

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*
2. Node Sets only: **Core XPath**
3. Bottom-Up Evaluation of **Core XPath**
4. Polynomial Time Evaluation of **Full XPath**

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

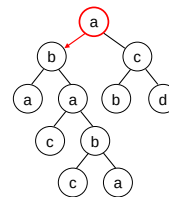
`//author/last` = select all last names of authors
`//strip/characters/character/text()` = select all character names from a DilbertML document

(return selected nodes in document order..)

3

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]`

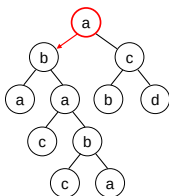
Thus, `p=p+1=2`

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

4

1. Top-Down Evaluation of *Simple Paths*

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



`//a/b` = `Q`

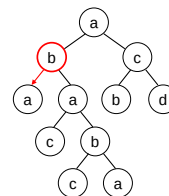
query match position: `p = 2`

`[startElement( a )]`

5

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)**

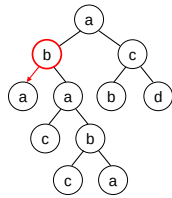
→ `p = 1`

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

6

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

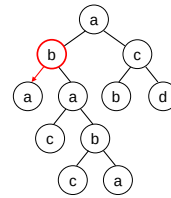
Push current match position p
for every **startElement**
(except for the root node)

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

7

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

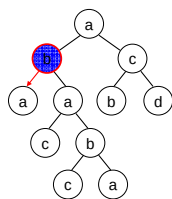
Push current match position
for every **startElement**
(except for the root node)

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

8

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

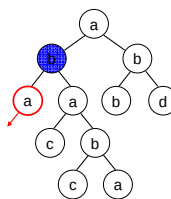
[startElement(a)]
[startElement(b)]

2

9

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

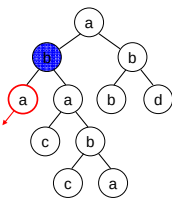
[startElement(a)]
[startElement(b)]
[startElement(a)]
↑
=Q[1]

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

10

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)]

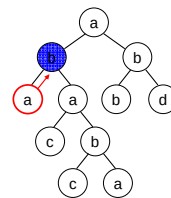
1

2

11

1. Top-Down Evaluation of Simple Paths

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



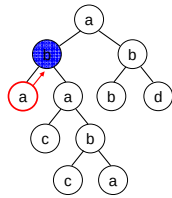
//a/b =Q
↑
query match position: p = 2

[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
↑
p = pop()

12

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

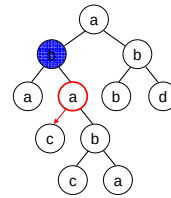
[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]

2

13

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

=Q[1]

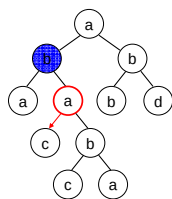
Thus, push(p)
and p=p+1=2

2

14

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

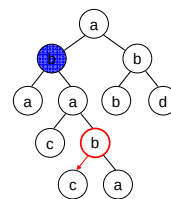
[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]

1
2

15

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)]

push(2)
p = pop() = 2

=Q[2]

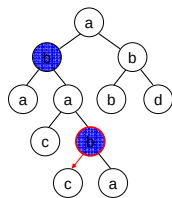
p=2=length(Q), thus,
current node is a **match!**
→ Mark it as **match/result**
→ push(p)
→ p = 1

1
2

16

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

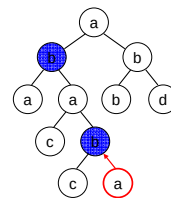
[startElement(a)]
[startElement(b)]
[startElement(a)]
[endElement(a)]
[startElement(a)]
[startElement(c)]
[endElement(c)]
[startElement(b)]

2
1
2

17

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

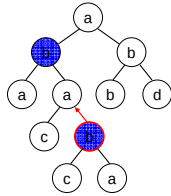
push(1)
p = pop() = 1
push(1)
p = pop() = 1

2
1
2

18

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

2
1
2

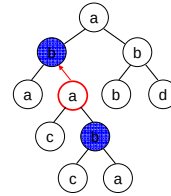
```
[startElement( a )]
[startElement( b )]
[startElement( a )]
[endElement( a )]
[startElement( a )]
[startElement( c )]
[endElement( c )]
[endElement( b )] p = pop() = 2
[startElement( c )] push(1)
[endElement( c )] p = pop() = 1
[startElement( a )] push(1)
[endElement( a )] p = pop() = 1
```

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

19

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

1
2

```
[startElement( a )]
[startElement( b )]
[startElement( a )]
[endElement( a )]
[startElement( a )]
[startElement( c )]
[endElement( c )]
[endElement( b )] p = pop() = 2
[endElement( a )] p = pop() = 1
[startElement( c )] push(1)
[endElement( c )] p = pop() = 1
[startElement( a )] push(1)
[endElement( a )] p = pop() = 1
```

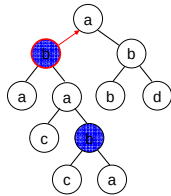
[endElement(b)] p = pop() = 2

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

20

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

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

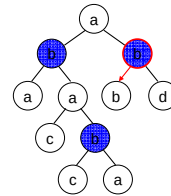
[endElement(a)] p = pop() = 1

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

21

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

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

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

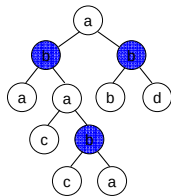
[endElement(b)] p = pop() = 2

[startElement(b)] → match

22

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 ☹

23

SAX-based path query evaluation (sketch):

1 Preparation:

- Represent path query $//t_1/t_2/\dots/t_{n-1}/text()$ via the step array $path[0] = t_1, path[1] = t_2, \dots, path[n-1] = text()$.
- Maintain an array index $i = 0 \dots n$, 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 transition¹⁴. 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 .

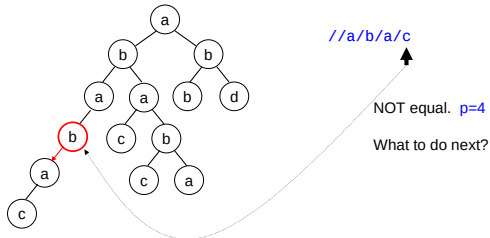
5 [characters]

If the current step $path[i] = text()$ we have found a match. Otherwise do nothing.

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

1. Top-Down Evaluation of Simple Paths

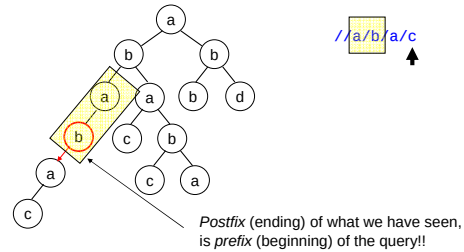
→ evaluate using one single pre-order traversal! (using a **stack**)



25

1. Top-Down Evaluation of Simple Paths

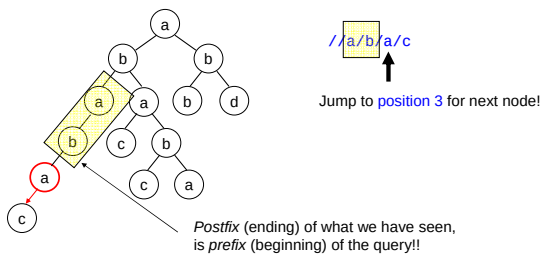
→ evaluate using one single pre-order traversal! (using a **stack**)



26

1. Top-Down Evaluation of Simple Paths

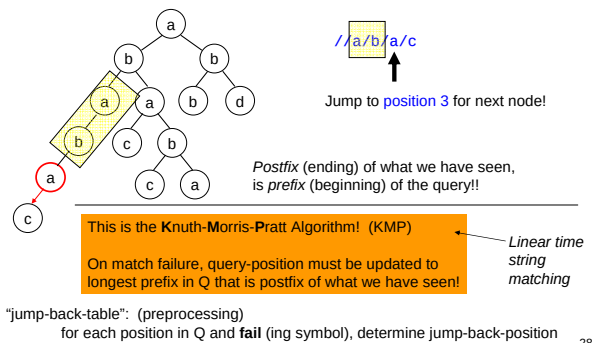
→ evaluate using one single pre-order traversal! (using a **stack**)



27

1. Top-Down Evaluation of Simple Paths

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



28

2. Core XPath

→ 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::*) ]`

Types
• Node Sets
• Booleans

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.)

29

2. Core XPath

→ 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::*) ]`

Types
• Node Sets
• Booleans

Thus in Core XPath

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

30

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 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

In doc order

reverse doc order

31

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 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
 U **axis(4)** ← time linear in #Nodes
 U **axis(5)** ← time linear in #Nodes
 U **axis(7)** ← time linear in #Nodes

at most #Nodes many

$O(\#Nodes \#Nodes) = \text{quadratic time } \ominus$

32

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 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

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.)

In doc order

33

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 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.)

34

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 1 0 0 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.)

35

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 1 0 0 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..

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

36

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
    ↑

```

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
    ↑

```

Idea
→ No node is visited >once!
e.g.: `ancestor( {5, 8, 9} )`

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

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
    ↑

```

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
    ↑

```

Idea
→ No node is visited >once!
e.g.: `ancestor( {5, 8, 9} )`

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

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
    ↑

```

Result Node Set

```

1 2 3 4 5 6 7 8 9 10 11
1 1 1 0 0 0 0 0 0 0 0
    ↑

```

Idea
→ No node is visited >once!
e.g.: `ancestor( {5, 8, 9} )`

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

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
    ↑

```

Result Node Set

```

1 2 3 4 5 6 7 8 9 10 11
1 1 1 0 0 0 1 0 0 0 0
    ↑

```

Idea
→ No node is visited >once!
e.g.: `ancestor( {5, 8, 9} )`

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

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
    ↑

```

Result Node Set

```

1 2 3 4 5 6 7 8 9 10 11
1 1 1 0 0 0 1 0 0 0 0
    ↑

```

Idea
→ No node is visited >once!
e.g.: `ancestor( {5, 8, 9} )`

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

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
    ↑

```

Result Node Set

```

1 2 3 4 5 6 7 8 9 10 11
1 1 1 0 0 0 1 0 0 0 0
    ↑

```

Idea
→ No node is visited >once!
e.g.: `ancestor( {5, 8, 9} )`

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

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.

43

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({ 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;
 }
}

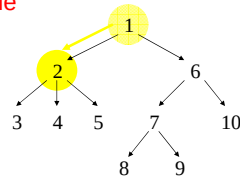
44

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	ns
1 2	2 6
2 3	3 4
6 7	4 5
7 8	7 10
	8 9

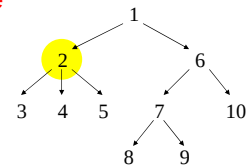
This example comes from
Georg Gottlob and Christoph Koch "XPath Query Processing".
Invited tutorial at DBPL 2003
<http://www.dbai.tuwien.ac.at/research/xmlaskforce/xpath-tutorial1.ppt.gz>

45

Example

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

descendant({ 1 }) =
(fc | ns)*(first-child({ 1 })) =
(fc | ns)*({ 2 }) =
{ 2 } +



fc	ns
1 2	2 6
2 3	3 4
6 7	4 5
7 8	7 10
	8 9

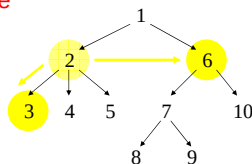
Result Node Set
1 2 3 4 5 6 7 8 9 10
0 1 0 0 0 0 0 0 0 0

46

Example

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

descendant({ 1 }) =
(fc | ns)*(first-child({ 1 })) =
(fc | ns)*({ 2 }) =
{ 2 } +
{ 3, 6 } +



fc	ns
1 2	2 6
2 3	3 4
6 7	4 5
7 8	7 10
	8 9

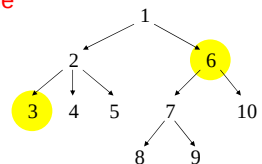
Result Node Set
1 2 3 4 5 6 7 8 9 10
0 1 1 0 0 1 0 0 0 0

47

Example

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

descendant({ 1 }) =
(fc | ns)*(first-child({ 1 })) =
(fc | ns)*({ 2 }) =
{ 2 } +
{ 3, 6 } +



fc	ns
1 2	2 6
2 3	3 4
6 7	4 5
7 8	7 10
	8 9

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

48

Example

$\text{descendant}(\text{node}) = (\text{first-child} \mid \text{next-sibling})^* (\text{first-child}(\text{node}))$

$\text{descendant}(\{1\}) =$
 $(\text{fc} \mid \text{ns})^*(\text{first-child}(\{1\})) =$
 $(\text{fc} \mid \text{ns})^*(\{2\}) =$
 $\{2\} +$
 $\{3, 6\} +$
 $\{4, 7\} +$

Result Node Set
 $\frac{1}{0} \frac{2}{1} \frac{3}{1} \frac{4}{1} \frac{5}{0} \frac{6}{1} \frac{7}{1} \frac{8}{0} \frac{9}{0} \frac{10}{0}$

fc		ns	
1	2	2	6
2	3	3	4
6	7	4	5
7	8	7	10
		8	9

49

Example

$\text{descendant}(\text{node}) = (\text{first-child} \mid \text{next-sibling})^* (\text{first-child}(\text{node}))$

$\text{descendant}(\{1\}) =$
 $(\text{fc} \mid \text{ns})^*(\text{first-child}(\{1\})) =$
 $(\text{fc} \mid \text{ns})^*(\{2\}) =$
 $\{2\} +$
 $\{3, 6\} +$
 $\{4, 7\} +$

Result Node Set
 $\frac{1}{0} \frac{2}{1} \frac{3}{1} \frac{4}{0} \frac{5}{1} \frac{6}{1} \frac{7}{1} \frac{8}{0} \frac{9}{0} \frac{10}{0}$

fc		ns	
1	2	2	6
2	3	3	4
6	7	4	5
7	8	7	10
		8	9

50

Example

$\text{descendant}(\text{node}) = (\text{first-child} \mid \text{next-sibling})^* (\text{first-child}(\text{node}))$

$\text{descendant}(\{1\}) =$
 $(\text{fc} \mid \text{ns})^*(\text{first-child}(\{1\})) =$
 $(\text{fc} \mid \text{ns})^*(\{2\}) =$
 $\{2\} +$
 $\{3, 6\} +$
 $\{4, 7\} +$
 $\{5, 8, 10\} +$

Result Node Set
 $\frac{1}{0} \frac{2}{1} \frac{3}{1} \frac{4}{1} \frac{5}{1} \frac{6}{1} \frac{7}{1} \frac{8}{1} \frac{9}{0} \frac{10}{1}$

fc		ns	
1	2	2	6
2	3	3	4
6	7	4	5
7	8	7	10
		8	9

51

Example

$\text{descendant}(\text{node}) = (\text{first-child} \mid \text{next-sibling})^* (\text{first-child}(\text{node}))$

$\text{descendant}(\{1\}) =$
 $(\text{fc} \mid \text{ns})^*(\text{first-child}(\{1\})) =$
 $(\text{fc} \mid \text{ns})^*(\{2\}) =$
 $\{2\} +$
 $\{3, 6\} +$
 $\{4, 7\} +$
 $\{5, 8, 10\} +$

Result Node Set
 $\frac{1}{0} \frac{2}{1} \frac{3}{1} \frac{4}{0} \frac{5}{1} \frac{6}{1} \frac{7}{1} \frac{8}{1} \frac{9}{0} \frac{10}{1}$

fc		ns	
1	2	2	6
2	3	3	4
6	7	4	5
7	8	7	10
		8	9

52

Example

$\text{descendant}(\text{node}) = (\text{first-child} \mid \text{next-sibling})^* (\text{first-child}(\text{node}))$

$\text{descendant}(\{1\}) =$
 $(\text{fc} \mid \text{ns})^*(\text{first-child}(\{1\})) =$
 $(\text{fc} \mid \text{ns})^*(\{2\}) =$
 $\{2\} +$
 $\{3, 6\} +$
 $\{4, 7\} +$
 $\{5, 8, 10\} +$
 $\{9\}$

Result Node Set
 $\frac{1}{0} \frac{2}{1} \frac{3}{1} \frac{4}{1} \frac{5}{1} \frac{6}{1} \frac{7}{1} \frac{8}{1} \frac{9}{1} \frac{10}{1}$

fc		ns	
1	2	2	6
2	3	3	4
6	7	4	5
7	8	7	10
		8	9

53

Example

$\text{descendant}(\text{node}) = (\text{first-child} \mid \text{next-sibling})^* (\text{first-child}(\text{node}))$

$\text{descendant}(\{1\}) =$
 $(\text{fc} \mid \text{ns})^*(\text{first-child}(\{1\})) =$
 $(\text{fc} \mid \text{ns})^*(\{2\}) =$
 $\{2\} +$
 $\{3, 6\} +$
 $\{4, 7\} +$
 $\{5, 8, 10\} +$
 $\{9\}$

Result Node Set
 $\frac{1}{0} \frac{2}{1} \frac{3}{1} \frac{4}{1} \frac{5}{1} \frac{6}{1} \frac{7}{1} \frac{8}{1} \frac{9}{1} \frac{10}{1}$

fc		ns	
1	2	2	6
2	3	3	4
6	7	4	5
7	8	7	10
		8	9

54

Core XPath

→ 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::*) ]`

Types
• Node Sets
• Booleans

→ For Core XPath we only need **Node Set** operations!!

- $\text{axis}(\text{Set1}) = \text{Set2}$
- $\text{U}(\text{Set1}, \text{Set2}) = \text{Set3}$ union of Set1 and Set2
- $\cap(\text{Set1}, \text{Set2}) = \text{Set3}$ intersection of Set1 and Set2
- $\neg(\text{Set1}, \text{Set2}) = \text{Set3}$ everything in Set1 but **not** in Set2
- $\text{lab}(a) = \text{Set1}$ 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: $O(\# \text{Nodes}) = O(|D|)$

Node Set operation: $O(|D|)$

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

linear time! ☺

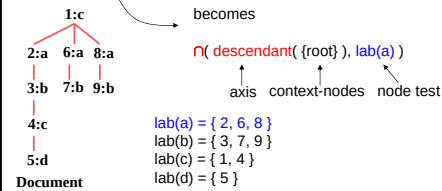
→ For Core XPath we only need **Node Set** operations!!

- $\text{axis}(\text{Set1}) = \text{Set2}$
- $\text{U}(\text{Set1}, \text{Set2}) = \text{Set3}$ union of Set1 and Set2
- $\cap(\text{Set1}, \text{Set2}) = \text{Set3}$ intersection of Set1 and Set2
- $\neg(\text{Set1}, \text{Set2}) = \text{Set3}$ everything in Set1 but **not** in Set2
- $\text{lab}(a) = \text{Set1}$ all nodes **labeled by a**

for everything else (steps, filters)

57

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



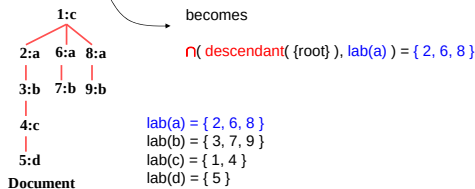
→ For Core XPath we only need **Node Set** operations!!

- $\text{axis}(\text{Set1}) = \text{Set2}$
- $\text{U}(\text{Set1}, \text{Set2}) = \text{Set3}$ union of Set1 and Set2
- $\cap(\text{Set1}, \text{Set2}) = \text{Set3}$ intersection of Set1 and Set2
- $\neg(\text{Set1}, \text{Set2}) = \text{Set3}$ everything in Set1 but **not** in Set2
- $\text{lab}(a) = \text{Set1}$ all nodes **labeled by a**

for everything else (steps, filters)

58

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



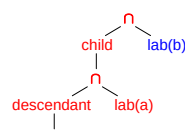
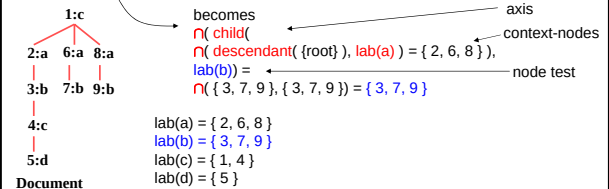
→ For Core XPath we only need **Node Set** operations!!

- $\text{axis}(\text{Set1}) = \text{Set2}$
- $\text{U}(\text{Set1}, \text{Set2}) = \text{Set3}$ union of Set1 and Set2
- $\cap(\text{Set1}, \text{Set2}) = \text{Set3}$ intersection of Set1 and Set2
- $\neg(\text{Set1}, \text{Set2}) = \text{Set3}$ everything in Set1 but **not** in Set2
- $\text{lab}(a) = \text{Set1}$ all nodes **labeled by a**

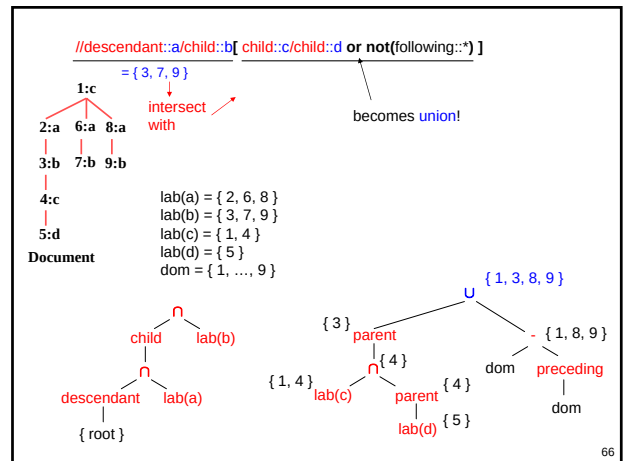
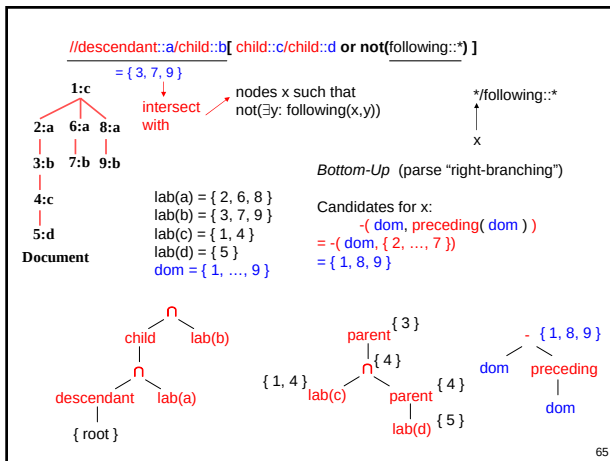
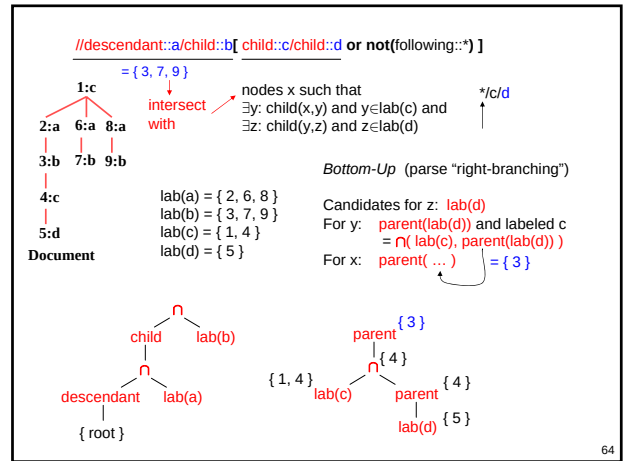
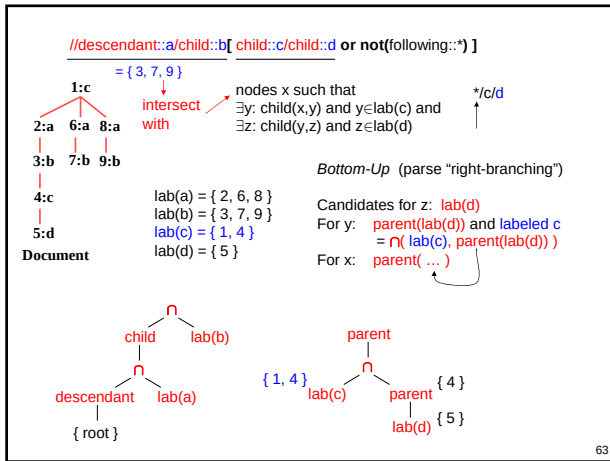
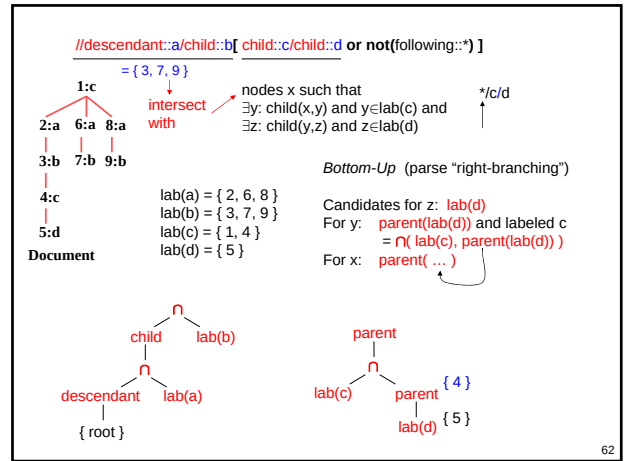
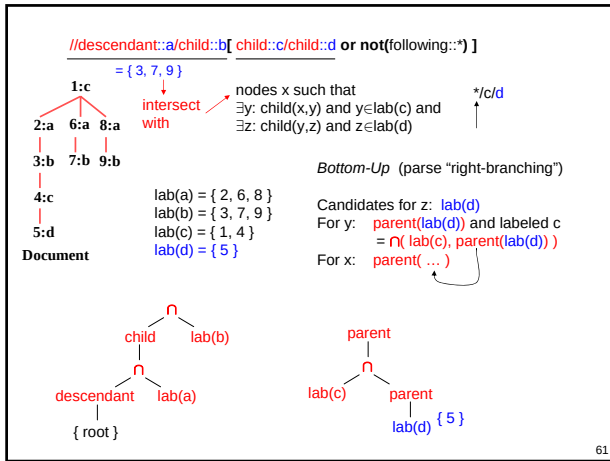
for everything else (steps, filters)

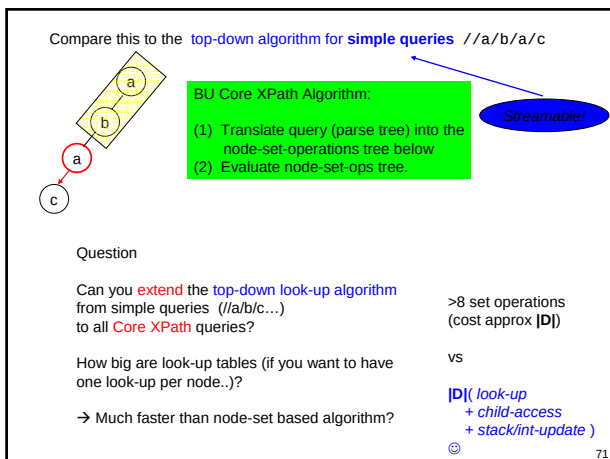
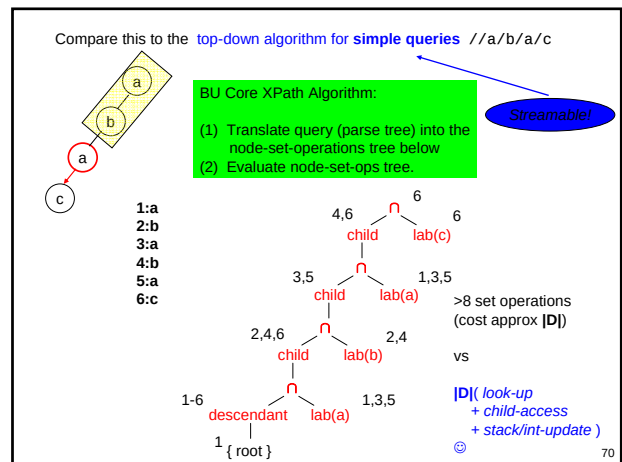
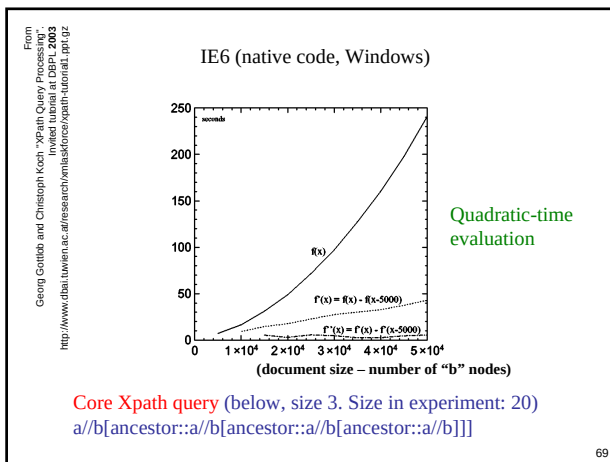
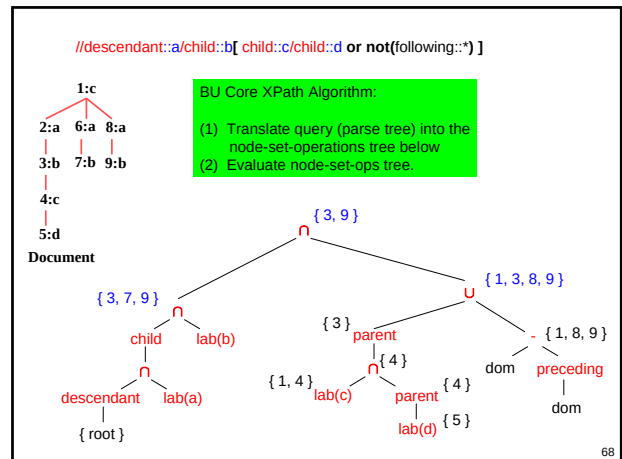
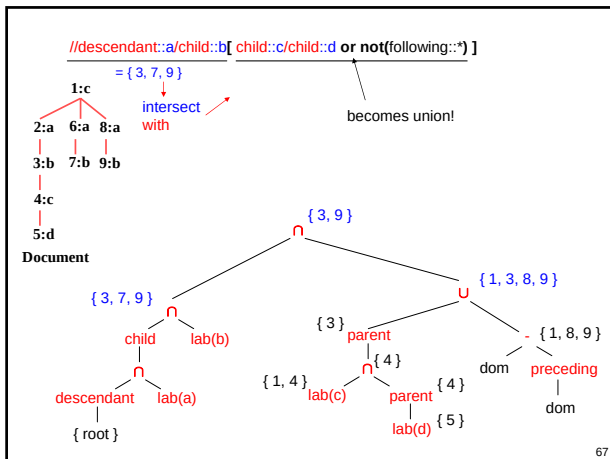
59

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



60





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. Contexts

context: $\langle x, k, n \rangle$
 node position size

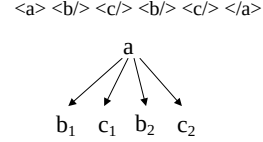
These values specify a current “situation” in which a query or subquery should be evaluated.
 Determined by preceding XSL or XPath computations.

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

73

Example of an XPath query not in Core XPath

Sample document D:



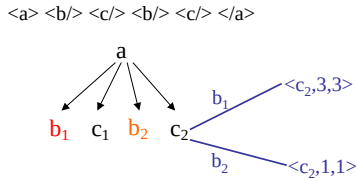
Sample query Q:

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

74

Example of an XPath query not in Core XPath

Sample document D:



Sample query Q:

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

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

75

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

Expr. E : Operator Signature	Semantics $\mathcal{E}_\uparrow[E]$
location step $\chi::t : \rightarrow \text{nset}$	$\{(x, k, n, \{x \mid x \in T(t)\}) \mid (x, k, n) \in C\}$
location step $E_1[E_2]$ over axis χ : $\text{nset} \times \text{bool} \rightarrow \text{nset}$	$\{(x, k, n, \{x \in S \mid (x, \text{id}_x(x, S), S , \text{true}) \in \mathcal{E}_\uparrow[E_2]\}) \mid (x, k, n, S) \in \mathcal{E}_\uparrow[E_1]\}$
location path $\pi : \text{nset} \rightarrow \text{nset}$	$C \times \{S \mid \exists k, n : (\text{root}, k, n, S) \in \mathcal{E}_\uparrow[\pi]\}$
location path $\pi_1/\pi_2 : \text{nset} \times \text{nset} \rightarrow \text{nset}$	$\{(x, k, n, z) \mid 1 \leq k \leq n \leq \text{dom} , (x, k_1, n_1, Y) \in \mathcal{E}_\uparrow[\pi_1], \bigcup_{y \in Y} (y, k_2, n_2, z) \in \mathcal{E}_\uparrow[\pi_2]\}$
location path $\pi_1 \mid \pi_2 : \text{nset} \times \text{nset} \rightarrow \text{nset}$	$\mathcal{E}_\uparrow[\pi_1] \cup \mathcal{E}_\uparrow[\pi_2]$
position() : $\rightarrow \text{num}$	$\{(x, k, n, k) \mid (x, k, n) \in C\}$
last() : $\rightarrow \text{num}$	$\{(x, k, n, n) \mid (x, k, n) \in C\}$

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

76

Example: Formal Semantics of XPath Relational Operators

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

77

Std. Semantics of Location Paths

$P[\chi::t[e_1] \dots [e_m]](x) :=$
begin
 $S := \{y \mid x\chi y, y \in T(t)\};$
for $1 \leq i \leq m$ (in ascending order) **do**
 $S := \{y \in S \mid [e_i](y, \text{id}_x(y, S), |S|) = \text{true}\};$
return S;
end;
 $P[\pi_1/\pi_2](x) := P[\pi_1](x) \cup P[\pi_2](x)$
 $P[\pi](x) := P[\pi](\text{root})$
 $P[\pi_1/\pi_2](x) := \bigcup_{y \in P[\pi_1](x)} P[\pi_2](y)$

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

78

Context-value Tables (CVT)

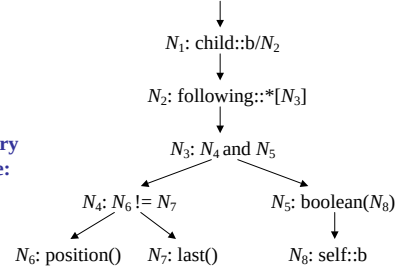
- Four types of values (*nset*, *num*, *str*, *bool*)
- Defined for each XPath expression *e*
- The CVT of *e* is a relation $R \subseteq C \times (nset \cup num \cup str \cup bool)$

79

Parse Tree of the Query

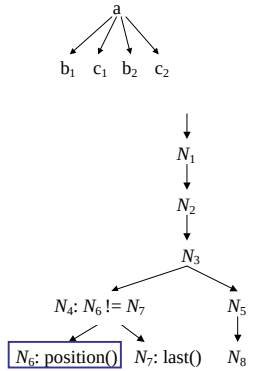
Query:
`child::b/following::*[position() != last() and self::b]`

Query Tree:



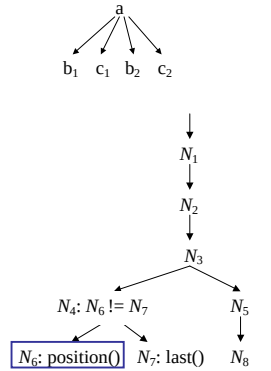
80

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



81

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

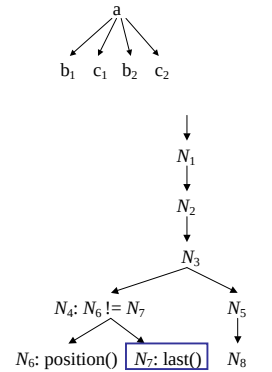


N6: position()			
cn	cp	cs	res
c1	1	3	1
b2	2	3	2
c2	3	3	3
b2	1	2	1
c2	2	2	2
c2	1	1	1

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

82

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

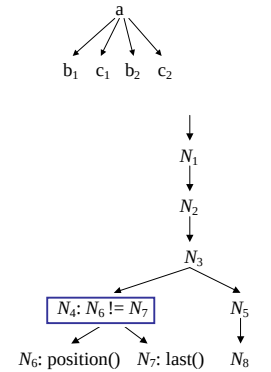


N6: position()			
cn	cp	cs	res
c1	1	3	1
b2	2	3	2
c2	3	3	3
b2	1	2	1
c2	2	2	2
c2	1	1	1

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

83

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

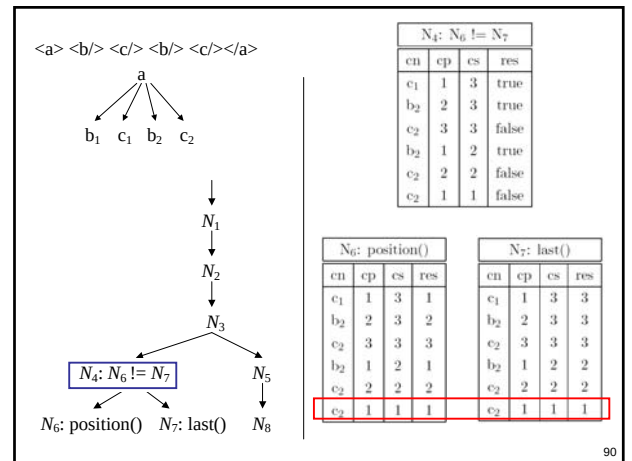
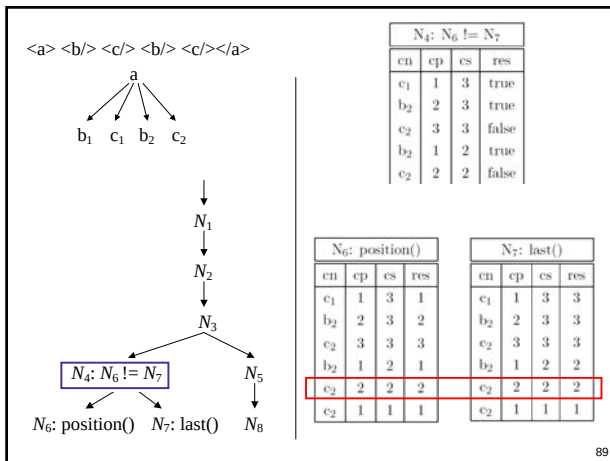
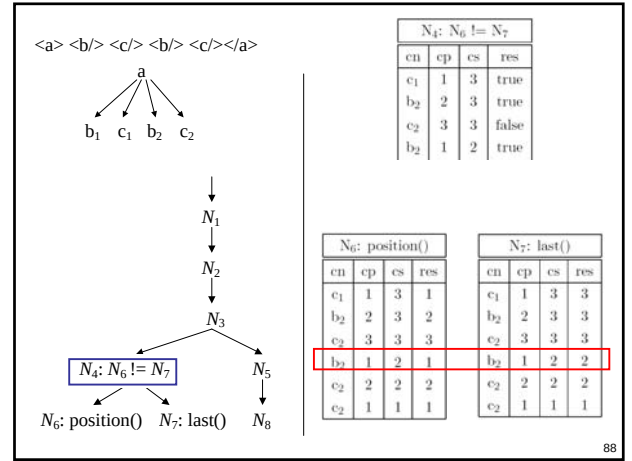
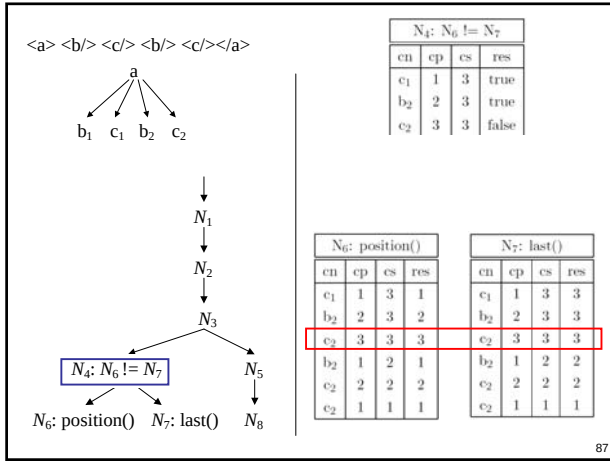
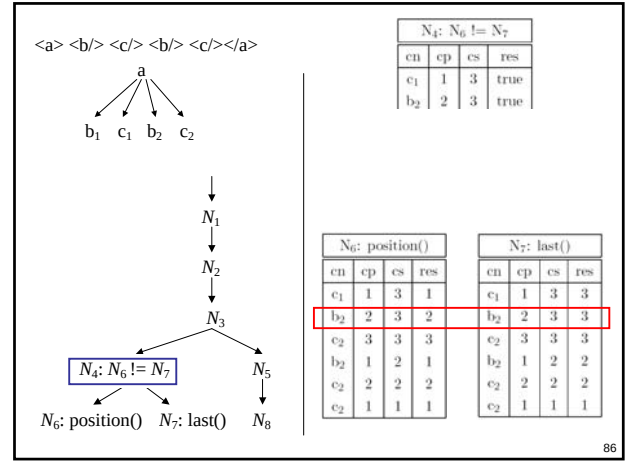
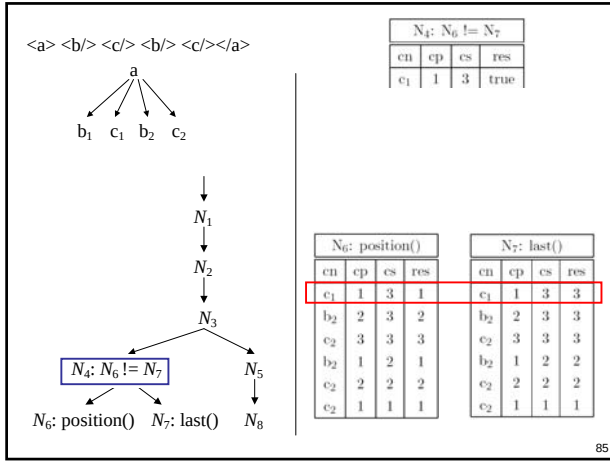


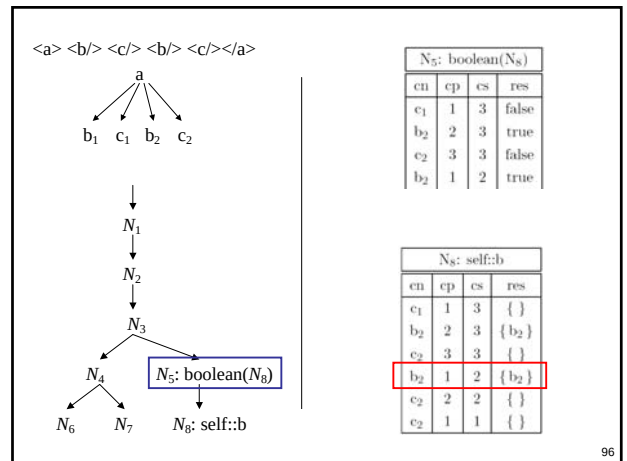
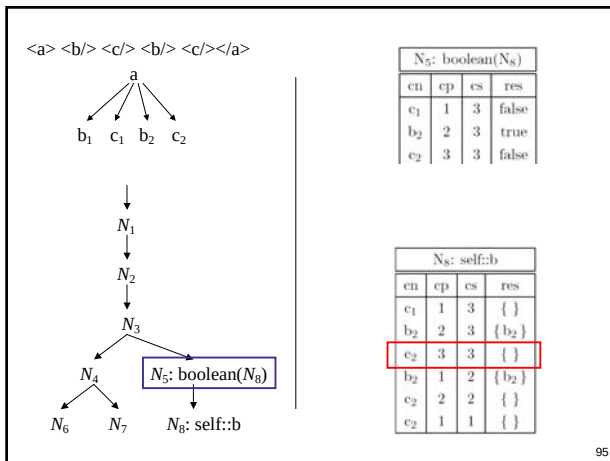
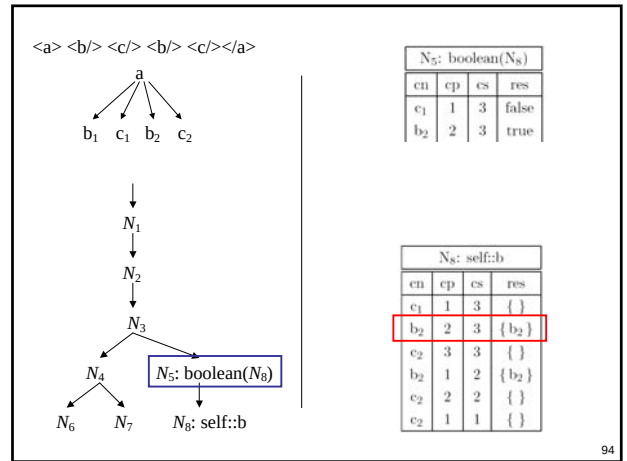
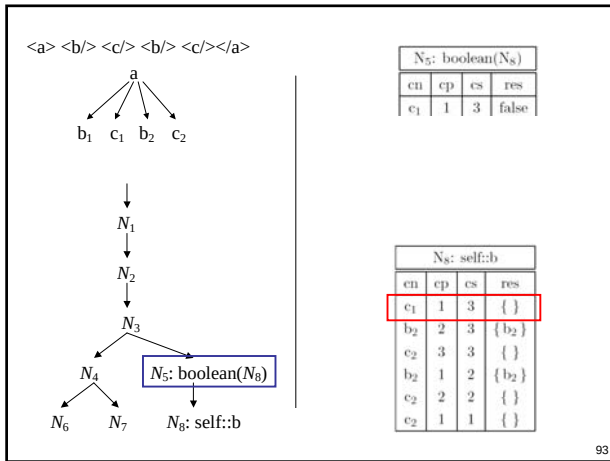
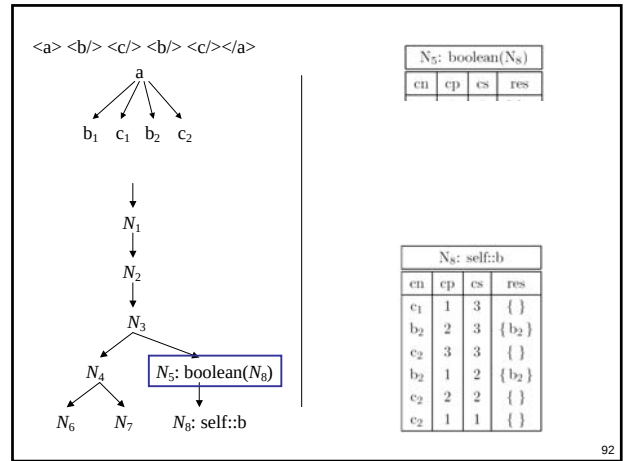
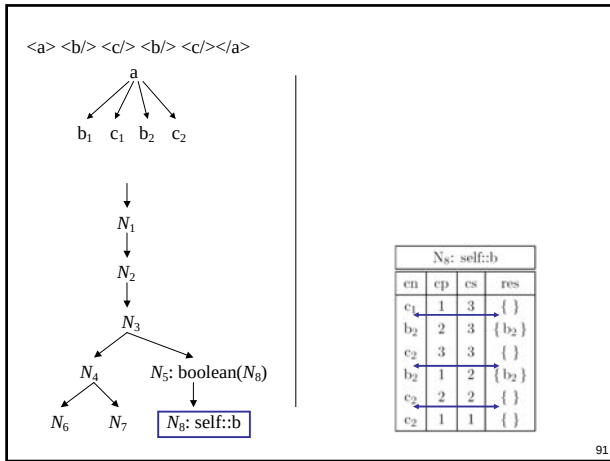
N4: N6 != N7			
cn	cp	cs	res
c1	1	3	3
b2	2	3	3
c2	3	3	3
b2	1	2	1
c2	2	2	2
c2	1	1	1

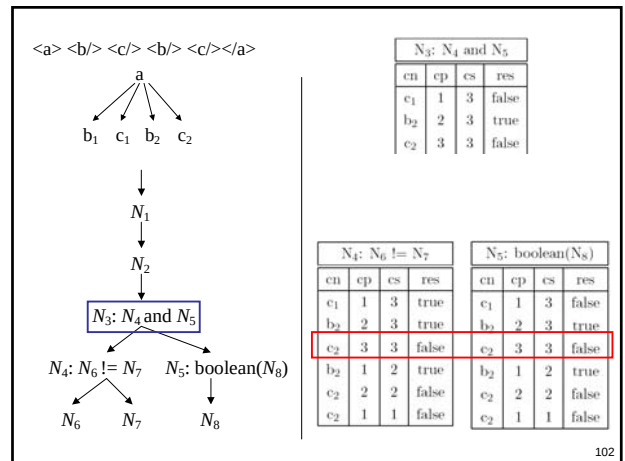
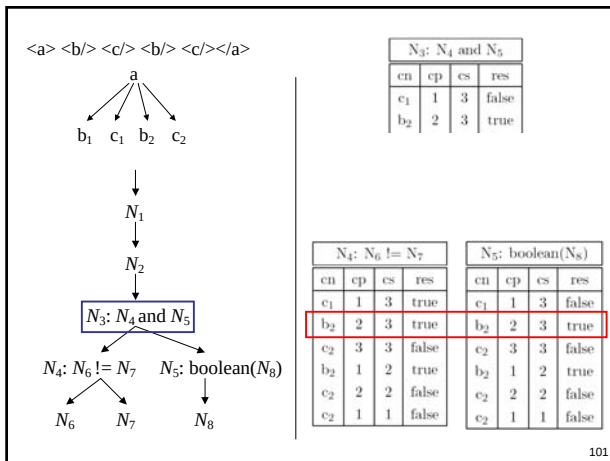
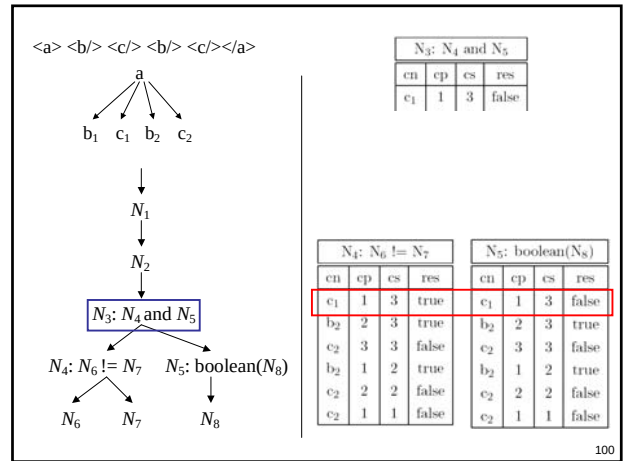
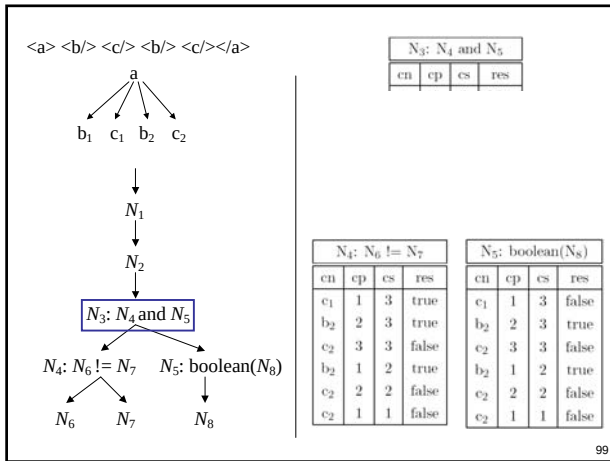
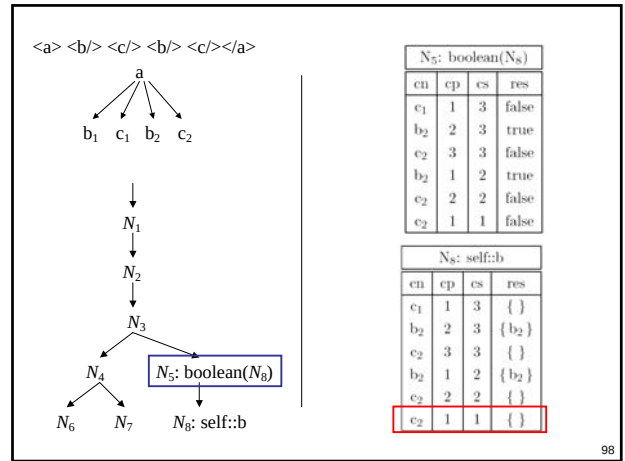
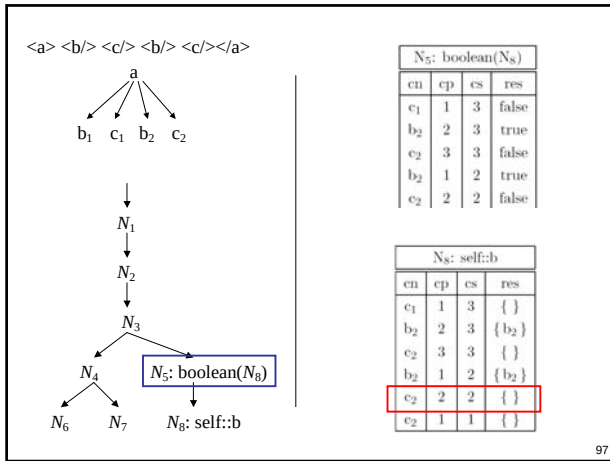
N6: position()			
cn	cp	cs	res
c1	1	3	1
b2	2	3	2
c2	3	3	3
b2	1	2	1
c2	2	2	2
c2	1	1	1

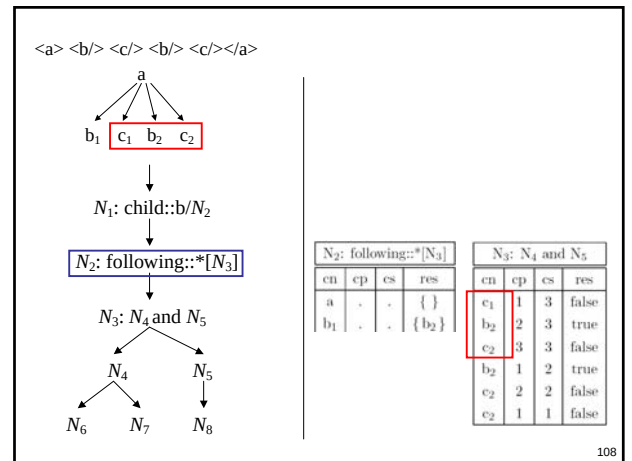
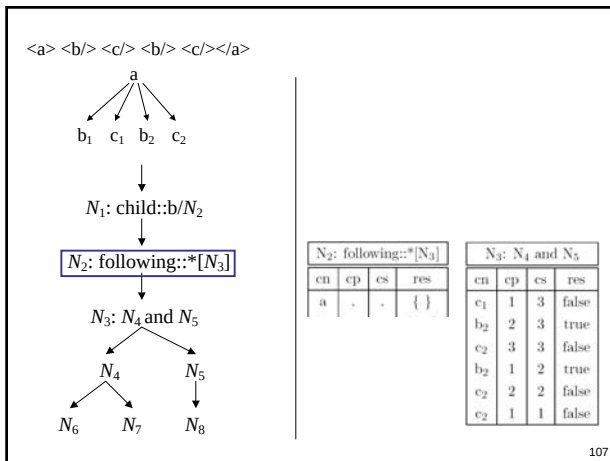
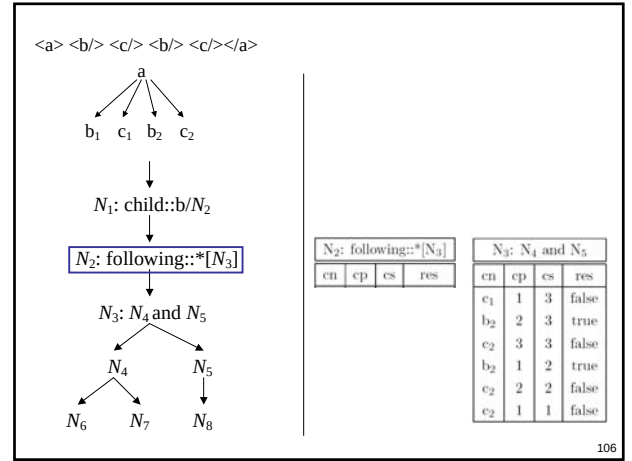
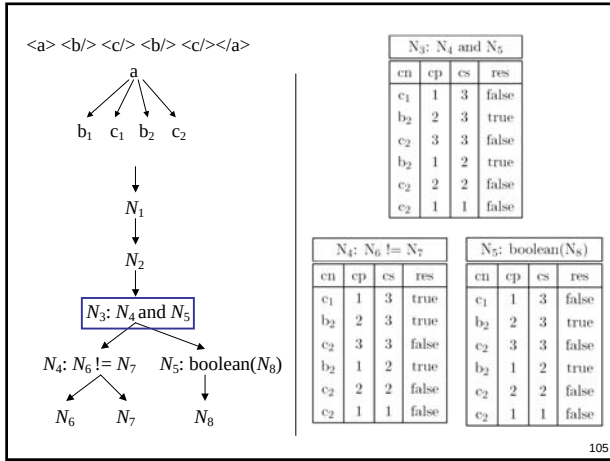
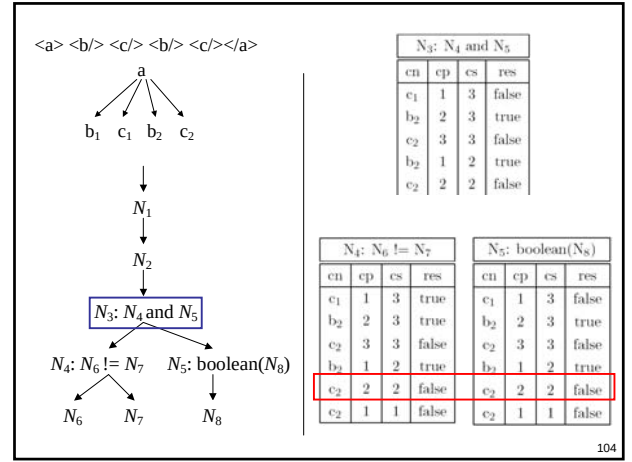
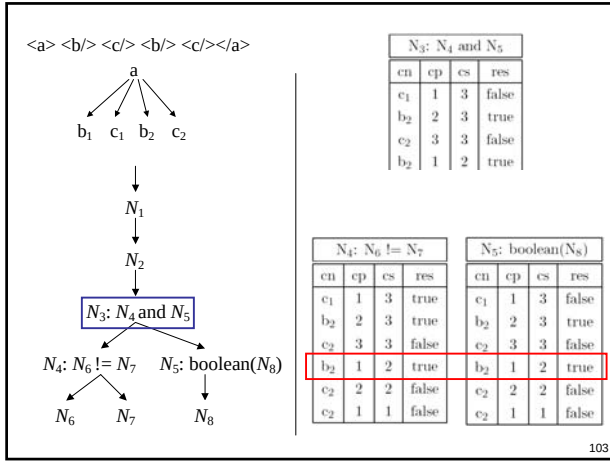
N7: last()			
cn	cp	cs	res
c1	1	3	3
b2	2	3	3
c2	3	3	3
b2	1	2	2
c2	2	2	2
c2	1	1	1

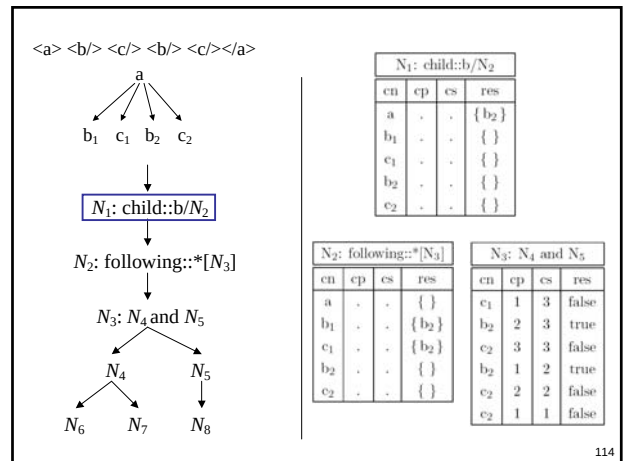
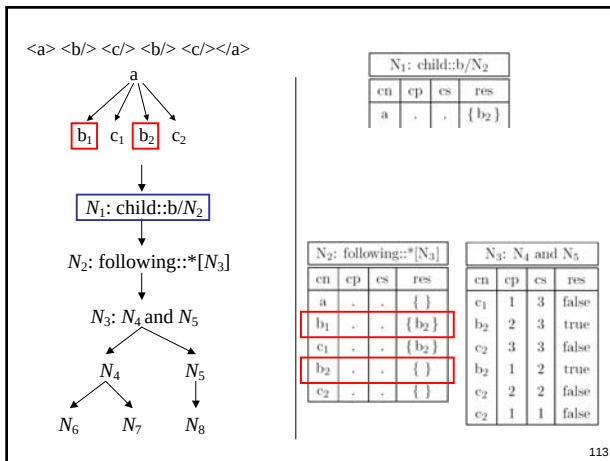
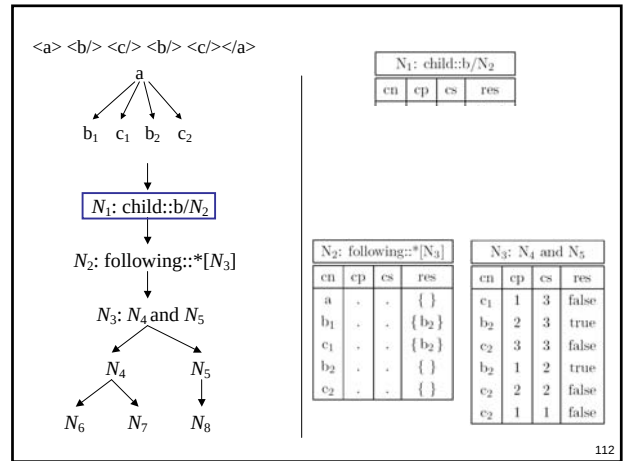
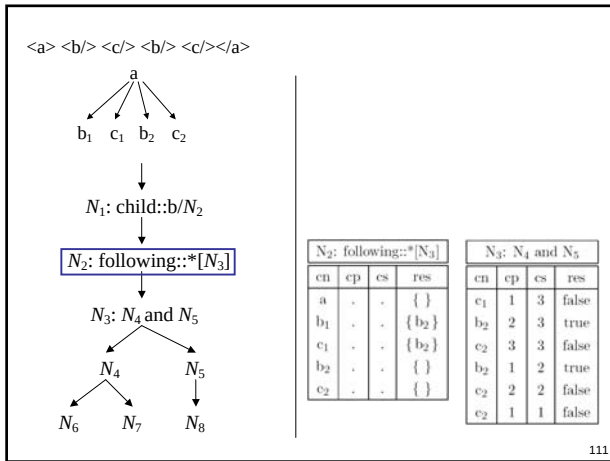
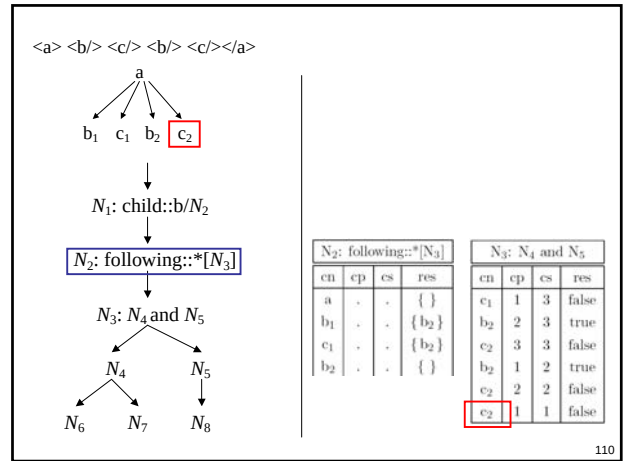
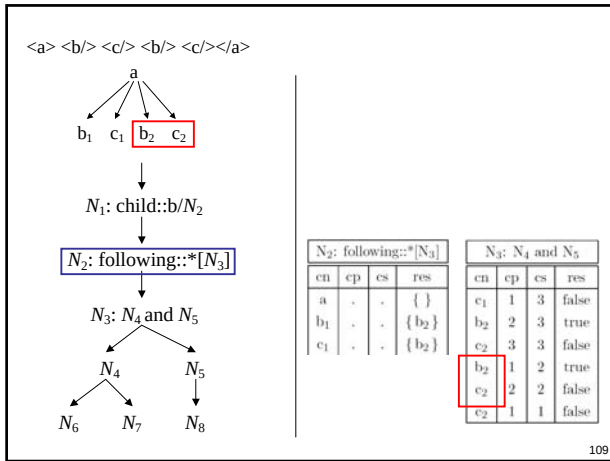
84











Context-Value Table Principle

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

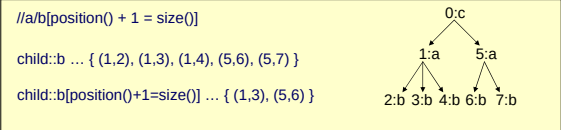
Efficiency of the PTIME Algorithm

- 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.

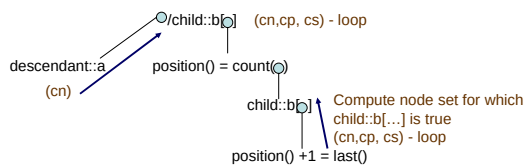


- Complexity lowered to time $O(|data|^4 * |query|^2)$, space $O(|data|^3 * |query|^2)$.

118

Context Simplification Technique

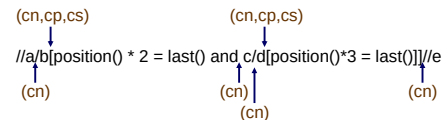
- Only materialize relevant context.
- Core Xpath evaluation algorithm for outermost and innermost paths $//a/b/c/d[...]/e[...](a/b/c)$.
- Treating “position” and “size” in a loop.
 - Because of tree shape of query, loops never have to be nested.



119

Linear Space Fragment

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



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

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 $O(|data|^2 * |query|^2)$.

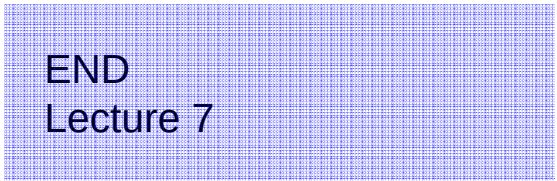
Wadler fragment

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

Core XPath

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

121



END
Lecture 7

122