XML and Databases
XPath evaluation (3)

Kim.Nguyen@nicta.com.au

Week 10

Recap from last week

1. Using automata to run queries in streaming

Recap from last week

1. Using automata to run queries in streaming
2. Handling filters with upward axes

Recap from last week

1. Using automata to run queries in streaming
2. Handling filters with upward axes

Recap from last week

1. Using automata to run queries in streaming
2. Handling filters with upward axes

Today

- How to add preceding-sibling/following-sibling ?
- What data structures to use for automata ?

Following-sibling

Fundamentally, not very different from `child`!

In a pre-order traversal:

`child` : From a node, go `firstChild` then `nextSibling`, . . . , `nextSibling` until `NULL` is found

`following-sibling` : From a node, go `nextSibling`, . . . , `nextSibling` until `NULL` is found

Following-sibling

Fundamentally, not very different from `child`!

In a pre-order traversal:

`child` : From a node, go `firstChild` then `nextSibling`, ..., `nextSibling` until `NULL` is found

`following-sibling` : From a node, go `nextSibling`, ..., `nextSibling` until `NULL` is found

What does it mean in terms of automata?

Following-sibling

Fundamentally, not very different from `child`!

In a pre-order traversal:

`child` : From a node, go `firstChild` then `nextSibling`, ..., `nextSibling` until `NULL` is found

`following-sibling` : From a node, go `nextSibling`, ..., `nextSibling` until `NULL` is found

What does it mean in terms of automata?

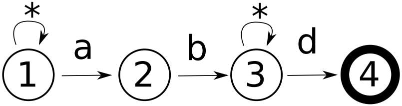
Add a new *kind* of transition

Example

```
//a/b//d
```

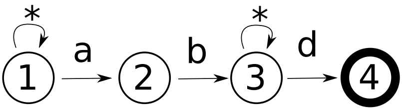
Example

`//a/b//d`



Example

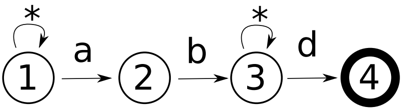
`//a/b//d`



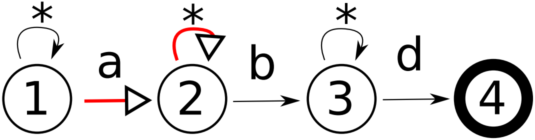
`//a/following-sibling::b//d`?

Example

`//a/b//d`

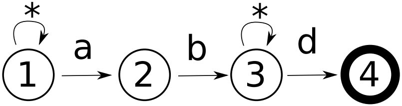


`//a/following-sibling::b//d?`

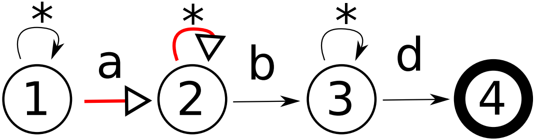


Example

`//a/b//d`



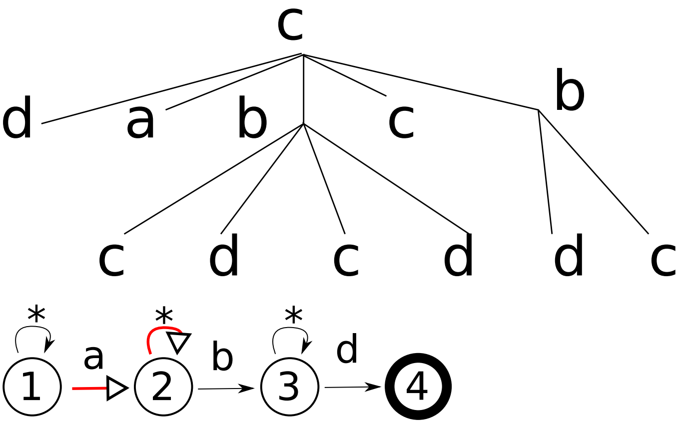
`//a/following-sibling::b//d`?

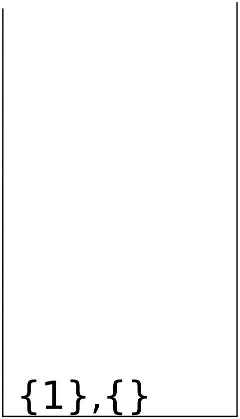


When we evaluate an automaton we can perform two kinds of transitions:

- When doing a "first child" move, we take black transitions (`Down`)
- When doing a "next sibling" move, we take red transitions (`Right`)

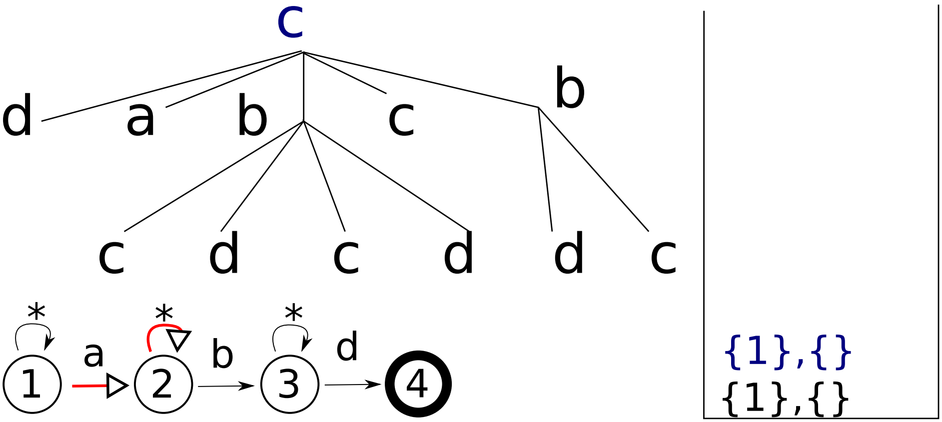
Example





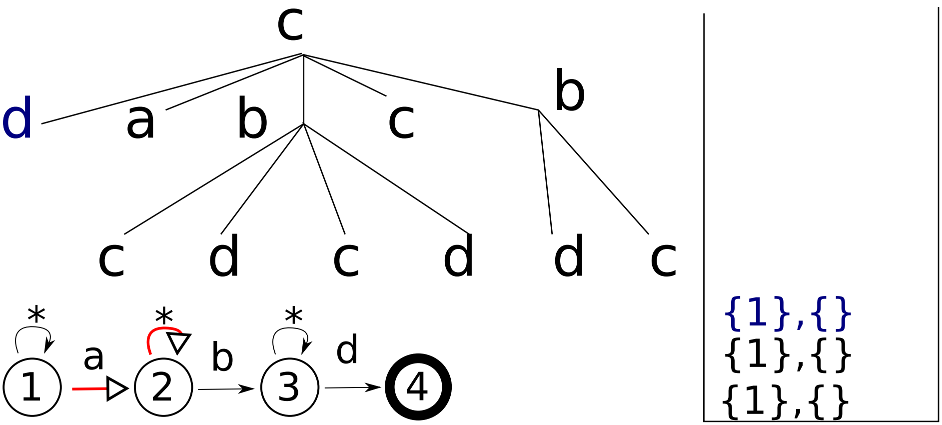
Initially, the stack contains the initial state

Example



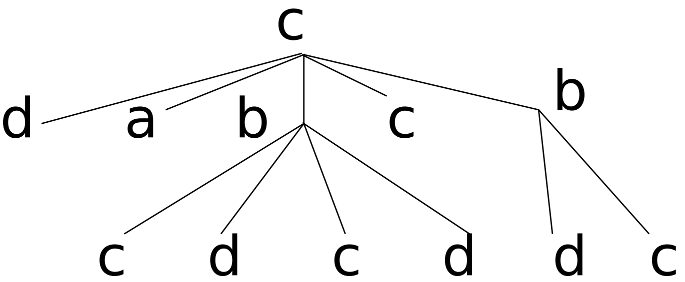
`startElement("c")`, one `Down` transition, no `Right` transition

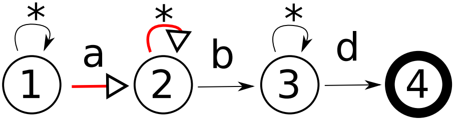
Example



`startElement("d")`, one `Down` transition, no `Right` transition

Example

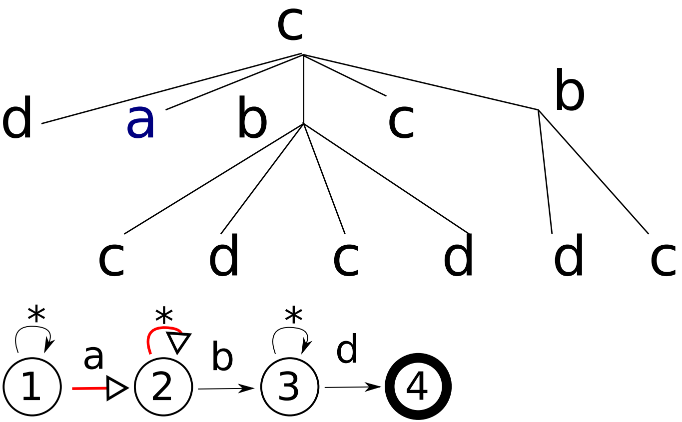


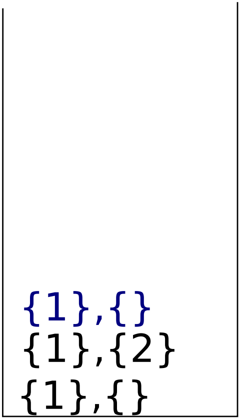


{1},{}
{1},{}

endElement("d"), replace last-sibling with the top of the stack

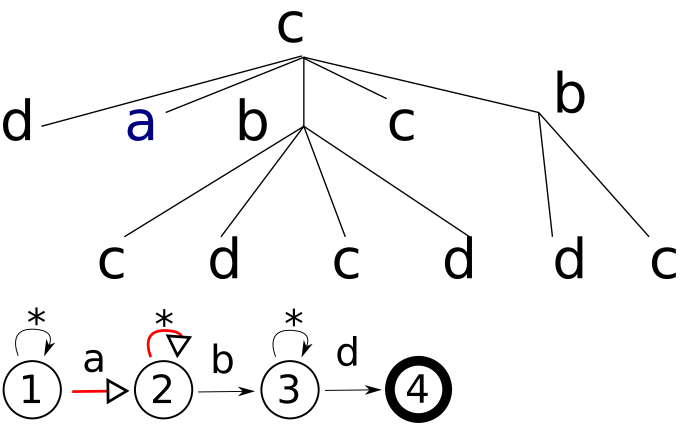
Example

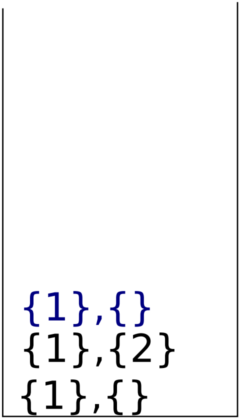




`startElement("a")`, one `Down` transition, one `Right` transition

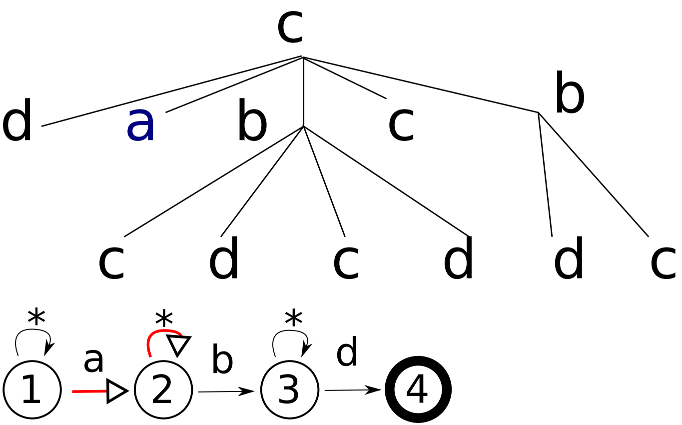
Example

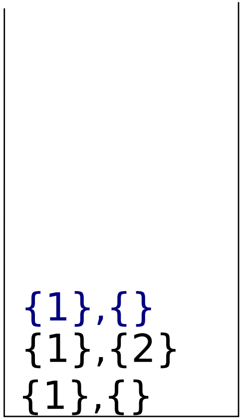




the Right transition goes to state 2, update the right of the stack

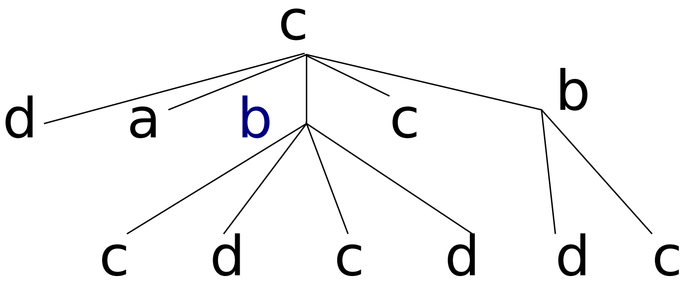
Example

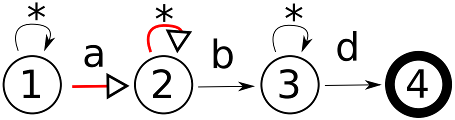




the `Down` transition goes to state 1, pushed on the stack

Example

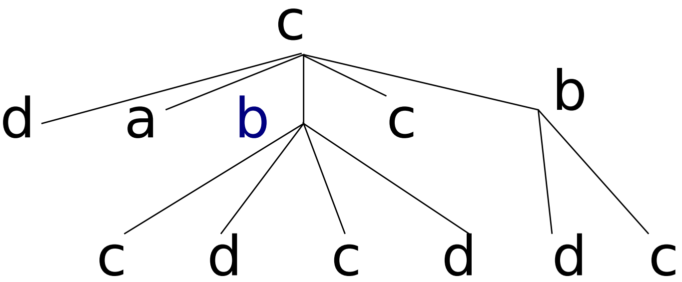


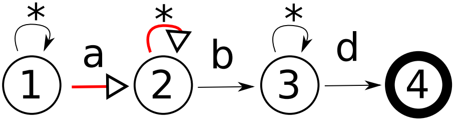


{1,3},{}
{1},{2}
{1},{}

`endElement("a")`

Example

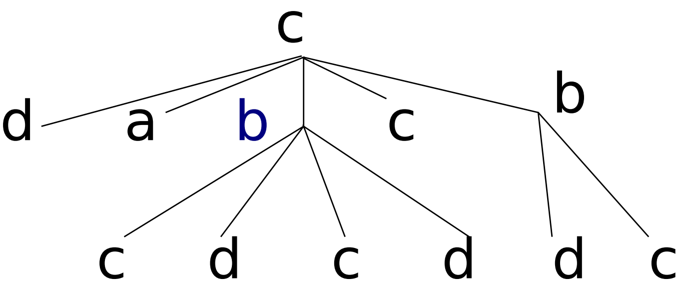


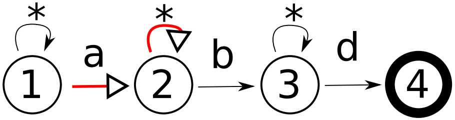


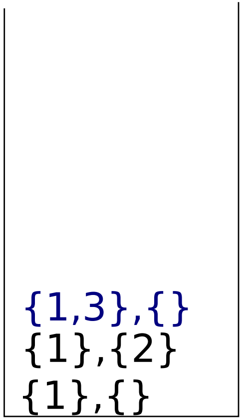
{1,3},{}
{1},{2}
{1},{}

From $\{1\} \cup \{2\}$ compute the b transition

Example

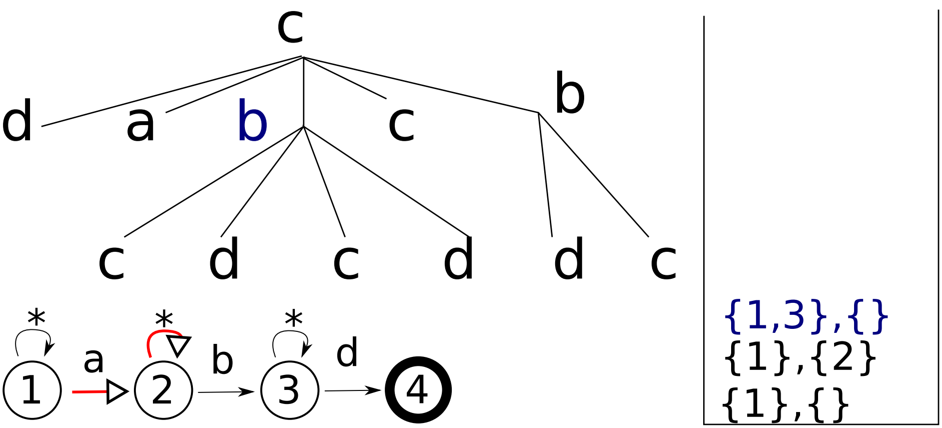






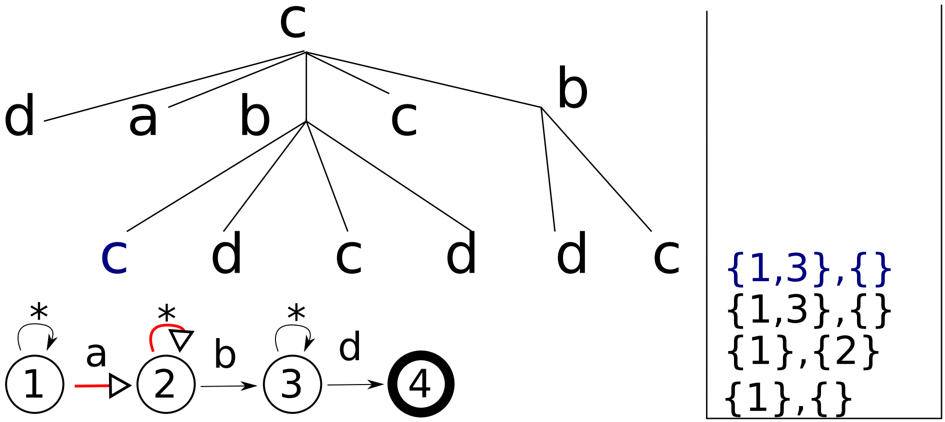
One Right (stay in state 2), replace right part of stack

Example

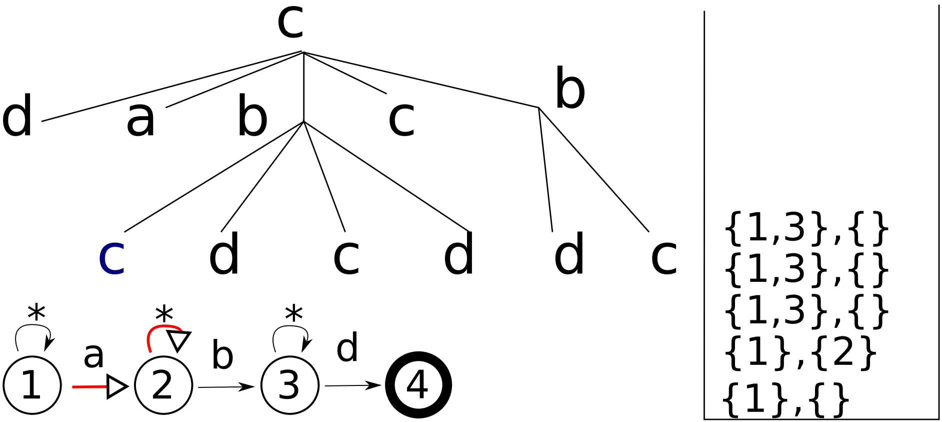


Two `Down` (stay ins state 1, go to state 3), push on the stack

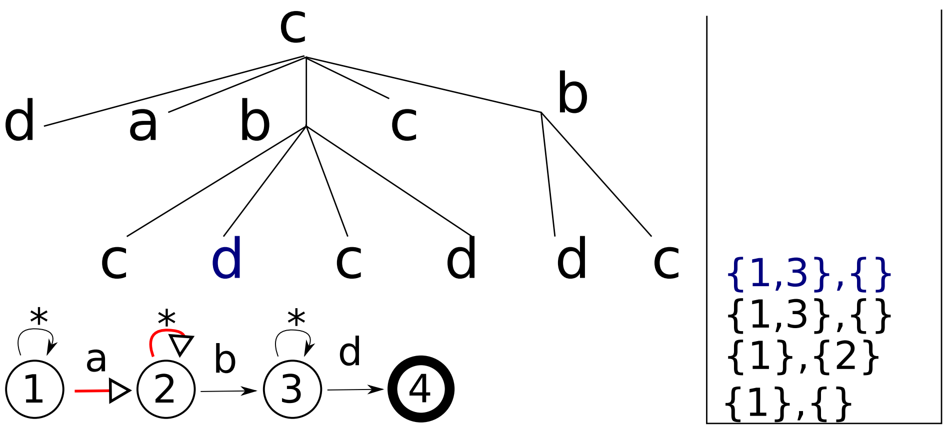
Example



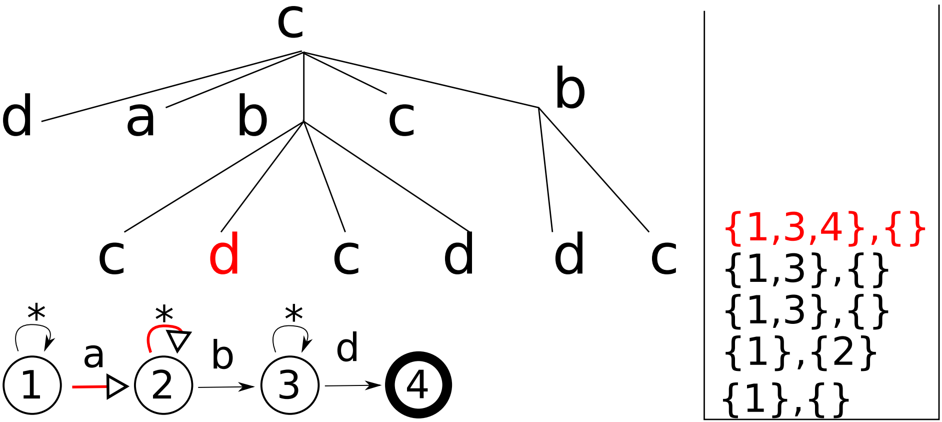
Example



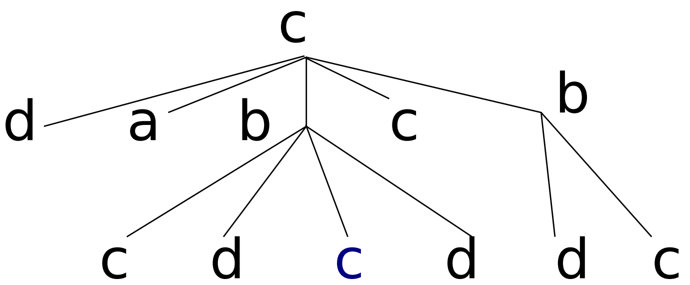
Example

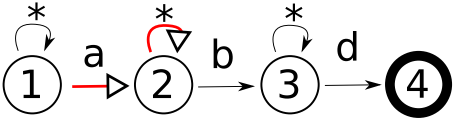


Example



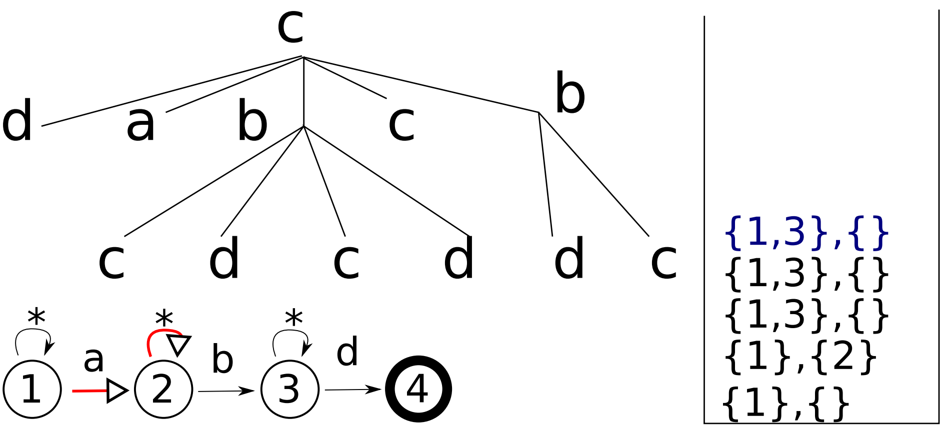
Example



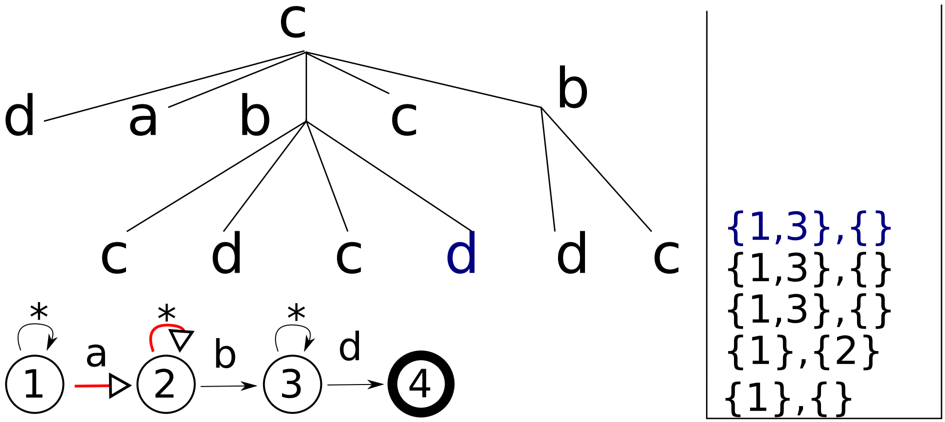


{1,3},{}
{1,3},{}
{1},{2}
{1},{}

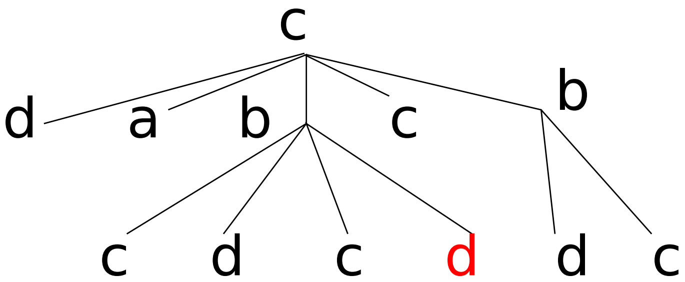
Example

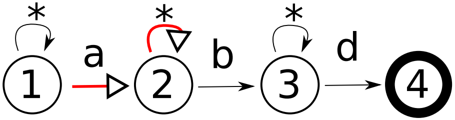


Example



Example





{1,3,4},{}
{1,3},{}
{1,3},{}
{1},{2}
{1},{}

Path with `following-sibling` and no filters

Adapt last week's algorithm:

- keep a stack of pairs of sets of states
- the first set of states represents the states of the parent
- the second set of states represents the states of the previous-sibling

Path with `following-sibling` and no filters

Adapt last week's algorithm:

- keep a stack of pairs of sets of states
- the first set of states represents the states of the parent
- the second set of states represents the states of the previous-sibling

More precisely, on `startElement`

1. Take the top of the stack $S_{\text{parent}}, S_{\text{presib}}$;
2. Compute the union $S = S_{\text{parent}} \cup S_{\text{presib}}$
3. Compute two new sets S'_{parent} and S'_{presib}
4. S'_{parent} is the set of states that can be reached from S with a `Down` transition S'_{presib} is the set of states that can be reached by a `Right` transition
5. At the top of the stack, replace S_{presib} with S'_{presib}
6. Push $S'_{\text{parent}}, \{\}$ at the top of the stack

Adding preceding-sibling in filters

Consider `//a//b[./parent::c/preceding-sibling::d]/c`.
What can we say about the `b` nodes ?

- They must have a parent `c`
- The parent must have a preceding-sibling `d`

This is true for all the nodes which are:

- below a `c`
- which is a following sibling of a `d`
- which can occur anywhere

$\Rightarrow$ `//d/following-sibling::c/*`
Only need to adapt last week's algorithm to following-sibling

Some more examples of rewriting of filters

`[./ancestor::d/preceding-sibling::e/parent::f]`
becomes
`//f/e/following-sibling::d//*`

Some more examples of rewriting of filters

```
[./ancestor::d/preceding-sibling::e/parent::f]
```
becomes
```
//f/e/following-sibling::d//*
```

```
[./preceding-sibling::e/ancestor::d/preceding-sibling::f]
```
becomes
```
//f/following-sibling::d//e/folowing-sibling::*
```

Some more examples of rewriting of filters

`[./ancestor::d/preceding-sibling::e/parent::f]`
becomes
`//f/e/following-sibling::d//*`

`[./preceding-sibling::e/ancestor::d/preceding-sibling::f]`
becomes
`//f/following-sibling::d//e/folowing-sibling::*`
What is the general rule ?

Rewriting backward filters

Let `[ f ]` be a filter with backward axes. We rewrite it into a path d. d and f do *NOT* compute the same results but, for every node selected by d, `[ f ]` is `true`. Let:

$$f = ./a_0\texttt{::}t_0/a_1\texttt{::}t_1/\ldots/a_n\texttt{::}t_n$$

where $a_i \in \{\texttt{parent}, \texttt{ancestor}, \texttt{preceding-sibling}\}$ andn t_i is a label or $*$.

$$//t_n/\bar{a}_n\texttt{::}t_{n-1}/\ldots/\bar{a}_0\texttt{::}*$$

where $\bar{a}_i$ is the inverse axis of a_i (`parent` is the inverse of `child`, `descendant` the inverse of `ancestor` and `preceding-sibling` the inverse of `following-sibling`)