XML and Databases

Lecture 7
Efficient XPath Evaluation

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2009

Outline

1. Top-Down Evaluation of *Simple Paths*

Simple paths are of the form

```
(1)  //tag_1/tag_2/…/tag_n
(2)  //tag_1/tag_2/…/tag_{n-1}/text()
```

Selects any node which is (1) labeled `tag_n` (2) a text node and

is child of a node labeled `tag_{n-1}`

is child of a node labeled `tag_{n-2}`

…

is child of a node labeled `tag_1`

Examples

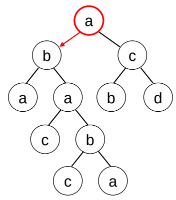
`//author/last` = select all last names of authors

`//strip/characters/character/text()` = select all character names from a DilbertML document

(return selected nodes in document order..)

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack**)



`//a/b` =Q

query match position: p = 1

[startElement(a)]

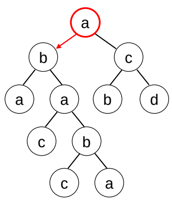
=Q[1]

Thus, p=p+1=2

→ *partial match*. If element name was different from "a", then p would remain equal to 1)

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack**)



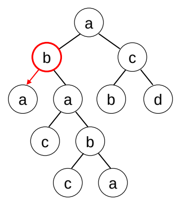
`//a/b` =Q

query match position: p = 2

[startElement(a)]

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack**)



`//a/b` =Q

query match position: p = 2

[startElement(a)]
[startElement(b)]

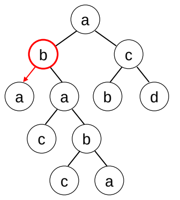
=Q[2]

p=2=length(Q), thus,
current node is a **match**!
→ Mark it as match/result
→ **push**(p)
→ p = 1

Push current match position p
for every startElement
(except for the root node)

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



//a/b =Q

query match position: p = 2

[startElement(a)]
[startElement(b)]

=Q[2]

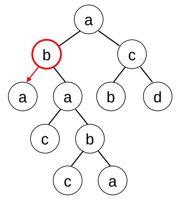
p=2=length(Q), thus,
current node is a **match**!
→ Mark it as match/result
→ **push**(p)
→ p = 1

Push current match position p
for every startElement
(except for the root node)

→ after closing endElement() we need to be in position p! (to match *next-sibling*)

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



//a/b =Q

query match position: p = 2

[startElement(a)]
[startElement(b)]

=Q[2]

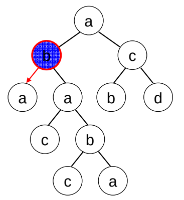
p=2=length(Q), thus,
current node is a **match**!
→ Mark it as match/result
→ **push**(p)
→ p = 1

Push current match position
for every startElement
(except for the root node)

Question Why is p set to 1? What if query was //a/a?

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



2

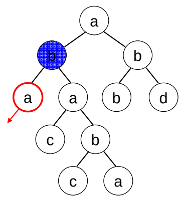
//a/b =Q

query match position: p = 1

[startElement(a)]
[startElement(b)]

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



2

//a/b =Q

query match position: p = 1

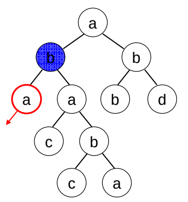
[startElement(a)]
[startElement(b)]
[startElement(a)]

=Q[1]

Thus, **push**(p)
and p=p+1=2

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



1
2

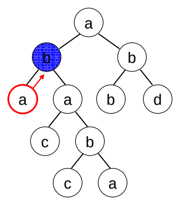
//a/b =Q

query match position: p = 2

[startElement(a)]
[startElement(b)]
[startElement(a)]

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



1
2

//a/b =Q

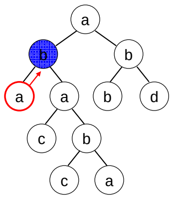
query match position: p = 2

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]

p = pop()

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



2

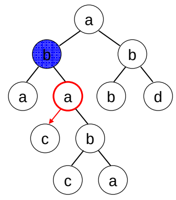
//a/b =Q

query match position: p = 1

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



2

//a/b =Q

query match position: p = 1

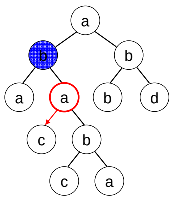
[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]

=Q[1]

Thus, **push**(p)
and p=p+1=2

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



1
2

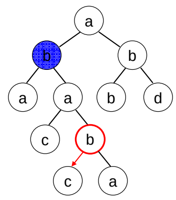
//a/b =Q

query match position: p = 2

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



1
2

//a/b =Q

query match position: p = 2

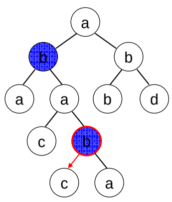
[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]
[startElement(c)] push(2)
[endElement(c)] p = pop() = 2
[startElement(b)]

=Q[2]

p=2=length(Q), thus,
current node is a **match**!
→ Mark it as match/result
→ **push**(p)
→ p = 1

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



2
1
2

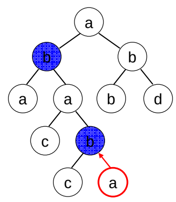
//a/b =Q

query match position: p = 1

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]
[startElement(c)]
[endElement(c)]
[startElement(b)]

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack)**



2
1
2

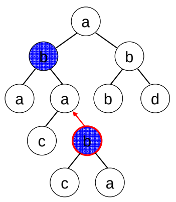
//a/b =Q

query match position: p = 1

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]
[startElement(c)]
[endElement(c)]
[startElement(b)]
[startElement(c)] push(1)
[endElement(c)] p = pop() = 1
[startElement(a)] push(1)
[endElement(a)] p = pop() = 1

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack**)



`//a/b =Q`

query match position: p = 2

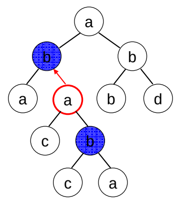
Stack: 2, 1, 2

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]
[startElement(c)]
[endElement(c)]
[startElement(b)]
[startElement(c)] push(1)
[endElement(c)] p = pop() = 1
[startElement(a)] push(1)
[endElement(a)] p = pop() = 1

[endElement(b)] p = pop() = 2

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack**)



`//a/b =Q`

query match position: p = 1

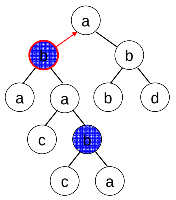
Stack: 1, 2

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]
[startElement(c)]
[endElement(c)]
[startElement(b)]
[startElement(c)] push(1)
[endElement(c)] p = pop() = 1
[startElement(a)] push(1)
[endElement(a)] p = pop() = 1

[endElement(b)] p = pop() = 2
[endElement(a)] p = pop() = 1

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack**)



`//a/b =Q`

query match position: p = 2

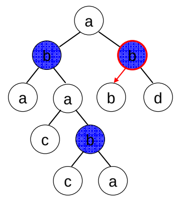
Stack: 2

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]
[startElement(c)]
[endElement(c)]
[startElement(b)]
[startElement(c)] push(1)
[endElement(c)] p = pop() = 1
[startElement(a)] push(1)
[endElement(a)] p = pop() = 1

[endElement(b)] p = pop() = 2
[endElement(a)] p = pop() = 1
[endElement(b)] p = pop() = 2

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack**)



`//a/b =Q`

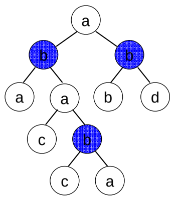
query match position: p = 2

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]
[startElement(c)]
[endElement(c)]
[startElement(b)]
[startElement(c)] push(1)
[endElement(c)] p = pop() = 1
[startElement(a)] push(1)
[endElement(a)] p = pop() = 1

[endElement(b)] p = pop() = 2
[endElement(a)] p = pop() = 1
[endElement(b)] p = pop() = 2
[startElement(b)] → **match**

1. Top-Down Evaluation of *Simple Paths*

→ evaluate in *one single pre-order traversal* (using a **stack**)



`//a/b =Q`

Linear time: O(#Nodes)
= O(|D|)
Size of document

Even: Streaming Algorithm! ☺

→ No need to store the document!!
Can evaluate on SAX event stream.

But, to print result subtrees we need an output buffer ☹

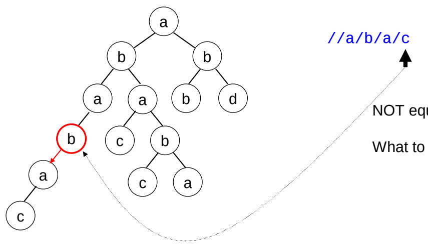
SAX-based path query evaluation (sketch):

1. Preparation:
   - Represent path query `//`t_1`/`t_2`/.../`t_{n-1}`/text()` via the *step array* $path[0] = t_1, path[1] = t_2, \ldots, path[n-1] =$ `text()`.
   - Maintain an array index $i = 0 \ldots n$, the *current step* in the path.
   - Maintain a stack S of index positions.
2. [*startDocument*]
   Empty stack S. We start with the first step.
3. [*startElement*]
   If the current step's tag name $path[i]$ and the reported tag name match, proceed to next step. Otherwise make a failure transition[14]. Remember how far we have come already: **push** the current step i onto S.
4. [*endElement*]
   The parser ascended to a parent element. Resume path traversal from where we have left earlier: **pop** old i from S.
5. [*characters*]
   If the current step $path[i] =$ `text()` we have found a match. Otherwise do nothing.

[14] This "Knuth-Morris-Pratt failure function" *fail*[] is to be explained in the tutorial.

1. Top-Down Evaluation of *Simple Paths*

→ evaluate using *one single pre-order traversal*! (using a **stack)**



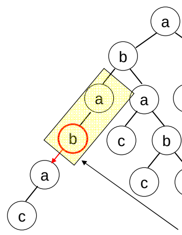
`//a/b/a/c`

NOT equal. p=4

What to do next?

1. Top-Down Evaluation of *Simple Paths*

→ evaluate using *one single pre-order traversal*! (using a **stack)**

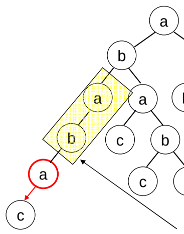


`//a/b/a/c`

Postfix (ending) of what we have seen,
is *prefix* (beginning) of the query!!

1. Top-Down Evaluation of *Simple Paths*

→ evaluate using *one single pre-order traversal*! (using a **stack)**



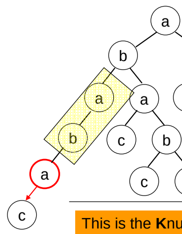
`//a/b/a/c`

Jump to position 3 for next node!

Postfix (ending) of what we have seen,
is *prefix* (beginning) of the query!!

1. Top-Down Evaluation of *Simple Paths*

→ evaluate using *one single pre-order traversal*! (using a **stack)**



`//a/b/a/c`

Jump to position 3 for next node!

Postfix (ending) of what we have seen,
is *prefix* (beginning) of the query!!

This is the **K**nuth-**M**orris-**P**ratt Algorithm! (KMP)

On match failure, query-position must be updated to longest prefix in Q that is postfix of what we have seen!

Linear time string matching

"jump-back-table": (preprocessing)
for each position in Q and **fail** (ing symbol), determine jump-back-position

2. Core XPath

Types
- Node Sets
- Booleans

→ all 12 axes

→ all node tests (but, here,
we will simply talk about *element nodes only*)

→ **filters** with logical operations: **and, or, not**

E.g. //descendant::a/child::b**[** child::c/child::d **or not(**following::*) **]**

Full XPath additionally has

→ Node set comparisons & operations (e.g., =, count)
→ Order functions (first, last, position)
→ Numerical operations (sum, +, -, *, div, mod, round, etc.)
and corresponding comparisons (=, !=, <, >, <=, >=)
→ String operations (contains, starts-with, translate, string-length, etc.)

2. Core XPath

Types
- Node Sets
- Booleans

→ all 12 axes

→ all node tests (but, here,
we will simply talk about *element nodes only*)

→ **filters** with logical operations: **and, or, not**

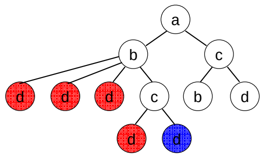
E.g. //descendant::a/child::b**[** child::c/child::d **or not(**following::*) **]**

Thus in Core XPath

→ Select nodes only depending on labels.
No counting. No values.

Axis Evaluation

Axis = **Node Set** (evaluated relative to context-node)



Node Set represented as *bit-field*
1|2|3|4|5|6|7|8|9|10|11
0|0|1|1|1|0|1|0|0|0 |0

Axis Evaluation
Maps a Node Set to a Node Set

Forward Axes:
→ `self`
→ `child`
→ `descendant-or-self`
→ `descendant`
→ `following`
→ `following-sibling`

Backward Axes:
→ `parent`
→ `ancestor`
→ `ancestor-or-self`
→ **`preceding`**
→ `preceding-sibling`

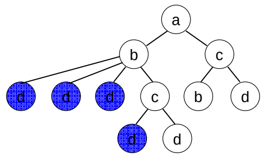
→ `attribute`

reverse doc order

In doc order

Axis Evaluation

Axis = **Node Set** (evaluated relative to context-node)



Node Set represented as *bit-field*
1|2|3|4|5|6|7|8|9|10|11
0|0|1|1|1|0|1|0|0|0 |0

Axis Evaluation
Maps a Node Set to a Node Set

Naïve:
Node-Set = { 3, 4, 5, 7 }

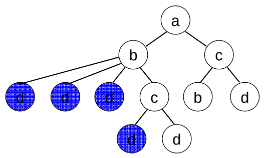
axis(Node-Set) = axis(3) ← time linear in #Nodes
∪ axis(4) ← time linear in #Nodes
∪ axis(5) ← time linear in #Nodes
∪ axis(7) ← time linear in #Nodes

at most #Nodes many

O(#Nodes #Nodes) =quadratic time ☹

Axis Evaluation

Axis = **Node Set** (evaluated relative to context-node)



Node Set represented as *bit-field*
1|2|3|4|5|6|7|8|9|10|11
0|0|1|1|1|0|1|0|0|0 |0

Axis Evaluation
Maps a Node Set to a Node Set

Can be done for ANY axis in
linear time wrt number of nodes
if we have *constant time look-up* for
• first-child(node), parent(node)
• next-sibling(node), previous-sibling(node)

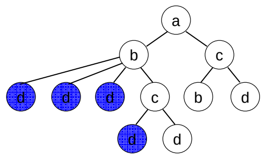
(for linear time forward axis evaluation, enough to have first-child/next-sibling.)

Forward Axes:
→ `self`
→ `child`
→ `descendant-or-self`
→ `descendant`
→ `following`
→ `following-sibling`

In doc order

Axis Evaluation

Axis = **Node Set** (evaluated relative to context-node)



Node Set represented as *bit-field*
1|2|3|4|5|6|7|8|9|10|11
0|0|1|1|1|0|1|0|0|0 |0

Axis Evaluation
Maps a Node Set to a Node Set

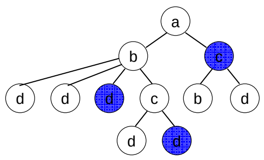
Idea
→ No node is visited >once!

Can be done for ANY axis in
linear time wrt number of nodes
if we have *constant time look-up* for
• first-child(node), parent(node)
• next-sibling(node), previous-sibling(node)

(for linear time forward axis evaluation, enough to have first-child/next-sibling.)

Axis Evaluation

Axis = **Node Set** (evaluated relative to context-node)



Node Set represented as *bit-field*
1|2|3|4|5|6|7|8|9|10|11
0|0|0|0|1|0|0|1|1|0 |0

Axis Evaluation
Maps a Node Set to a Node Set

Idea
→ No node is visited >once!

e.g.: ancestor({5, 8, 9})

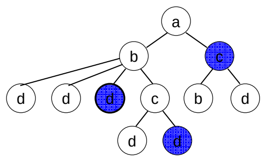
look-up parents, check if we are in result set already..

Can be done for ANY axis in
linear time wrt number of nodes
if we have *constant time look-up* for
• first-child(node), parent(node)
• next-sibling(node), previous-sibling(node)

(for linear time forward axis evaluation, enough to have first-child/next-sibling.)

Axis Evaluation

Axis = **Node Set** (evaluated relative to context-node)



Node Set
1|2|3|4|5|6|7|8|9|10|11
0|0|0|0|1|0|0|1|1|0 |0

parent
1|2|3|4|5|6|7|8|9|10|11
-|1|2|2|2|2|6|6|1|9 |9

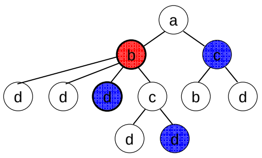
Idea
→ No node is visited >once!

e.g.: ancestor({5, 8, 9})

look-up parents, check if we are in result set already..

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)



Node Set
1|2|3|4|5|6|7|8|9|10|11
0|0|0|0|1|0|0|1|1|0 |0

parent
1|2|3|4|5|6|7|8|9|10|11
-|1|2|2|2|2|6|6|1|9 |9

Idea
→ No node is visited >once!

e.g.: ancestor({5, 8, 9})

look-up parents, check if we are in result set already..

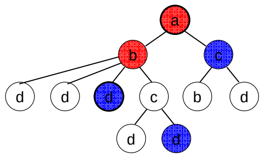
Result Node Set
1|2|3|4|5|6|7|8|9|10|11
0|1|0|0|0|0|0|0|0|0 |0

After **one** parent look-up.

37

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)



Node Set
1|2|3|4|5|6|7|8|9|10|11
0|0|0|0|1|0|0|1|1|0 |0

parent
1|2|3|4|5|6|7|8|9|10|11
-|1|2|2|2|2|6|6|1|9 |9

Idea
→ No node is visited >once!

e.g.: ancestor({5, 8, 9})

look-up parents, check if we are in result set already..

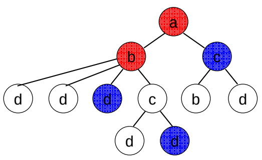
Result Node Set
1|2|3|4|5|6|7|8|9|10|11
1|1|0|0|0|0|0|0|0|0 |0

After **2** parent look-ups.

38

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)



Node Set
1|2|3|4|5|6|7|8|9|10|11
0|0|0|0|1|0|0|1|1|0 |0

parent
1|2|3|4|5|6|7|8|9|10|11
-|1|2|2|2|2|6|6|1|9 |9

move to next node in Node Set

Idea
→ No node is visited >once!

e.g.: ancestor({5, 8, 9})

look-up parents, check if we are in result set already..

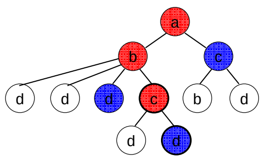
Result Node Set
1|2|3|4|5|6|7|8|9|10|11
1|1|0|0|0|0|0|0|0|0 |0

After **3** parent look-ups.

39

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)



Node Set
1|2|3|4|5|6|7|8|9|10|11
0|0|0|0|1|0|0|1|1|0 |0

parent
1|2|3|4|5|6|7|8|9|10|11
-|1|2|2|2|2|6|6|1|9 |9

Idea
→ No node is visited >once!

e.g.: ancestor({5, 8, 9})

look-up parents, check if we are in result set already..

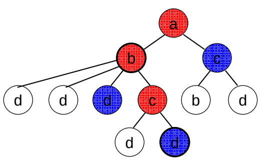
Result Node Set
1|2|3|4|5|6|7|8|9|10|11
1|1|0|0|0|1|0|0|0|0 |0

After **4** parent look-ups.

40

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)



Node Set
1|2|3|4|5|6|7|8|9|10|11
0|0|0|0|1|0|0|1|1|0 |0

parent
1|2|3|4|5|6|7|8|9|10|11
-|1|2|2|2|2|6|6|1|9 |9

already in result set

move to next node in Node Set

Idea
→ No node is visited >once!

e.g.: ancestor({5, 8, 9})

look-up parents, check if we are in result set already..

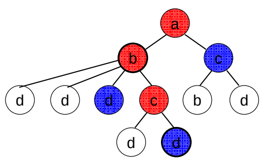
Result Node Set
1|**2**|3|4|5|6|7|8|9|10|11
1|**1**|0|0|0|1|0|0|0|0 |0

After **5** parent look-ups.

41

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)



Node Set
1|2|3|4|5|6|7|8|9|10|11
0|0|0|0|1|0|0|1|1|0 |0

parent
1|2|3|4|5|6|7|8|9|10|11
-|1|2|2|2|2|6|6|1|9 |9

No next node in Node Set

already in result set

Finished!

Idea
→ No node is visited >once!

e.g.: ancestor({5, 8, 9})

look-up parents, check if we are in result set already..

Result Node Set
1|2|3|4|5|6|7|8|9|10|11
1|1|0|0|0|1|0|0|0|0 |0

After **6** parent look-ups.
\+ **5** result look-ups.

42

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)

Similarly:
For all other axes!

Forward-axes only:
binary (top-down) tree encoding
provides easy
linear time evaluation!

+backward axes?

Question
do you see how this works for
e.g., descendant axis?

Recall:
to access parent / ancestors on
binary tree, keep dynamically
list of all ancestors.

Idea
→ No node is visited >once!

e.g.: ancestor({5, 8, 9})

look-up parents, check if we are
in result set already..

Result Node Set
1|2|3|4|5|6|7|8|9|10|11
1|1|0|0|0|1|0|0|0|0 |0

After **6** parent look-ups.
\+ **5** result look-ups.

Axis Evaluation

Axis = Node Set (evaluated relative to context-node)

Question
do you see how this works for
e.g., descendant axis?

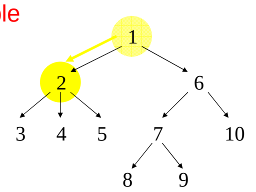
descendant(node) = (first-child | next-sibling)* (first-child(node))

descendant($\underbrace{\{\text{ node\_1, node\_2, …, node\_k }\}}_{S}$) =

```
repeat{
  pick node N in S;
  (for N's descendants M in pre-order)
    {
      if (not(M in result set))
         add(M to result set) else break;
    }
}
```


Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



Node Set
1|2|3|4|5|6|7|8|9|10
1|0|0|0|0|0|0|0|0|0

descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +

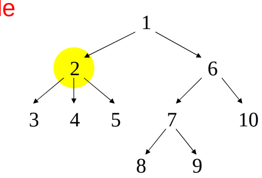
fc	
1	2
2	3
6	7
7	8

ns	
2	6
3	4
4	5
7	10
8	9

This example comes from
Georg Gottlob and Christoph Koch "XPath Query Processing".
Invited tutorial at DBPL **2003**
http://www.dbai.tuwien.ac.at/research/xmlaskforce/xpath-tutorial1.ppt.gz

Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +

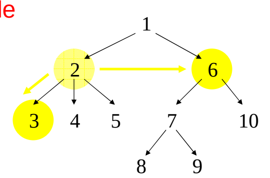
Result Node Set
1|2|3|4|5|6|7|8|9|10
0|1|0|0|0|0|0|0|0|0

fc	
1	2
2	3
6	7
7	8

ns	
2	6
3	4
4	5
7	10
8	9

Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +
{ 3, 6 } +

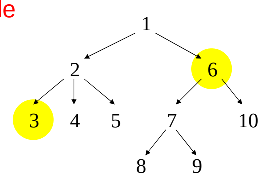
Result Node Set
1|2|3|4|5|6|7|8|9|10
0|1|1|0|0|1|0|0|0|0

fc	
1	2
2	3
6	7
7	8

ns	
2	6
3	4
4	5
7	10
8	9

Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +
{ 3, 6 } +

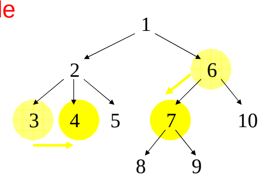
Result Node Set
1|2|3|4|5|6|7|8|9|10
0|1|1|0|0|1|0|0|0|0

fc	
1	2
2	3
6	7
7	8

ns	
2	6
3	4
4	5
7	10
8	9

Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +
{ 3, 6 } +
{ 4, 7 } +

Result Node Set
1|2|3|4|5|6|7|8|9|10
0|1|1|1|0|1|1|0|0|0

fc

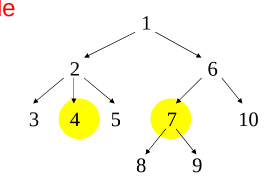
1	2
2	3
6	7
7	8

ns

2	6
3	4
4	5
7	10
8	9

Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +
{ 3, 6 } +
{ 4, 7 } +

Result Node Set
1|2|3|4|5|6|7|8|9|10
0|1|1|1|0|1|1|0|0|0

fc

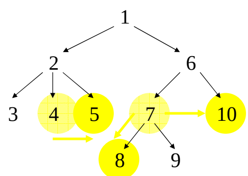
1	2
2	3
6	7
7	8

ns

2	6
3	4
4	5
7	10
8	9

Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +
{ 3, 6 } +
{ 4, 7 } +
{ 5, 8, 10 } +

Result Node Set
1|2|3|4|5|6|7|8|9|10
0|1|1|1|1|1|1|1|0|1

fc

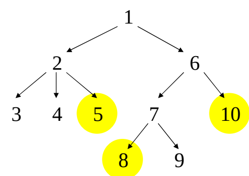
1	2
2	3
6	7
7	8

ns

2	6
3	4
4	5
7	10
8	9

Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +
{ 3, 6 } +
{ 4, 7 } +
{ 5, 8, 10 } +

Result Node Set
1|2|3|4|5|6|7|8|9|10
0|1|1|1|1|1|1|1|0|1

fc

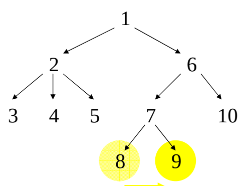
1	2
2	3
6	7
7	8

ns

2	6
3	4
4	5
7	10
8	9

Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +
{ 3, 6 } +
{ 4, 7 } +
{ 5, 8, 10 } +
{ 9 }

Result Node Set
1|2|3|4|5|6|7|8|9|10
0|1|1|1|1|1|1|1|1|1

fc

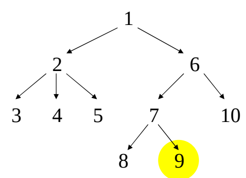
1	2
2	3
6	7
7	8

ns

2	6
3	4
4	5
7	10
8	9

Example

descendant(node) =
(first-child | next-sibling)* (first-child(node))



descendant({ 1 }) =

(fc | ns)*(first-child({ 1 })) =

(fc | ns)*({ 2 }) =

{ 2 } +
{ 3, 6 } +
{ 4, 7 } +
{ 5, 8, 10 } +
{ 9 }

Result Node Set
1|2|3|4|5|6|7|8|9|10
0|1|1|1|1|1|1|1|1|1

fc

1	2
2	3
6	7
7	8

ns

2	6
3	4
4	5
7	10
8	9

Core XPath

Types
- Node Sets
- Booleans

→ all 12 axes

→ all node tests (but, here, we will simply talk about *element nodes only*)

→ **filters** with logical operations: **and, or, not**

E.g. //descendant::a/child::b[child::c/child::d **or not(**following::*)]

➔ For Core XPath we only need Node Set operations!!

- axis(Set1) = Set2
- ∪(Set1, Set2) = Set3 union of Set1 and Set2
- ∩(Set1, Set2) = Set3 intersection of Set1 and Set2
- -(Set1, Set2) = Set3 everything in Set1 but not in Set2
- lab(a) = Set1 all nodes labeled by a

55

3. Bottom-Up Evaluation of Core XPath

With respect to query-tree (parse tree)

NOT with respect to document tree!!

(algorithm f. simple paths is top-down wrt document tree)

56

3. Bottom-Up Evaluation of Core XPath

Axis evaluation: O(#Nodes) = O(|D|)
Size of the Document

Node Set operation: O(|D|)

Size of the Query (= #steps)

Core XPath query Q can be evaluated in time O(|Q| |D|)

linear time! ☺

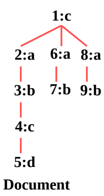
➔ For Core XPath we only need Node Set operations!!

- axis(Set1) = Set2 — used for **or**'s
- ∪(Set1, Set2) = Set3 union of Set1 and Set2
- ∩(Set1, Set2) = Set3 intersection of Set1 and Set2 — used for **not**'s
- -(Set1, Set2) = Set3 everything in Set1 but not in Set2
- lab(a) = Set1 all nodes labeled by a — for node tests

for everything else (steps, filters)

57

//descendant::a/child::b[child::c/child::d **or not(**following::*)]



becomes

∩(descendant({root}), lab(a))

axis context-nodes node test

lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }

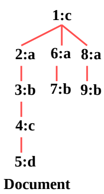
➔ For Core XPath we only need Node Set operations!!

- axis(Set1) = Set2 — used for **or**'s
- ∪(Set1, Set2) = Set3 union of Set1 and Set2
- ∩(Set1, Set2) = Set3 intersection of Set1 and Set2 — used for **not**'s
- -(Set1, Set2) = Set3 everything in Set1 but not in Set2
- lab(a) = Set1 all nodes labeled by a — for node tests

for everything else (steps, filters)

58

//descendant::a/child::b[child::c/child::d **or not(**following::*)]



becomes

∩(descendant({root}), lab(a)) = { 2, 6, 8 }

lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }

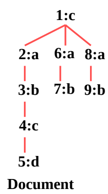
➔ For Core XPath we only need Node Set operations!!

- axis(Set1) = Set2 — used for **or**'s
- ∪(Set1, Set2) = Set3 union of Set1 and Set2
- ∩(Set1, Set2) = Set3 intersection of Set1 and Set2 — used for **not**'s
- -(Set1, Set2) = Set3 everything in Set1 but not in Set2
- lab(a) = Set1 all nodes labeled by a — for node tests

for everything else (steps, filters)

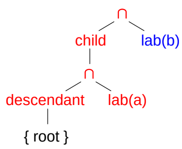
59

//descendant::a/child::b[child::c/child::d **or not(**following::*)]

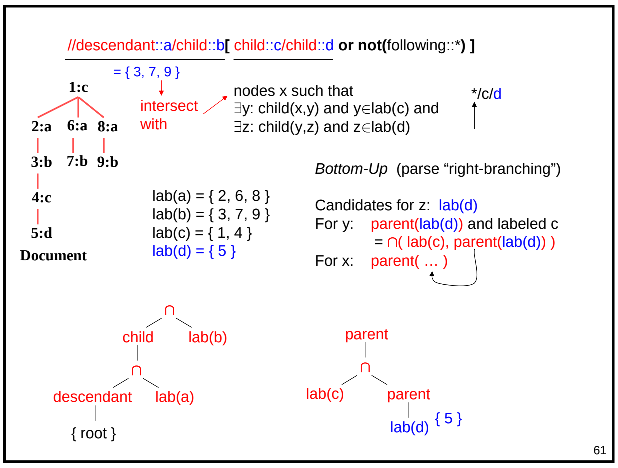


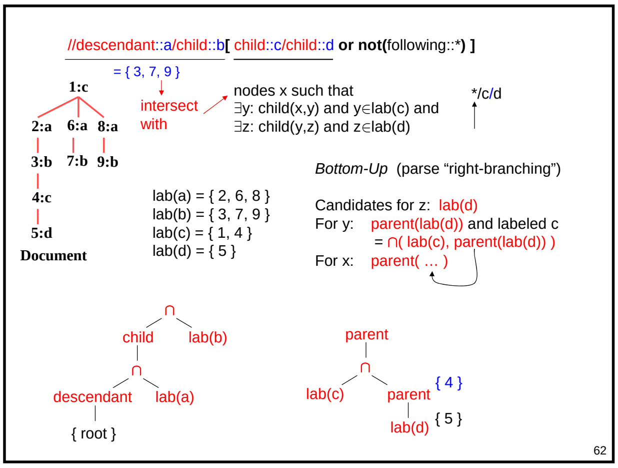
becomes — axis
∩(child(— context-nodes
∩(descendant({root}), lab(a)) = { 2, 6, 8 }),
lab(b)) = — node test
∩({ 3, 7, 9 }, { 3, 7, 9 }) = { 3, 7, 9 }

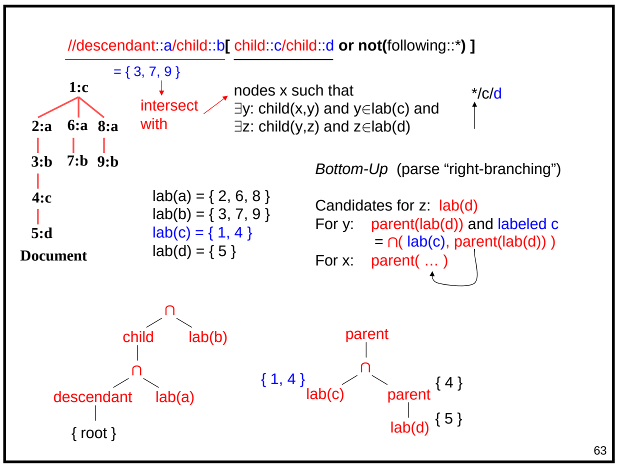
lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }

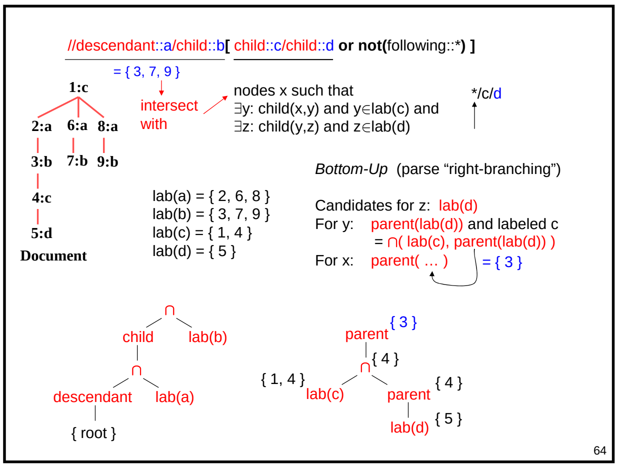


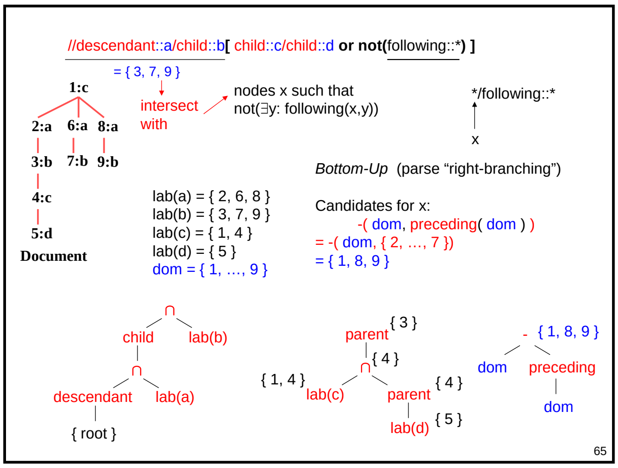
60

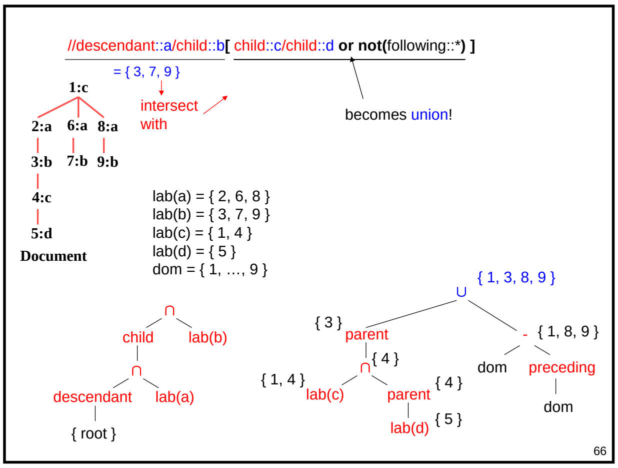

//descendant::a/child::b[child::c/child::d or not(following::*)]
= { 3, 7, 9 }
intersect with
nodes x such that
∃y: child(x,y) and y∈lab(c) and
∃z: child(y,z) and z∈lab(d)
*/c/d
Bottom-Up (parse "right-branching")
lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }
Candidates for z: lab(d)
For y: parent(lab(d)) and labeled c
= ∩(lab(c), parent(lab(d)))
For x: parent(…)
1:c 2:a 6:a 8:a 3:b 7:b 9:b 4:c 5:d
Document
child lab(b) descendant lab(a) { root }
parent lab(c) parent lab(d) { 5 }
61

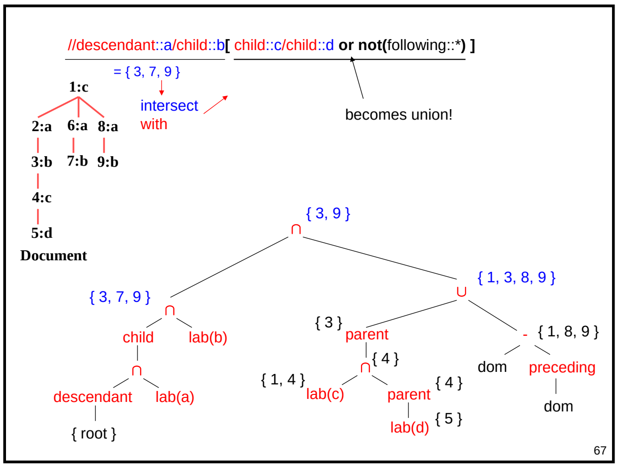

//descendant::a/child::b[child::c/child::d or not(following::*)]
= { 3, 7, 9 }
intersect with
nodes x such that
∃y: child(x,y) and y∈lab(c) and
∃z: child(y,z) and z∈lab(d)
*/c/d
Bottom-Up (parse "right-branching")
lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }
Candidates for z: lab(d)
For y: parent(lab(d)) and labeled c
= ∩(lab(c), parent(lab(d)))
For x: parent(…)
Document
child lab(b) descendant lab(a) { root }
parent lab(c) parent { 4 } lab(d) { 5 }
62

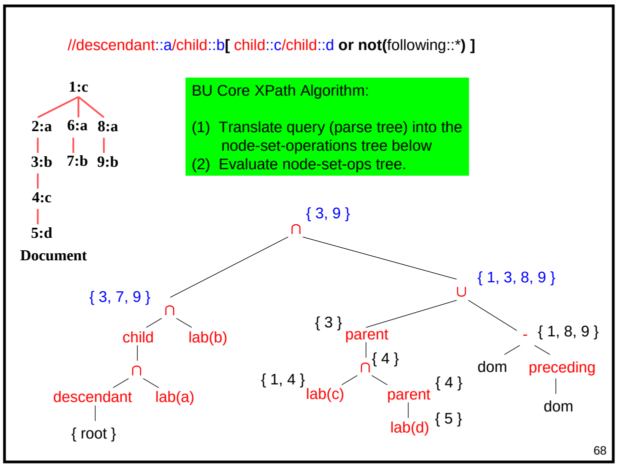

//descendant::a/child::b[child::c/child::d or not(following::*)]
= { 3, 7, 9 }
intersect with
nodes x such that
∃y: child(x,y) and y∈lab(c) and
∃z: child(y,z) and z∈lab(d)
*/c/d
Bottom-Up (parse "right-branching")
lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }
Candidates for z: lab(d)
For y: parent(lab(d)) and labeled c
= ∩(lab(c), parent(lab(d)))
For x: parent(…)
Document
child lab(b) descendant lab(a) { root }
parent { 1, 4 } lab(c) parent { 4 } lab(d) { 5 }
63

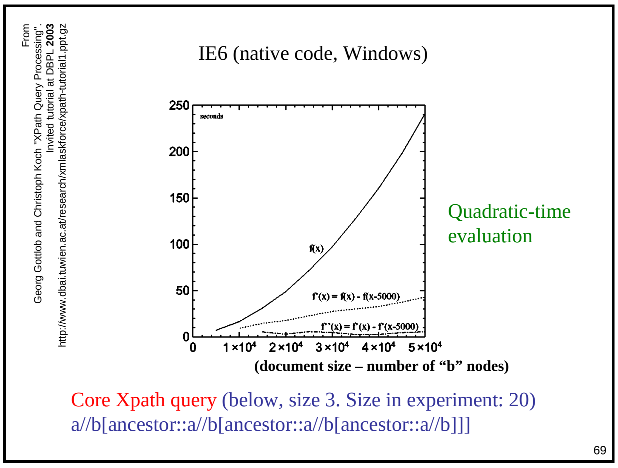

//descendant::a/child::b[child::c/child::d or not(following::*)]
= { 3, 7, 9 }
intersect with
nodes x such that
∃y: child(x,y) and y∈lab(c) and
∃z: child(y,z) and z∈lab(d)
*/c/d
Bottom-Up (parse "right-branching")
lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }
Candidates for z: lab(d)
For y: parent(lab(d)) and labeled c
= ∩(lab(c), parent(lab(d)))
For x: parent(…) = { 3 }
Document
child lab(b) descendant lab(a) { root }
parent { 3 } { 4 } { 1, 4 } lab(c) parent { 4 } lab(d) { 5 }
64

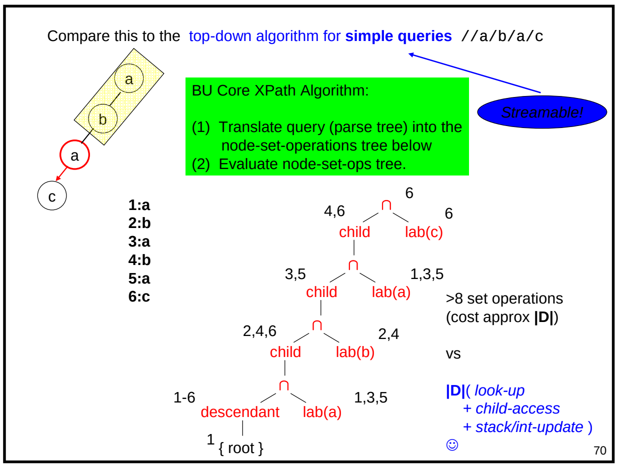

//descendant::a/child::b[child::c/child::d or not(following::*)]
= { 3, 7, 9 }
intersect with
nodes x such that
not(∃y: following(x,y))
/following::
x
Bottom-Up (parse "right-branching")
lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }
dom = { 1, …, 9 }
Candidates for x:
-(dom, preceding(dom))
= -(dom, { 2, …, 7 })
= { 1, 8, 9 }
Document
child lab(b) descendant lab(a) { root }
parent { 3 } { 4 } { 1, 4 } lab(c) parent { 4 } lab(d) { 5 }
- { 1, 8, 9 } dom preceding dom
65

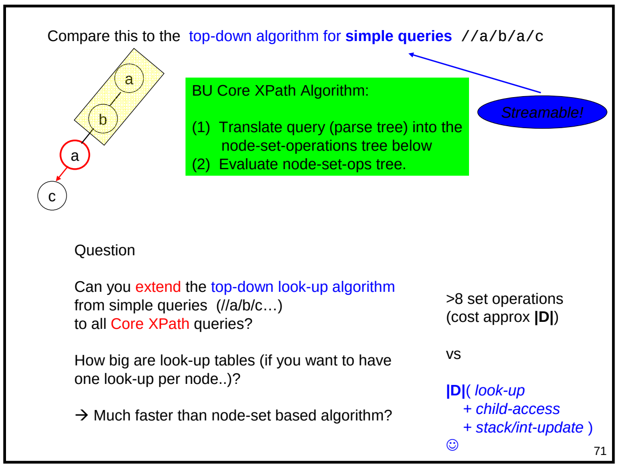

//descendant::a/child::b[child::c/child::d or not(following::*)]
= { 3, 7, 9 }
intersect with
becomes union!
lab(a) = { 2, 6, 8 }
lab(b) = { 3, 7, 9 }
lab(c) = { 1, 4 }
lab(d) = { 5 }
dom = { 1, …, 9 }
Document
∪ { 1, 3, 8, 9 }
child lab(b) descendant lab(a) { root }
{ 3 } parent { 4 } { 1, 4 } lab(c) parent { 4 } lab(d) { 5 }
- { 1, 8, 9 } dom preceding dom
66











4. Polynomial Time Evaluation of Full XPath

All following slides are taken from
Georg Gottlob and Christoph Koch "XPath Query Processing".
Invited tutorial at DBPL **2003**
http://www.dbai.tuwien.ac.at/research/xmlaskforce/xpath-tutorial1.ppt.gz

72

Contexts

XPath expressions are evaluated w.r.t. <u>Contexts</u>

context: <x, k, n>

node position size

These values specify a current "situation" in which a query or subquery should be evaluated.
Determined by preceding XSL or Xpath computations.

Previously computed node-set $\{n_1, n_3, n_5, n_9\}$

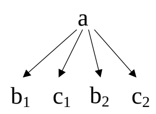
Continuation of computation $\langle n_5, 3, 4\rangle$

This is the context information used for the further query evaluation Starting at n_5

Example of an Xpath query not in Core XPath

Sample document *D*:

<a> <b/> <c/> <b/> <c/>



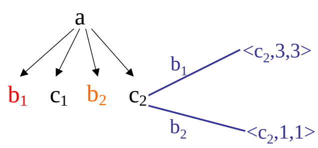
Sample query *Q*:

child::b/following::*[position() > 2]

Example of an Xpath query not in Core XPath

Sample document *D*:

<a> <b/> <c/> <b/> <c/>



Sample query *Q*:

child::b/following::*[position() > 2]

result node-sets S   **evaluated for each x∈ S, w.r.t context of x in S**

$\mathcal{E}_\uparrow$: Expression $\rightarrow$ nset $\cup$ num $\cup$ str $\cup$ bool

Expr. E : Operator Signature	**Semantics $\mathcal{E}_\uparrow[\![E]\!]$**
location step $\chi{::}t$: $\rightarrow$ nset	$\{\langle x_0, k_0, n_0, \{x \mid x_0 \chi x,\ x \in T(t)\}\rangle \mid \langle x_0, k_0, n_0\rangle \in \mathbf{C}\}$
location step $E[e]$ over axis χ: nset $\times$ bool $\rightarrow$ nset	$\{\langle x_0, k_0, n_0, \{x \in S \mid \langle x, \mathrm{idx}_\chi(x,S), \lvert S\rvert, \mathrm{true}\rangle \in \mathcal{E}_\uparrow[\![e]\!]\}\rangle \mid \langle x_0, k_0, n_0, S\rangle \in \mathcal{E}_\uparrow[\![E]\!]\}$
location path $/\pi$: nset $\rightarrow$ nset	$\mathbf{C} \times \{S \mid \exists k, n : \langle \mathrm{root}, k, n, S\rangle \in \mathcal{E}_\uparrow[\![\pi]\!]\}$
location path π_1/π_2 : nset $\times$ nset $\rightarrow$ nset	$\{\langle x, k, n, z\rangle \mid 1 \le k \le n \le \lvert\mathrm{dom}\rvert, \langle x, k_1, n_1, Y\rangle \in \mathcal{E}_\uparrow[\![\pi_1]\!], \bigcup_{y\in Y} \langle y, k_2, n_2, z\rangle \in \mathcal{E}_\uparrow[\![\pi_2]\!]\}$
location path $\pi_1 \mid \pi_2$: nset $\times$ nset $\rightarrow$ nset	$\mathcal{E}_\uparrow[\![\pi_1]\!] \cup \mathcal{E}_\uparrow[\![\pi_2]\!]$
position() : $\rightarrow$ num	$\{\langle x, k, n, k\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$
last() : $\rightarrow$ num	$\{\langle x, k, n, n\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$

$$\mathcal{E}_\uparrow[\![Op(e_1, \ldots, e_m)]\!] := \{\langle \vec{c}, \mathcal{F}[\![Op]\!](v_1, \ldots, v_m)\rangle \mid \vec{c} \in \mathbf{C}, \langle \vec{c}, v_1\rangle \in \mathcal{E}_\uparrow[\![e_1]\!], \ldots, \langle \vec{c}, v_m\rangle \in \mathcal{E}_\uparrow[\![e_m]\!]\}$$

Example: Formal Semantics of Xpath Relational Operators

$\mathcal{F}[\![RelOp : \mathrm{nset} \times \mathrm{nset} \rightarrow \mathrm{bool}]\!](S_1, S_2)$	$\exists n_1 \in S_1, n_2 \in S_2 : \mathrm{strval}(n_1)\ RelOp\ \mathrm{strval}(n_2)$
$\mathcal{F}[\![RelOp : \mathrm{nset} \times \mathrm{num} \rightarrow \mathrm{bool}]\!](S, v)$	$\exists n \in S : \mathrm{to\_number}(\mathrm{strval}(n))\ RelOp\ v$
$\mathcal{F}[\![RelOp : \mathrm{nset} \times \mathrm{str} \rightarrow \mathrm{bool}]\!](S, s)$	$\exists n \in S : \mathrm{strval}(n)\ RelOp\ s$
$\mathcal{F}[\![RelOp : \mathrm{nset} \times \mathrm{bool} \rightarrow \mathrm{bool}]\!](S, b)$	$\mathcal{F}[\![\mathrm{boolean}]\!](S)\ RelOp\ b$
$\mathcal{F}[\![EqOp : \mathrm{bool} \times (\mathrm{str} \cup \mathrm{num} \cup \mathrm{bool}) \rightarrow \mathrm{bool}]\!](b, x)$	$b\ EqOp\ \mathcal{F}[\![\mathrm{boolean}]\!](x)$
$\mathcal{F}[\![EqOp : \mathrm{num} \times (\mathrm{str} \cup \mathrm{num}) \rightarrow \mathrm{bool}]\!](v, x)$	$v\ EqOp\ \mathcal{F}[\![\mathrm{number}]\!](x)$
$\mathcal{F}[\![EqOp : \mathrm{str} \times \mathrm{str} \rightarrow \mathrm{bool}]\!](s_1, s_2)$	$s_1\ EqOp\ s_2$
$\mathcal{F}[\![GtOp : (\mathrm{str} \cup \mathrm{num} \cup \mathrm{bool}) \times (\mathrm{str} \cup \mathrm{num} \cup \mathrm{bool}) \rightarrow \mathrm{bool}]\!](x_1, x_2)$	$\mathcal{F}[\![\mathrm{number}]\!](x_1)\ GtOp\ \mathcal{F}[\![\mathrm{number}]\!](x_2)$

Std. Semantics of Location Paths

$P[\![\chi{::}t[e_1]\cdots[e_m]]\!](x) :=$
begin
$\quad S := \{y \mid x \chi y,\ y \in T(t)\};$
$\quad$**for** $1 \le i \le m$ (in ascending order) **do**
$\qquad S := \{y \in S \mid [\![e_i]\!](y, \mathrm{idx}_\chi(y, S), \lvert S\rvert) = \mathrm{true}\};$
$\quad$**return** S;
end;
$P[\![\pi_1 | \pi_2]\!](x) := P[\![\pi_1]\!](x) \cup P[\![\pi_2]\!](x)$
$P[\![/\pi]\!](x) := P[\![\pi]\!](\mathrm{root})$
$\boxed{P[\![\pi_1/\pi_2]\!](x) := \bigcup_{y \in P[\![\pi_1]\!](x)} P[\![\pi_2]\!](y)}$

First formal semantics of a relevant fragment of XPath: Phil Wadler 1999

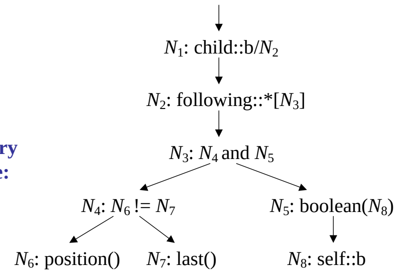
Context-value Tables (CVT)

- Four types of values (*nset, num, str, bool*)
- Defined for each XPath expression *e*
- The CVT of *e* is a relation $R \subseteq C \times$ (*nset* $\cup$ *num* $\cup$ *str* $\cup$ *bool*)

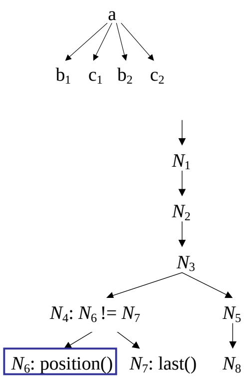
Parse Tree of the Query

Query:
child::b/following::*[position() != last() and self::b]

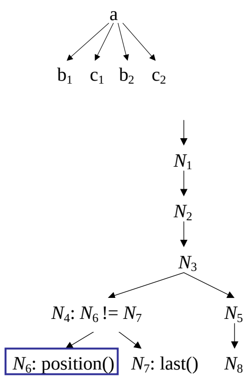
Query Tree:



<a> <b/> <c/> <b/> <c/>



<a> <b/> <c/> <b/> <c/>

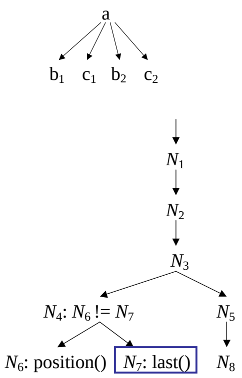


N_6: position()

cn	cp	cs	res
c_1	1	3	1
b_2	2	3	2
c_2	3	3	3
b_2	1	2	1
c_2	2	2	2
c_2	1	1	1

(In fact, this is only a relevant subset of the full tables.)

<a> <b/> <c/> <b/> <c/>



N_6: position()

cn	cp	cs	res
c_1	1	3	1
b_2	2	3	2
c_2	3	3	3
b_2	1	2	1
c_2	2	2	2
c_2	1	1	1

N_7: last()

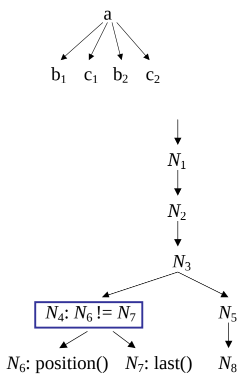
cn	cp	cs	res
c_1	1	3	3
b_2	2	3	3
c_2	3	3	3
b_2	1	2	2
c_2	2	2	2
c_2	1	1	1

(In fact, this is only a relevant subset of the full tables.)

<a> <b/> <c/> <b/> <c/>

N_4: N_6 != N_7

cn	cp	cs	res



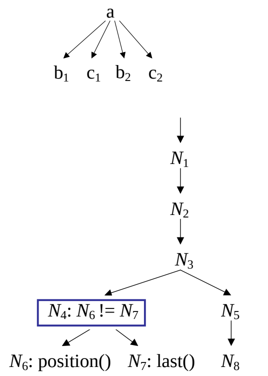
N_6: position()

cn	cp	cs	res
c_1	1	3	1
b_2	2	3	2
c_2	3	3	3
b_2	1	2	1
c_2	2	2	2
c_2	1	1	1

N_7: last()

cn	cp	cs	res
c_1	1	3	3
b_2	2	3	3
c_2	3	3	3
b_2	1	2	2
c_2	2	2	2
c_2	1	1	1

<a> <b/> <c/> <b/> <c/>



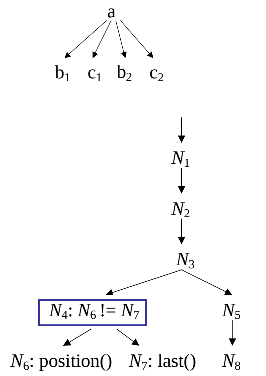
N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true

N_6: position()			
cn	cp	cs	res
c_1	1	3	1
b_2	2	3	2
c_2	3	3	3
b_2	1	2	1
c_2	2	2	2
c_2	1	1	1

N_7: last()			
cn	cp	cs	res
c_1	1	3	3
b_2	2	3	3
c_2	3	3	3
b_2	1	2	2
c_2	2	2	2
c_2	1	1	1

85

<a> <b/> <c/> <b/> <c/>



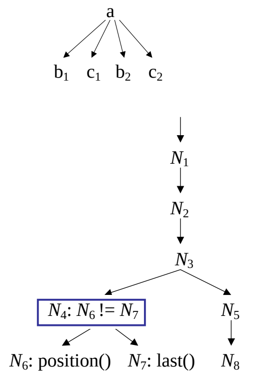
N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true

N_6: position()			
cn	cp	cs	res
c_1	1	3	1
b_2	2	3	2
c_2	3	3	3
b_2	1	2	1
c_2	2	2	2
c_2	1	1	1

N_7: last()			
cn	cp	cs	res
c_1	1	3	3
b_2	2	3	3
c_2	3	3	3
b_2	1	2	2
c_2	2	2	2
c_2	1	1	1

86

<a> <b/> <c/> <b/> <c/>



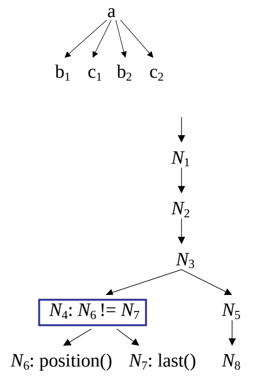
N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false

N_6: position()			
cn	cp	cs	res
c_1	1	3	1
b_2	2	3	2
c_2	3	3	3
b_2	1	2	1
c_2	2	2	2
c_2	1	1	1

N_7: last()			
cn	cp	cs	res
c_1	1	3	3
b_2	2	3	3
c_2	3	3	3
b_2	1	2	2
c_2	2	2	2
c_2	1	1	1

87

<a> <b/> <c/> <b/> <c/>



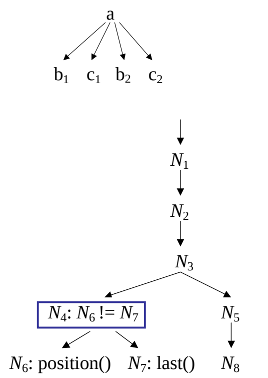
N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true

N_6: position()			
cn	cp	cs	res
c_1	1	3	1
b_2	2	3	2
c_2	3	3	3
b_2	1	2	1
c_2	2	2	2
c_2	1	1	1

N_7: last()			
cn	cp	cs	res
c_1	1	3	3
b_2	2	3	3
c_2	3	3	3
b_2	1	2	2
c_2	2	2	2
c_2	1	1	1

88

<a> <b/> <c/> <b/> <c/>



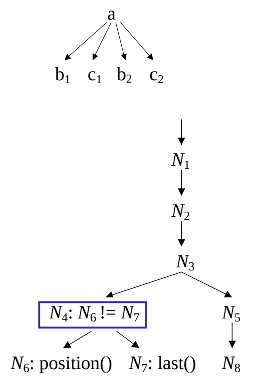
N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false

N_6: position()			
cn	cp	cs	res
c_1	1	3	1
b_2	2	3	2
c_2	3	3	3
b_2	1	2	1
c_2	2	2	2
c_2	1	1	1

N_7: last()			
cn	cp	cs	res
c_1	1	3	3
b_2	2	3	3
c_2	3	3	3
b_2	1	2	2
c_2	2	2	2
c_2	1	1	1

89

<a> <b/> <c/> <b/> <c/>



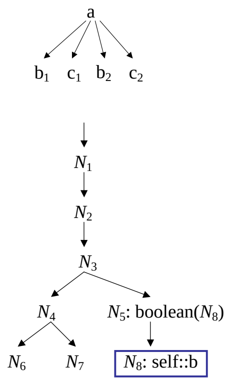
N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_6: position()			
cn	cp	cs	res
c_1	1	3	1
b_2	2	3	2
c_2	3	3	3
b_2	1	2	1
c_2	2	2	2
c_2	1	1	1

N_7: last()			
cn	cp	cs	res
c_1	1	3	3
b_2	2	3	3
c_2	3	3	3
b_2	1	2	2
c_2	2	2	2
c_2	1	1	1

90

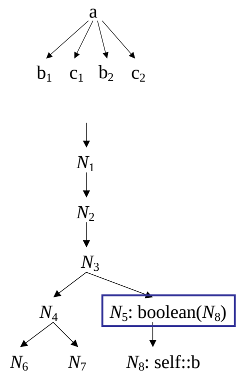
<a> <b/> <c/> <b/> <c/>



N_8: self::b			
cn	cp	cs	res
c_1	1	3	{ }
b_2	2	3	{ b_2 }
c_2	3	3	{ }
b_2	1	2	{ b_2 }
c_2	2	2	{ }
c_2	1	1	{ }

91

<a> <b/> <c/> <b/> <c/>

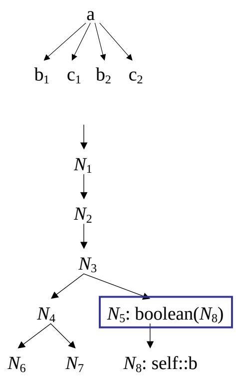


N_5: boolean(N_8)			
cn	cp	cs	res

N_8: self::b			
cn	cp	cs	res
c_1	1	3	{ }
b_2	2	3	{ b_2 }
c_2	3	3	{ }
b_2	1	2	{ b_2 }
c_2	2	2	{ }
c_2	1	1	{ }

92

<a> <b/> <c/> <b/> <c/>

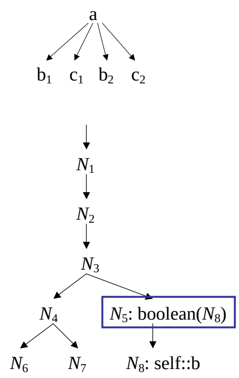


N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false

N_8: self::b			
cn	cp	cs	res
c_1	1	3	{ }
b_2	2	3	{ b_2 }
c_2	3	3	{ }
b_2	1	2	{ b_2 }
c_2	2	2	{ }
c_2	1	1	{ }

93

<a> <b/> <c/> <b/> <c/>

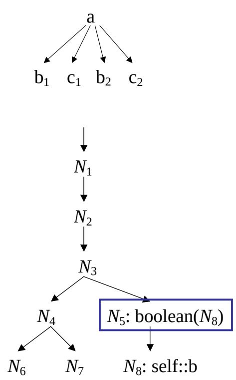


N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true

N_8: self::b			
cn	cp	cs	res
c_1	1	3	{ }
b_2	2	3	{ b_2 }
c_2	3	3	{ }
b_2	1	2	{ b_2 }
c_2	2	2	{ }
c_2	1	1	{ }

94

<a> <b/> <c/> <b/> <c/>

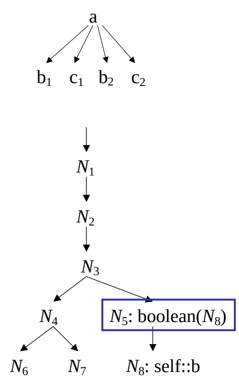


N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false

N_8: self::b			
cn	cp	cs	res
c_1	1	3	{ }
b_2	2	3	{ b_2 }
c_2	3	3	{ }
b_2	1	2	{ b_2 }
c_2	2	2	{ }
c_2	1	1	{ }

95

<a> <b/> <c/> <b/> <c/>

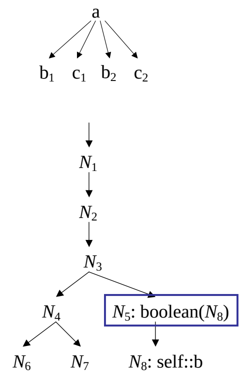


N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true

N_8: self::b			
cn	cp	cs	res
c_1	1	3	{ }
b_2	2	3	{ b_2 }
c_2	3	3	{ }
b_2	1	2	{ b_2 }
c_2	2	2	{ }
c_2	1	1	{ }

96

<a> <b/> <c/> <b/> <c/>

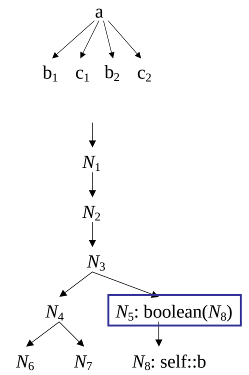


N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false

N_8: self::b			
cn	cp	cs	res
c_1	1	3	{ }
b_2	2	3	{ b_2 }
c_2	3	3	{ }
b_2	1	2	{ b_2 }
c_2	2	2	{ }
c_2	1	1	{ }

97

<a> <b/> <c/> <b/> <c/>

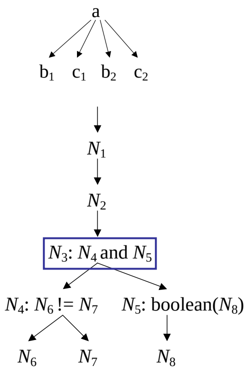


N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_8: self::b			
cn	cp	cs	res
c_1	1	3	{ }
b_2	2	3	{ b_2 }
c_2	3	3	{ }
b_2	1	2	{ b_2 }
c_2	2	2	{ }
c_2	1	1	{ }

98

<a> <b/> <c/> <b/> <c/>



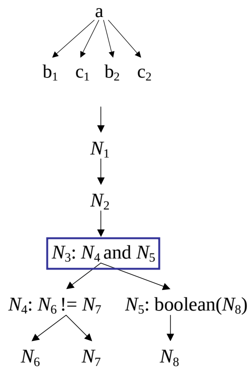
N_3: N_4 and N_5			
cn	cp	cs	res

N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

99

<a> <b/> <c/> <b/> <c/>



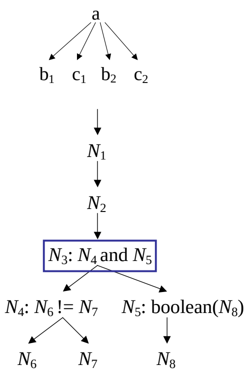
N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false

N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

100

<a> <b/> <c/> <b/> <c/>



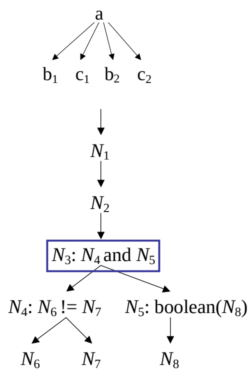
N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true

N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

101

<a> <b/> <c/> <b/> <c/>



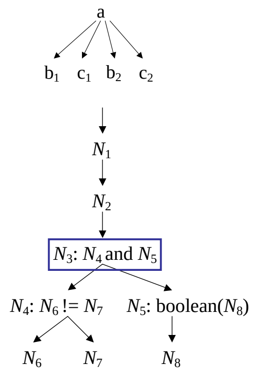
N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false

N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

102

<a> <b/> <c/> <b/> <c/>

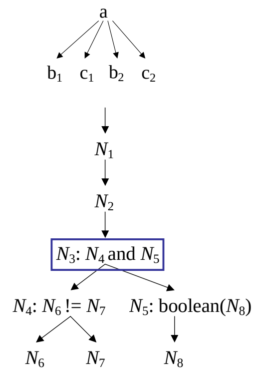


N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true

N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

<a> <b/> <c/> <b/> <c/>

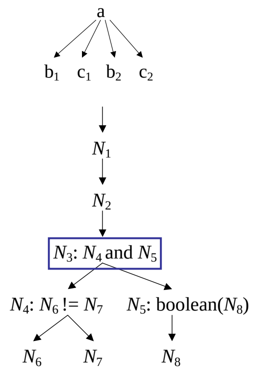


N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false

N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

<a> <b/> <c/> <b/> <c/>

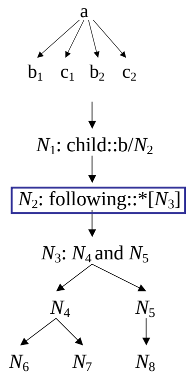


N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_4: N_6 != N_7			
cn	cp	cs	res
c_1	1	3	true
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

N_5: boolean(N_8)			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

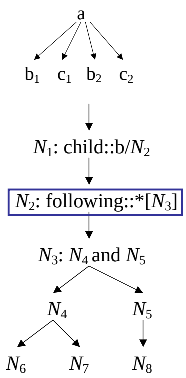
<a> <b/> <c/> <b/> <c/>



N_2: following::*[N_3]			
cn	cp	cs	res

N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

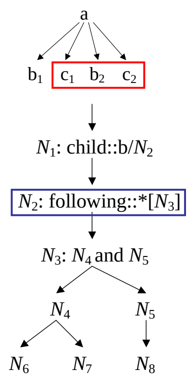
<a> <b/> <c/> <b/> <c/>



N_2: following::*[N_3]			
cn	cp	cs	res
a	.	.	{ }

N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

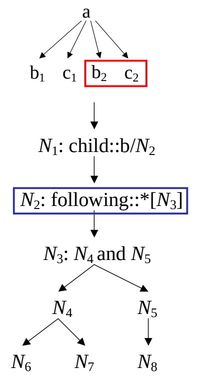
<a> <b/> <c/> <b/> <c/>



N_2: following::*[N_3]			
cn	cp	cs	res
a	.	.	{ }
b_1	.	.	{ b_2 }

N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

<a> <b/> <c/> <b/> <c/>

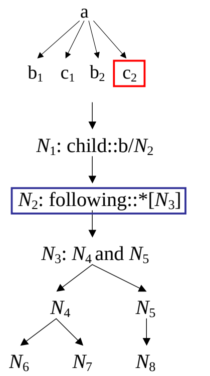


N_2: following::*[N_3]			
cn	cp	cs	res
a	.	.	{ }
b_1	.	.	{ b_2 }
c_1	.	.	{ b_2 }

N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

109

<a> <b/> <c/> <b/> <c/>

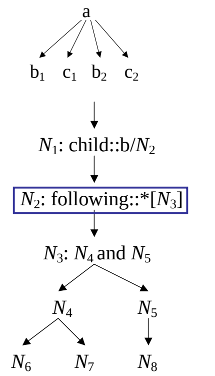


N_2: following::*[N_3]			
cn	cp	cs	res
a	.	.	{ }
b_1	.	.	{ b_2 }
c_1	.	.	{ b_2 }
b_2	.	.	{ }

N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

110

<a> <b/> <c/> <b/> <c/>



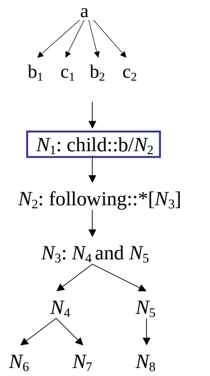
N_2: following::*[N_3]			
cn	cp	cs	res
a	.	.	{ }
b_1	.	.	{ b_2 }
c_1	.	.	{ b_2 }
b_2	.	.	{ }
c_2	.	.	{ }

N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

111

<a> <b/> <c/> <b/> <c/>

N_1: child::b/N_2			
cn	cp	cs	res



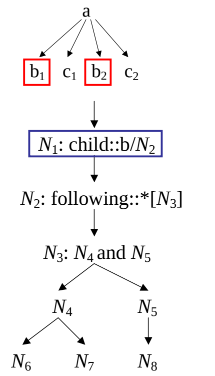
N_2: following::*[N_3]			
cn	cp	cs	res
a	.	.	{ }
b_1	.	.	{ b_2 }
c_1	.	.	{ b_2 }
b_2	.	.	{ }
c_2	.	.	{ }

N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

112

<a> <b/> <c/> <b/> <c/>

N_1: child::b/N_2			
cn	cp	cs	res
a	.	.	{ b_2 }



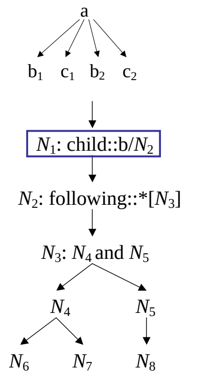
N_2: following::*[N_3]			
cn	cp	cs	res
a	.	.	{ }
b_1	.	.	{ b_2 }
c_1	.	.	{ b_2 }
b_2	.	.	{ }
c_2	.	.	{ }

N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

113

<a> <b/> <c/> <b/> <c/>

N_1: child::b/N_2			
cn	cp	cs	res
a	.	.	{ b_2 }
b_1	.	.	{ }
c_1	.	.	{ }
b_2	.	.	{ }
c_2	.	.	{ }



N_2: following::*[N_3]			
cn	cp	cs	res
a	.	.	{ }
b_1	.	.	{ b_2 }
c_1	.	.	{ b_2 }
b_2	.	.	{ }
c_2	.	.	{ }

N_3: N_4 and N_5			
cn	cp	cs	res
c_1	1	3	false
b_2	2	3	true
c_2	3	3	false
b_2	1	2	true
c_2	2	2	false
c_2	1	1	false

114

Context-Value Table Principle

if CVT for each operation $Op(e_1,\ldots,e_n)$ can be computed in polynomial time given the CVTs for sub-expressions $e_1,\ldots,e_n$

then CVT of overall query can be computed (bottom-up) in polynomial time.

115

Time and Space Bounds

Bottom-up evaluation based on CVT:
- **Time O(|data|^5 * |query|^2), Space O(|data|^4 * |query|^2).**

Space bound (n … number of nodes in input document.):
- Contexts are at most triples: at most n^3 contexts.
- Sizes of values:
  - Node sets: at most O(n)
  - Strings, numbers: at most O(|data|* |query|) – (iterated concatenation of strings, multiplication of numbers)

⇒ Each CVT is of size (|data|^4 * |query|).

Need to compute a CVT for each query node and each input node
→ (|data| * |query|) (|data|^4 * |query|)

Time bound: most expensive computation is O(n^2) – Relational operation "=" on node sets (e.g. a/b//c[d//e/f/g = h/i//j])

116

Efficiency of the PTIME Algorithm

- Time Complexity $O(|D|^5 * |Q|^2)$
- Space Complexity $O(|D|^4 * |Q|^2)$
- In practice, most queries run in quadratic time
- This is for main-memory implementations.
- Adaptation to secondary storage algorithms with PTIME complexity is easy (but with worse bounds than the ones given above).

117

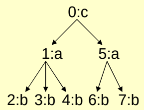
Alternative Context Representation

- Contexts represented as
  ("previous context node", "current context node")
  rather than
  ("context node", "position", "size").
- Need to recompute "position" and "size" on demand.

//a/b[position() + 1 = size()]

child::b … { (1,2), (1,3), (1,4), (5,6), (5,7) }

child::b[position()+1=size()] … { (1,3), (5,6) }

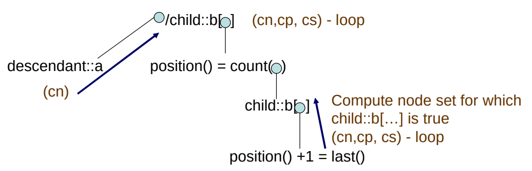


- Complexity lowered to time O(|data|^4 * |query|^2), space O(|data|^3 * |query|^2).

118

Context Simplification Technique

1. Only materialize relevant context.
2. Core Xpath evaluation algorithm for outermost and innermost paths //a/b/c//d[…]/e[…(a/b/c)].
3. Treating "position" and "size" in a loop.
   - Because of tree shape of query, loops never have to be nested.



119

Linear Space Fragment

- "Wadler Fragment" [Wadler, 1999]: Core Xpath + position(), last(), and arithmetics.
- Evaluation in *quadratic time and linear space*.

(cn,cp,cs) (cn,cp,cs)

//a/b[position() * 2 = last() and c/d[position()*3 = last()]]//e

(cn) (cn) (cn) (cn)

- For x in [[//a]] compute contexts (y,p,n) in x.[[b]]
  Compute Y = { y | (y,p,n) ∈ x.[[b]] and p*2=n }.
- Similarly, compute Z = { z | z.[[d[position()*3 = last()]]] is true}.
- Compute X = { x | z ∈ Z, x ∈ z.[[child::c]]$^{-1}$ } – in linear time.
- Result is { w | v ∈ X ∩ Y, w ∈ v.[[descendant::e]] }.

120

Summary

Full XPath

- Bottom-up algorithm based on CVT
  - Time O(|data|^5 * |query|^2), space O(|data|^4 * |query|^2).
- Top-down evaluation
  - Time O(|data|^4 * |query|^2), space O(|data|^3 * |query|^2).
- Context-reduction technique
  - Time O(|data|^4 * |query|^2), space O(|data|^2 * |query|^2).

Wadler fragment

- Time O(|data|^2 * |query|^2), space O(|data| * |query|).

Core Xpath

- Time and space O(|data| * |query|).

121

END
Lecture 7

122