let vs. for

- Both bind variables

```
for $x in (1,2,3)
let $y := ("a","b")
return ($x,$y)
...
```

(1,"a","b",2,"a","b",3,"a","b")

Let vs. for

- Both bind variables

```
let $x := (1,2,3)
for $y in ("a","b")
return ($x,$y)
...
```

(1,2,3,"a",1,2,3,"b")

Let vs. for

- Both bind variables

```
for $x in (1,2,3)
for $y in ("a","b")
return ($x,$y)
...
(1,"a",1,"b",2,"a",2,"b",3,"a",3,"b")
```


Putting it all together

- A join: pairs of books having author in common, ordered by year of first one

```
let $books := document("books.xml")/books
for $x in $books/book, $y in $books/book
let $year := $x/year/text()
where $x/author/text() = $y/author/text()
order by $year
return <result>{$x},{$y}</result>
```


Evaluating a join naively

- Iterates over all pairs of $x,$y
- Evaluates test
- Generates result for each pair satisfying test
- Problem: Quadratic.
  - Can do better using hash or sort join algorithms
  - Especially for large data
- XML databases can do this
- **Unordered** mode helps

Conditionals

```
if ...test... then ... else ...
if ...test... then ...
```

- Evaluate test
  - if true, evaluate then-branch
  - if false, evaluate else-branch
  - or () if no else-branch specified

Built-in functions

- Includes all XPath primitive functions
  - `first(), last(), position(), not(),` etc.
- equality: has same (strange) semantics as in XPath
  - i.e., `(1,2) = (2,3)` evaluates to true
- Also `document(<xmlfile>)`
  - loads in an XML file and binds it to a value

Set operations

- These are also allowed in XPath 2.0
- Union `e1 union e2`:
  - Path[p union p′](T) = Path[p](T) ∪ Path[p′](T)
- Intersection `e1 intersect e2`:
  - Path[p intersect p′](T) = Path[p](T) ∩ Path[p′](T)
- Difference `e1 except e2`:
  - Path[p except p′](T) = Path[p](T) ∖ Path[p′](T)

Aggregation and emptiness tests

`sum(), average(), min(), max(), count()`

- calculate corresponding functions on numerical sequences (like in SQL)
  - (can also use in XPath)

`empty(), exists()`

- test whether a sequence is empty or nonempty

Quantifiers

`some $x in ...exp1... satisfies ...exp2...`

- true iff `exp2` evaluates to true for **some** bindings of `$x` to element of `exp1`
  - `exists(for $x in p where q return <z/>)`

`every $x in ...exp1... satisfies ...exp2...`

- true iff `exp2` evaluates to true for **all** bindings of `$x` to element of `exp1`
  - `empty(for $x in p where not(q) return <z/>)`

Ordering & duplicates

- XQuery values are ordered sequences
- Can turn ordering off: `unordered {...}`
  - which enables more optimizations
- Or require it: `ordered {...}`
- Can also eliminate duplicates
  - `fn:remove-duplicates()`
- This happens automatically with some operations
  - such as `union`

Quiz

- Starting with XML that lists cities, states and **optional** nicknames:

```
<cities><city><name>New York City</name>
              <state>NY</state>
              <nickname>The Big Apple</nickname>
        </city> ...
</cities>
```

1. Write query that **ignores** state and lists city **by nickname if any**; otherwise **uses the name**
2. Write query that produces a list of states, each containing a list of city names in that state.
3. ... And gives a count of the number of cities in each state.

User-definable functions

- Can define functions to abbreviate parts of queries

```
define function f($x,$y) {
  for $z in $x/a, $w in $y/b
  where $z/text() = $y/text()
  return <result>{$z}{$w}</result>
}
```


Functions can be recursive!

- example: recursive parts query

```
define function totalcost($x) {
    for $y in $x/part
    return $x/price + totalcost($y)
}
```

Turing-completeness

- Due to recursive functions, XQuery is a fully Turing-complete language
  - even without arithmetic
  - can simulate tape, arithmetic using trees
  - Big contrast to SQL, which lacks general recursion
- Can write whole Web applications using XQuery + web server interface library
- In practice, XQuery engines focus optimization effort on FLWOR queries

Types

- XQuery has a native regular expression-based type system
  - Basic idea: if `$x : element {(a,(b,a)*,c)}`
  - then `for $y in $x return $y : (a|b|c)*`
- We will cover types and regular expression typing in more detail later
  - including XML Document Type Definitions, XML Schemas
  - and more precise systems for path/query typing

Semantics

- XQuery Formal Semantics
  - uses operational rules to explain meaning of XQuery expressions
  - Also formalizes typing rules
- Will also look at this in more detail later
  - needed for proving correctness of type systems, optimizations

Next time

- XSLT
- Type systems, XML DTDs
- Review assignment (due **Monday 4pm**):
  - XSLT overview
  - Read about XML Schemas
  - Read "Keys for XML"