XML and Databases

Exam Preparation
Part 3

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2010

(4)[4] Consider a (pre,post) table: Given a pre-order number x, the mapping post(x) returns the post-order of the node with pre-order x. Write pseudo code that, for a node p, prints pre-numbers of

a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute in an optimal way all the preceding nodes of p_1,…,p_n.

Let pre be numbered

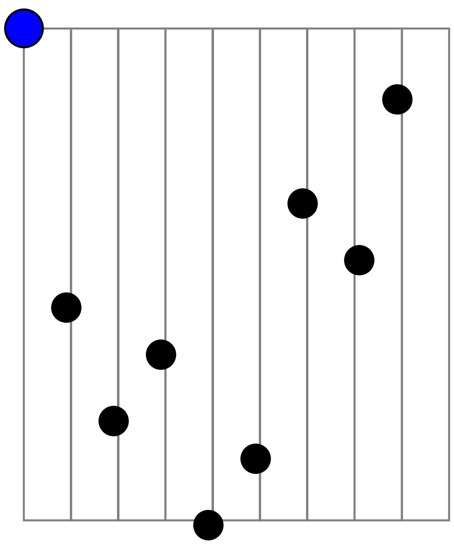
1,2,3,..,MaxPre

```
void printDescendants(int pre){

    for(int i = pre+1; (i<MaxPre &&
                        post(i)<post(pre)); i++)
      print(i);
}
```

(4)[4] Consider a (pre,post) table: Given a pre-order number x,
the mapping post(x) returns the post-order of the node with pre-order x.
Write pseudo code that, for a node p, prints pre-numbers of
a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute
in an optimal way all the preceding nodes of p_1,…,p_n.

Let pre be numbered

1,2,3,..,MaxPre



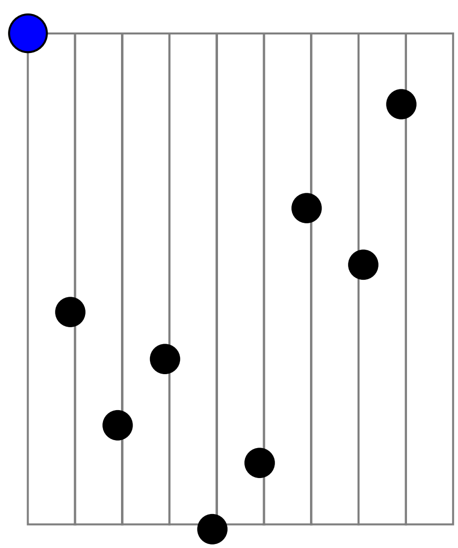
```
void printChildren(int pre){

    int barrier = 0;

    for(int i = pre+1; (i<MaxPre &&
                post(i)<post(pre)); i++)

      if(post(i)>barrier){
        print(i);
        barrier = post(i)
      }
}
```

(4)[4] Consider a (pre,post) table: Given a pre-order number x, the mapping post(x) returns the post-order of the node with pre-order x. Write pseudo code that, for a node p, prints pre-numbers of

a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute in an optimal way all the preceding nodes of p_1,…,p_n.

Let pre be numbered

1,2,3,..,MaxPre

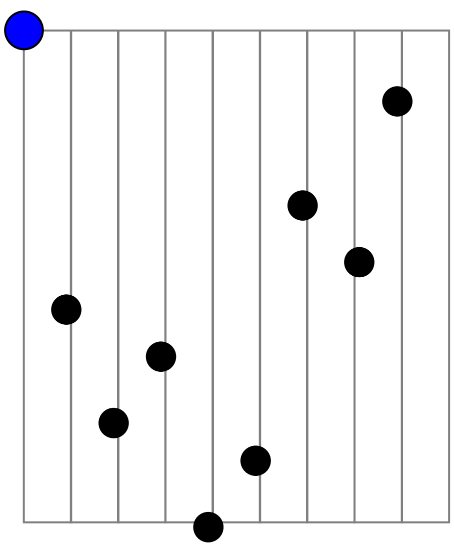


```
void followingSiblings(int pre){

    int barrier = post(pre);

    for(int i = pre+1; i<MaxPre; i++)

     if(post(i)>barrier){
        print(i);
        barrier = post(i)
      }
}
```

```
(4)[4] Consider a (pre,post) table: Given a pre-order number x,
the mapping post(x) returns the post-order of the node with pre-order x.
Write pseudo code that, for a node p, prints pre-numbers of
a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute
in an optimal way all the preceding nodes of p_1,…,p_n.
```

Let pre be numbered

1,2,3,..,MaxPre



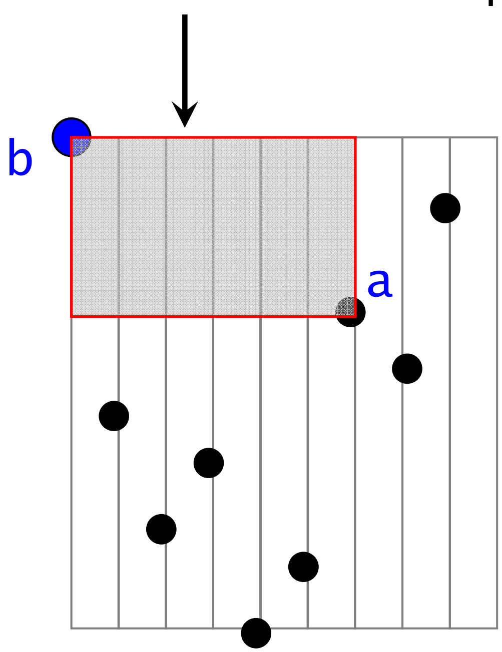
```
void twoAway(int pre){

    for(int i = 1; i<MaxPre; i++)

     if(!isChild(i,pre) &&
        !isChild(pre,i))
       print(i);
}
```

→ Descendants of children
→ (Following plus Preceding) without parent

```
(4)[4] Consider a (pre,post) table: Given a pre-order number x,
the mapping post(x) returns the post-order of the node with pre-order x.
Write pseudo code that, for a node p, prints pre-numbers of
```

```
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute
in an optimal way all the preceding nodes of p_1,…,p_n.
```

If a is child of b,
then this must be empty!

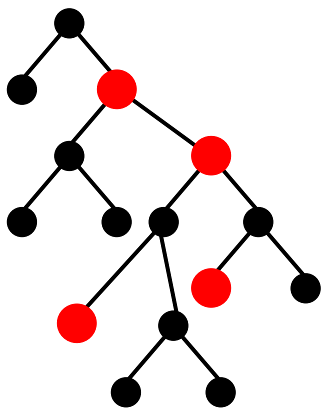



```
void twoAway(int pre){

    for(int i = 1; i<MaxPre; i++)

     if(!isChild(i,pre) &&
        !isChild(pre,i))
       print(i);
}
Boolean isChild(int a, int b){
  if not(a>b && post(a)<post(b))
     return false;
  for(int i=a--; i>b; i--)
    if(post(i)>post(a)) return false;
  return true
}
```

```
(4)[4] Consider a (pre,post) table: Given a pre-order number x,
the mapping post(x) returns the post-order of the node with pre-order x.
Write pseudo code that, for a node p, prints pre-numbers of
a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute
in an optimal way all the preceding nodes of p_1,…,p_n.
```

---

f) Take the node with largest pre-value (p_n)
and compute the preceding nodes of that!



Formally, first, take all lowest independent nodes.
Second, out of those, pick the one with maximal pre-number.

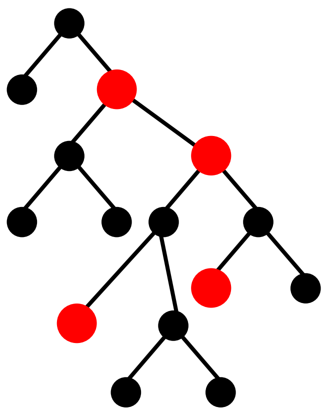
(this coincides with simply picking the node with
largest pre number)

```
(4)[4] Consider a (pre,post) table: Given a pre-order number x,
the mapping post(x) returns the post-order of the node with pre-order x.
Write pseudo code that, for a node p, prints pre-numbers of
a) its descendants
b) its children
c) its following-siblings
d) all following nodes that are leaves
e) all nodes that are at least two edges away from p
f) Given a sequence of nodes p_1,…,p_n in pre-order, how can you compute
in an optimal way all the preceding nodes of p_1,…,p_n.
```

---

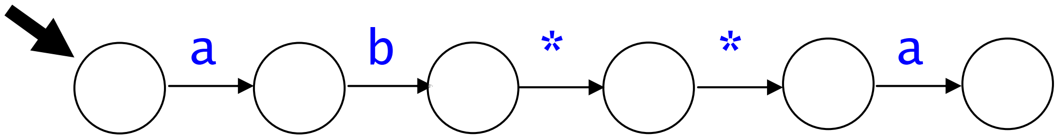
f) Take the node with largest pre-value (p_n)

and compute the preceding nodes of that!

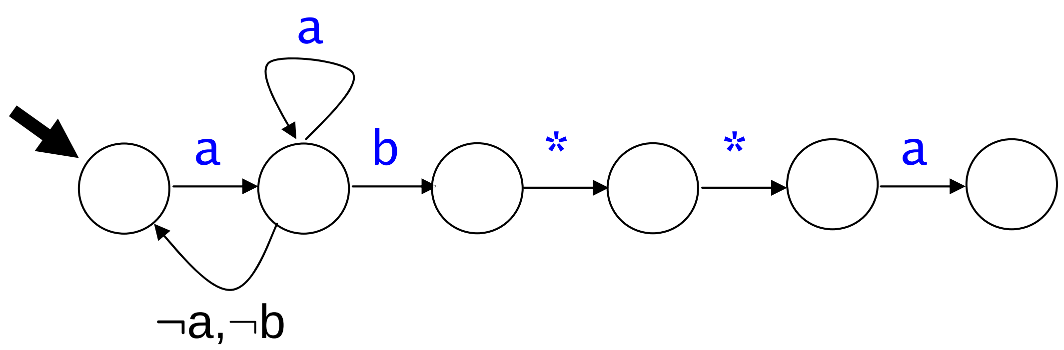


```
void printPreceding_List(List l){
    pre = l.nth(l.length);

    for(int i = pre-1; i>=0; i--)
     if post(i)<post(pre) print(i);
}
```

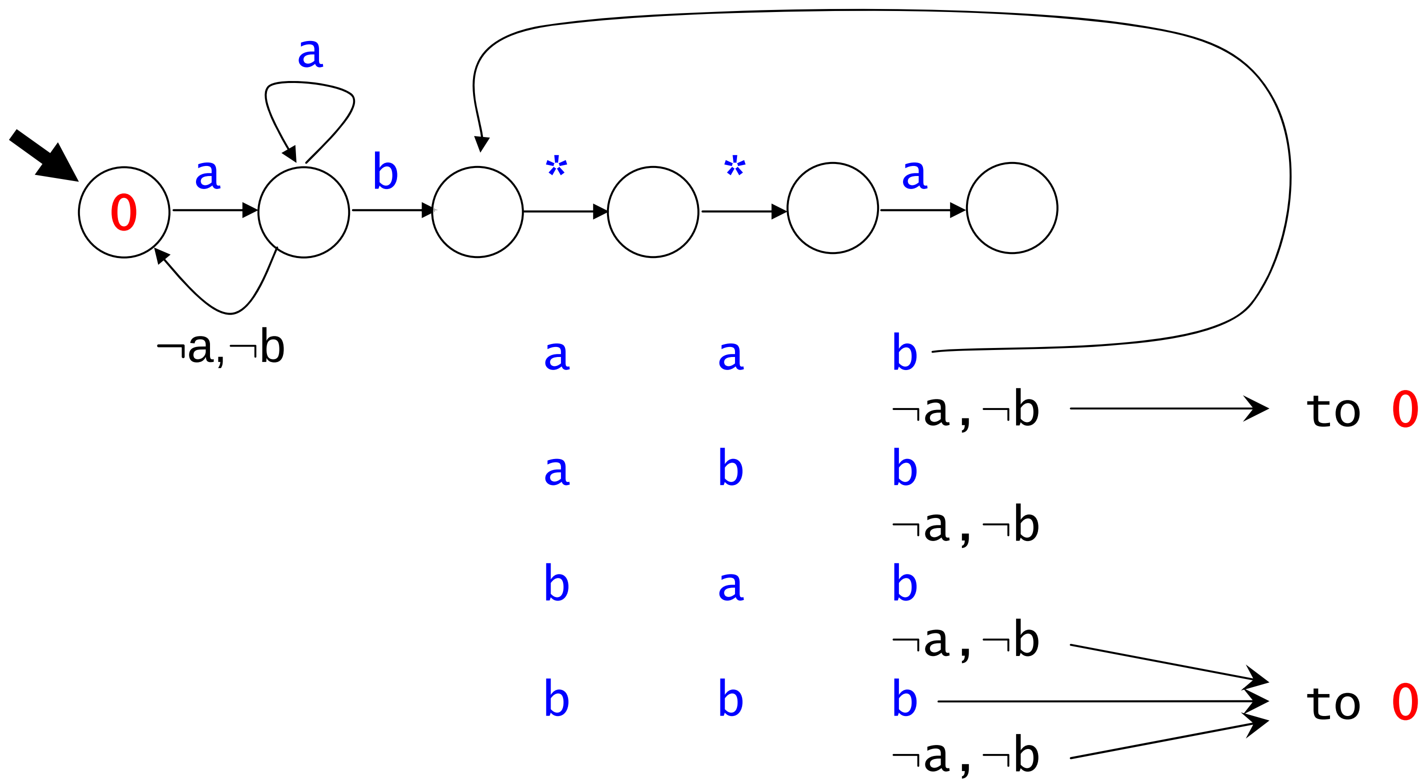
8)[2] Show the "KMP-automaton" for //a/b/*/*/a
and show the automaton run on the input abababa.



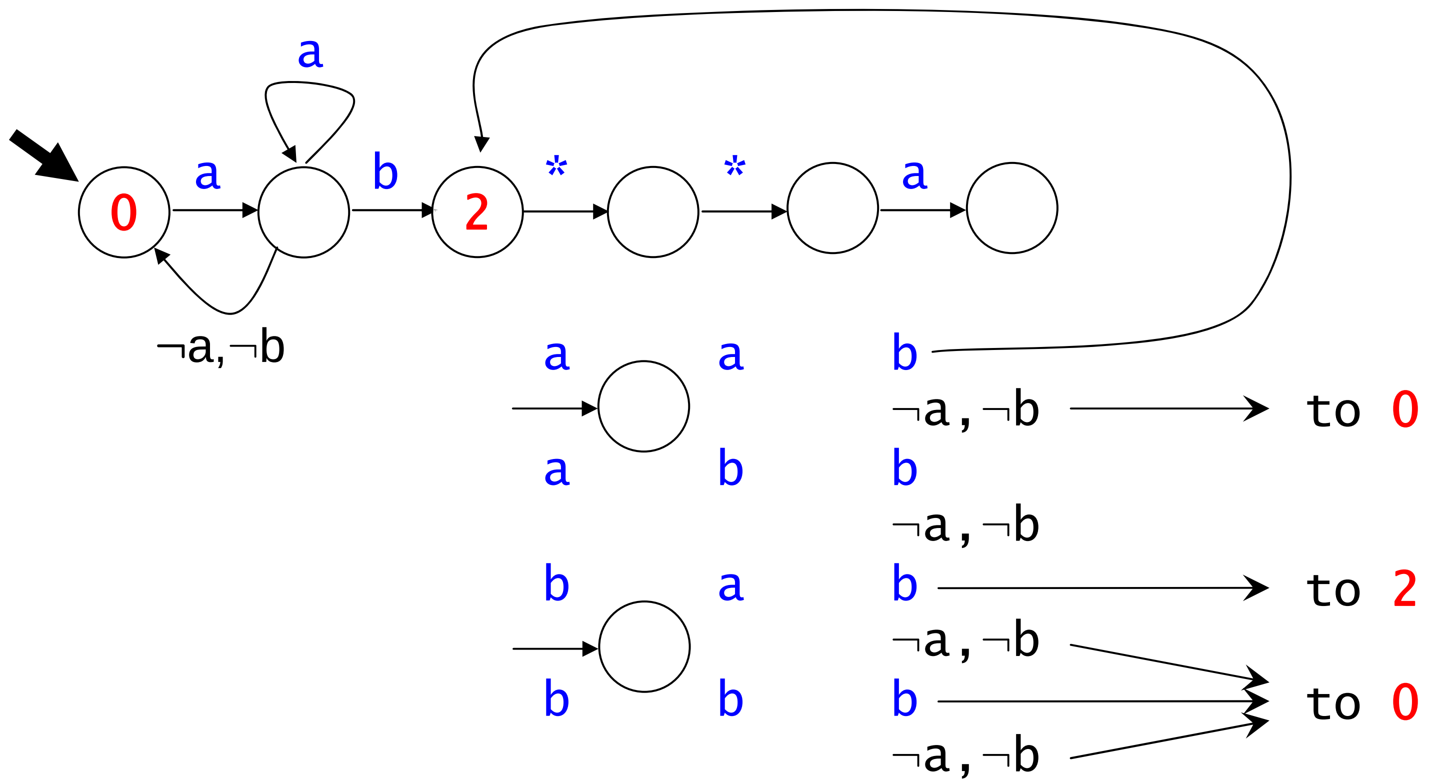
8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.



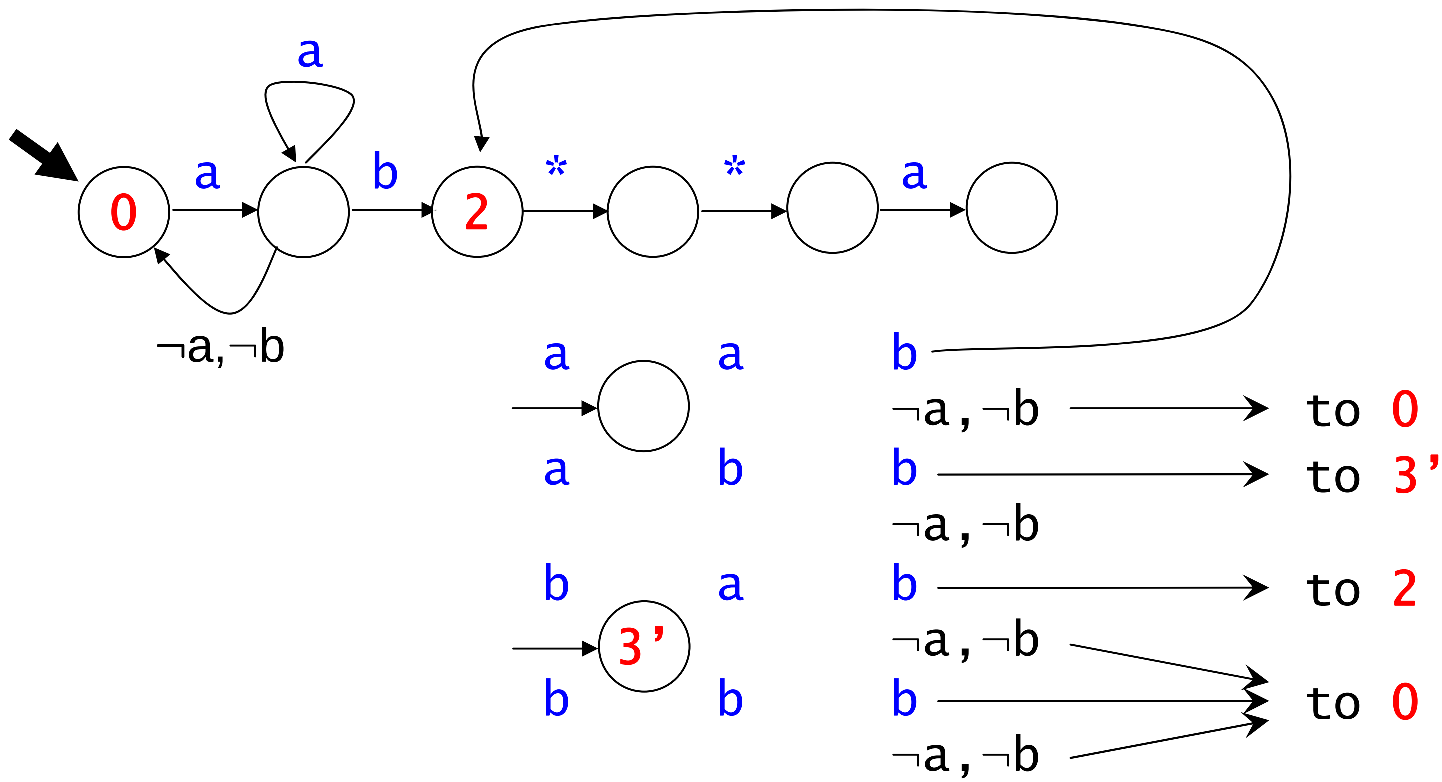
8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.



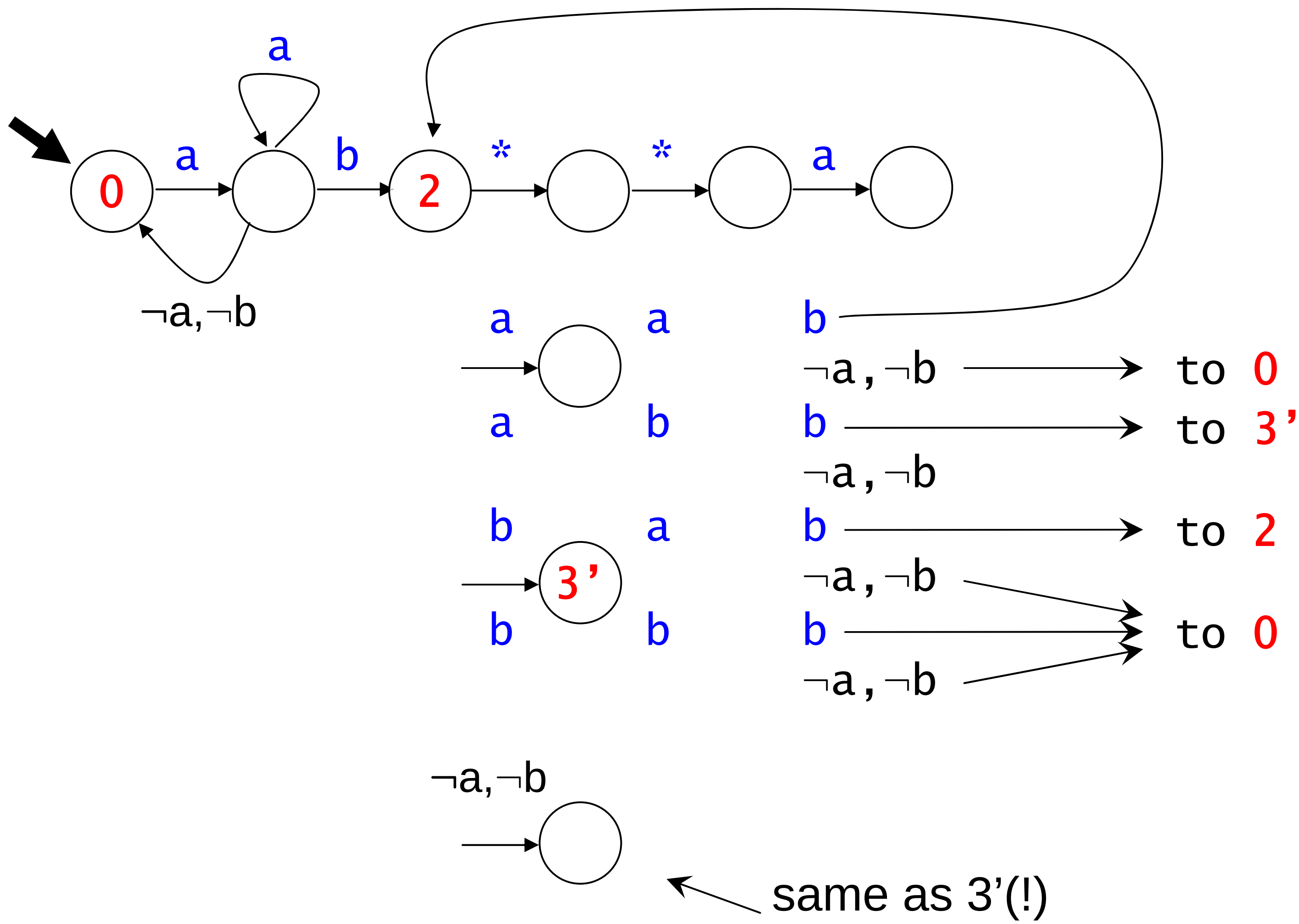
8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.



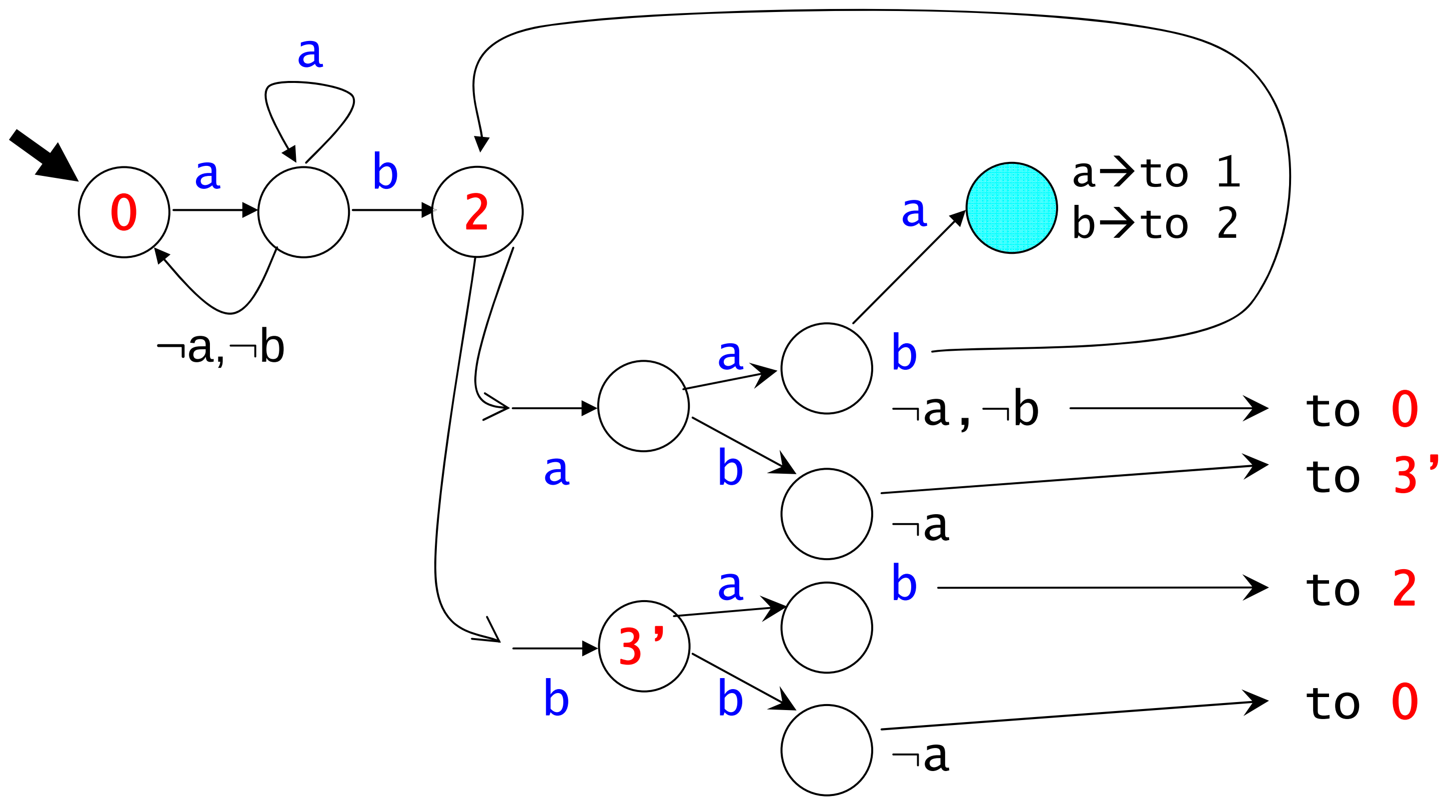
8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.



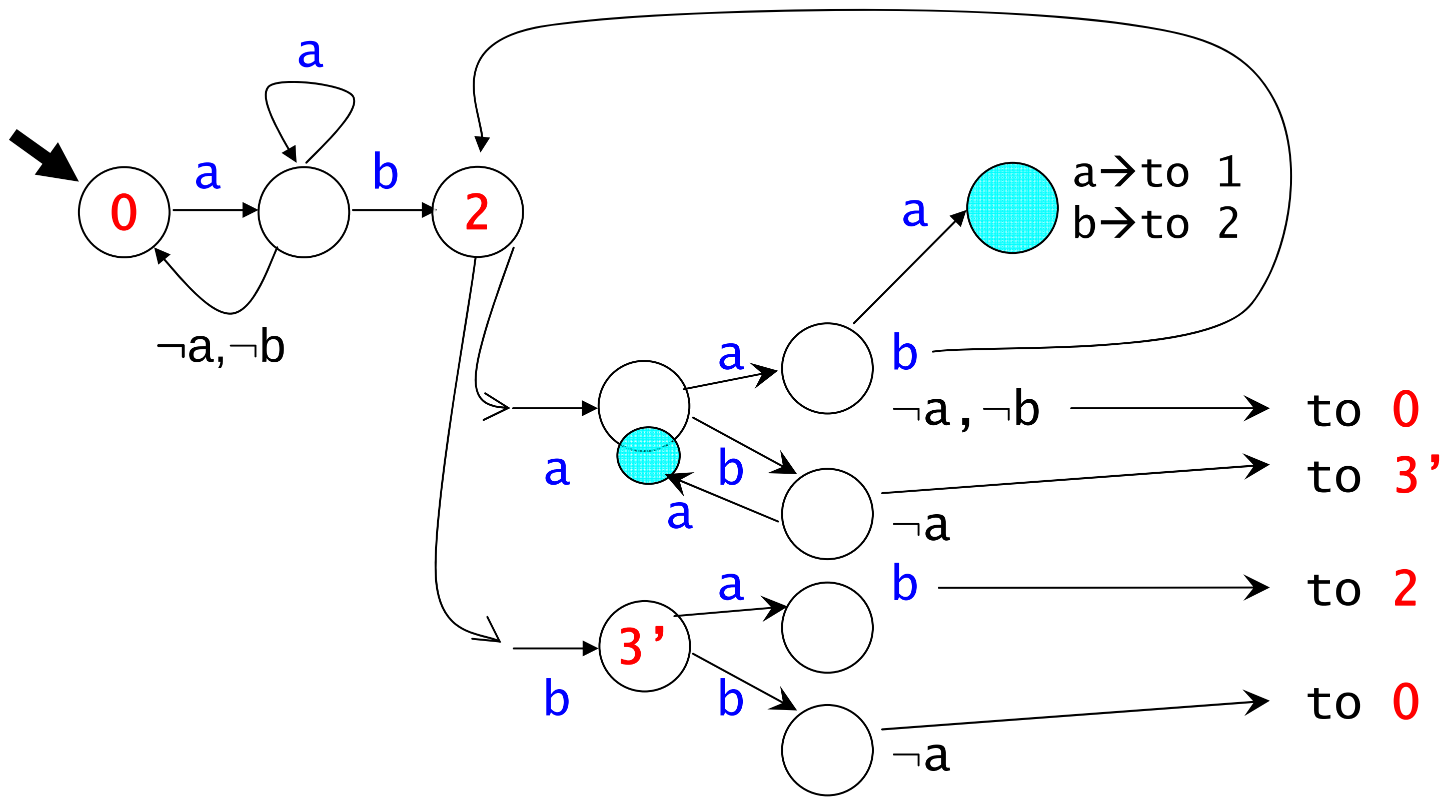
8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.



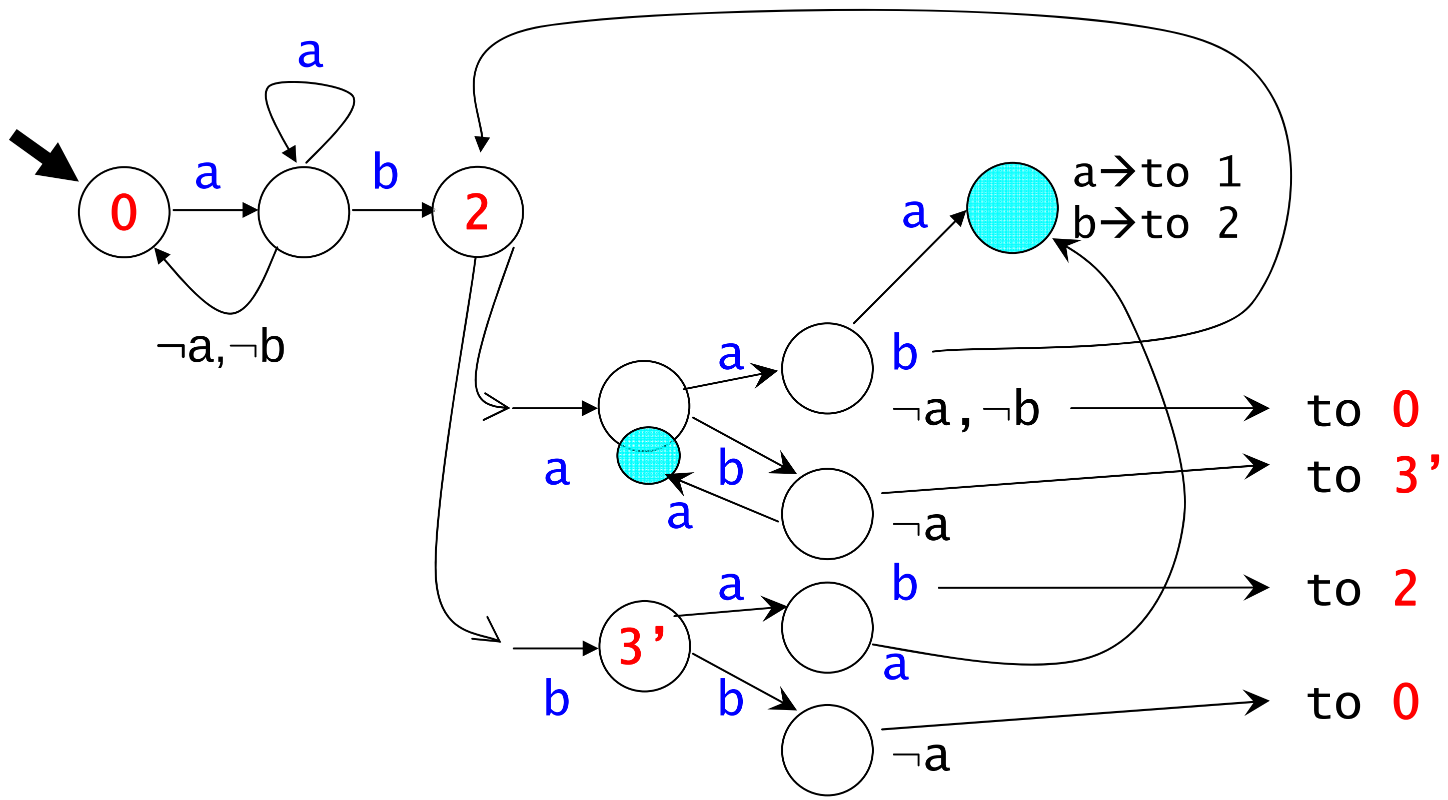
8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.



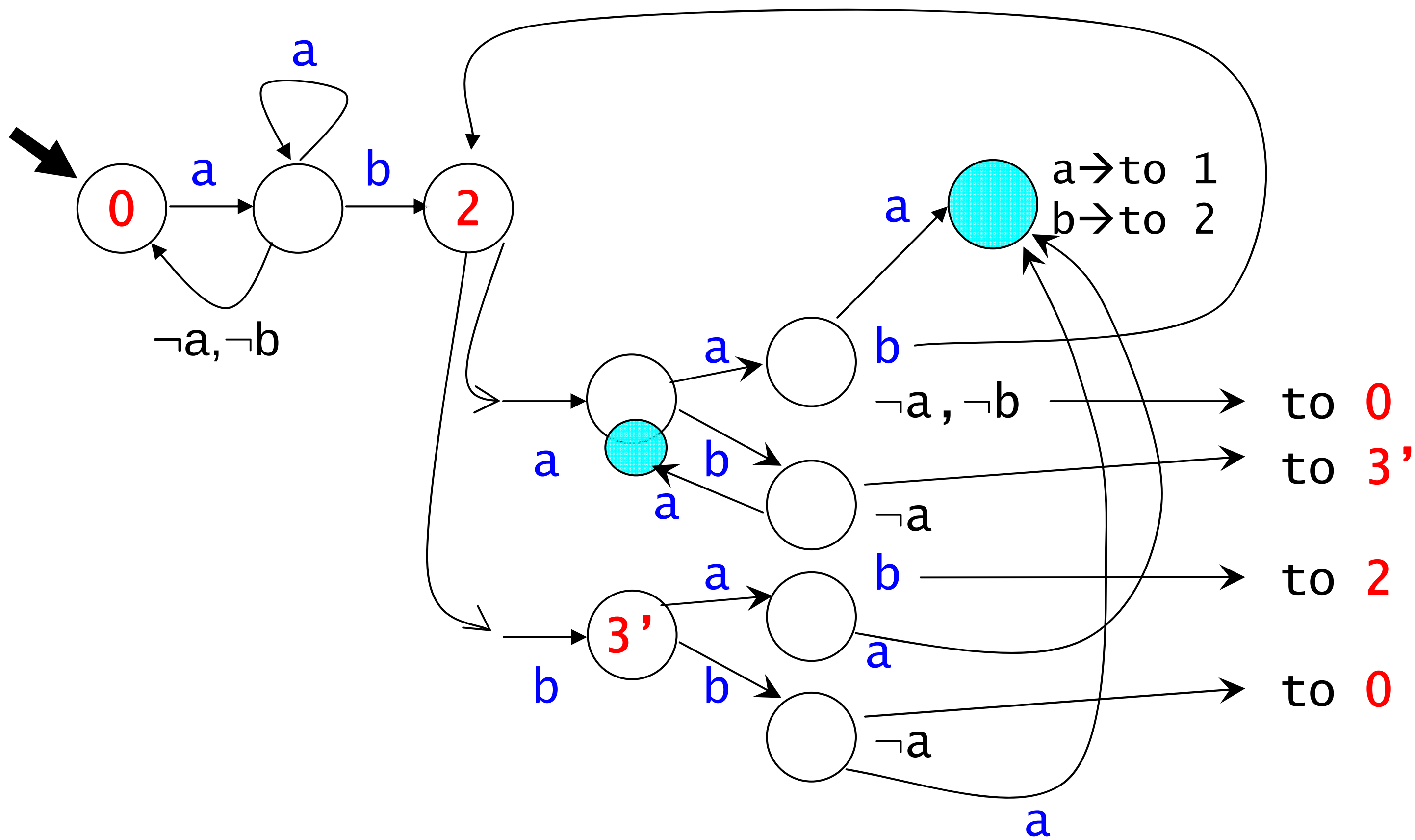
8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.



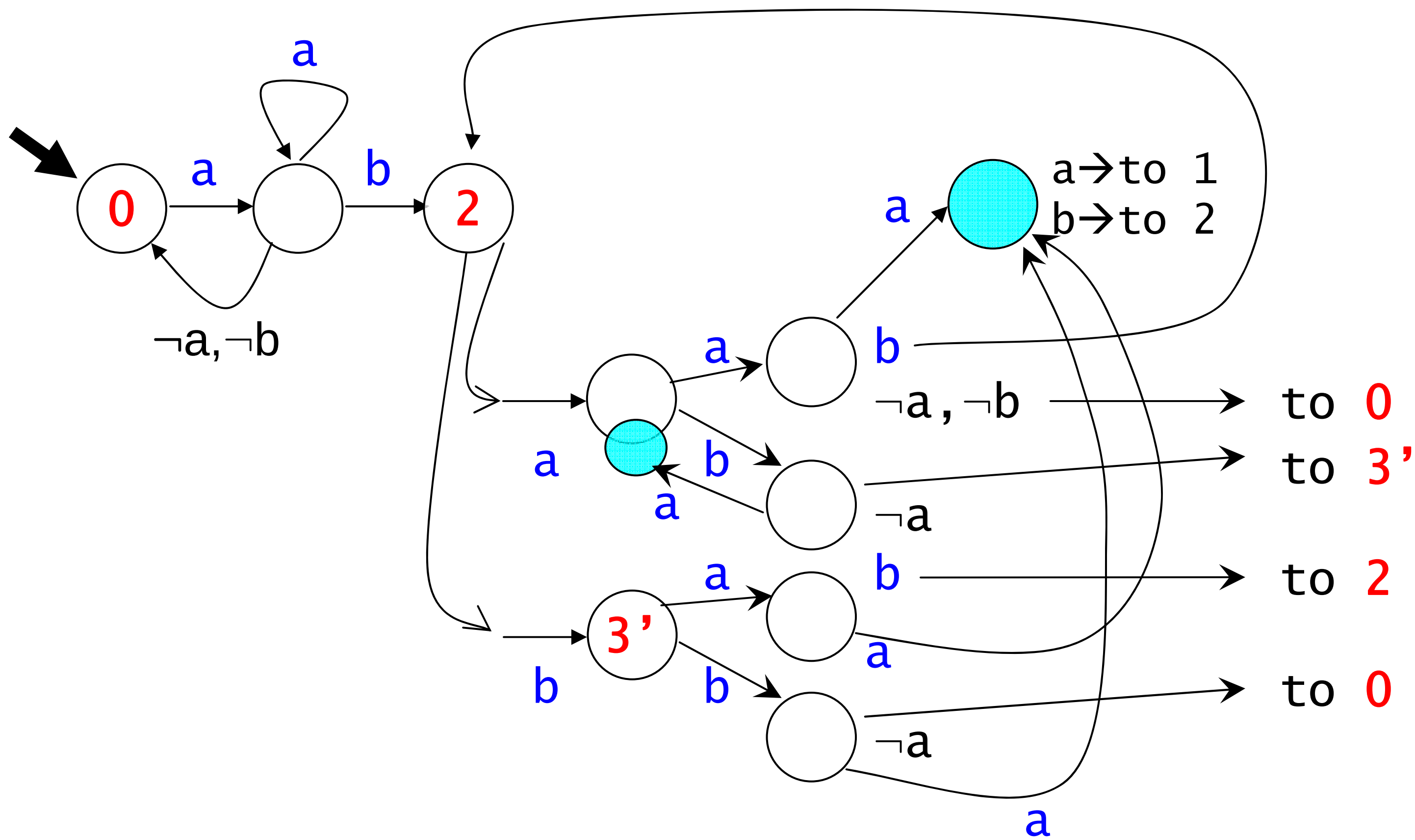
8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.



8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.

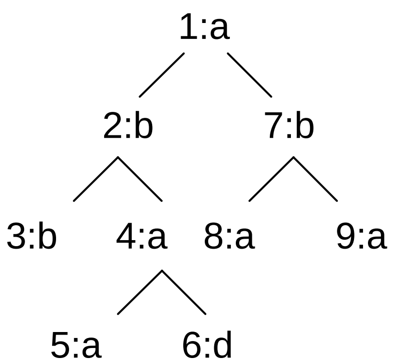


8)[2] Show the “KMP-automaton” for //a/b/*/*/a
and show the automaton run on the input abababa.



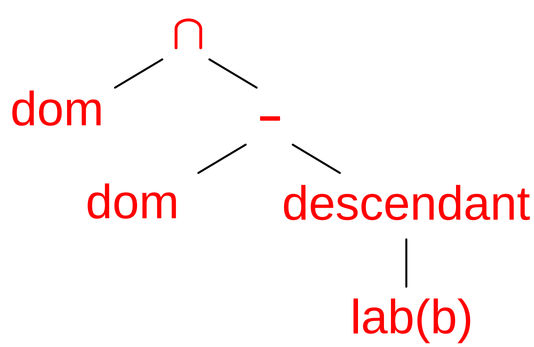
… already finished! ☺

7)[2] Consider the tree T in (6). Show in detail the *bottom-up evaluation* for the query Q = //*[not(ancestor::b)]. First, give for Q the corresponding evaluation tree over ∩, ∪ , lab(b), child, descendant, etc. Then show the actual subsets of { 1, 2, …, 9 } which are selected by the different nodes of the evaluation tree.

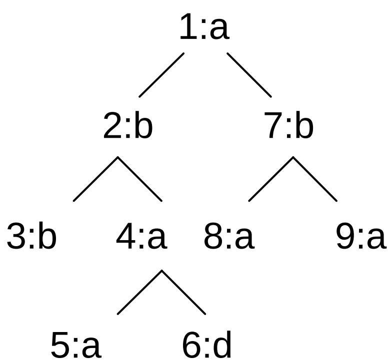


//*[not(ancestor::b)]

Nodes x, such that there is
no ancestor labeled b
➔ Intersect x with nodes that are
not descendants of b-nodes!



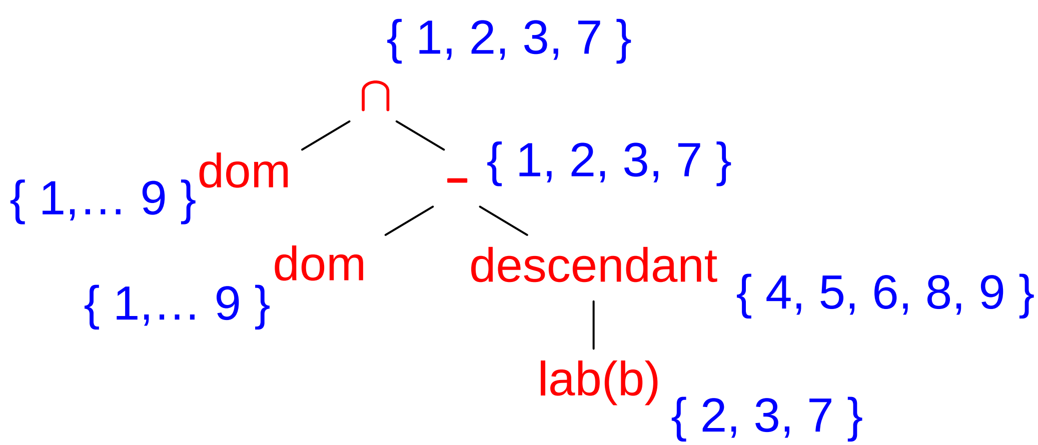
7)[2] Consider the tree T in (6). Show in detail the *bottom-up evaluation* for the query Q = //*[not(ancestor::b)]. First, give for Q the corresponding evaluation tree over ∩, ∪ , lab(b), child, descendant, etc. Then show the actual subsets of { 1, 2, …, 9 } which are selected by the different nodes of the evaluation tree.



//*[not(ancestor::b)]

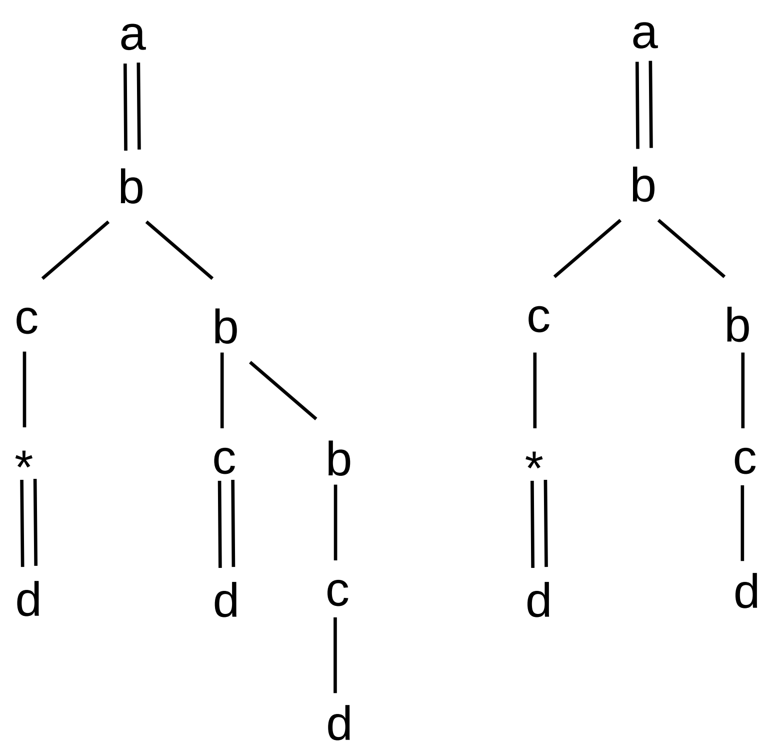
Nodes x, such that there is no ancestor labeled b

➔ Intersect x with nodes that are not descendants of b-nodes!



9)[2] Using the canonical model, show that p is contained in q, for

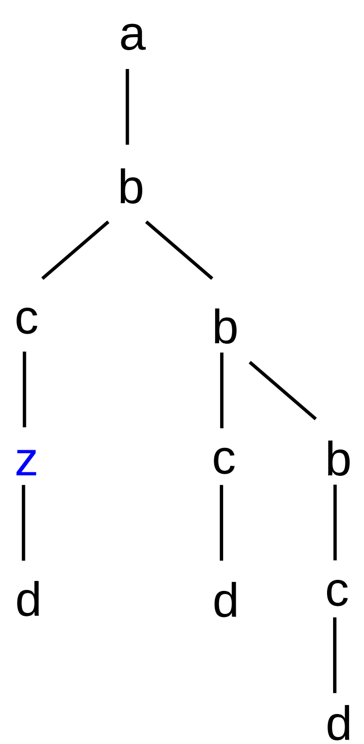
p = /a[.//b[c/*//d]/b[c//d]/b[c/d]]
q = /a[.//b[c/*//d]/b[c/d]]

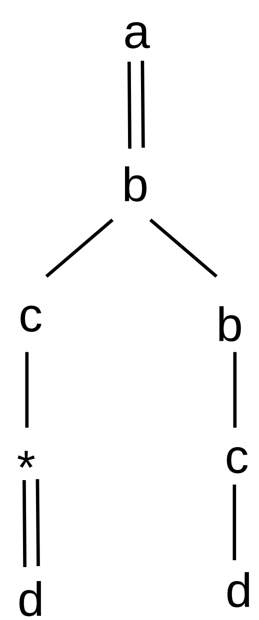


9) **[2]** Using the canonical model, show that p is contained in q, for

p = /a[.//b[c/*//d]/b[c//d]/b[c/d]]

q = /a[.//b[c/*//d]/b[c/d]]



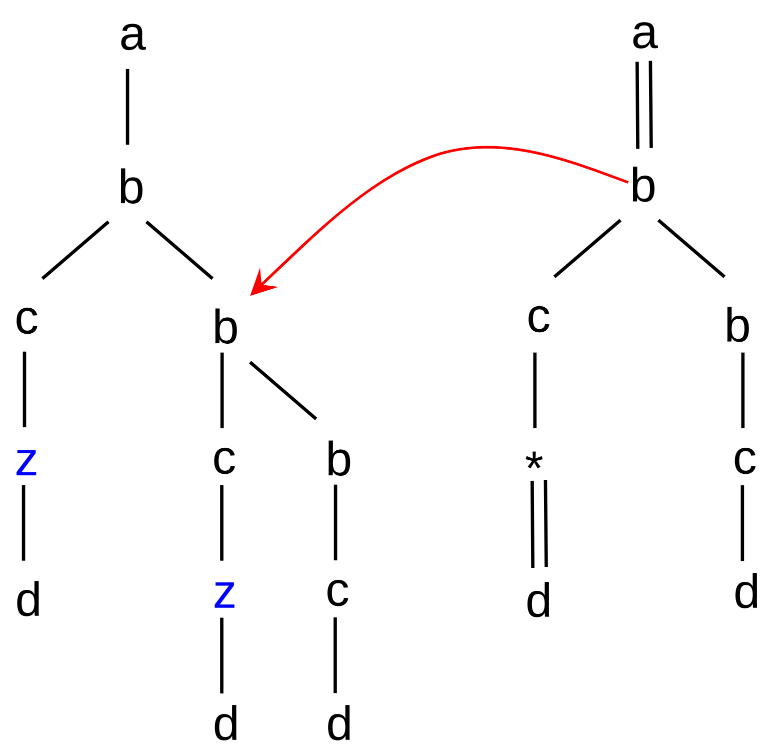


Step 1: in p, replace *'s by z's
Step 2: in p, for every //, replace it by zero or one z

→ tree p[0,0,0]: *is instance of pattern* q? -- Yes!

9) **[2]** Using the canonical model, show that p is contained in q, for

p = /a[.//b[c/*//d]/b[c//d]/b[c/d]]
q = /a[.//b[c/*//d]/b[c/d]]

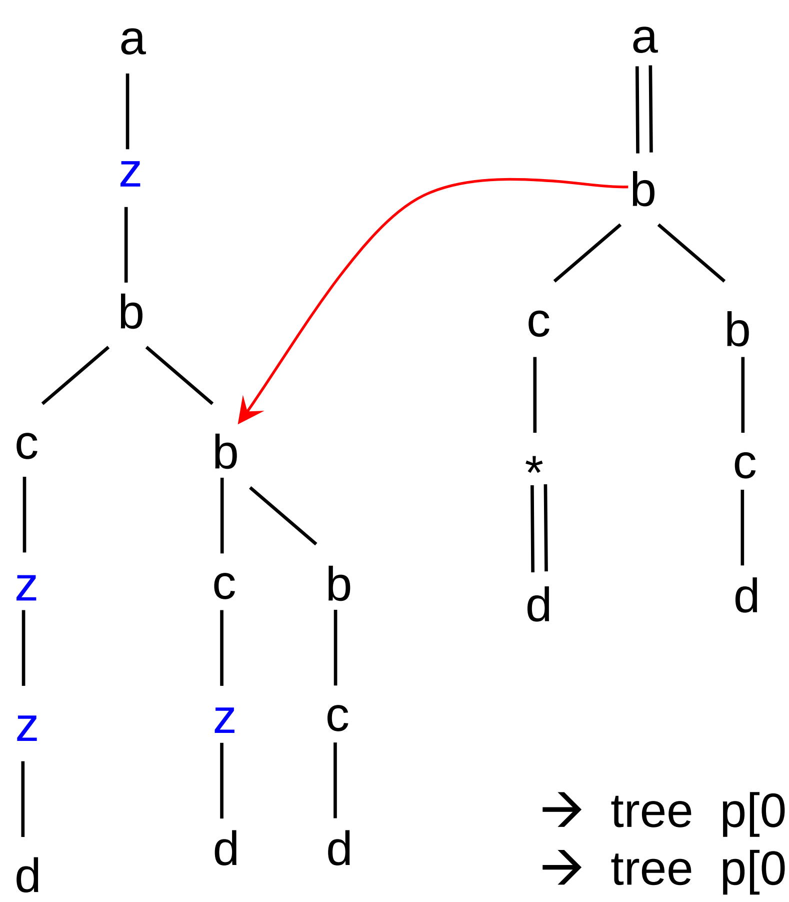


Step 1: in p, replace *'s by z's
Step 2: in p, for every //, replace it by zero or one z

→ tree p[0,0,0]: is instance of pattern q? -- Yes!
→ tree p[0,0,1]: *is instance of pattern* q? -- Yes!

9) **[2]** Using the canonical model, show that p is contained in q, for

p = /a[.//b[c/*//d]/b[c//d]/b[c/d]]
q = /a[.//b[c/*//d]/b[c/d]]



Step 1: in p, replace *'s by z's
Step 2: in p, for every //, replace it by zero or one z

→ tree p[0,0,0]: is instance of pattern q? -- Yes!
→ tree p[0,0,1]: is instance of pattern q? -- Yes!
…
→ tree p[1,1,1]: *is instance of pattern* q?

10)**[5]** a) Why is the Glushkov automaton important for DTDs? How is it used to check whether an XML document is valid for a given DTD?
b) Give the Glushkov automaton for E = (a? b? c?)*
c) How many edges, in terms of m, does the Glushkov automaton have for the expression (a_1? a_2? a_3? … a_m?)* ?
d) For a deterministic expression of length m and an input of length n, how much time is needed to check the input against the expression?
How is this different for general (non deterministic) expressions?
e) It is known that no equivalent deterministic expression exists for E=(a|b)*a(a|b). Show two expressions E1 and E1 such that (E1 | E2) is equivalent to E. (This proves that det. expressions are not closed under union).

---

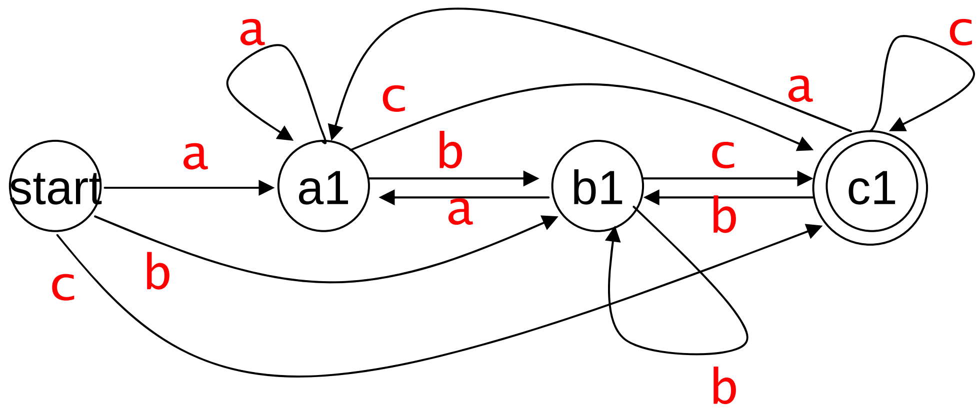
a) Why is the Glushkov automaton important for DTDs?
How is it used to check whether an XML document is valid for a given DTD?

Answer a)

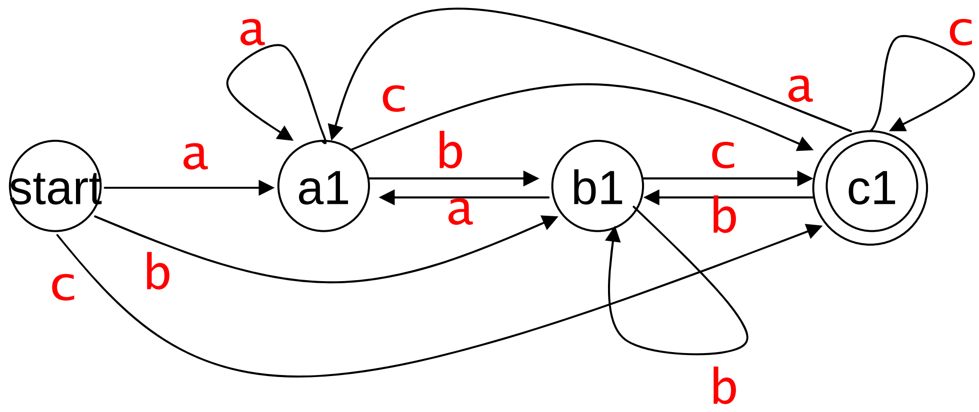
The specification of DTDs requires that every regular expression that appears in a DTD must be 1-unambiguous.
A regular expression is 1-unambiguous, if its Glushkov automaton is deterministic.

Given a DTD **D**, we can built a Glushkov automaton for each regular expression that appears in the **D**.
We validate a given XML document against **D** by a top-down traversal through the document tree. At any node we check whether its sequence of children is recognized by the Glushkov automaton for that label.

b) Give the Glushkov automaton for E = (a? b? c?)*



b) Give the Glushkov automaton for E = (a? b? c?)*



c) How many edges, in terms of m, does the Glushkov automaton have for the expression (a_1? a_2? a_3? … a_m?)* ?

Answer: m(m+1)

(for every state, incuding the initial one, we have m transitions)

d) For a deterministic expression D of length m and an input of length n, how much time is needed to check the input against the expression? How is this different for general (non deterministic) expressions?

Answer

Step 1: construct the Glushkov automaton for E.
This takes time O(m^2).

Step 2: run the automaton over the input.
This takes time O(n).

Total amount of time: O(m^2 + n)

--

For a general expression: O(m^3 * n) or O(2^m * n)