Lexical Analysis

Lecture 03

Timeline

- 1930s: Turing Machine
- 1940's & 1950s: Finite Automata,
  FOTRAN and COBOL (Did not have formal definition)
- 1951: Stephen Cole Kleene invents regular expressions
- Late 1950s: Chomsky Normal Form, Formal Grammar
  S → LT (L= Letter and T= Tail piece of an identifier)
  Where everything is a sentence in Chomsky Notation
- 1960: ALGOL was introduced with notations similar to Chomsky Notations to define what is a legal program
  <identifier>::=<letter><tail>
  (::= is backus-naur form or simply BFN it similar to XML, isn't it ?)
- 1969: S. Cook, extended Turing study, traceable and intractable (NP-hard problem)

Example or Regular Expression / Regular Definition:

Regular Expression for numbers

digit $\rightarrow$ 0|1|…|9

digits $\rightarrow$ digit digit*

optional_fraction $\rightarrow$ .digits|ε

optional_exponent $\rightarrow$ (E (+|-| ε) digits) | ε

num $\rightarrow$ digits optional_fraction optional_exponent

Using shorthands:

digit $\rightarrow$ 0|1|…|9

digits $\rightarrow$ digit$^+$

optional_fraction $\rightarrow$ (.digits)?

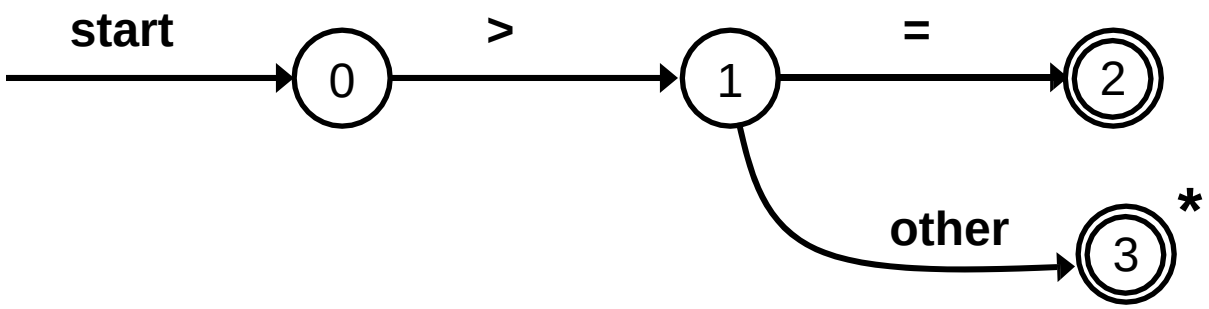
optional_exponent $\rightarrow$ (E (+|-| ε) digits) ?

num $\rightarrow$ digits optional_fraction optional_exponent

Transition Diagram

- A stylized flowchart produced intermediately in the construction of lexical analyzer
- Depicts the actions take place in lexical analyzer

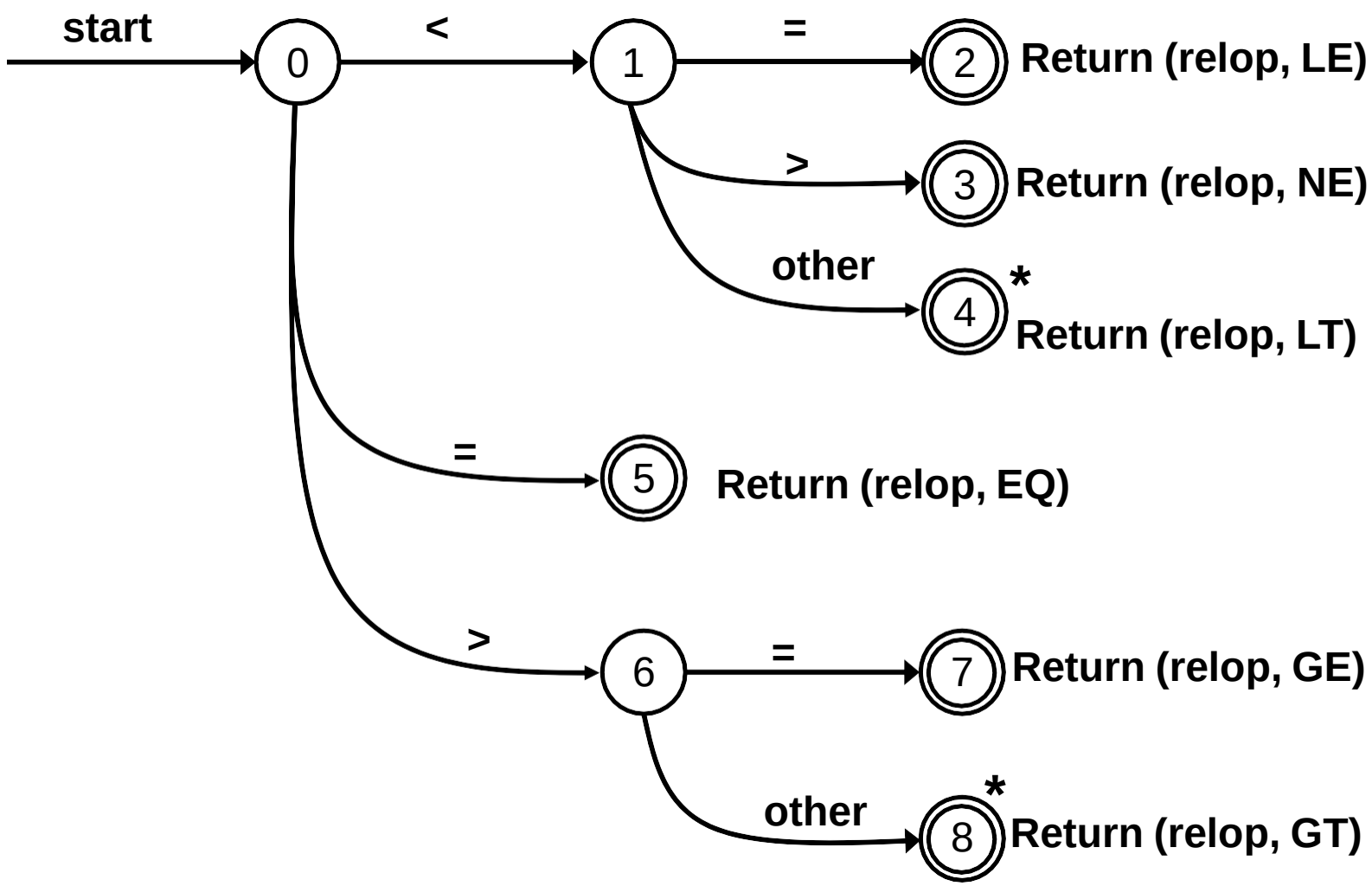
- ***states***: positions in a transition diagram
- ***edges***: arrows connecting the states
- ***start state***: initial state of transition diagram
- ***accepting state***: token recognized
- ***action***: (optional) associated with a state that is executed when the state is entered



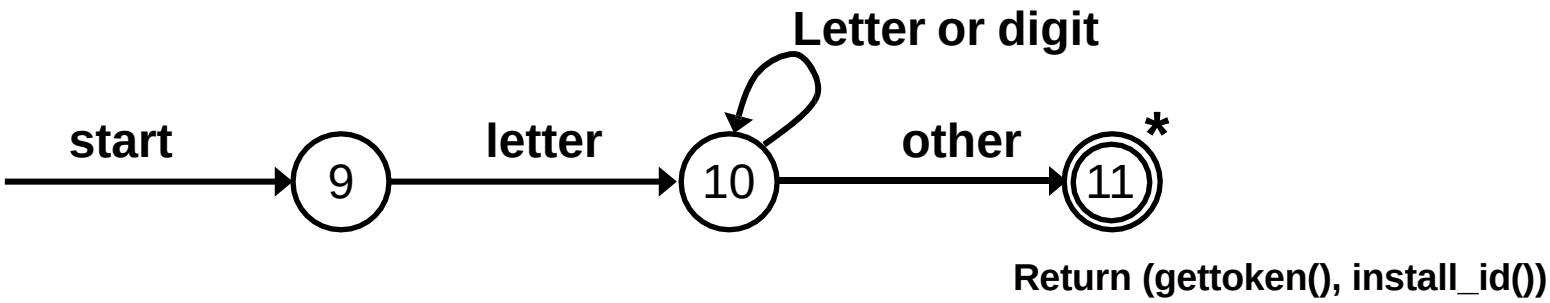
Example

- Transition diagram for token **relop**

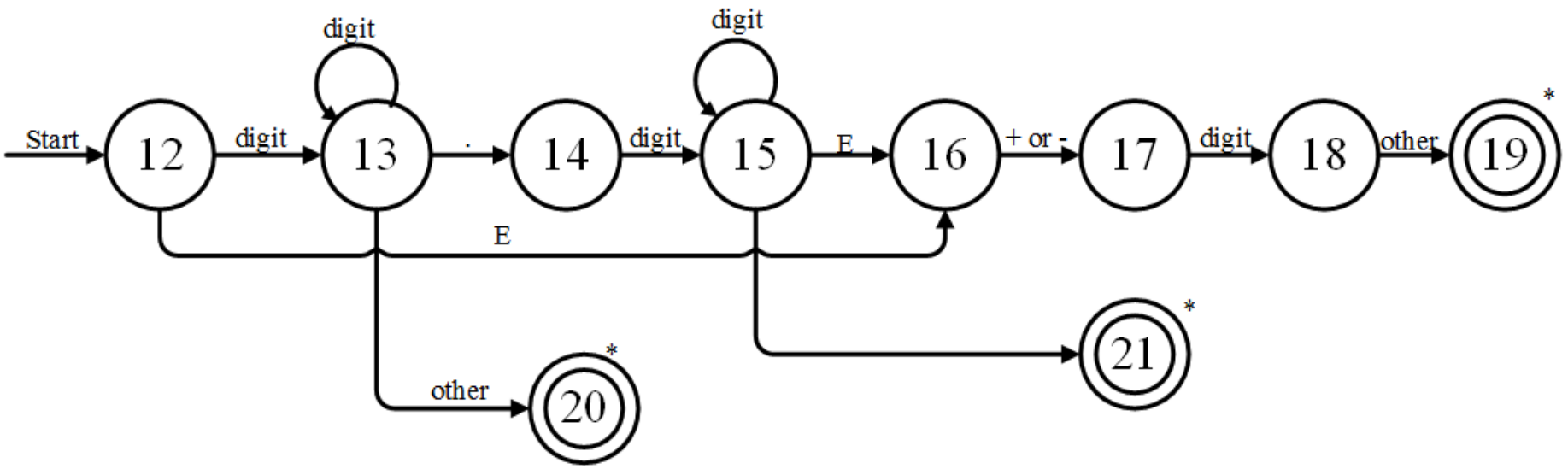
< | > | <= | >= | = | <>



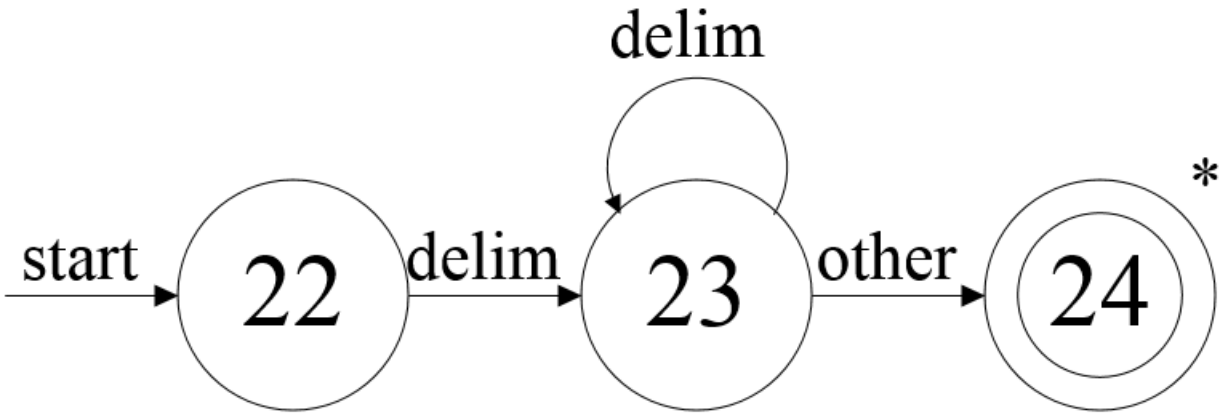
Transition diagram for Reserved Word or Identifier



Transition Diagram for Unsigned Numbers

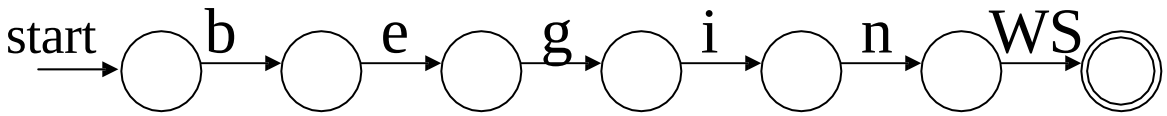


Transition Diagram for Whitespace

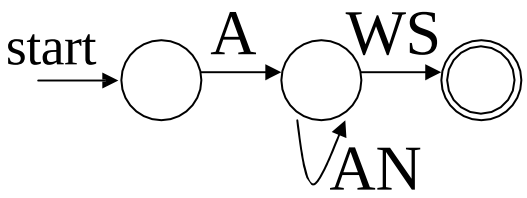


Capturing Multiple Tokens

Capturing keyword “begin”



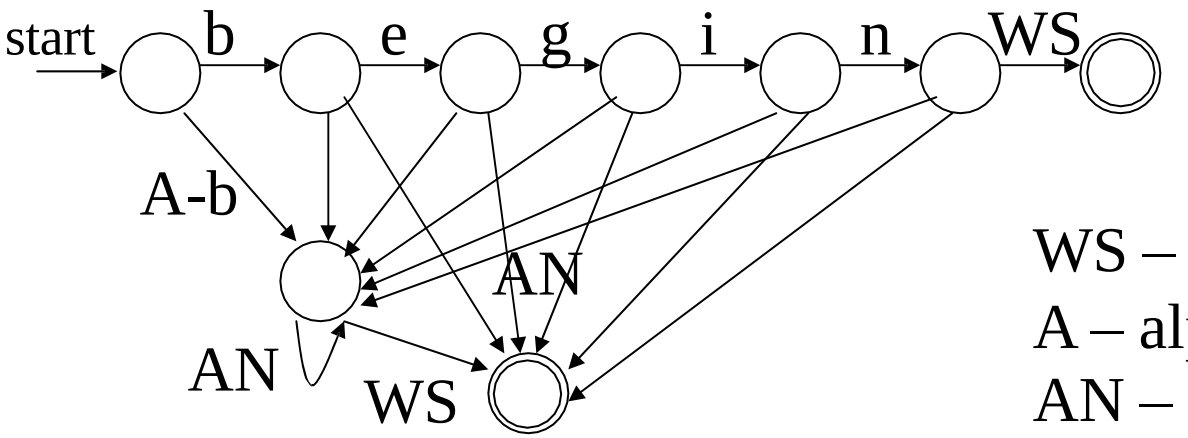
Capturing variable names



WS – white space
A – alphabetic
AN – alphanumeric

What if both need to happen at the same time?

Capturing Multiple Tokens



WS – white space
A – alphabetic
AN – alphanumeric

Machine is much more complicated – just for these two tokens!

Architecture of a Tansition Diagram Based Lexical Analyzer

```
TOKEN getRelop() {
    TOKEN retToken = new(RELOP) ;
    while(1) {  /* repeat character processing until
                   a return or failure occurs */
    switch(state) {
        case 0:
            c = nextChar();
            if(c =='<')state=1;
            elseif(c =='=')state=5;
            else if ( c == '>' ) state = 6;
            /* lexeme is not a relop */
            else fail() ;
            break;
        case 1: ...
        ...
        case 8:
            retract();
            retToken.attribute = GT;
            return(retToken);

    } // end switch
    } // end while
}
```

Implementing a Transition Diagram

- Systematic approach for all transition diagrams
  - Program size $\propto$ number of states and edges
- We try each diagram and when we fail then we go to try the next diagram

- Transition diagram for WS should be placed at the beginning rather at the end
  - Generalize: frequently occurring tokens should come earlier

Finite State Automata (FSAs)

- **AKA "Finite State Machines", "Finite Automata", "FA"**
- Regular Expression = Specification
- Finite Automata = Implementation
- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state} \xrightarrow{\text{input}} \text{state}$
- Two types
  - Deterministic (DFA)
  - Non-deterministic (NFA)

Finite Automata

- Transition

$$S_1 \xrightarrow{a} S_2$$

- Is read

In state s_1 on input "a" go to state s_2

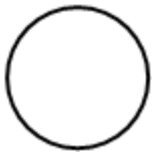
- If end of input
  - If in accepting state => accept,

othewise => reject

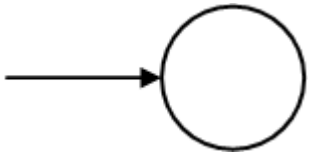
- If no transition possible => reject

Finite Automata State Graphs

- A state

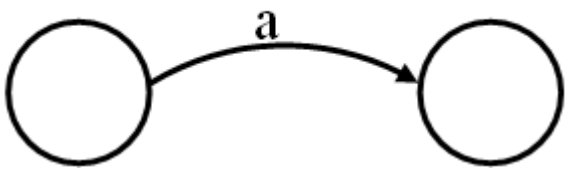


- The start state



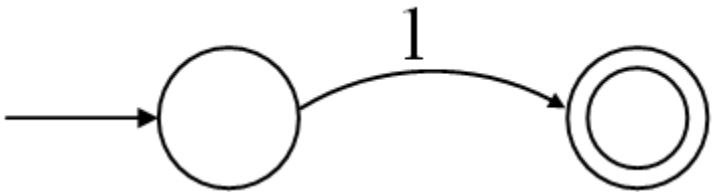
- An accepting state

- A transition



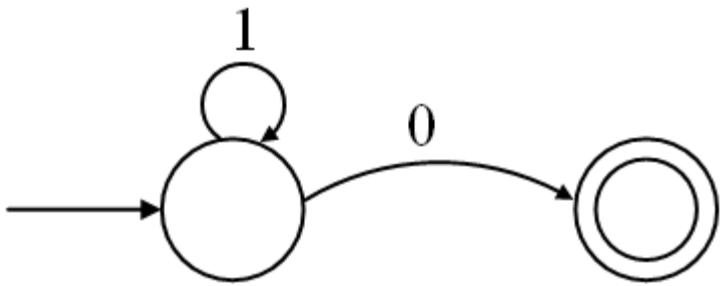
Example

- A finite automaton that accepts only “1”



- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

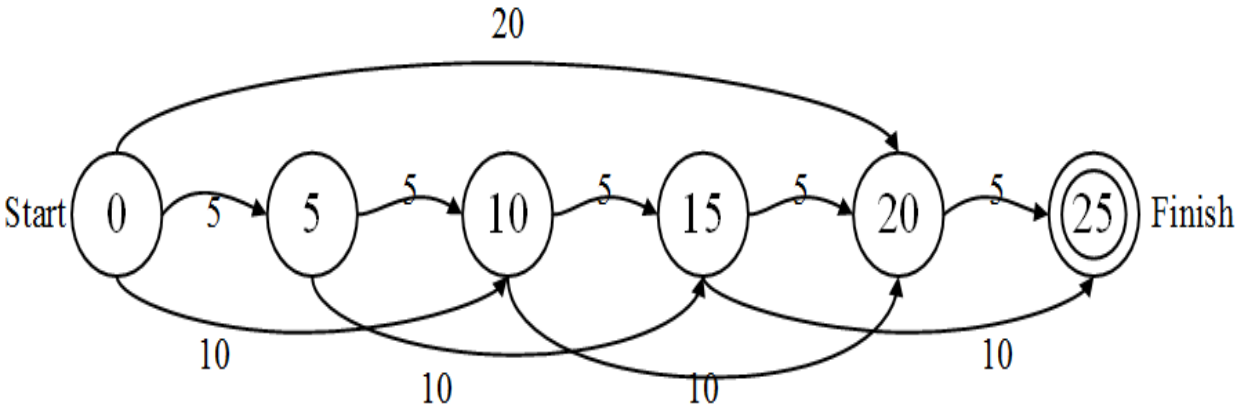
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



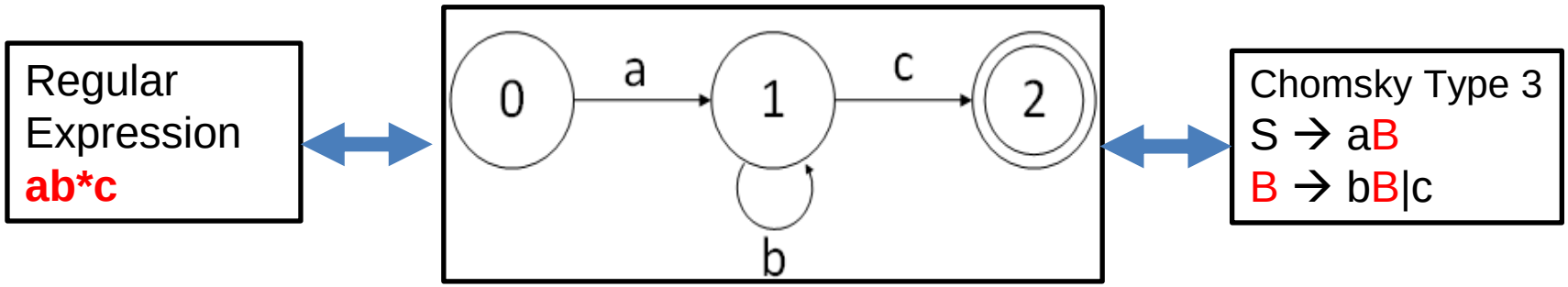
- 10
- 110
- 111110
- 111111110
- ~~1011~~ NOT ACCEPTED

Real Life example: Vending Machine

Electromechanical Vending Machine



- A simple vending machine
- If you give it 25 taka, it will vend you a bottle of coca cola.
- This machine can only accept 5tk, 10tk and 20tk.
- Consider it as a language to build up 25tk
- It can happen as {5,5,5,5,5}, {10,10,5} and many more.
- If you are on the 15 state, all the machine knows “It is on the 15 state”. If you ask the machine, “*how did you get here?*”, it will not be able to answer. It retains no memory of the coins it had to get to state 15.
- These are machines with **no need of memory**



- Input strings: ac
  abc
  abbc
  abbbc
  ....
- Kleen's Notation for this state diagram is: **ab*c**
  where, b* means zero or more occurrences
- Check how compact ab*c is than state diagram
- The automaton diagram, the regular expression and the Comsky notation they all are equivalent

Deterministic Finite Automata

A **deterministic finite automaton** (DFA) is a mathematical model that consists of

1. A set of states S
   - $S = \{s_0, s_1, \ldots, s_N\}$
2. A set of input symbols Σ
   - $\Sigma = \{a, b, \ldots\}$
3. A transition function that maps state/symbol pairs to a state:
   $S \times \Sigma \rightarrow S$
4. A special state s_0 called the start state
5. A set of states F (subset of S) of final states

INPUT: string

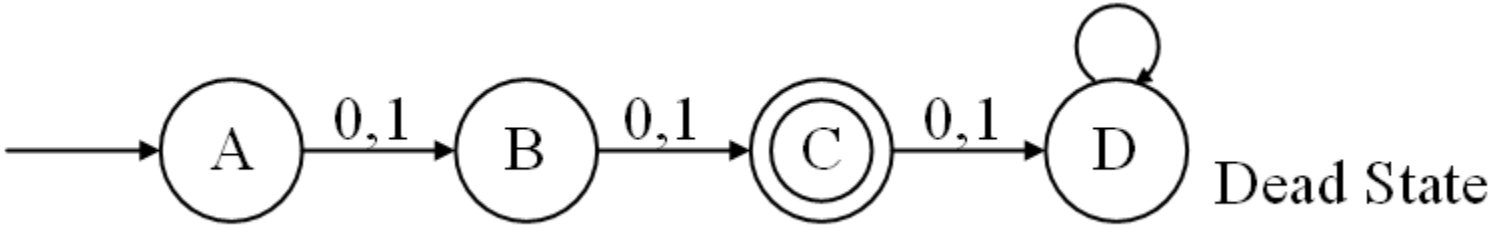
OUTPUT: yes or no

DFA Execution

```
DFA(int start_state) {
    state current = start_state;
    input_element = next_token();
    while (input to be processed) {
        current = transition(current,table[input_element])
        if current is an error state return No;
        input_element = next_token();
     }
     if current is a final state return Yes;
     else return No;
}
```

DFA Example

- Construct a DFA that accepts set of all strings over {0,1} of length 2



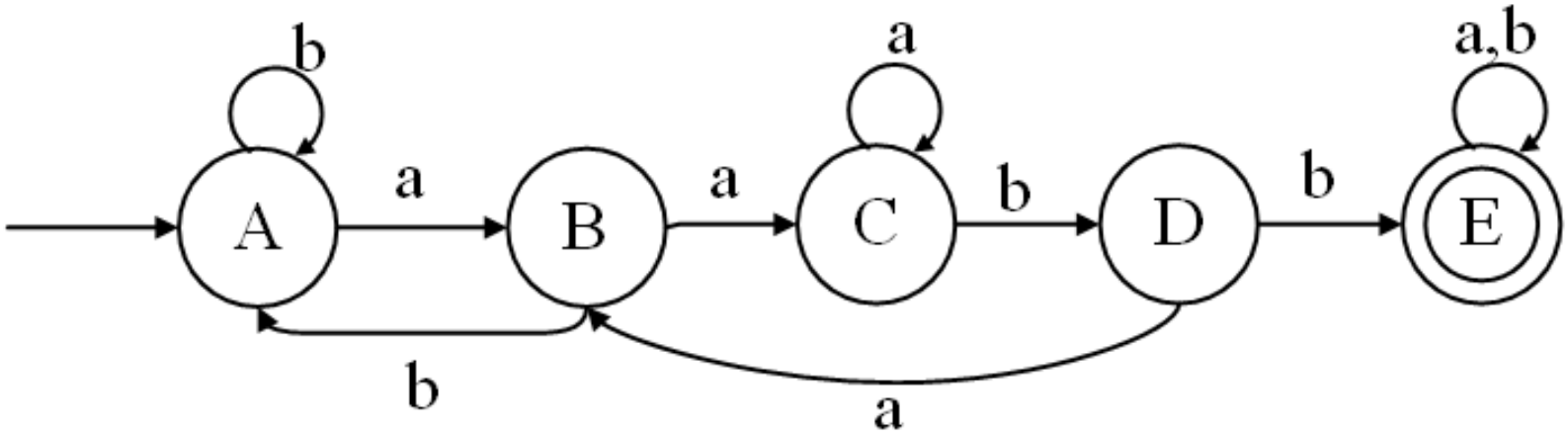
- Input String: 00

- Input String: 10

- Input String: 001

DFA Example

- Construct a DFA that accepts any string over {a,b} that does not contain the string ***aabb*** in it
- Alphabet {0,1}
- Let us try to design a simpler problem by constructing a DFA that accepts all strings over {a,b} that **contains** the string *aabb* in it



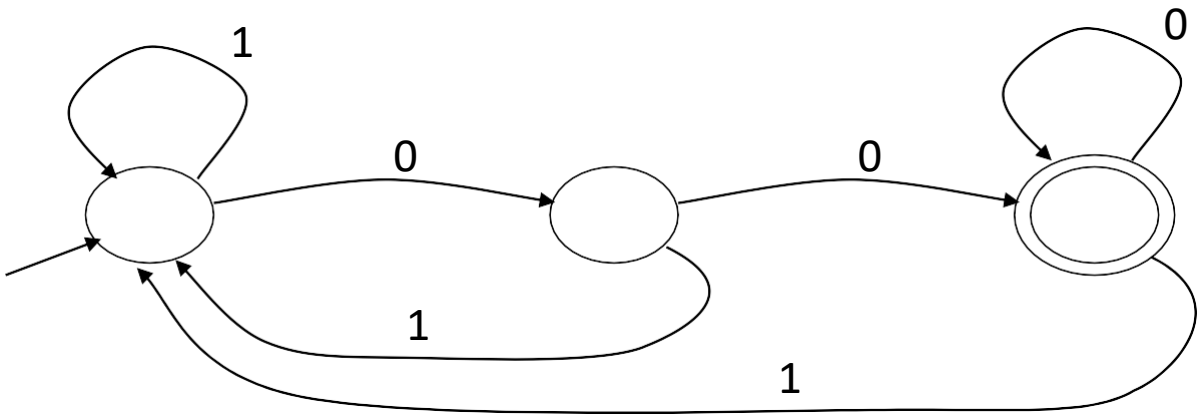
- This is the simplified DFA accepting all the string aabb

DFA Example

- To construct a DFA that accepts any string over {a,b} that does not contain the string ***aabb*** in it we need to flip the simplified DFA which accepts aabb
- Alphabet {0,1}
- Let us try to design a simpler problem by constructing a DFA that accepts all strings over {a,b} that **contains** the string *aabb* in it
- It means making the final state non-final and non-final state final

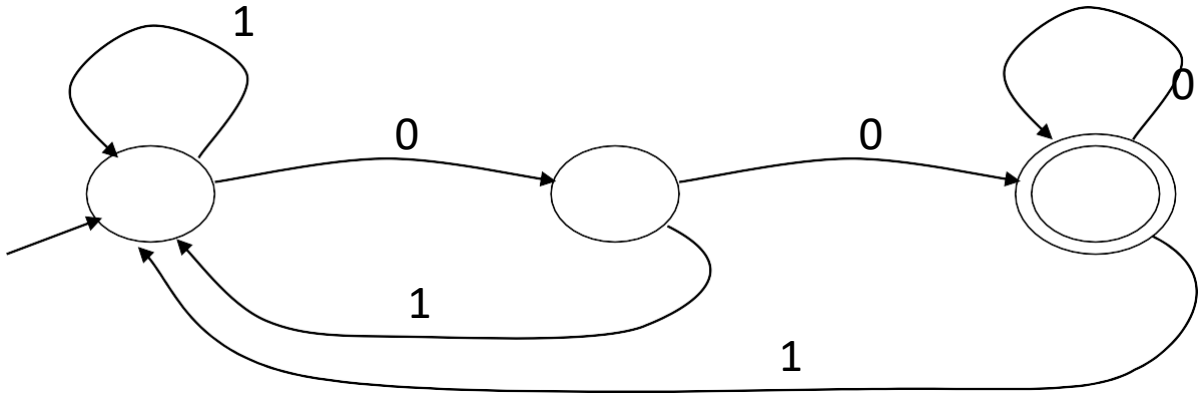
Another DFA Example

- Alphabet {0,1}
- What language does this recognize?



Solution

- Alphabet {0,1}
- What language does this recognize?

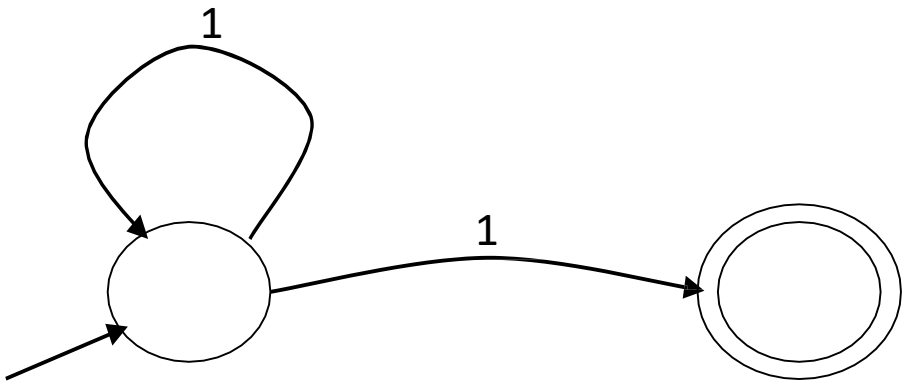


- 00
- 000
- 111100
- 1111001100
- **All strings terminated by at least two zeros**
- Regular Expression:

1*(0|1)*00
(0|1)*000*

A Different Example

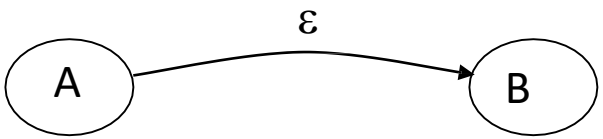
- Alphabet still { 0, 1 }



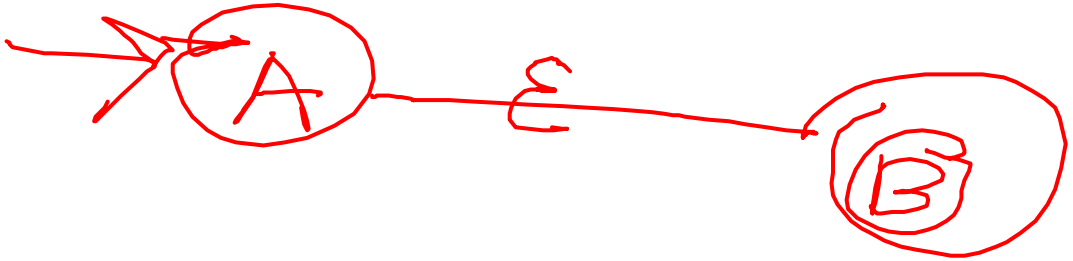
- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state
- This leads to Non-Determinism. It might seem like it is not correct, but this defines a new type of finite automata called **Non-Deterministic Finite Automata**
- In NFA, multiple transition will be possible from one state based on the same input symbol

Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input



Nondeterministic Finite Automata (NFA)

- In NFA, given the current state there could be multiple states
- The next state may be chosen random
- It can choose empty input (Epsilon moves)
- All the next state may be chosen in parallel

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

Dead State → DFA

Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input.
  - From one state it cannot got to two different states from the same input symbol
- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

DFA vs NFA

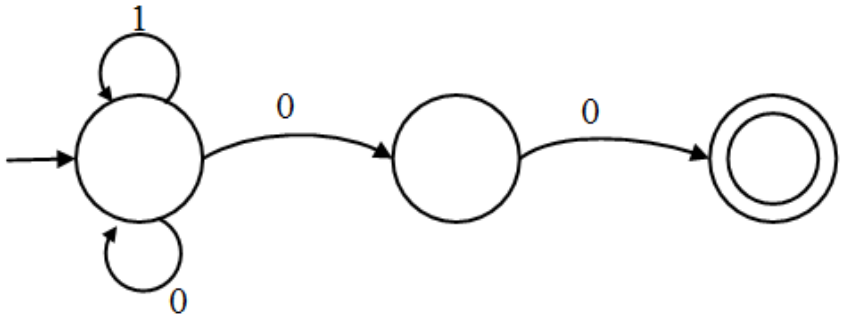
- Both DFA and NFA can recognize same set of languages (regular languages)
- But – time-space trade space exists
- DFAs are faster recognizers (easier to implement)
  - Can be much bigger too..
  - There are no choices to consider

DFA vs NFA

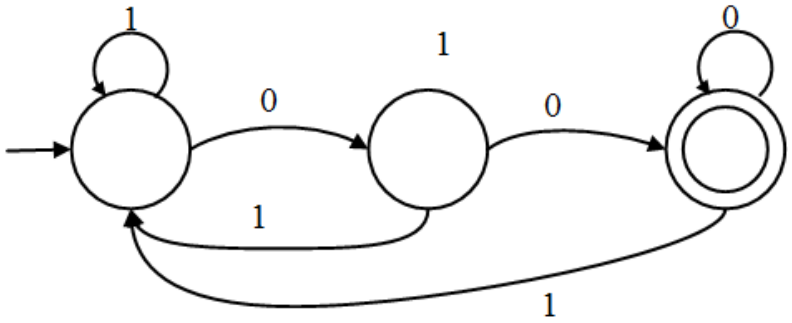
- For a given language the NFA can be simpler than the DFA
- **All binary strings ending with two zeros:**

1010100

NFA



DFA

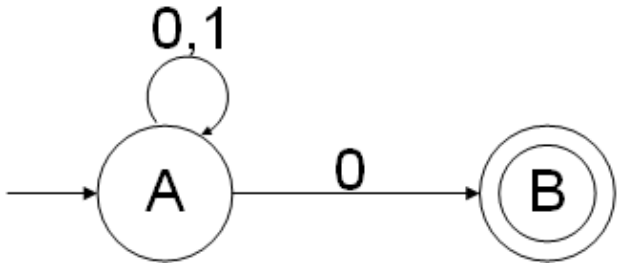


- DFA can be exponentially larger than NFA
- For the input string, the NFA can backtrack

NFA Example 1

- For a given language the NFA can be simpler than the DFA
- **All binary strings ending with ~~two~~ zeros:**

One



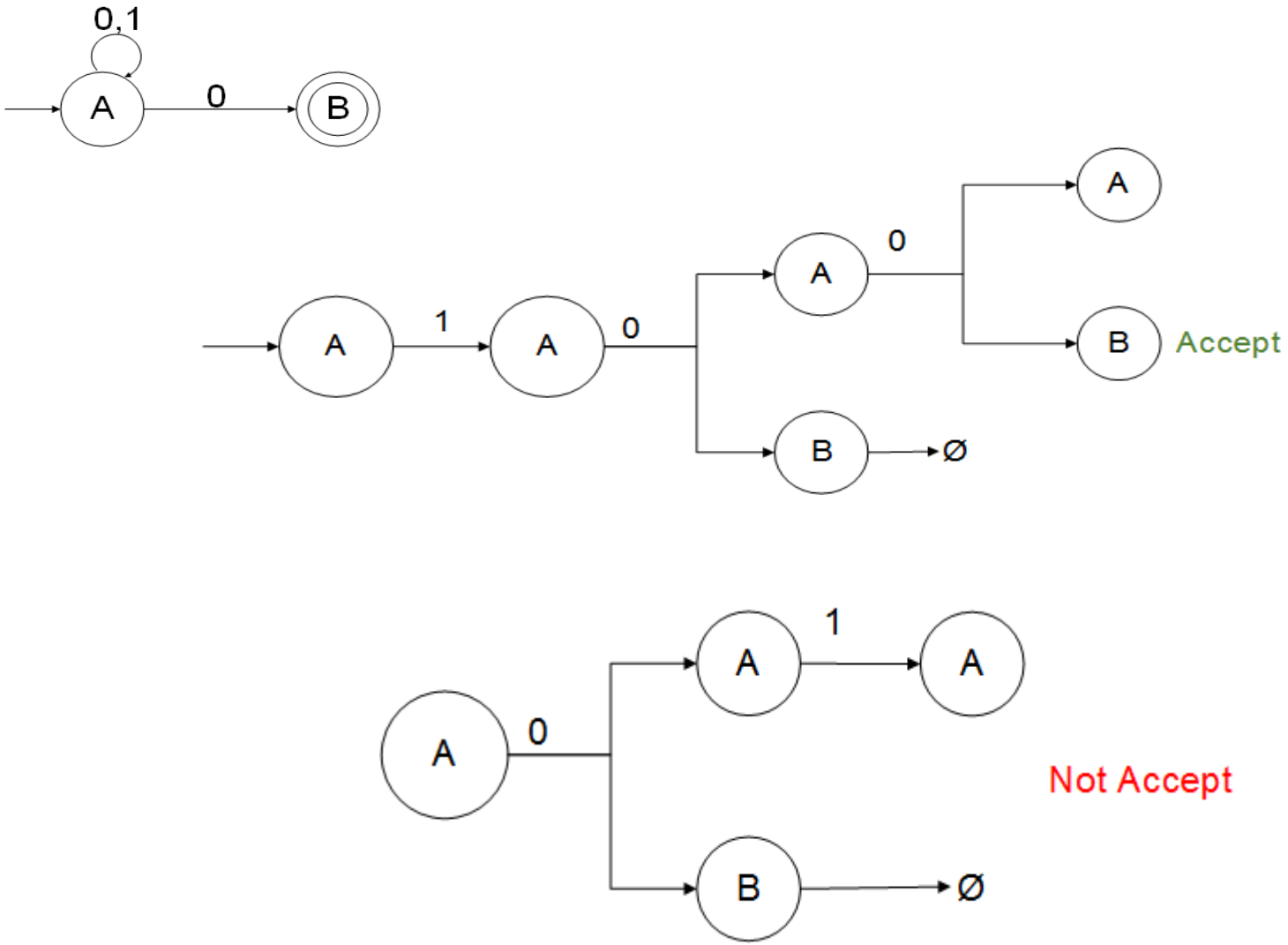
An input alphabet $\Sigma = \{0, 1\}$
A set of states S = A, B
A start state n= A
A set of accepting states F $\subseteq$ S = B

Transition Table:

STATE	0	1
A	A,B	A
B	ø	ø

Ø = Going nowhere

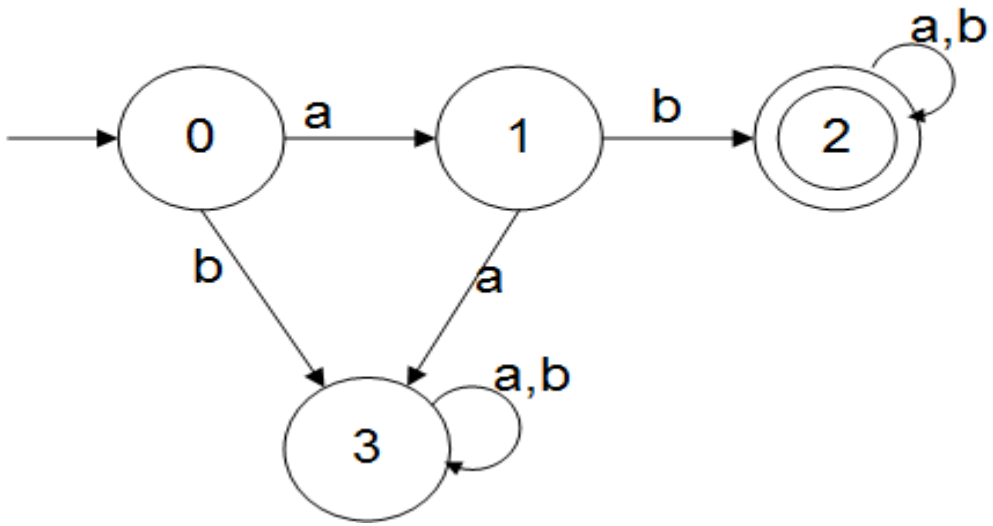
NFA Example 1



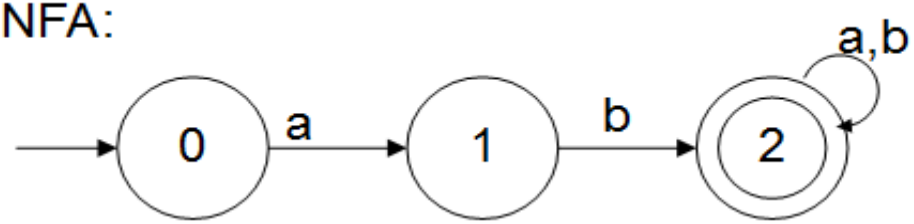
NFA Example 1

- **Design a NFA over an alphabet $\Sigma = \{a, b\}$ such that every string accepted must start with ab**

DFA:

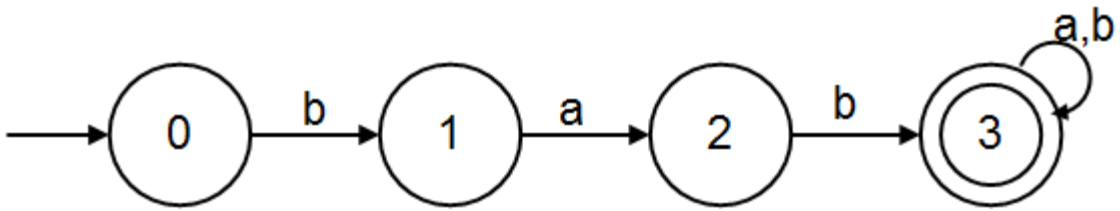


NFA:

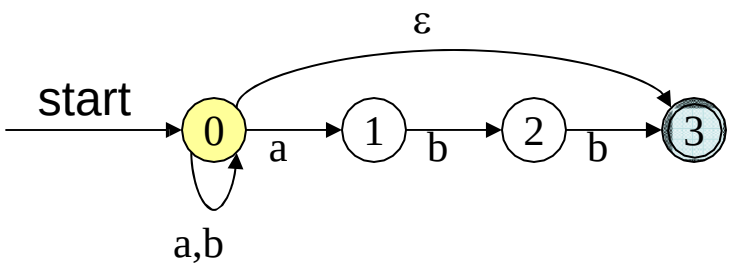


NFA Example 2

- **Design a NFA over an alphabet $\Sigma = \{a, b\}$ such that every string accepted must start with bab**



Example: NFA



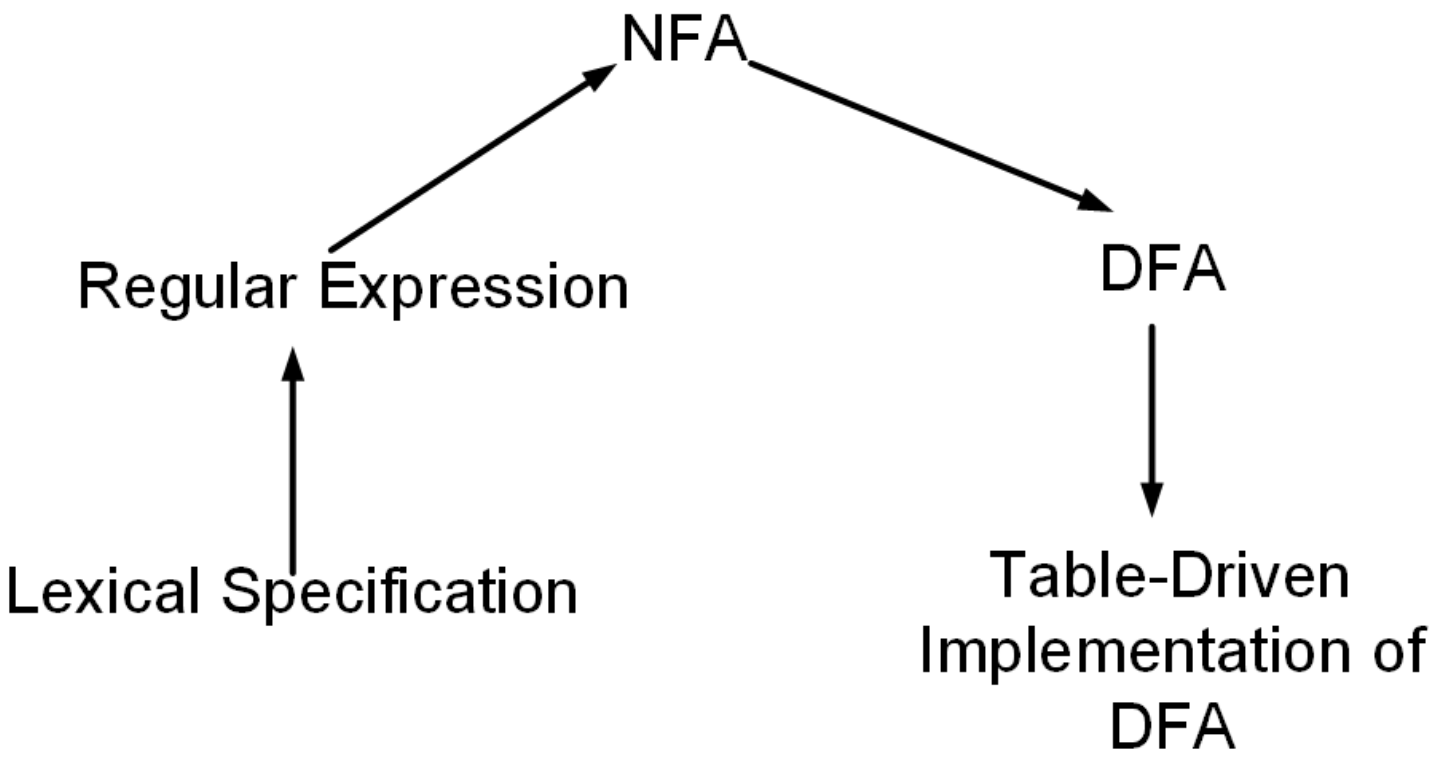
S = { 0,1,2,3}
S_0 = 0
Σ = {a,b}
F = {3}

Transition Table:

STATE	a	b	ε
0	0,1	0	3
1		2	
2		3	
3			

Regular Expression to Finite Automata

High Level Sketch:



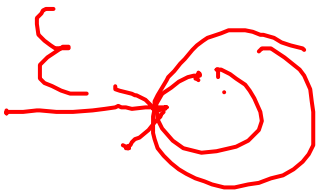
Relation between RE, NFA and DFA

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.

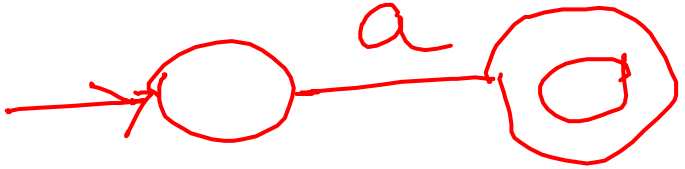
These facts tell us that REs, NFAs and DFAs have equivalent expressive power.

All three describe the class of regular languages.

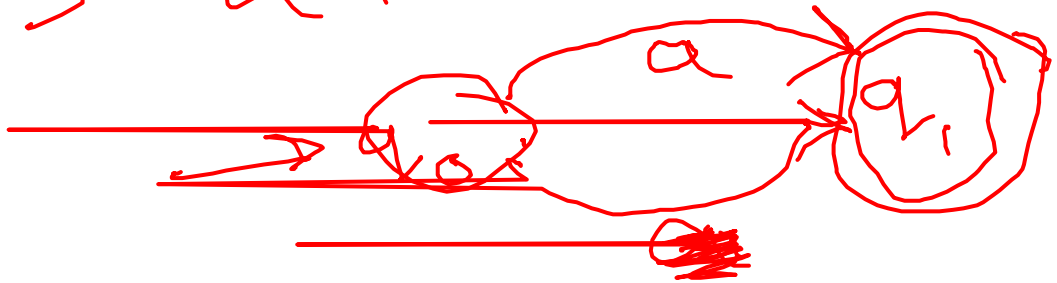
+ ε



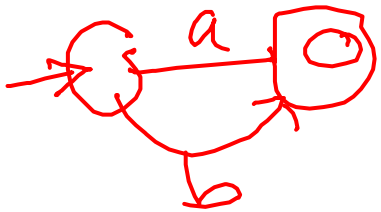
2. a



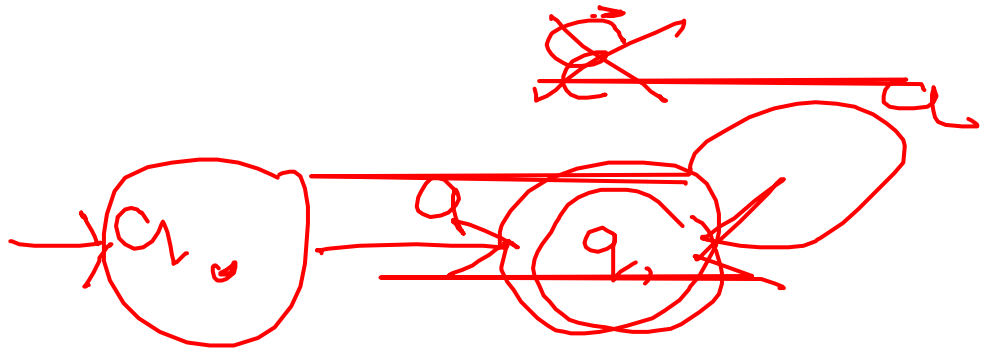
3. a + b →



a + b ≡ a|b



a +

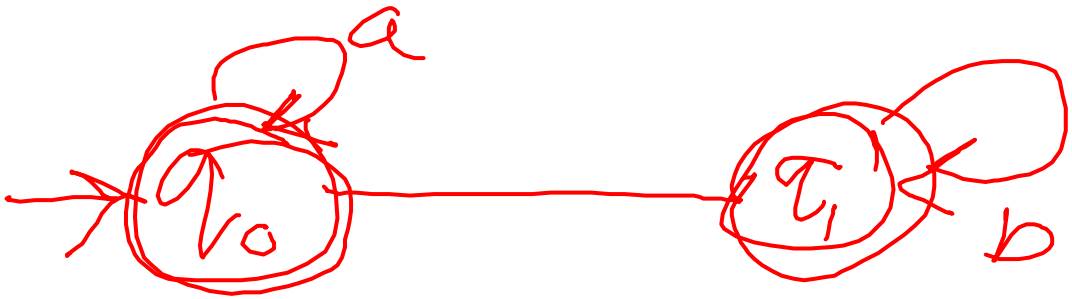


a*

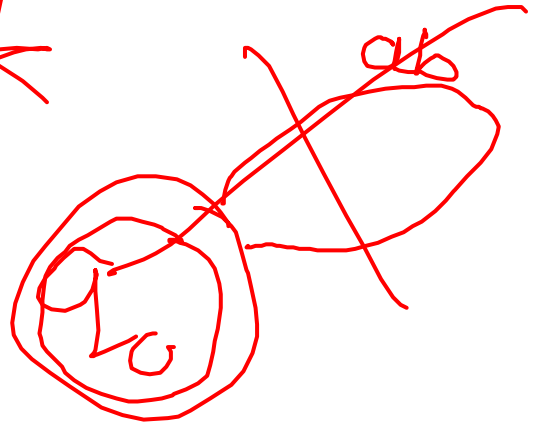
a . b

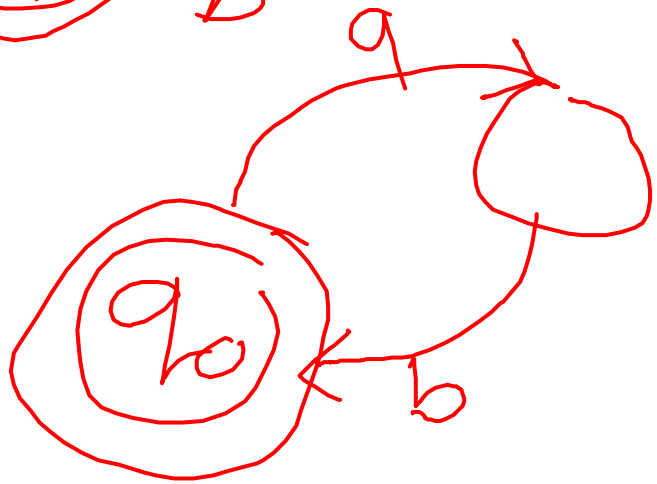
$(a+b)^*$ =

a^*b^* =

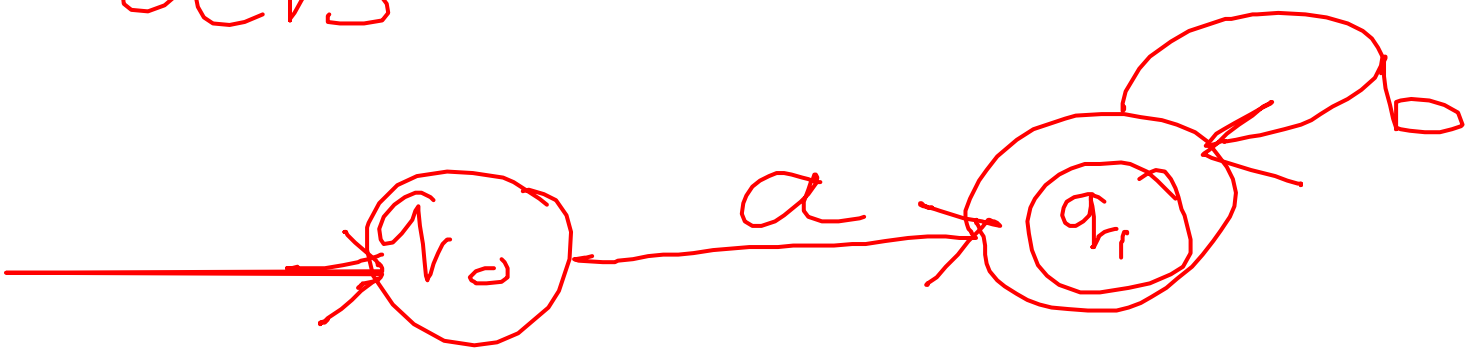


$(ab)^*$





ab^*



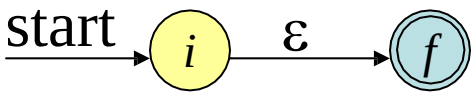
Converting Regular Expressions to NFAs

Thompson's Construction

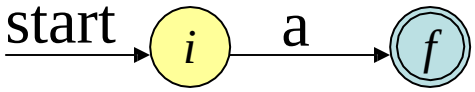
The **regular expressions** over finite Σ are the strings over the alphabet Σ + {), (, |, *} such that:

- { } (empty set) is a regular expression for the empty set

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



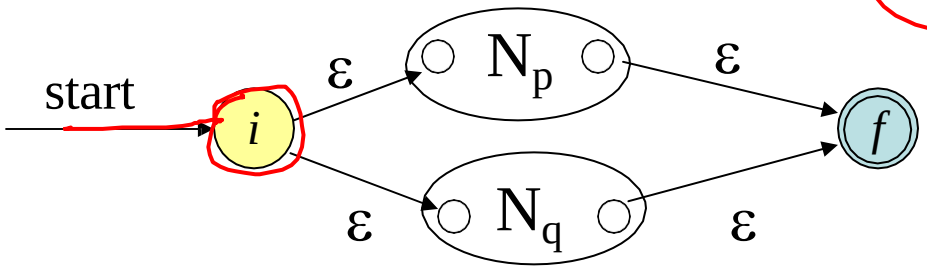
- *a* is a regular expression denoting {*a* } for any *a* in Σ



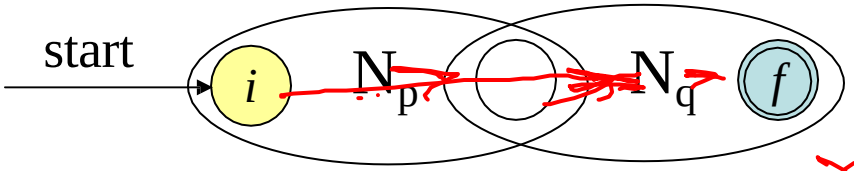
Converting Regular Expressions to NFAs

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

P | Q (union)



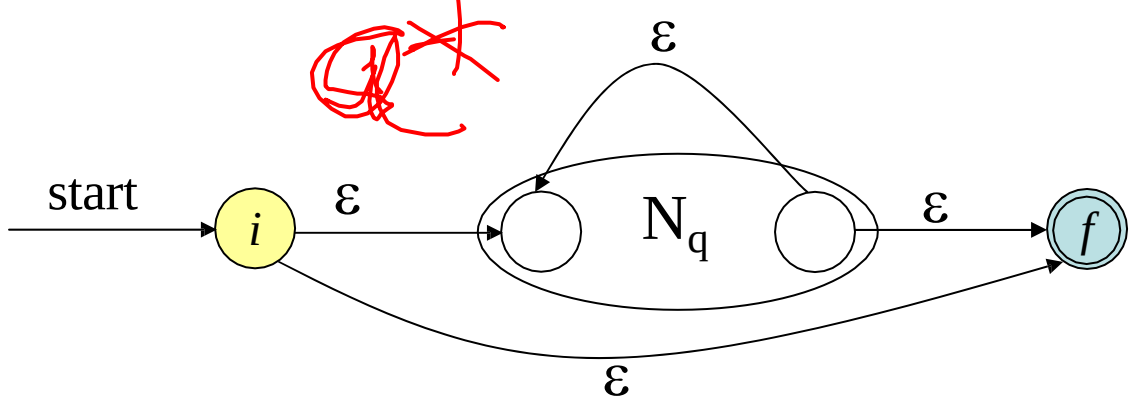
PQ (concatenation)

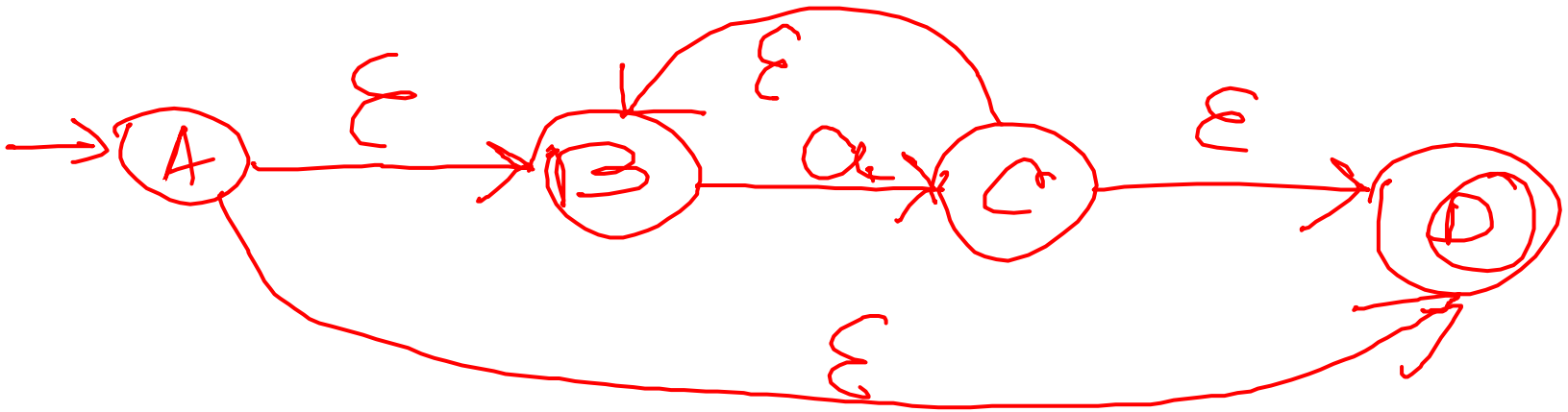


Converting Regular Expressions to NFAs

If Q is a regular expression with NFA N_q:

Q* (closure)

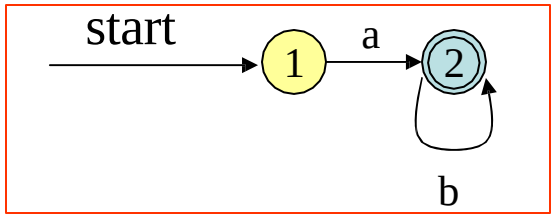




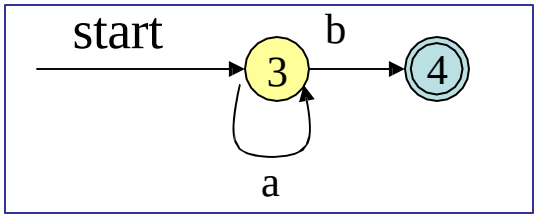
Example (ab* | a*b)*

Starting with:

ab*

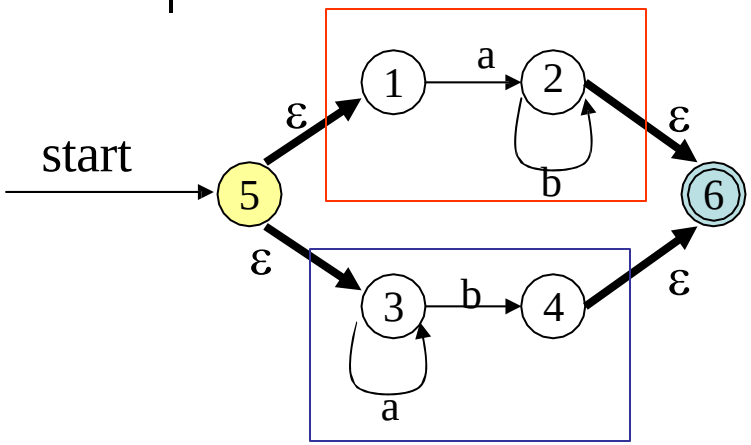


a*b

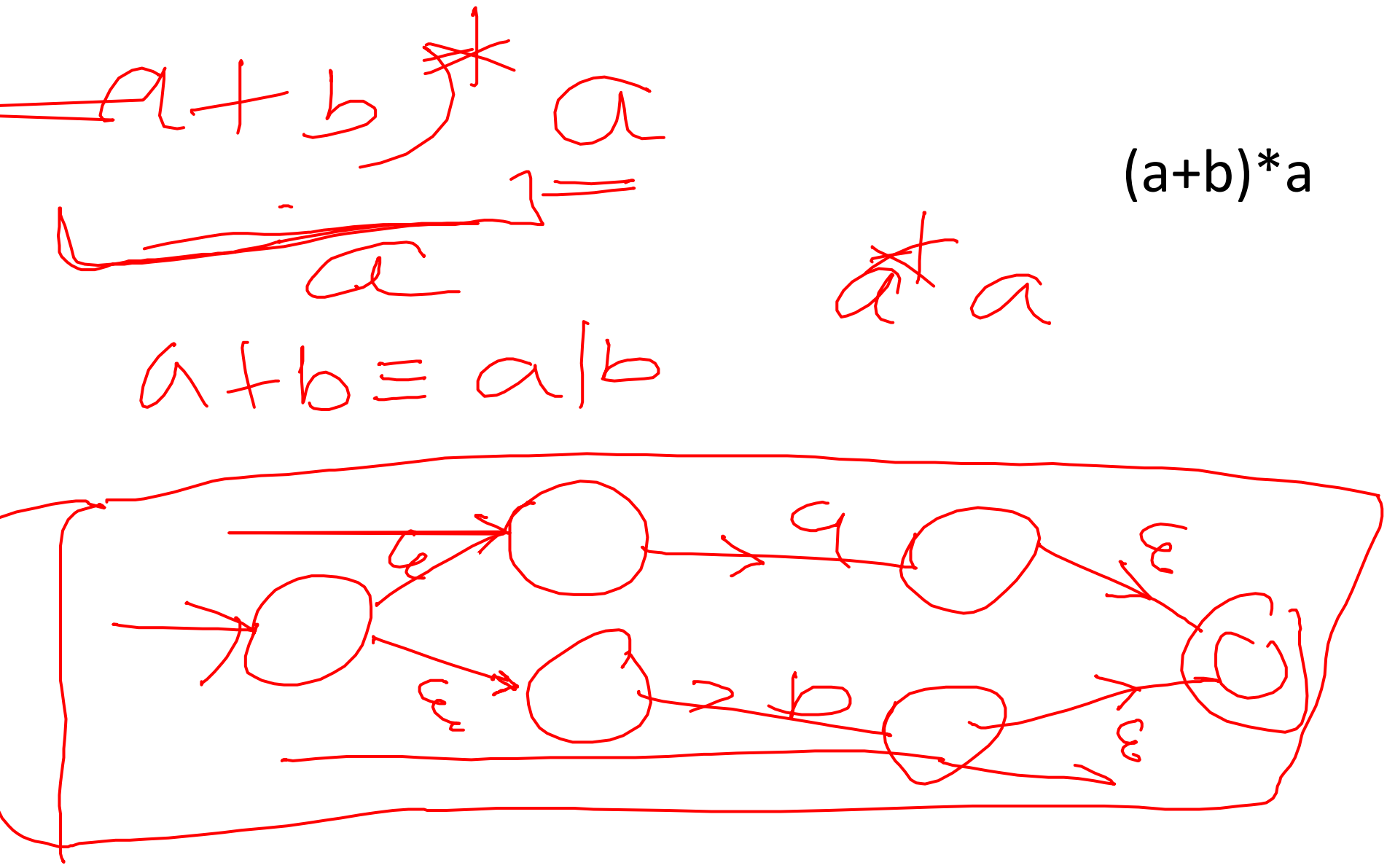


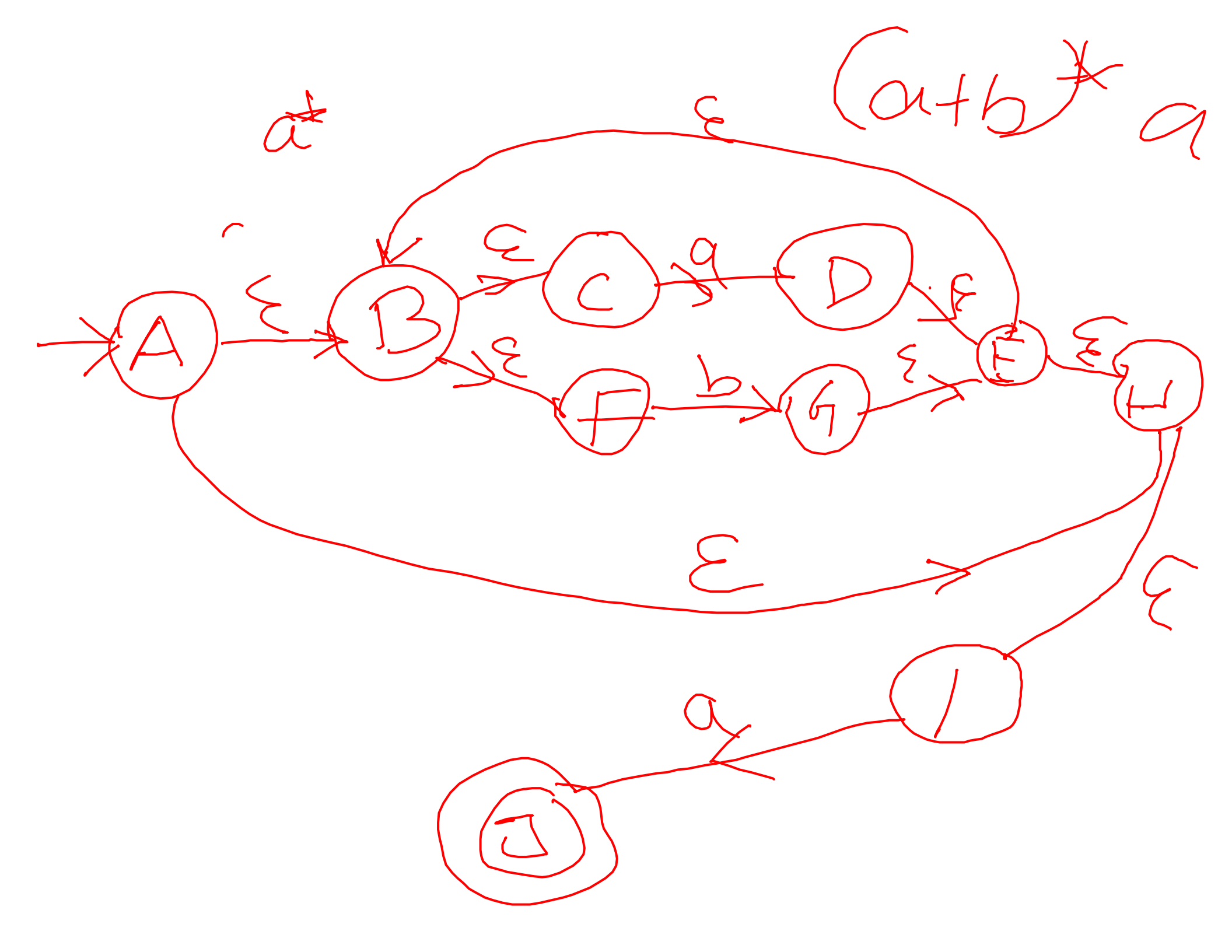
---

ab* | a*b



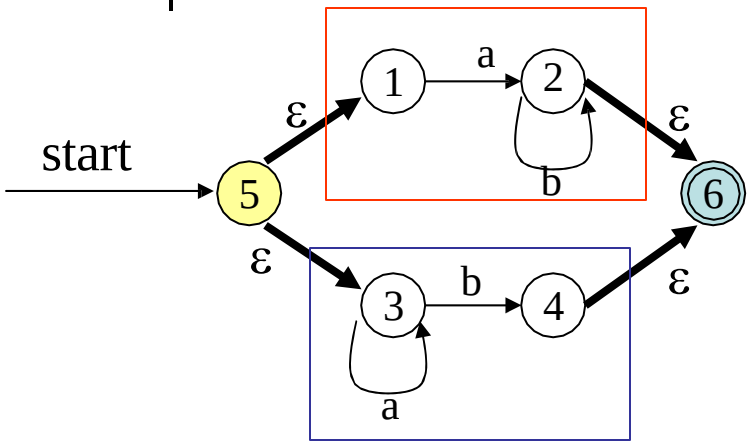
(a+b)*a



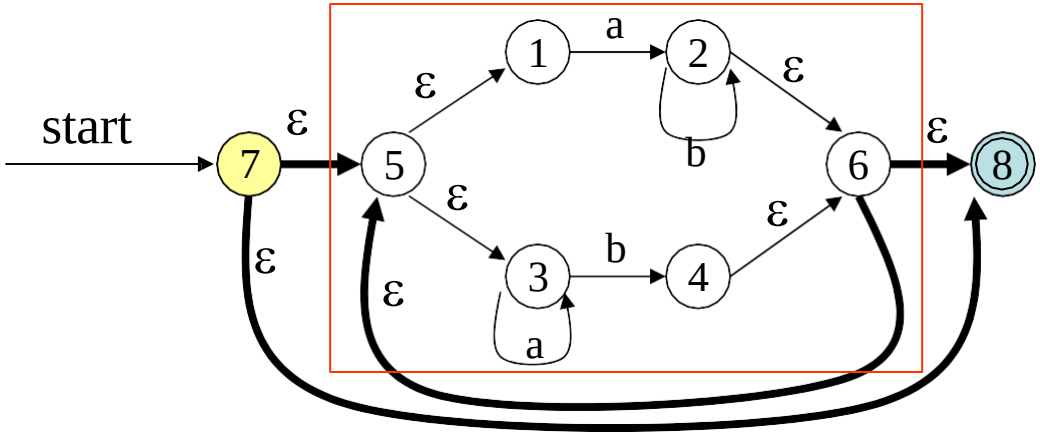

(a+b)*a
a*
ε
A
ε
B
ε
C
a
D
ε
E
ε
H
ε
ε
F
b
G
ε
ε
a
I
a
J

Example (ab* | a*b)*

ab* | a*b

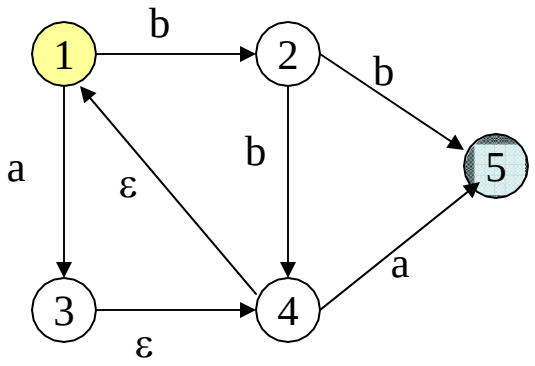


(ab* | a*b)*



Terminology: ε-closure

Defn: 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}

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: $S_D \times \Sigma \rightarrow S_D$

Algorithm: Computation of ε-closure

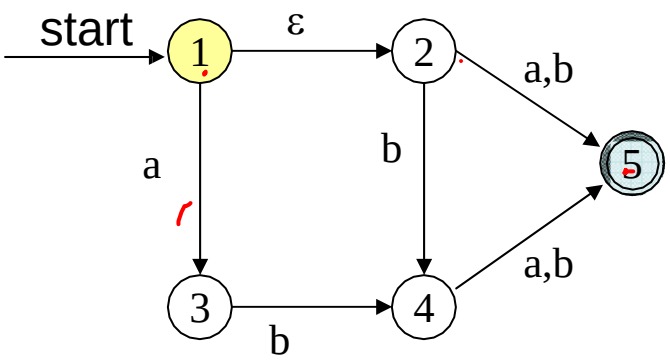
push all states a ∈ T onto stack STK
initialize: ε-closure(T) = T
while STK is not empty **do begin**
    pop *t*, the top element, off STK
    for each stat *u* with and edge from *t* to *u* labeled ε **do begin**
        if *u* is not in ε-closure(T) **do begin**
            add *u* to ε-closure(T)
            push *u* onto STK
        end if
    end for
end while

Algorithm: Subset Construction

s_D = ε-closure(s_N) -- create start state for DFA
S_D = {s_D} (unmarked)
while there is some unmarked state R in S_D
    mark state R
    for all a in Σ do
        s = ε-closure(T_N(R,a));
        if s not already in S_D then add it (unmarked)
        T_D(R,a) = s;
    end for
end while
F_D = any element of S_D that contains a state in F_N

Example 1: Subset Construction

NFA



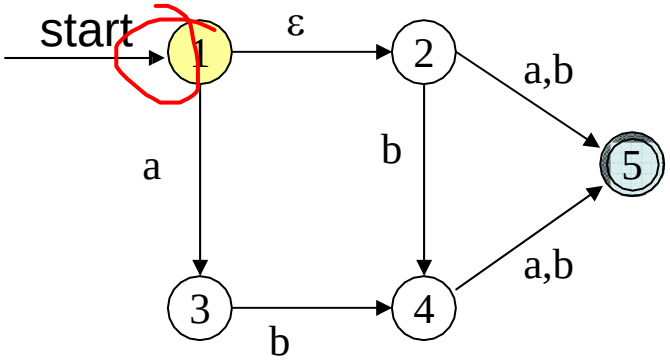
NFA N with

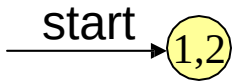
- State set S_N = {1,2,3,4,5},
- Alphabet Σ = {a,b}
- Start state s_N=1,
- Final states F_N={5},
- Transition function T_N: S_N x {$\Sigma \cup \varepsilon$} → S_N

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

Example 1: Subset Construction

NFA

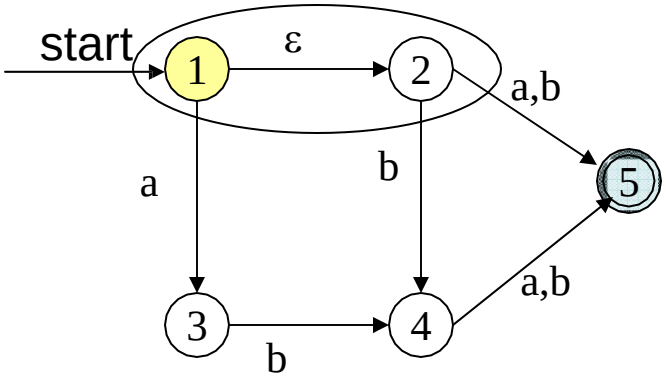


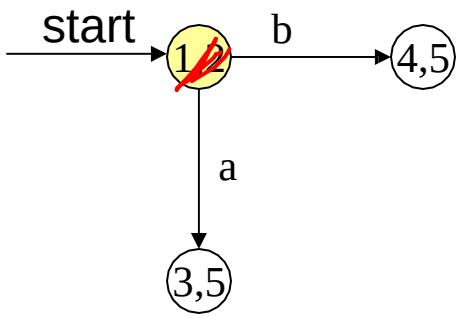


	a	b
{1,2}		

Example 1: Subset Construction

NFA

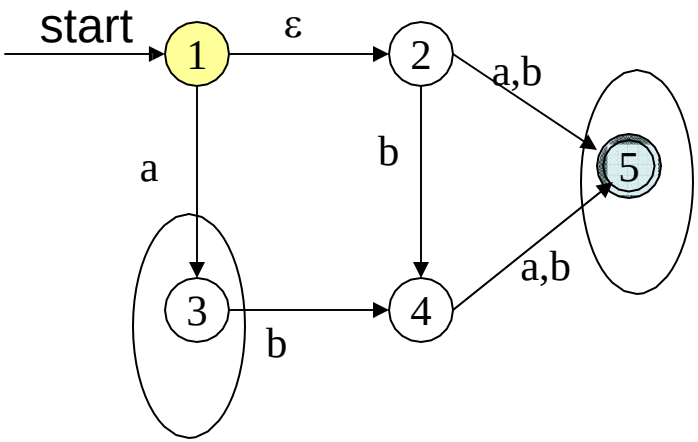


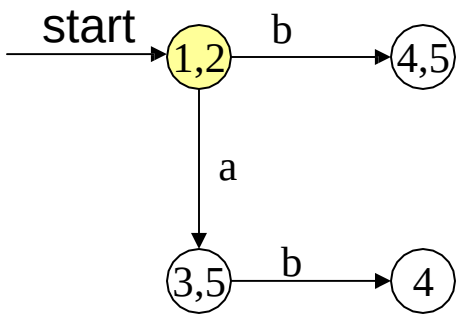


	a	b
{1,2}	{3,5}	{4,5}
{3,5}		
{4,5}		

Example 1: Subset Construction

NFA

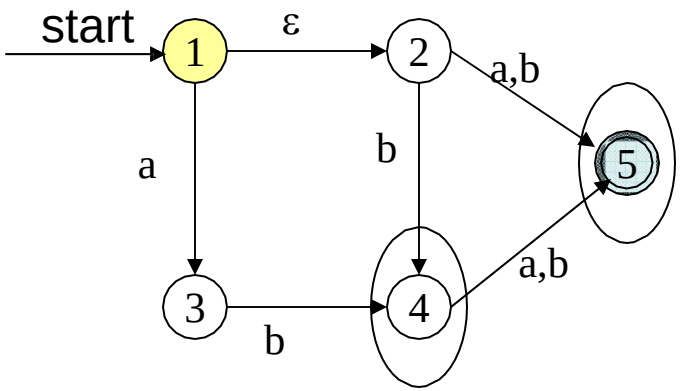


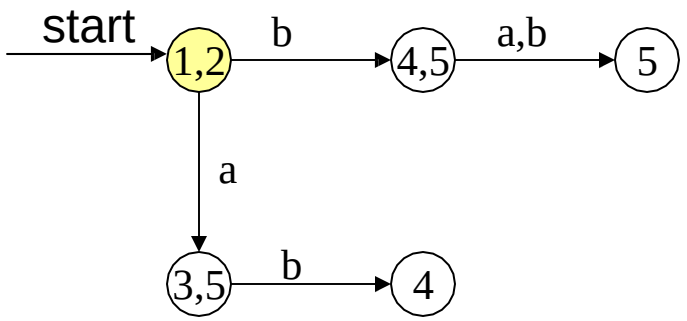


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}		
{4}		

Example 1: Subset Construction

NFA

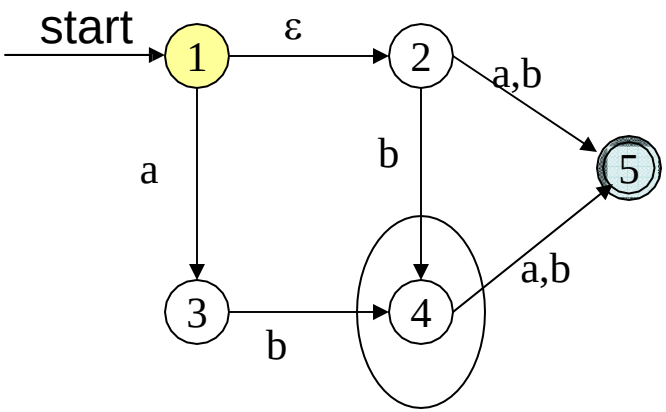


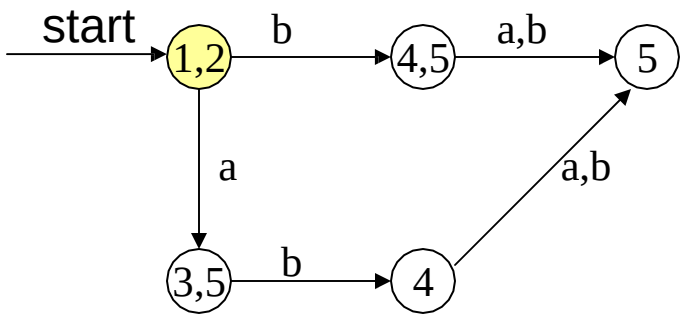


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}		
{5}		

Example 1: Subset Construction

NFA

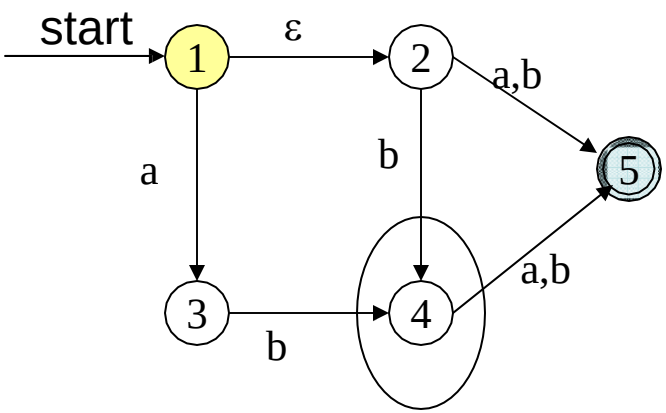


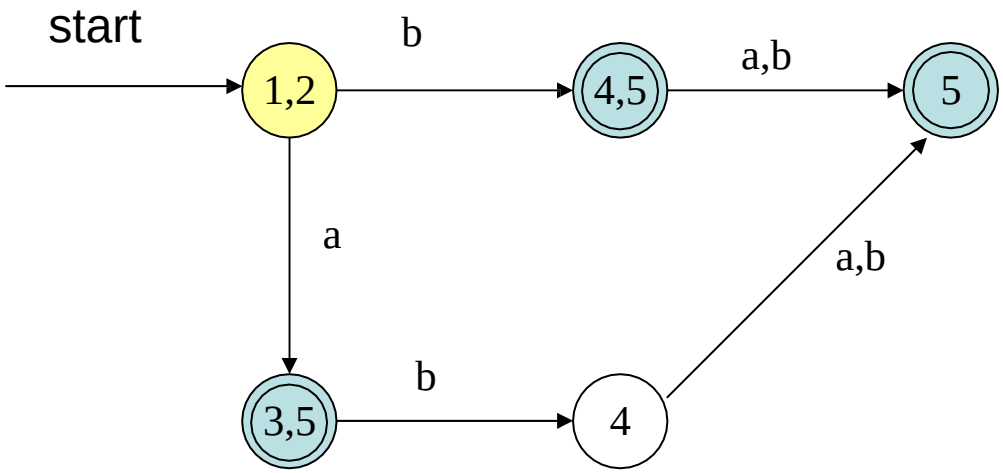


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

Example 1: Subset Construction

NFA



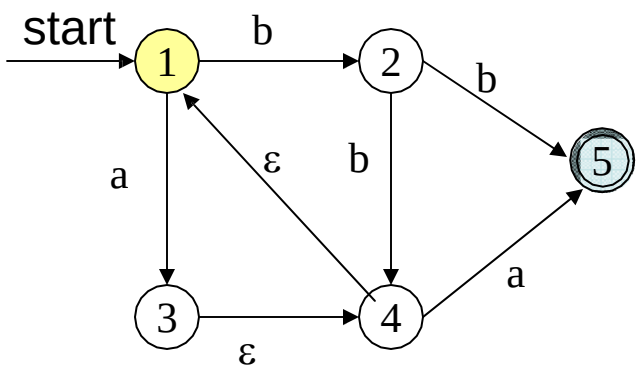


All final states since the NFA final state is included

	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

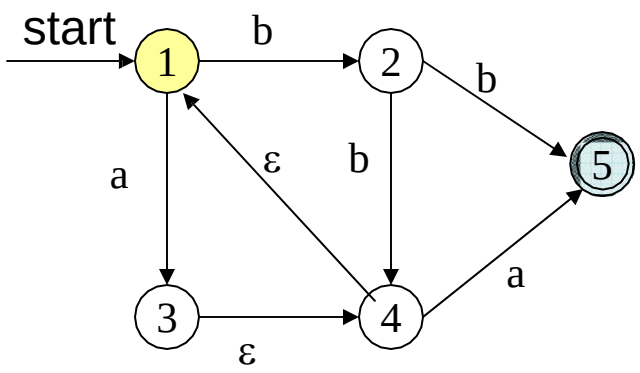
Example 2: Subset Construction

NFA

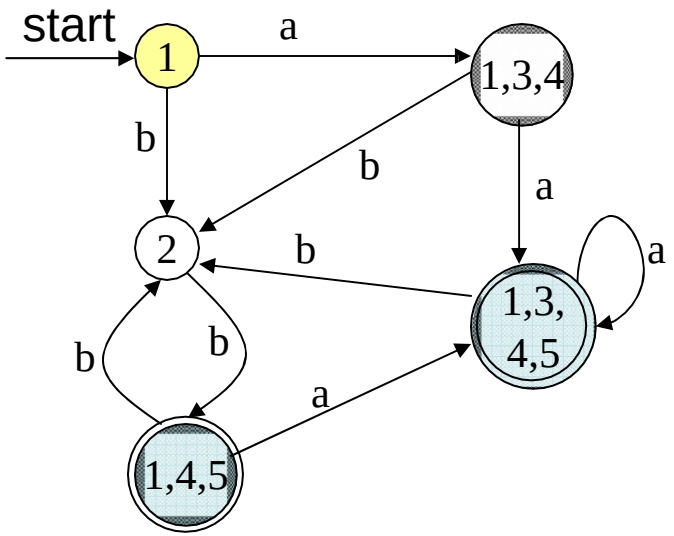


Example 2: Subset Construction

NFA

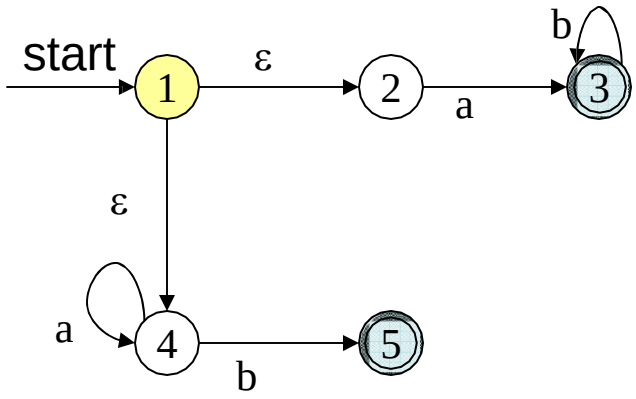


DFA

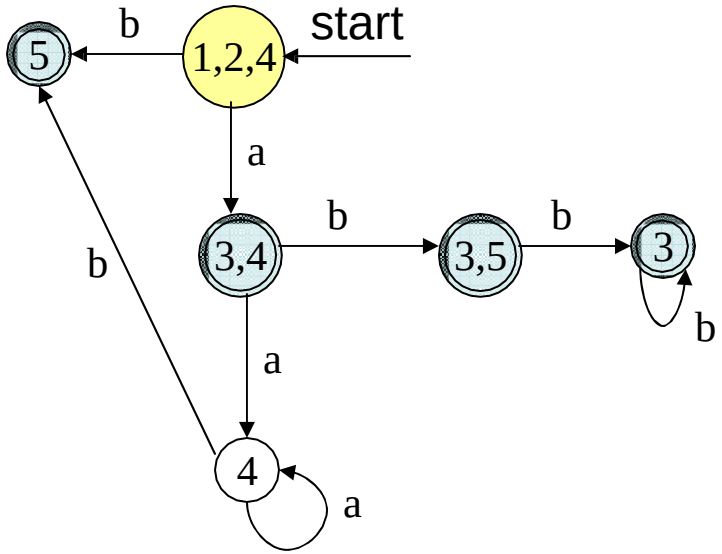


Example 3: Subset Construction

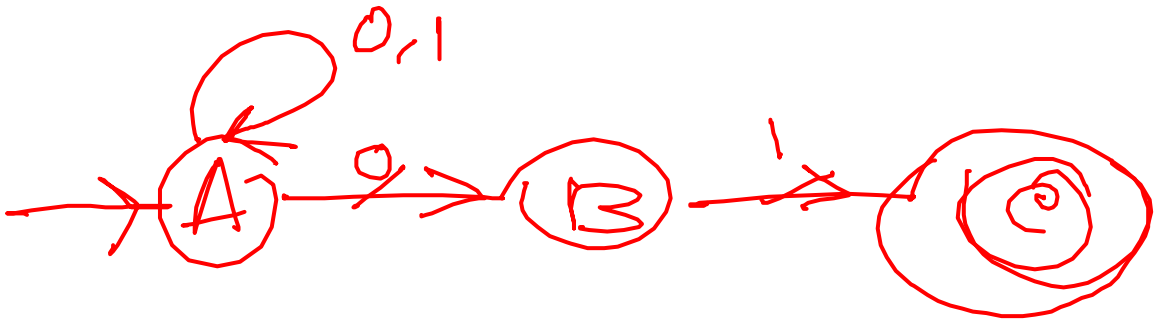
NFA



DFA



$\Sigma = 0, 1$



000 0111 01 =

01

1101

00101

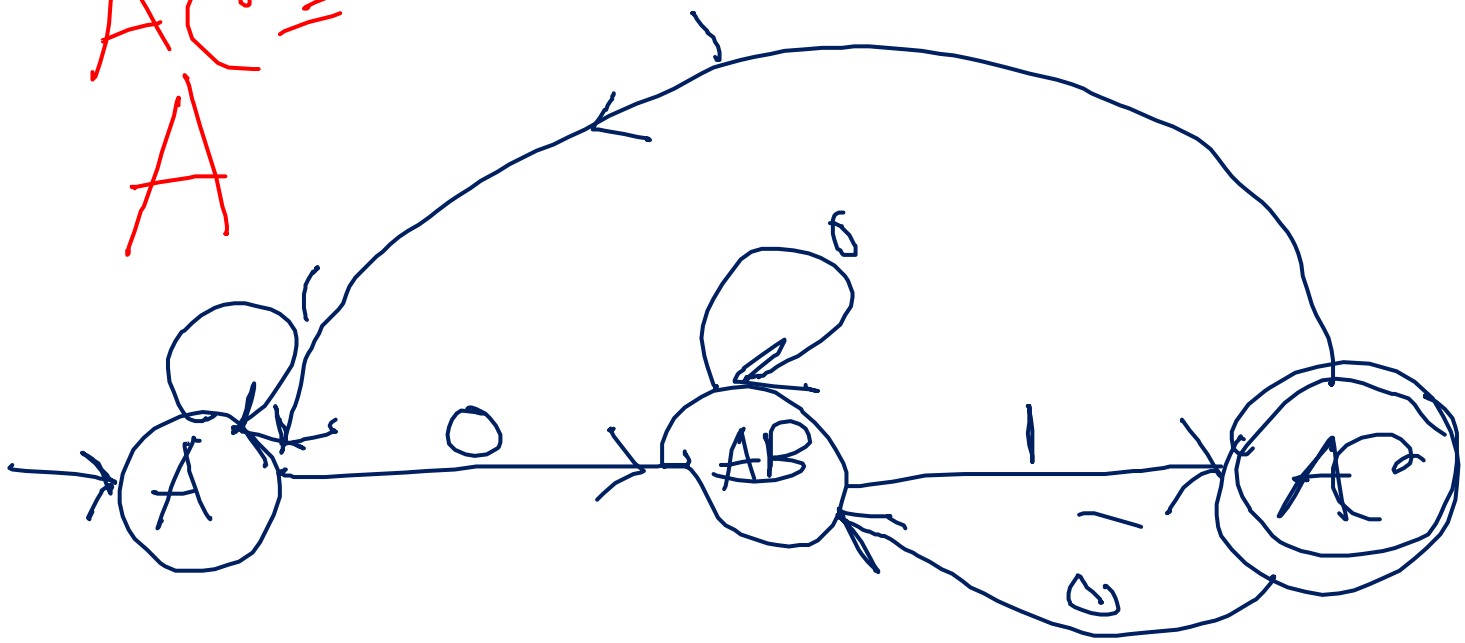
...

	0	1
→ A	AB	A
B	—	C
C	—	—

	0	0
A	AB	A
B	-	C
C	-	-

Dead State

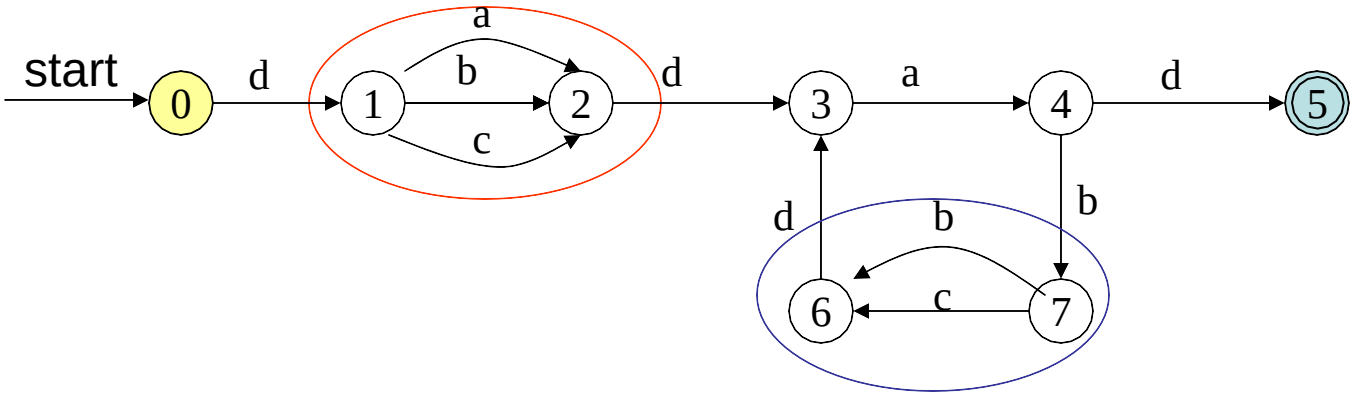
	0	1
→ A	AB	A
AB	AB	AC
A	AB	A



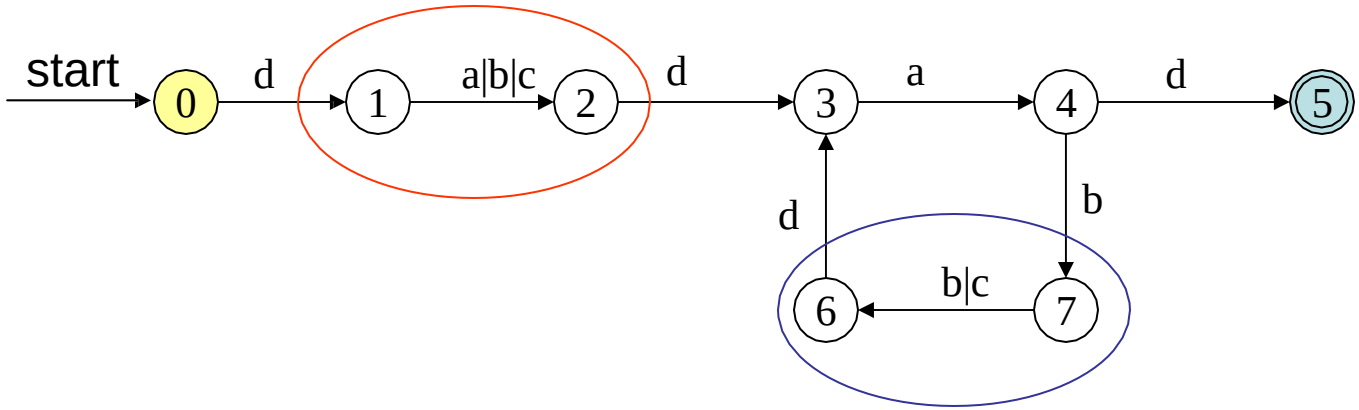
Converting DFAs to REs

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.

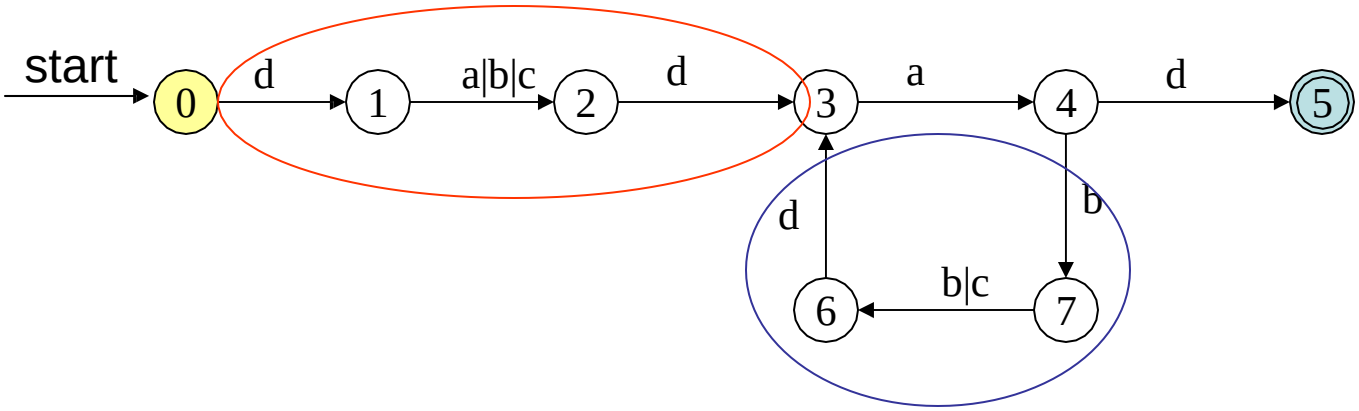
Example



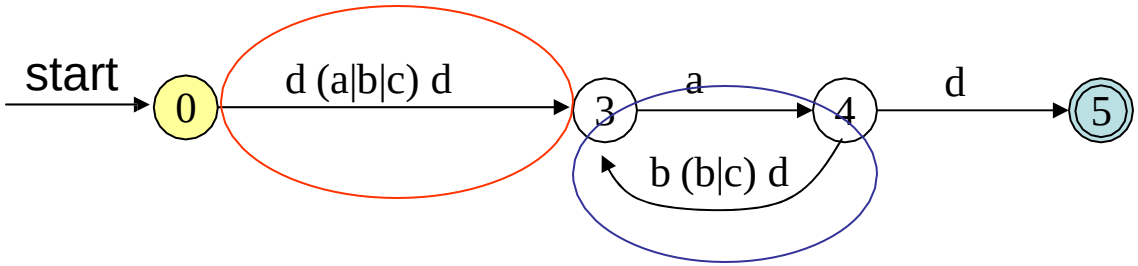
parallel edges become alternation



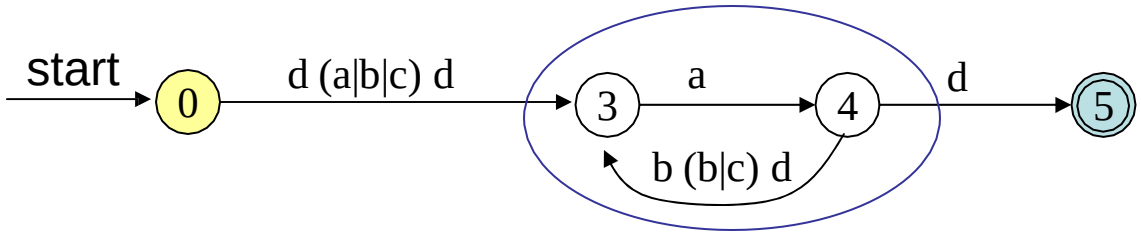
Example



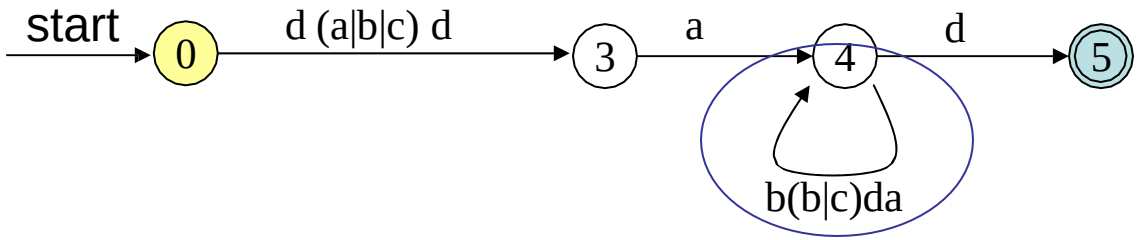
serial edges become concatenation



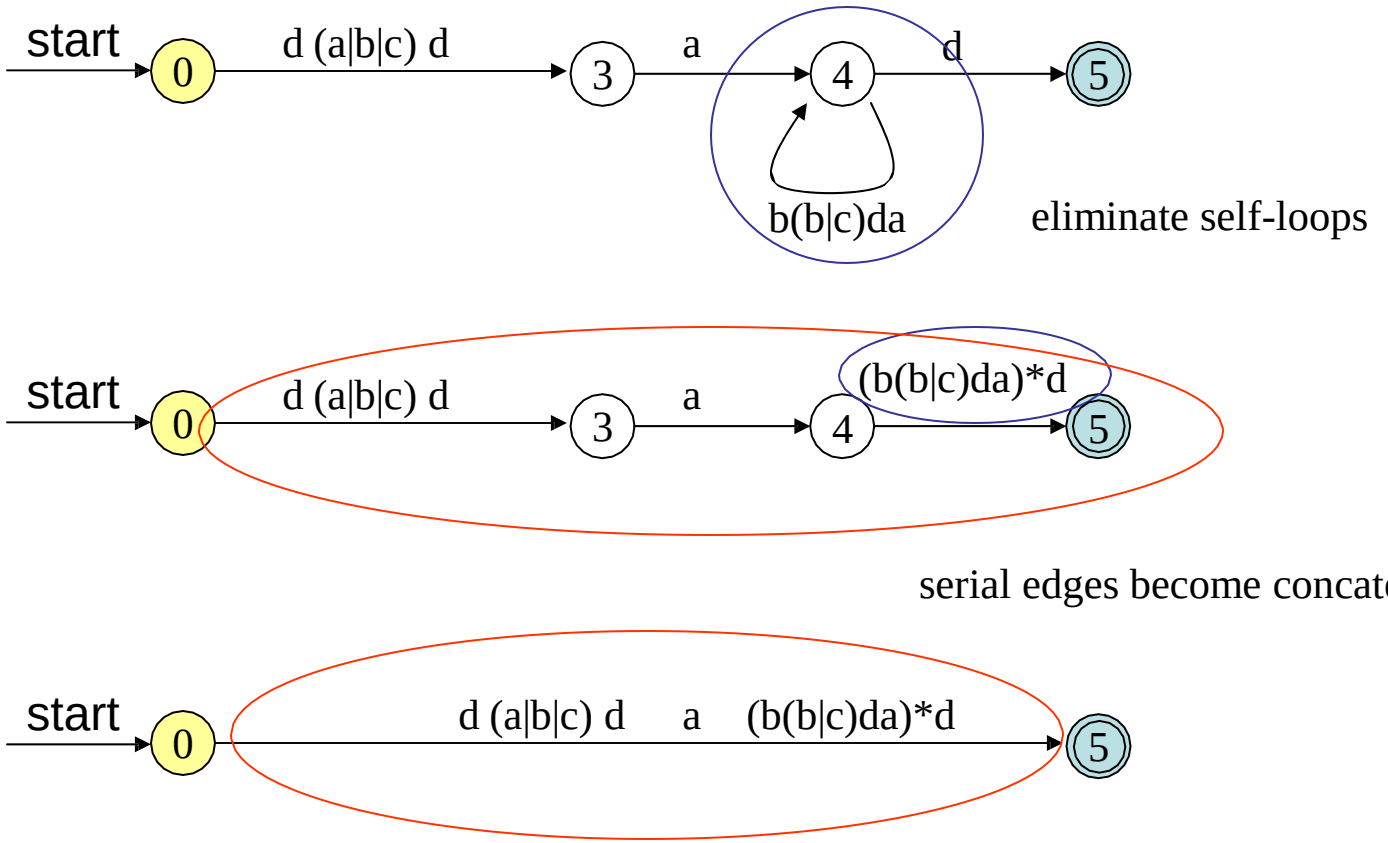
Example



Find paths that can be “shortened”



Example


start
0
d (a|b|c) d
3
a
4
d
5
b(b|c)da
eliminate self-loops
start
0
d (a|b|c) d
3
a
4
(b(b|c)da)*d
5
serial edges become concatenation
start
0
d (a|b|c) d
a
(b(b|c)da)*d
5

Describing Regular Languages

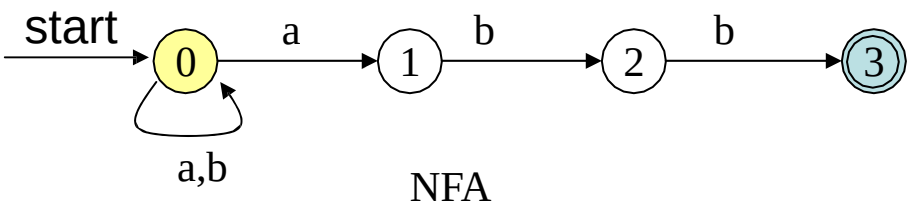
- Generate ***all*** strings in the language
- Generate ***only*** strings in the language

Try the following:

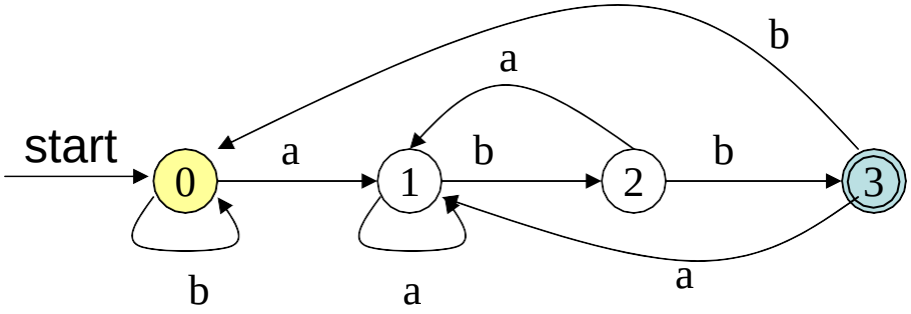
- Strings of {*a*,*b*} that end with '*abb*'
- Strings of {*a*,*b*} where every *a* is followed by at least one *b*

Strings of (a|b)* that end in abb

re: (a|b)*abb



NFA



DFA

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. (May have exponentially as many states, but...)

Suggestions for writing NFA/DFA/RE

- Typically, one of these formalisms is more natural for the problem. Start with that and convert if necessary.
- In DFAs, each state typically captures some partial solution
- Be sure that you include all relevant edges (ask – does every state have an outgoing transition for all alphabet symbols?)

Non-Regular Languages

Not all languages are regular”

- The language *ww* where *w*=(a|b)*

Non-regular languages cannot be described using REs, NFAs and DFAs.