FINITE AUTOMATA

Finite Automata is one of the mathematical models that consist of a number of states and edges. It is a transition diagram that recognizes a regular expression or grammar.

Types of Finite Automata

There are tow types of Finite Automata :

- Non-deterministic Finite Automata (NFA)
- Deterministic Finite Automata (DFA)

Non-deterministic Finite Automata

NFA is a mathematical model that consists of five tuples denoted by
$M = \{Q_n, \Sigma, \delta, q_0, f_n\}$
Q_n – finite set of states
Σ – finite set of input symbols
δ – transition function that maps state-symbol pairs to set of states
q_0 – starting state
f_n – final state

Deterministic Finite Automata

DFA is a special case of a NFA in which
i) no state has an ε-transition.
ii) there is at most one transition from each state on any input.

DFA has five tuples denoted by
$M = \{Q_d, \Sigma, \delta, q_0, f_d\}$
Q_d – finite set of states
Σ – finite set of input symbols
δ – transition function that maps state-symbol pairs to set of states
q_0 – starting state
f_d – final state

Construction of DFA from regular expression

The following steps are involved in the construction of DFA from regular expression:
i) Convert RE to NFA using Thomson's rules
ii) Convert NFA to DFA
iii) Construct minimized DFA

B.S. Abdur Rahman ™ Crescent Institute of Science & Technology
Deemed to be University u/s 3 of the UGC Act, 1956

CSD 3202 - Compiler Design
III YEAR CSE

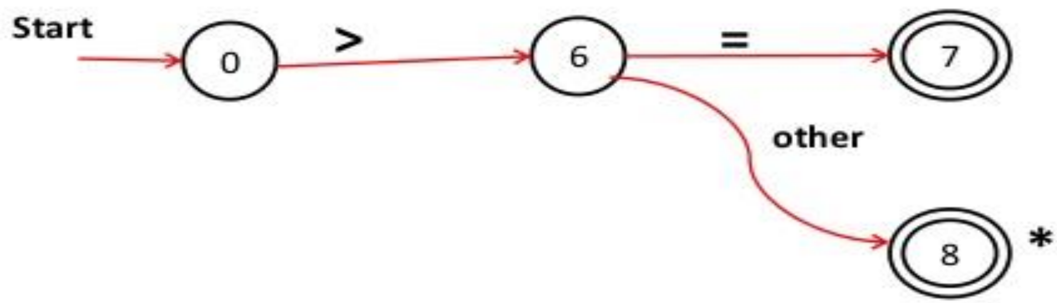
Transition diagrams

- Transition diagrams are also called finite automata.
- We have a collection of STATES drawn as nodes in a graph.
- TRANSITIONS between states are represented by directed edges in the graph.
- Each transition leaving a state s is labeled with a set of input characters that can occur after state s.

Transition diagrams (Conti…)

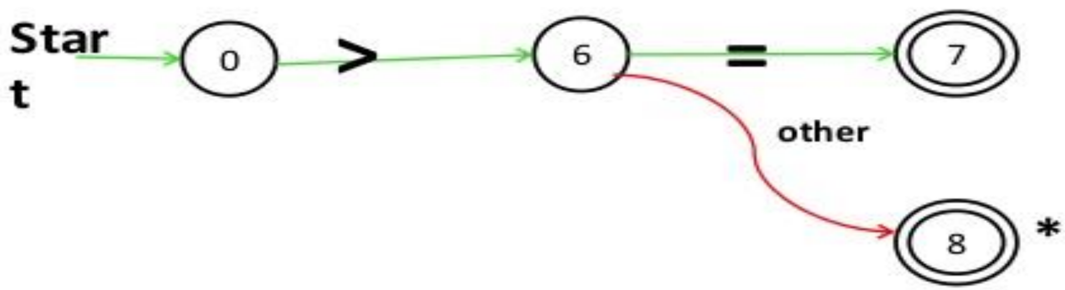
- For now, the transitions must be DETERMINISTIC.
- Each transition diagram has a single START state and a set of TERMINAL STATES.
- The label OTHER on an edge indicates all possible inputs not handled by the other transitions.
- Usually, when we recognize OTHER, we need to put it back in the source stream since it is part of the next token. This action is denoted with a * next to the corresponding state.

Relational operator (1/3)

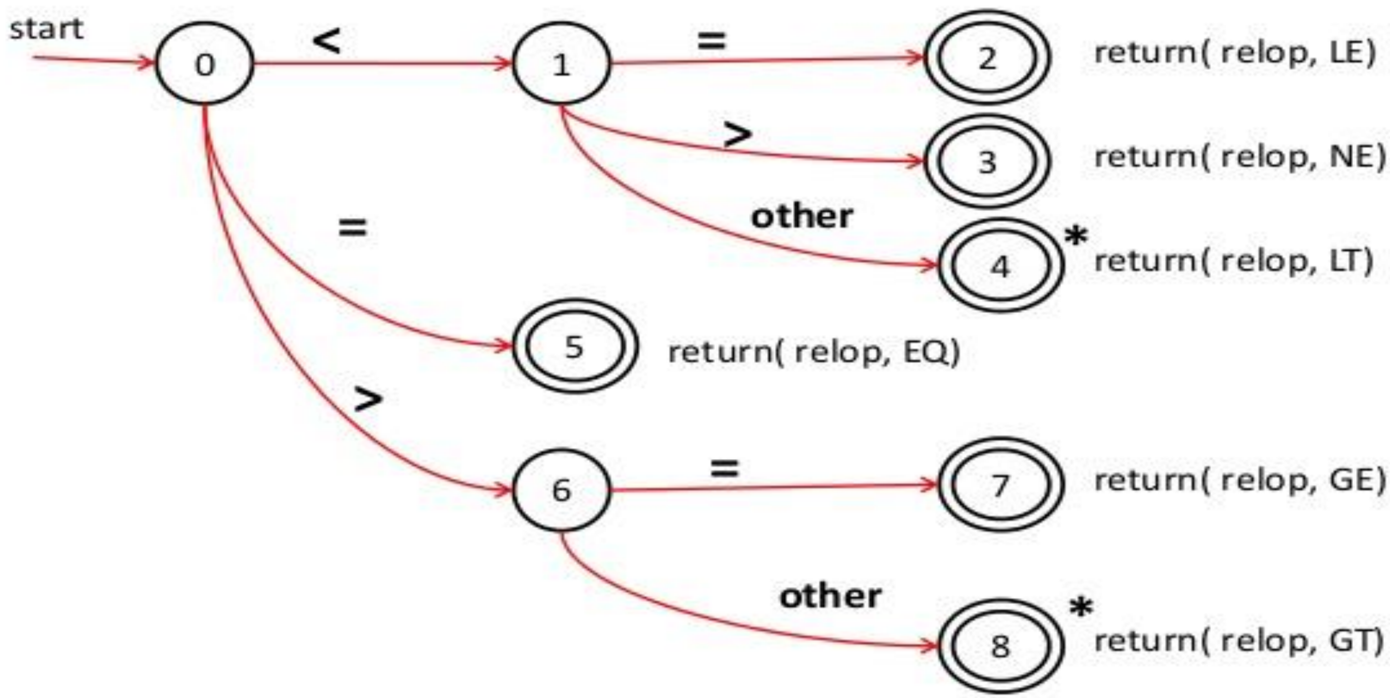


Relational operator (2/3)

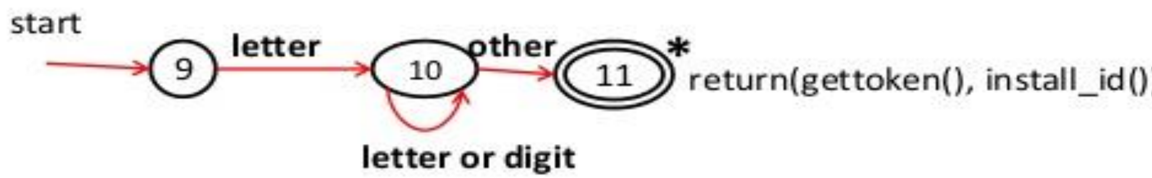
For eg: >=

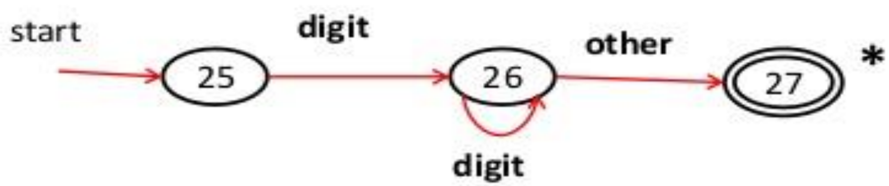


Relational operator (3/3)



Identifier and number





Automatic Generation of Lexical Analysers

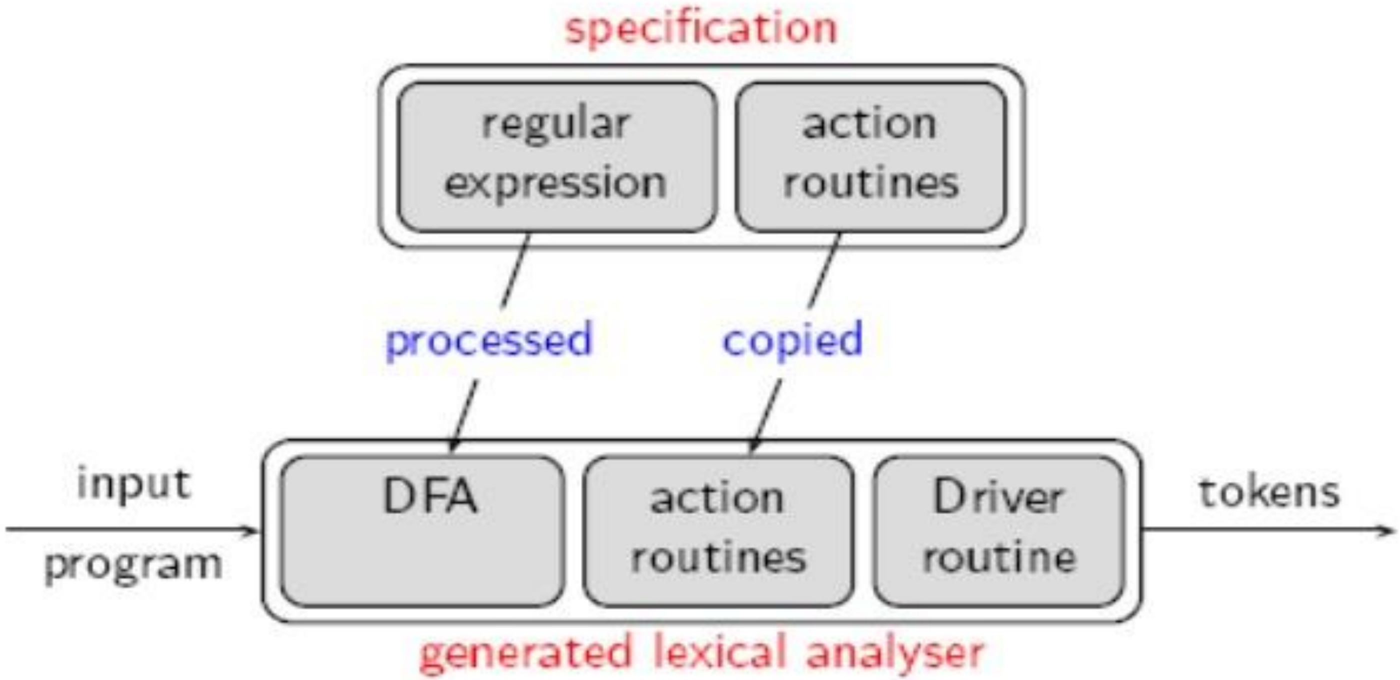
Inputs to the lexical analyser generator:

- A specification of the tokens of the source language, consisting of:
  - a regular expression describing each token, and
  - a code fragment describing the action to be performed, on identifying each token.

The generated lexical analyser consists of:

- A *deterministic finite automaton (DFA)* constructed from the token specification.
- A *code fragment* (a driver routine) which can traverse *any DFA*.
- Code for the *action specifications*.

Automatic Generation of Lexical Analysers

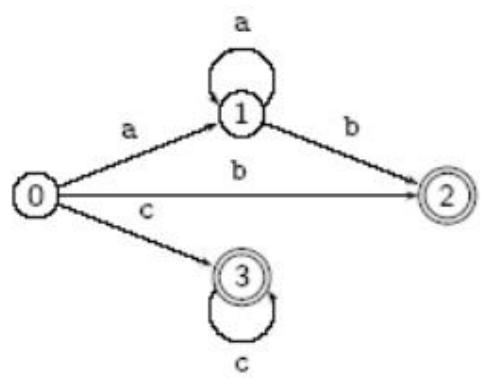


Example of Lexical Analyser Generation

Suppose a language has two tokens

Pattern	Action
`a*b`	`{ printf( "Token 1 found");}`
`c+`	`{ printf( "Token 2 found");}`

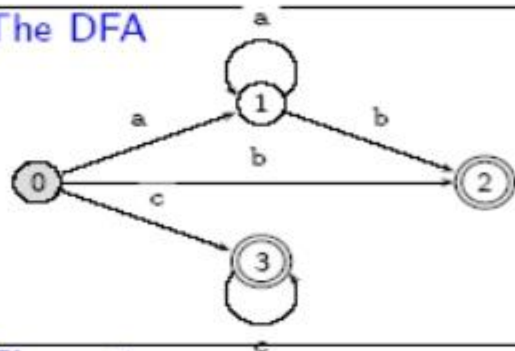
From the description, construct a structure called a deterministic finite automaton (DFA).



Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

Input: aabadbcc↩

Output:

Example of Lexical Analyser Generation

In summary:

- The DFA, the driver routine and the action routines taken together, constitute the lexical analyser.
- - actions are supplied as part of specification.
  - driver routine is common to all generated lexical analyzers

  The only issue – how are the patterns, specified by regular expressions, converted to a DFA.

  In two steps:

  - Convert regular expression into NFA.
  - Convert NFA to DFA.

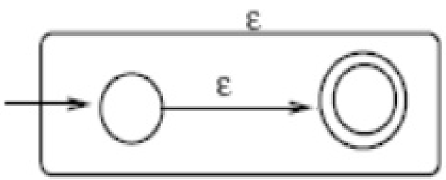
Converting Regular Expressions to NFA

In two parts;

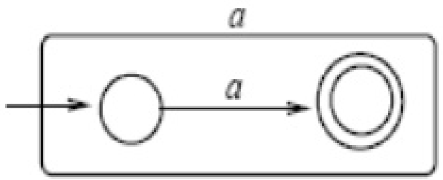
- First convert the regular expression corresponding to each token into a NFA.
  - Invariant: A single final state corresponding to each token.
- Join the NFAs obtained for all the tokens.

Converting Regular Expressions to DFA

RE for ϵ

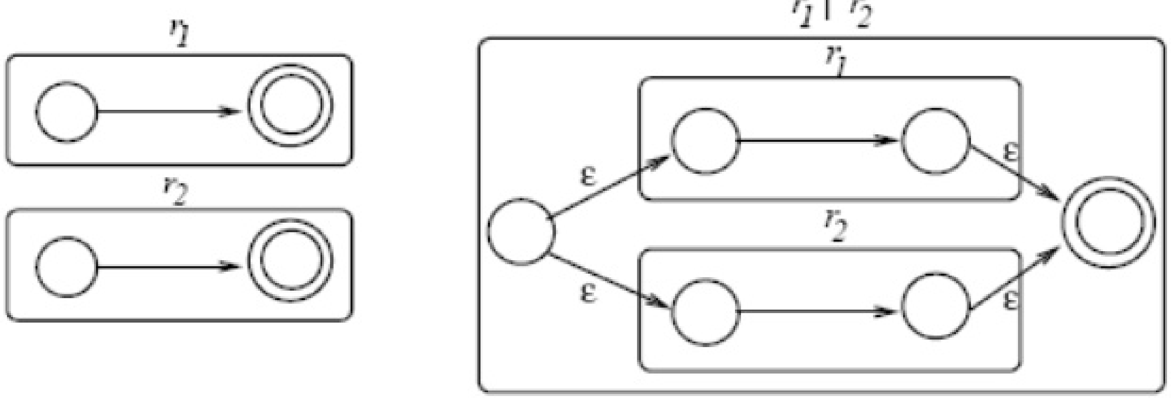


RE for a

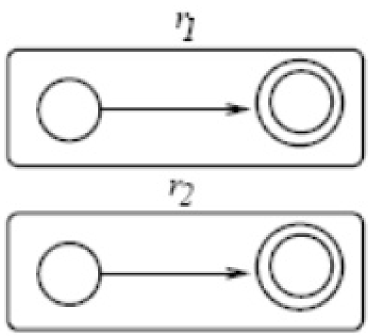


Converting Regular Expressions to NFA

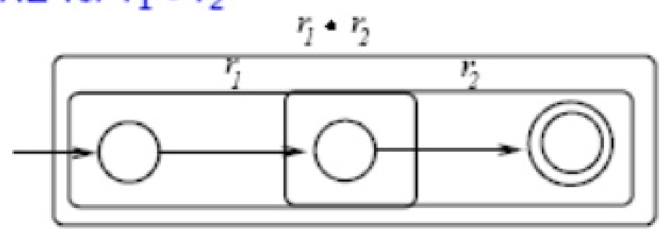
RE for $r_1|r_2$



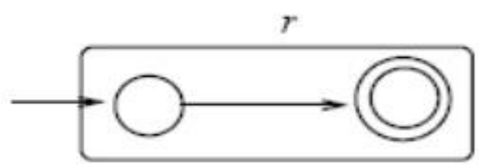
Converting Regular Expressions to NFA



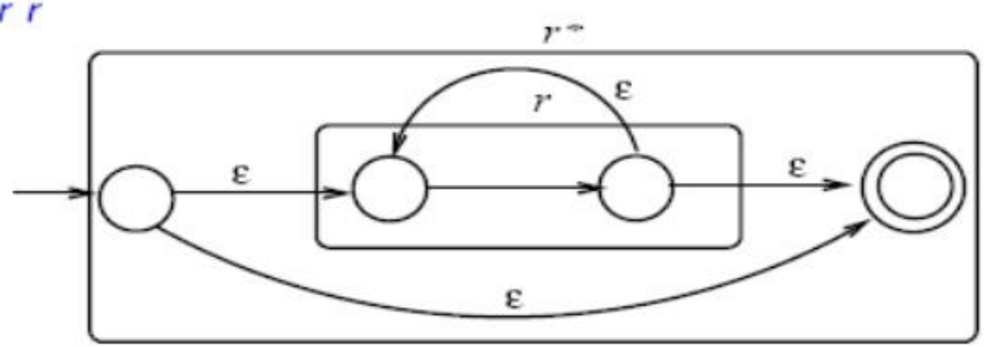
RE for $r_1 \bullet r_2$



Converting Regular Expressions to NFA



RE for r



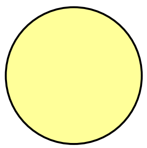
Finite Automata

- Finite Automata are recognizers.
  - FA simply say “Yes” or “No” about each possible input string.
  - A FA can be used to recognize the tokens specified by a regular expression
  - Use FA to design of a Lexical Analyzer Generator
- Two kind of the Finite Automata
  - Nondeterministic finite automata (NFA)
  - Deterministic finite automata (DFA)
- Both DFA and NFA are capable of recognizing the same languages.

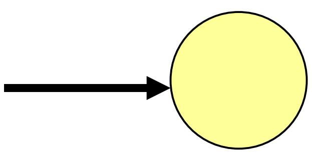
NFA Definitions

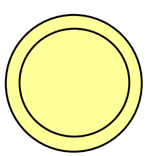
- NFA = { S, Σ, δ, s_0, F }
  - A finite set of states S
  - A set of input symbols Σ
    - input alphabet, ε is not in Σ
  - A transition function δ
    - $\delta : S \times \Sigma \rightarrow S$
  - A special start state s_0
  - A set of final states F, $F \subseteq S$ (accepting states)

Transition Graph for FA

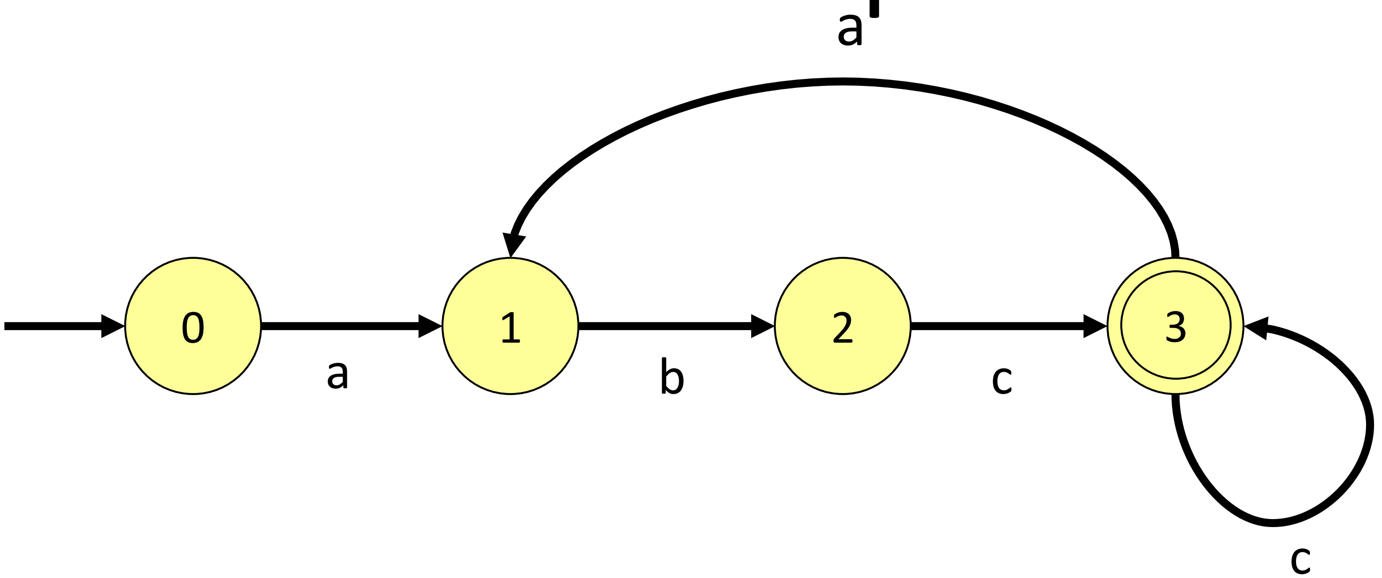
 is a state

 is a transition

 is a the start state

 is a final state

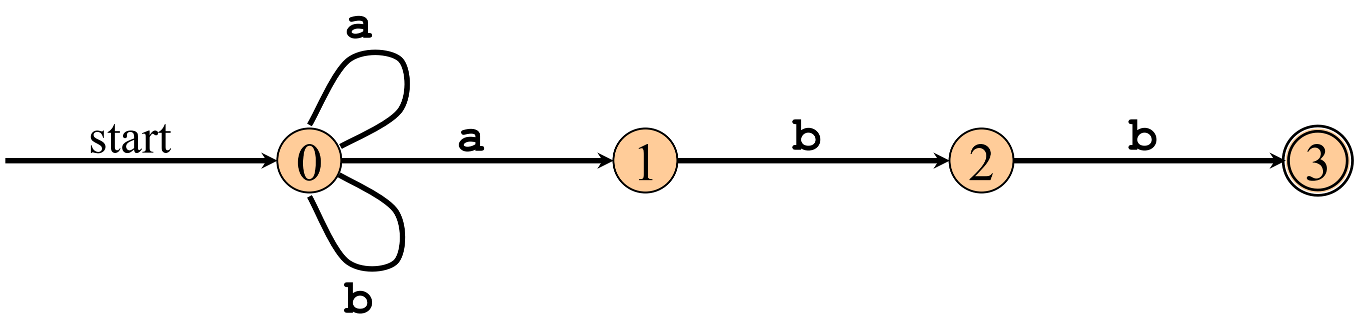
Example



- This machine accepts abccabc, but it rejects abcab.
- This machine accepts $(abc^{+})^{+}$.

Transition Table

- The mapping δ of an NFA can be represented in a transition table

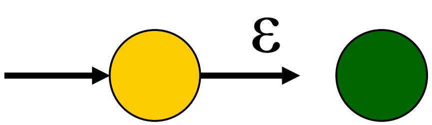


$$\delta(0, \mathbf{a}) = \{0,1\}$$
$$\delta(0, \mathbf{b}) = \{0\}$$
$$\delta(1, \mathbf{b}) = \{2\}$$
$$\delta(2, \mathbf{b}) = \{3\}$$

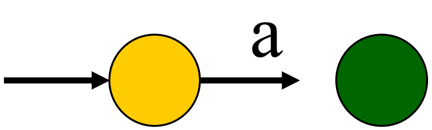
STATE	a	b	ε
0	{0, 1}	{0}	–
1	–	{2}	–
2	–	{3}	–
3	–	–	–

Construction of an NFA from a Regular Expression

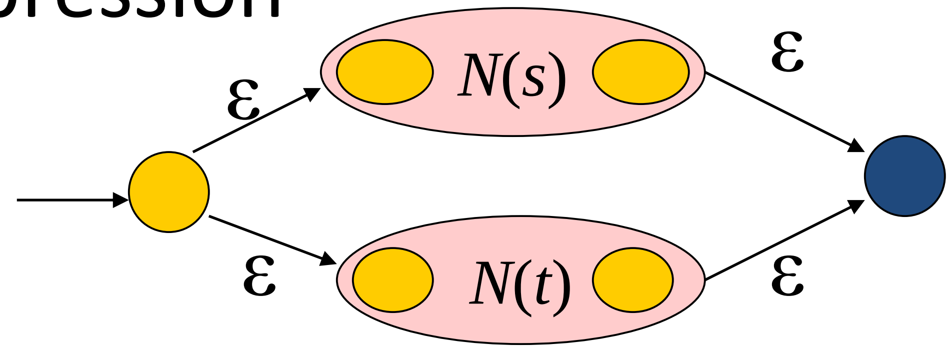
ε



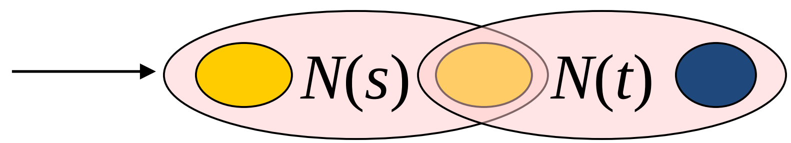
a



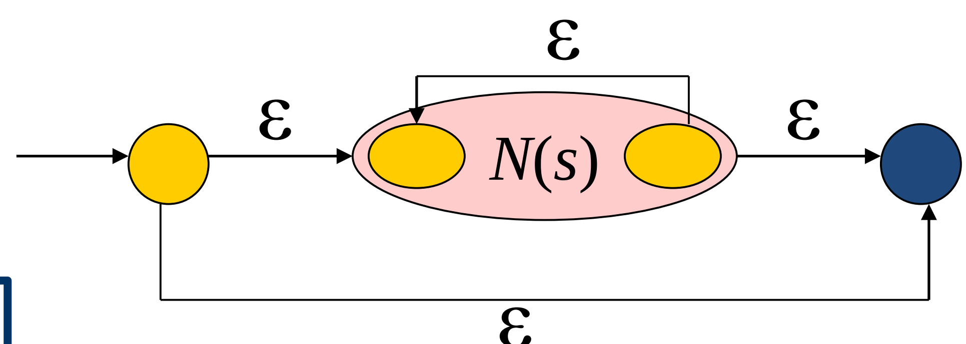
$s \mid t$



$s\ t$



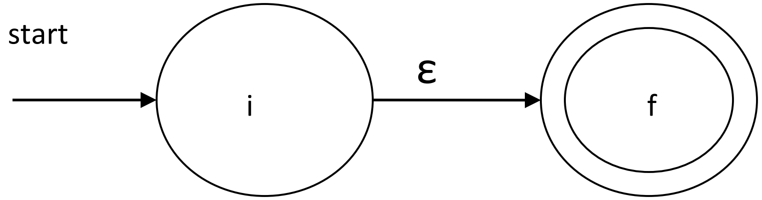
s^*



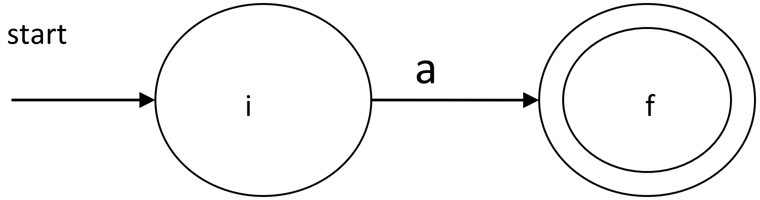
Use Thompson's Construction

NFA

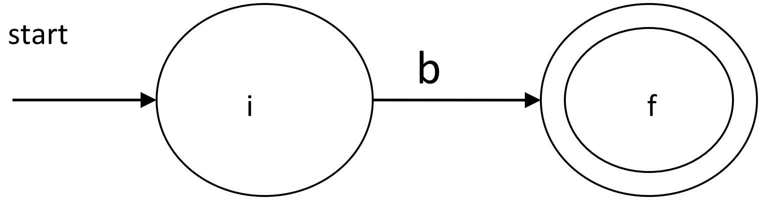
FOR INPUT ε



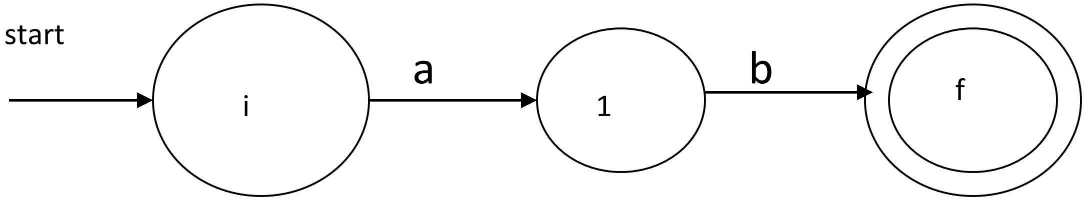
FOR INPUT a



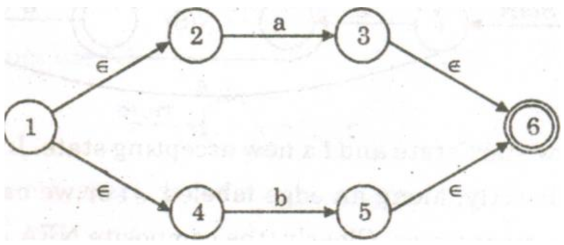
FOR INPUT b



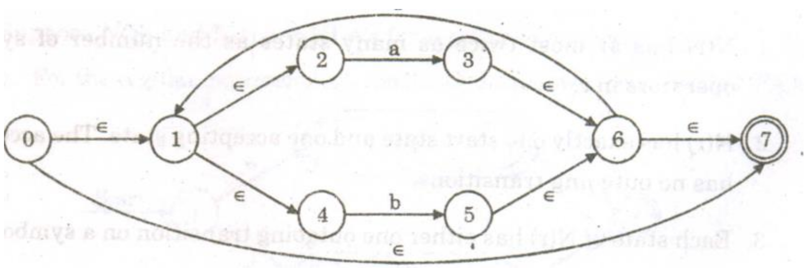
- FOR INPUT ab



- FOR INPUT a/b

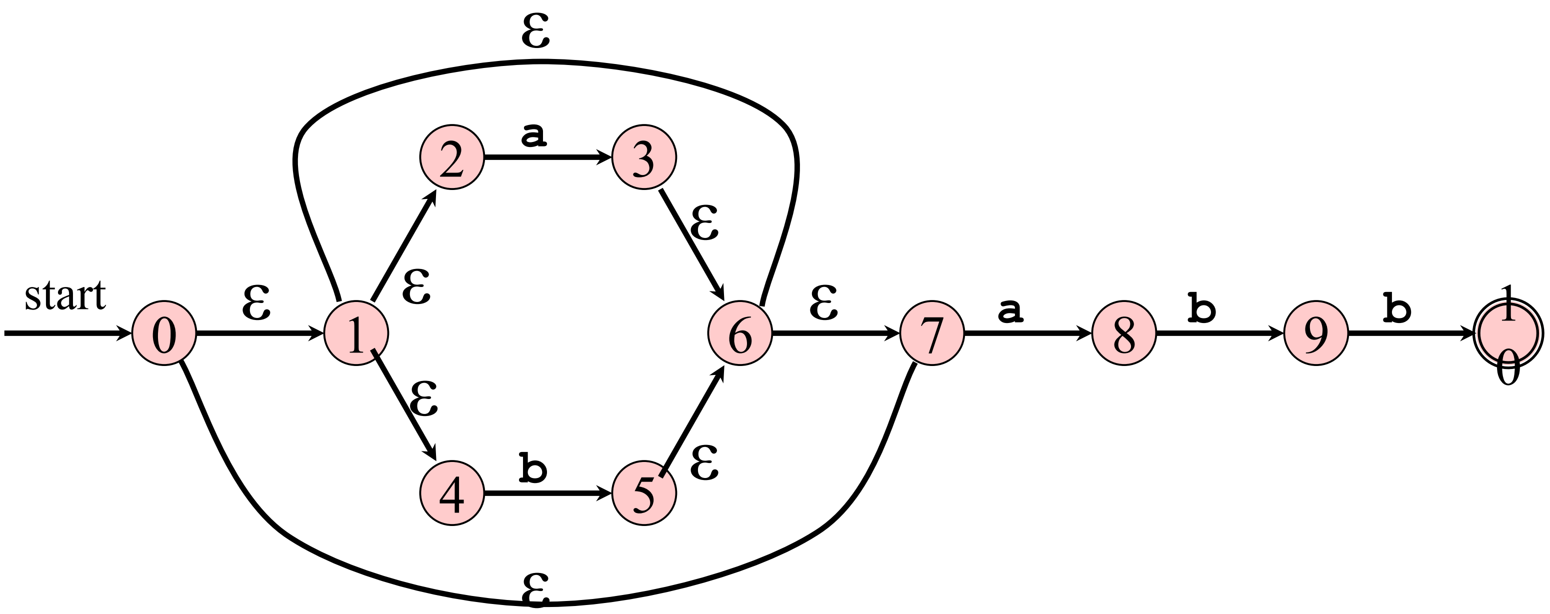


- FOR INPUT (a/b)*



Example

- **(a | b)* a b b**



The input. symbol alphabet is {a, b}.

ε - closure (0)= {0, 1, 2, 4, 7} =A

ε -closure (move (A, a)) =

ε - Closure (Move ({O, I, 2,4, 7} ,a)) =

ε- Closure ({3, 8}) = { 1, 2, 3, 4, 6, 7, 8 }= B

ε -closure (move (A, b)) =

ε - Closure (move ({ 0, 1, 2, 4,7 },b)) =

ε - Closure (5) = { I, 2, 4, 5, 6, 7 } = C

- ε - Closure (move (B, a)) =
  - **ε - Closure (move ({I, 2, 3, 4, 6, 7, 8}, a)) =**
  - **ε - Closure (3, 8) = B**
- ε- Closure (move (B, b)) =

ε -Closure (move ({ I, 2, 3,4, 6, 7, 8 }, b)) =

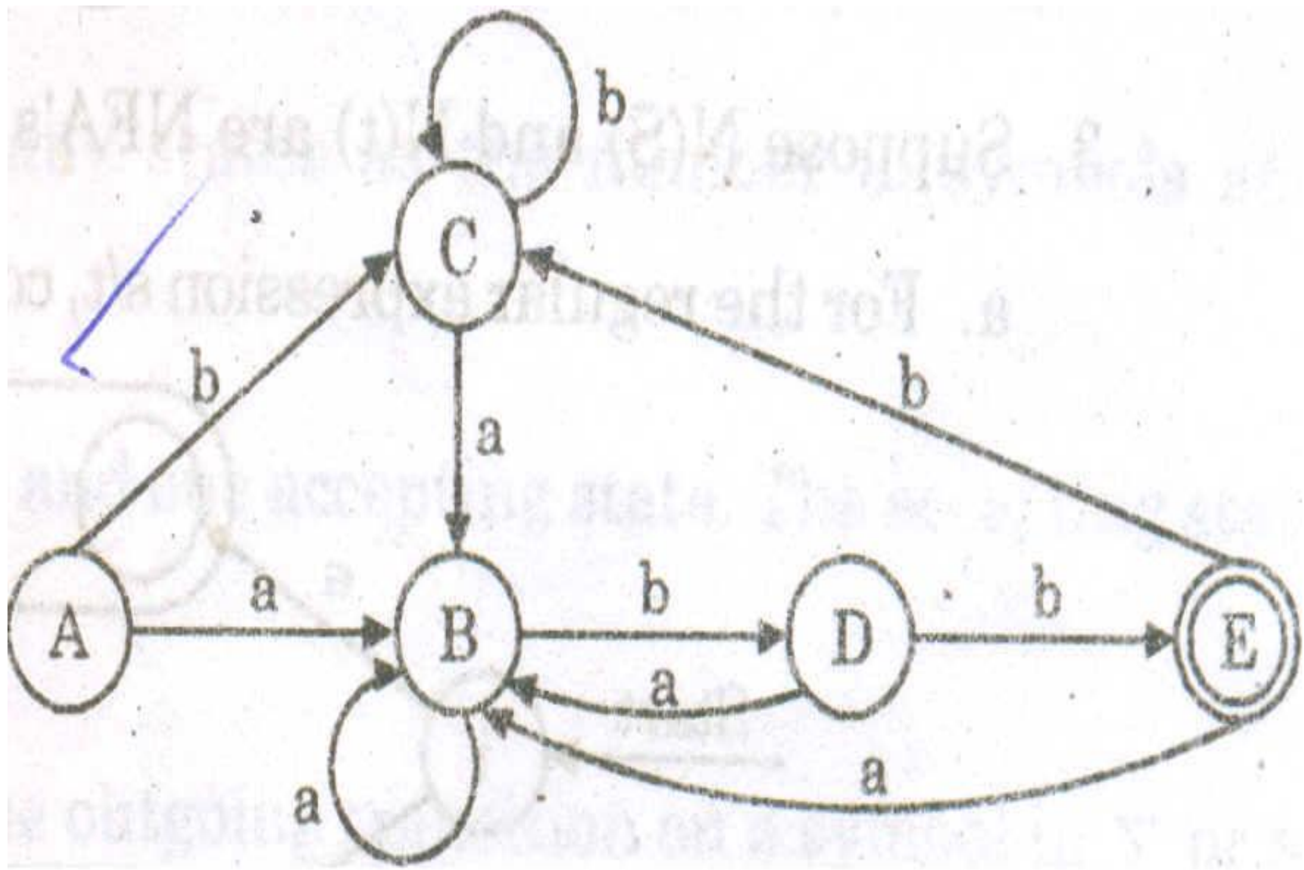
ε - Closure (5, 9) = { I, 2, 4, 5, 6, 7, 9 }=D

- ε - Closure (move (C, a)) =
  - ε - Closure (move ({ I, 2, 4, 5, 6, 7 }, a)= ε - Closure (3,8) == Btransition [C, a] = B
- ε- Closure (move (C, b)) == ε - Closure (move ({ I, 2, 4, 5, 6, 7 } , b)) = ε- Closure (5)= Ctransition [C, b] = C

- ε- Closure (move (D, a))= ε - Closure (move ({I, 2, 4,5, 6, 7,9}, a)) = ε- Closure (3, 8) = B.transition (D, a] = B.
- ε - Closure (move (D, b))= ε - Closure (move ({I, 2, 4, 5, 6, 7, 9} b) = ε. Closure (5, 10)= {1, 2, 4, 5; 6, 7, 1O} = E
- ε. Closure (move (E, a)) =ε - Closure (move ({1, 2, 4, 5,6, 7, 10}, a)= ε - Closure (3, 8) = Btransition [E, a] = B.
- ε - Closure (move (E, b))=ε - Closure (move ({1, 2, 4, 5, 6, 7, 1O}, b))= ε- Closure (5) = C transition [E, b] = C

Transition Table &Transition Diagram

State	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Minimizing the Number of States of a DFA

- **The initial partition consists of two groups.**
- **(ABCD), consisting of non- final states, (E) the final states.**
- **Now consider (ABCD), on input a, each of these states goes to B, so they could all be placed in one group as far as input a is concerned.**
- **However, on input b, A, Band C go to members of the group (ABCD) ,while D goes to E, a member of another group.**
- **Thus, new (ABCD) must be split into *tvio* groups (ABC) and (D). Thenew value of is (ABC) (D) (E).**
- **(ABC) must be split into two groups (AC) (B).**
- **Since on input b, A and C each go to C, while B goes to D, a member of a group different from that of C. The next split is (AC) (B) (D) (E).**

(AC) (B) (D) (E).

TD

State	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

MINIMIZED TD

State	a	b
A(START)	B	A
B	B	D
C	B	E
D(ACCEPT)	B	A

Conversion of an NFA to a DFA

- The subset construction algorithm converts an NFA into a DFA using the following operation.

Operation	Description
ε- *closure*(*s*)	Set of NFA states reachable from NFA state s on ε-transitions alone.
ε- *closure*(T)	Set of NFA states reachable from some NFA state s in set T on ε-transitions alone. $= \cup_{s \text{ in } T}$ ε- *closure*(*s*)
move(T, a)	Set of NFA states to which there is a transition on input symbol **a** from some state s in T

Subset Construction(1)

Initially, ε-*closure*(*s*0) is the only state in *Dstates* and it is unmarked;

while (there is an unmarked state *T* in *Dstates)* **{**

    mark *T;*

    for (each input symbol $a \in \Sigma$) {

        U = ε-*closure*(*move*(*T, a*));

        if (U is not in Dstates*)*

            add U as an unmarked state to Dstates

        Dtran[T, a] = U

    }

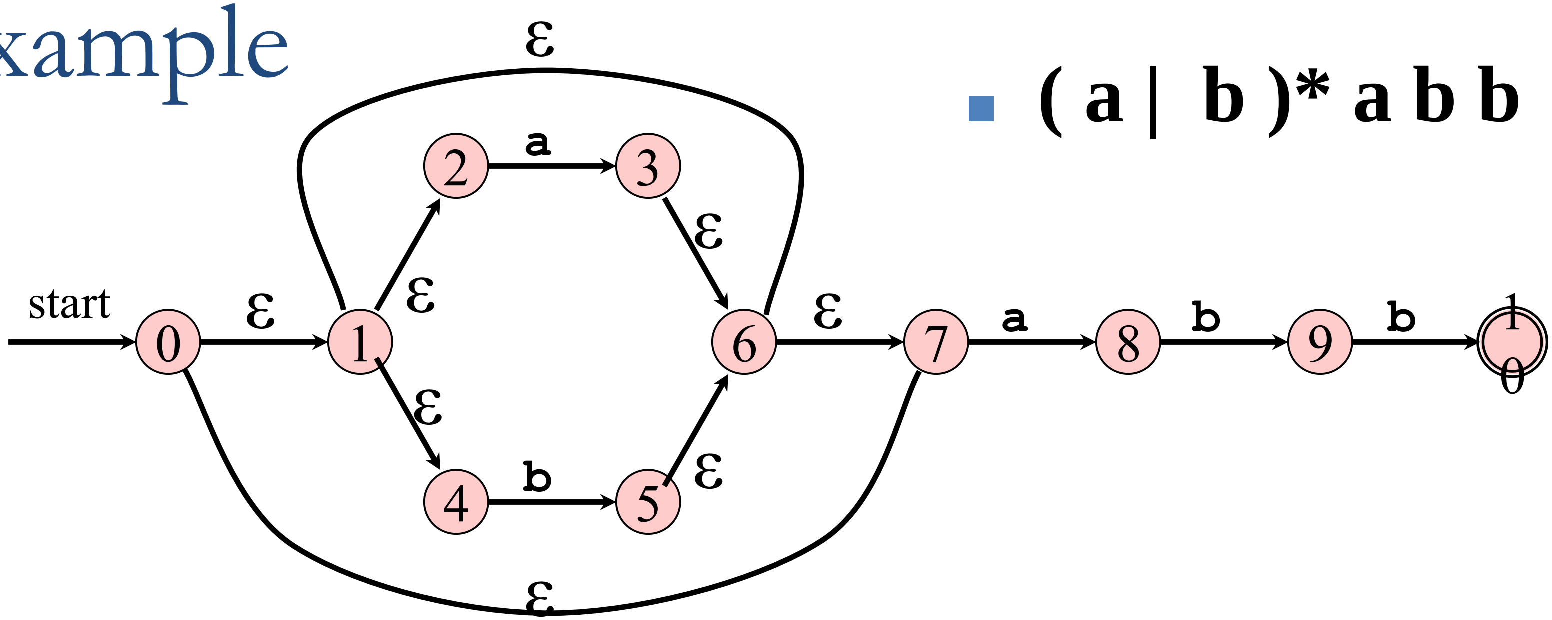
}

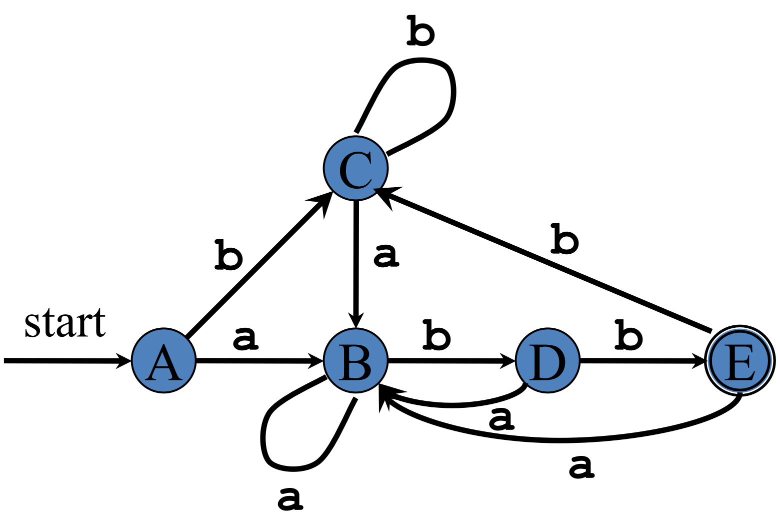
Computing ε- *closure*(T)

```
push all states of T onto stack;
initialize ε-closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off 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;
                }
}
```

Example

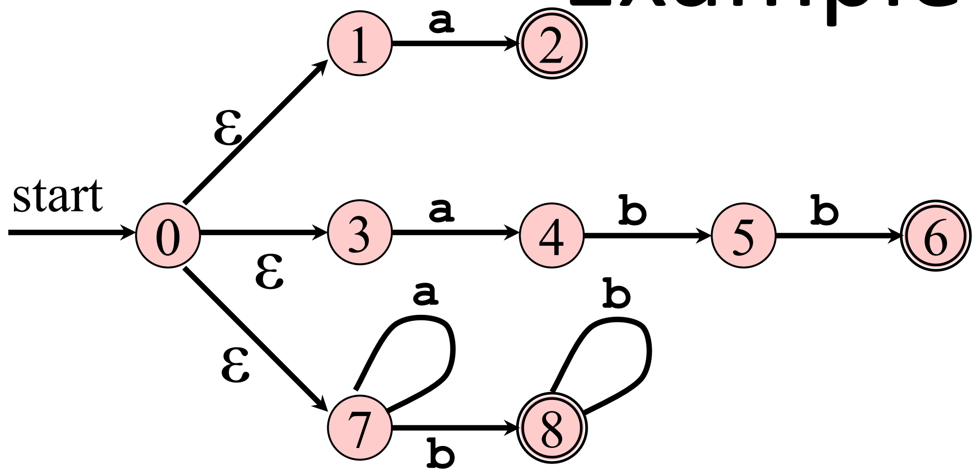
- **(a | b)* a b b**



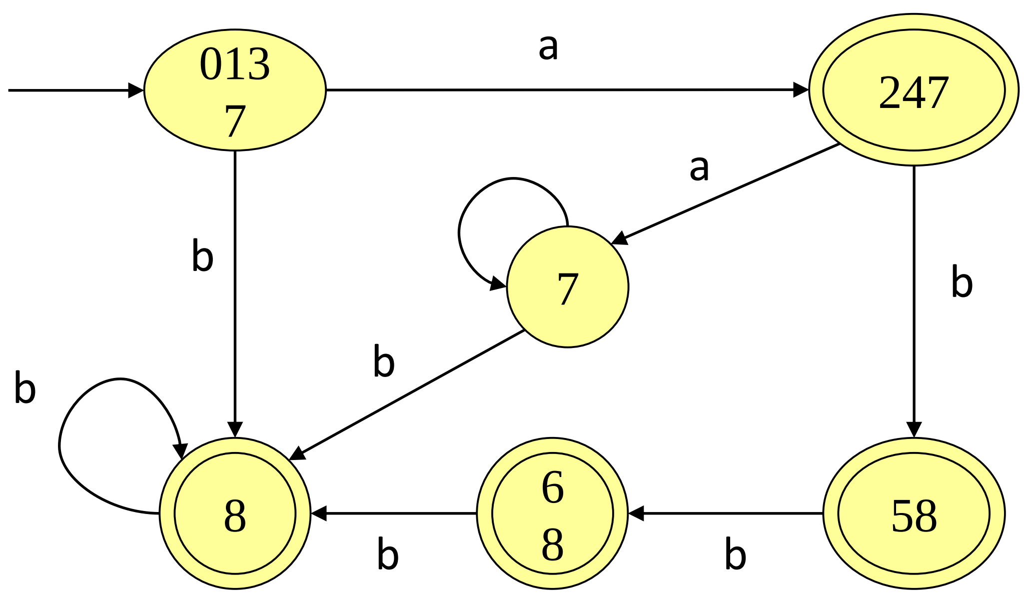


NFA State	DFA State	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,3,5,6,7,10}	E	B	C

Example



- **a**
- **abb**
- **a^*b^+**



Dstates

A = {0,1,3,7}

B = {2,4,7}

C = {8}

D = {7}

E = {5,8}

F = {6,8}

Minimizing the DFA

- Step 1
  - Start with an initial partition II with two group: F and S-F (aceepting and nonaccepting)
- Step 2
  - Split Procedure
- Step 3
  - If ($II_{new} = II$)

    $II_{final} = II$ and continue step 4

    else

    $II = II_{new}$ and go to step 2
- Step 4
  - Construct the minimum-state DFA by II_{final} group.
  - Delete the dead state

Split Procedure

Initially, let $\Pi_{new} = \Pi$

for (each group G of Π) {

  Partition G into subgroup such that

    two states **s** and **t** are in the same subgroup

    if and only if

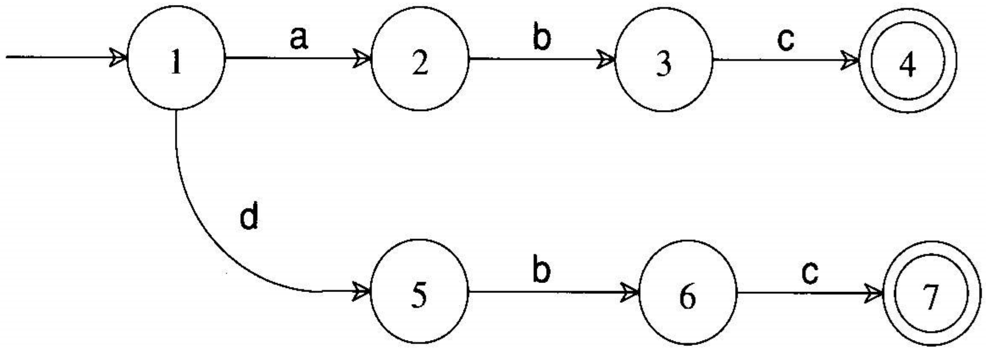
    for all input symbol a, states s and t have transition on a to states in the same group of Π.

  /* at worst, a state will be in a subgroup by itself */

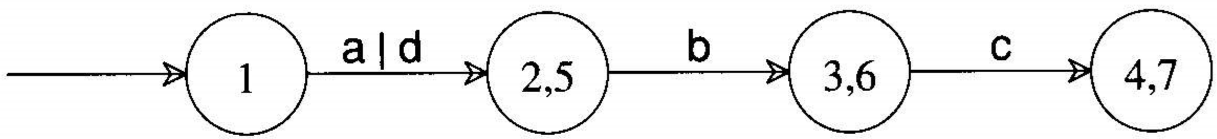
  replace G in Π_{new} by the set of all subgroup formed

}

Example



- initially, two sets {1, 2, 3, 5, 6}, {4, 7}.
- {1, 2, 3, 5, 6} splits {1, 2, 5}, {3, 6} on c.
- {1, 2, 5} splits {1}, {2, 5} on b.



Minimizing the DFA

- Major operation: partition states into equivalent classes according to
  - final / non-final states
  - transition functions

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

(A B C D E)
(A B C D) (E)
(A B C) (D) (E)
(A C) (B) (D) (E)

	a	b
A C	B	A C
B	B	D
D	B	E
E	B	A C

Important States of an NFA

- The "important states" of an NFA are those without an ε-transition, that is
  - if $move(\{s\}, a) \neq \emptyset$ for some a then s is an important state
- The subset construction algorithm uses only the important states when it determines $\varepsilon\text{-}closure\ (\ move(T, a)\ )$
- Augment the regular expression r with a special end symbol # to make accepting states important: the new expression is $r\#$

Converting a RE Directly to a DFA

- Construct a syntax tree for (r)#
- Traverse the tree to construct functions *nullable*, *firstpos*, *lastpos*, and *followpos*
- Construct DFA D by algorithm 3.62

Function Computed From the Syntax Tree

- *nullable(n)*
  - The subtree at node n generates languages including the empty string
- *firstpos(n)*
  - The set of positions that can match the first symbol of a string generated by the subtree at node n
- *lastpos(n)*
  - The set of positions that can match the last symbol of a string generated be the subtree at node n
- *followpos(i)*
  - The set of positions that can follow position i in the tree

Rules for Computing the Function

Node n	$\boldsymbol{nullable(n)}$	$\boldsymbol{firstpos(n)}$	$\boldsymbol{lastpos(n)}$
A leaf labeled by $\boldsymbol{\varepsilon}$	**true**	$\emptyset$	$\emptyset$
A leaf with position $\boldsymbol{i}$	**false**	$\{i\}$	$\{i\}$
$\mathbf{n = c_1 \mid c_2}$	$nullable(c_1)$ **or** $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$	$lastpos(c_1) \cup lastpos(c_2)$
$\mathbf{n = c_1 \; c_2}$	$nullable(c_1)$ **and** $nullable(c_2)$	**if** **(** $nullable(c_1)$) $firstpos(c_1) \cup firstpos(c_2)$ **else** $firstpos(c_1)$	**if** **(** $nullable(c_2)$) $lastpos(c_1) \cup lastpos(c_2)$ **else** $lastpos(c_2)$
$\mathbf{n = c_1^*}$	*true*	$firstpos(c_1)$	$lastpos(c_1)$

Computing *followpos*

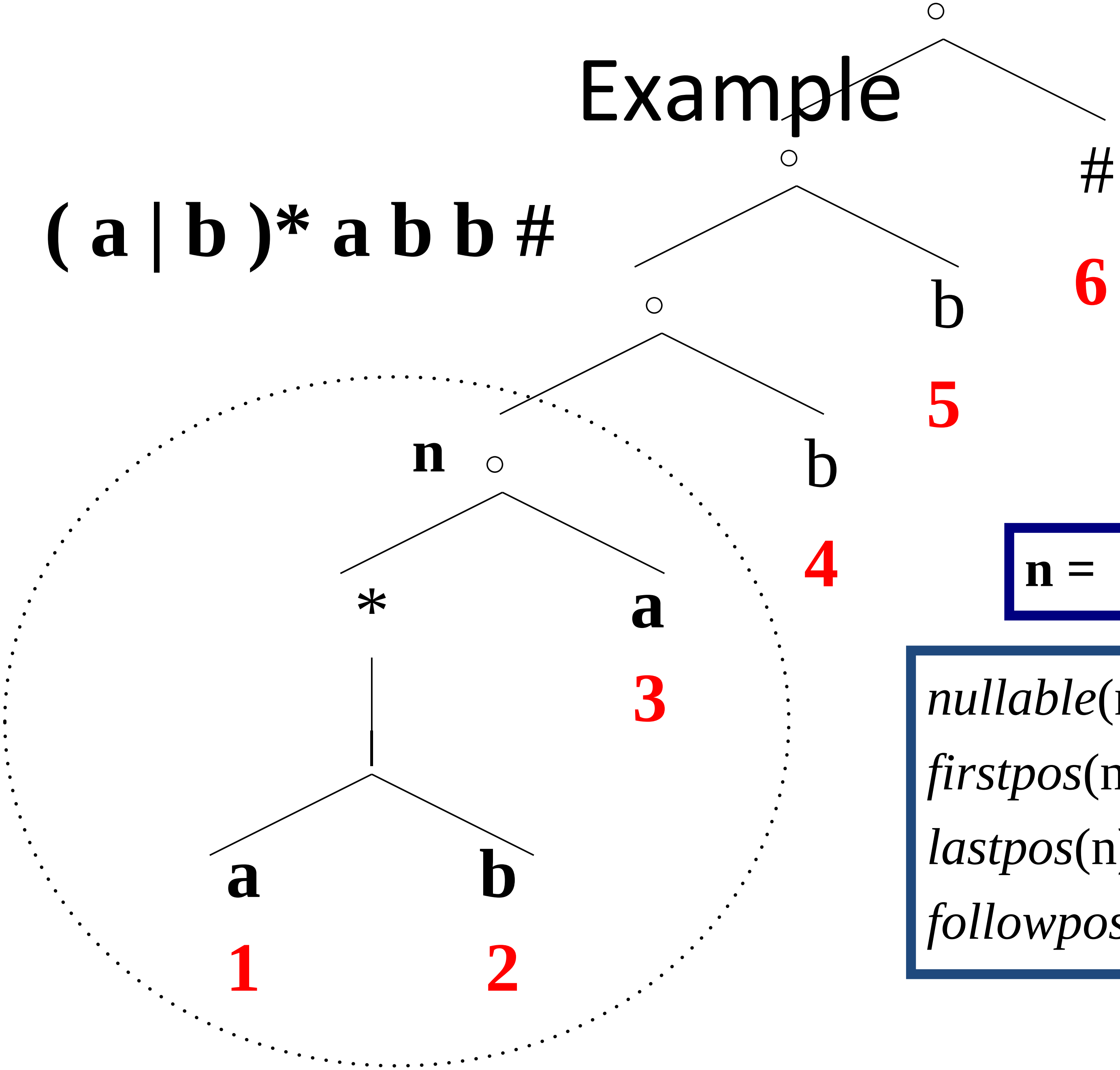
```
for (each node n in the tree)
{
    //n is a cat-node with left child c1 and right child c2
    if ( n == c1. c2)
        for (each i in lastpos(c1) )
            followpos(i) = followpos(i) ∪ firstpos(c2);
    else if (n is a star-node)
        for ( each i in lastpos(n) )
            followpos(i) = followpos(i) ∪ firstpos(n);
}
```

Converting a RE Directly to a DFA

```
Initialize Dstates to contain only the unmarked state
firstpos(n0), where n0 is the root of syntax tree T for
(r)#;
while ( there is an unmarked state S in Dstates ) {
  mark S;
  for ( each input symbol a ∈ Σ ) {
      let U be the union of followpos(p)
          for all p in S that correspond to a;
      if (U is not in Dstates )
            add U as an unmarked state to Dstates
      Dtran[S,a] = U;
  }
}
```

Example

(a | b)* a b b #



n = (a | b)* a

nullable(n) = **false**

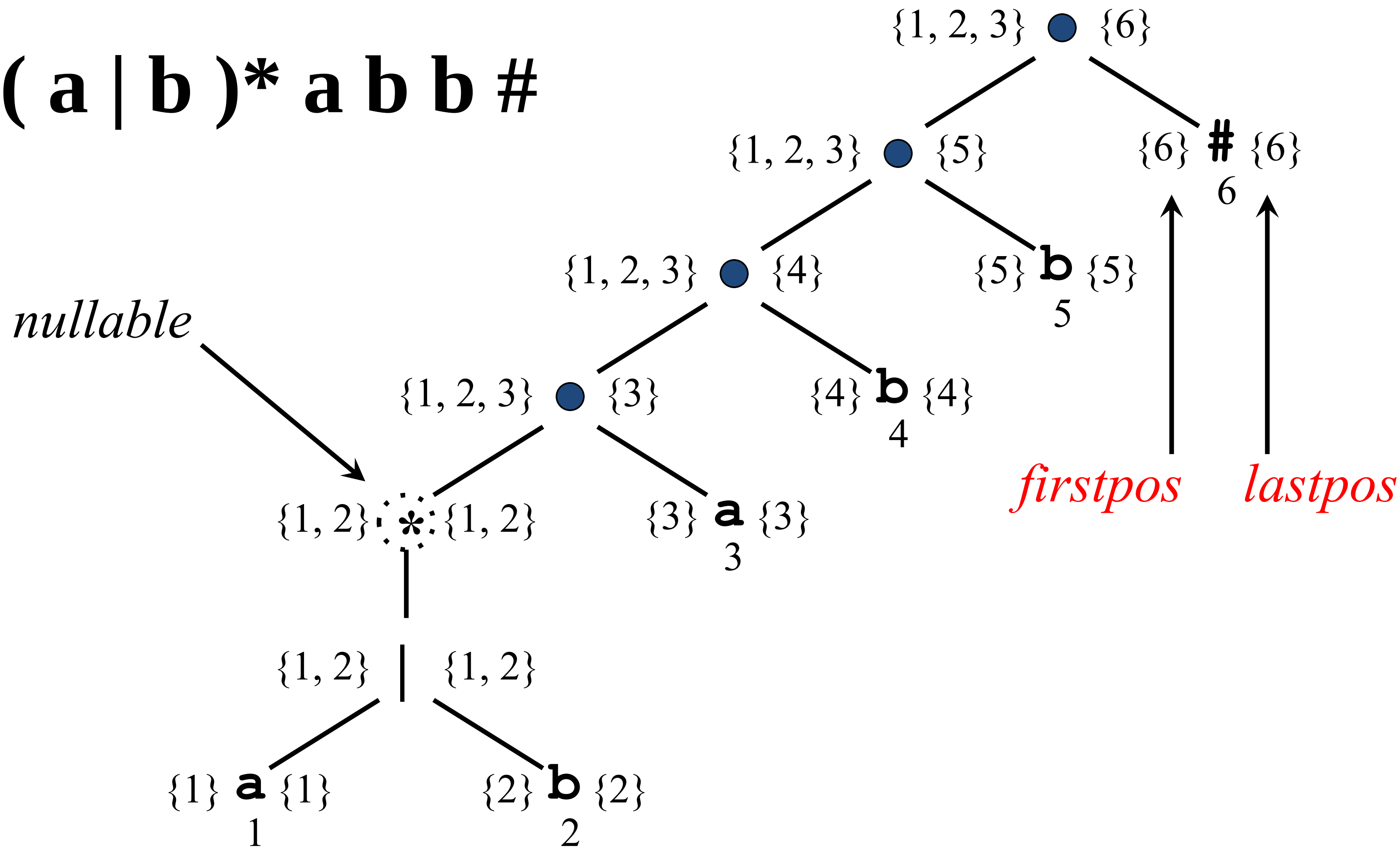
firstpos(n) = { 1, 2, 3 }

lastpos(n) = { 3 }

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

Example

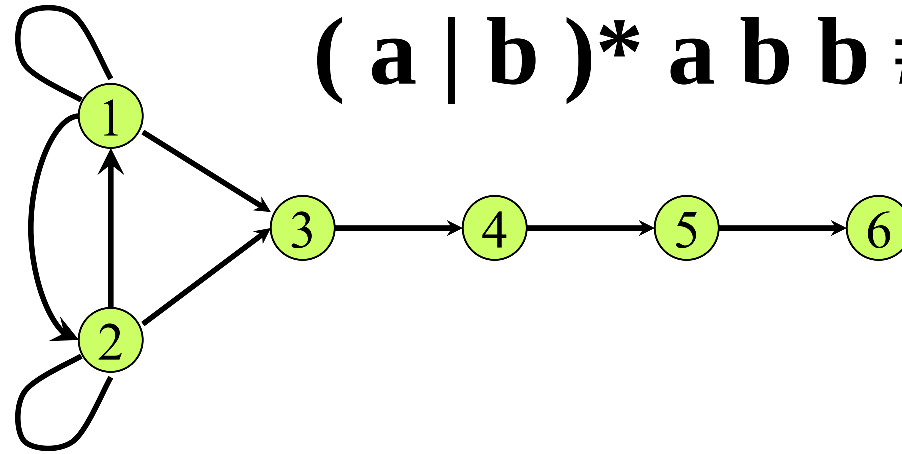
(a | b)* a b b #

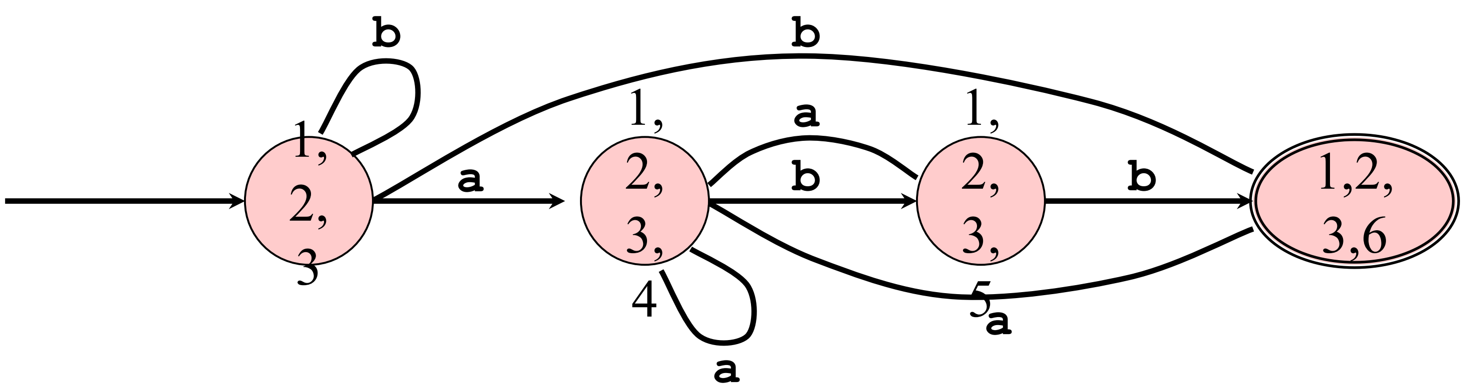


Example

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

(a | b)* a b b #





DFA

- DFA is a special case of an NFA
  - There are no moves on input ε
  - For each state s and input symbol a, there is exactly one edge out of s labeled a.
- Both DFA and NFA are capable of recognizing the same languages.

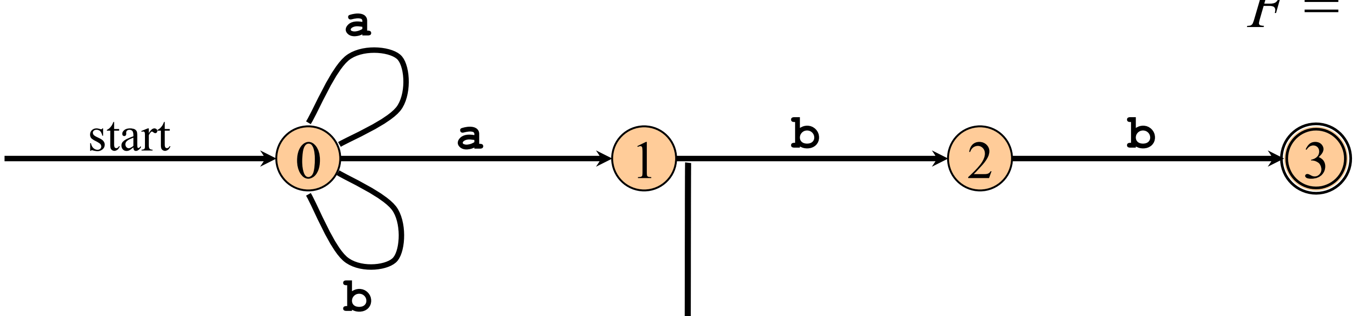
Simulating a DFA

- Input
  - An input string *x* terminated by an end-of-file character **eof**. A DFA *D* with start state s0, accepting states F, and transition function *move*.
- Output
  - Answer “yes” if *D* accepts *x*; “no” otherwise.

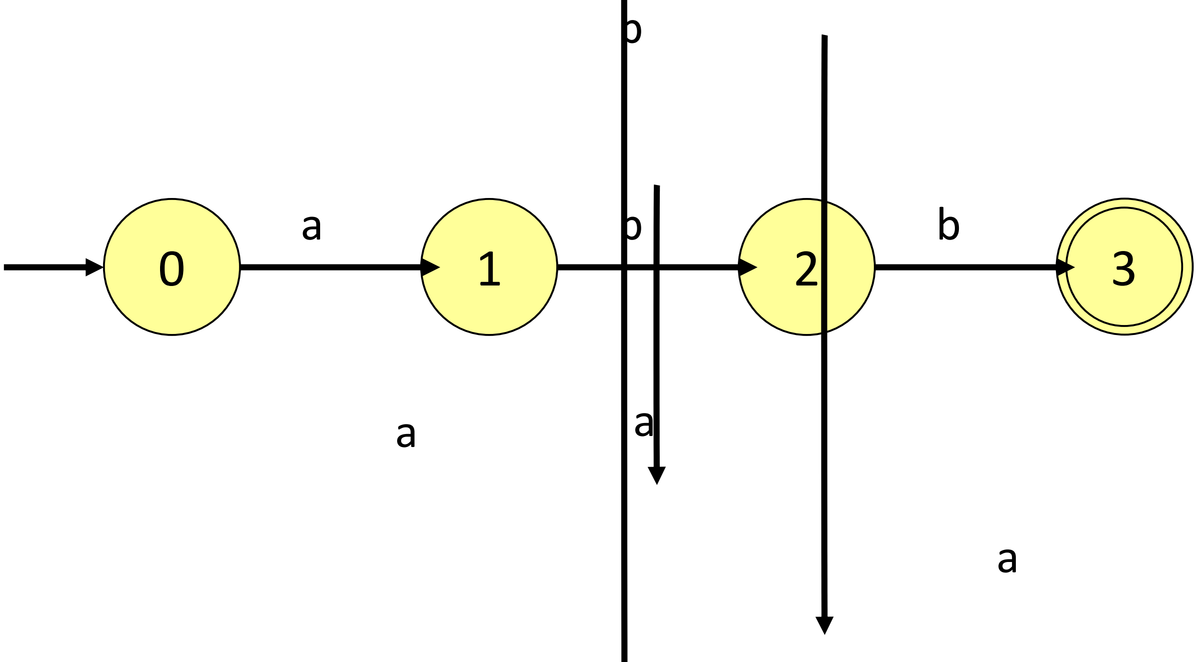
```
s = s0
c = nextChar();
while ( c != eof ) {
   s = move(s, c);
   c = nextChar();
}
if (s is in F )
   return “yes”;
else
   return “no”;
```

NFA vs DFA

$$S = \{0,1,2,3\}$$
$$\Sigma = \{\mathtt{a}, \mathtt{b}\}$$
$$s_0 = 0$$
$$F = \{3\}$$

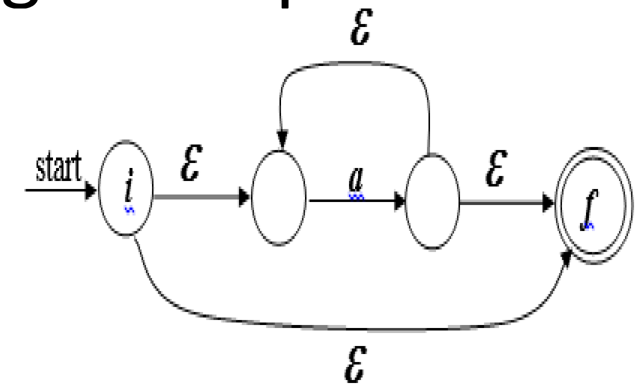


(a | b)*abb

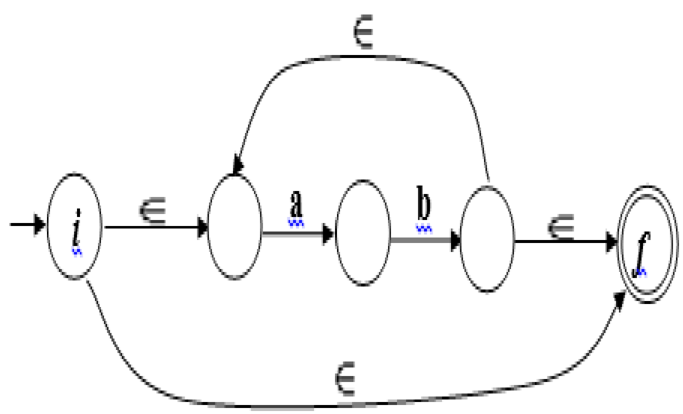


From a Regular Expression to an NFA

4-For the regular expression ***a**** construct the following co

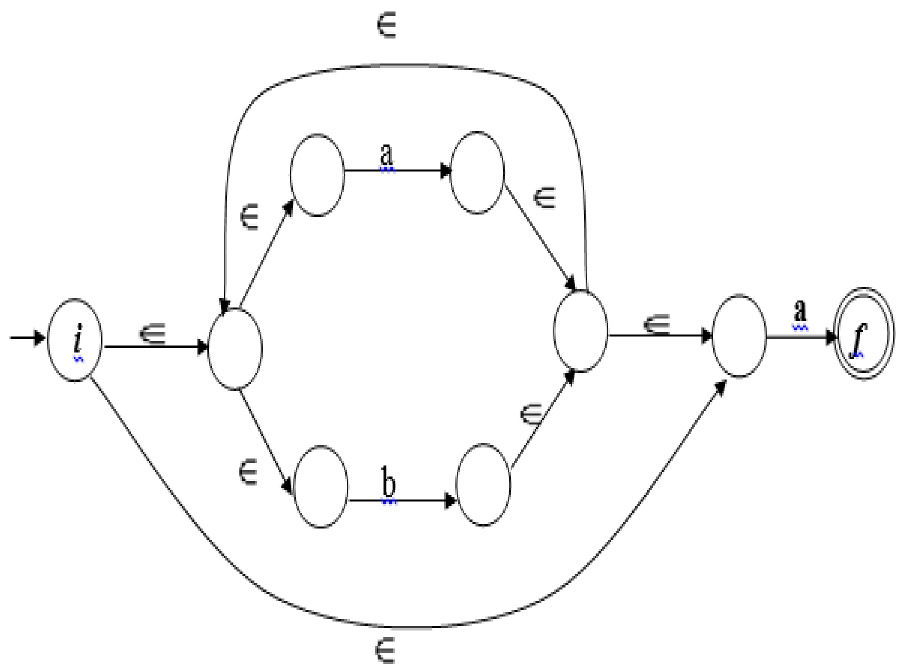


- RE = (***ab***)



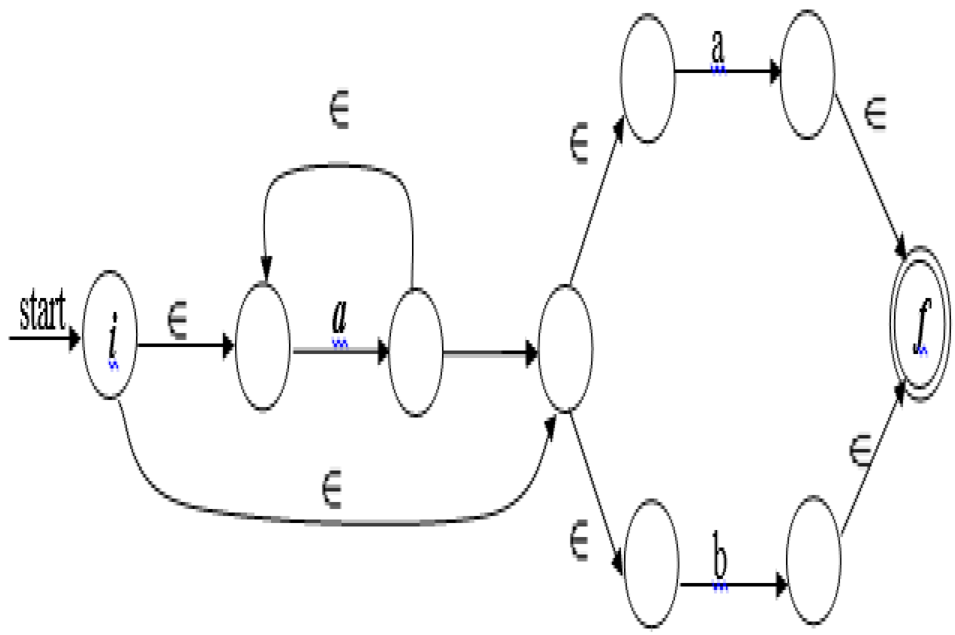
From a Regular Expression to an NFA

RE = (***a*** | ***b***)****a***



From a Regular Expression to an NFA

RE = ***a**** (***a*** | ***b***)



The Regular Language

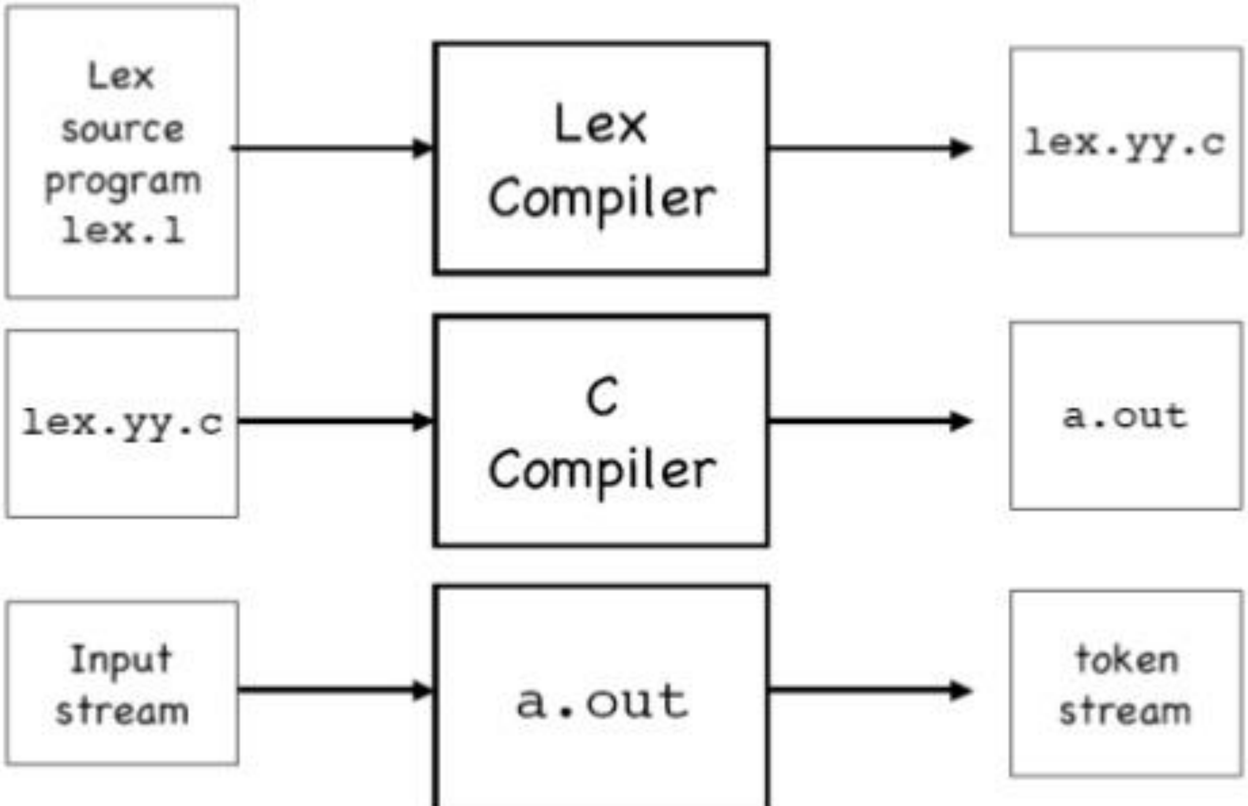
- The regular language defined by an NFA is the set of input strings it accepts.
  - Example: **`(a□b)*abb`** for the example NFA
- An NFA **accepts** an input string *x* if and only if
  - there is some path with edges labeled with symbols from *x* in sequence from the start state to some accepting state in the transition graph
  - A state transition from one state to another on the path is called a **move.**

Time and Space Complexity

Automaton	*Space (worst case)*	*Time (worst case)*
NFA	$O(\lvert r \rvert)$	$O(\lvert r \rvert \times \lvert x \rvert)$
DFA	$O(2^{\lvert r \rvert})$	$O(\lvert x \rvert)$

Lexical Analysis With Lex

Lexical analysis with Lex



Lex source program format

- The Lex program has three sections, separated by %%:

```
declarations
%%
transition rules
%%
auxiliary code
```

Declarations section

- Code between %{ and }% is inserted directly into the lex.yy.c. Should contain:
  - Manifest constants (#define for each token)
  - Global variables, function declarations, typedefs
- Outside %{ and }%, REGULAR DEFINITIONS are declared. Examples:

delim	[\t\n]
ws	{delim}$^+$
letter	[A-Za-z]

Each definition is a name followed by a pattern. Declared names can be used in later patterns, if surrounded by { }.

Translation rules section

Translation rules take the form

p_1 { $action_1$ }

p_2 { $action_2$ }

… …

p_n { $action_n$ }

Where pi is a regular expression and $action_i$ is a C program fragment to be executed whenever p_i is recognized in the input stream.

In regular expressions, references to regular definitions must be enclosed in {} to distinguish them from the corresponding character sequences.