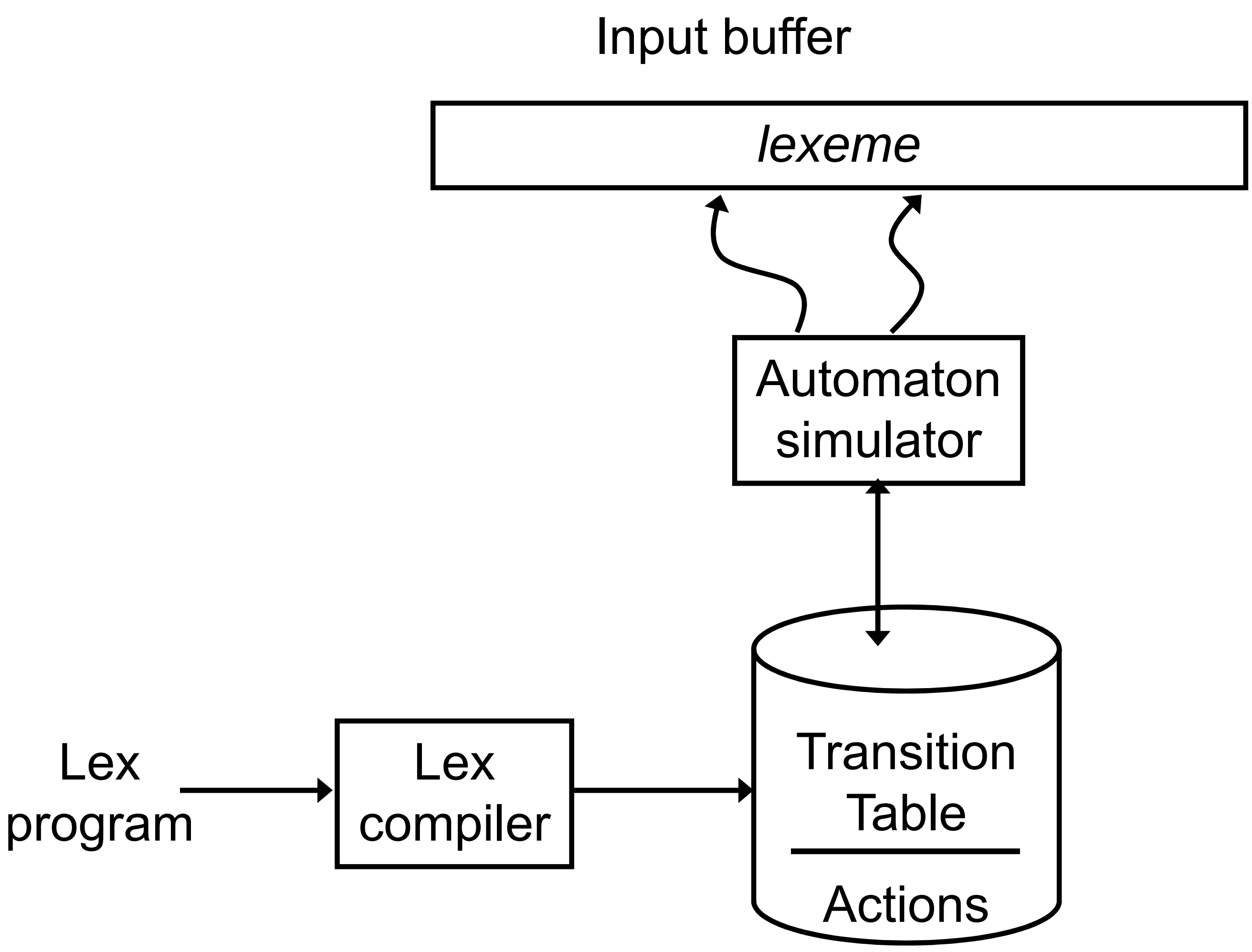
Lexical Analysis

Lecture 04

Structure of the Generated Analyzer



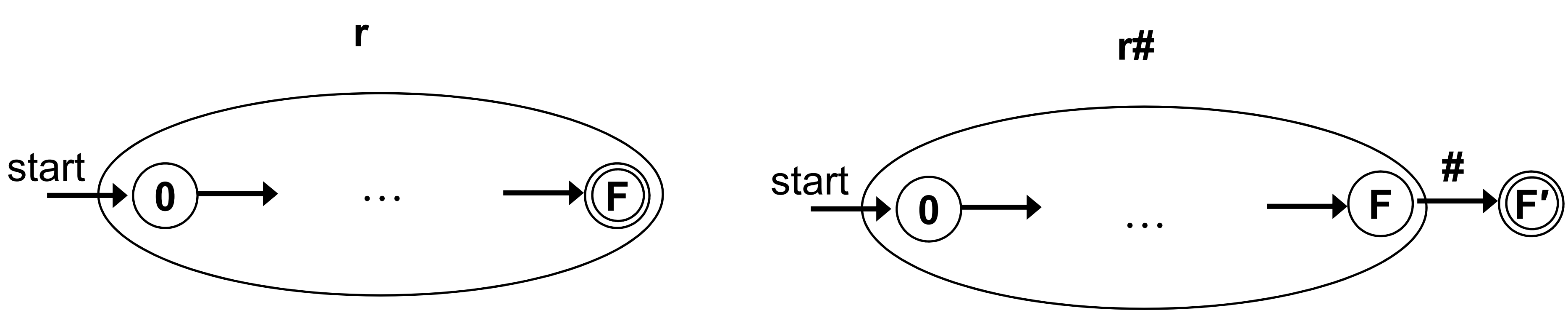
Implementing the Lookahead Operator

- Implementing r1/r2 : match r1 when followed by r2
- • e.g. *a*b+/a*c* accepts a string *bac* but not *abd*

- Reading Assignment
  - Implementing the Lookahead Operator

Regular Expression to DFA

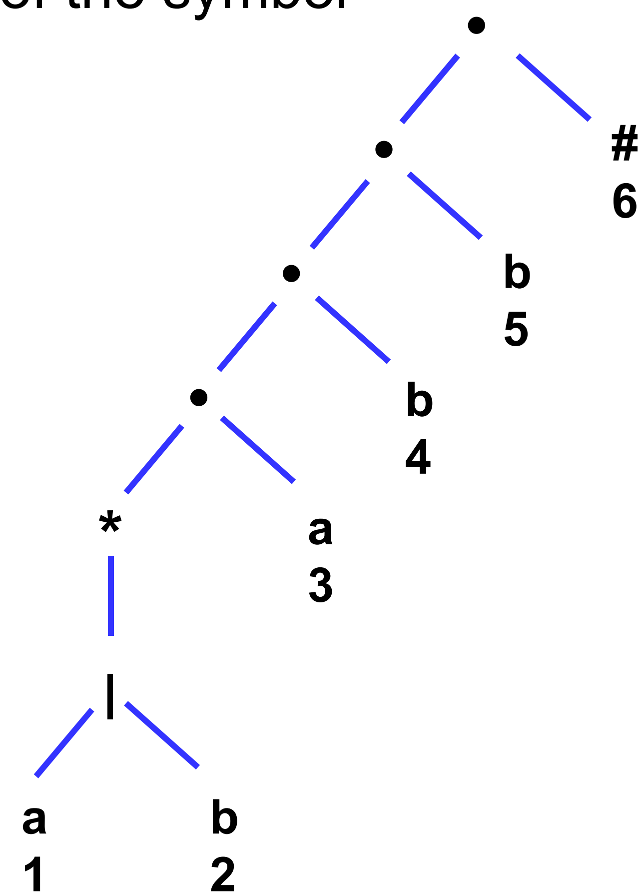
- Important States of NFA
  - If it has a non-ε out-transition
  - *move(s,a)* is non-empty if s is important
  - Accepting states are not important states
    - Adding a unique marker # after the RE r (i.e. r#) we can make the accepting states important
    - Now a state with a transition on # will be accepting state



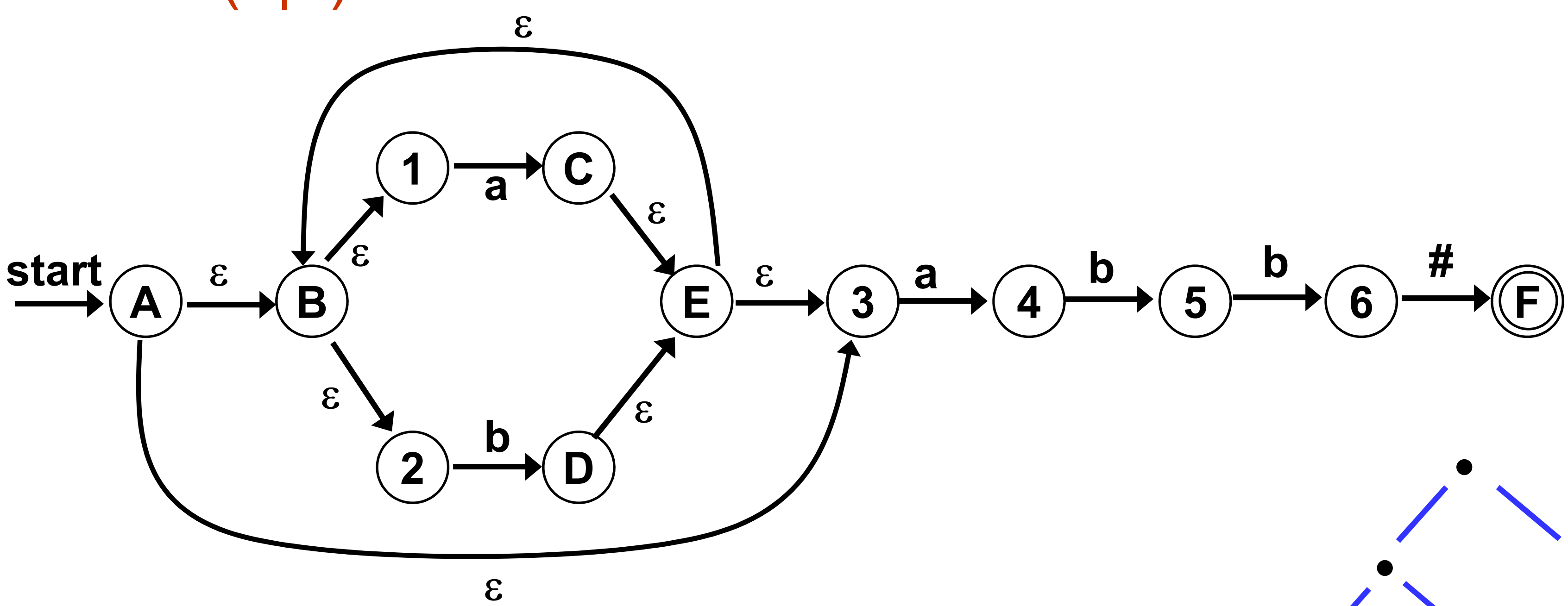
Syntax Tree

- Augmented RE (r#) can be represented by a syntax tree
  - Leaves contain: Alphabet symbols or ε
    - Each non-ε leaf is associated with a unique number-*position* of the leaf and *position* of the symbol
  - Internal nodes contain: Operators
    - *cat-node*, *or-node* or *star-node*

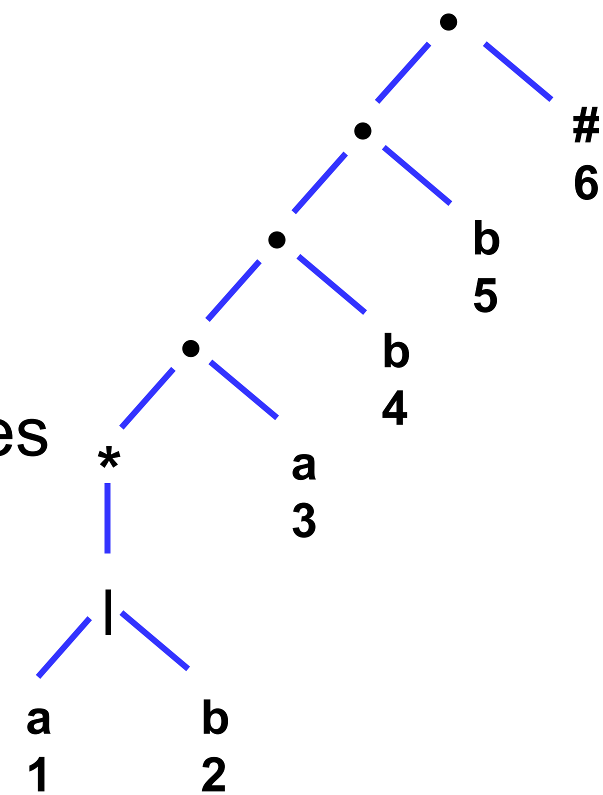
- Syntax tree for r# = (a|b)*abb#



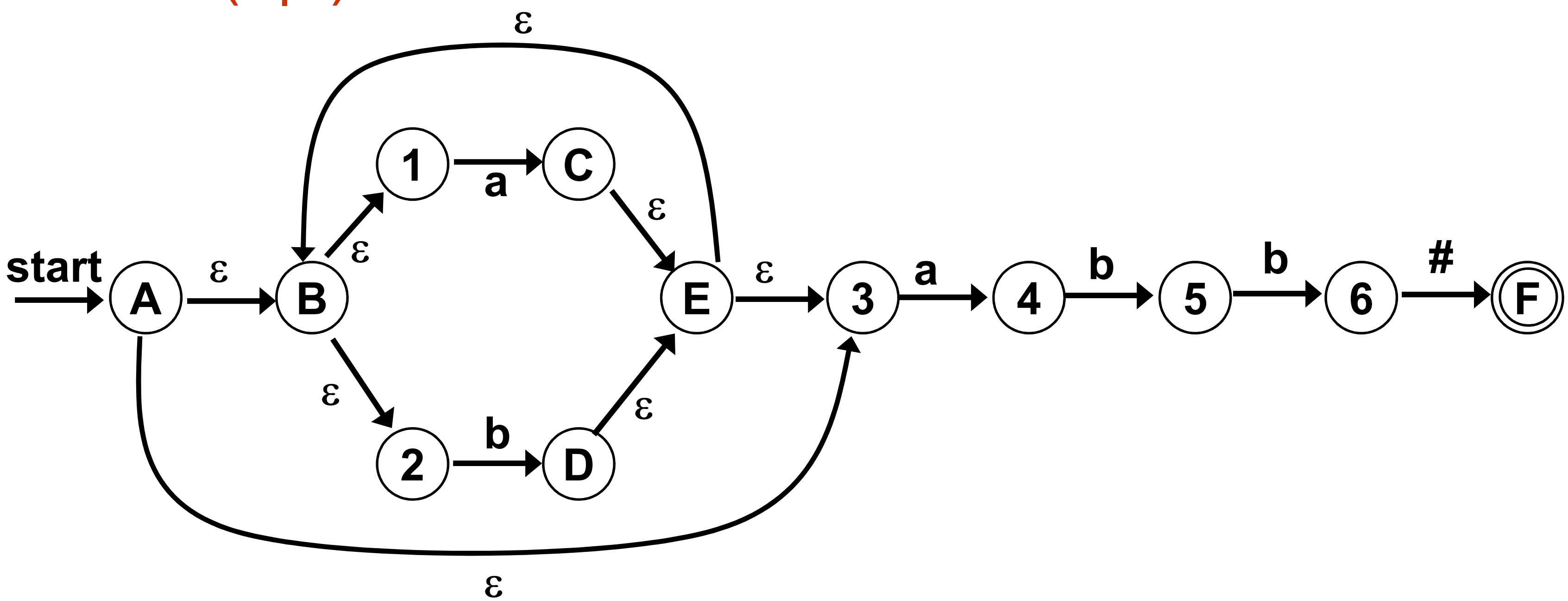
NFA for (a|b)*abb#

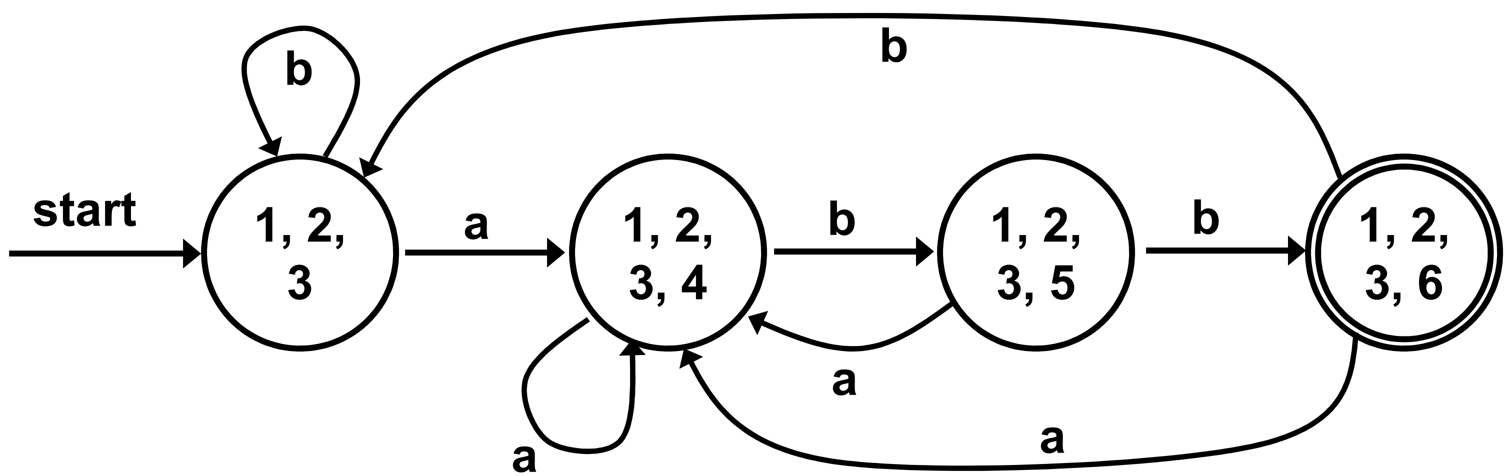


- Lettered states are non-important states
- Number states are important states
  – Numbers correspond to the number in syntax tree



DFA for (a|b)*abb#



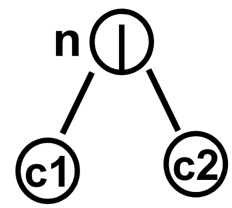


Terminology

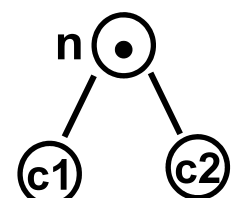
- Nullable:
  - Nodes that are the root of some sub-expression that generate empty string

- If n is a leaf labeled by ε then
  - **nullable (n) = true**

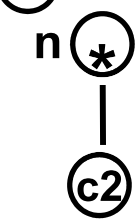
- If n is a leaf labeled with position *i*
  - **nullable (n) = false**



- If n is an or-node (|) with children c1 and c2
  - **nullable (n) = nullable(c1) or nullable (c2)**



- If n is an cat-node (•) with children c1 and c2
  - **nullable (n) = nullable(c1) and nullable (c2)**



- If n is an star-node (*) with children c1
  - **nullable (n) = true**

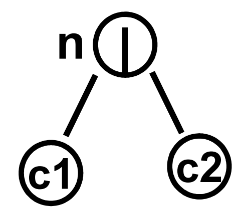
Terminology

- Firstpos(n):
  - Set of positions that can match the first symbol of a string generated by the sub-expression rooted at n

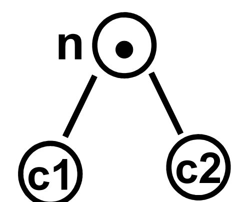
- If n is a leaf labeled by ε then
  - **firstpos (n) = $\varnothing$**

- If n is a leaf labeled with position i
  - **firstpos (n) = {i}**

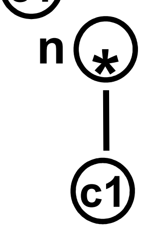
- If n is an or-node (|) with children c1 and c2
  - **firstpos (n) = firstpos(c1) $\cup$ firstpos (c2)**



- If n is a cat-node ($\bullet$) with children c1 and c2
  - **firstpos(n) = If nullable (c1) then firstpos(c1) $\cup$ firstpos (c2)**
    else firstpos(c1)

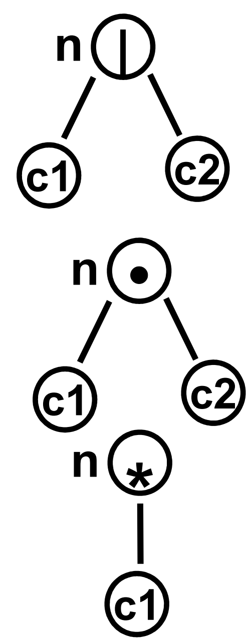


- If n is an star-node (*) with children c1
  - **firstpos (n) = firstpos(c1)**

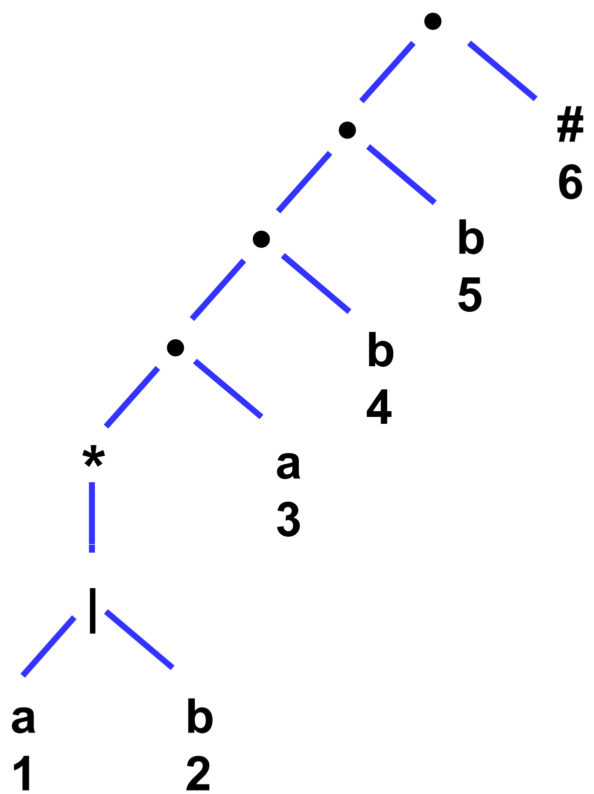


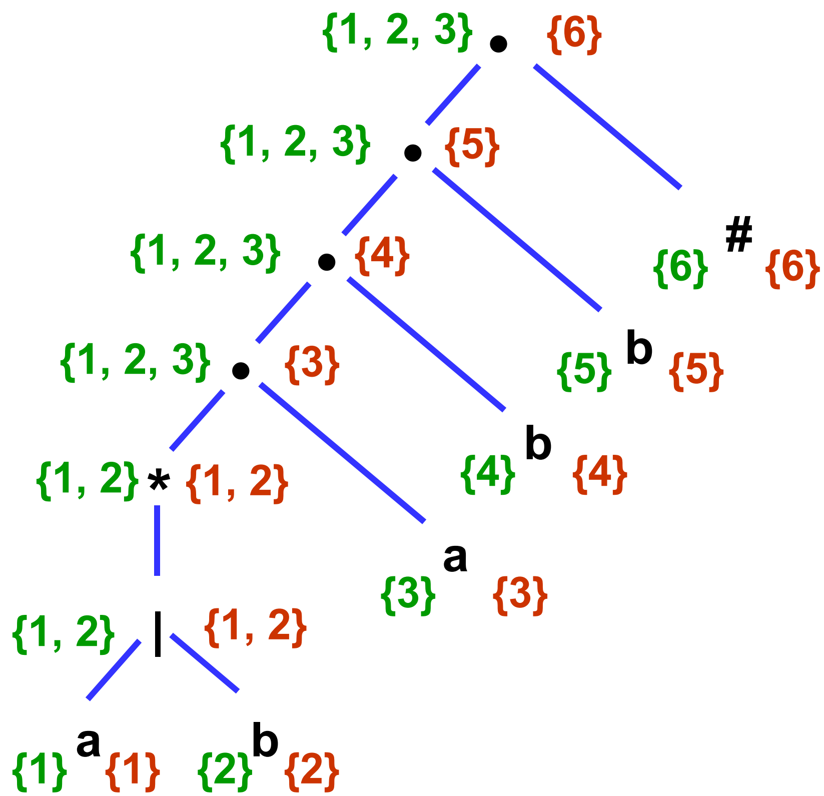
Terminology

- Lastpos(n):
  - Set of positions that can match the last symbol of a string generated by the sub-expression rooted at n
- If n is a leaf labeled by ε then
  - **lastpos (n) = ∅**
- If n is a leaf labeled with position *i*
  - **lastpos (n) = {i}**
- If n is an or-node (|) with children c1 and c2
  - **lastpos (n) = lastpos(c1) ∪ lastpos (c2)**
- If n is an cat-node (•) with children c1 and c2
  - **lastpos(n) = If nullable (c2) then lastpos(c1) ∪ lastpos (c2)**
  - **else lastpos(c2)**
- If n is an star-node (*) with children c1
  - **lastpos (n) = lastpos(c1)**



firstpos and *lastpos* example





Terminology

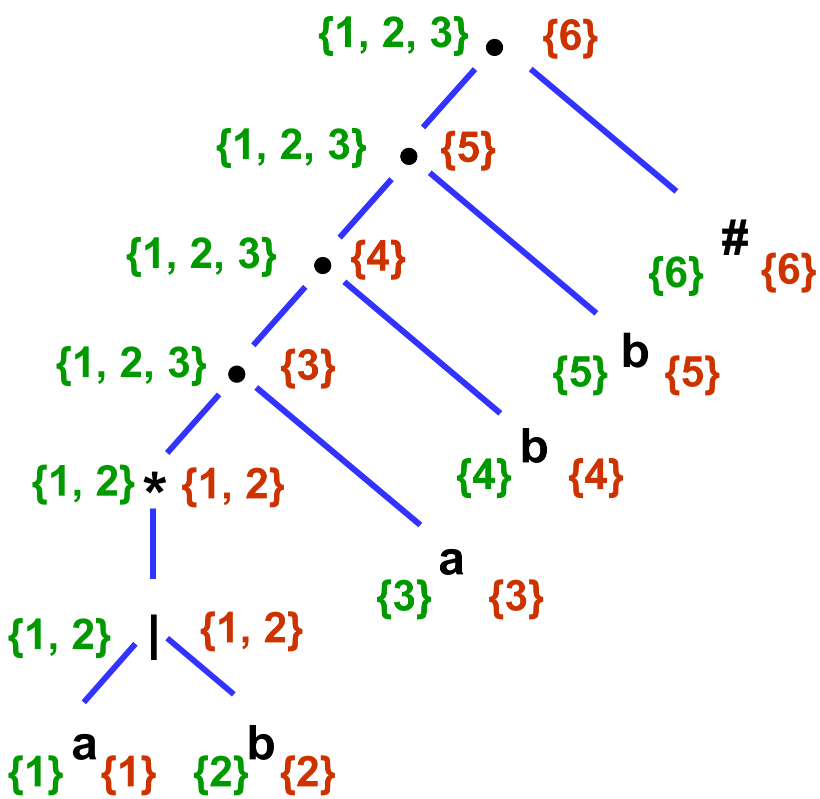
- Followpos(i):
  - Tells what positions can follow position i in the syntax tree

- **Rule 1:**
  If $\boldsymbol{n}$ is a cat-node with left child $\boldsymbol{c1}$ and right child $\boldsymbol{c2}$ and $\boldsymbol{i}$ is a position in lastpos ($\boldsymbol{c1}$), then all positions in firstpos($\boldsymbol{c2}$) are in followpos(i)

- **Rule 2:**
  If $\boldsymbol{n}$ is a star node, and $\boldsymbol{i}$ is a position in lastpos($\boldsymbol{n}$), then all positions in firstpos($\boldsymbol{n}$) are in followpos($\boldsymbol{i}$)

- After computing firstpos and lastpos for each node follow pos of each position can be computed by making depth-first traversal of the syntax tree

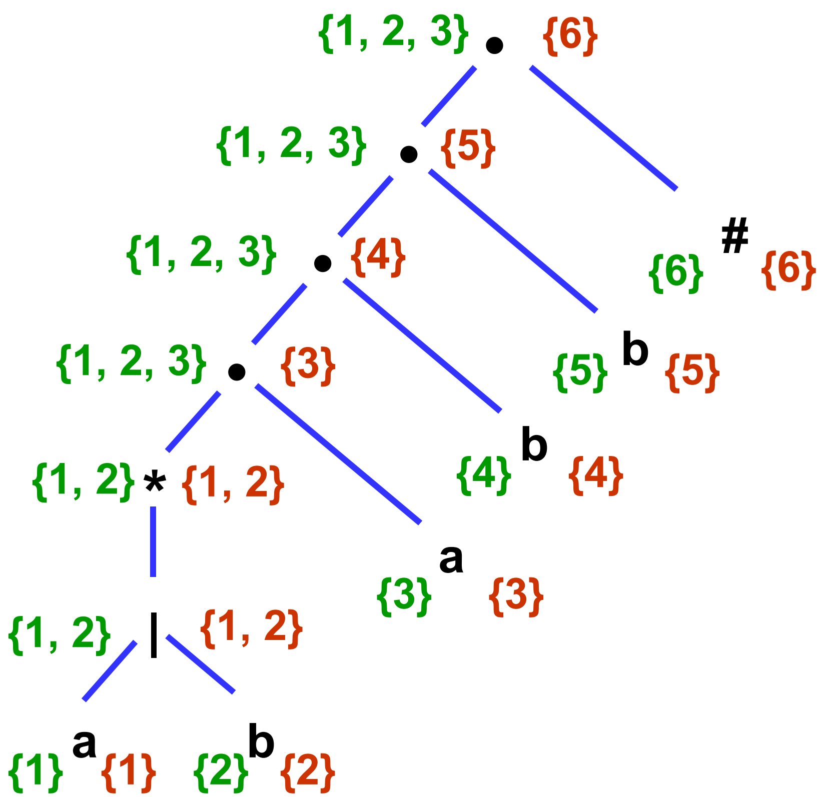
followpos example



Node	followpos
1	{1,2,
2	{1,2,
3	{
4	{
5	{
6	{

- At star-node:
  - *lastpos*(*) = {1,2} and *firstpos*(*)={1,2}
  - According to Rule 2:
    - followpos{1} = {1,2}
    - followpos{2} = {1,2}

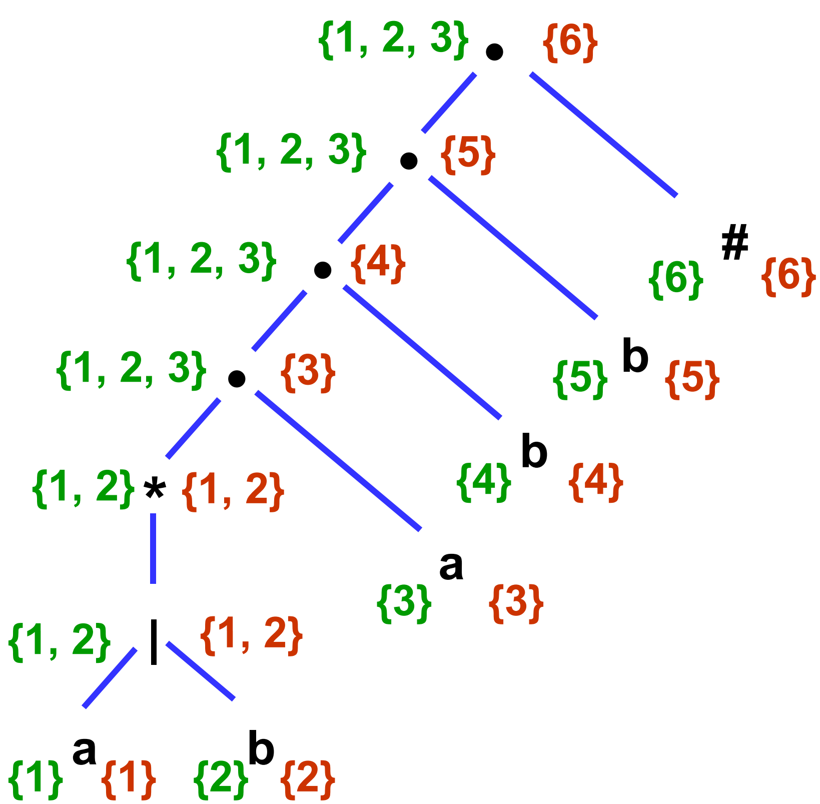
followpos example



Node	Followpos
1	{1,2,3
2	{1,2,3
3	{
4	{
5	{
6	{

- At cat-node above the star-node, '*' is left child and 'a' is right child
  - *lastpos*(*) = {1,2} and *firstpos(a)*={3}
  - According to Rule 1:
    - » followpos{1} = {3}
    - » followpos{2} = {3}

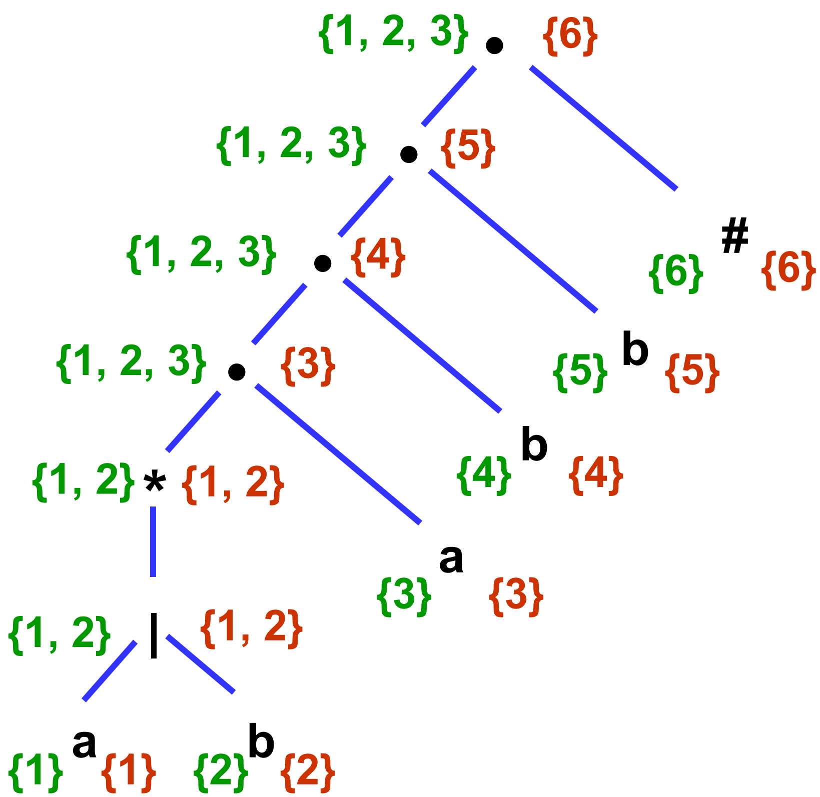
followpos example



Node	Followpos
1	{1,2,3
2	{1,2,3
3	{4
4	{5
5	{6
6	{

- At next cat-node '•' is left child and 'b' is right child
  - *lastpos*(•) = {3} and *firstpos(b)* = {4}
  - According to Rule 1:
    - » followpos{3} = {4}
- Similarly, followpos{4}={5} and followpos{5}={6}

followpos example



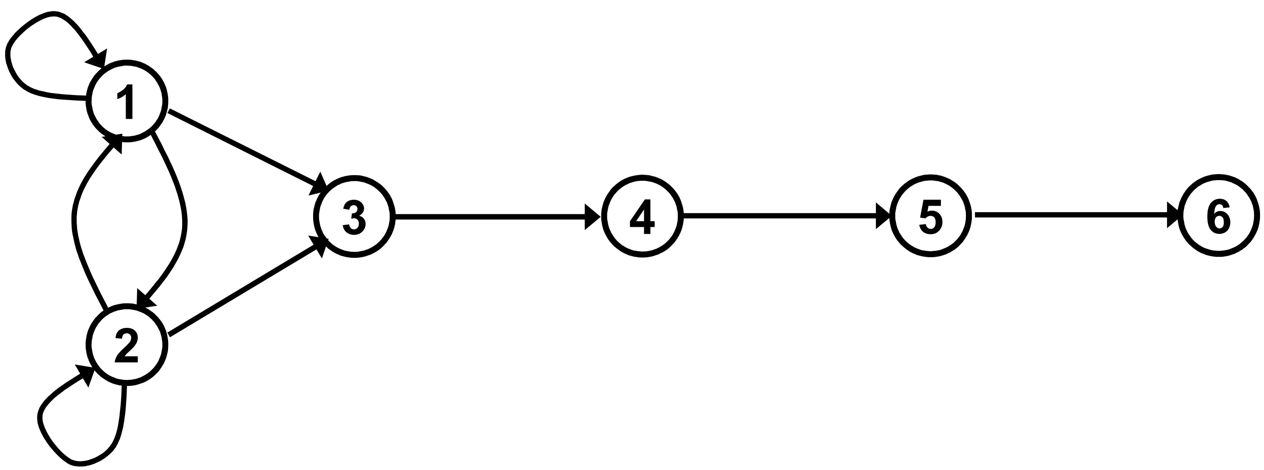
Node	followpos
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	-

followpos graph

- A node for each position
- Edge from node i to node j if j ∈ followpos{i}

Node	followpos
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	-

- *followpos* graph becomes equivalent NFA without ε-transition if
  - All positions in *firstpos* of root become start state
  - Label edge {i,j} by the symbol at position j
  - Position associated with # only accepting state



Construction of DFA from RE

- Input: A regular expression r
- Output: A DFA D that recognizes L(r)

- Method:

1. Construct syntax tree ST for augmented RE r#
2. Construct the functions nullable, firstpos, lastpos and followpos for ST
3. Construct Dstates: set of states of D
   Dtrans: transition table for D

Construction of DFA from RE

- Algorithm

Initially, the only unmarked state in **Dstates** is *firstpos(***root***)*

while there is an unmarked state **T** in **Dstates** do begin

Mark T;

For each input symbol **a** do begin

Let **U** be the set of positions that are in followpos(**p**) for some position **p** in **T** such that the symbol at position **p** is **a**

If **U** is not empty and is not in **Dstates** then

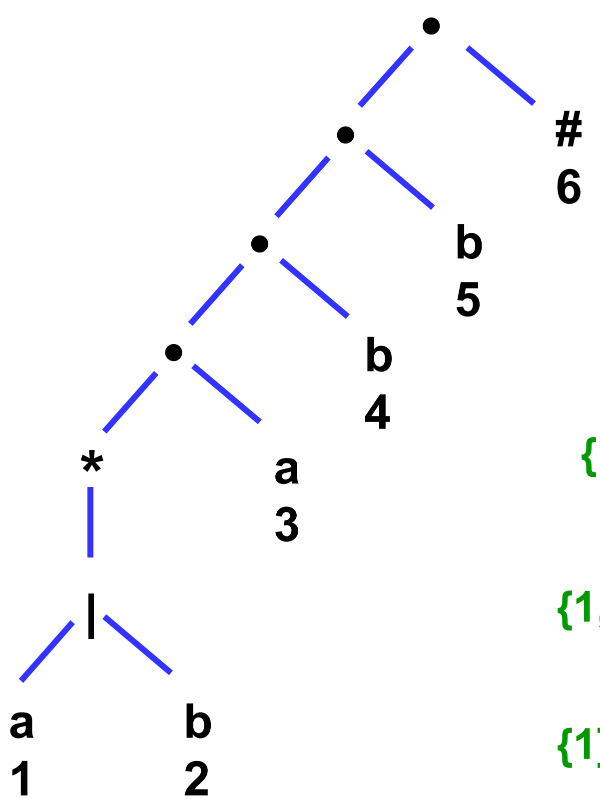
Add **U** as an unmarked states to **Dstates**

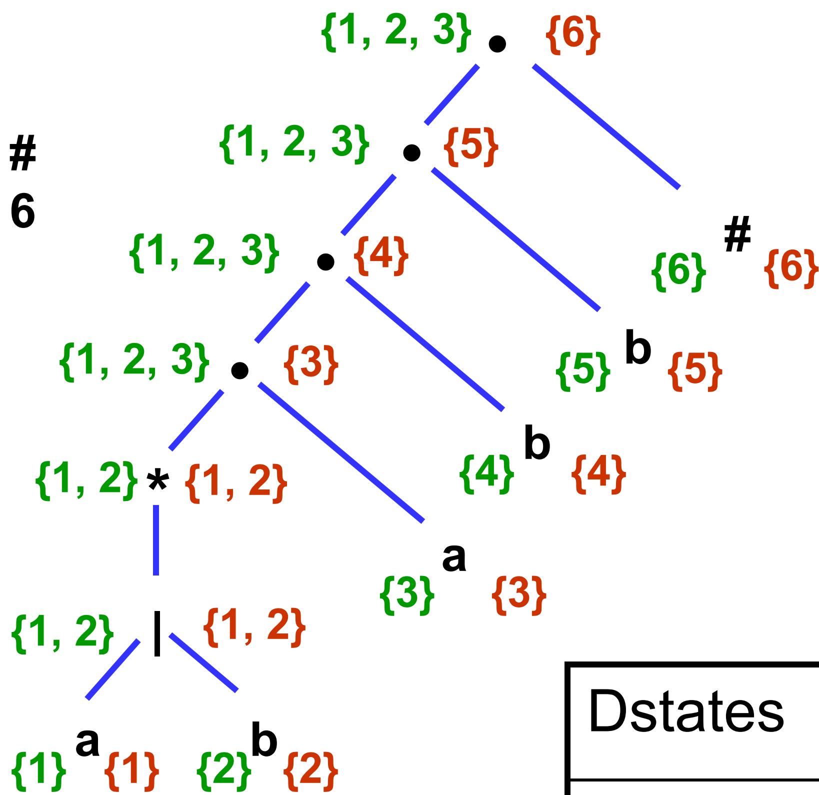
Dtrans[T,a]=U

End

end

DFA for (a|b)*abb#





Node	followpos
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	-

firstpos{root} = {1,2,3} ≡ A (unmarked)

For the input symbol **a,** positions are 1, 3

∴ followpos(1) ∪ followpos{3}

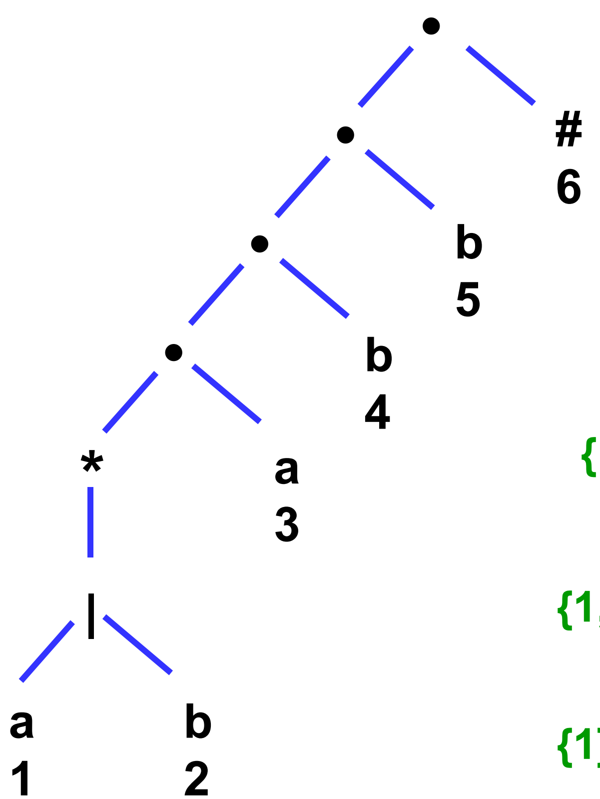
={1,2,3,4} ≡ B

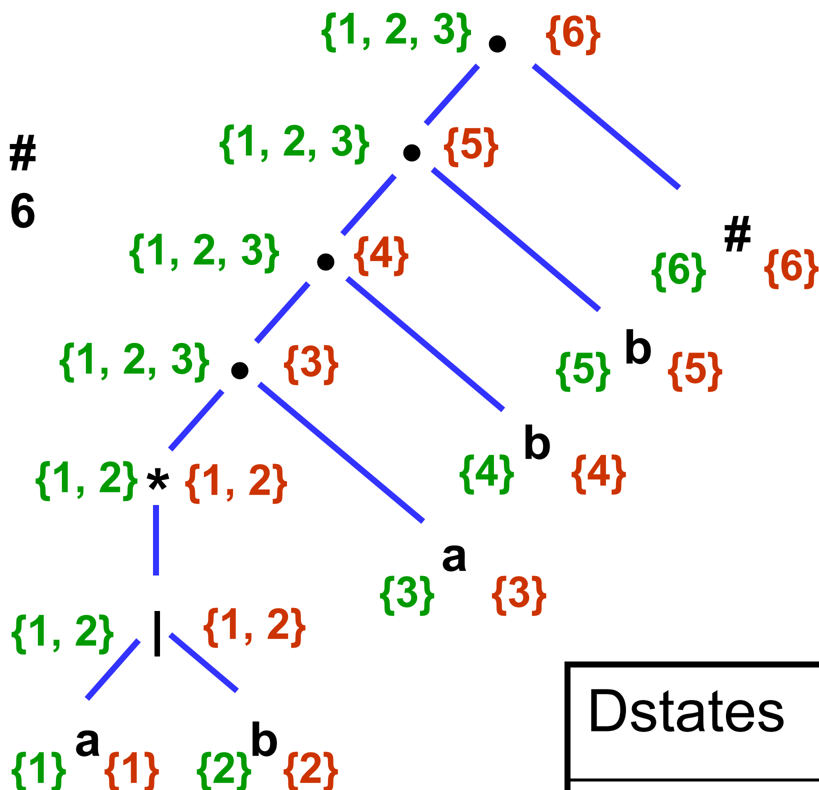
For the input symbol **b,** positions are 2

∴ followpos(2)= {1,2,3,} ≡ A

Dstates	a	b
{1,2,3} ≡ A	B	A
{1,2,3,4} ≡ B		

DFA for (a|b)*abb#





Node	followpos
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	-

{1,2,3,4} ≡ B (unmarked)

For the input symbol **a,** positions are 1, 3

∴ followpos(1) ∪ followpos{3}

=\{1,2,3,4\} ≡ B

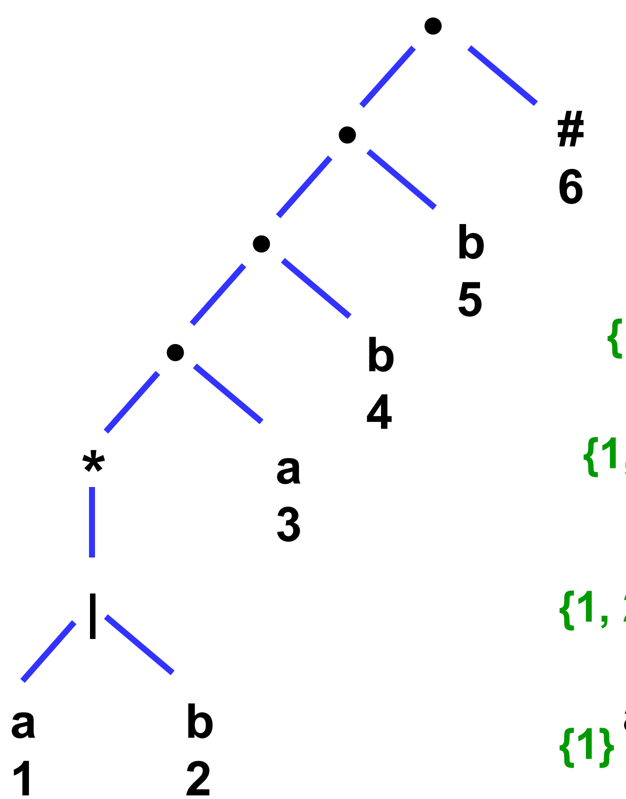
For the input symbol **b,** positions are 2, 4

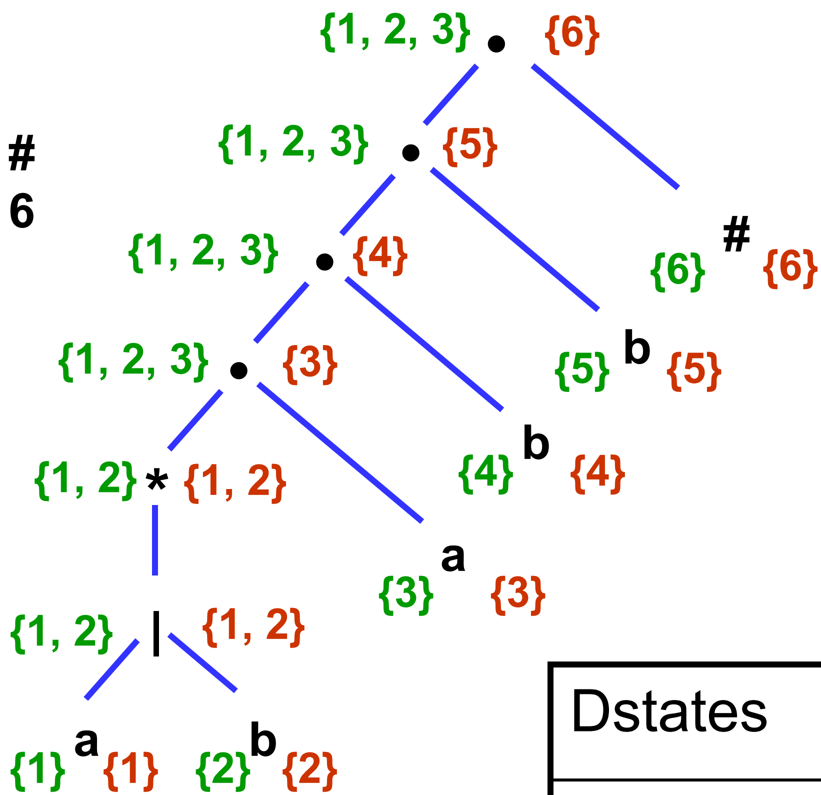
∴ followpos(2) ∪ followpos{4}

= {1,2,3,5} ≡ C

Dstates	a	b
{1,2,3} ≡ A	B	A
{1,2,3,4} ≡ B	B	C
{1,2,3,5} ≡ C		

DFA for (a|b)*abb#





Node	followpos
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	-

{1,2,3,5} ≡ C (unmarked)

For the input symbol **a,** positions are 1, 3

$\therefore$ followpos(1) $\cup$ followpos{3}

=\{1,2,3,4\} ≡ B

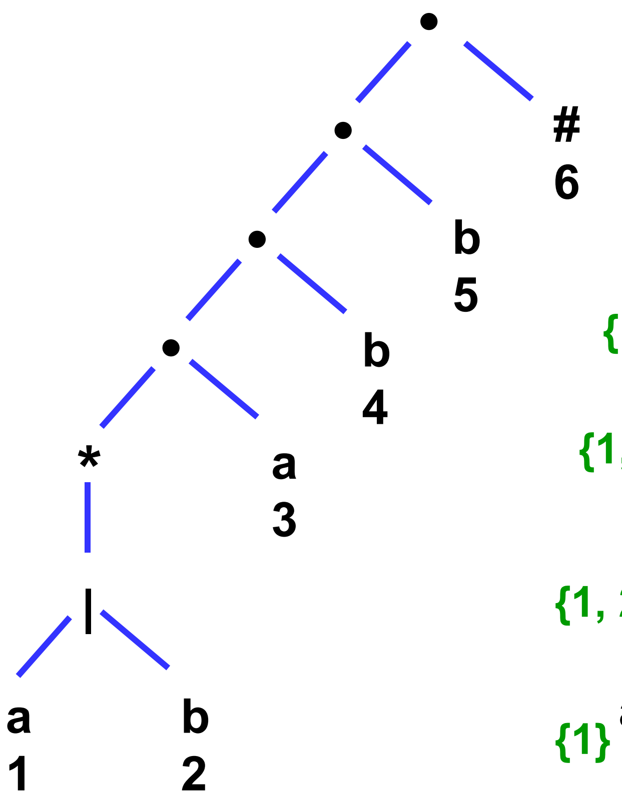
For the input symbol **b,** positions are 2, 5

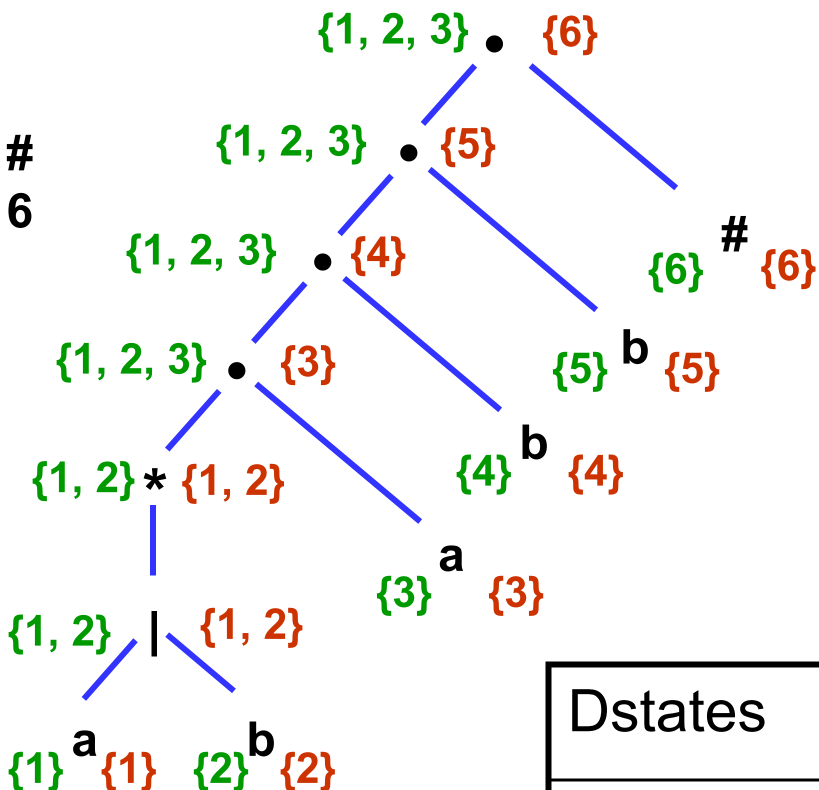
$\therefore$ followpos(2) $\cup$ followpos{5}

= {1,2,3,6} ≡ D

Dstates	a	b
{1,2,3} ≡ A	B	A
{1,2,3,4} ≡ B	B	C
{1,2,3,5} ≡ C	B	D
{1,2,3,6} ≡ D		

DFA for (a|b)*abb#





Node	followpos
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	-

{1,2,3,6} ≡ D (unmarked)

For the input symbol **a,** positions are 1, 3

∴ followpos(1) ∪ followpos{3}

={1,2,3,4} ≡ B

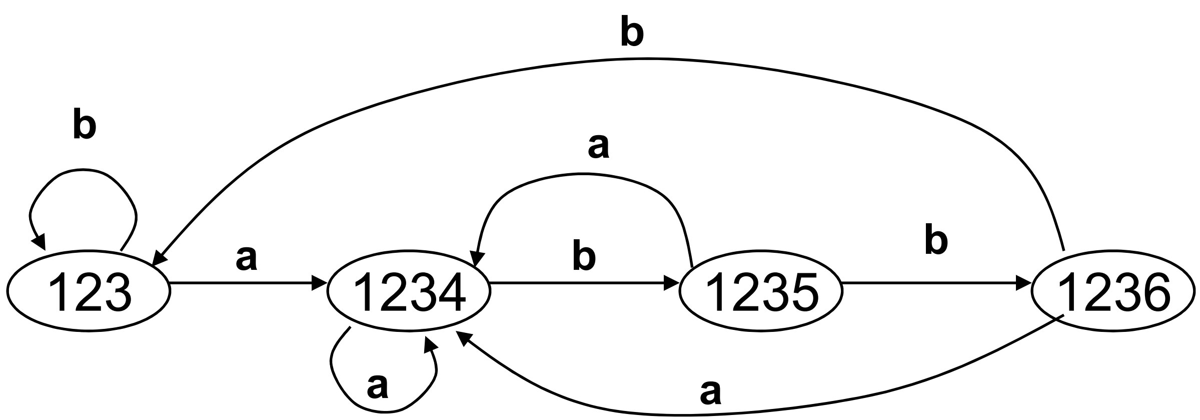
For the input symbol **b,** positions are 2

∴ followpos(2)

= {1,2,3} ≡ A

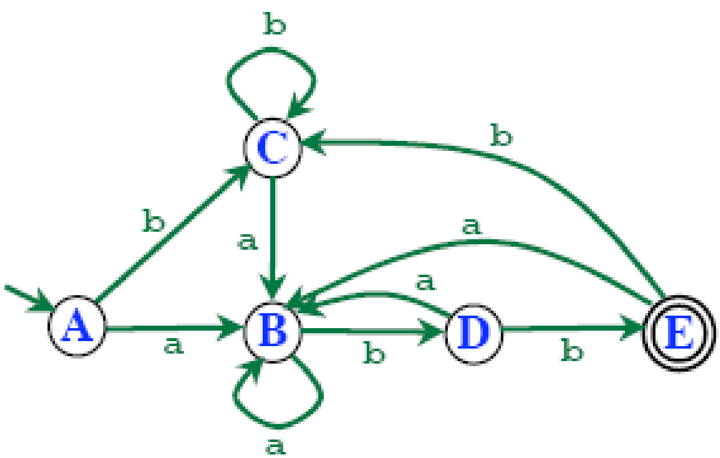
Dstates	a	b
{1,2,3} ≡ A	B	A
{1,2,3,4} ≡ B	B	C
{1,2,3,5} ≡ C	B	D
{1,2,3,6} ≡ D	B	A

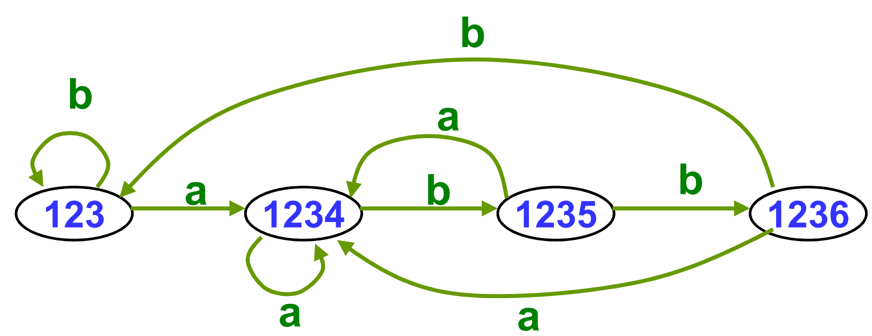
DFA for (a|b)*abb#



DFA State Minimization

- More than one DFA can recognize the same language

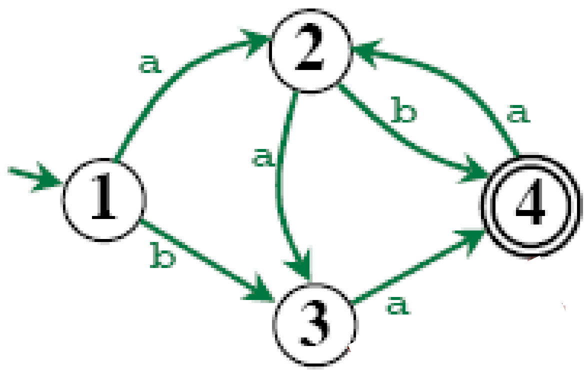


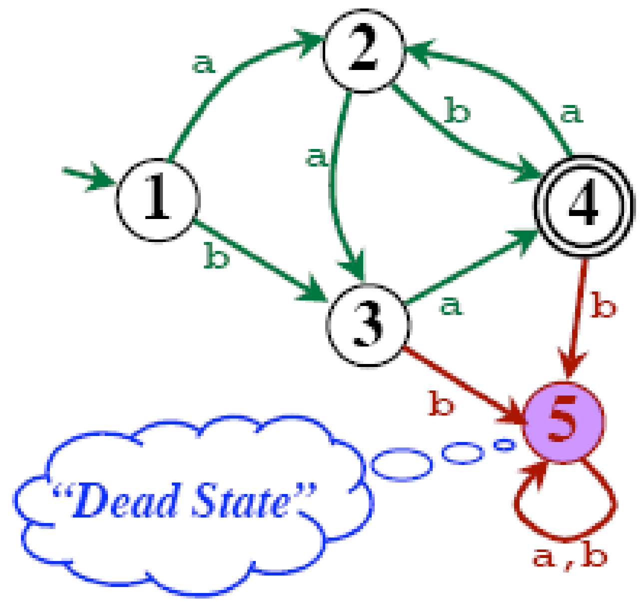


- Two automata are the same up to state names
  - If one can be transformed into the other by changing the names only

Dead State

- A state to which every missing transition is forwarded as well as it has transition to itself for each input symbol

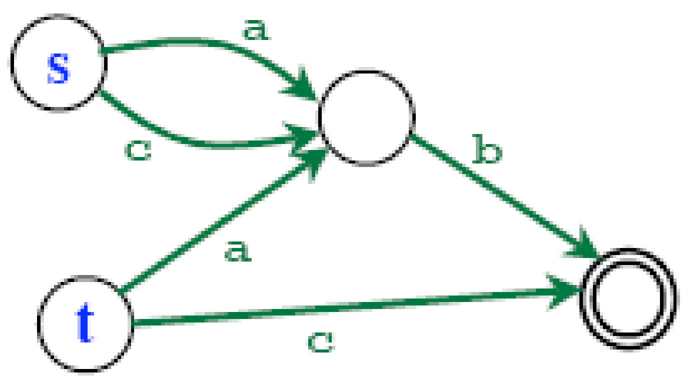




Distinguishable states

- State s is "distinguished" from state t by some string w iff:
  - starting at s, given characters w, the DFA ends up accepting,
  - ... but starting at t, the DFA does not accept.

Example:



"**ab**" does not distinguish s and t.

But "**c**" distinguishes s and t.

Partitioning a Set

- A partitioning of a set...

  ...breaks the set into non-overlapping subsets.

  (The partition breaks the set into "groups")

- ***Example:***

  S = {A, B, C, D, E, F, G}

  π = {(A B) (C D E F) (G) }

  π_2 = {(A) (B C) (D E F G) }

- We can "refine" a partition...

  π_i = { (A B C) (D E) (F G) }

  π_{i+1} = { (A C) (B) (D) (E) (F G) }

Hopcroft's Algorithm

Consider the set of states.

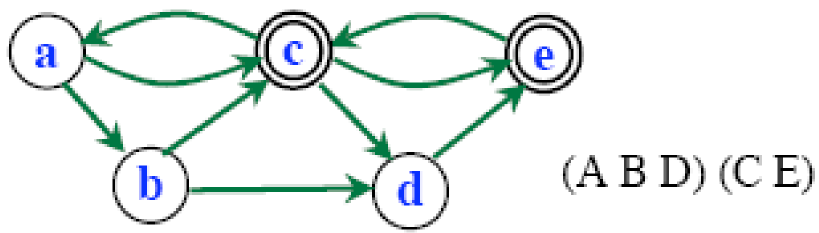
Partition it...

- **Final States**
- **All Other States**

Repeatedly "refine" the partioning.

Two states will be placed in different groups

... If they can be "distinguished"



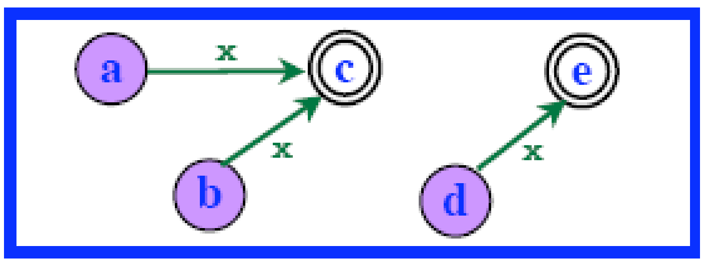
Repeat until no group contains states that can be distinguished.

Each group in the partitioning becomes one state in a newly constructed DFA

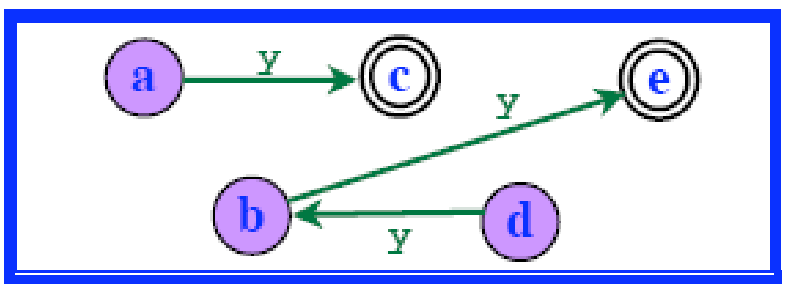
DFA_{MIN} = The minimal DFA

How to Refine a Partitioning?

- $\pi_i = \{ \underbrace{(\ A\ B\ D\ )}_{P_1} \underbrace{(\ C\ E\ )}_{P_2} \}$
- Consider one group... (A B D)
- Look at output edges on some symbol (e.g., "**x**")



- On "**x**", all states in P_1 go to states belonging to the same group.



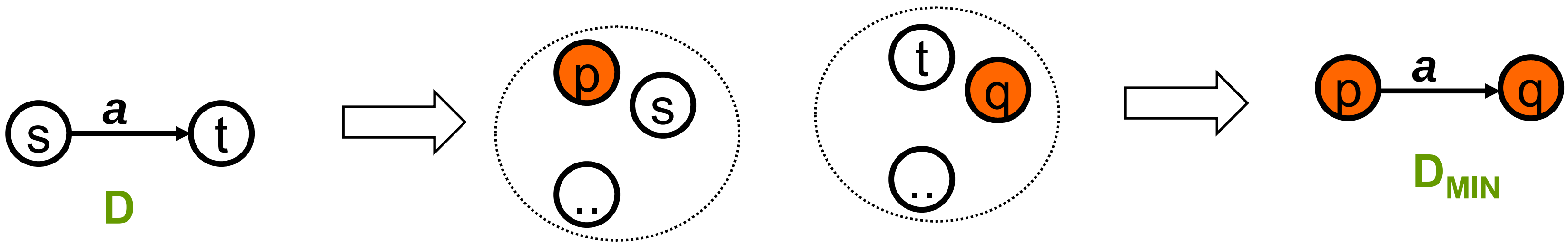
Now consider another symbol (e.g., "**y**")
D is distinguished from A and B!
So refine the partition!
$\pi_{i+1} = \{ \underbrace{(\ A\ B\ )}_{P_3} \underbrace{(\ D\ )}_{P_4} \underbrace{(\ C\ E\ )}_{P_2} \}$

Hopcroft's Algorithm

1. Start with an initial partition π of D with two groups, F and S-F
2. Repeat
3. π_{new} = newPartition (π)
4. IF $\pi_{new} = \pi$, Set $\pi_{final} = \pi$ and Break
5. Else Set $\pi = \pi_{new}$
6. Choose one state in each group as the representative for the group.
    1. This representatives will be the states of the D_{MIN}
    2. The start state of D_{MIN} is the representative of the group containing the start state of D
    3. The accepting state of D_{MIN} is the representatives of those groups that contain an accepting state of D
    4. Transition Rule

Hopcroft's Algorithm

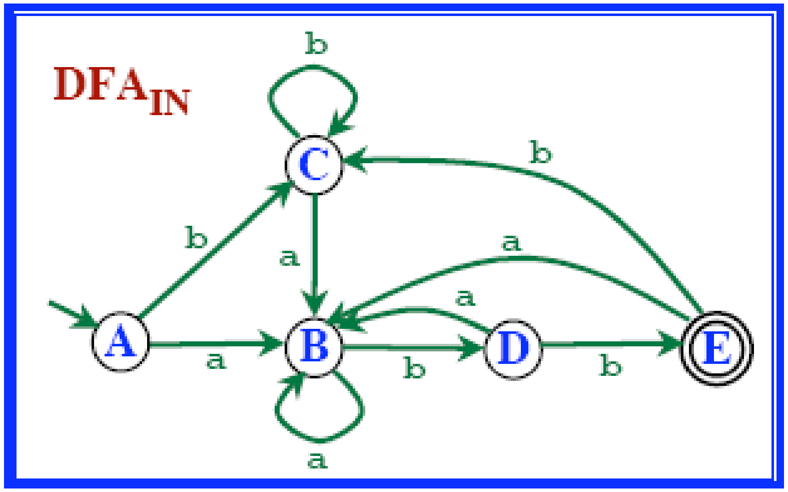
4. Transition Rule for D_{MIN}

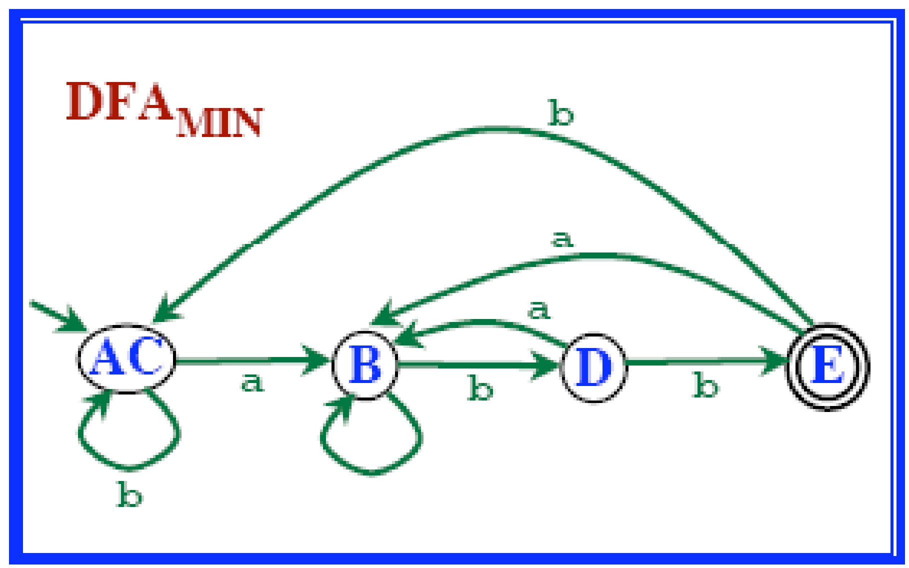


newPartition (π)

1. Set $\pi_{new} = \pi$
2. For (each group G of π)
3. partition G into subgroups such that two states s and t are in the same subgroup iff for all input symbols a, state s and t have transition on a to states in the same group of π
4. replace G in π by the set of all subgroups found

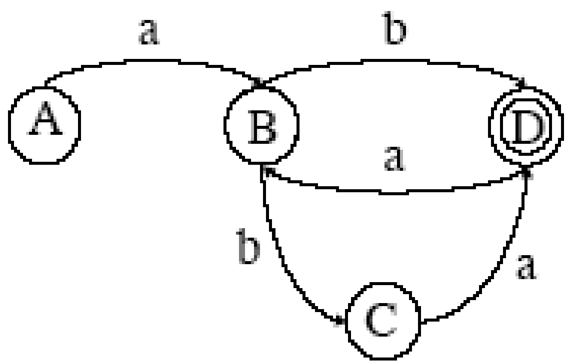
Example





NFA to Regular Expression

What is the RE for the following NFA



We can write

- A = a B
- B = b D | b C
- C = a D
- D = a B | ε

NFA to Regular Expression

Three steps in the algorithm (apply in any order):

- Substitution: for B = X pick every A = B | T and replace to get A = X | T
- Factoring: (R S) | (R T) = R (S | T) and (R T) | (S T) = (R | S) T
- Arden's Rule: For any set of strings S and T, the equation X = (S X) | T has X = (S*) T as a solution.

NFA to Regular Expression

1. Starting Expressions
   - A = a B
   - B = b D | b C
   - D = a B | ε
   - C = a D
2. Substitute:
   - A = a B
   - B = b D | b a D
   - D = a B | ε
3. Factor:
   - A = a B
   - B = (b | b a) D
   - D = a B | ε
4. Substitute:
   - A = a (b | b a) D
   - D = a (b | b a) D | ε

NFA to Regular Expression

4.

- $A = a\,(b \mid b\,a)\,D$
- $D = a\,(b \mid b\,a)\,D \mid \varepsilon$

5. Factor:

- $A = (a\,b \mid a\,b\,a)\,D$
- $D = (a\,b \mid a\,b\,a)\,D \mid \varepsilon$

6. Arden:

- $A = (a\,b \mid a\,b\,a)\,D$
- $D = (a\,b \mid a\,b\,a)^*\,\varepsilon$

7. Remove epsilon:

- $A = (a\,b \mid a\,b\,a)\,D$
- $D = (a\,b \mid a\,b\,a)^*$

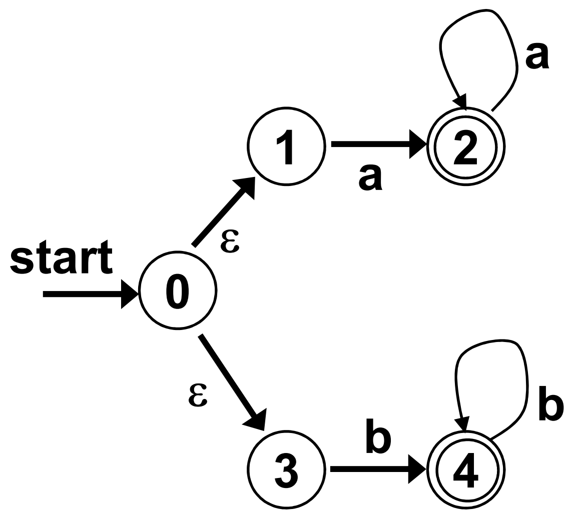
8. Substitute:

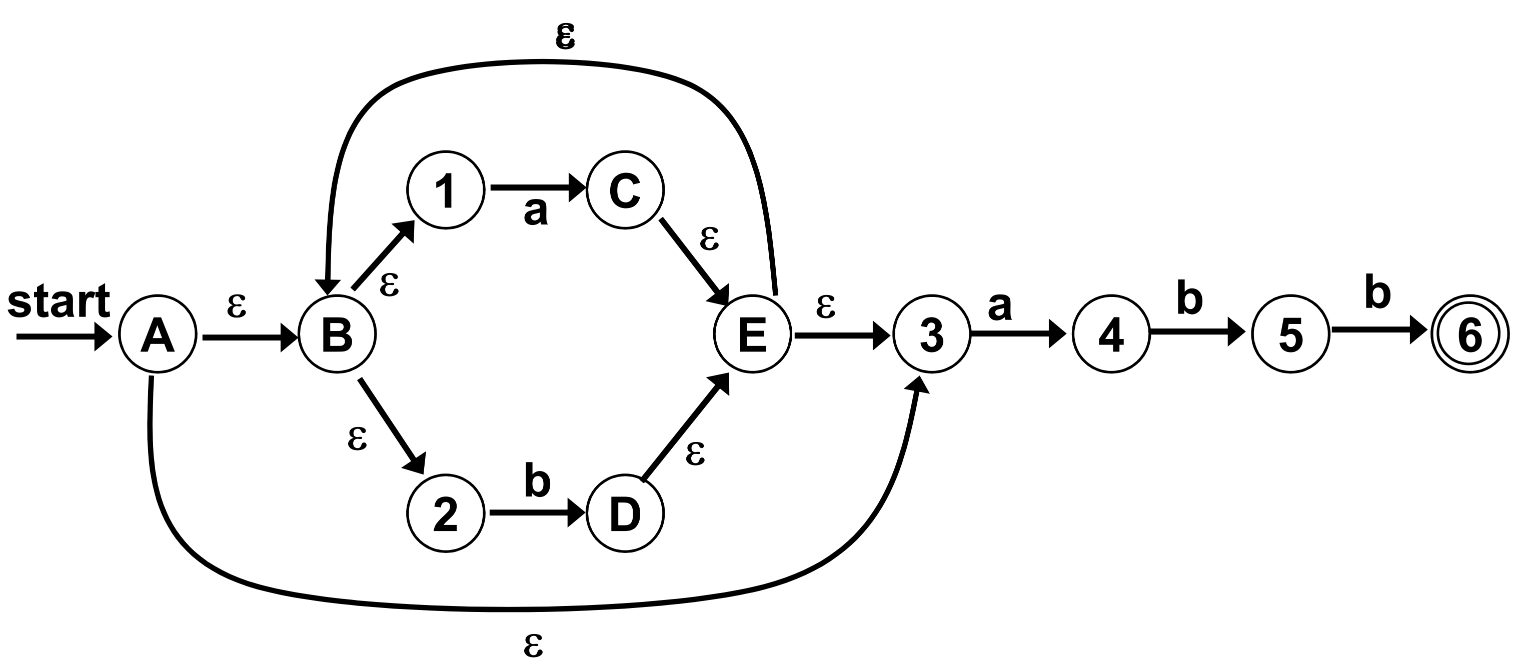
- $A = (a\,b \mid a\,b\,a)$
- $(a\,b \mid a\,b\,a)^*$

9. Simplify:

- $A = (a\,b \mid a\,b\,a)+$

NFA to Regular Expression





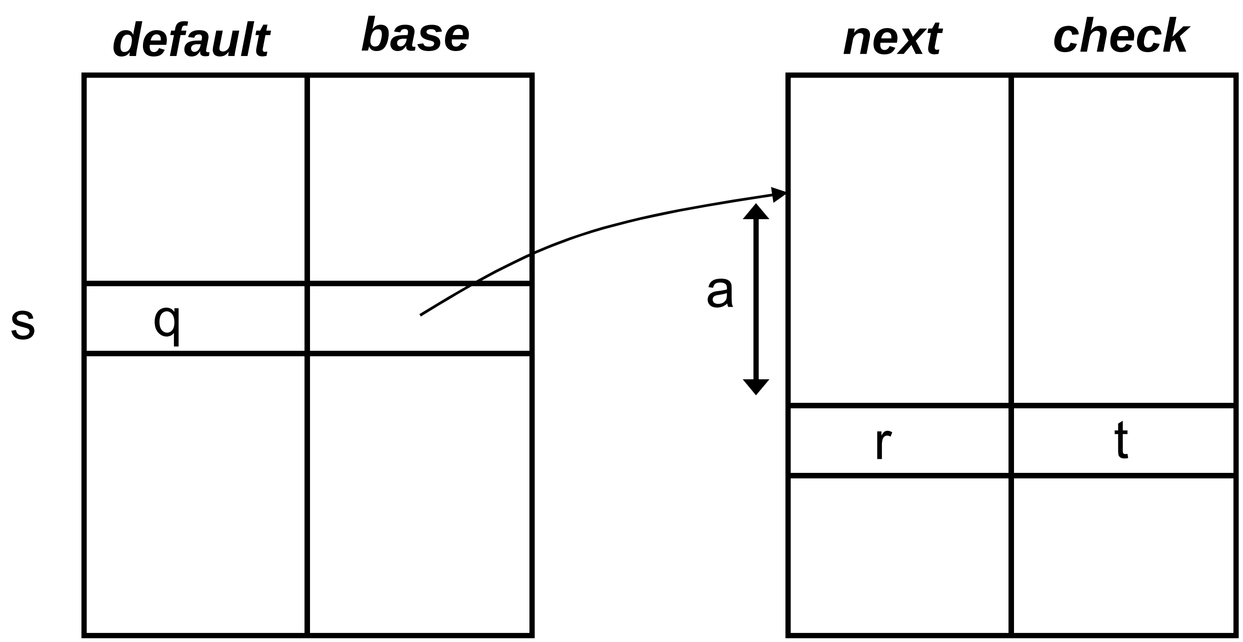
Trading Time for Space in DFA Simulation

- 2D array storing the transition table
- Adjacency list, more space efficient but slower

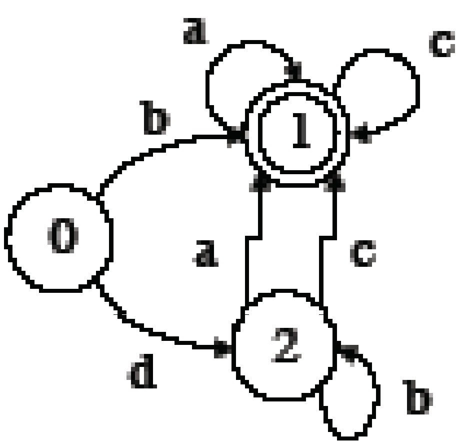
- Merge two ideas: array structures used for sparse tables like DFA transition tables

Trading Time for Space in DFA Simulation

- The required Data Structure is four arrays
- base: used to determine the base location of entries for a state
- next: used to give us the next state
- check: used to tell whether the entry is valid or not
- default: used to determine an alternative base location



Trading Time for Space in DFA Simulation



	a	b	c	d
0	-	1	-	2
1	1	-	1	-
2	1	2	1	-

Trading Time for Space in DFA Simulation

	a	b	c	d
0	-	1	-	2
1	1	-	1	-
2	1	2	1	-

base

0	2
1	4
2	0

			-	1	-	2		
					1	-	1	-
	1	2	1	-				
next	1	2	1	1	1	2	1	-
	0	1	2	3	4	5	6	7
check	2	2	2	0	1	0	1	-

nextstate(s, x) :
 L := base[s] + x
 return next[L] if check[L] eq s

Trading Time for Space in DFA Simulation

	a	b	c	d
0	-	1	-	2
1	1	-	1	-
2	1	2	1	-

		-	1	-	2		
				1	-	1	-
	-	2	-	-			
next	-	2	1	1	2	1	-
	0	1	2	3	4	5	6
check	-	2	0	1	0	1	-

base

		default
0	1	-
1	3	-
2	0	1

nextstate(s, x) :

L := base[s] + x

return next[L] **if** check[L] **eq** s

else return *nextstate*(default[s], x)