Lexical Analysis
Part-2

Finite State Automata (FSAs)

- **AKA "Finite State Machines", "Finite Automata", "FA"**
- A *recognizer* for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language and "no" otherwise.
  - The regular expression is compiled into a recognizer by constructing a generalized transition diagram called a finite automaton.
- One start state
- Many final states
- Each state is labeled with a state name
- Directed edges, labeled with symbols
- Two types
  - Deterministic (DFA)
  - Non-deterministic (NFA)

Simulation of a Finite Automata (FA)

```
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

Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) is mathematical model that consists of

1. A set of states S
2. A set of input symbols Σ
3. A transition function that maps state/symbol pairs to a set of states
4. A special state s_0 called the start state
5. A set of states F (subset of S) of final states

INPUT: string

OUTPUT: yes or no

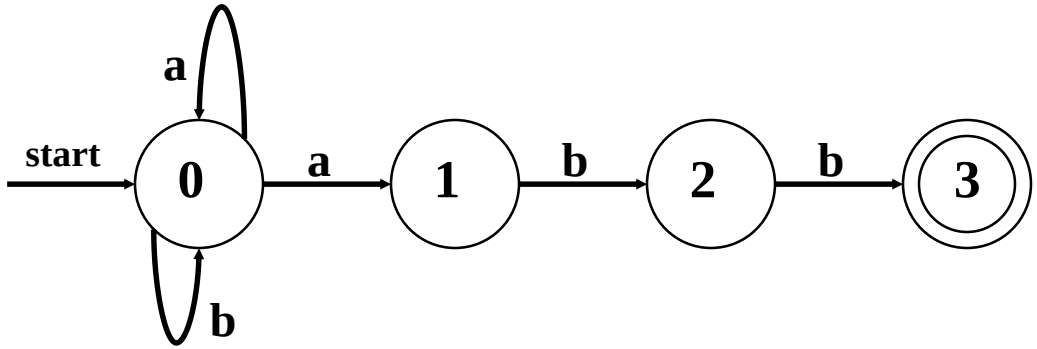
Example – NFA : (a|b)*abb

$S = \{ 0, 1, 2, 3 \}$

$s_0 = 0$

$F = \{ 3 \}$

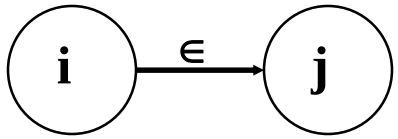
$\Sigma = \{ a, b \}$



state \ input	a	b
0	{ 0, 1 }	{ 0 }
1	--	{ 2 }
2	--	{ 3 }

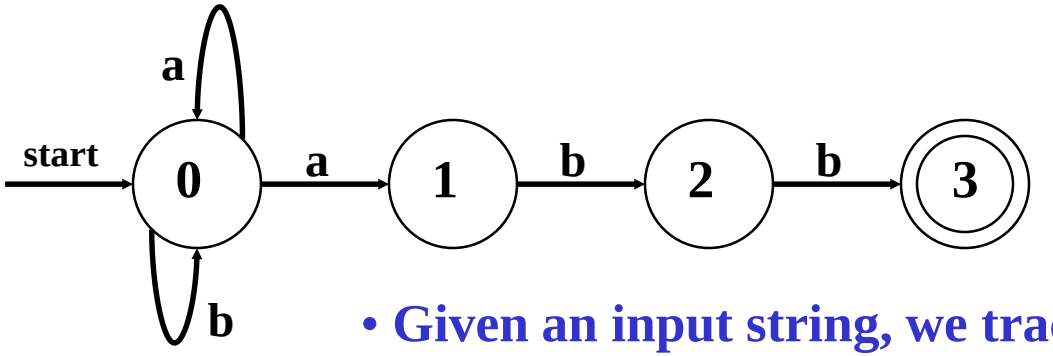
Transition Table

∈ (null) moves possible



Switch state but do not use any input symbol

How Does An NFA Work ?



- Given an input string, we trace moves
- If no more input & in final state, ACCEPT

EXAMPLE:

Input: ababb

move(0, a) = 1

move(1, b) = 2

move(2, a) = ? (undefined)

REJECT !

-OR-

move(0, a) = 0

move(0, b) = 0

move(0, a) = 1

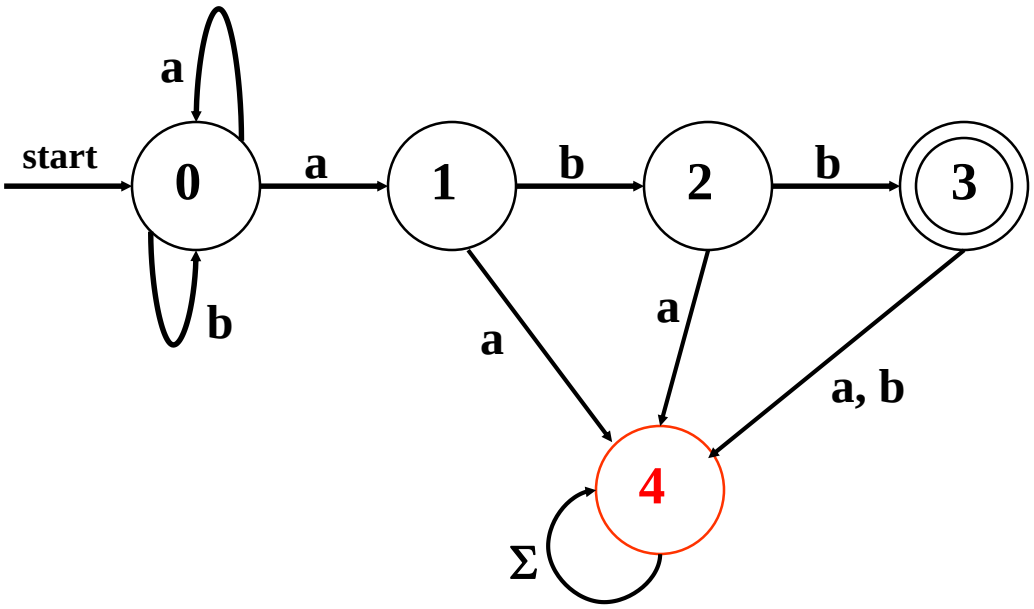
move(1, b) = 2

move(2, b) = 3

ACCEPT !

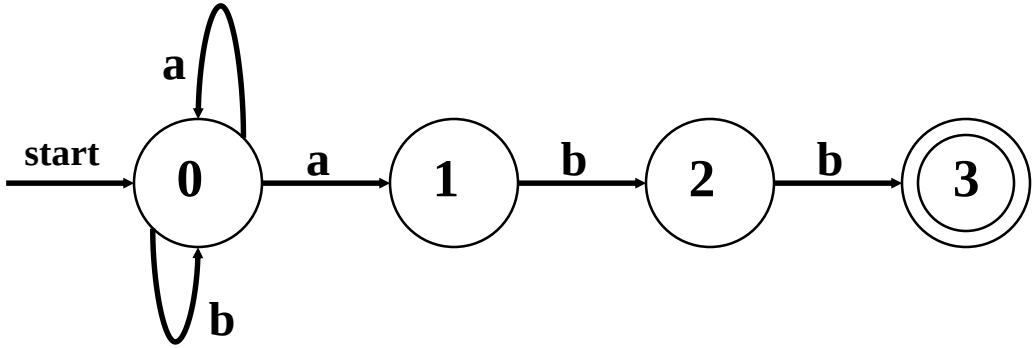
Handling Undefined Transitions

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



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 <u>not accepted</u> along the valid path:

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

Deterministic Finite Automata

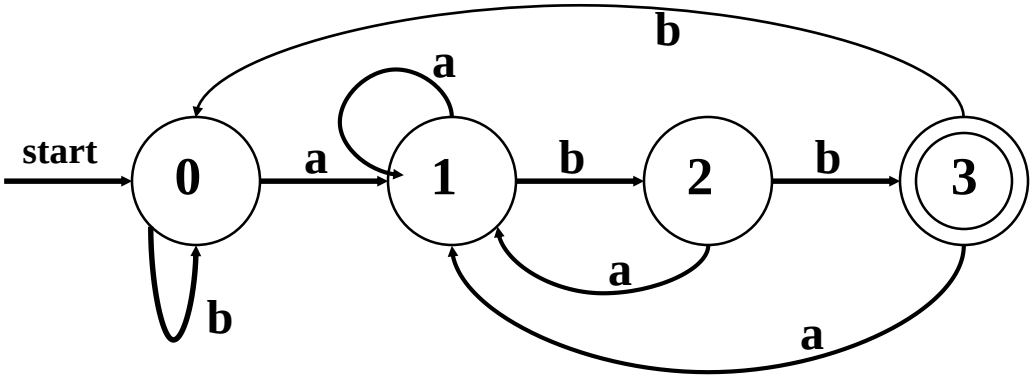
A DFA is an NFA with the following restrictions:

- $\in$ moves are not 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.

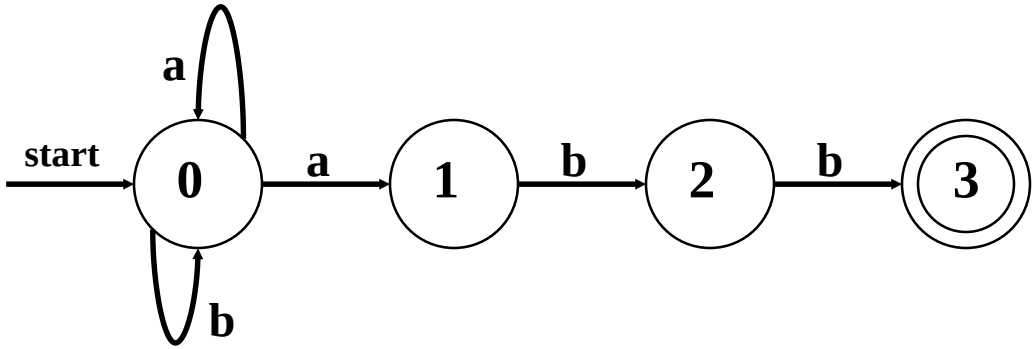
```
s ← s0
c ← nextchar;
while c ≠ eof do
  s ← move(s,c);
  c ← nextchar;
end;
if s is in F then return "yes"
             else return "no"
```

Example – DFA : (a|b)*abb



DFA

Recall the original NFA:



Relation between RE, NFA and DFA

1. There is an algorithm for converting any RE into an NFA.
2. There is an algorithm for converting any NFA to a DFA.
3. There is an algorithm for converting any DFA to a RE.

These facts tell us that REs, NFAs and DFAs have equivalent expressive power.

All three describe the class of regular languages.

NFA vs DFA

An NFA may be simulated by algorithm, when NFA is constructed from the R.E

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 uses it to recognize input

The space requirement of a DFA can be large. The RE (a+b)*a(a+b)(a+b)….(a+b) [n-1 (a+b) at the end] has no DFA with less than 2^n states. Fortunately, such RE in practice does not occur often.

	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.

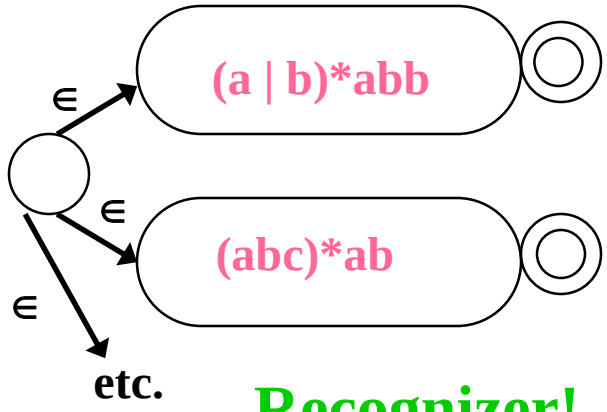
Lexical Analyzer

- **Designing Lexical Analyzer Generator**

  Reg. Expr. → NFA construction

  NFA → DFA conversion

  DFA simulation for lexical analyzer



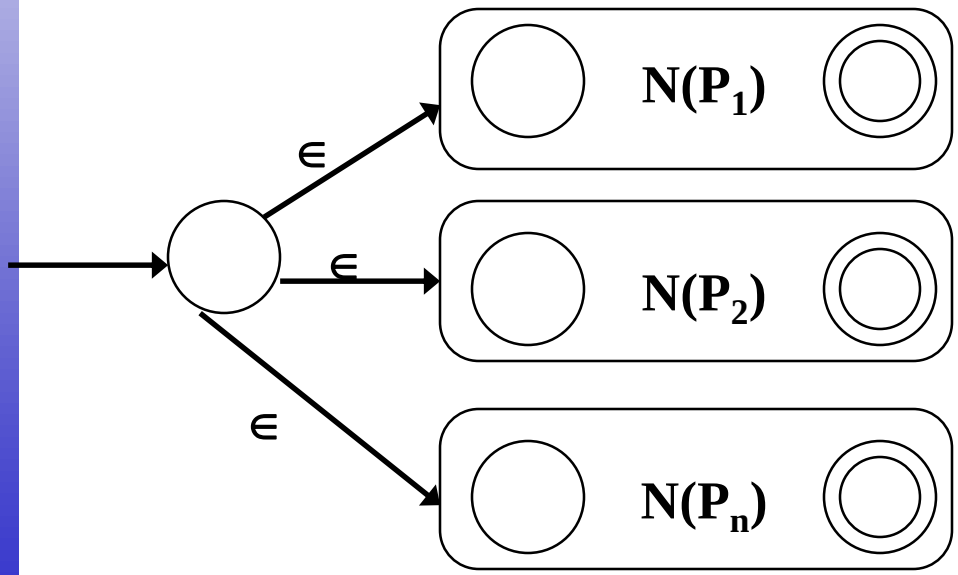
Recognizer!

- Each pattern recognizes lexemes. Longest prefix of input is matched. In case of tie the first pattern listed is chosen

- Each pattern described by regular expression

Lex Specification → Lexical Analyzer

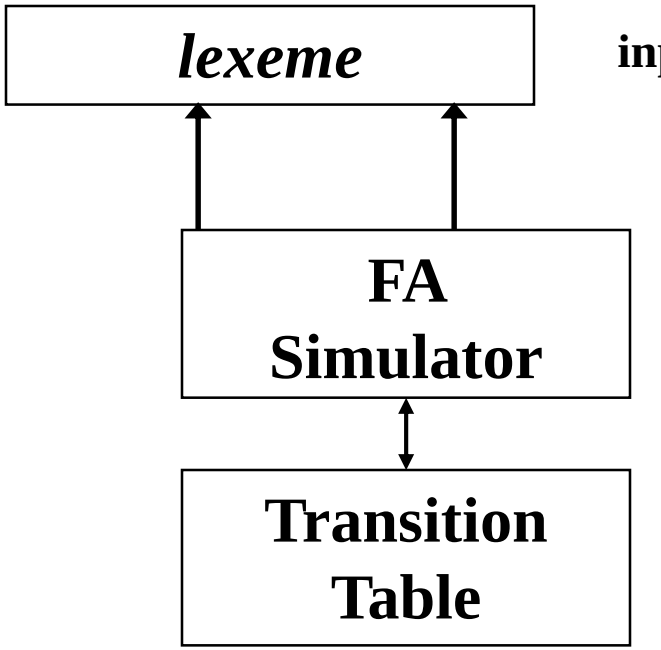
- **Let $P_1, P_2, \ldots, P_n$ be Lex patterns**
  (regular expressions for valid tokens in prog. lang.)
- **Construct $N(P_1), N(P_2), \ldots N(P_n)$**
- **Note: accepting state of $N(P_i)$ will be marked by P_i**
- **Construct NFA:**



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

Pictorially

(a) Lex Compiler



(b) Schematic lexical analyzer

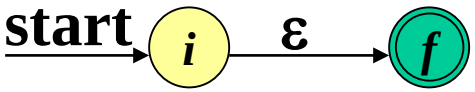
Converting Regular Expressions to NFAs

- How do we define an NFA that accepts a regular expression?
- It is very simple. Remember that a regular expression is formed by the use of alternation, concatenation, and repetition.
- Thus all we need to do is to know how to build the NFA for a single symbol, and how to compose NFAs.

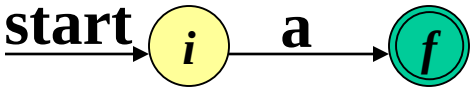
Converting Regular Expressions to NFAs

Thompson's Construction

- Empty string ε is a regular expression denoting { ε }



- *a* is a regular expression denoting {*a*} for any *a* in Σ

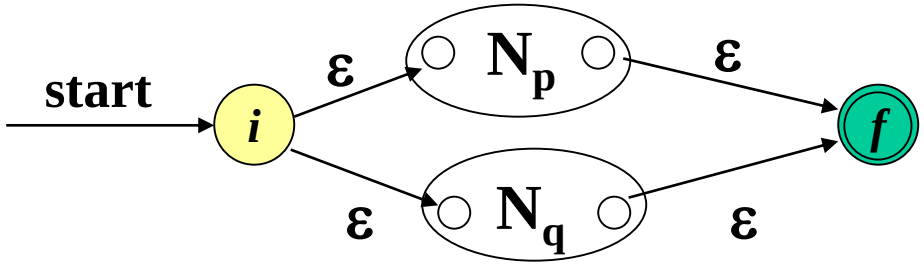


Converting Regular Expressions to NFAs

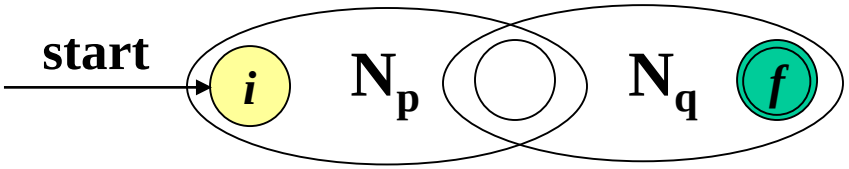
Composing NFAs with Alternation and Concatenation

If P and Q are regular expressions with NFAs N_p, N_q:

P | Q (alternation)



PQ (concatenation)

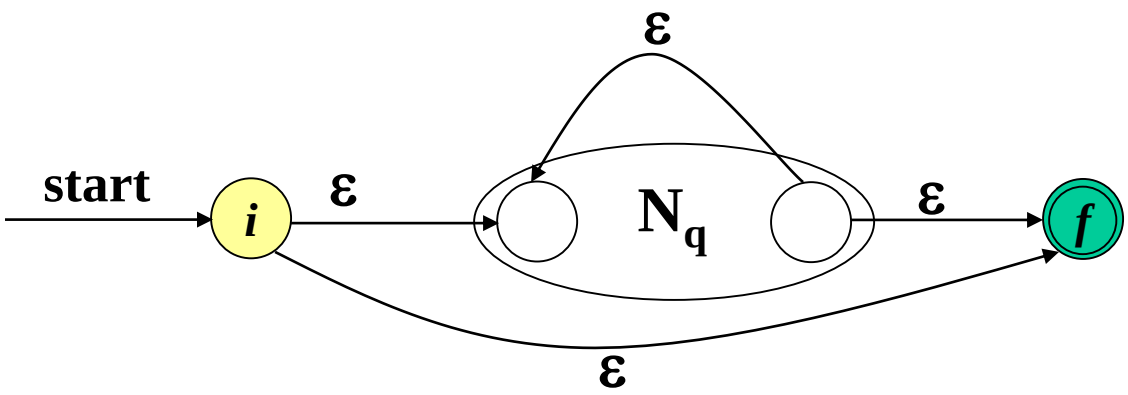


Converting Regular Expressions to NFAs

Composing NFAs with Repeation

If Q is a regular expression with NFA N_q:

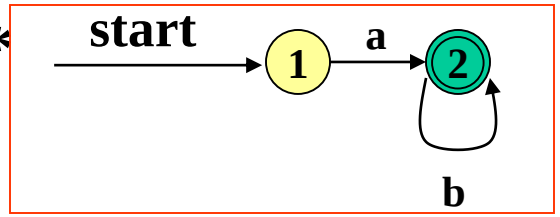
Q* (closure)



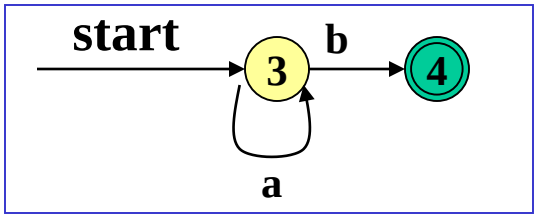
Example (ab* | a*b)*

Starting with:

ab*

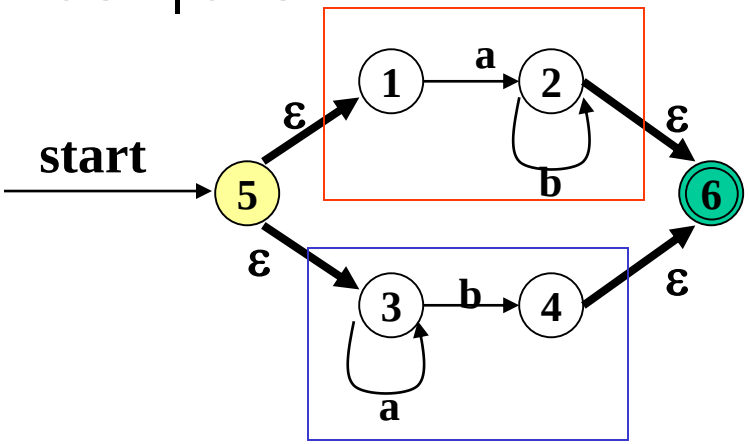


a*b



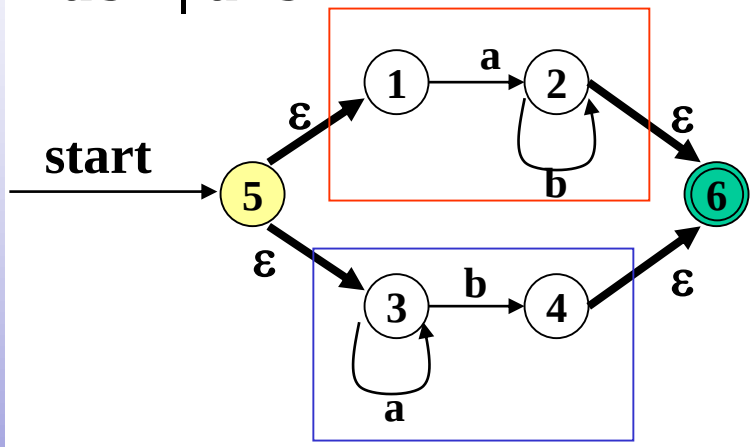
---

ab* | a*b

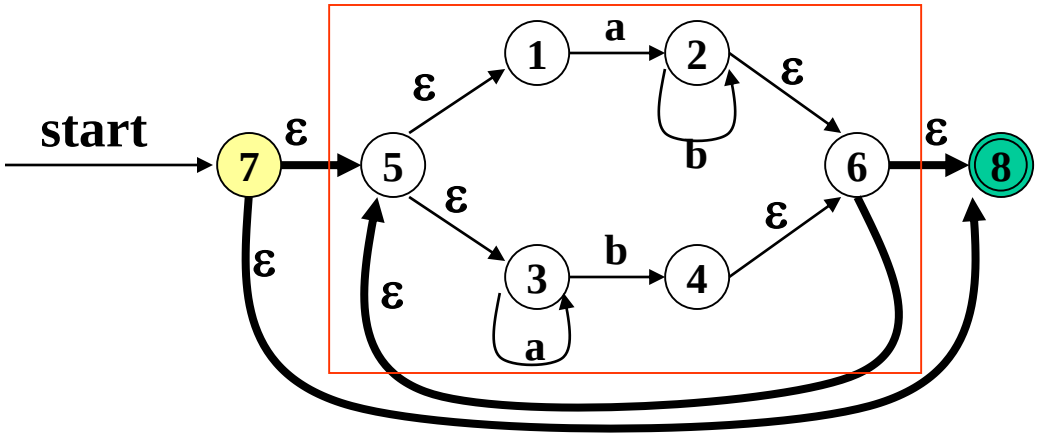


Example (ab* | a*b)*

ab* | a*b



(ab* | a*b)*



Properties of Construction

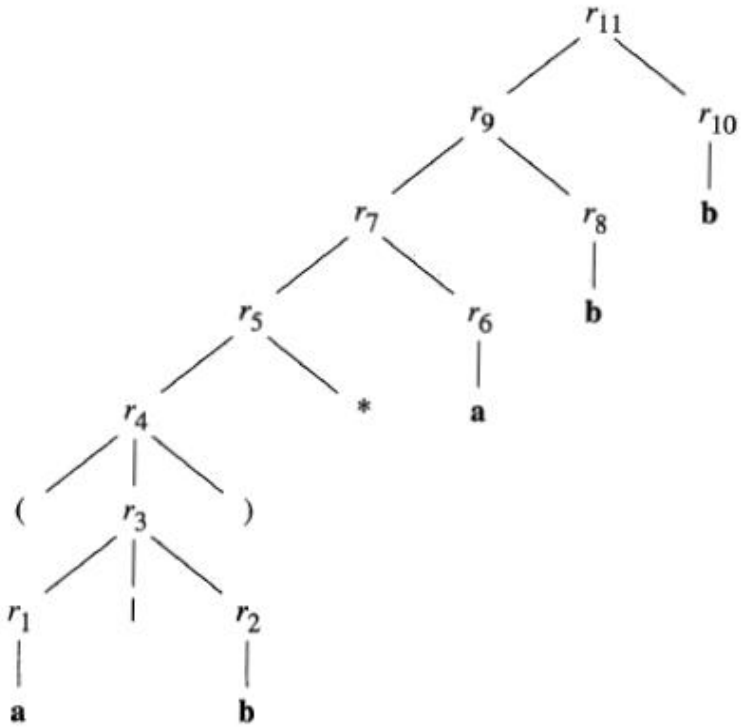
Let r be a regular expression, with NFA N(r), then:

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 $\in$-transition**

Detailed Example

(a|b)*abb

Parse Tree for this regular expression:

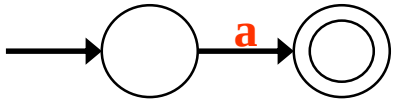


What is the NFA? Let's construct it !

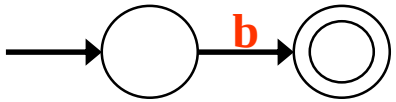
Detailed Example – Construction(1)

(a|b)*abb

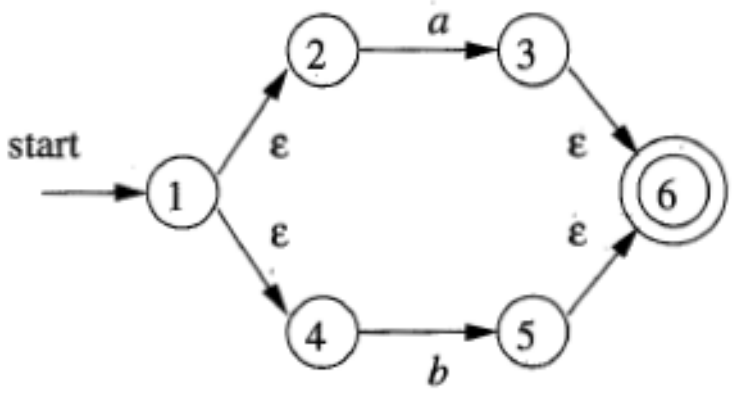
r_1:



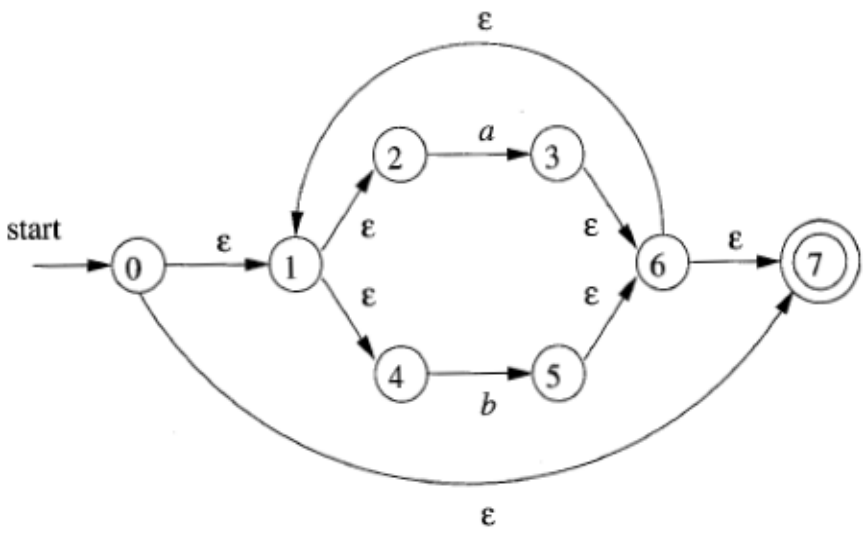
r_2:



r_3:r1|r2



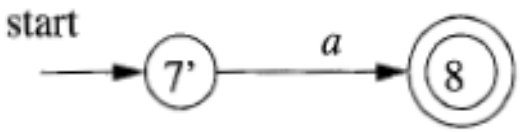
r_4:(r3)*



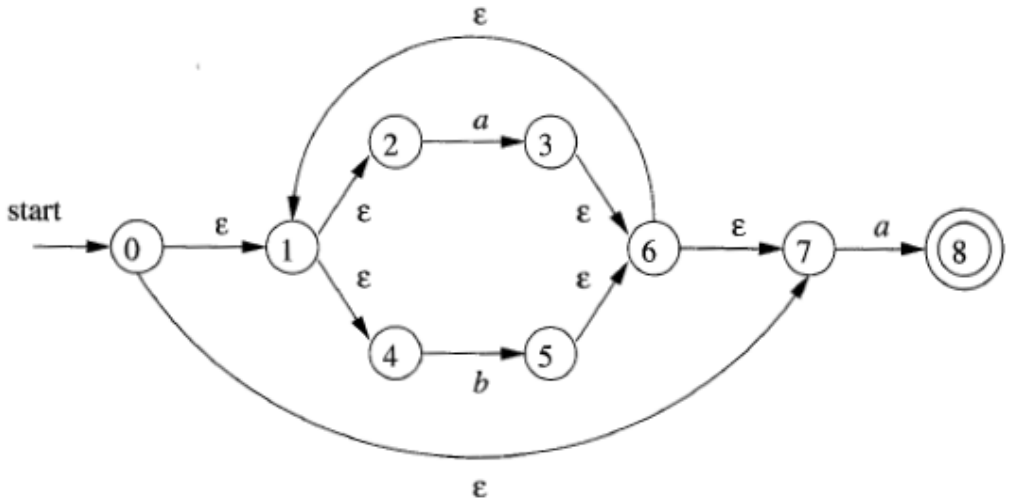
Detailed Example – Construction(2)

(a|b)*abb

r_5



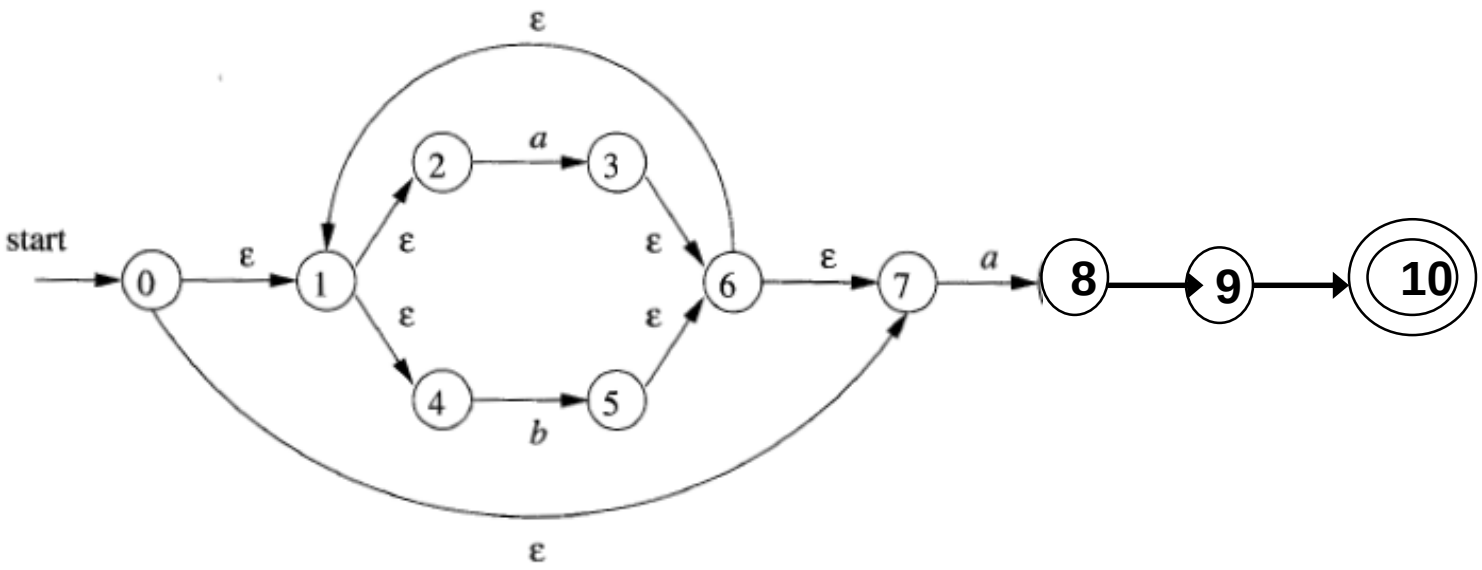
r_6: r4 r5



Detailed Example – Final Step

(a|b)*abb

r_{10}

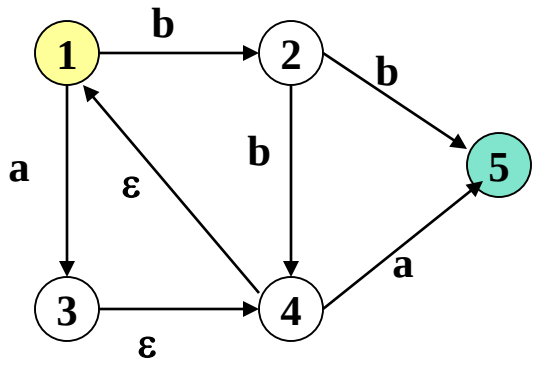


Converting NFAs to DFAs (subset construction)

- **Idea**: Each state in the new DFA will correspond to some set of states from the NFA. The DFA will be in state $\{s_0,s_1,\ldots\}$ after input if the NFA could be in *any* of these states for the same input.
- **Input**: NFA N with state set S_N, alphabet Σ, start state s_N, final states F_N, transition function T_N: $S_N \times \{\Sigma \cup \varepsilon\} \rightarrow S_N$
- **Output**: DFA D with state set S_D, alphabet Σ, start state $s_D = \varepsilon\text{-closure}(s_N)$, final states F_D, transition function T_D: $S_D \times \Sigma \rightarrow S_D$

Terminology: ε-closure

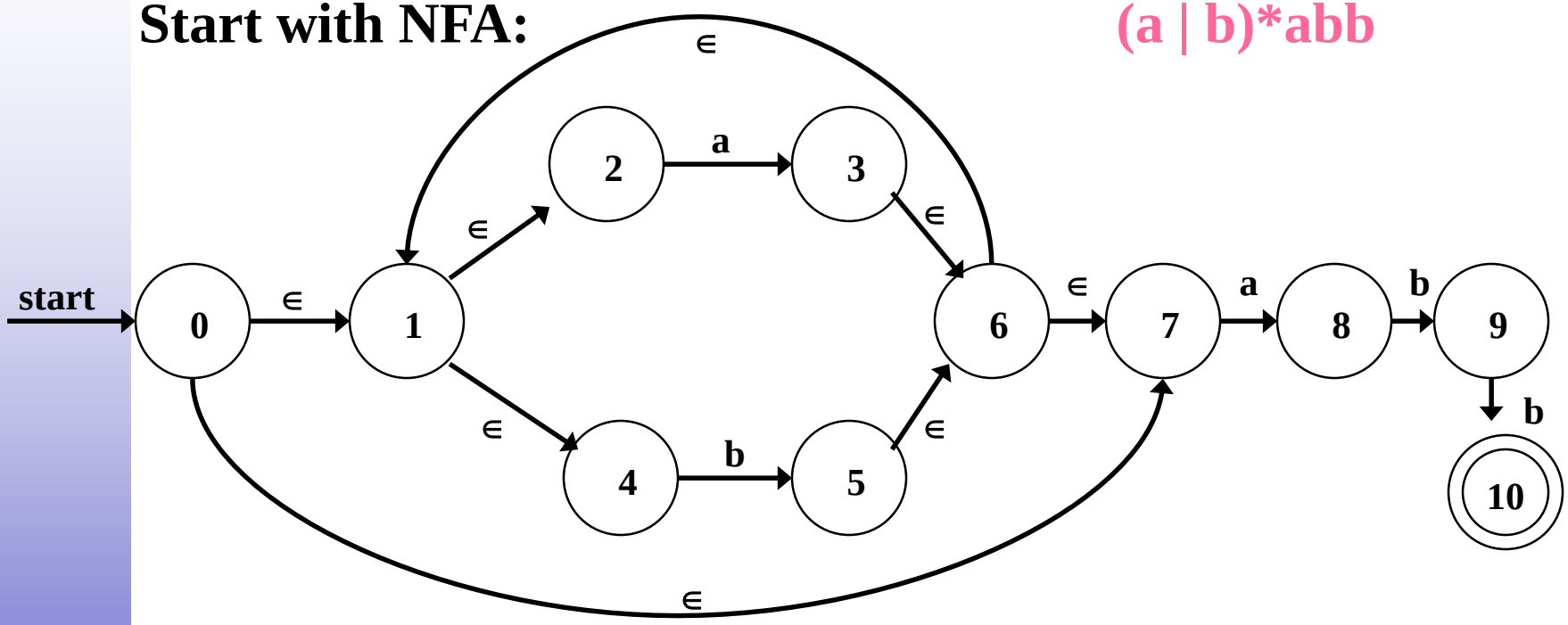
ε-closure(T) = T + all NFA states reachable from any state in T using only ε transitions.



ε-closure({1,2,5}) = {1,2,5}
ε-closure({4}) = {1,4}
ε-closure({3}) = {1,3,4}
ε-closure({3,5}) = {1,3,4,5}

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)

5^{th} , 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.

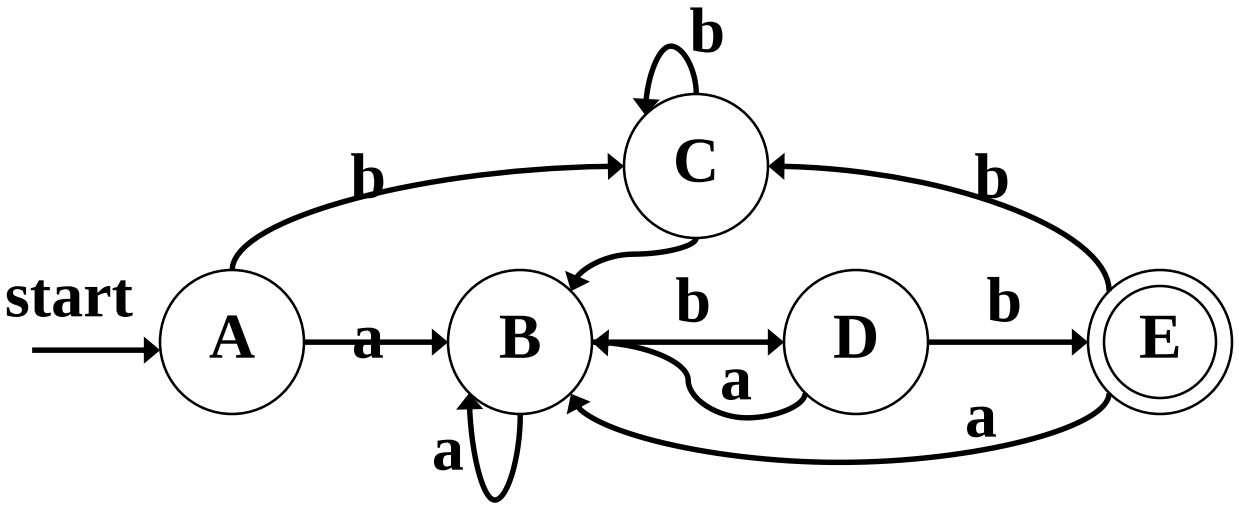
b : ∈-closure(*move*(*E*,b)) = ∈-closure(*move*({1,2,4,5,6,7,10},b))}
= {1,2,4,5,6,7} = C

Define Dtran[E,b] = C.

Conversion Example – continued (4)

This gives the transition table Dtran for the DFA of:

Dstates	Input Symbol	
	a	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, $\in$-closure(s_0) 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 := $\in$-closure(*move*(T,*a*));

        if U is not in Dstates **then**

            add U as an unmarked state to Dstates;

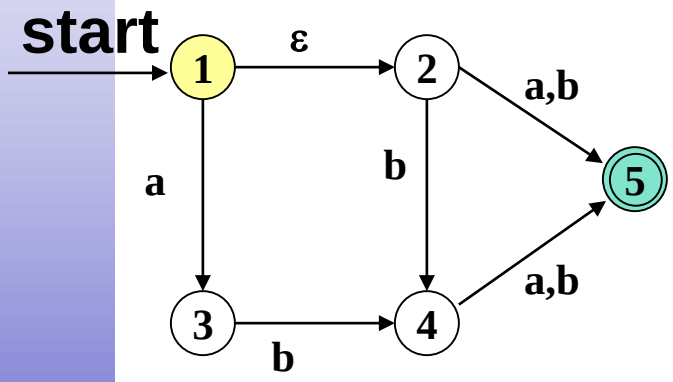
        Dtran[T,*a*] := U

    end

end

Example 2: Subset Construction

NFA



NFA N with

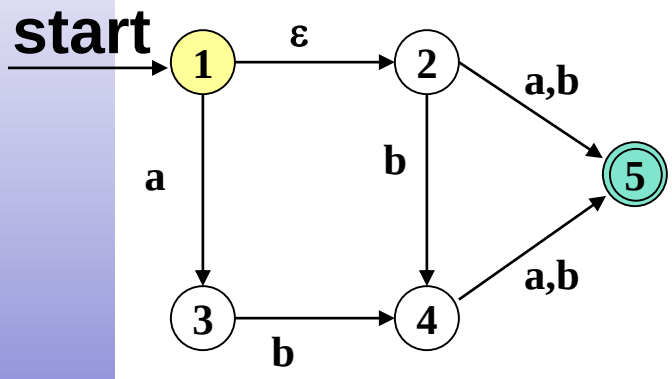
- **State set S_N = {1,2,3,4,5},**
- **Alphabet Σ = {a,b}**
- **Start state s_N=1,**
- **Final states F_N={5},**
- **Transition function**

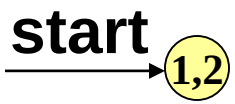
$$T_N: S_N \times \{\Sigma \cup \varepsilon\} \rightarrow S_N$$

	a	b	ε
1	3	-	2
2	5	5, 4	-
3	-	4	-
4	5	5	-
5	-	-	-

Example 2: Subset Construction

NFA

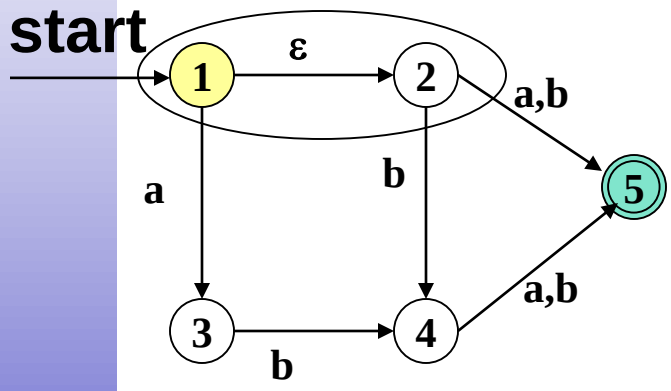


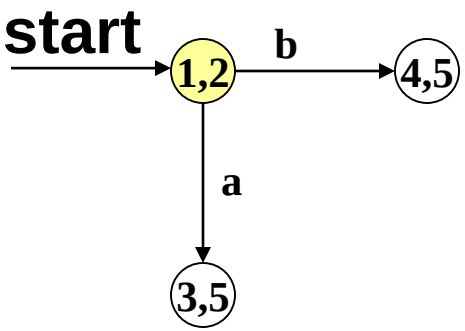


T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}		

Example 2: Subset Construction

NFA

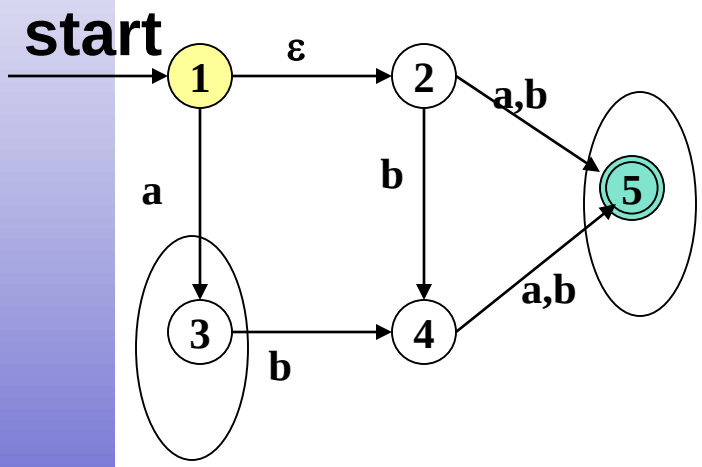


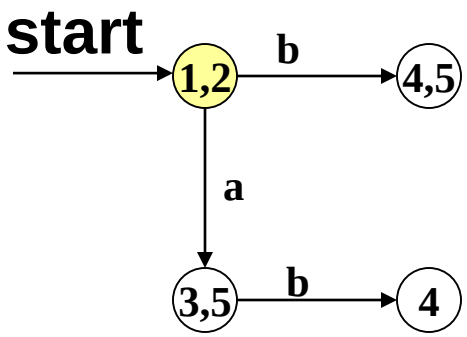


T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}		
{4,5}		

Example 2: Subset Construction

NFA

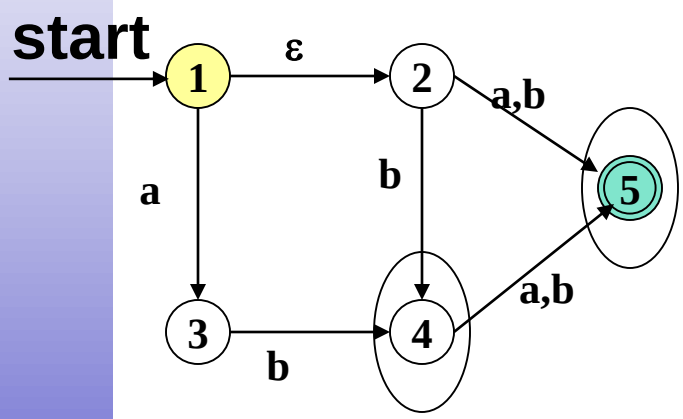


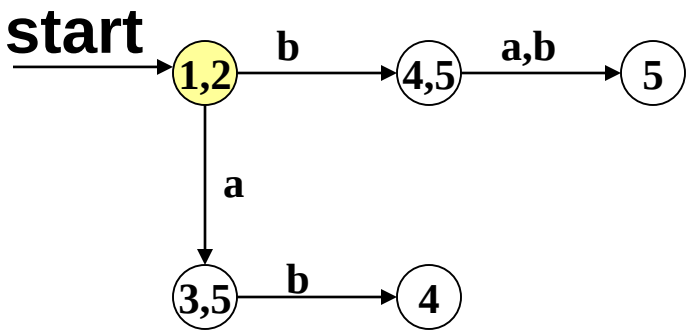


T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}		
{4}		

Example 2: Subset Construction

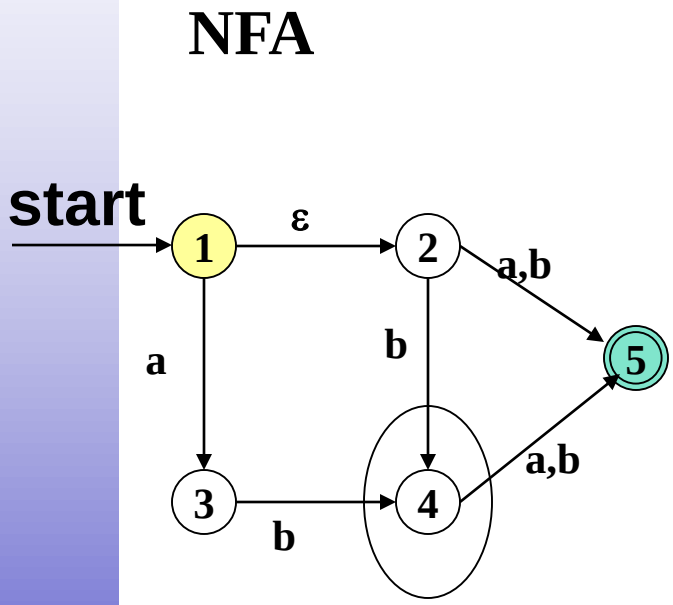
NFA

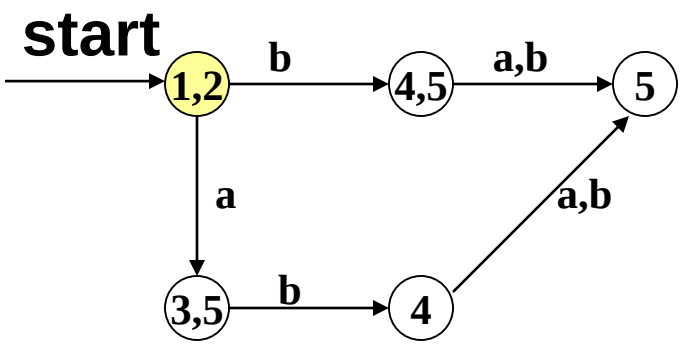




T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}		
{5}		

Example 2: Subset Construction

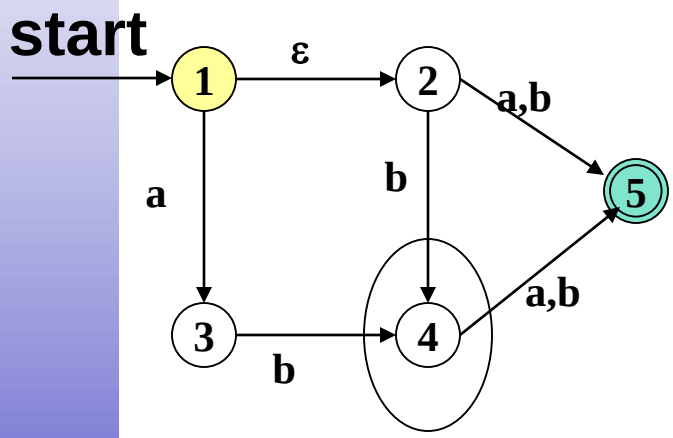


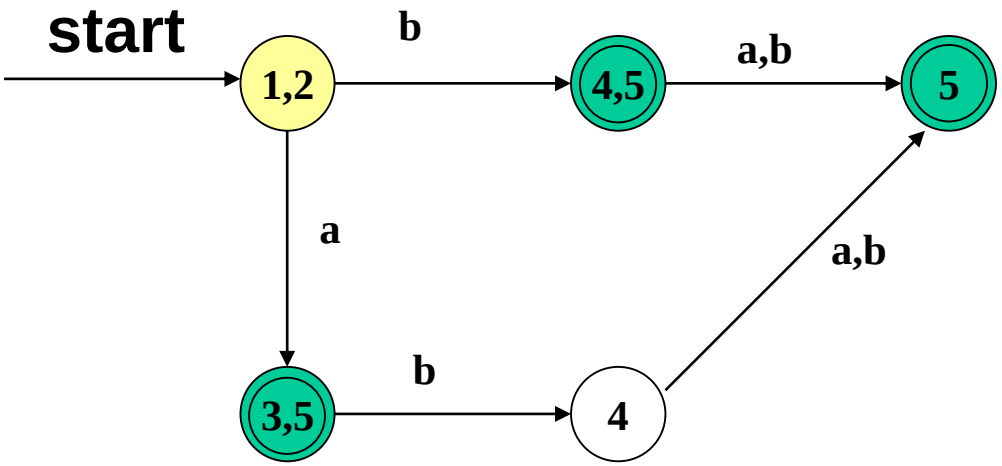


T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

Example 2: Subset Construction

NFA



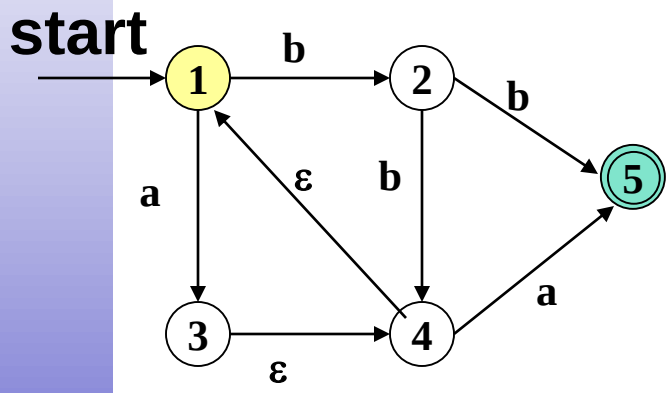


All final states since the NFA final state is included

T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

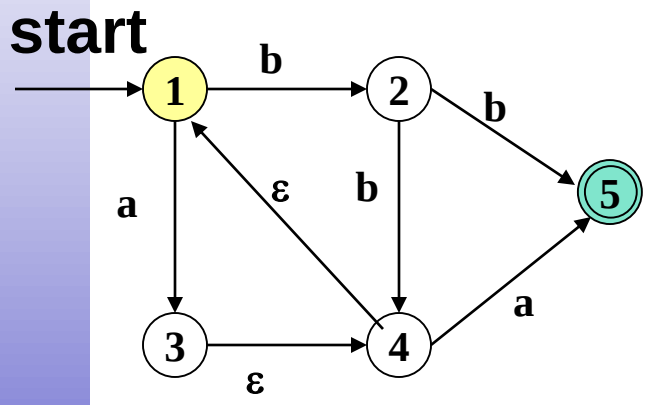
Example 3: Subset Construction

NFA

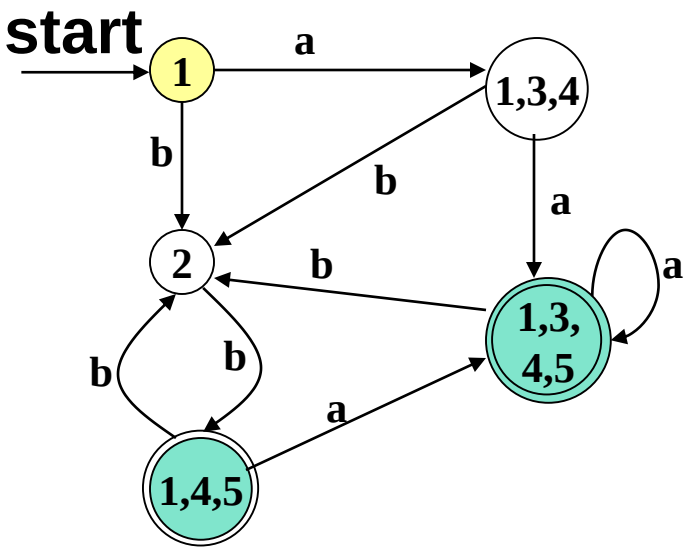


Example 3: Subset Construction

NFA

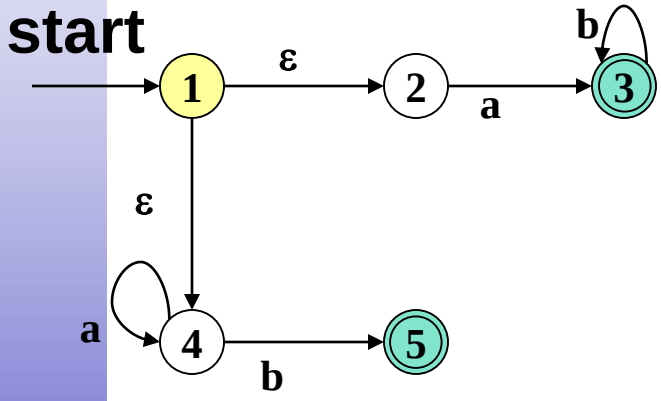


DFA

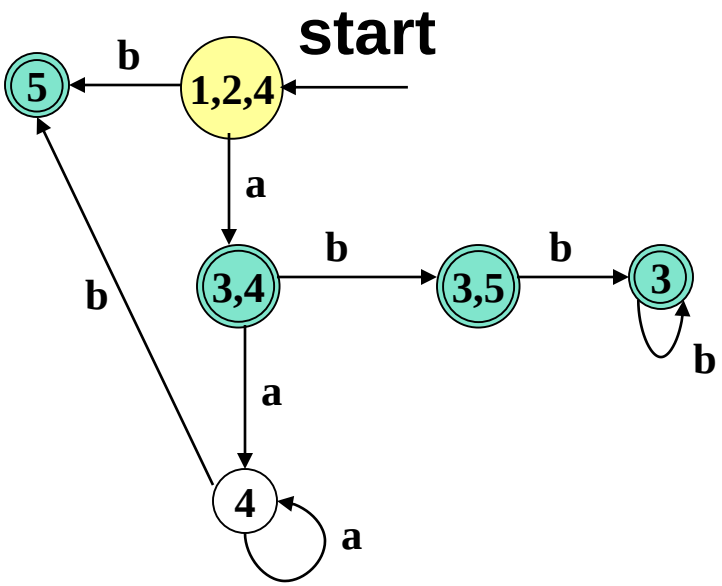


Example 4: Subset Construction

NFA



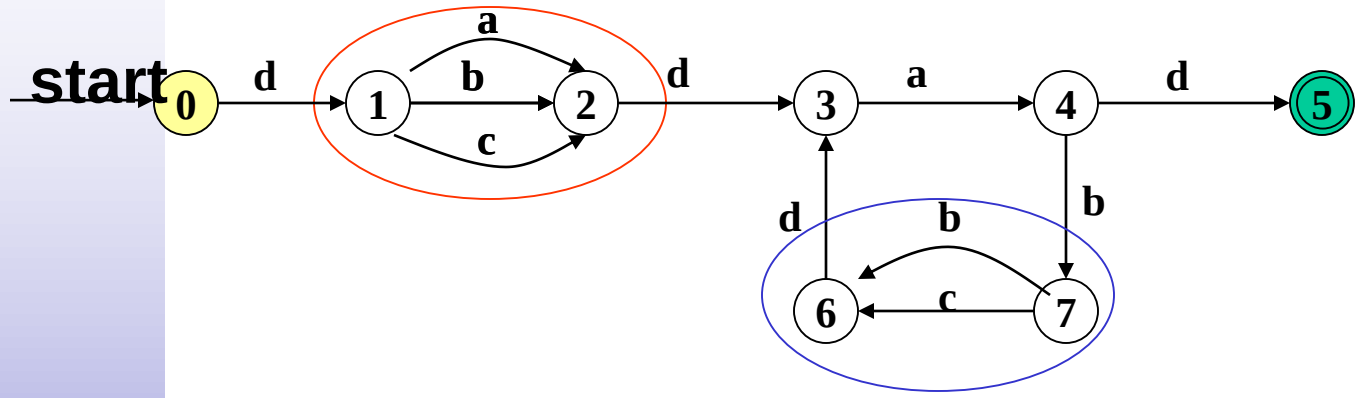
DFA



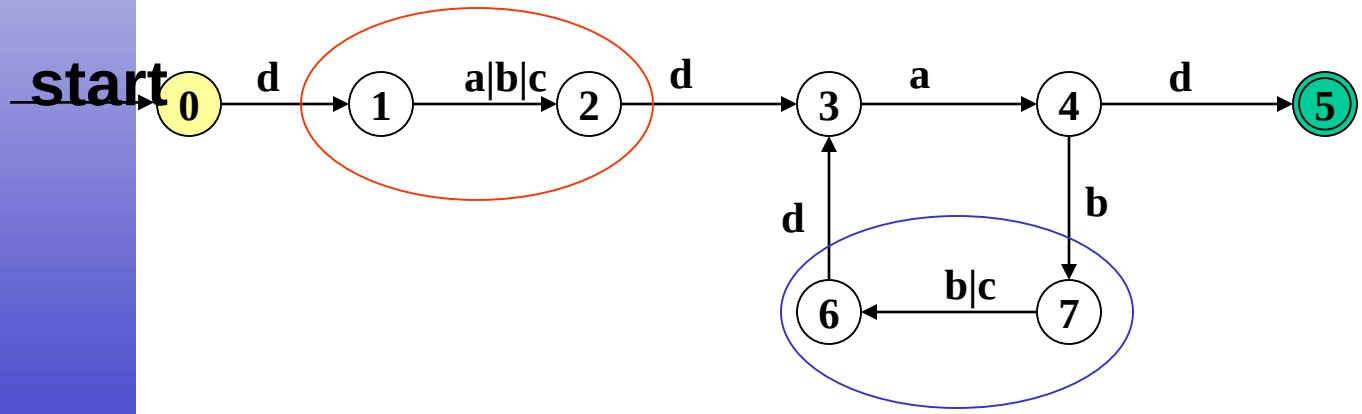
Converting DFAs to REs

1. Combine serial links by concatenation
2. Combine parallel links by alternation
3. Remove self-loops by Kleene closure
4. Select a node (other than initial or final) for removal. Replace it with a set of equivalent links whose path expressions correspond to the in and out links
5. Repeat steps 1-4 until the graph consists of a single link between the entry and exit nodes.

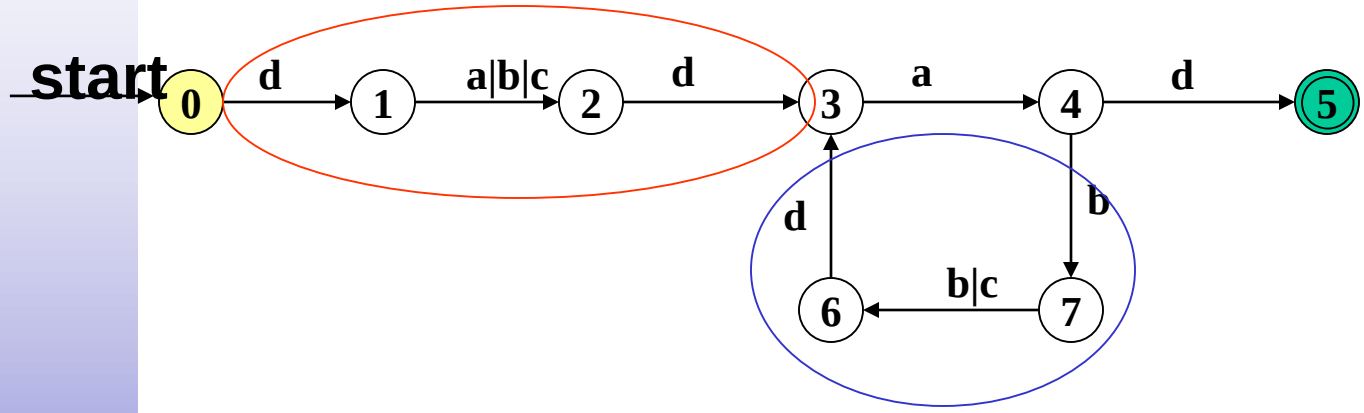
Example



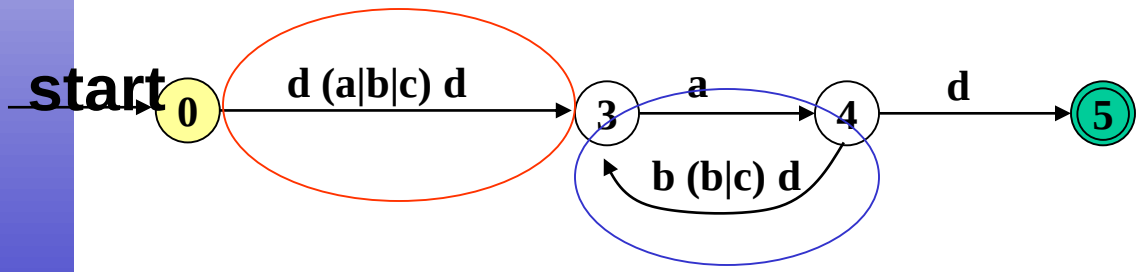
parallel edges become alternation



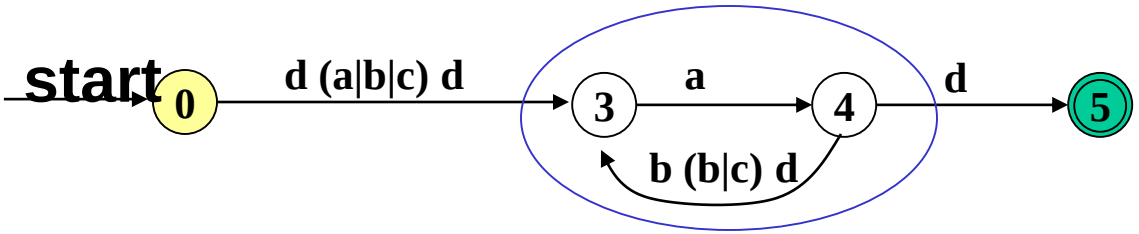
Example



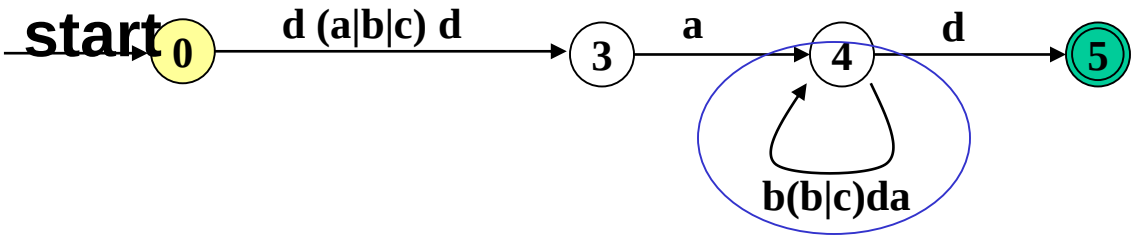
serial edges become concatenation



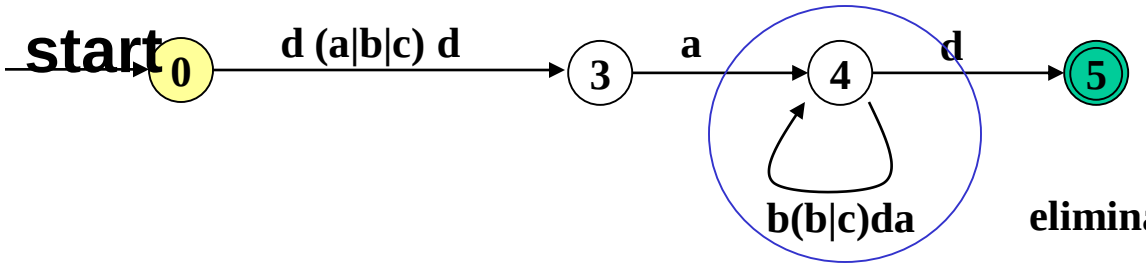
Example



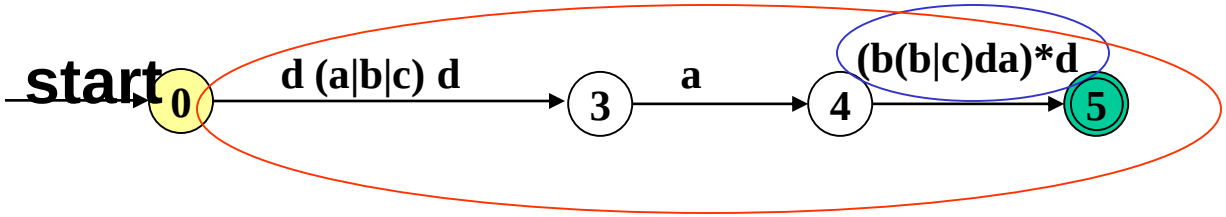
Find paths that can be “shortened”



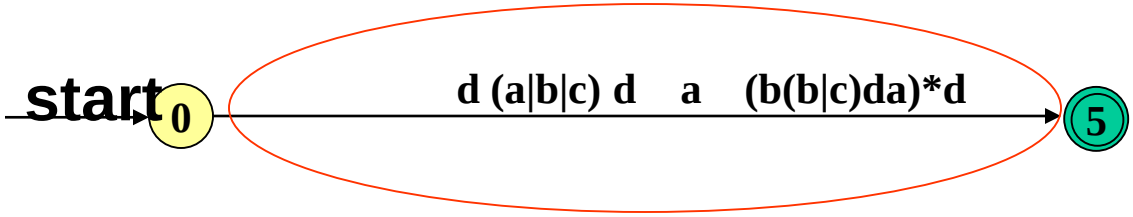
Example



eliminate self-loops



serial edges become concatenation



Relationship among RE, NFA, DFA

- The set of strings recognized by an NFA can be described by a Regular Expression.
- The set of strings described by a Regular Expression can be recognized by an NFA.
- The set of strings recognized by an DFA can be described by a Regular Expression.
- The set of strings described by a Regular Expression can be recognized by an DFA.
- DFAs, NFAs, and Regular Expressions all have the same “power”. They describe “Regular Sets” (“Regular Languages”)
- The DFA may have a lot more states than the NFA.

Reading Materials

- Chapter -3 of your Text book:
  - Compilers: Principles, Techniques, and Tools
- https://www.geeksforgeeks.org/compiler-design-tutorials/

End of slide