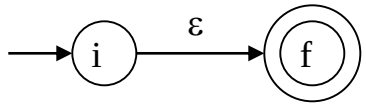
CS 2201: Regular Expression

Converting A Regular Expression into A NFA (Thomson's Construction)

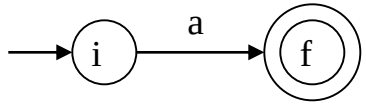
- One way to convert a regular expression into a NFA
- Many others do exist !
- Thomson's construction is simple and systematic
  - *It guarantees that the resulting NFA will have exactly one final state, and one start state*
- Construction starts from simplest parts (alphabet symbols)
  - To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA

Thomson's Construction (cont.)

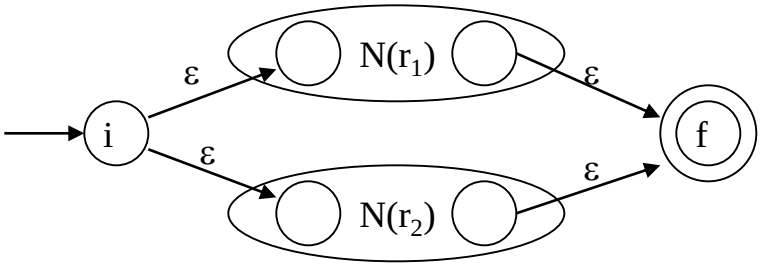
- Recognize an empty string ε



- Recognize a symbol a in the alphabet Σ



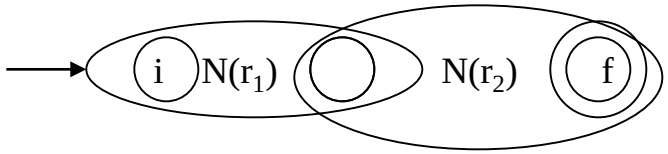
- If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2
  - For regular expression $r_1 \mid r_2$



NFA for $r_1 \mid r_2$

Thomson's Construction (cont.)

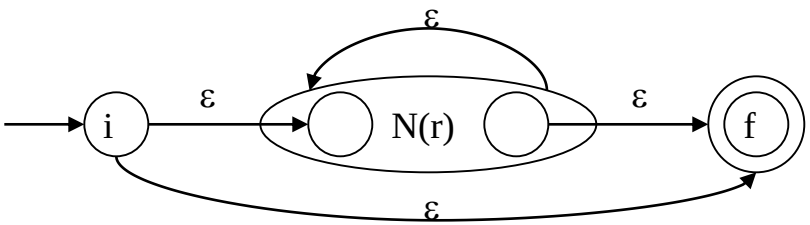
- For regular expression $r_1\ r_2$



Final state of $N(r_2)$ becomes final state of $N(r_1 r_2)$

NFA for $r_1\ r_2$

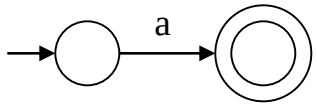
- For regular expression r^*



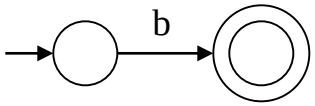
NFA for r^*

Thomson's Construction (Example - (a|b) * a)

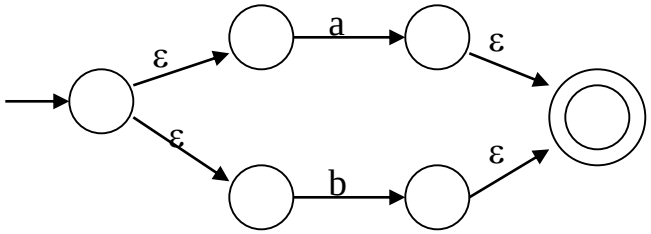
a:



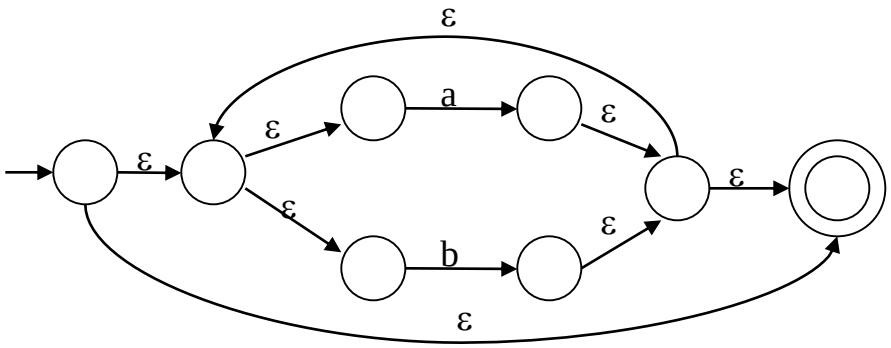
b:



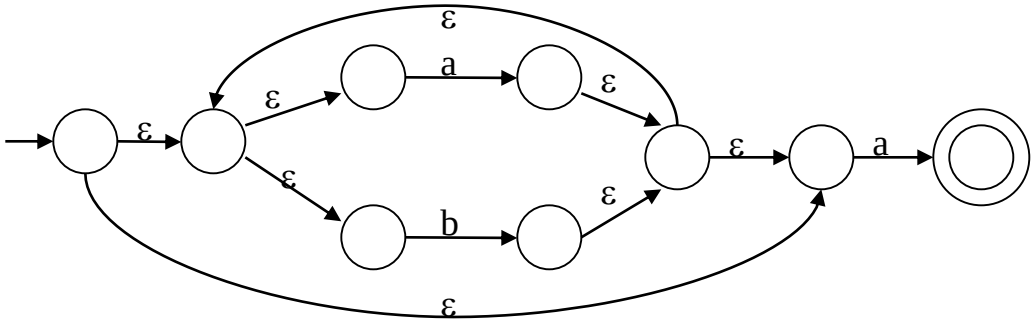
(a | b)



(a|b) *



(a|b) * a



Converting a NFA into a DFA (subset construction)

put ε-closure($\{s_0\}$) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S_1 in DS) do
  begin
    mark S_1
    for each input symbol a do
      begin
        $S_2 \leftarrow$ ε-closure(move(S_1,a))
        if (S_2 is not in DS) then
            add S_2 into DS as an unmarked state
        transfunc[S_1,a] $\leftarrow$ S_2
      end
  end

ε-closure($\{s_0\}$) is the set of all states, accessible from s_0 by ε-transition.

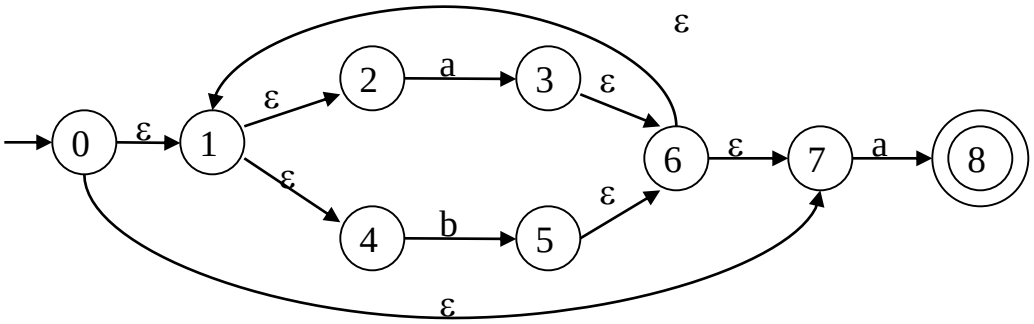
set of states to which there is a transition on a from a state s in S_1

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure($\{s_0\}$)

Computing ε-closure(T)

```
 push all states of T onto stack;
initialize ε-closure(T) to T;
While (stack is not empty) {
      Pop t, the top element, off the stack;
      For (each state u with an edge from t to u labeled ε)
           if (u is not in ε-closure(T) ){
                add u to ε-closure(T)
                push u onto stack;
           }
}
```

Converting a NFA into a DFA (Example)



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into DS as an unmarked state

⇓ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into DS

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into DS

transfunc[S_0,a] ← S_1 transfunc[S_0,b] ← S_2

⇓ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_1,a] ← S_1 transfunc[S_1,b] ← S_2

⇓ mark S_2

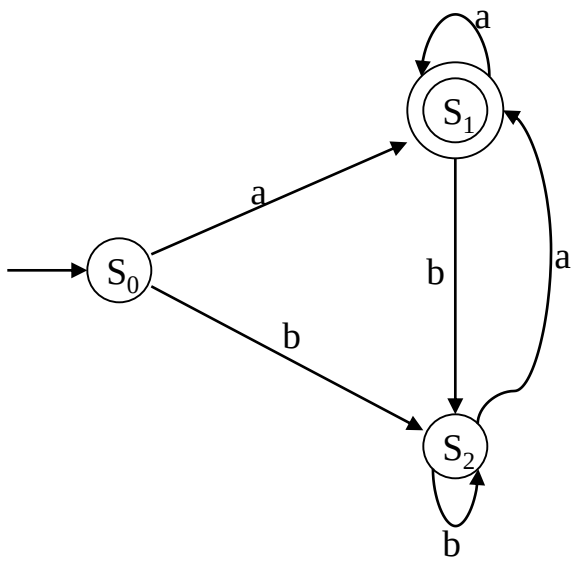
ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_2,a] ← S_1 transfunc[S_2,b] ← S_2

Converting a NFA into a DFA (Example – cont.)

S_0 is the start state of DFA since 0 is a member of S_0={0,1,2,4,7}
S_1 is an accepting state of DFA since 8 is a member of S_1 = {1,2,3,4,6,7,8}



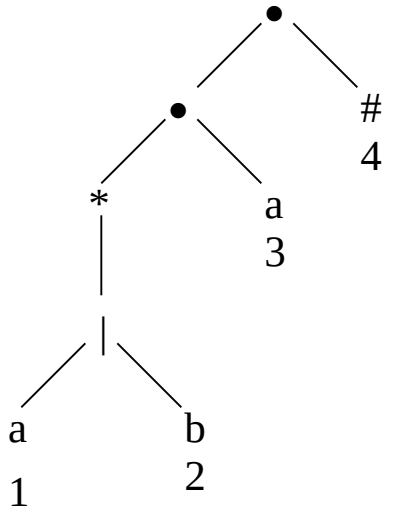
Converting Regular Expressions Directly to DFAs

- Regular expression can be directly converted into a DFA (without creating a NFA first)
- Augment the given regular expression by concatenating it with a special symbol #

  r ➔ (r)# augmented regular expression
- Create a syntax tree for this augmented regular expression
- Syntax tree
  - *Leaves*: alphabet symbols (including # and the empty string) in the augmented regular expression
  - *Intermediate nodes*: operators
- Number each alphabet symbol (including #) depending upon the positions

Regular Expression ➔ DFA (cont.)

$(a|b)^* a$ ➔ $(a|b)^* a \#$ augmented regular expression



Syntax tree of $(a|b)^* a \#$

- each symbol is numbered (positions)
- each symbol is at a leave
- inner nodes are operators

followpos

Define the function **followpos** for the positions (positions assigned to leaves)

followpos(i) -- set of positions which can follow the position i in the strings generated by the augmented regular expression

For example, $(a|b)^* a\ \#$

1 2 3 4

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

followpos is just defined for leaves, it is not defined for inner nodes.

firstpos, lastpos, nullable

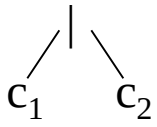
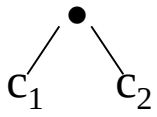
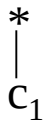
- To evaluate *followpos*, we need three more functions to be defined for the nodes (not just for leaves) of the syntax tree

- **firstpos(n)** -- set of the positions of the **first** symbols of strings generated by the sub-expression rooted by n

- **lastpos(n)** -- set of the positions of the **last** symbols of strings generated by the sub-expression rooted by n

- **nullable(n)** -- *true* if the empty string is a member of strings generated by the sub-expression rooted by n
  false otherwise

How to evaluate firstpos, lastpos, nullable?

n	nullable(n)	firstpos(n)	lastpos(n)
leaf labeled ε	true	Φ	Φ
leaf labeled with position i	false	{i}	{i}
	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
	nullable(c_1) and nullable(c_2)	if (nullable(c_1)) firstpos(c_1) ∪ firstpos(c_2) else firstpos(c_1)	if (nullable(c_2)) lastpos(c_1) ∪ lastpos(c_2) else lastpos(c_2)
	true	firstpos(c_1)	lastpos(c_1)

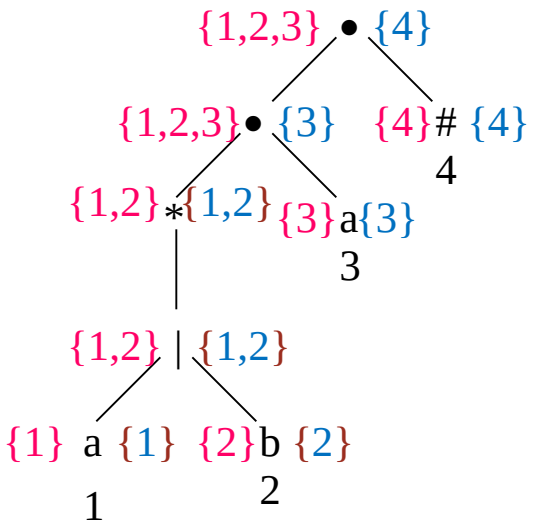
How to evaluate followpos?

Two-rules define the function *followpos*:

1. If **n** is concatenation-node with left child c_1 and right child c_2, and **i** is a position in **lastpos(c_1)**, then all positions in **firstpos (c_2)** are in **followpos (i)**
2. If **n** is a star-node, and **i** is a position in **lastpos(n),** then all positions in **firstpos(n)** are in **followpos(i)**.

- If *firstpos* and *lastpos* have been computed for each node, *followpos* of each position can be computed by making one *depth-first traversal* of the syntax tree

Example -- (a | b)* a



pink – firstpos
blue – lastpos

Then we can calculate followpos

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

- *After we calculate follow positions, we are ready to create DFA for the regular expression*

Algorithm (RE ➔ DFA)

- Create the syntax tree of (r) #
- Calculate the functions: followpos, firstpos, lastpos, nullable
- Put firstpos (root) into the states of DFA as an unmarked state
- *while* (there is an unmarked state S in the states of DFA) *do*
  - mark **S**
  - *for each* input symbol **a** *do*
    - let $s_1,\ldots,s_n$ are positions in **S** and symbols in those positions are **a**
    - **S'** ← followpos(s_1) ∪ … ∪ followpos(s_n)
    - **move(S, a)** ← **S'**
    - if (**S'** is not empty and not in the states of DFA)
      - put **S'** into the states of DFA as an unmarked state

- *the start state of DFA is firstpos (root)*
- *the accepting states of DFA are all states containing the position of #*

Example -- (a | b) * a

1 2 3 4

followpos(1)={1,2,3} followpos(2)={1,2,3} followpos(3)={4} followpos(4)={}

S_1=firstpos(root)={1,2,3}

⇓ mark S_1

a: followpos (1) ∪ followpos(3)={1,2,3,4}=S_2 move(S_1,a)=S_2

b: followpos (2)={1,2,3}=S_1 move(S_1,b)=S_1

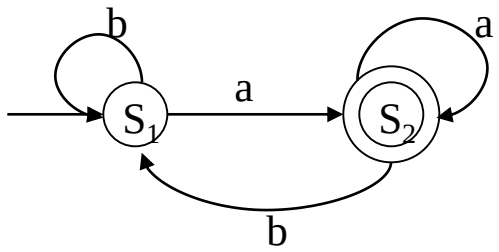
⇓ mark S_2

a: followpos(1) ∪ followpos(3)={1,2,3,4}=S_2 move(S_2,a)=S_2

b: followpos(2)={1,2,3}=S_1 move(S_2,b)=S_1

start state: S_1

accepting states: {S_2}



Example -- (a | ε) b c*

1 2 3 4

followpos(1)={2} followpos(2)={3,4} followpos(3)={3,4} followpos(4)={}

S_1=firstpos(root)={1,2}

⇓ mark S_1

a: followpos(1)={2}=S_2 move(S_1,a)=S_2

b: followpos(2)={3,4}=S_3 move(S_1,b)=S_3

⇓ mark S_2

b: followpos(2)={3,4}=S_3 move(S_2,b)=S_3

⇓ mark S_3

c: followpos(3)={3,4}=S_3 move(S_3,c)=S_3

start state: S_1

accepting states: {S_3}