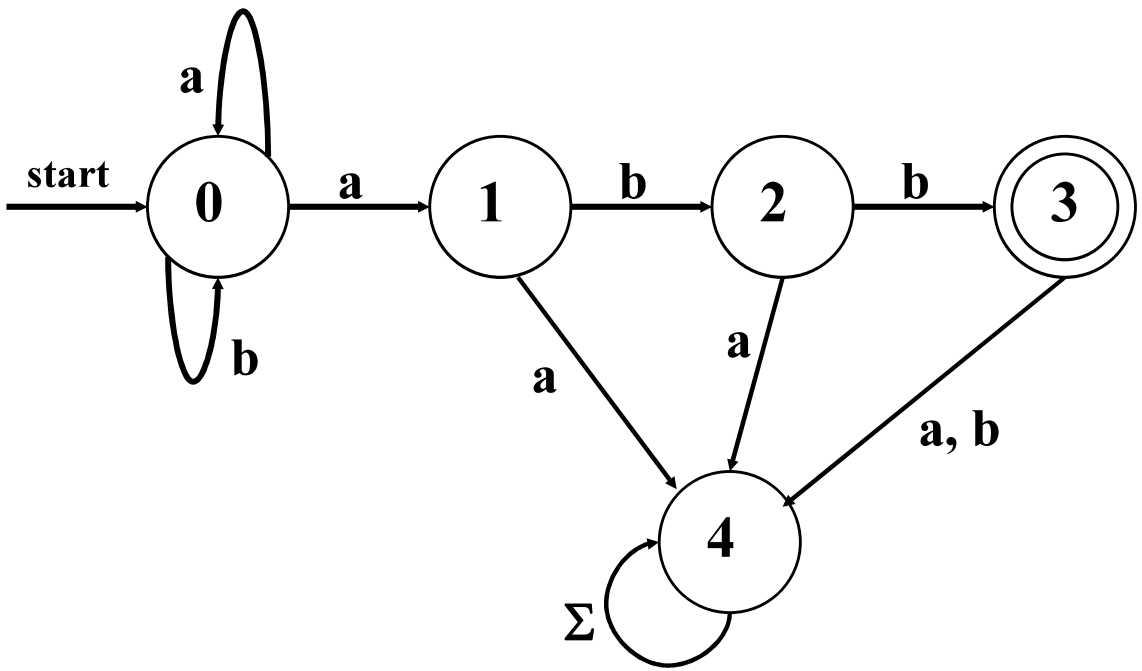
Handling Undefined Transitions

We can handle undefined transitions by defining one more state, a "death" state, and transitioning all previously undefined transition to this death state.



NFA- Regular Expressions & Compilation

Problems with NFAs for Regular Expressions:

1. Valid input might not be accepted
2. NFA may behave differently on the same input

Relationship of NFAs to Compilation:

1. Regular expression "recognized" by NFA
2. Regular expression is "pattern" for a "token"
3. Tokens are building blocks for lexical analysis
4. Lexical analyzer can be described by a collection of NFAs. Each NFA is for a language token.

Second NFA Example

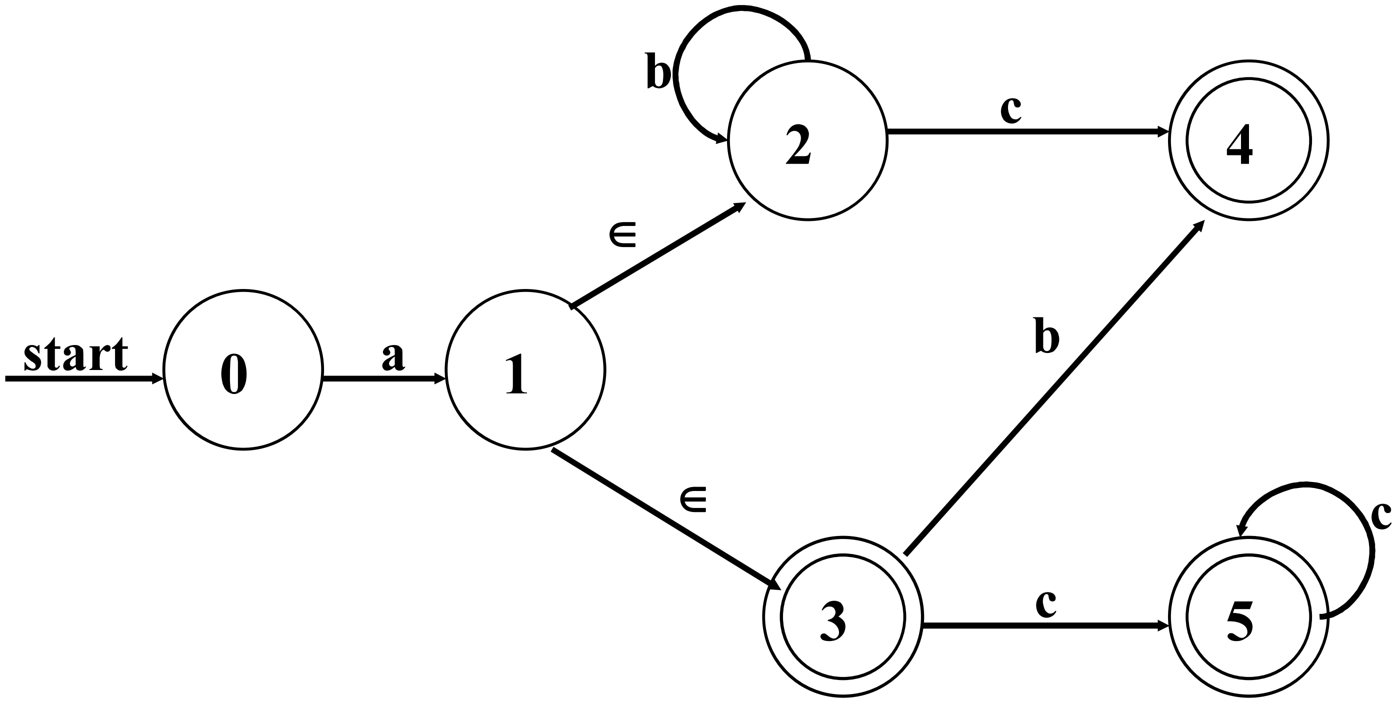
Given the regular expression : $(a\ (b^*c))\ |\ (a\ (b\ |\ c^+)?)$

Find a transition diagram NFA that recognizes it.

Second NFA Example - Solution

Given the regular expression : **$(a\ (b^*c))\ |\ (a\ (b\ |\ c^+)?)$**

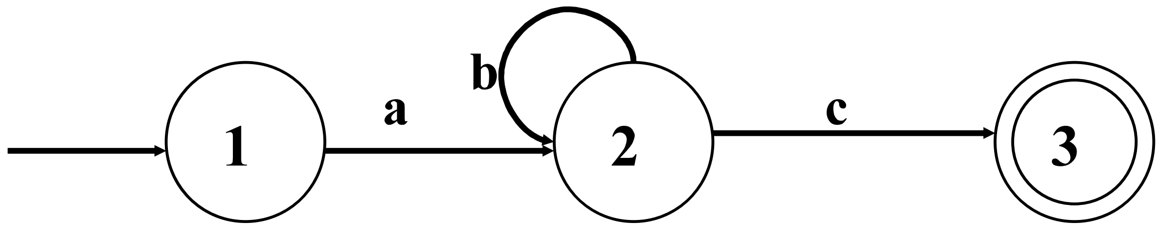
Find a transition diagram NFA that recognizes it.



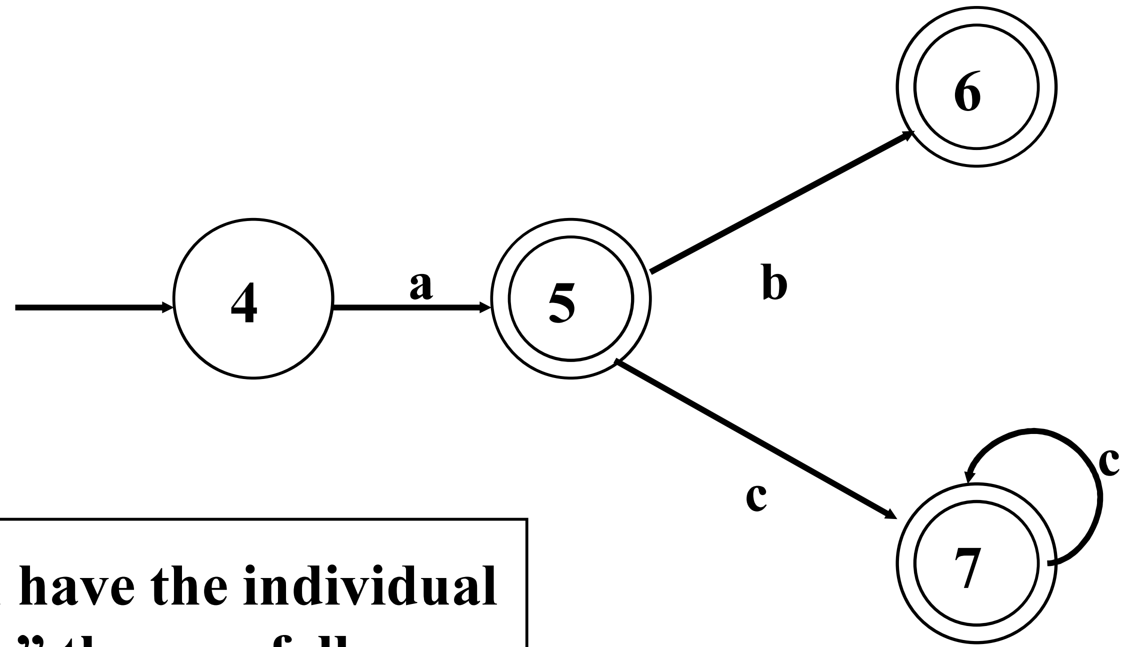
String abbc can be accepted.

Alternative Solution Strategy

a (b*c)

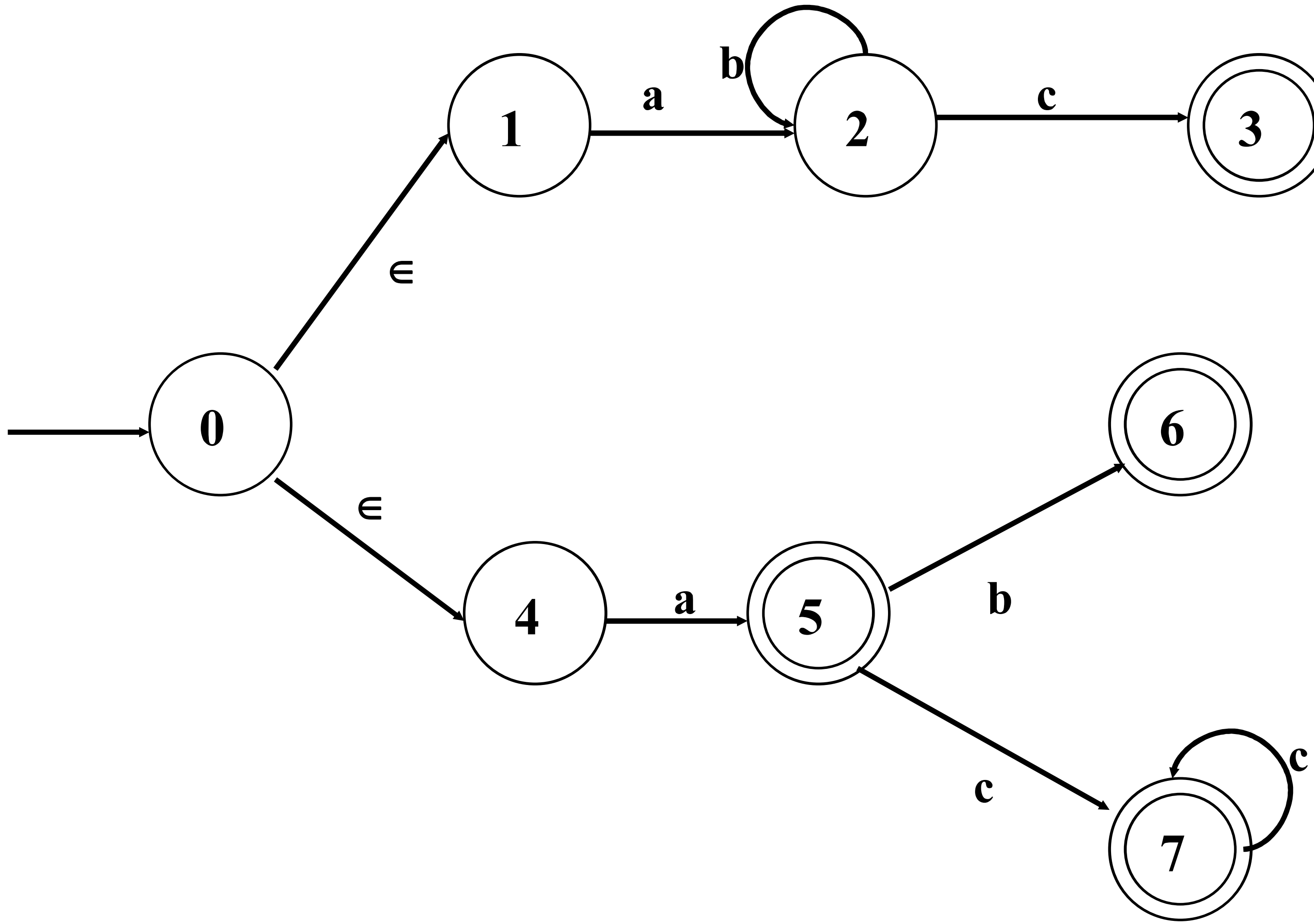


a (b | c+)?



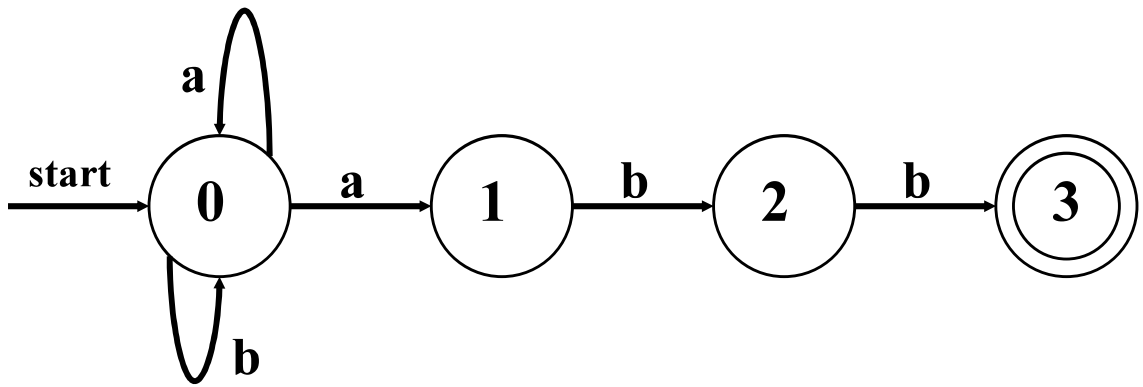
Now that you have the individual diagrams, "or" them as follows:

Using Null Transitions to "OR" NFAs



Other Concepts

Not all paths may result in acceptance.



aabb is accepted along path : $0 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3$

BUT… it is not accepted along the valid path:

$$0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0$$

Deterministic Finite Automata

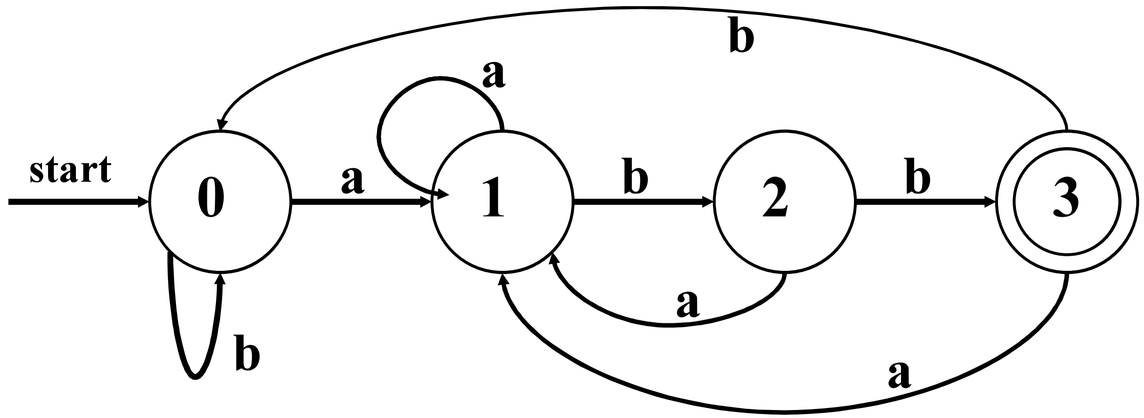
A DFA is an NFA with the following restrictions:

- **$\in$ moves are <u>not</u> allowed**
- **For every state s $\in$S, there is one and only one path from s for every input symbol a $\in \Sigma$.**

Since transition tables don't have any alternative options, DFAs are easily simulated via an algorithm.

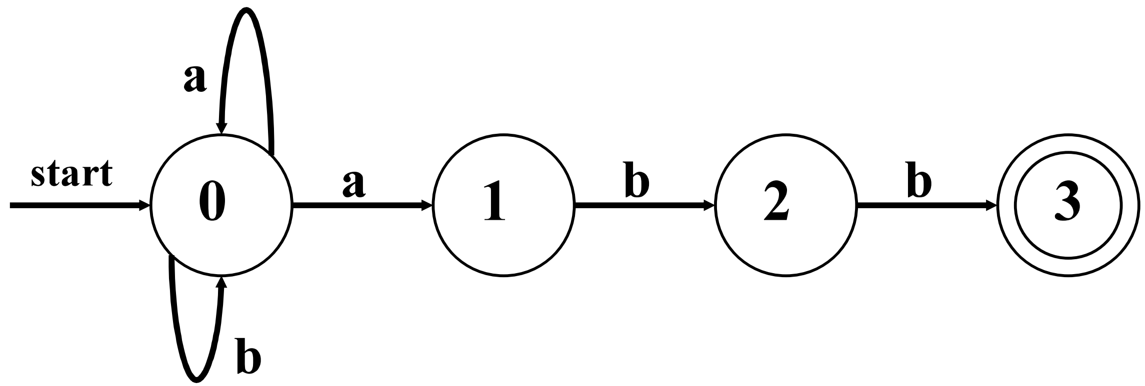
```
s ← s0
c ← nextchar;
while c ≠ eof do
   s ← move(s,c);
   c ← nextchar;
end;
if s is in F then return "yes"
             else return "no"
```

Example - DFA



What Language is Accepted?

Recall the original NFA:



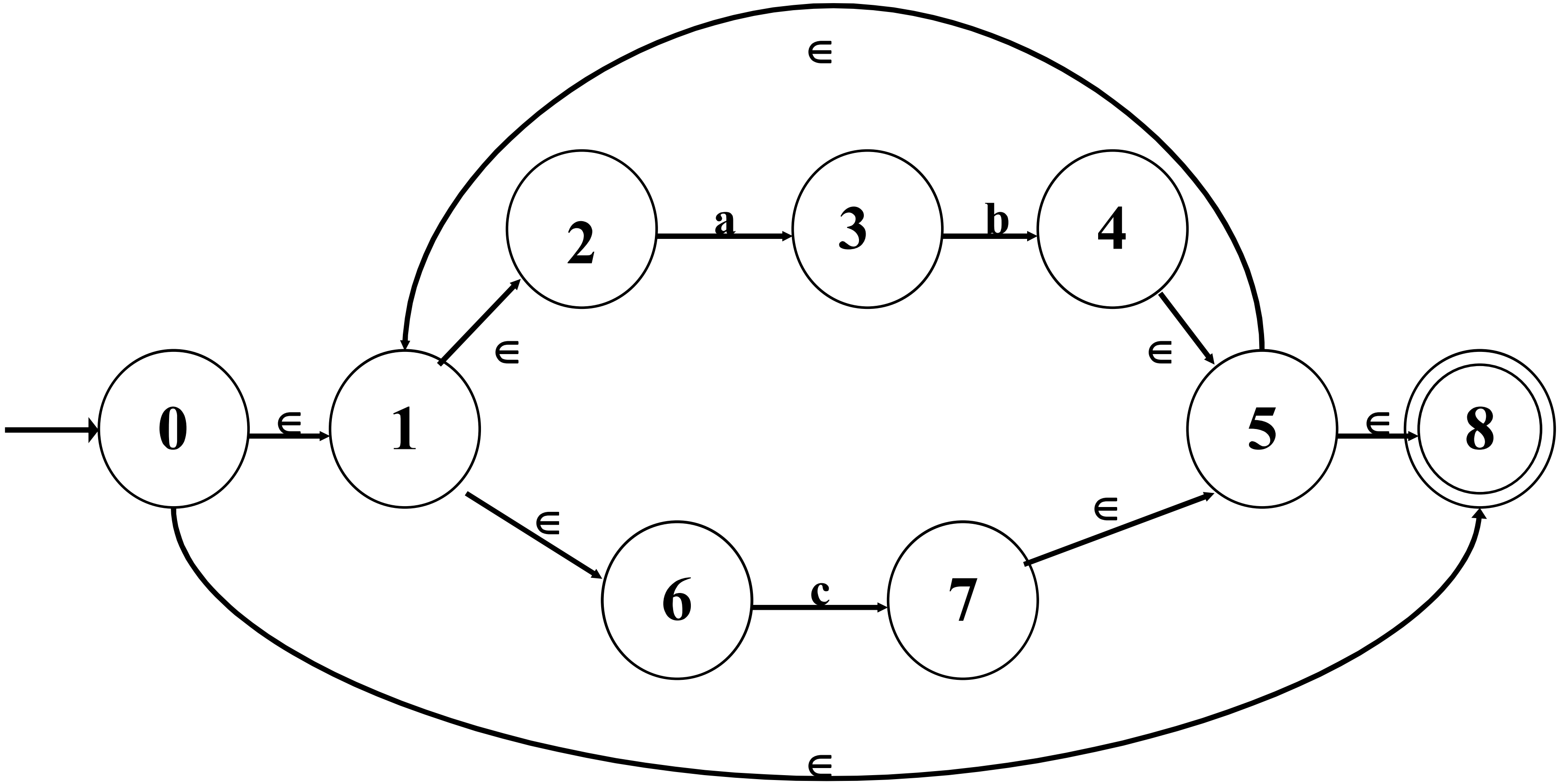
Conversion : NFA → DFA Algorithm

- **Algorithm Constructs a Transition Table for DFA from NFA**
- **Each state in DFA corresponds to a SET of states of the NFA**
- **Why does this occur ?**
  - **$\in$ moves**
  - **non-determinism**

  Both require us to characterize multiple situations that occur for accepting the same string.

  (Recall : Same input can have multiple paths in NFA)
- **Key Issue : Reconciling AMBIGUITY !**

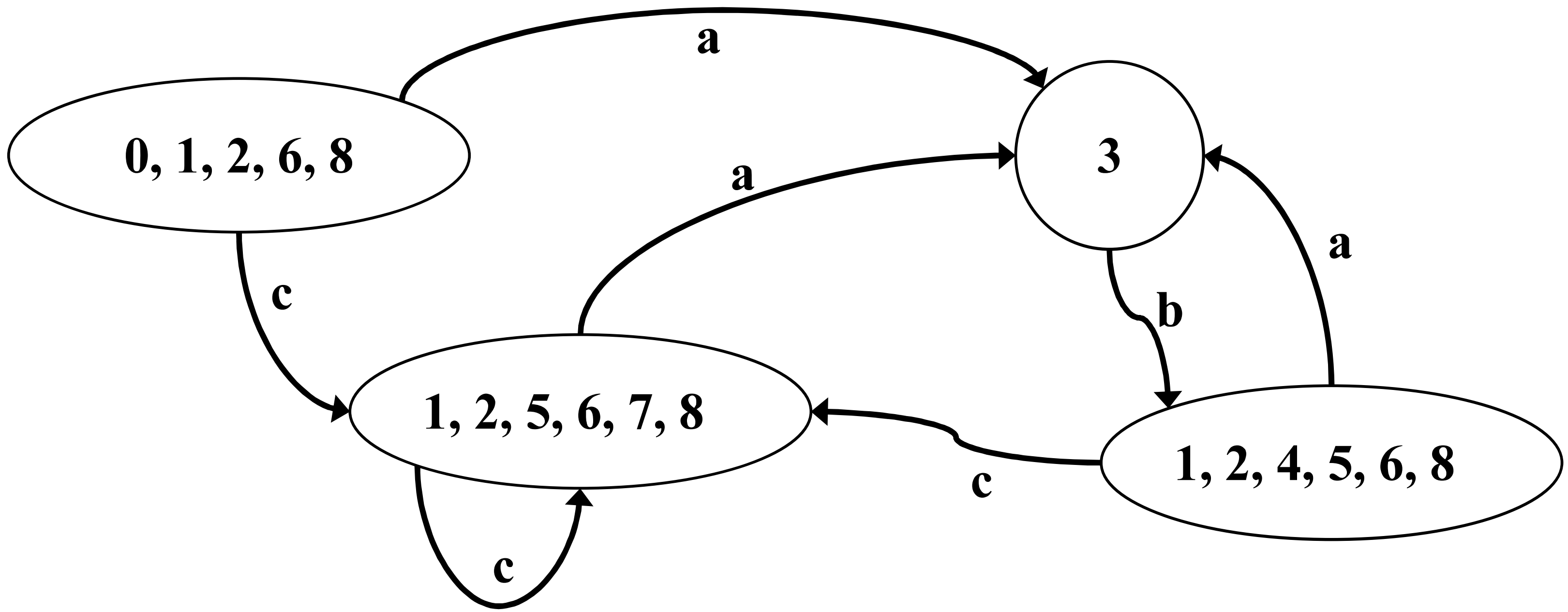
Converting NFA to DFA – 1st Look



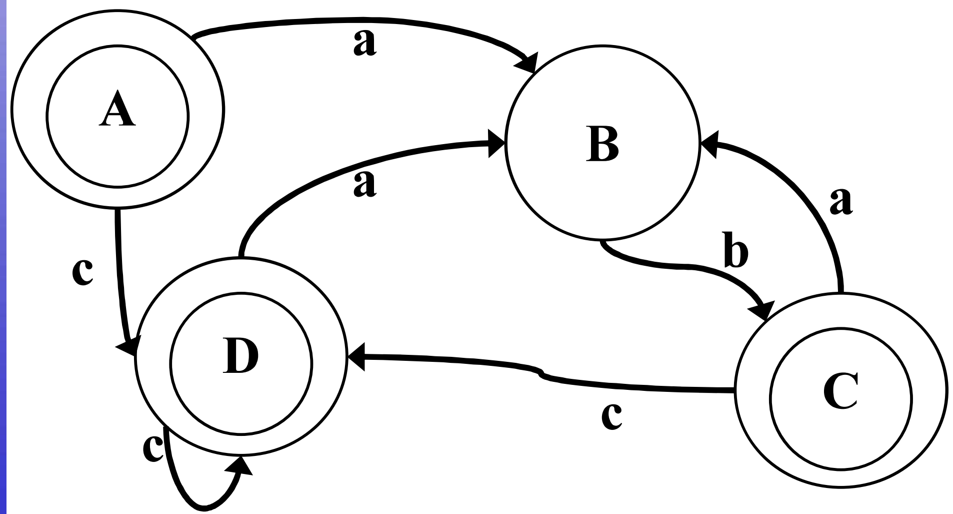
From State 0, Where can we move without consuming any input ?

This forms a new state: 0,1,2,6,8   What transitions are defined for this new state ?

The Resulting DFA



Which States are FINAL States ?



How do we handle alphabet symbols not defined for A, B, C, D ?

Algorithm Concepts

NFA $N = (S, \Sigma, s_0, F, MOVE)$

$\in$-Closure(s) : $s \in S$

: set of states in S that are reachable from s via $\in$-moves of N that originate from s.

No input is consumed

$\in$-Closure(T) : $T \subseteq S$

: NFA states reachable from all $t \in T$ on $\in$-moves only.

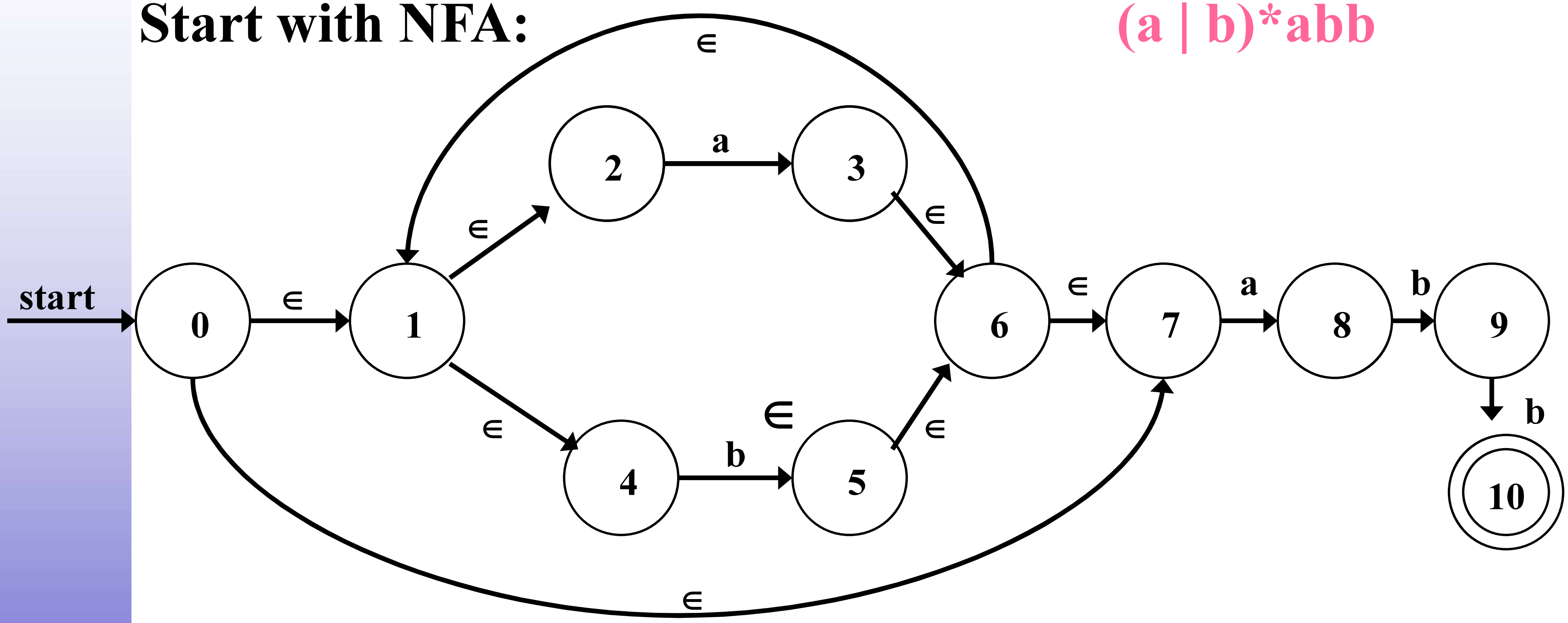
***move*(T,a)** : $T \subseteq S$, $a \in \Sigma$

: Set of states to which there is a transition on input a from some $t \in T$

These 3 operations are utilized by algorithms / techniques to facilitate the conversion process.

Illustrating Conversion – An Example

Start with NFA: (a | b)*abb



First we calculate: ∈-closure(0) (i.e., state 0)

∈-closure(0) = {0, 1, 2, 4, 7} (all states reachable from 0 on ∈-moves)

Let A={0, 1, 2, 4, 7} be a state of new DFA, D.

Conversion Example – continued (1)

2nd , we calculate : a : ∈-closure(*move*(A,a)) and
b : ∈-closure(*move*(A,b))

a : ∈-closure(*move*(A,a)) = ∈-closure(*move*({0,1,2,4,7},a))}
adds {3,8} (since *move*(2,a)=3 and *move*(7,a)=8)

From this we have : ∈-closure({3,8}) = {1,2,3,4,6,7,8}
(since 3→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let B={1,2,3,4,6,7,8} be a new state. Define Dtran[A,a] = B.

b : ∈-closure(*move*(A,b)) = ∈-closure(*move*({0,1,2,4,7},b))

adds {5} (since *move*(4,b)=5)

From this we have : ∈-closure({5}) = {1,2,4,5,6,7}
(since 5→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let C={1,2,4,5,6,7} be a new state. Define Dtran[A,b] = C.

Conversion Example – continued (2)

3rd , we calculate for state B on {a,b}

a : ∈-closure(*move*(B,a)) = ∈-closure(*move*({1,2,3,4,6,7,8},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[B,a] = B.

b : ∈-closure(*move*(B,b)) = ∈-closure(*move*({1,2,3,4,6,7,8},b))}
= {1,2,4,5,6,7,9} = D

Define Dtran[B,b] = D.

4th , we calculate for state C on {a,b}

a : ∈-closure(*move*(C,a)) = ∈-closure(*move*({1,2,4,5,6,7},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[C,a] = B.

b : ∈-closure(*move*(C,b)) = ∈-closure(*move*({1,2,4,5,6,7},b))}
= {1,2,4,5,6,7} = C

Define Dtran[C,b] = C.

Conversion Example – continued (3)

5th , we calculate for state D on {a,b}

a : ∈-closure(*move*(D,a)) = ∈-closure(*move*({1,2,4,5,6,7,9},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[D,a] = B.

b : ∈-closure(*move*(D,b)) = ∈-closure(*move*({1,2,4,5,6,7,9},b))}
= {1,2,4,5,6,7,10} = E

Define Dtran[D,b] = E.

Finally, we calculate for state E on {a,b}

a : ∈-closure(*move*(E,a)) = ∈-closure(*move*({1,2,4,5,6,7,10},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[E,a] = B.

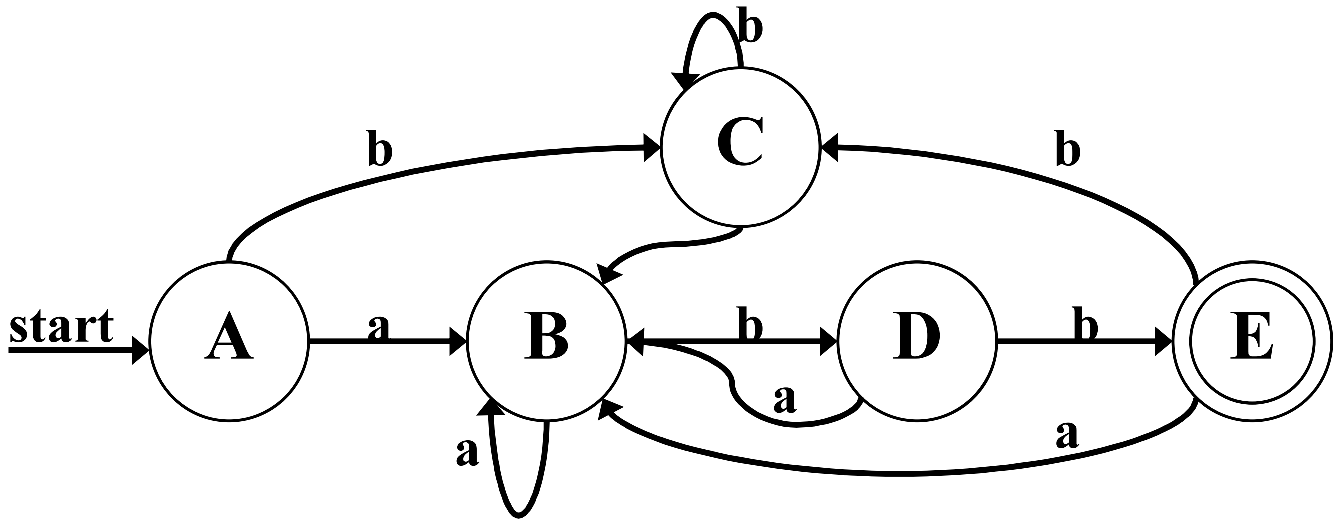
b : ∈-closure(*move*(*E*,b)) = ∈-closure(*move*({1,2,4,5,6,7,10},b))}
= {1,2,4,5,6,7} = C

Define Dtran[E,b] = C.

Conversion Example – continued (4)

This gives the transition table Dtran for the DFA of:

Dstates	Input Symbol a	Input Symbol b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Algorithm For Subset Construction

computing the ∈-closure

```
push all states in T onto stack;
initialize ∈-closure(T) to T;
while stack is not empty do begin
   pop t, the top element, off the stack;
   for each  state u with edge from t to u labeled ∈  do
       if  u is not in ∈-closure(T)  do begin
           add u to ∈-closure(T) ;
           push u onto stack
   end
end
```

Algorithm For Subset Construction – (2)

```
initially, ∈-closure(s0) is only (unmarked) state in Dstates;
while there is unmarked state T in Dstates do begin
   mark T;
   for each input symbol a do begin
         U := ∈-closure(move(T,a));
         if U is not in Dstates then
              add U as an unmarked state to Dstates;
         Dtran[T,a] := U
   end
end
```

Regular Expression to NFA Construction

We now focus on transforming a Reg. Expr. to an NFA

This construction allows us to take:

- **Regular Expressions (which describe tokens)**
- **To an NFA (to characterize language)**
- **To a DFA (which can be "computerized")**

The construction process is component-wise

Builds NFA from components of the regular expression in a special order with particular techniques.

NOTE: Construction is "syntax-directed" translation, i.e., syntax of regular expression is determining factor for NFA construction and structure.

Motivation: Construct NFA For:

∈:

a :

b:

ab:

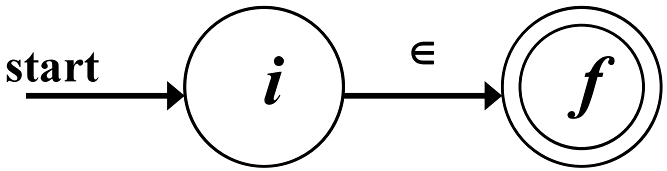
∈ | ab :

a*

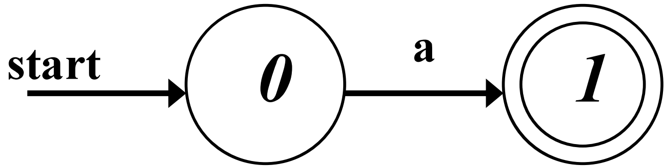
(∈ | ab)* :

Motivation: Construct NFA For:

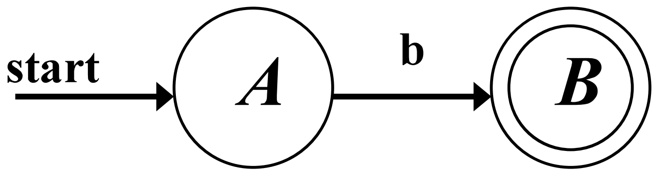
∈ :



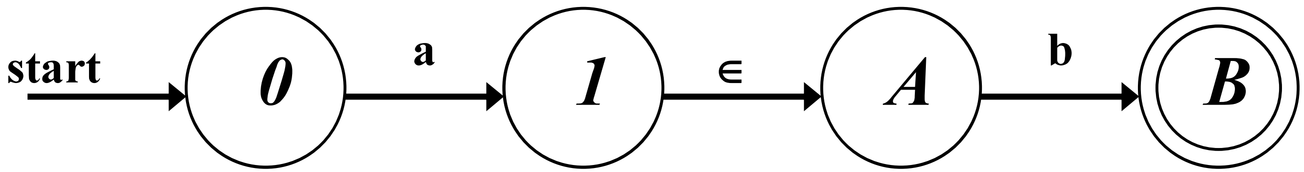
a :



b:



ab:



∈ | ab :

a*

(∈ | ab)* :

Construction Algorithm : R.E. → NFA

Construction Process :

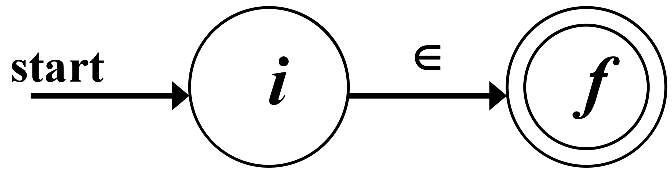
1st : Identify subexpressions of the regular expression

- $\in$
- **Σ symbols**
- **r | s**
- **rs**
- **r***

2nd : Characterize "pieces" of NFA for each subexpression

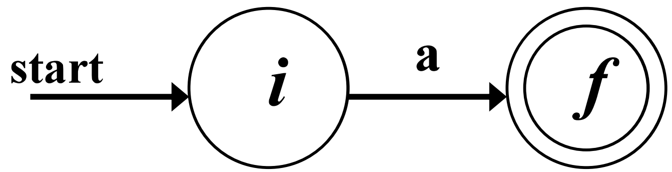
Piecing Together NFAs
Algorithm: Thompson's Construction

1. For $\in$ in the regular expression, construct NFA



L(∈)

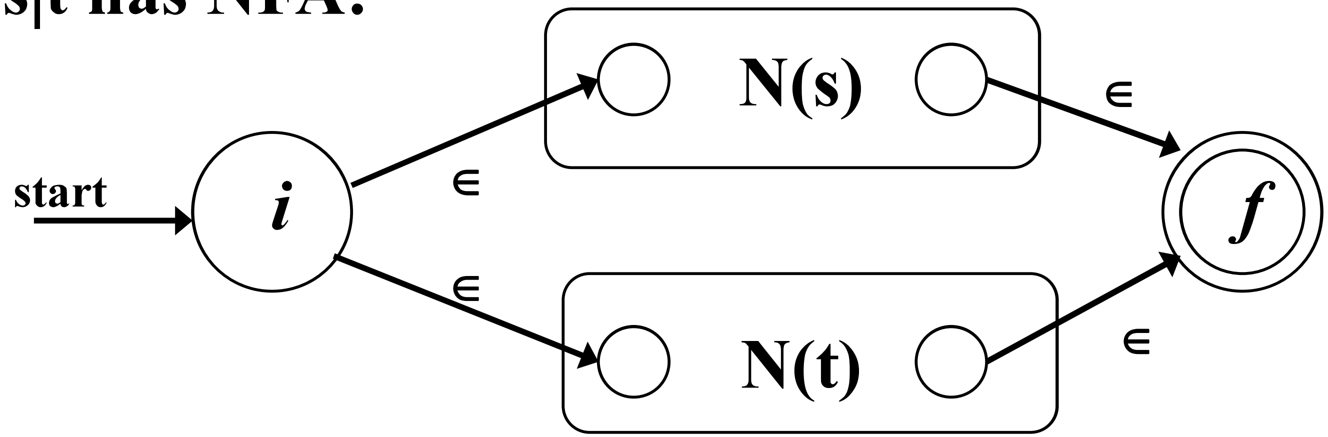
2. For a $\in \Sigma$ in the regular expression, construct NFA



L(a)

Piecing Together NFAs – continued(1)

3.(a) If s, t are regular expressions, N(s), N(t) their NFAs s|t has NFA:

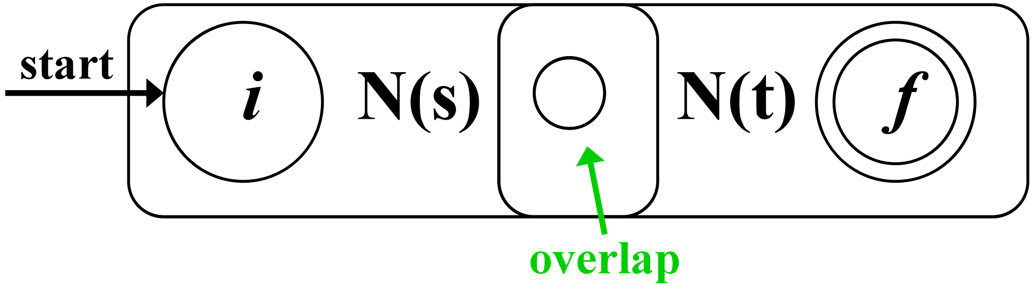


L(s) ∪ L(t)

where *i* and *f* are new start / final states, and ∈-moves are introduced from *i* to the old start states of N(s) and N(t) as well as from all of their final states to *f*.

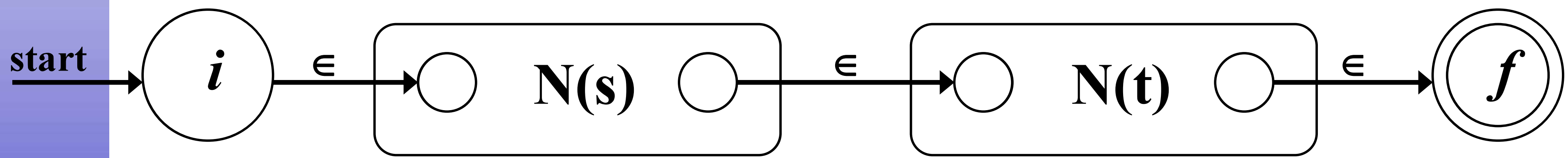
Piecing Together NFAs – continued(2)

3.(b) If s, t are regular expressions, N(s), N(t) their NFAs st (concatenation) has NFA:



L(s) L(t)

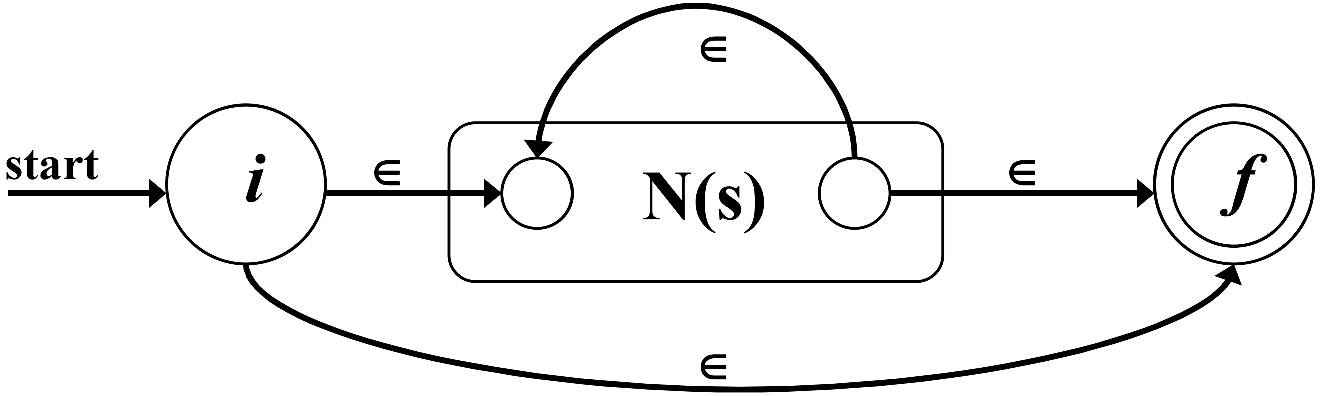
Alternative:



where *i* is the start state of N(s) (or new under the alternative) and *f* is the final state of N(t) (or new). Overlap maps final states of N(s) to start state of N(t).

Piecing Together NFAs – continued(3)

3.(c) If s is a regular expressions, N(s) its NFA, s* (Kleene star) has NFA:



where : *i* is new start state and *f* is new final state

∈-move *i* to *f* (to accept null string)

∈-moves *i* to old start, old final(s) to *f*

∈-move old final to old start (WHY?)

Properties of Construction

Let r be a regular expression, with NFA N(r), then

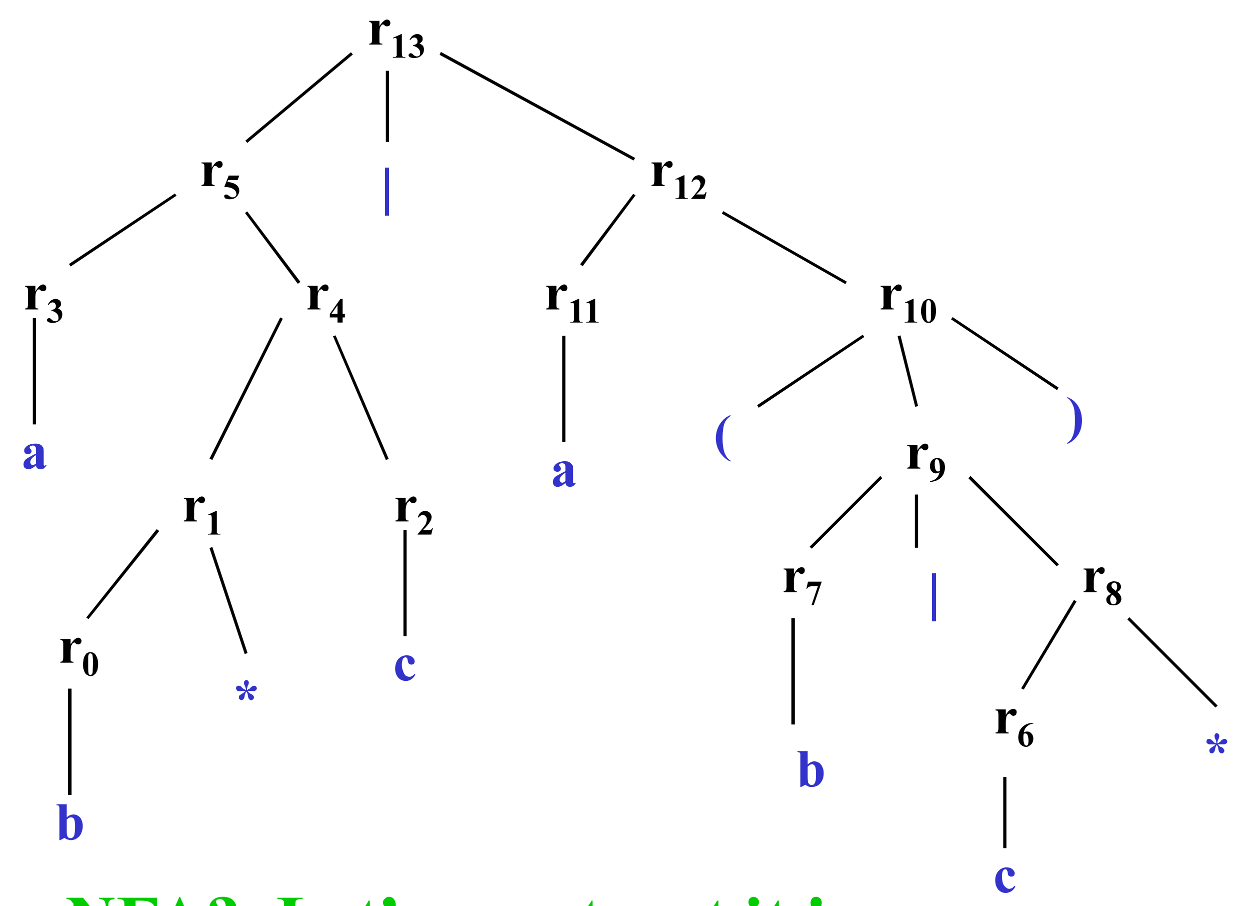
1. **N(r) has #of states $\leq$ 2*(#symbols + #operators) of r**
2. **N(r) has exactly one start and one accepting state**
3. **Each state of N(r) has at most one outgoing edge $a \in \Sigma$ or at most two outgoing $\in$'s**
4. **BE CAREFUL to assign unique names to all states !**

Detailed Example

See example in textbook for (a | b)*abb

2nd Example - (ab*c) | (a(b|c*))

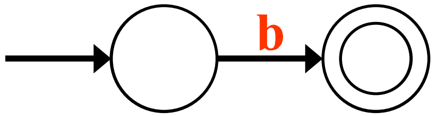
Parse Tree for this regular expression:



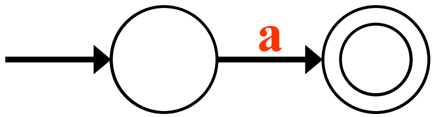
What is the NFA? Let's construct it !

Detailed Example – Construction(1)

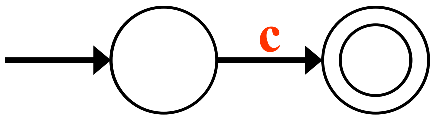
r_0:



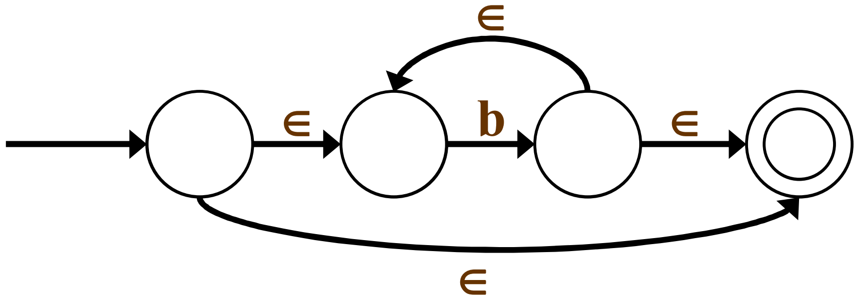
r_3:



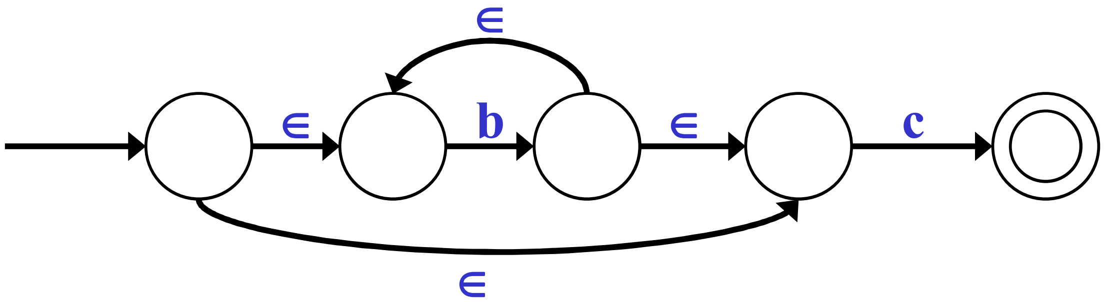
r_2:



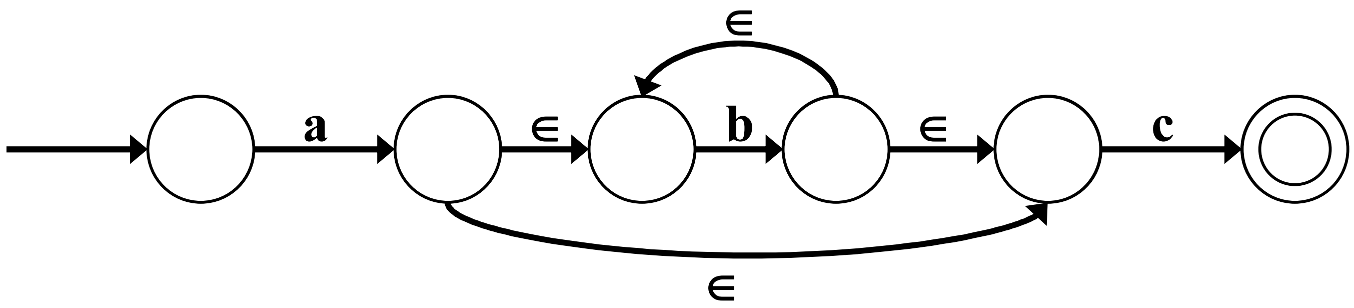
r_1:



$r_4 : r_1\ r_2$



$r_5 : r_3\ r_4$



Detailed Example – Construction(2)

r_7:

r_{11}:

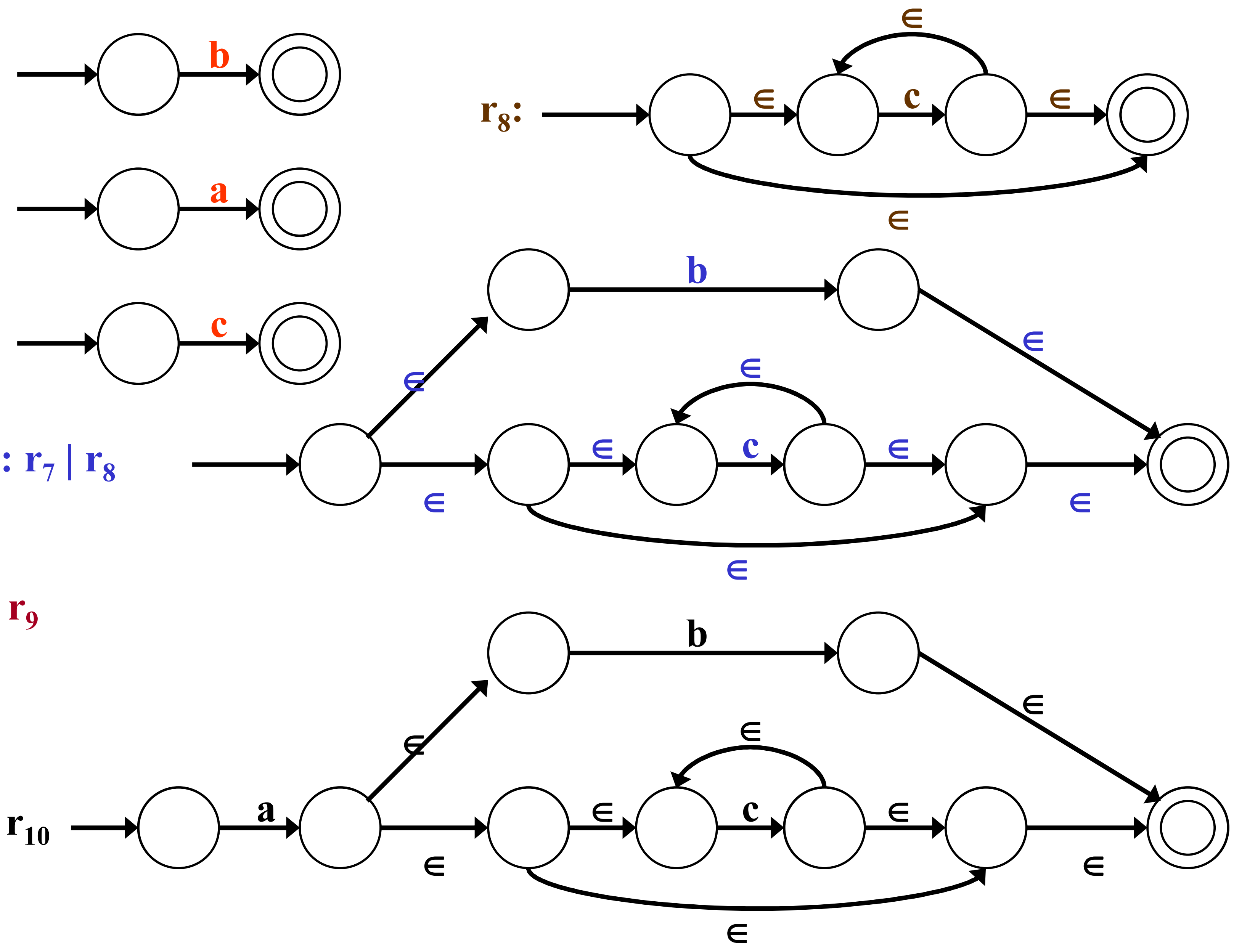
r_6:

r_8:

$r_9 : r_7 \mid r_8$

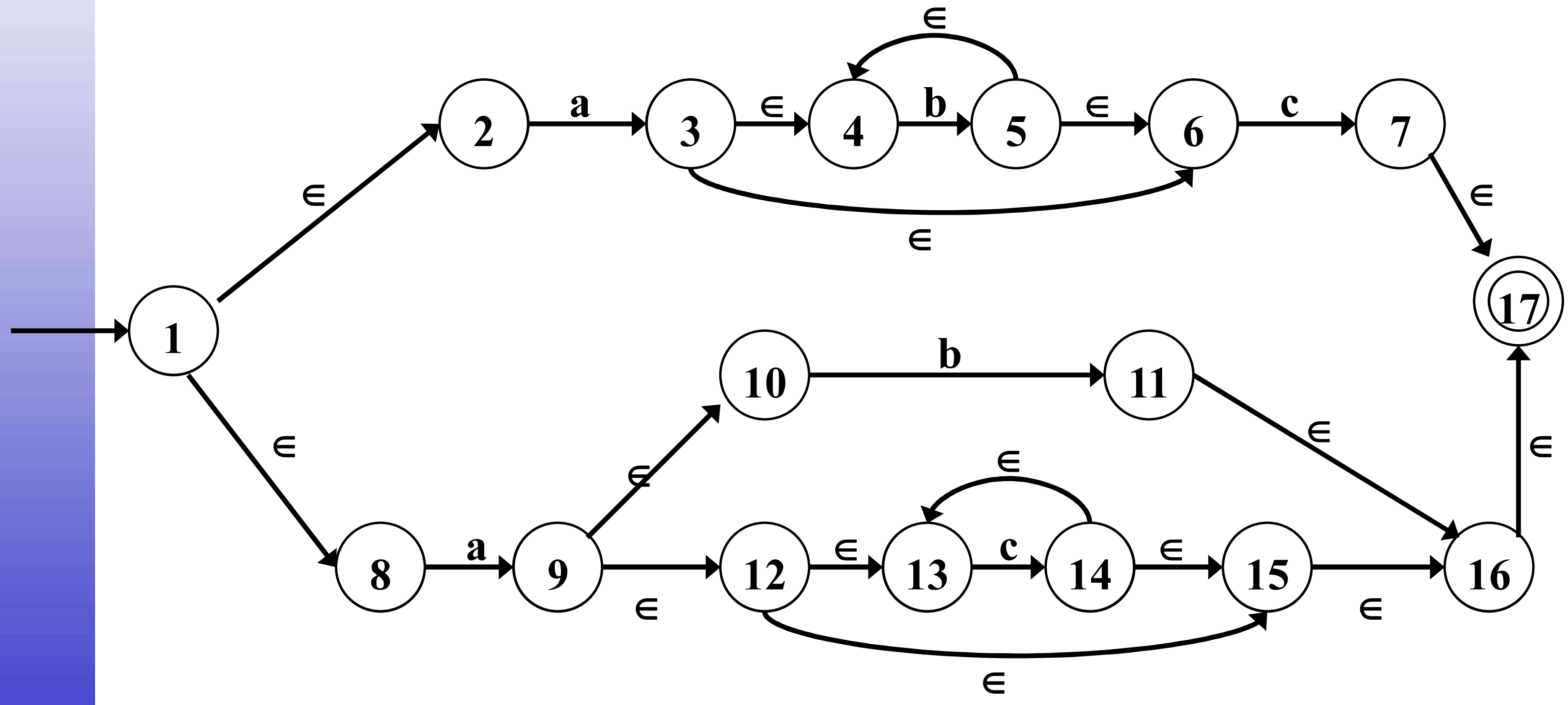
$r_{10} : r_9$

$r_{12} : r_{11}\ r_{10}$



Detailed Example – Final Step

$r_{13} : r_5 \mid r_{12}$



Direct Simulation of an NFA

```
s ← s0
c ← nextchar;
while c ≠ eof do
   s ← move(s,c);
   c ← nextchar;
end;
if s is in F then return "yes"
             else return "no"
```

DFA simulation

```
S ← ∈-closure({s0})
c ← nextchar;
while c ≠ eof do
   S ← ∈-closure(move(S,c));
   c ← nextchar;
end;
if S∩F≠∅ then return "yes"
              else return "no"
```

NFA simulation

Final Notes : R.E. to NFA Construction

- ✓So, an NFA may be simulated by algorithm, when NFA is constructed using Previous techniques
- ✓Algorithm run time is proportional to $|N| * |x|$ where $|N|$ is the number of states and $|x|$ is the length of input
- ✓Alternatively, we can construct DFA from NFA and use the resulting Dtran to recognize input:

Assignment

No marks will be awarded (Part of the Syllabus)

	space required	time to simulate
NFA	$O(\|r\|)$	$O(\|r\|*\|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$

where $|r|$ is the length of the regular expression.

Pulling Together Concepts

- **Designing Lexical Analyzer Generator**

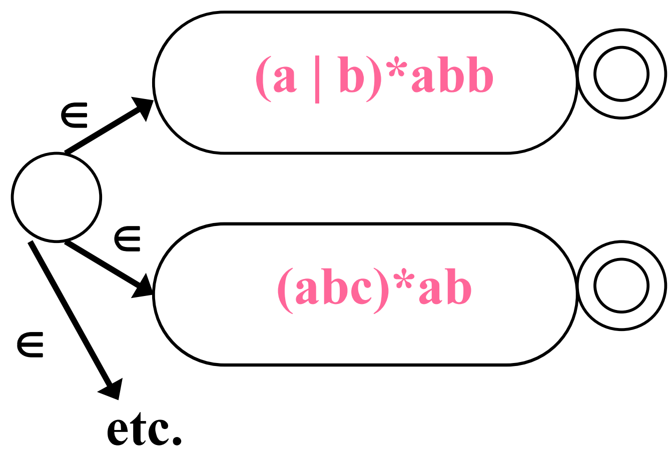
  Reg. Expr. → NFA construction

  NFA → DFA conversion

  DFA simulation for lexical analyzer

- **Recall Lex Structure**

Pattern	**Action**	e.g.
Pattern	**Action**	
…	…	



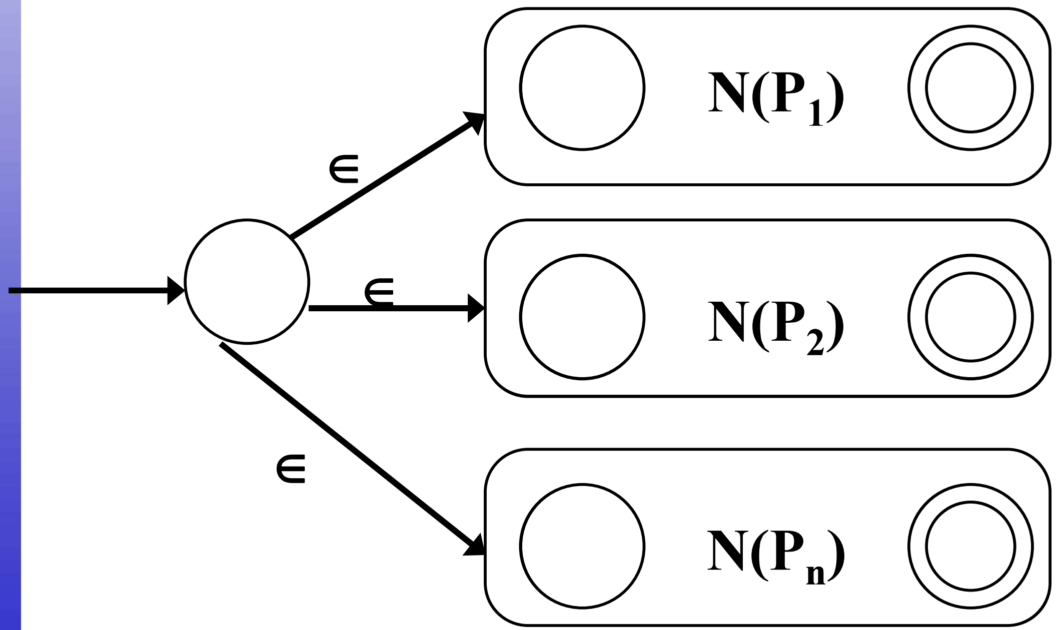
Recognizer!

- Each pattern recognizes lexemes

- Each pattern described by regular expression

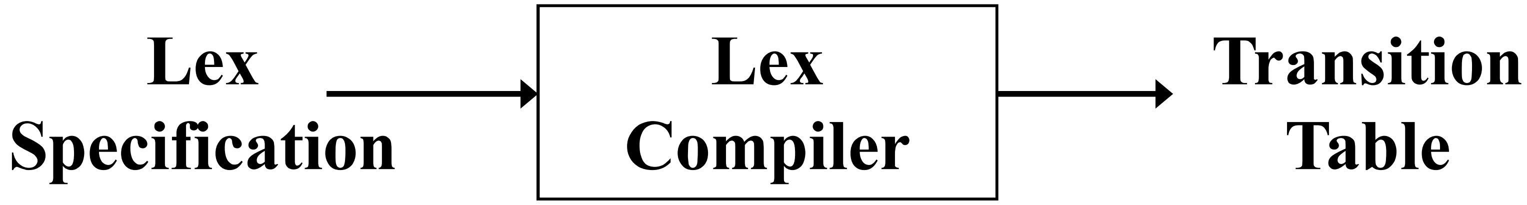
Lex Specification → Lexical Analyzer

- Let P_1, P_2, … , P_n be Lex patterns
  (regular expressions for valid tokens in prog. lang.)
- Construct $N(P_1)$, $N(P_2)$, … $N(P_n)$
- Note: accepting state of $N(P_i)$ will be marked by P_i
- Construct NFA:

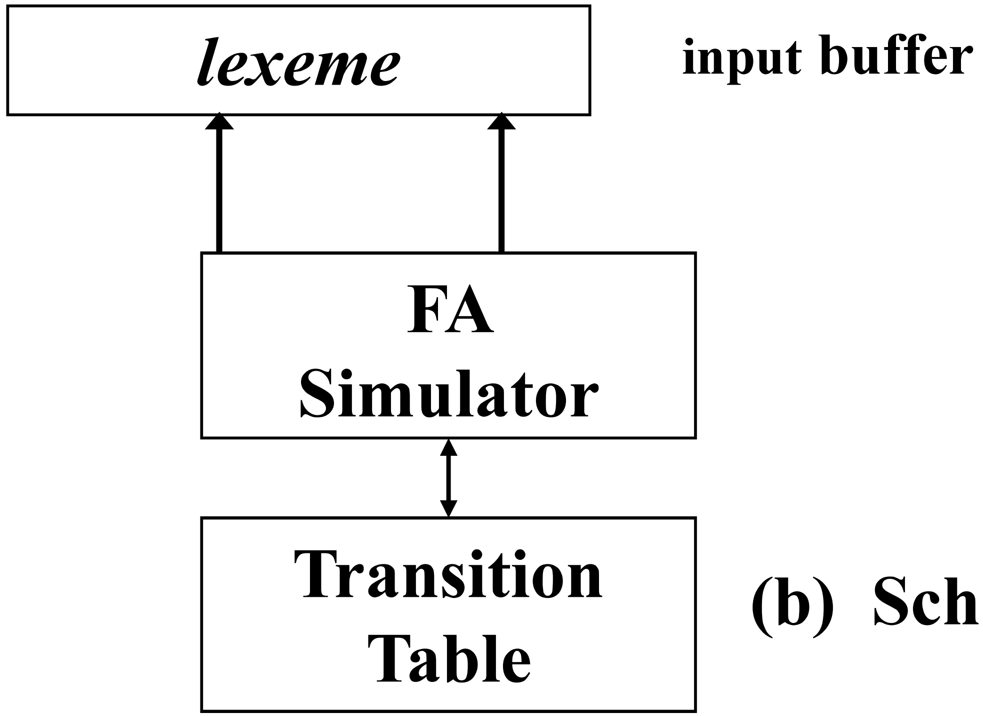


- Lex applies conversion algorithm to construct DFA that is equivalent!

Pictorially



(a) Lex Compiler



(b) Schematic lexical analyzer

Pattern Matching Based on NFA

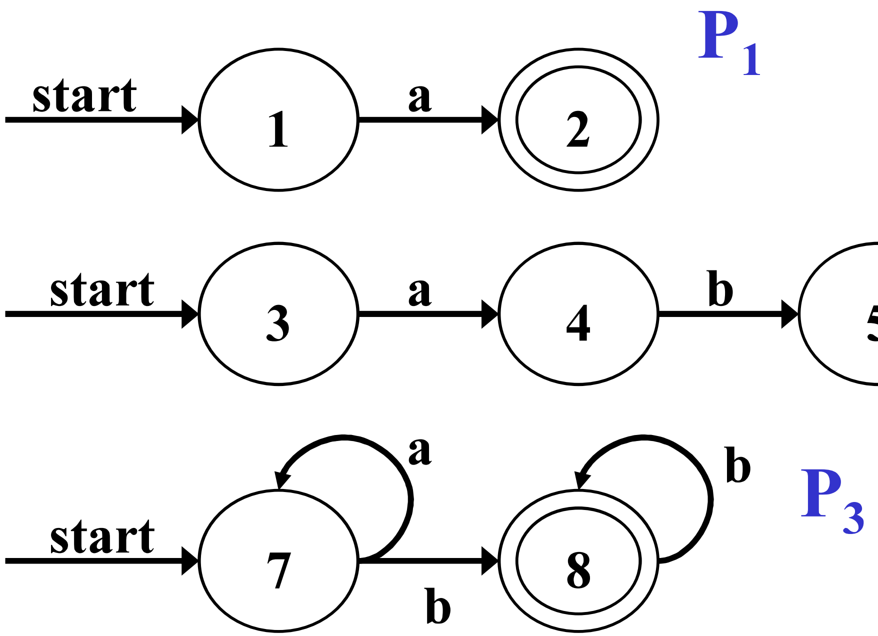
P_1 : a {action}

P_2 : abb {action}

P_3 : a^*b^+ {action}

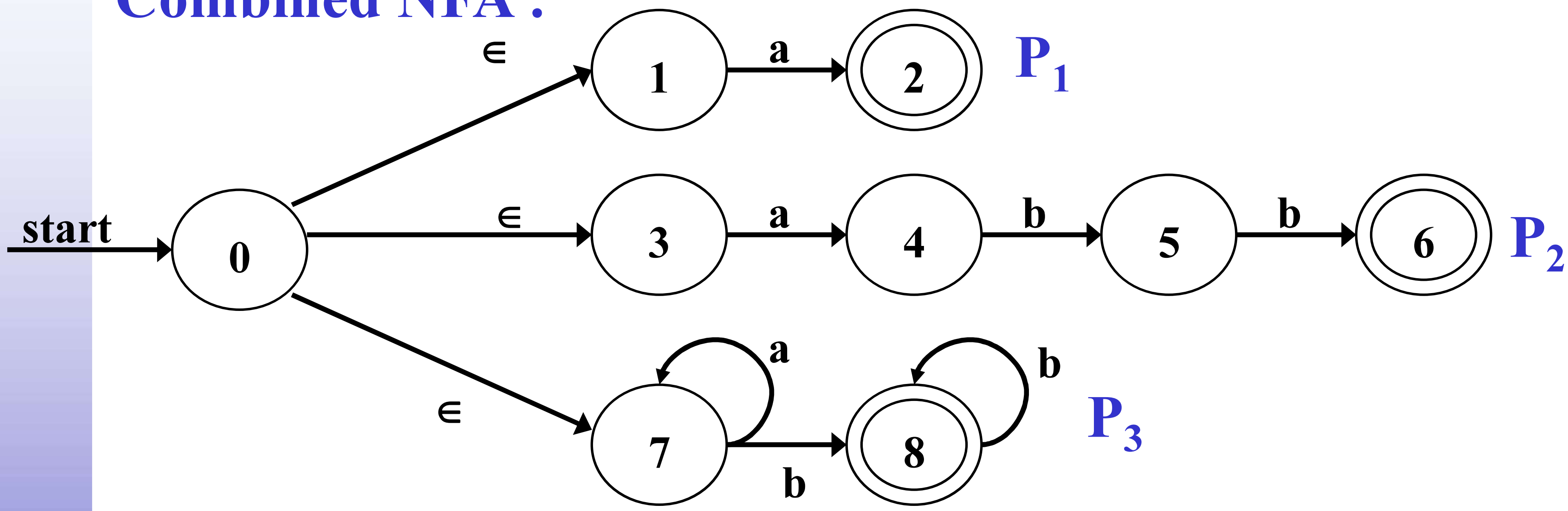
3 patterns

NFA's :



Pattern Matching Based on NFA continued (2)

Combined NFA :



Examples

		a		a		b		a	
	{0,1,3,7}		{2,4,7}		{7}		{8}		death
pattern matched:	-		P_1		-		P_3		-

		a		b		b	
	{0,1,3,7}		{2,4,7}		{5,8}		{6,8}
pattern matched:	-		P_1		P_3		P_2,P_3 ⇐ break tie in favor of P_2

DFA for Lexical Analyzers

Alternatively Construct DFA:

keep track of correspondence between patterns and new accepting states

	Input Symbol		
STATE	**a**	**b**	**Pattern**
{0,1,3,7}	{2,4,7}	{8}	none
{2,4,7}	{7}	{5,8}	P_1
{8}	-	{8}	P_3
{7}	{7}	{8}	none
{5,8}	-	{6,8}	P_3
{6,8}	-	{8}	P_2

break tie in favor of P_2

Example

Input: aaba

$\{0,1,3,7\} \longrightarrow \{2,4,7\} \longrightarrow \{7\} \longrightarrow \{8\}$

Input: aba

$\{0,1,3,7\} \longrightarrow \{2,4,7\} \longrightarrow \{5,8\} \longrightarrow P_3$

	Input Symbol		
STATE	**a**	**b**	**Pattern**
{0,1,3,7}	{2,4,7}	{8}	none
{2,4,7}	{7}	{5,8}	P_1
{8}	-	{8}	P_3
{7}	{7}	{8}	none
{5,8}	-	{6,8}	P_3
{6,8}	-	{8}	P_2

Optimization of DFA based Pattern Matching

Our Target:

1. **Construct DFA directly from Regular Expression**
2. **Minimizes no of states of DFA**
3. **Produce fast but more compact representations for transition table of DFA than a straightforward two dimensional table**

Important States of an NFA

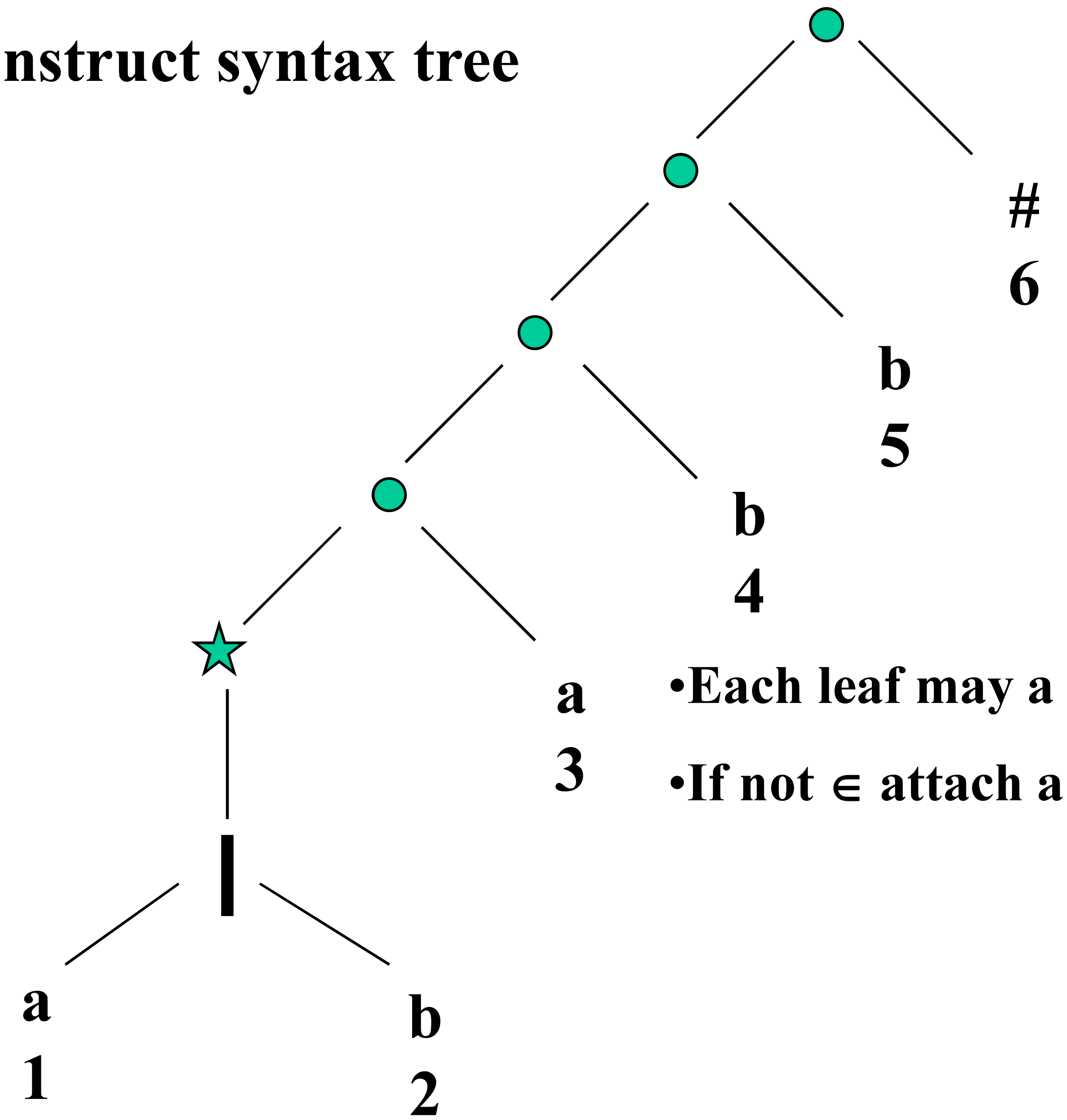
Important States of NFA:

if it has a non-∈ out-transition

- The subset construction algorithm uses only the important states in a subset T when it determines ∈-closure(move(T,a))
- The resulting NFA has exactly one accepting state, but the accepting state is not important because it has no transition leaving it.
- We give the accepting state a transition on #, making it important state of NFA.
- By using augmented regular expression (r)#, we can forget about accepting states as the subset construction proceeds.
- When the construction is complete, any DFA state with a transition on # must be an accepting state.

Example: (a | b)*abb

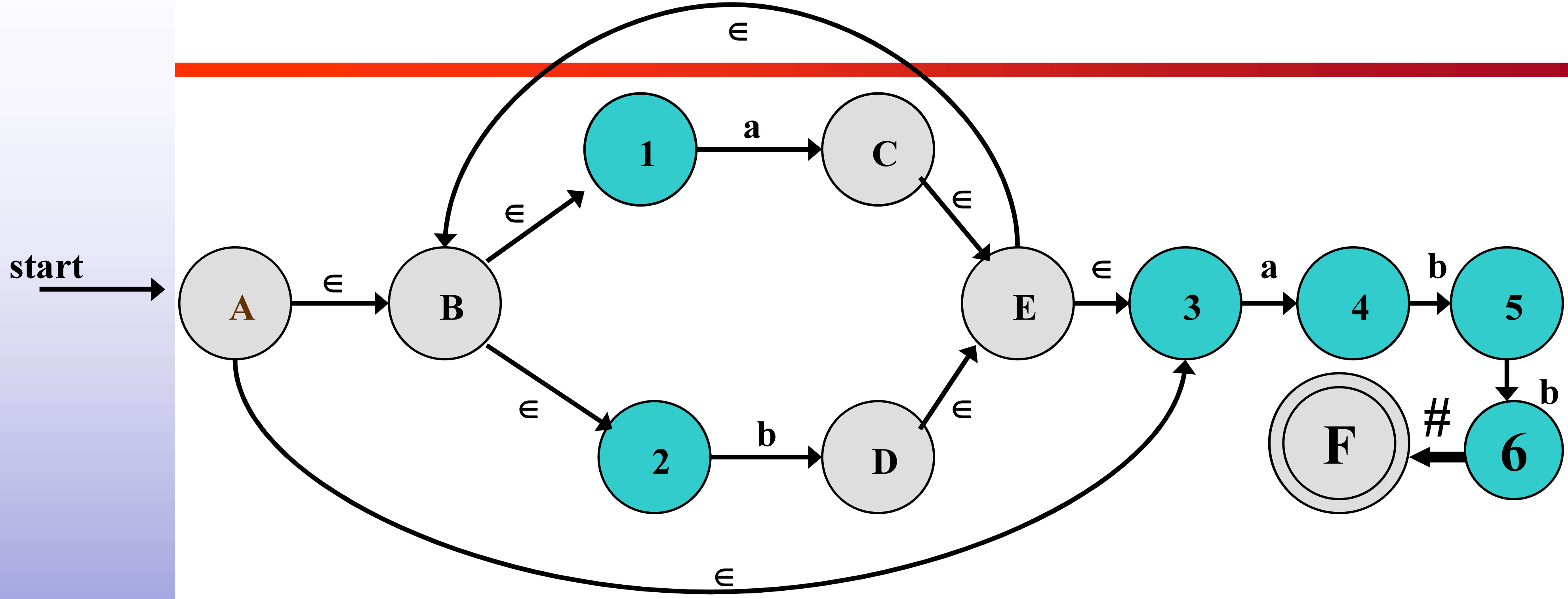
Construct syntax tree

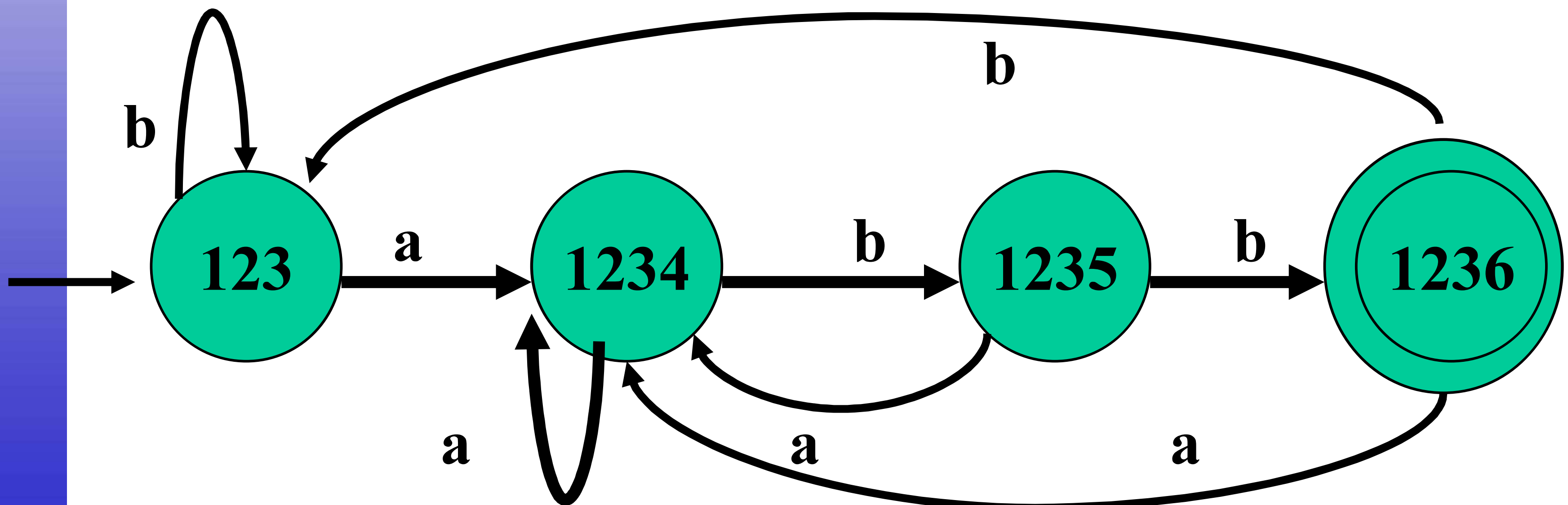


•Each leaf may a symbol / ∈

•If not ∈ attach a unique integer

NFA Vs DFA



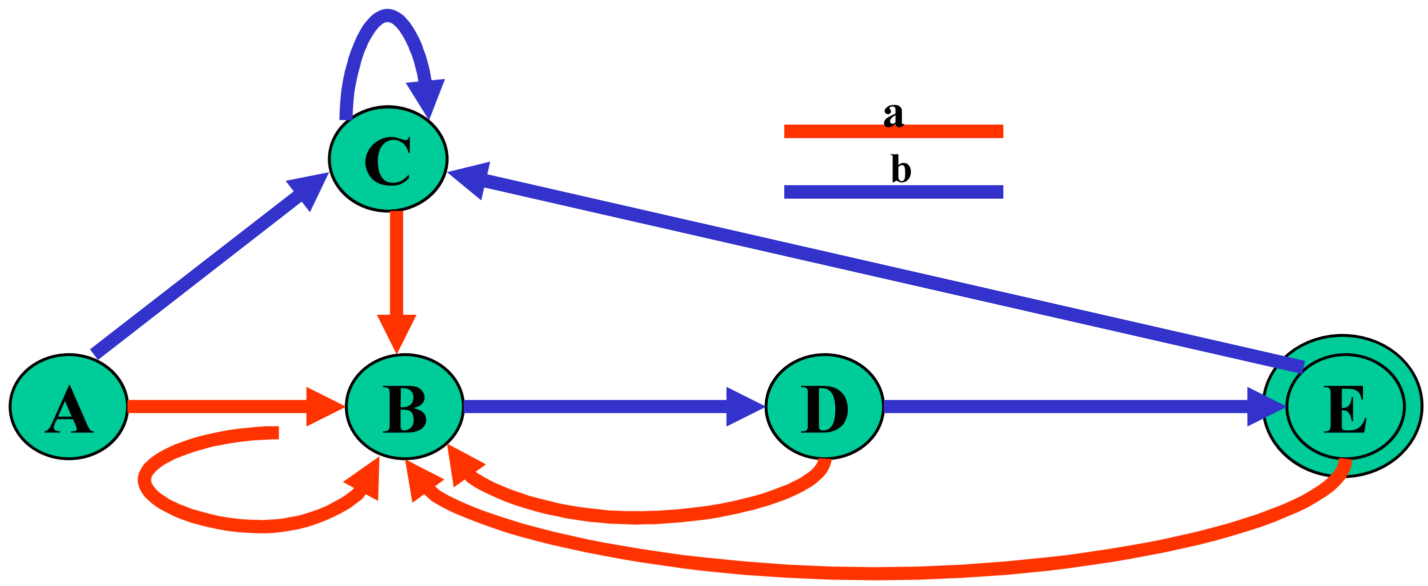


Minimizing the Number of States of DFA

1. Construct initial partition Π of S with two groups: accepting/ non-accepting.
2. (Construct Π_{new})For each group *G* of Π **do begin**
    1. Partition *G* into subgroups such that two states s,t of *G* are in the same subgroup iff for all symbols a states s,t have transitions on a to states of the same group of Π.
    2. Replace *G* in Π_{new} by the set of all these subgroups.
3. Compare Π_{new} and Π. If equal, Π_{final}:= Π then proceed to 4,

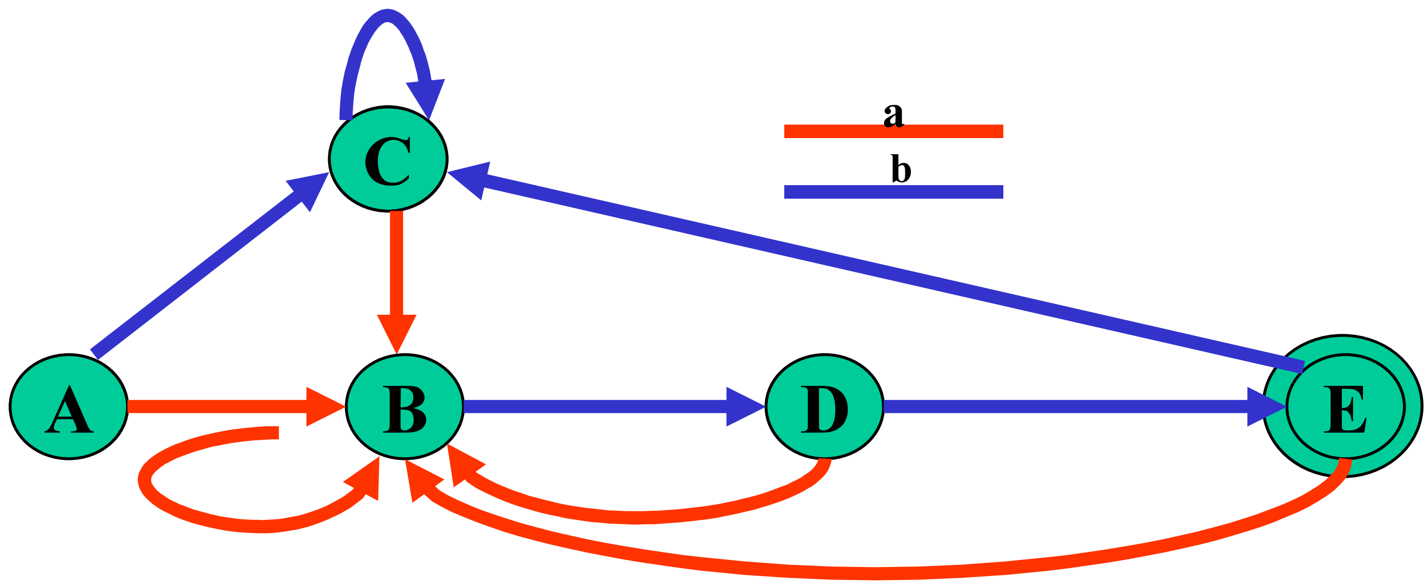
   else set $\Pi := \Pi_{new}$ and goto 2.
4. Aggregate states belonging in the groups of Π_{final}
5. If M' has a dead state, that is, a state d that is not a accepting and that has transitions to itself on all input symbols, then remove d from M'. Also remove any states not reachable from the start state. Any transitions to d from other states become undefined.

Example of minimization of DFA States



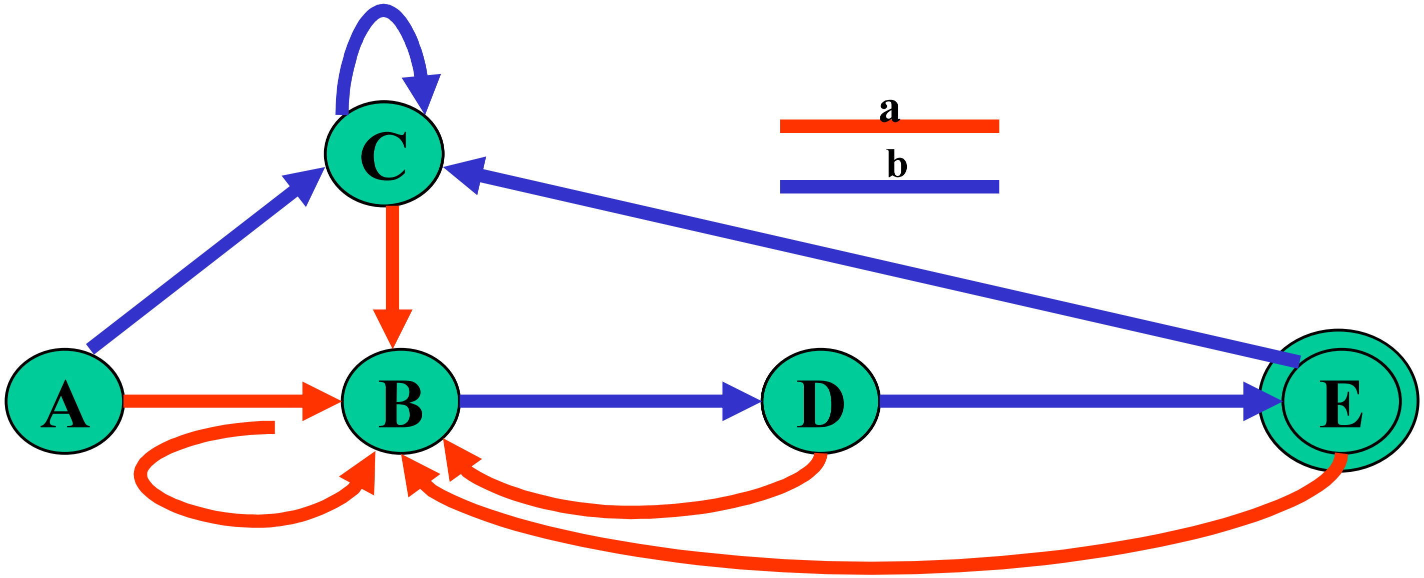
G1	G2
E	ABCD
No Change	a→ G2 b→ G2 G1 (ABC) (D)

Example of minimization of DFA States



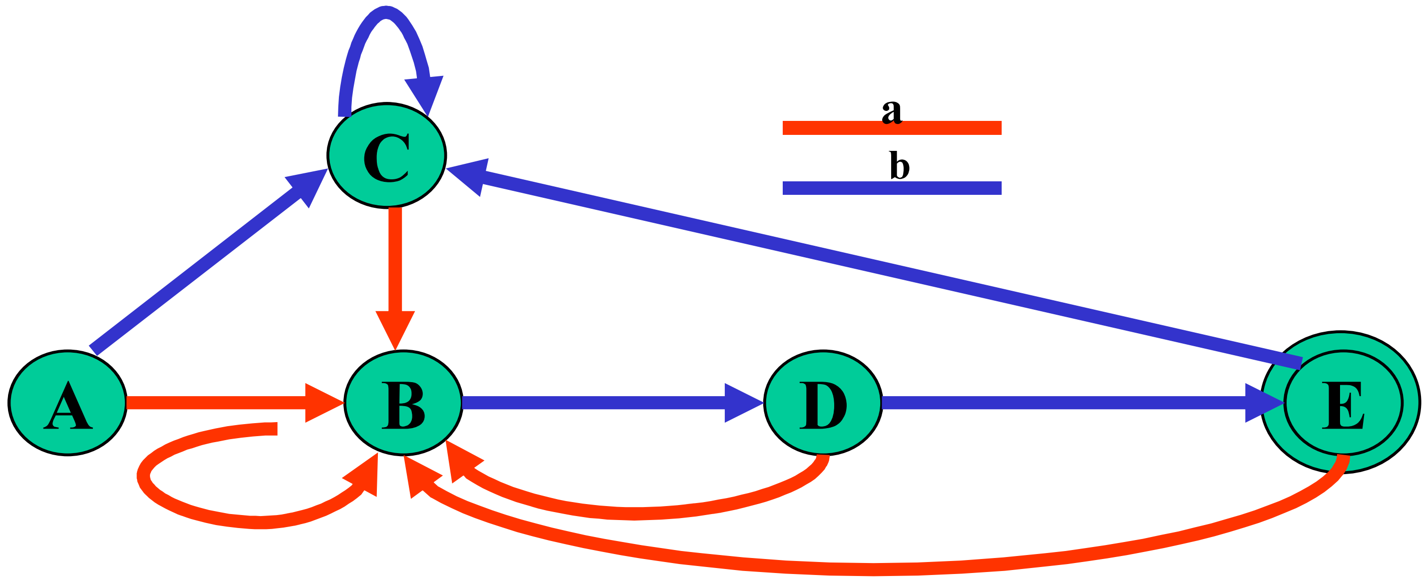
G1	G2	G3
ABC	D	E
a→ G1 b→ G1 G2 (AC) (B)	No Change	No change

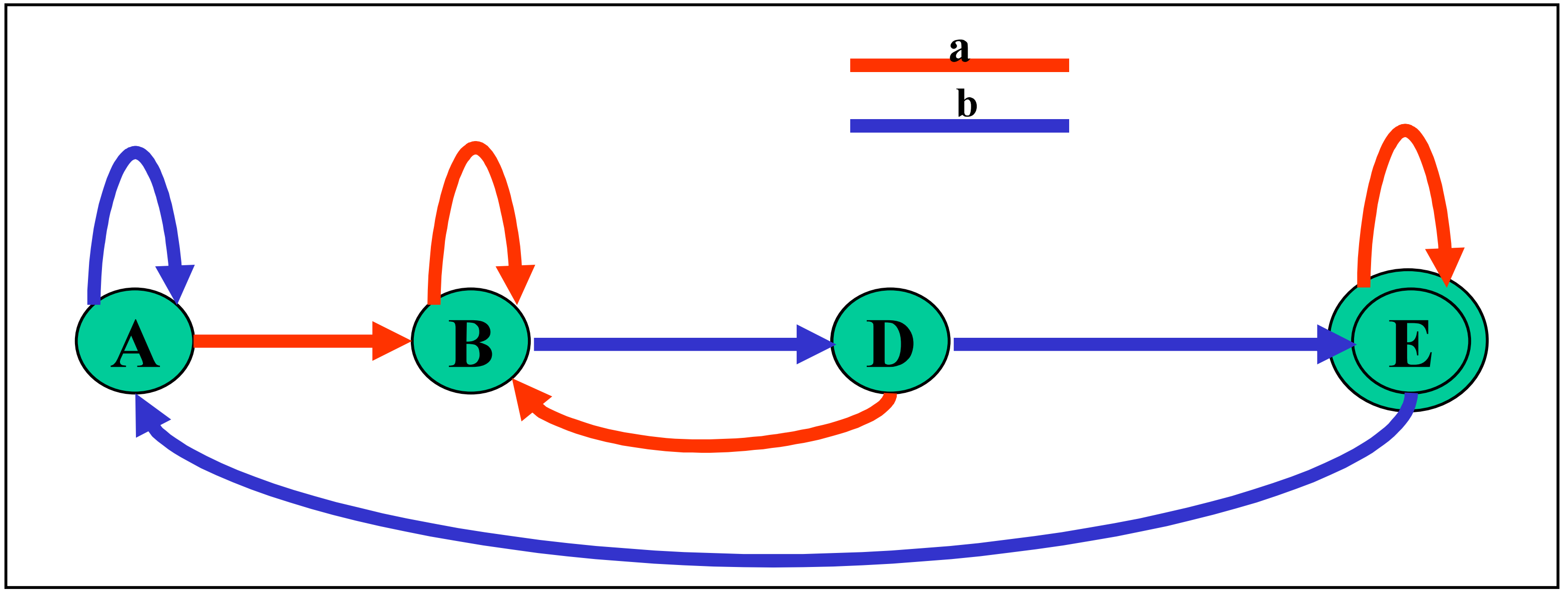
Example of minimization of DFA States



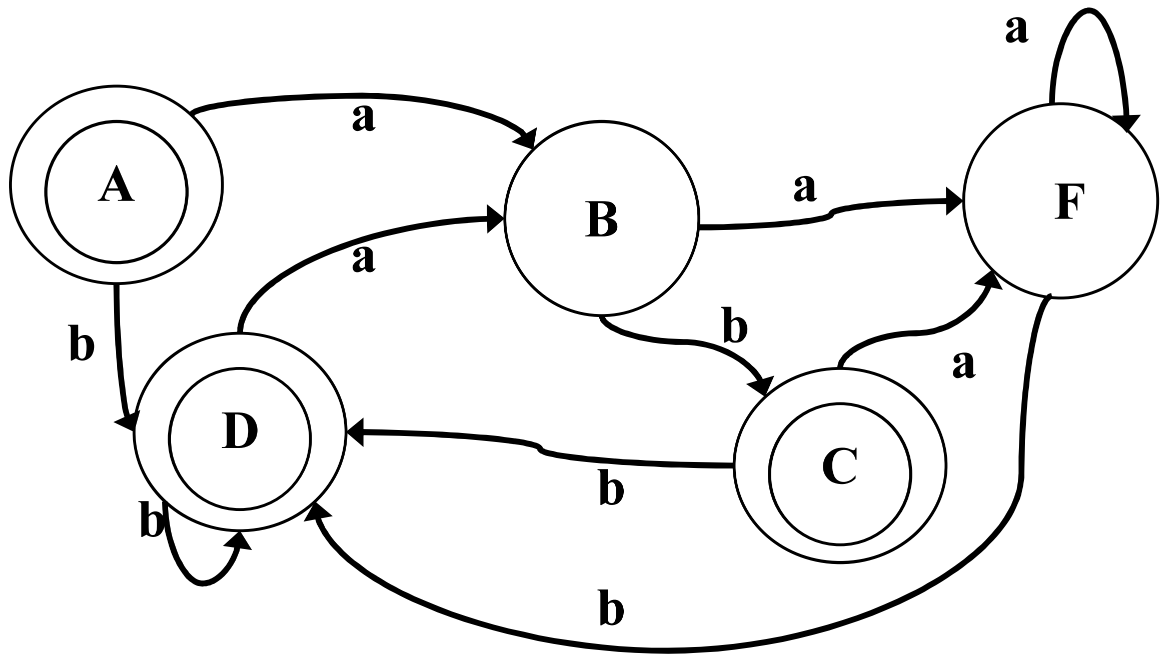
G1	G2	G3	G4
AC	B	D	E
No Change	No Change	No Change	No change

Example of minimization of DFA States





example



Minimized DFA: