Theory of Computation

DEPARTMENT OF ISE

R V College of Engineering

Dr.Anala M R

Professor

Department of ISE

R V College of Engineering

Introduction to Automata Theory

Unit 1

What is Automata Theory?

- *Study of abstract computing devices, or "machines"*
- Automaton = an abstract computing device
  - Note: A "device" need not even be a physical hardware!
- A fundamental question in computer science:
  - Find out what different models of machines can do and cannot do
  - The *theory of computation*
- Computability vs. Complexity

Alphabet

An alphabet is a finite, non-empty set of symbols

- We use the symbol ∑ (sigma) to denote an alphabet
- Examples:
  - Binary: ∑ = {0,1}
  - All lower case letters: ∑ = {a,b,c,..z}
  - Alphanumeric: ∑ = {a-z, A-Z, 0-9}
  - Digits:∑ = { 0-9}

Strings

A string or word is a finite sequence of symbols chosen from ∑

- ***Empty string is ε (or "epsilon")***

- Length of a string *w,* denoted by "|*w*|", is equal to the *number of (non-ε) characters in the string*
  - *E.g., x = 010100* *|x| = 6*
  - *x = 01 ε 0 ε 1 ε 00 ε* *|x| = ?*

- *xy* = concatentation of two strings *x* and *y*

Powers of an alphabet

Let $\sum$ be an alphabet.

- $\sum^k$ = the set of all strings of length k
- $\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \ldots$
- $\sum^+ = \sum^1 \cup \sum^2 \cup \sum^3 \cup \ldots$

Languages

*L is a said to be a language over alphabet ∑, only if L ⊆ ∑**

➔ this is because ∑* is the set of all strings (of all possible length including 0) over the given alphabet ∑

Examples:

1. Let L be *the* language of all strings consisting of *n* 0's followed by *n* 1's:
   L = {ε, 01, 0011, 000111,…}
2. Let L be *the* language of all strings of with equal number of 0's and 1's:
   L = {ε, 01, 10, 0011, 1100, 0101, 1010, 1001,…}

Definition: Ø denotes the Empty language

- Let L = {ε}; Is L=Ø? NO

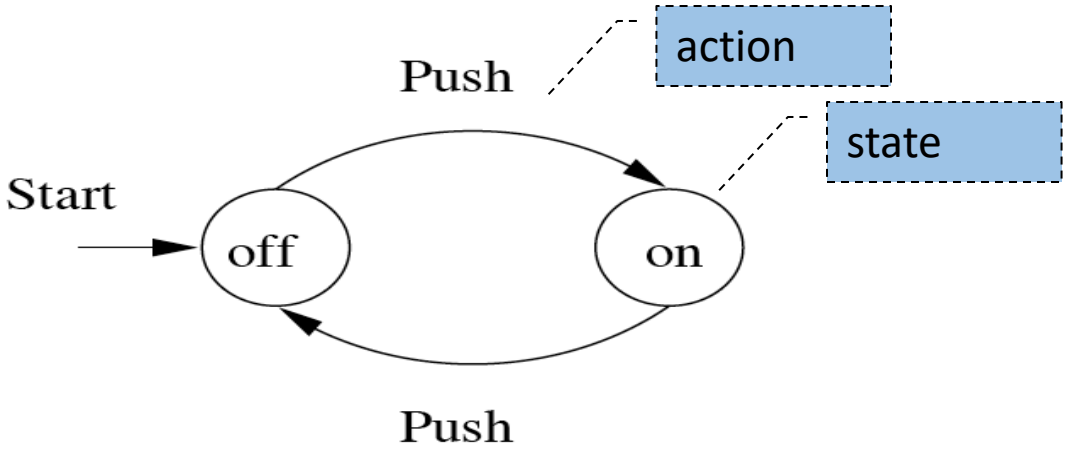
Finite Automata-Applications

- Software for designing and checking the behavior of digital circuits
- Lexical analyzer of a typical compiler
- Software for scanning large bodies of text (e.g., web pages) for pattern finding
- Software for verifying systems of all types that have a finite number of states

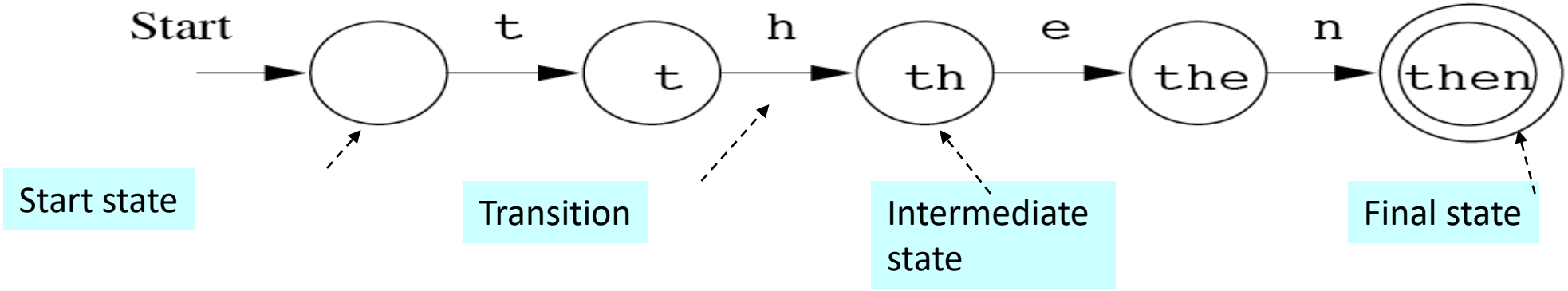
  (e.g., stock market transaction, communication/network protocol)

Finite Automata : Examples

- On/Off switch



- Model to recognize word "*then*"



Finite Automaton (FA)

- A state diagram that comprehensively captures all possible states and transitions that a machine can take while responding to a stream or sequence of input symbols

- ***Recognizer for "Regular Languages"***

- Deterministic Finite Automata (DFA)
  - The machine can exist in only one state at any given time
- Non-deterministic Finite Automata (NFA)
  - The machine can exist in multiple states at the same time

Deterministic Finite Automata - Definition

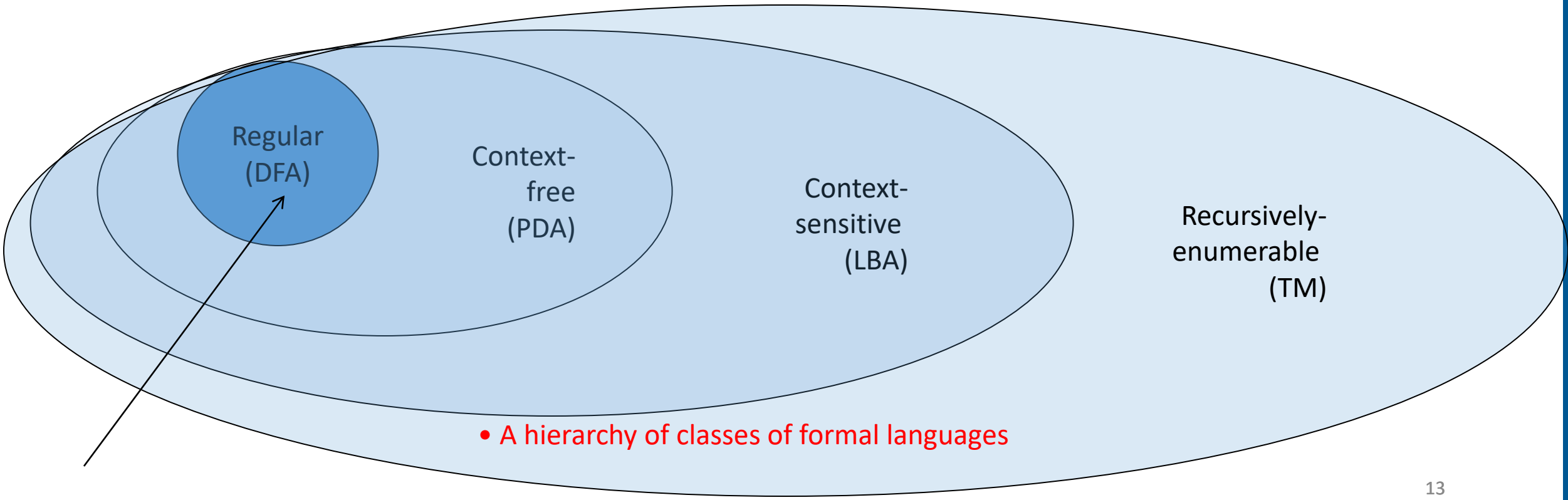
- A Deterministic Finite Automaton (DFA) consists of:
  - Q ==> a finite set of states
  - ∑ ==> a finite set of input symbols (alphabet)
  - q_0 ==> a start state
  - F ==> set of accepting states
  - δ ==> a transition function, which is a mapping between Q x ∑ ==> Q
- A DFA is defined by the 5-tuple:
  - {Q, ∑ , q_0,F, δ }

What does a DFA do on reading an input string?

- Input: a word w in ∑*
- Question: Is w acceptable by the DFA?
- Steps:
  - Start at the "start state" q_0
  - For every input symbol in the sequence w do
    - Compute the next state from the current state, given the current input symbol in w and the transition function
  - If after all symbols in w are consumed, the current state is one of the accepting states (F) then *accept w;*
  - Otherwise, *reject w.*

Regular Languages

- Let L(A) be a language *recognized* by a DFA A.
  - Then L(A) is called a "*Regular Language*".



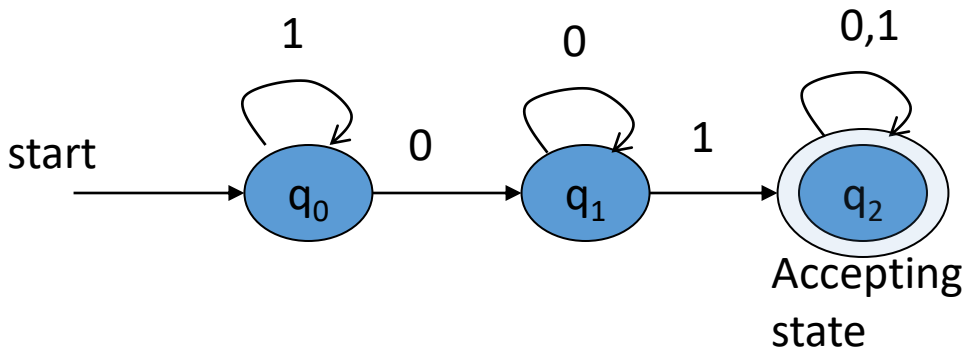
Example #1

- Build a DFA for the following language:
  - L = {w | w is a binary string that contains 01 as a substring}
- Steps for building a DFA to recognize L:
  - ∑ = {0,1}
  - Decide on the states: Q
  - Designate start state and final state(s)
  - δ: Decide on the transitions:
- "Final" states == same as "accepting states"
- Other states == same as "non-accepting states"

Regular expression: (0+1)*01(0+1)*

DFA for strings containing 01

- What makes this DFA deterministic?



- What if the language allows empty strings?

- Q = $\{q_0,q_1,q_2\}$
- ∑ = {0,1}
- start state = q_0
- F = $\{q_2\}$
- Transition table

states	δ	0	1
→	q_0	q_1	q_0
	q_1	q_1	q_2
	$*q_2$	q_2	q_2

symbols

Example #2

- Build a DFA for the following language:
  - L = {w | w is a binary string beginning with 01}

Example #3

- Build a DFA for the following language:
  - L = {w | w is a binary string ending with 01}

Example #4

- Build a DFA for the following language:

  L = {aWa | ∑ ={a , b} }

Example #5

- Build a DFA for the following language:

  L = { w | where leftmost symbol differs from rightmost symbol, ∑ ={a,b}}

Example #6

- Build a DFA for the following language:

  L = {W: |W| mod 5 !=0, W €(0,1)}

Example #7

- Build a DFA for the following language:

  $L=\{ab^3Wb^2 : W \in \{a,b\}^*\}$

Example #8

- Build a DFA for the following language:

For $\Sigma = \{a,b\}$, onstruct dfa's that accept the sets consisting of

(a) all strings with exactly one a,

(b) all strings with at least one a,

(c) all strings with no more than three a's,

Example #9

- Build a DFA for the following language:

$L = \{w_1abw_2 : w1 \in \{a,b\}^*, w2 \in \{a,b\}^*\}$

Example #10

- Build a DFA for the following language:

  L = { Beginning with aab, ∑ ={a , b} }

Example #10

- Build a DFA for the following language:

  L = { Ending with aab, ∑ ={a , b} }

Example #10

- Build a DFA for the following language:

  L = {substring aab, ∑ ={a , b} }

Example #10

- Build a DFA for the following language:

  L = { Not containing substring aab, ∑ ={a , b} }

Extended Transition Function (DFA) δ^*

Let $M = (Q, \Sigma, q_0, A, \delta)$ be a finite automaton. We define the extended transition function

$$\delta^* : Q \times \Sigma^* \rightarrow Q$$

as follows:

1. For every $q \in Q$, $\delta^*(q, \Lambda) = q$
2. For every $q \in Q$, every $y \in \Sigma^*$, and every $\sigma \in \Sigma$,

$$\delta^*(q, y\sigma) = \delta(\delta^*(q, y), \sigma)$$

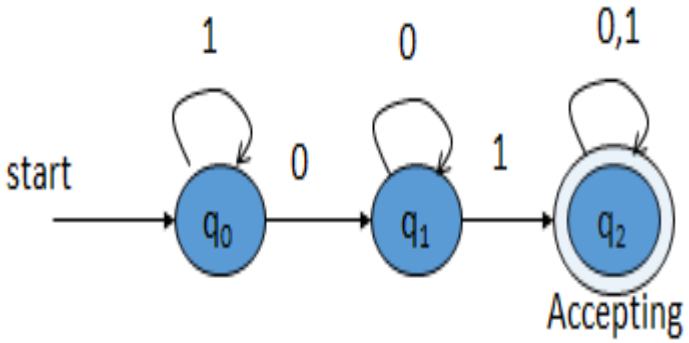
Extension of transitions ($\hat{\delta}$) to Paths (δ)

- $\hat{\delta}$ *(q,w) = destination state* from state *q* on input string *w*

- $\hat{\delta}$ *(q,wa)* = δ ($\hat{\delta}$*(q,w), a)*

Extension of transitions ($\hat{\delta}$) to Paths (δ)

- using the input sequence w=10101



$\hat{\delta}(q0, \varepsilon) = q0$

$\hat{\delta}(q0, 1) = \delta(\hat{\delta}(q0, \varepsilon), 1) = \delta(q0,1) = q0$

$\hat{\delta}(q0, 10) = \delta(\hat{\delta}(q0, 1), 0) = \delta(q0,0) = q1$

$\hat{\delta}(q0, 101) = \delta(\hat{\delta}(q0, 10), 1) = \delta(q1,1) = q2$

$\hat{\delta}(q0, 1010) = \delta(\hat{\delta}(q0, 101), 0) = \delta(q2,1) = q2$

$\hat{\delta}(q0, 10101) = \delta(\hat{\delta}(q0, 1010), 1) = \delta(q2,1) = q2$

Let $M = (Q, \Sigma, q_0, A, \delta)$ be a finite automaton. We define the extended transition function

$$\delta^* : Q \times \Sigma^* \to Q$$

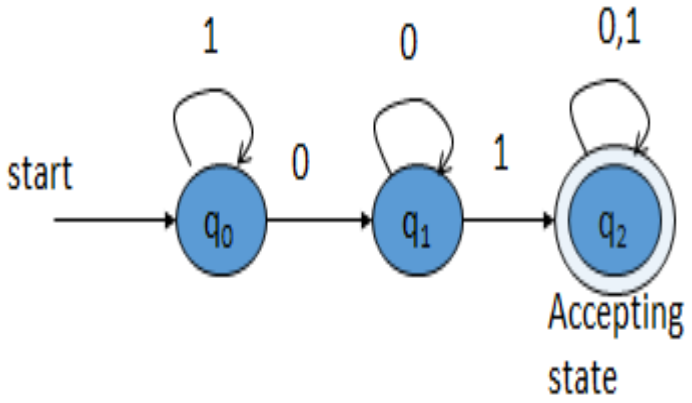
as follows:

1. For every $q \in Q$, $\delta^*(q, \Lambda) = q$
2. For every $q \in Q$, every $y \in \Sigma^*$, and every $\sigma \in \Sigma$,

$$\delta^*(q, y\sigma) = \delta(\delta^*(q, y), \sigma)$$

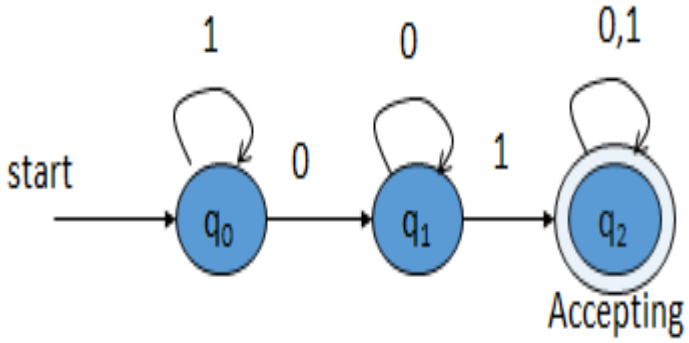
Extension of transitions ($\hat{\delta}$) to Paths (δ)-Example

- using the input sequence w=1001



Extension of transitions (δ) to Paths (δ)-Example

- using the input sequence w=1001



$\hat{\delta}(q0, \varepsilon) = q0$

$\hat{\delta}(q0, 1) = \delta(\hat{\delta}(q0, \varepsilon), 1) = \delta(q0, 1) = q0$

$\hat{\delta}(q0, 10) = \delta(\hat{\delta}(q0, 1), 0) = \delta(q0, 0) = q1$

$\hat{\delta}(q0, 100) = \delta(\hat{\delta}(q0, 10), 0) = \delta(q1, 0) = q1$

$\hat{\delta}(q0, 1001) = \delta(\hat{\delta}(q0, 100), 1) = \delta(q1, 1) = q2$

Let $M = (Q, \Sigma, q_0, A, \delta)$ be a finite automaton. We define the extended transition function

$$\delta^* : Q \times \Sigma^* \to Q$$

as follows:

1. For every $q \in Q$, $\delta^*(q, \Lambda) = q$
2. For every $q \in Q$, every $y \in \Sigma^*$, and every $\sigma \in \Sigma$,

$$\delta^*(q, y\sigma) = \delta(\delta^*(q, y), \sigma)$$

Language of a DFA -Definition

A DFA A accepts string *w* if there is a path from q_0 to an accepting (or final) state that is labeled by *w*

- *i.e.,* $L(A) = \{ w \mid \delta(q_0, w) \in F \}$
- *I.e., L(A) = all strings that lead to an accepting state from* q_0

Let $M = (Q, \Sigma, q_0, A, \delta)$ be an FA, and let $x \in \Sigma^*$. The string x is *accepted* by M if

$$\delta^*(q_0, x) \in A$$

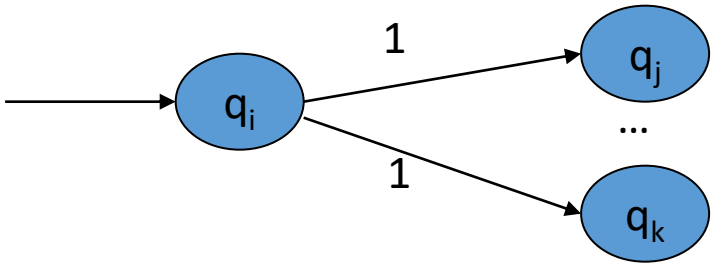
and is *rejected* by M otherwise. The *language* accepted by M is the set

$$L(M) = \{x \in \Sigma^* \mid x \text{ is accepted by } M\}$$

If L is a language over Σ, L is accepted by M if and only if $L = L(M)$.

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA)
  - Implying that the machine can exist in more than one state at the same time
  - Transitions could be non-deterministic



- Each transition function therefore maps to a set of states

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA) consists of:
  - Q ==> a finite set of states
  - ∑ ==> a finite set of input symbols (alphabet)
  - q_0 ==> a start state
  - F ==> set of accepting states
  - δ ==> a transition function, which is a mapping between Q x ∑ ==> subset of Q (2^Q)

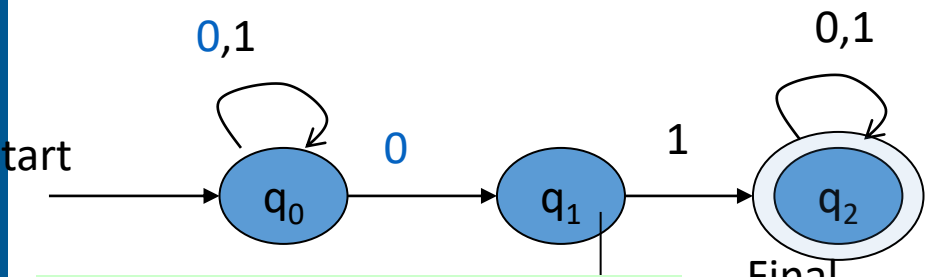
- An NFA is also defined by the 5-tuple:
  - {Q, ∑ , q_0,F, δ }

How to use an NFA?

- Input: a word w in ∑*
- Question: Is w acceptable by the NFA?
- Steps:
  - Start at the "start state" q_0
  - For every input symbol in the sequence w do
    - Determine all possible next states from all current states, given the current input symbol in w and the transition function
  - If after all symbols in w are consumed and if at least one of the current states is a final state then *accept w;*
  - Otherwise, *reject w.*

NFA for strings containing 01

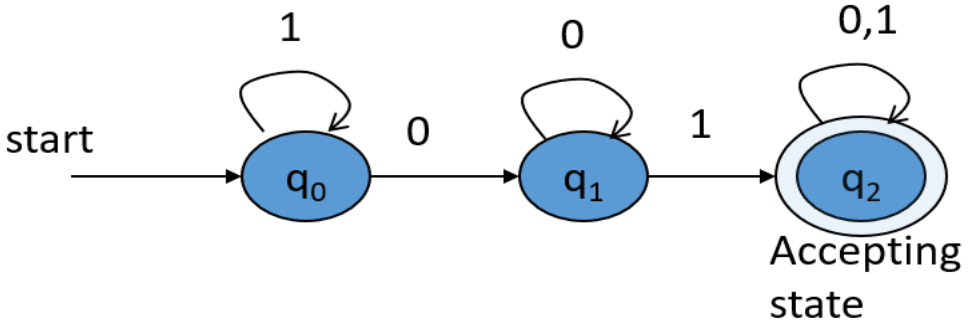
Why is this non-deterministic?



What will happen if at state q_1 an input of 0 is received?

- Q = $\{q_0,q_1,q_2\}$
- Σ = {0,1}
- start state = q_0
- F = $\{q_2\}$
- Transition table

	symbols	
δ	**0**	**1**
states → q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Φ	$\{q_2\}$
$*q_2$	$\{q_2\}$	$\{q_2\}$



Example #2

- Build an NFA for the following language:

  $L = \{ w \mid w \text{ ends in } 01 \}$

Example #3

- Build an NFA for the following language:

  $L = \{aWa \mid \sum = \{a , b\}\}$

Example #4

- Build an NFA for the following language:

  $L=\{ab^3Wb^2 : W \in \{a,b\}^*\}$

Example #5

- Build an NFA for the following language:
  L = { w | where leftmost symbol differs from rightmost symbol, ∑ ={a,b}}

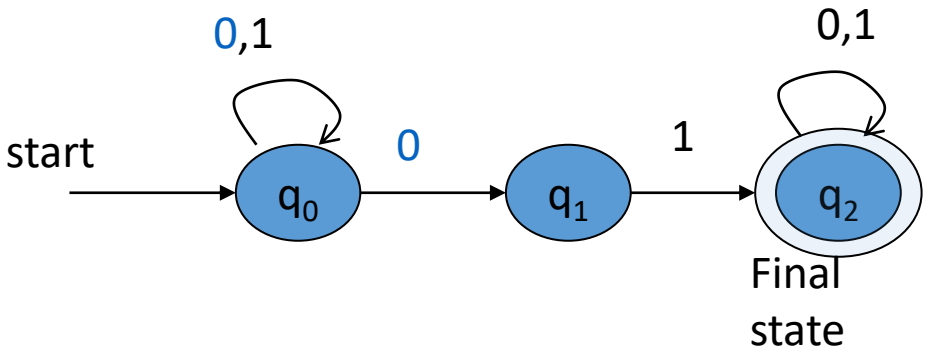
Example #6

- Build an NFA for the following language:

$$L = \{ w \mid w \text{ is } abab^n \text{ or } aba^n \mid n>=0\}$$

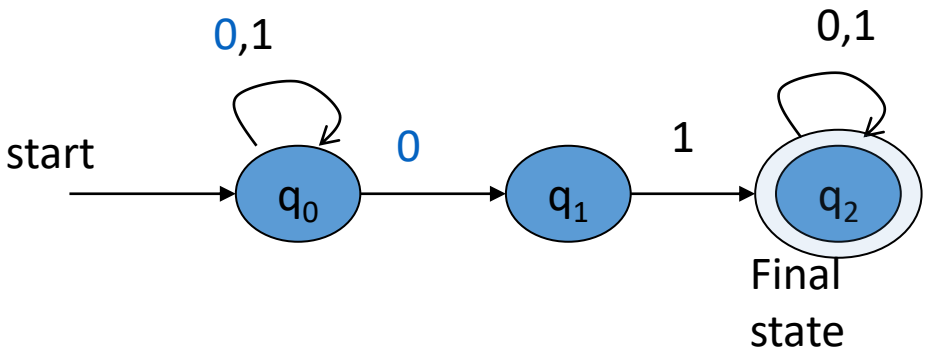
Extension of δ to NFA Paths- Example(1010)

- $\hat{\delta}(q0, \varepsilon) = \{q0\}$
- $\hat{\delta}(q0,1) = \delta(q0,1) = \{q0\}$
- $\hat{\delta}(q0,10) = \delta(q0,0) = \{q0,q1\}$
- $\hat{\delta}(q0,101) = \delta(q0,1) \cup \delta(q1,1) = \{q0,q2\}$
- $\hat{\delta}(q0,1010) = \delta(q0,0) \cup \delta(q2,0) = \{qo,q1,q2\}$



Extension of δ to NFA Paths- Example(1100)

- $\hat{\delta}(q0, \varepsilon) = \{q0\}$
- $\hat{\delta}(q0,1) = \delta(q0,1) = \{q0\}$
- $\hat{\delta}(q0,11) = \delta(q0,1) = \{q0\}$
- $\hat{\delta}(q0,110) = \delta(q0,0) = \{q0,q1\}$
- $\hat{\delta}(q0,1100) = \delta(q0,0) \cup \delta(q1,0) = \{qo,q1\}$



Language of an NFA

- An NFA accepts *w* if *there exists at least one* path from the start state to an accepting (or final) state that is labeled by *w*

- $L(N) = \{ w \mid \hat{\delta}(q_0, w) \cap F \neq \Phi \}$

Advantages & Caveats for NFA

- Great for modeling regular expressions
  - String processing - e.g., grep, lexical analyzer
- Could a non-deterministic state machine be implemented in practice?
  - Probabilistic models could be viewed as extensions of non-deterministic state machines (e.g., toss of a coin, a roll of dice)
    - They are not the same though
  - A parallel computer could exist in multiple “states” at the same time

But, DFAs and NFAs are equivalent in their power to capture langauges !!

Differences: DFA vs. NFA

- **DFA**

1. All transitions are deterministic
   - Each transition leads to exactly one state
2. For each state, transition on all possible symbols (alphabet) should be defined
3. Accepts input if the last state visited is in F
4. Sometimes harder to construct because of the number of states
5. Practical implementation is feasible

- **NFA**

1. Some transitions could be non-deterministic
   - A transition could lead to a subset of states
2. Not all symbol transitions need to be defined explicitly (if undefined will go to an error state – this is just a design convenience, not to be confused with "non-determinism")
3. Accepts input if *one of* the last states is in F
4. Generally easier than a DFA to construct
5. Practical implementations limited but emerging (e.g., Micron automata processor)

Equivalence of DFA & NFA

- Theorem:
  - A language L is accepted by a DFA *if and only if* it is accepted by an NFA.

Should be true for any L

Proof:

1. If part:
   - Prove by showing every NFA can be converted to an equivalent DFA (in the next few slides…)

2. Only-if part is trivial:
   - Every DFA is a special case of an NFA where each state has exactly one transition for every input symbol. Therefore, if L is accepted by a DFA, it is accepted by a corresponding NFA.

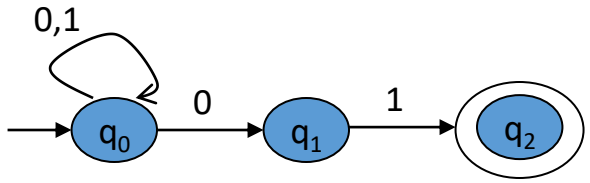
NFA to DFA by subset construction

- Let N = $\{Q_N, \sum, \delta_N, q_0, F_N\}$
- Goal: Build D=$\{Q_D, \sum, \delta_D, \{q_0\}, F_D\}$ s.t. L(D)=L(N)
- Construction:
  1. Q_D= all subsets of Q_N (i.e., power set)
  2. F_D=set of subsets S of Q_N s.t. $S \cap F_N \neq \Phi$
  3. δ_D: for each subset S of Q_N and for each input symbol a in $\sum$:
     - $\delta_D(S,a) = \bigcup_{p \text{ in } s} \delta_N(p,a)$

NFA to DFA construction: Example

- *L = {w | w ends in 01}*

NFA:



0. Enumerate all possible subsets
1. Determine transitions

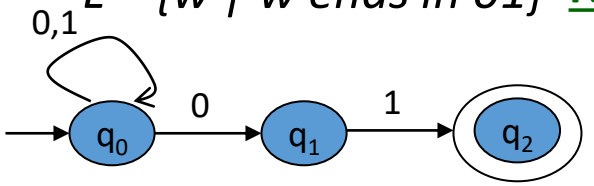
DFA:

δ_D	0	1
Ø	Ø	Ø
→ $[q_0]$	$\{q_0,q_1\}$	$\{q_0\}$
$[q_1]$	Ø	$\{q_2\}$
*$[q_2]$	Ø	Ø
$[q_0,q_1]$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
*$[q_0,q_2]$	$\{q_0,q_1\}$	$\{q_0\}$
*$[q_1,q_2]$	Ø	$\{q_2\}$
*$[q_0,q_1,q_2]$	$\{q_0,q_1\}$	$\{q_0,q_2\}$

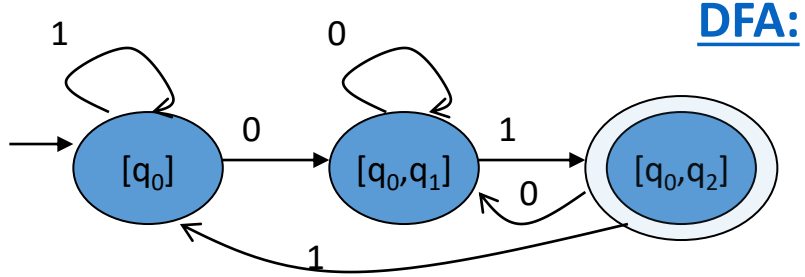
Idea: To avoid enumerating all of power set, do "lazy creation of states"

NFA to DFA construction: Example

- *L = {w | w ends in 01}* **NFA:**



DFA:



δ_D	0	1
$[q_0]$	$[q_0,q_1]$	$[q_0]$
→ $[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_2]$
*$[q_0,q_2]$	$[q_0,q_1]$	$[q_0]$

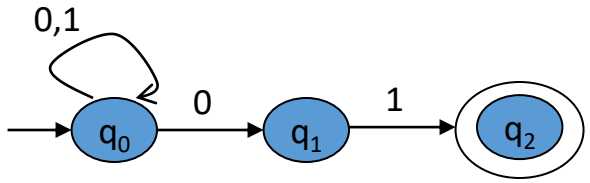
0. Enumerate all possible subsets
1. Determine transitions
2. Retain only those states reachable from $\{q_0\}$

Idea: To avoid enumerating all of power set, do "lazy creation of states"

NFA to DFA construction: Example

- *L = {w | w ends in 01}*

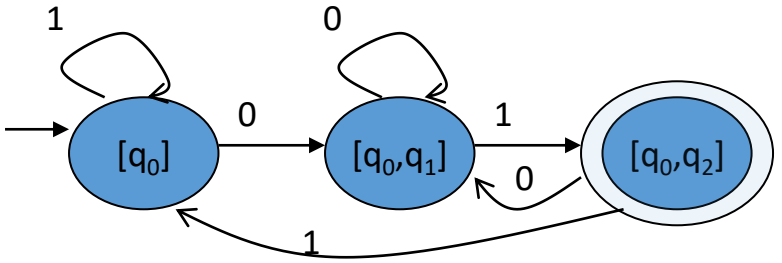
NFA:



δ_N	0	1
q_0 →	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
$*q_2$	Ø	Ø

DFA:

δ_D	0	1
Ø	Ø	Ø
$[q_0]$	$\{q_0,q_1\}$	$\{q_0\}$
~~$[q_1]$~~	~~Ø~~	~~$\{q_2\}$~~
~~$*[q_2]$~~	~~Ø~~	~~Ø~~
~~$[q_0,q_1]$~~	~~$\{q_0,q_1\}$~~	~~$\{q_0,q_2\}$~~
$*[q_0,q_2]$	$\{q_0,q_1\}$	$\{q_0\}$
$*[q_1,q_2]$	Ø	$\{q_2\}$
~~$*[q_0,q_1,q_2]$~~	~~$\{q_0,q_1\}$~~	~~$\{q_0,q_2\}$~~



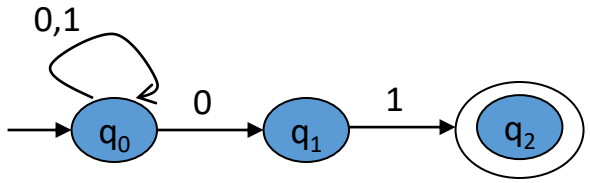
δ_D	0	1
→ $[q_0]$	$[q_0,q_1]$	$[q_0]$
$[q_0,q_1]$	$[q_0,q_1]$	$[q_0,q_2]$
$*[q_0,q_2]$	$[q_0,q_1]$	$[q_0]$

0. Enumerate all possible subsets
1. Determine transitions
2. Retain only those states reachable from $\{q_0\}$

NFA to DFA: Repeating the example using *LAZY CREATION*

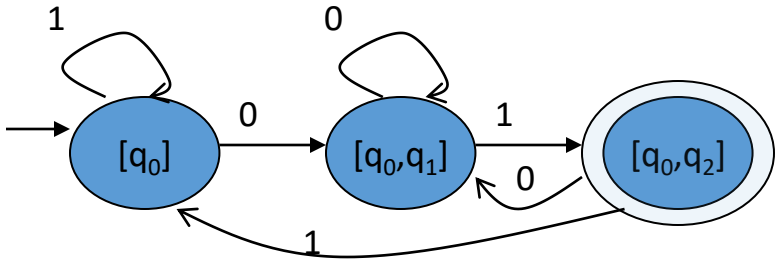
- *L = {w | w ends in 01}*

NFA:



δ_N	0	1
→ q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$
$*q_2$	Ø	Ø

DFA:



δ_D	0	1
→ $[q_0]$	$[q_0,q_1]$	$[q_0]$

Main Idea:

Introduce states as you go (on a need basis)

Steps for converting NFA to DFA

- **Step 1:** Initially $Q_D = \phi$
- **Step 2:** Add q0 of NFA to Q_D. Find the transitions from this start state.
- **Step 3:** In Q_D, find the possible set of states for each input symbol. If this set of states is not in Q_D, then add it to Q_D.

For each set $S \subseteq Q_N$ and for each input symbol a in Σ,

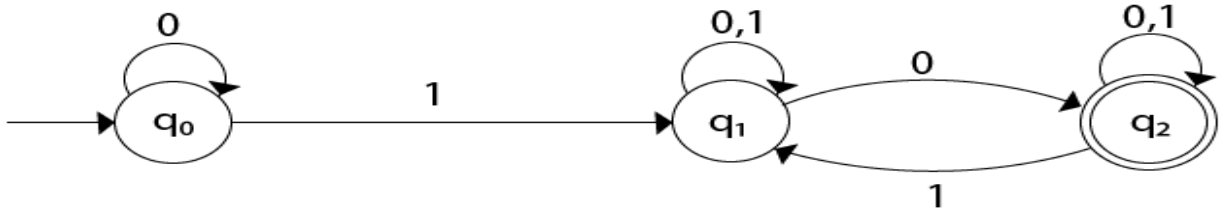
$$\delta_D(S, a) = \bigcup_{p \text{ in } S} \delta_N(p, a)$$

- **Step 4:** In DFA, the final state F_D will be all the states which contain F_N (final states of NFA)

F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N's states that include at least one accepting state of N.

NFA TO DFA :Example 1

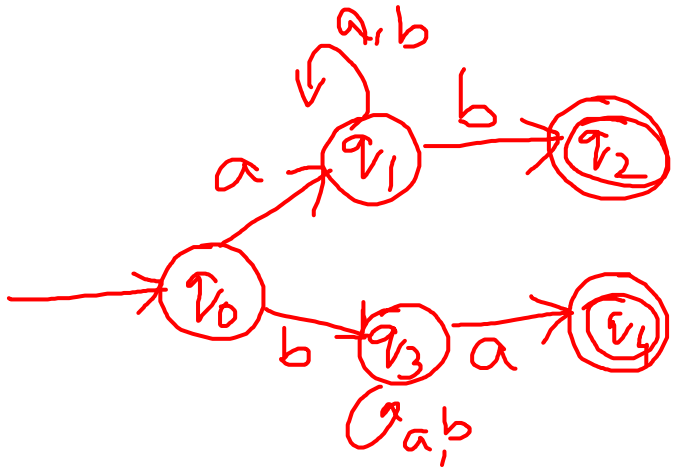
- δ([q0], 0) = [q0]
- δ([q0], 1) = [q1]
- δ([q1], 0) = [q1, q2]
- δ([q1], 1) = [q1]
- δ([q1, q2], 0) = δ(q1, 0) ∪ δ(q2, 0)
  = {q1, q2} ∪ {q2}
  = [q1, q2]
- δ([q1, q2], 1) = δ(q1, 1) ∪ δ(q2, 1)
  = {q1} ∪ {q1, q2}
  = {q1, q2}
  = [q1, q2]



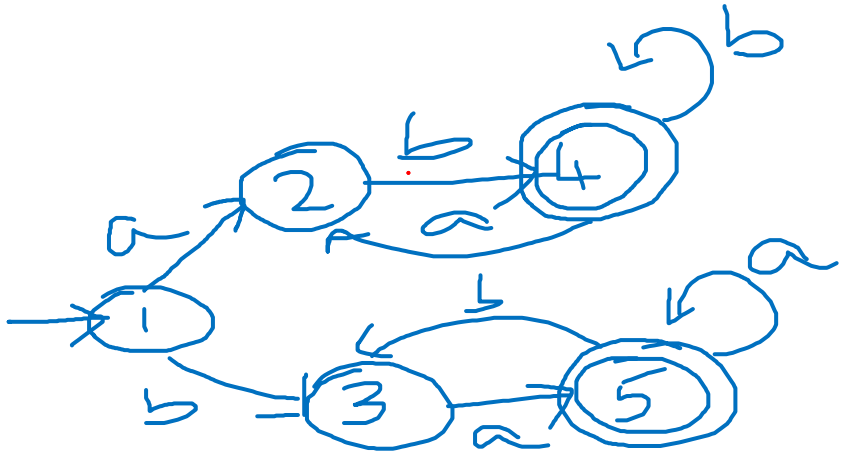
State	0	1
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q1, q2]	[q1, q2]	[q1, q2]

NFA to DFA

- L = { w | where leftmost symbol differs from rightmost symbol, ∑ ={a,b}}

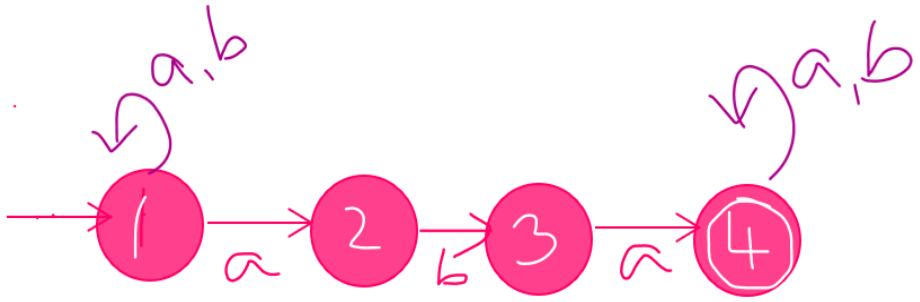


	State	a	b
1	→[q0]	q1	q3
2	[q1]	q1	[q1,q2]
3	q3	[q3,q4]	q3
4	*[q1, q2]	[q1]	[q1,q2]
5	*[q3,q4]	[q3,q4]	q3



NFA to DFA

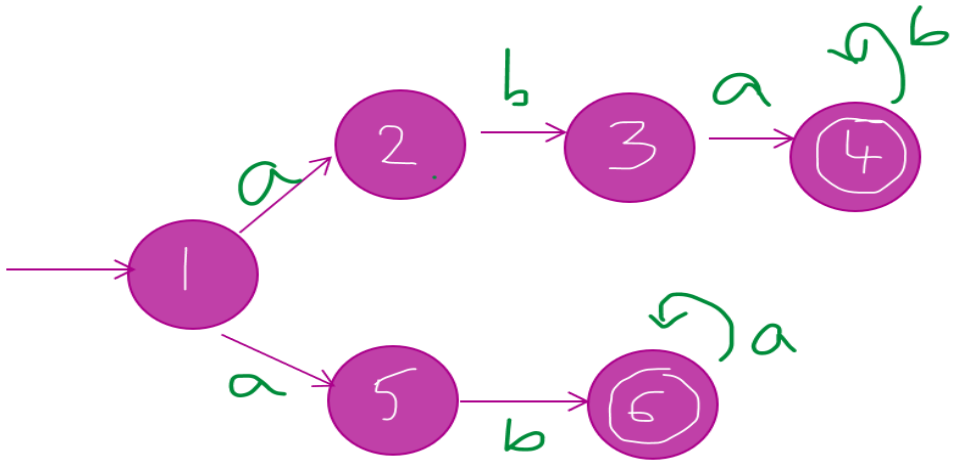
- NFA to accept strings containing aba as substring



State	a	b
→[1]		

NFA to DFA

- L = { w | w is $abab^n$ or aba^n |n>=0}



A bad case where #states(DFA)>>#states(NFA)

Properties of DFAs and NFAs

- The machine never really terminates.
  - It is always waiting for the next input symbol or making transitions.
- The machine decides when to consume the next symbol from the input and when to ignore it.
  - (but the machine can never skip a symbol)
- => A transition can happen even *without* really consuming an input symbol (think of consuming ε as a free token) – if this happens, then it becomes an ε-NFA slides).
- A single transition *cannot* consume more than one (non-ε) symbol.

FA with ε-Transitions

- We can allow explicit ε-transitions in finite automata
  - i.e., a transition from one state to another state without consuming any additional input symbol
  - Explicit ε-transitions between different states introduce non-determinism
  - Makes it easier sometimes to construct NFAs

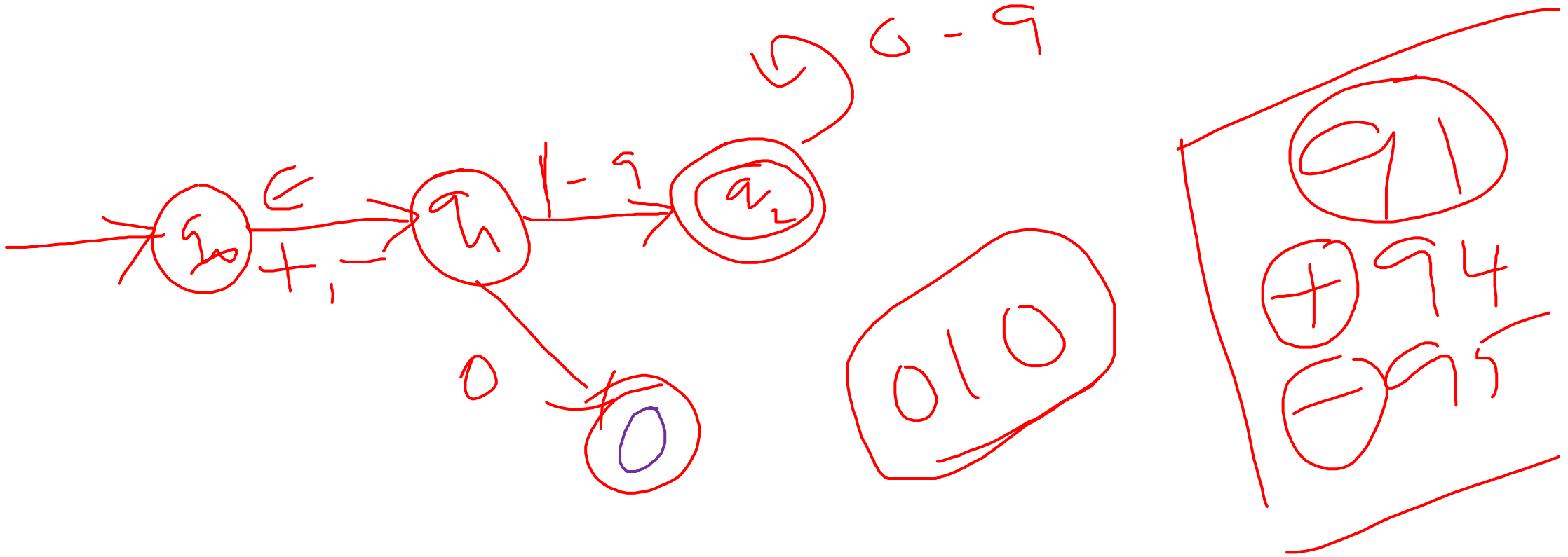
Definition: ε-NFAs are those NFAs with at least one explicit ε-transition defined.

- ε -NFAs have one more column in their transition table

ε-Non-deterministic Finite Automata (ε-NFA)

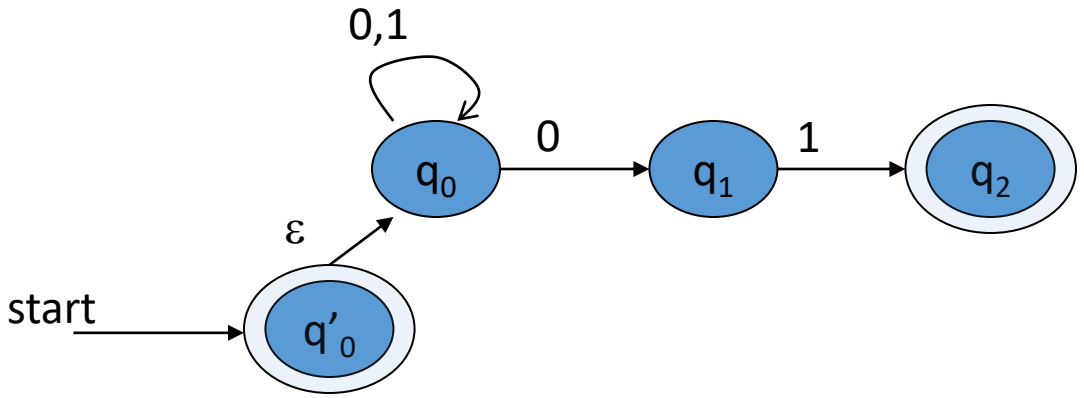
- A ε-NFA consists of:
  - Q ==> a finite set of states
  - ∑ ==> a finite set of input symbols (alphabet)
  - q_0 ==> a start state
  - F ==> set of accepting states
  - δ ==> a transition function, which is a mapping between Q x (∑ U ε)==> subset of Q (2^Q)
- An ε-NFA is also defined by the 5-tuple:
  - {Q, ∑ , q_0,F, δ }

ε-NFA to accept integers



Example of an ε-NFA

L = {w | w is empty, or if non-empty will end in 01}



δ_E	0	1	ε
→ *q'_0	Ø	Ø	$\{q'_0, q_0\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
*q_2	Ø	Ø	$\{q_2\}$

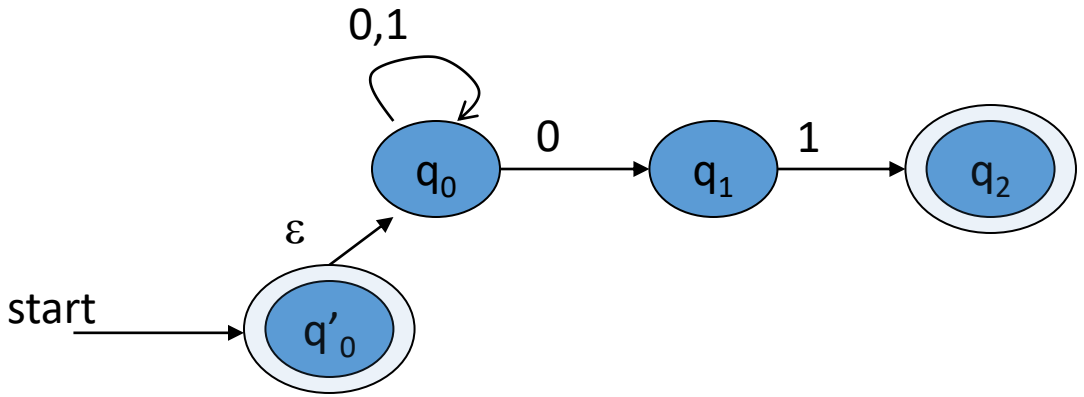
ECLOSE(q'_0)
ECLOSE(q_0)
ECLOSE(q_1)
ECLOSE(q_2)

- ε-closure of a state q, ***ECLOSE(q)***, is the set of all states (including itself) that can be *reached* from q by repeatedly making an arbitrary number of ε-transitions.

Example of an ε-NFA

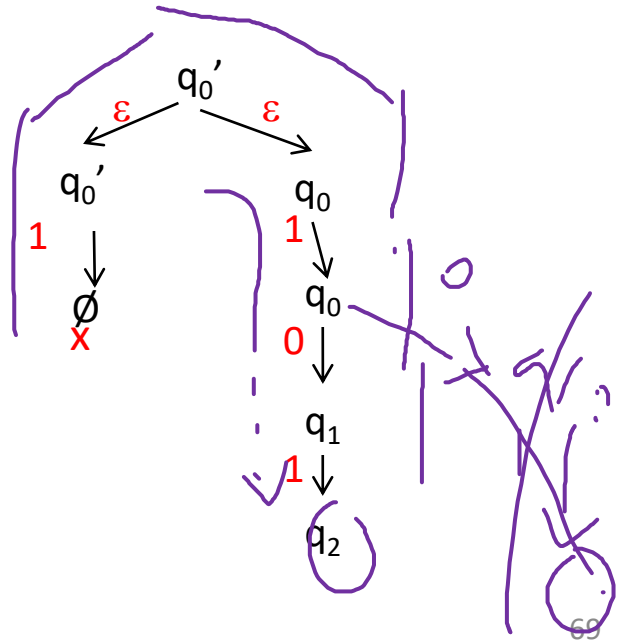
To simulate any transition:
Step 1) Go to all immediate destination states.
Step 2) From there go to all their ε-closure states as well.

L = {w | w is empty, or if non-empty will end in 01}



δ_E	0	1	ε	
→ *q'_0	Ø	Ø	$\{q'_0, q_0\}$	ECLOSE(q'_0)
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$	ECLOSE(q_0)
q_1	Ø	$\{q_2\}$	$\{q_1\}$	
*q_2	Ø	Ø	$\{q_2\}$	

Simulate for w=101:

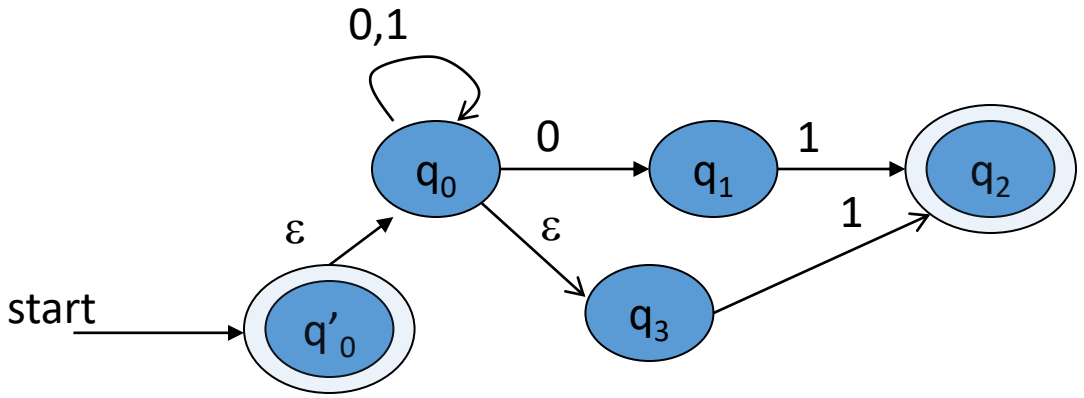


Example of another ε-NFA

To simulate any transition:

Step 1) Go to all immediate destination states.

Step 2) From there go to all their ε-closure states as well.



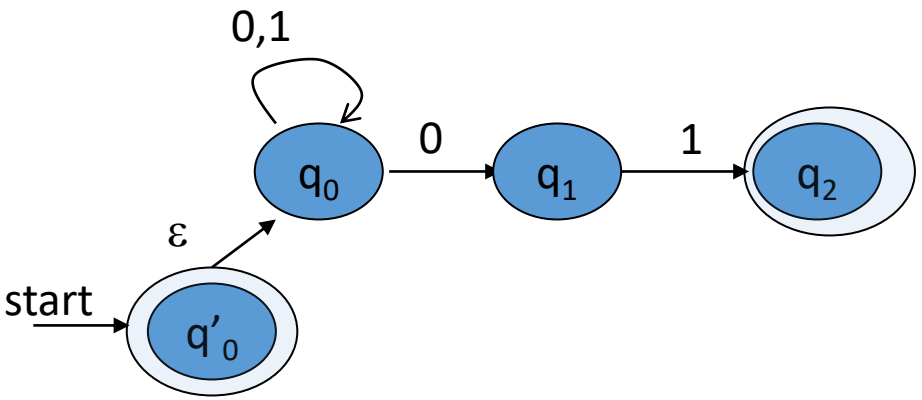
δ_E	0	1	ε
→ $*q'_0$	Ø	Ø	$\{q'_0,q_0,q_3\}$
q_0	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0,q_3\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
$*q_2$	Ø	Ø	$\{q_2\}$
q_3	Ø	$\{q_2\}$	$\{q_3\}$

Simulate for w=101:

?

Example: ε-NFA ➔ DFA

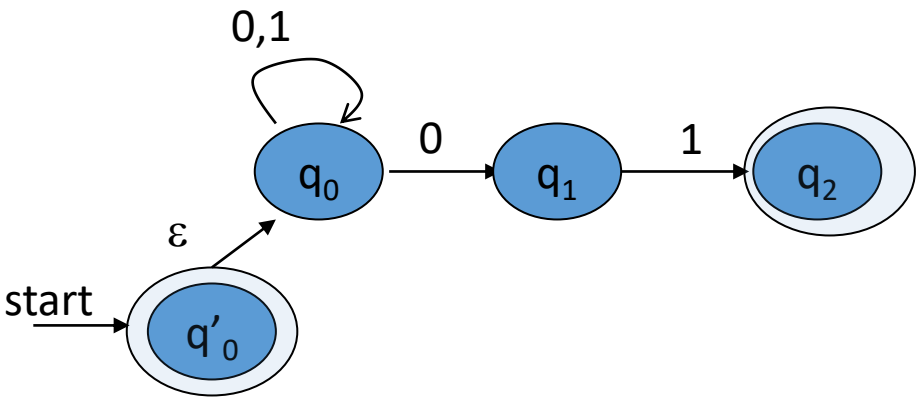
L = {w | w is empty, or if non-empty will end in 01}



δ_E	0	1	ε
$*q'_0$	Ø	Ø	$\{q'_0, q_0\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
$*q_2$	Ø	Ø	$\{q_2\}$

Example: ε-NFA ➔ DFA

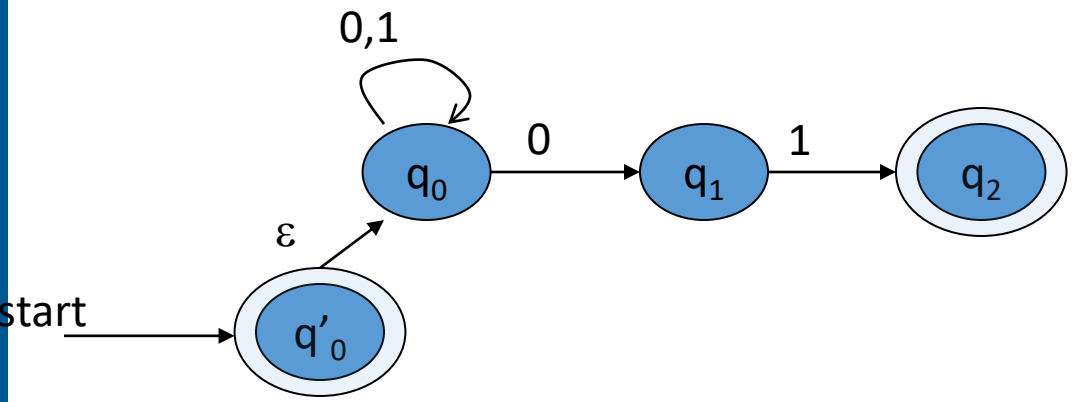
L = {w | w is empty, or if non-empty will end in 01}

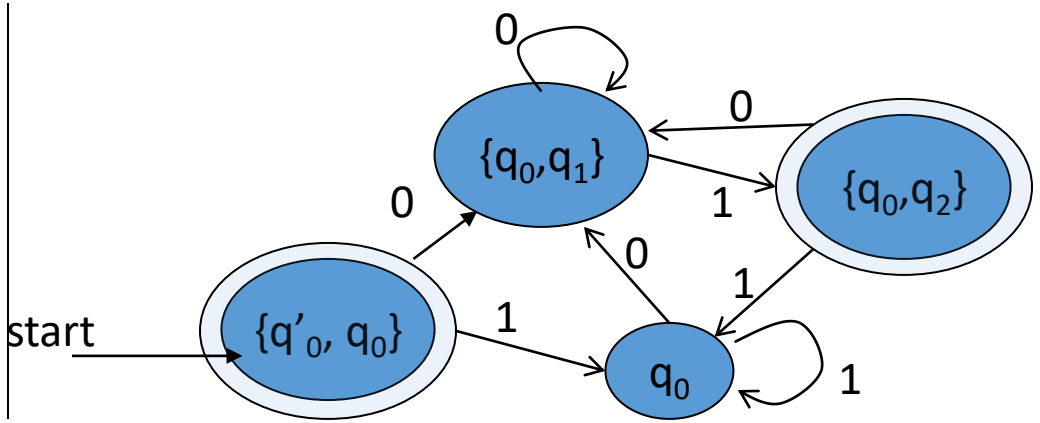


δ_E	0	1	ε
$*q'_0$	Ø	Ø	$\{q'_0, q_0\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
$*q_2$	Ø	Ø	$\{q_2\}$

Example: ε -NFA ➔ DFA

L = {w | w is empty, or if non-empty will end in 01}

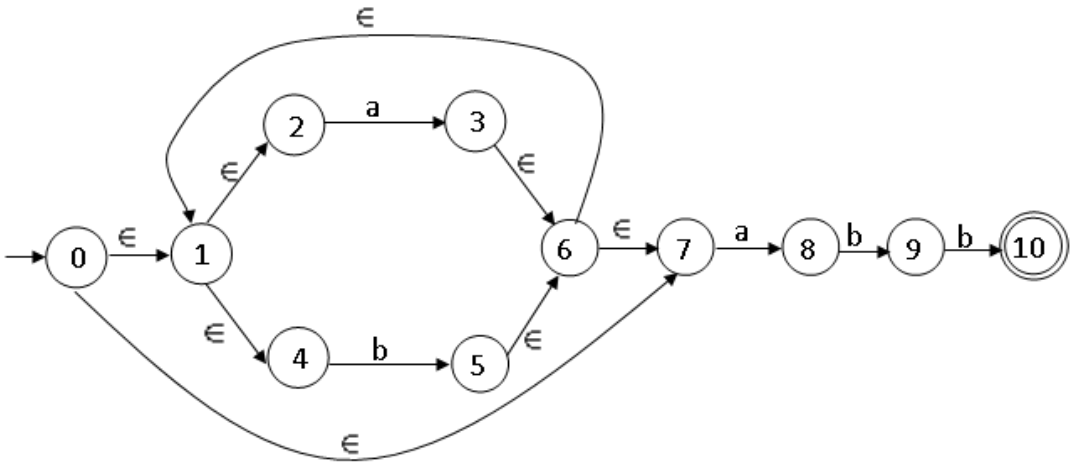


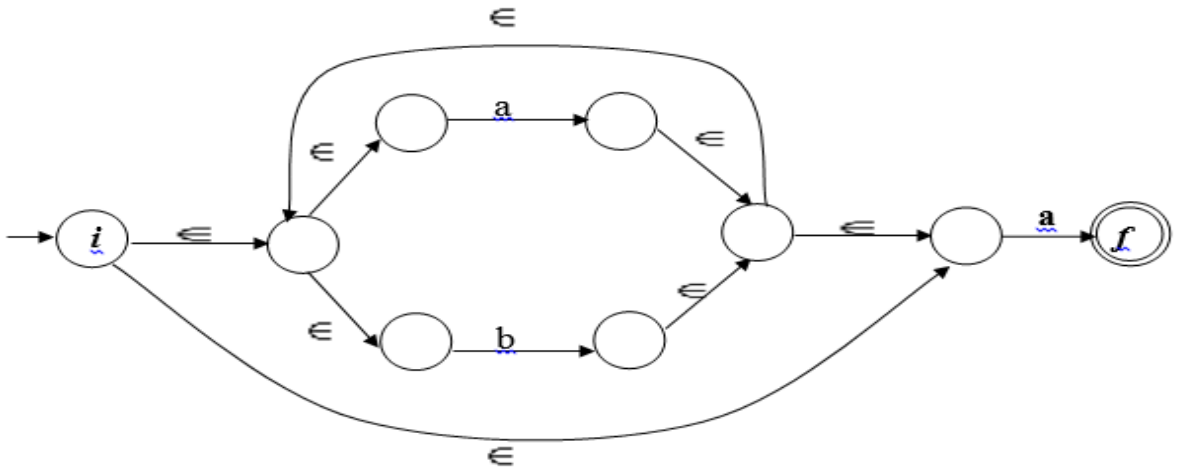


δ_E	0	1	ε
→ *q'_0	Ø	Ø	$\{q'_0,q_0\}$
q_0	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
*q_2	Ø	Ø	$\{q_2\}$

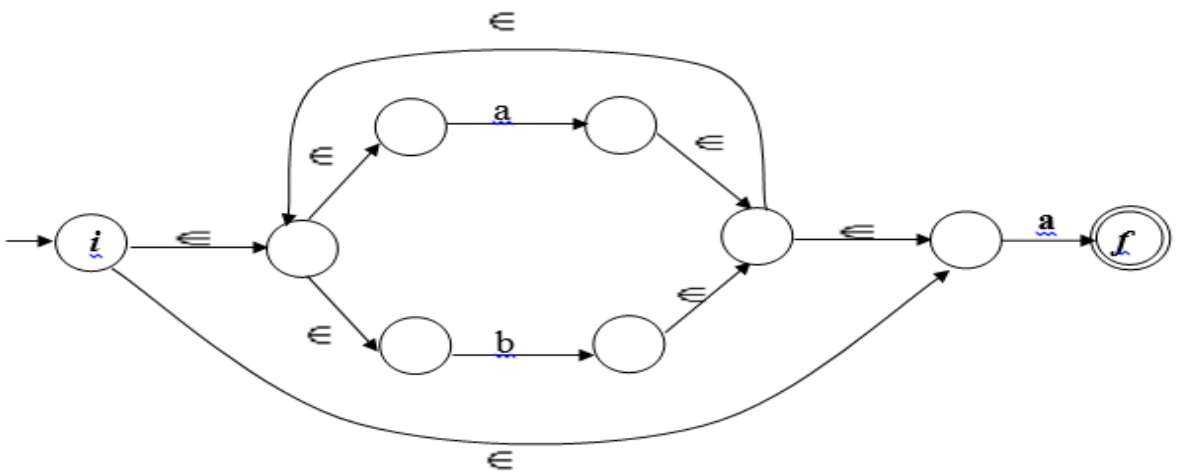
δ_D	0	1
→ *$\{q'_0,q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
*$\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$

Find ECLOSE of all the states

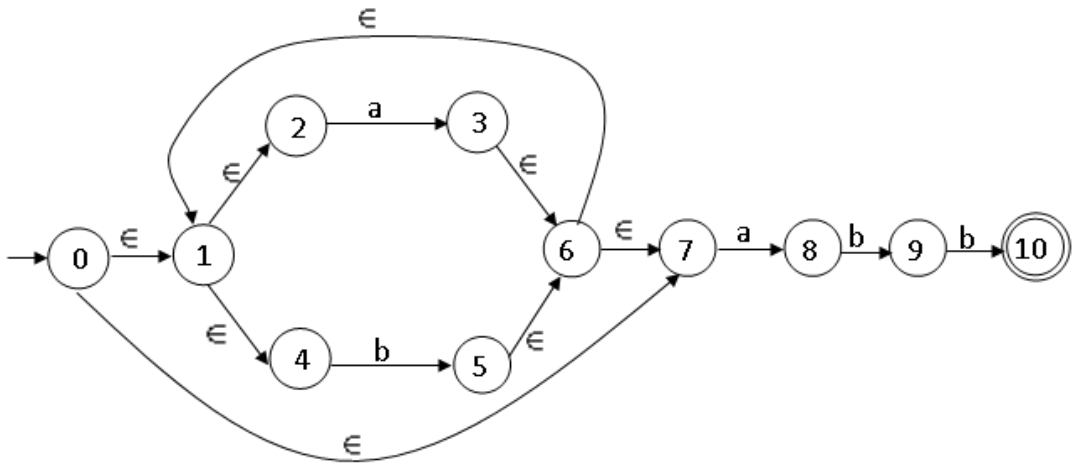




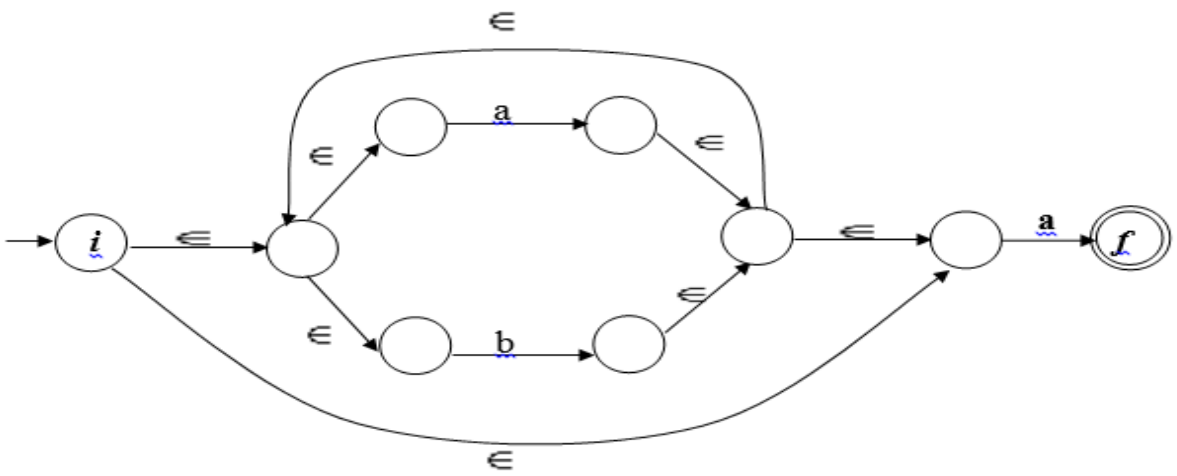
Find ECLOSE of all the states



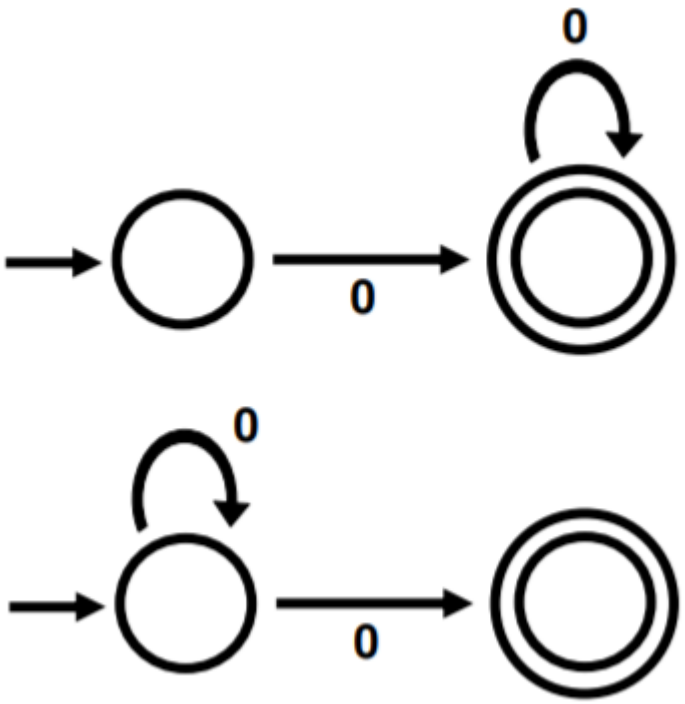
Find ECLOSE of all the states

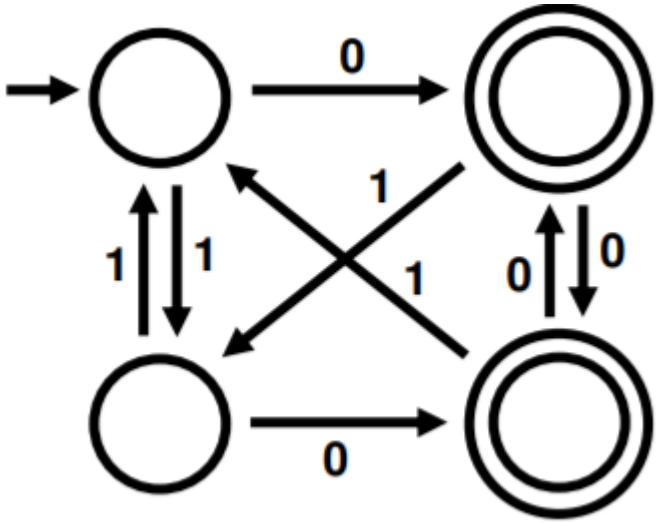


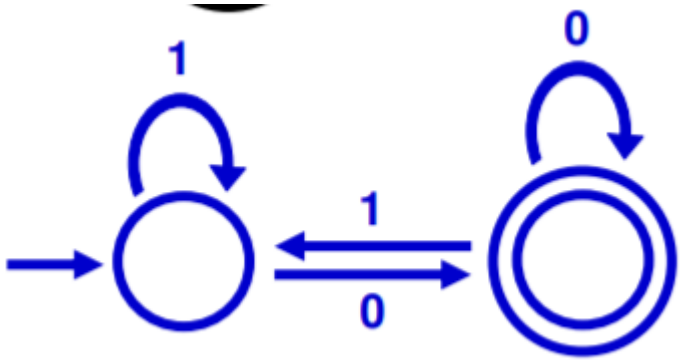
Find ECLOSE of all the states



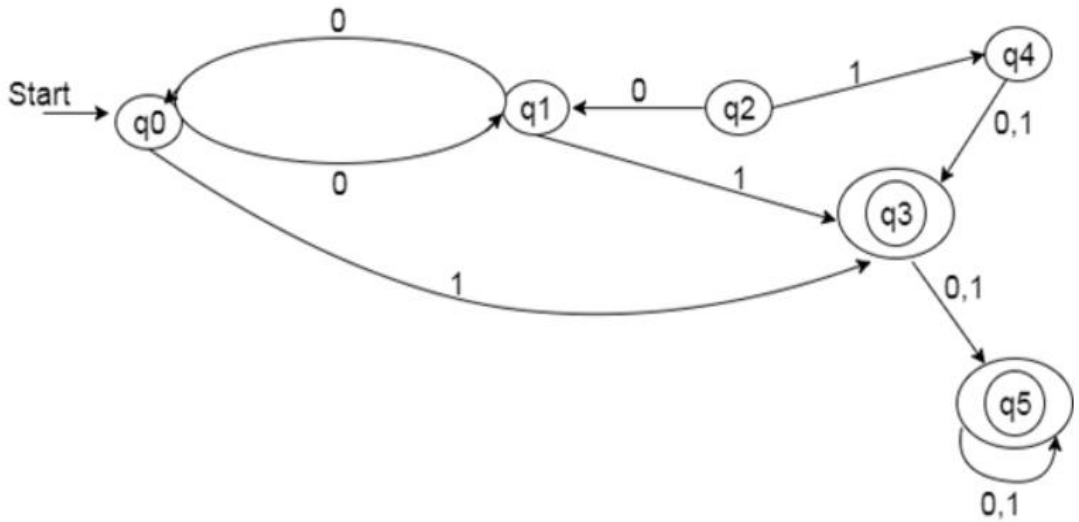
Equivalent DFAs

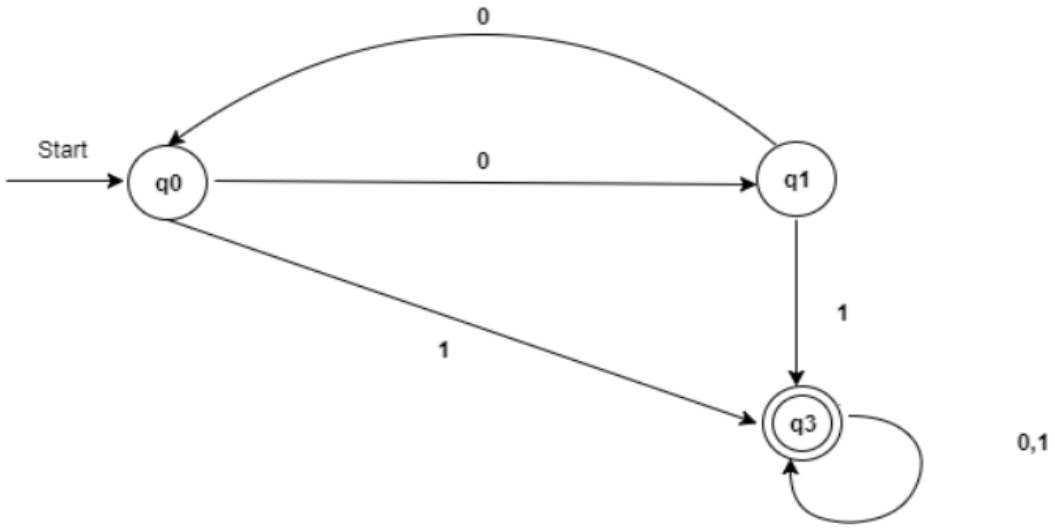






Equivalent DFAs





Minimization of DFAs

Distinguishable and Indistinguishable states

- Two states p and q of a DFA are called indistinguishable if

$$\delta^*(p, w) \in F \text{ implies } \delta^*(q, w) \in F,$$

AND

$$\delta^*(p, w) \notin F \text{ implies } \delta^*(q, w) \notin F,$$

for all w ∈ Σ * . If, on the other hand, there exists some string w ∈ Σ * such that

$$\delta^*(p, w) \in F \text{ and } \delta^*(q, w) \notin F,$$

or vice versa, then the states p and q are said to be distinguishable by a string w.

Table Filling Algorithm

- procedure: mark

1. Remove all inaccessible states.

Any state not part of some path is inaccessible.

2. Consider all pairs of states (p, q).

If p ∈ F and q ∉ F or vice versa, mark the pair (p, q) as distinguishable.

3. Repeat the following step until no previously unmarked pairs are marked.

For all pairs (p, q) and all a ∈ Σ, compute δ(p, a)= r and δ (q, a) = s .

If the pair (r ,s) is marked as distinguishable, mark (p, q) as distinguishable.

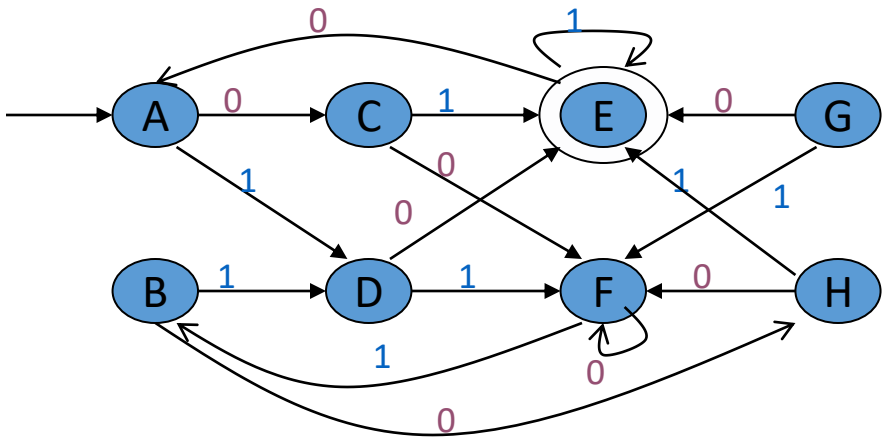
Applications of interest

- Comparing two DFAs:
  - $L(DFA_1) == L(DFA_2)$?

- How to minimize a DFA?
  1. Remove unreachable states
  2. Identify & condense equivalent states into one

Computing equivalent states in a DFA

Table Filling Algorithm



A	=							
B	=	=						
C	x	x	=					
D	x	x	x	=				
E	x	x	x	x	=			
F	x	x	x	x	x	=		
G	x	x	x	=	x	x	=	
H	x	x	=	x	x	x	x	=
	A	B	C	D	E	F	G	H

Pass #0

1. Mark accepting states ≠ non-accepting states

Pass #1

1. Compare every pair of states
2. Distinguish by one symbol transition
3. Mark = or ≠ or blank(tbd)

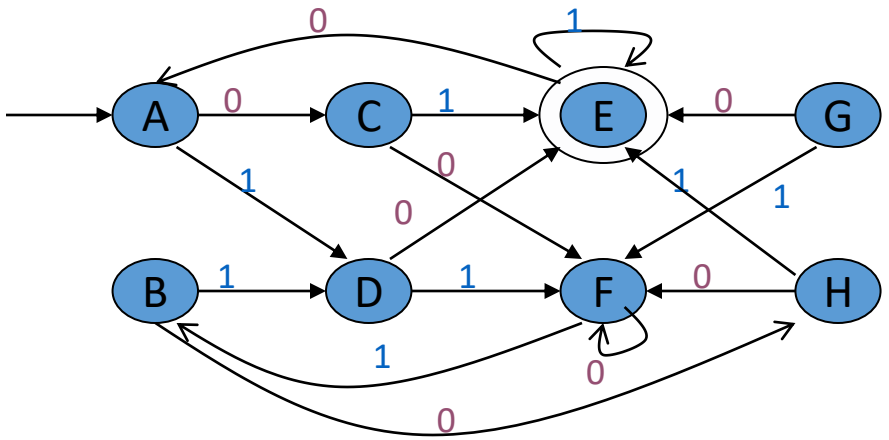
Pass #2

1. Compare every pair of states
2. Distinguish by up to two symbol transitions (until different or same or tbd)

….

(keep repeating until table complete)

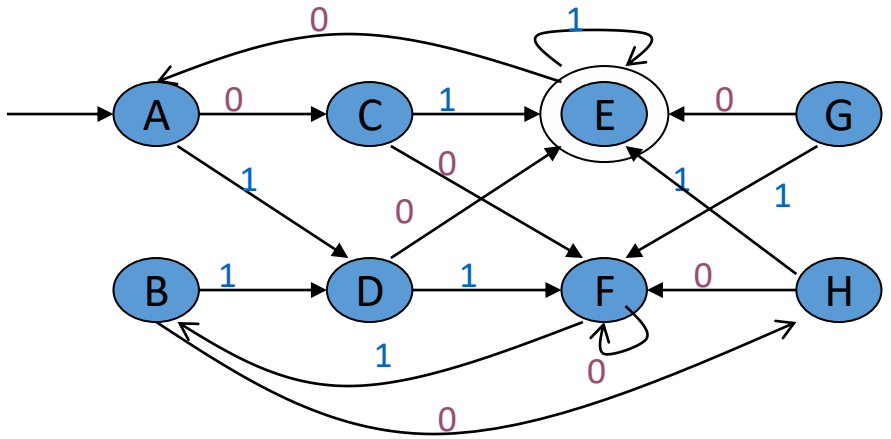
Table Filling Algorithm - step by step



A	=							
B		=						
C			=					
D				=				
E					=			
F						=		
G							=	
H								=
	A	B	C	D	E	F	G	H

Table Filling Algorithm - step by step

States	0	1
→A	C	D
B	H	D
C	F	E
D	E	F
*E	A	E
F	F	B
G	E	F
H	F	E

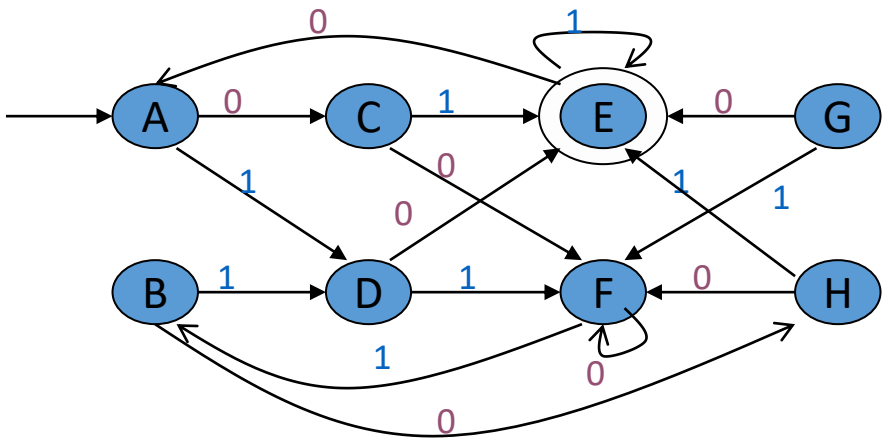


1. Mark X between accepting vs. non-accepting state

A	=							
B		=						
C			=					
D				=				
E	X	X	X	X	=			
F					X	=		
G					X		=	
H					X			=
	A	B	C	D	E	F	G	H

Table Filling Algorithm - step by step

States	0	1
→A	C	D
B	H	D
C	F	E
D	E	F
*E	A	E
F	F	B
G	E	F
H	F	E

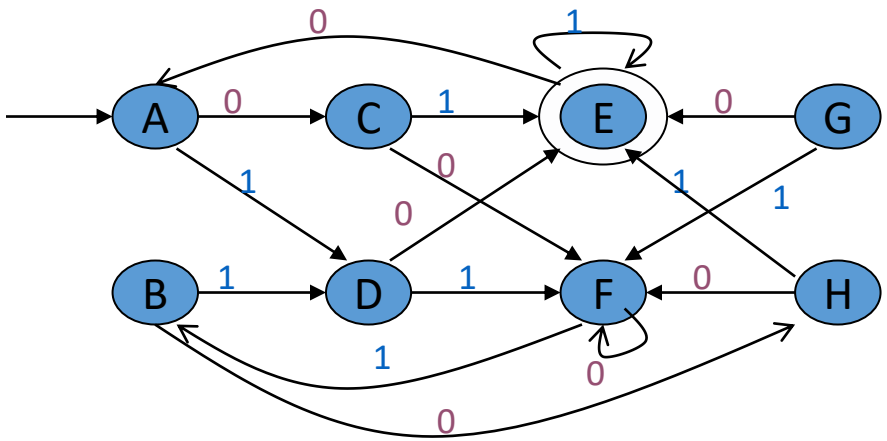


	A	B	C	D	E	F	G	H
A	=							
B		=						
C	X		=					
D	X			=				
E	X	X	X	X	=			
F					X	=		
G	X				X		=	
H	X				X			=

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

Table Filling Algorithm - step by step

States	0	1
→A	C	D
B	H	D
C	F	E
D	E	F
*E	A	E
F	F	B
G	E	F
H	F	E

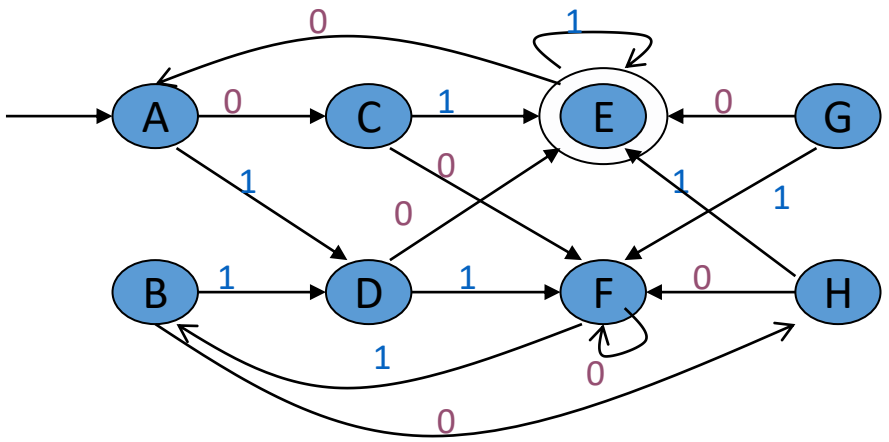


	A	B	C	D	E	F	G	H
A	=							
B		=						
C	X	X	=					
D	X	X		=				
E	X	X	X	X	=			
F					X	=		
G	X	X			X		=	
H	X	X			X			=

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

Table Filling Algorithm - step by step

States	0	1
→A	C	D
B	H	D
C	F	E
D	E	F
*E	A	E
F	F	B
G	E	F
H	F	E

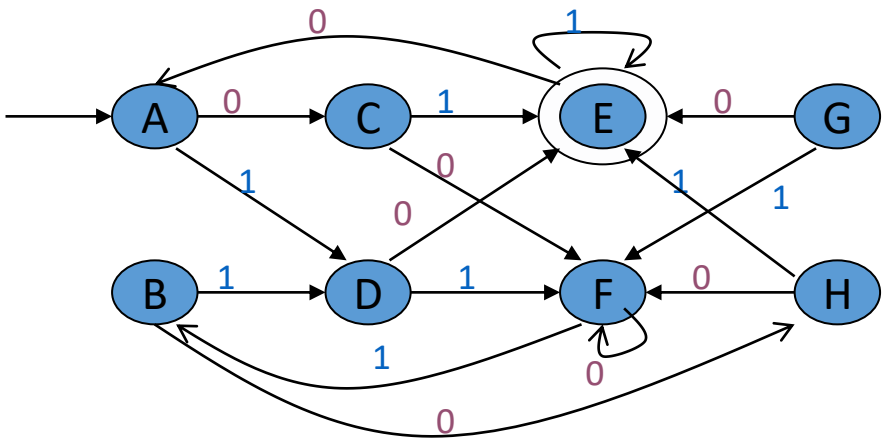


	A	B	C	D	E	F	G	H
A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X		X	=		
G	X	X	X		X		=	
H	X	X	=		X			=

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

Table Filling Algorithm - step by step

States	0	1
→A	C	D
B	H	D
C	F	E
D	E	F
*E	A	E
F	F	B
G	E	F
H	F	E

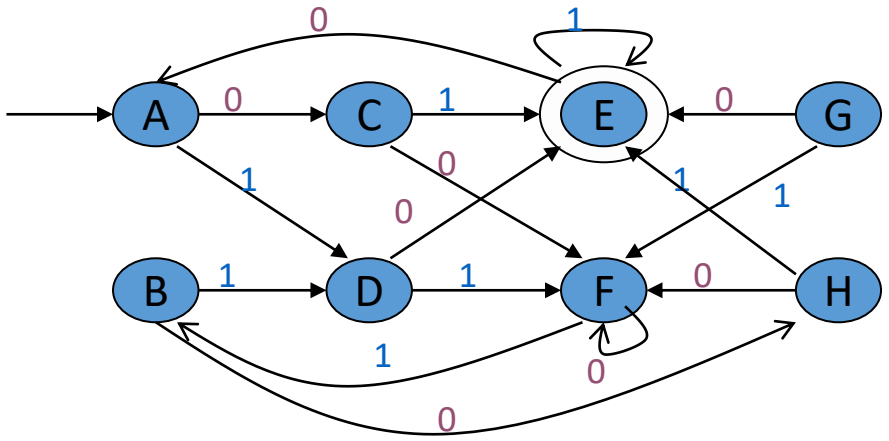


A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X	X	X	=		
G	X	X	X	=	X		=	
H	X	X	=	X	X			=
	A	B	C	D	E	F	G	H

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

Table Filling Algorithm - step by step

States	0	1
→A	C	D
B	H	D
C	F	E
D	E	F
*E	A	E
F	F	B
G	E	F
H	F	E

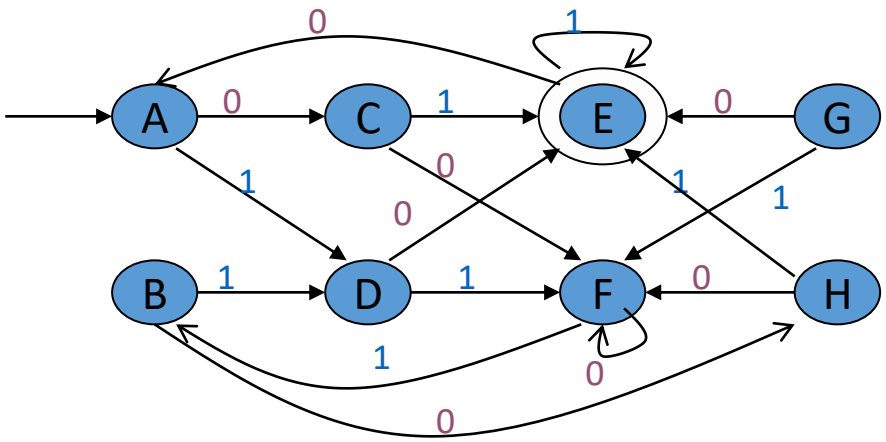


A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X		=
	A	B	C	D	E	F	G	H

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

Table Filling Algorithm - step by step

States	0	1
→A	C	D
B	H	D
C	F	E
D	E	F
*E	A	E
F	F	B
G	E	F
H	F	E

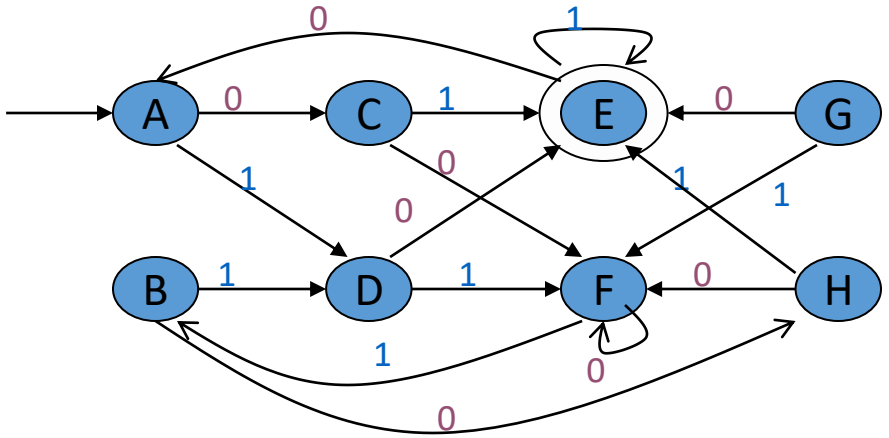


A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X	X	=
	A	B	C	D	E	F	G	H

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

Table Filling Algorithm - step by step

States	0	1
→A	C	D
B	H	D
C	F	E
D	E	F
*E	A	E
F	F	B
G	E	F
H	F	E



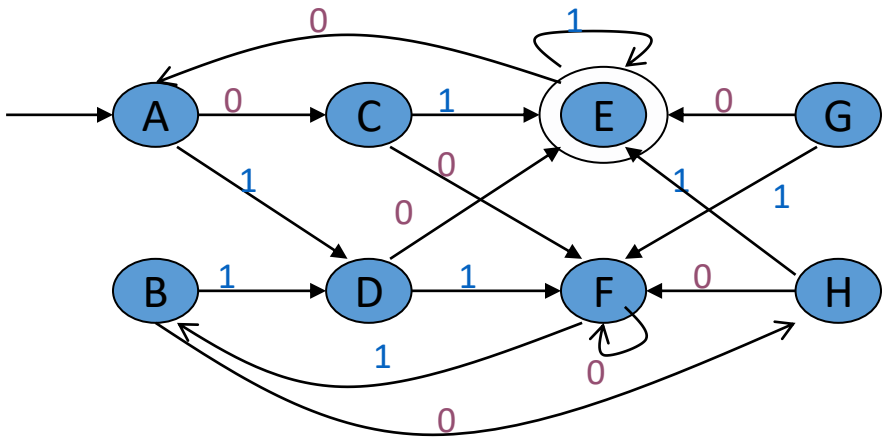
A	=							
B	=	=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F	X	X	X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X	X	=
	A	B	C	D	E	F	G	H

1. Mark X between accepting vs. non-accepting state
2. Pass 1:
   Look 1- hop away for distinguishing states or strings
3. Pass 2:
   Look 1-hop away again for distinguishing states or strings

continue….

Table Filling Algorithm - step by step

States	0	1
→A	C	D
B	H	D
C	F	E
D	E	F
*E	A	E
F	F	B
G	E	F
H	F	E



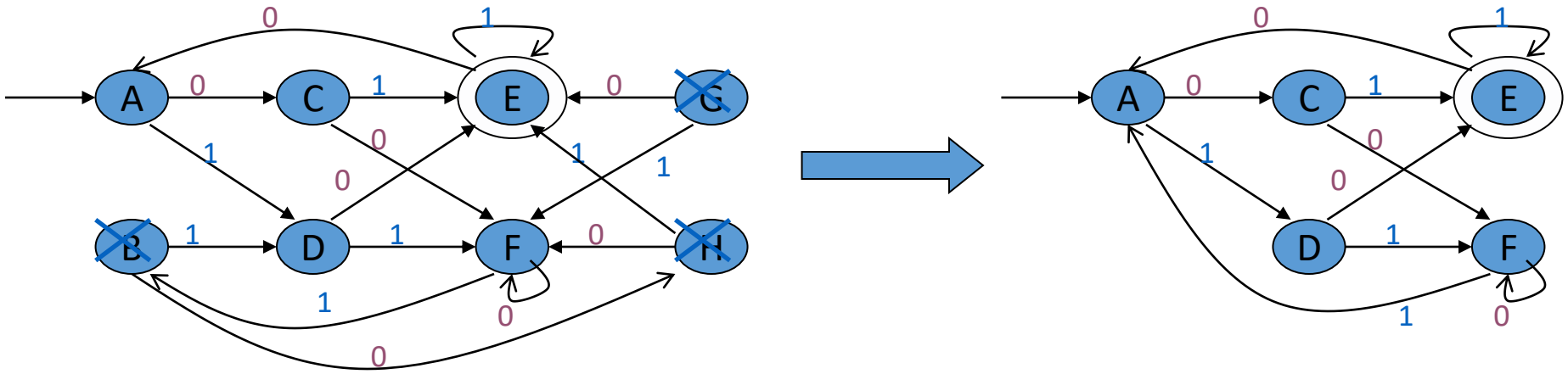
	A	B	C	D	E	F	G	H
A	=							
B	=	=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F	X	X	X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X	X	=

1. Mark X between accepting vs. non-accepting state
2. Pass 1:
   Look 1- hop away for distinguishing states or strings
3. Pass 2:
   Look 1-hop away again for distinguishing states or strings

continue….

Equivalences:
- A=B
- C=H
- D=G

Table Filling Algorithm - step by step



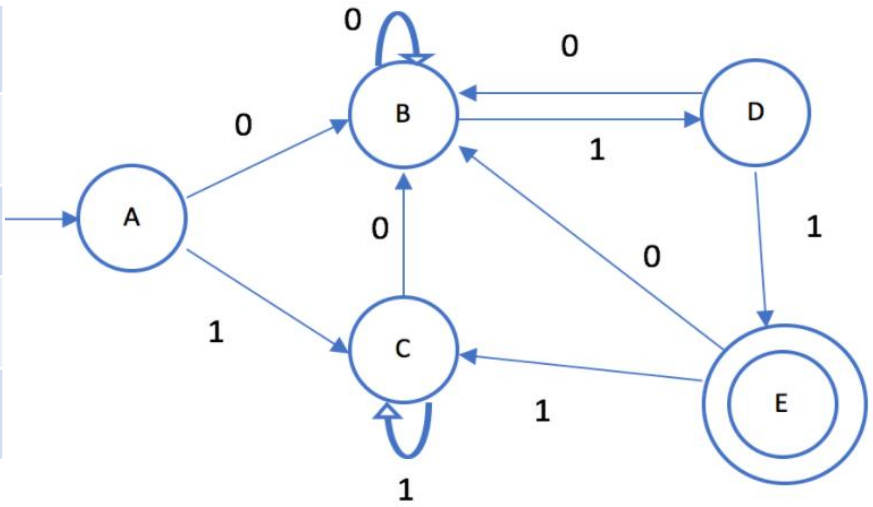
Retrain only one copy for each equivalence set of states

Equivalences:
- A=B
- C=H
- D=G

Example 2

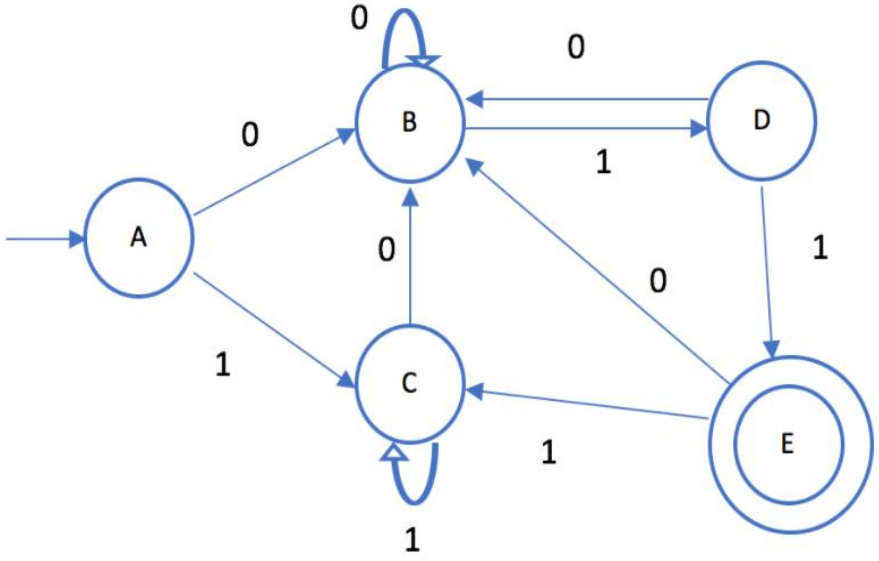
A	=				
B		=			
C			=		
D				=	
E					=
	A	B	C	D	E

States	0	1
→A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

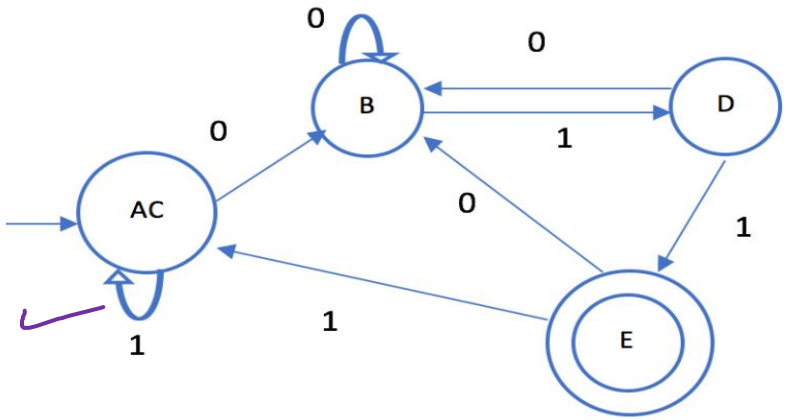


Example 2

States	0	1
→A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

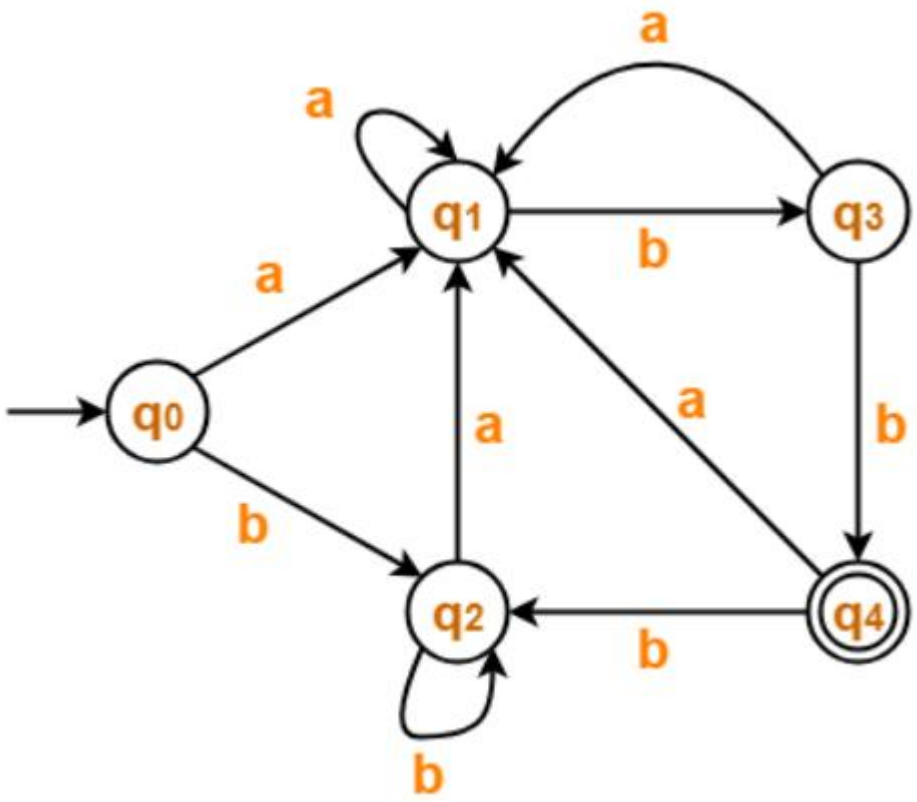


A	=				
B	x	=			
C	=	x	=		
D	x	x	x	=	
E	x	x	x	x	=
	A	B	C	D	E



Example 3

States	a	b
→**q0**	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	*q4
***q4**	q1	q2

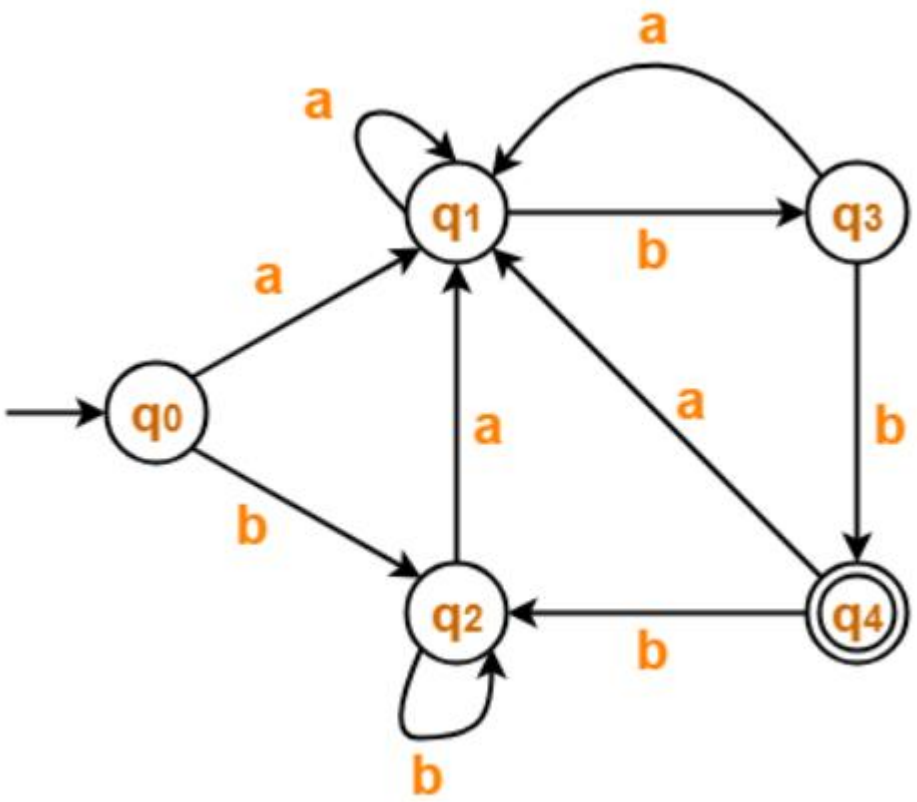


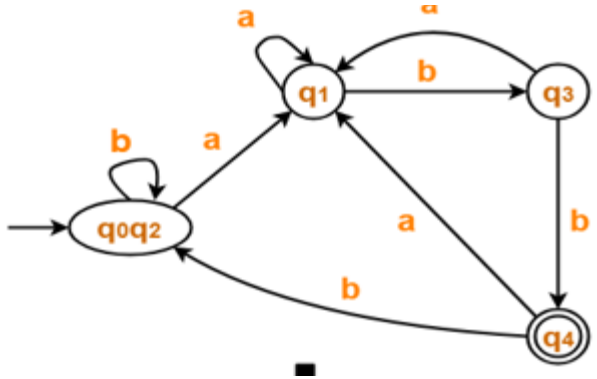
q0	=				
q1		=			
q2			=		
q3				=	
q4					=
	q0	q1	q2	q3	q4

Example 3

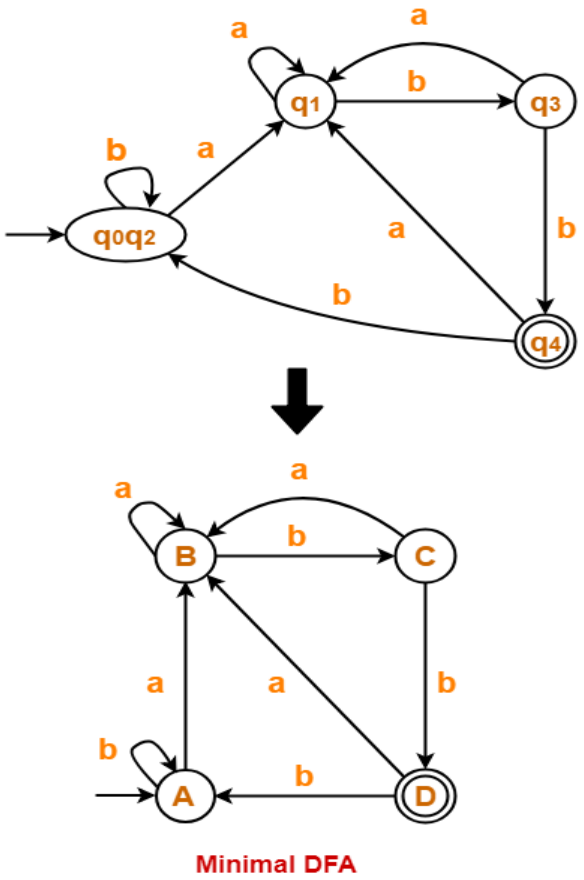
States	a	b
→**q0**	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	*q4
***q4**	q1	q2

q0	=				
q1	X	=			
q2	=	X	=		
q3	X	X	X	=	
q4	X	X	X	X	=
	q0	q1	q2	q3	q4





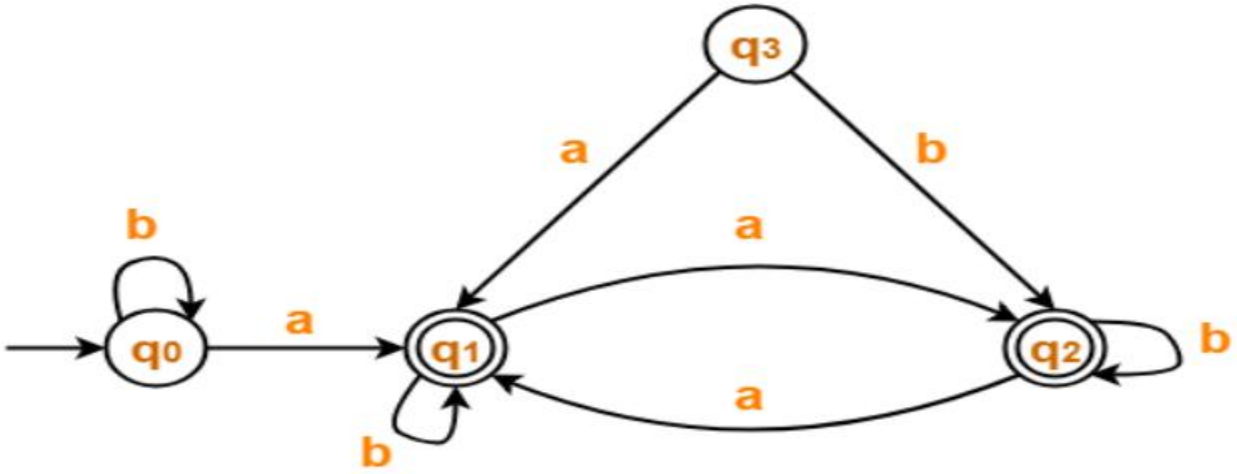
Example 3



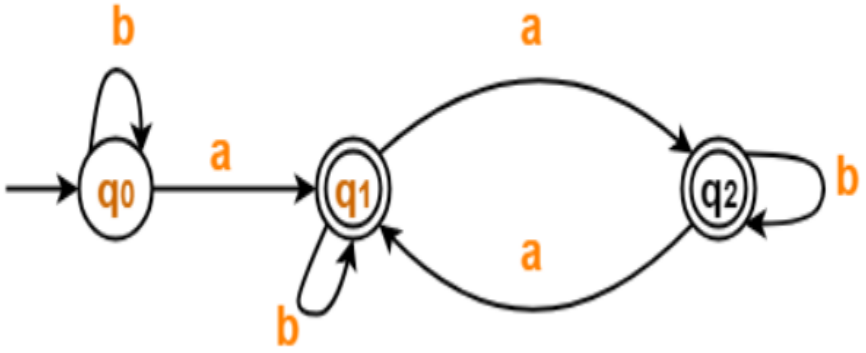
Minimal DFA

States	a	b
→**q0**	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	*q4
***q4**	q1	q2

Example 4

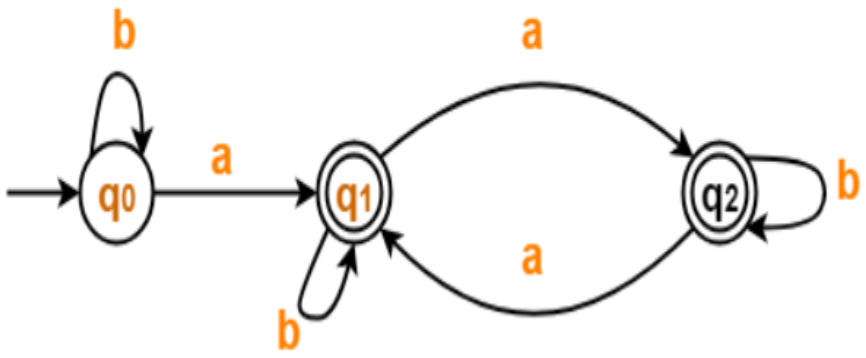


States	a	b
→**q0**	*q1	q0
***q1**	*q2	*q1
***q2**	*q1	*q2



Example 4

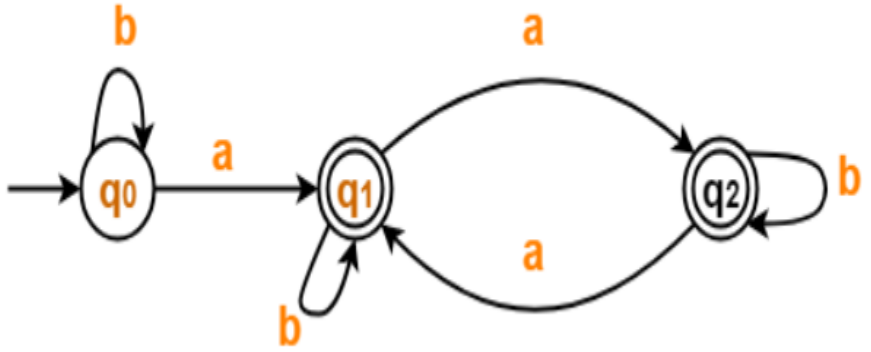
States	a	b
→**q0**	*q1	q0
***q1**	*q2	*q1
***q2**	*q1	*q2



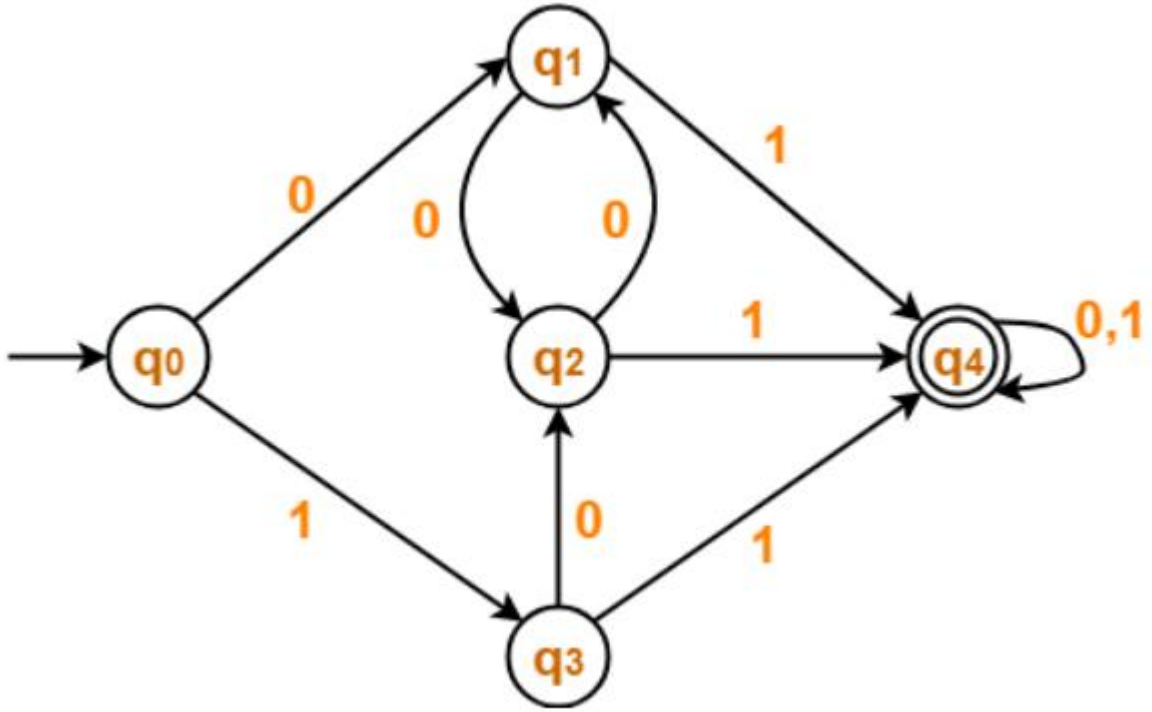
q0	=		
q1		=	
q2			=
	q0	q1	q2

Example 4

States	a	b
→**q0**	q1,q2	q0
***q1,q2**	q1,q2	q1,q2



Example 5



States	0	1
→**q0**	q1	q3
q1	q2	*q4
q2	q1	*q4
q3	q2	*q4
***q4**	*q4	*q4

https://www.gatevidyalay.com/minimization-of-dfa-minimize-dfa-example/

Example 5

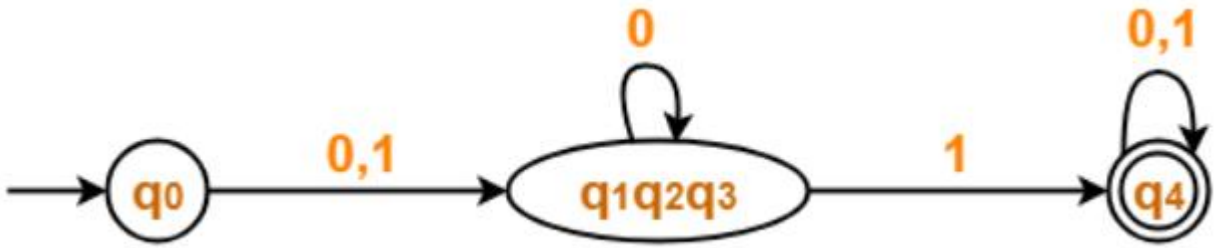
States	0	1
→**q0**	q1	q3
q1	q2	*q4
q2	q1	*q4
q3	q2	*q4
***q4**	*q4	*q4

q0	=				
q1		=			
q2			=		
q3				=	
q4					=
	q0	**q1**	**q2**	**q3**	**q4**

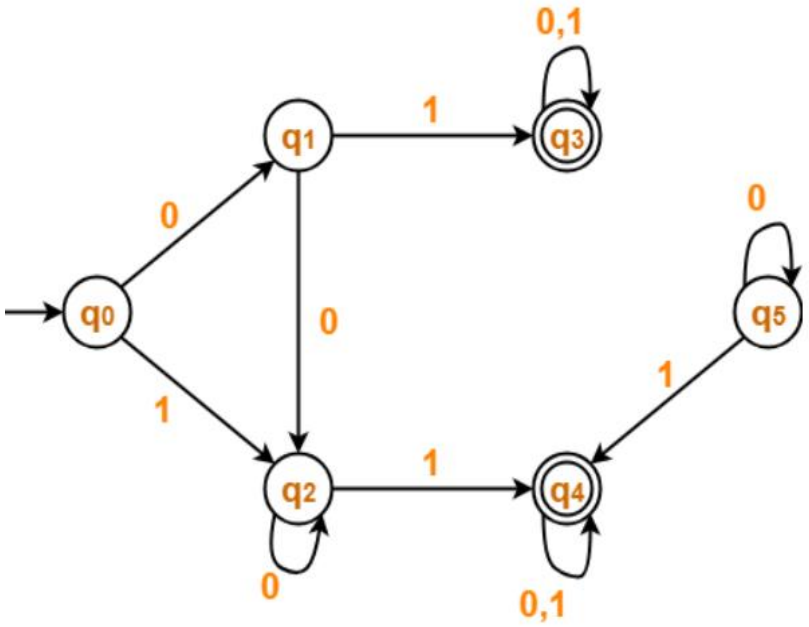
Example 5

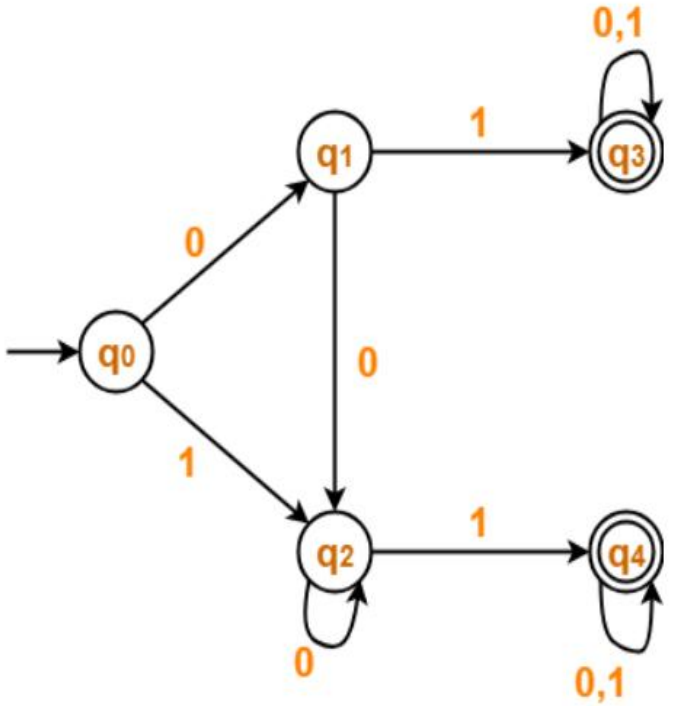
States	0	1
→**q0**	q1	q3
q1	q2	*q4
q2	q1	*q4
q3	q2	*q4
***q4**	*q4	*q4

q0	=				
q1	X	=			
q2	X	=	=		
q3	X	=	=	=	
q4	X	X	X	X	=
	q0	**q1**	**q2**	**q3**	**q4**



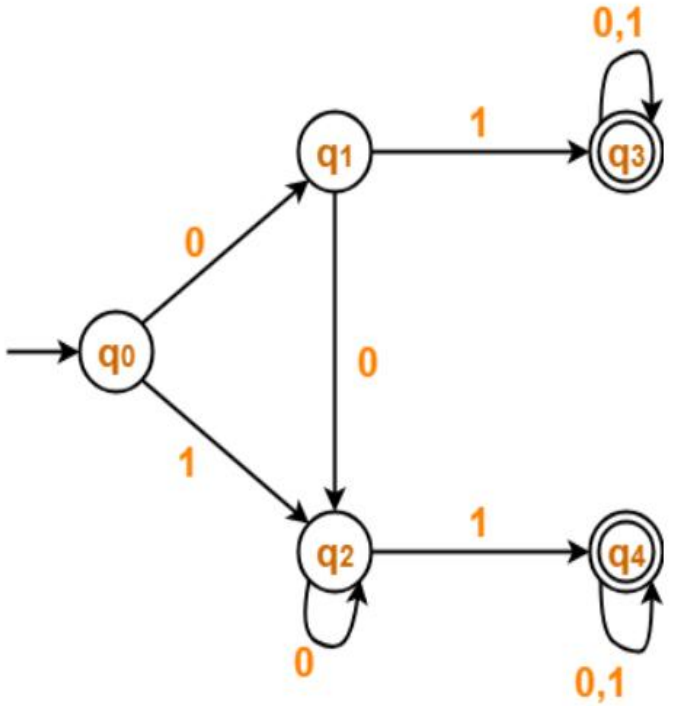
Example 6





Example 6

States	0	1
→**q0**	q1	q2
q1	q2	*q3
q2	q2	*q4
***q3**	*q3	*q3
***q4**	*q4	*q4



Example 6

q0	=				
q1		=			
q2			=		
q3				=	
q4					=
	q0	**q1**	**q2**	**q3**	**q4**

States	0	1
→**q0**	q1	q2
q1	q2	*q3
q2	q2	*q4
***q3**	*q3	*q3
***q4**	*q4	*q4

Example 6

q0	=				
q1	X	=			
q2	X	=	=		
q3	X	X	X	=	
q4	X	X	X	=	=
	q0	**q1**	**q2**	**q3**	**q4**

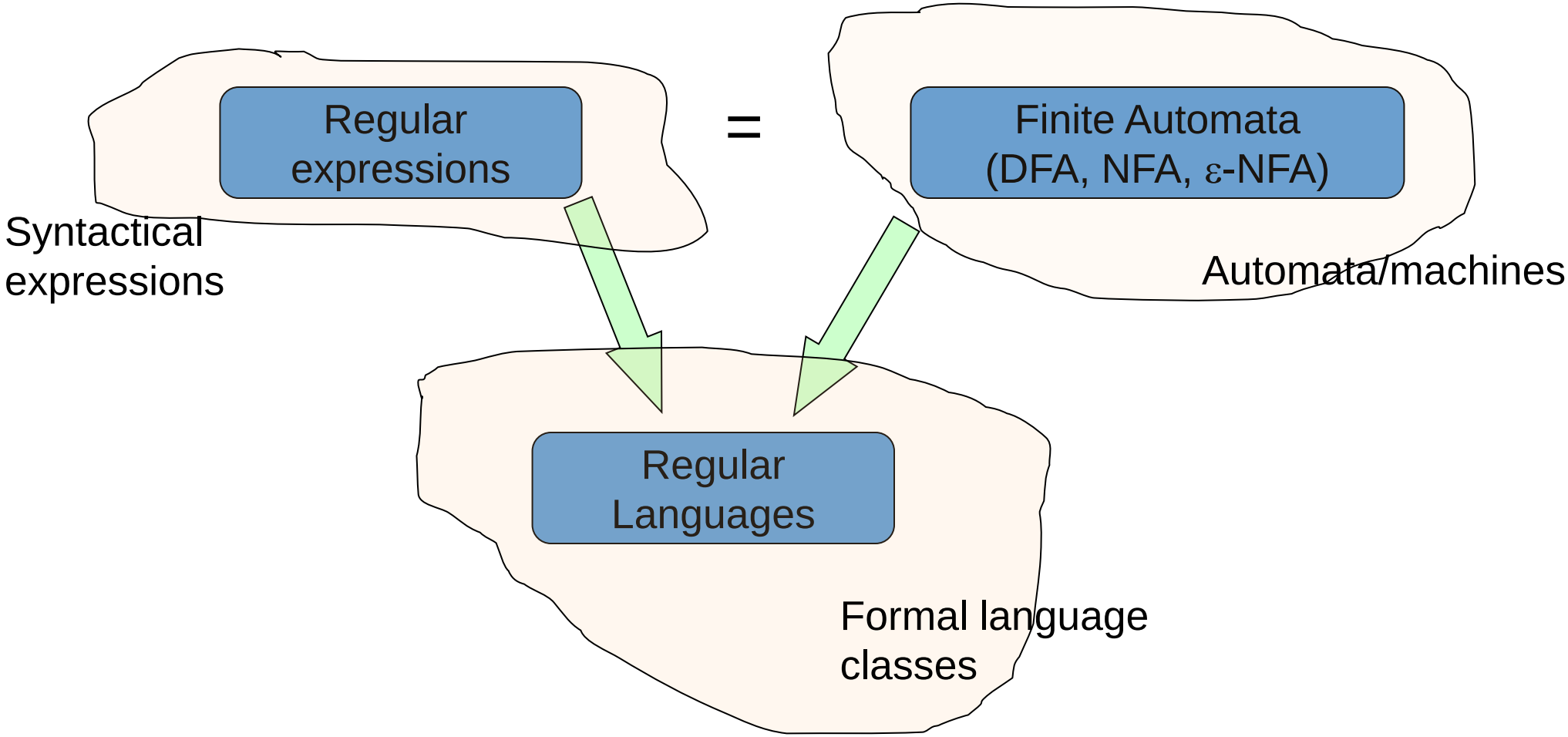
Regular Expressions

Regular Expressions vs. Finite Automata

- Offers a declarative way to express the pattern of any string we want to accept
  - E.g., 01*+ 10*
- Automata => more machine-like

  < input: string , output: [accept/reject] >
- Regular expressions => more program syntax-like
- Unix environments heavily use regular expressions
  - E.g., bash shell, grep, vi & other editors, sed
- Perl scripting – good for string processing
- Lexical analyzers such as Lex or Flex

Regular Expressions



Language Operators

- Union of two languages:
  - **L U M** = all strings that are either in L or M
  - Note: A union of two languages produces a third language

- Concatenation of two languages:
  - **L . M** = all strings that are of the form *xy*
    s.t., x ∈ L and y ∈ M
  - The *dot* operator is usually omitted
    - i.e., **LM** is same as L.M

"i" here refers to how many strings to concatenate from the parent language L to produce strings in the language L^i

Kleene Closure (the * operator)

- Kleene Closure of a given language L:
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{w \mid \text{for some } w \in L\}$
  - $L^2 = \{ w_1w_2 \mid w_1 \in L, w_2 \in L \text{ (duplicates allowed)}\}$
  - $L^i = \{ w_1w_2\ldots w_i \mid \text{all w's chosen are} \in L \text{ (duplicates allowed)}\}$
  - (Note: the choice of each w_i is independent)
  - $L^* = \bigcup_{i \geq 0} L^i$ (arbitrary number of concatenations)

Example:

- Let L = { 1, 00}
  - $L^0 = \{\varepsilon\}$
  - $L^1 = \{1,00\}$
  - $L^2 = \{11,100,001,0000\}$
  - $L^3 = \{111,1100,1001,10000,000000,00001,00100,0011\}$
  - $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$

Kleene Closure (special notes)

- L^* is an infinite set iff $|L| \geq 1$ and $L \neq \{\varepsilon\}$
- If $L=\{\varepsilon\}$, then $L^* = \{\varepsilon\}$
- If $L = \Phi$, then $L^* = \{\varepsilon\}$

Σ^* denotes the set of all words over an alphabet Σ

- Therefore, an abbreviated way of saying there is an arbitrary language L over an alphabet Σ is:
  - $L \subseteq \Sigma^*$

Building Regular Expressions

- Let E be a regular expression and the language represented by E is L(E)

- Then
  - (E) = E
  - L(E + F) = L(E) U L(F)
  - L(E F) = L(E) L(F)
  - L(E*) = (L(E))*

- a+Φ=a
- a+Ɛ= a+Ɛ
- a. Φ= Φ
- a. Ɛ=a

Examples

Write regular expression to accept

- One occurrence of a
- One or more occurrence of a
- Zero or more occurrence of ab
- Strings of a’s and b’s of any length
- Strings of a’s and b’s of length four
- Alternate occurrence of a’s and b’s
- Strings of a’s and b’s with atleast 1 a’s
- Strings of a’s and b’s exactly 2 a’s

Examples

Write regular expression to accept

- Strings with o's and 1's with even length
- Strings with odd number of 1's
- Strings of o's and 1's with length 6 or less
- Strings of o's and 1's ending with 1 and not containing consecutive zero's
- String with next to last symbol is o
- Strings of a's and b's beginning and ending with same character
- Strings of a's and b's with atleast one a and one b

Examples

- Build an NFA for the following language:
  L = {aWa | ∑ ={a , b}
- Build an NFA for the following language:
  L={ab^3Wb^2 : W € {a,b}*}
- Build an NFA for the following language:
  L = { w | where leftmost symbol differs from rightmost symbol, ∑ ={a,b}}
- Build an NFA for the following language:
  L = { w | w is $abab^n$ or aba^n |n>=0}
- Build a DFA for the following language:
  L = {W: |W| mod 5 !=0, W €(0,1)}

For Σ= {*a*,*b*}, onstruct dfa's that accept the sets consisting of

(a) all strings with exactly one *a*,

(b) all strings with at least one a,

(c) all strings with no more than three a's,

Example

1. L = { w | w is a binary string which does not contain two consecutive 0s or two consecutive 1s anywhere)

E.g., w = 01010101 is in L, while w = 10010 is not in L

2. L2 = { w | w is a binary string which ends with 1 does not contain 00}.

E.g., w = 01010101 is in L, while w = 10010 is not in L

3. L3 = { w | w is a binary string which ends with 01}.

E.g., w = 00101 is in L, while w = 10010 is not in L

4. L3 = { w | w is a binary string which begins with 01}.

E.g., w = 00101 is in L, while w = 10010 is not in L

Example

- *L = { w | w is a binary string which does not contain two consecutive 0s or two consecutive 1s anywhere)*
  - E.g., w = 01010101 is in L, while w = 10010 is not in L
- Goal: Build a regular expression for L
- Four cases for w:
  - Case A: w starts with 0 and |w| is even
  - Case B: w starts with 1 and |w| is even
  - Case C: w starts with 0 and |w| is odd
  - Case D: w starts with 1 and |w| is odd
- Regular expression for the four cases:
  - Case A: (01)*
  - Case B: (10)*
  - Case C: 0(10)*
  - Case D: 1(01)*
- Since L is the union of all 4 cases:
  - Reg Exp for L = (01)* + (10)* + 0(10)* + 1(01)*
- If we introduce ε then the regular expression can be simplified to:
  - Reg Exp for L = (ε +1)(01)*(ε +0)

Some more examples:

- ***L2 = { w | w is a binary string which ends with 1 does not contain 00}.***
  - E.g., w = 01010101 is in L, while w = 10010 is not in L
- Goal: Build a regular expression for L2
  - Reg Exp for L = (1+01)*(1+01)

- ***L3 = { w | w is a binary string which ends with 01}.***
- Goal: Build a regular expression for L3
  - Reg Exp for L = (1+0)*01

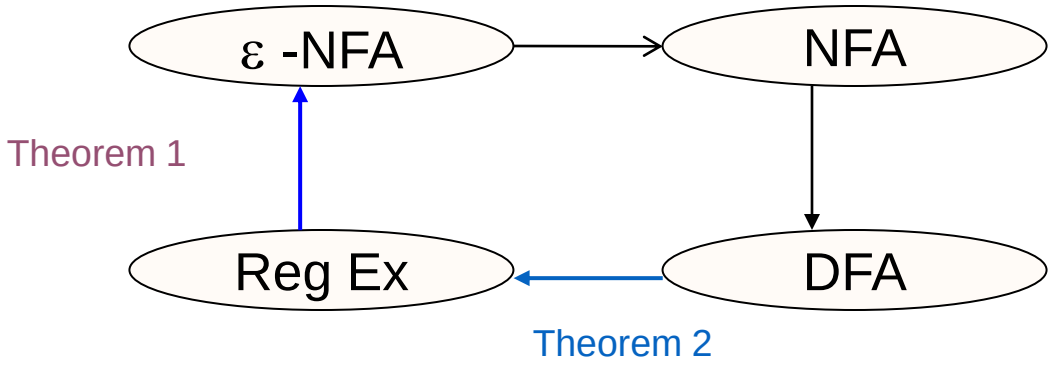
Precedence of Operators

- Highest to lowest
  - * operator (star)
  - . (concatenation)
  - + operator
- Example:
  - 01* + 1 = (0 . ((1)*)) + 1

Finite Automata (FA) & Regular Expressions (RegEx)

- To show that they are interchangeable, consider the following theorems:
  - *Theorem 1: For every DFA A there exists a regular expression R such that L(R)=L(A)*
  - *Theorem 2: For every regular expression R there exists an ε-NFA E such that L(E)=L(R)*

Proofs
in the book

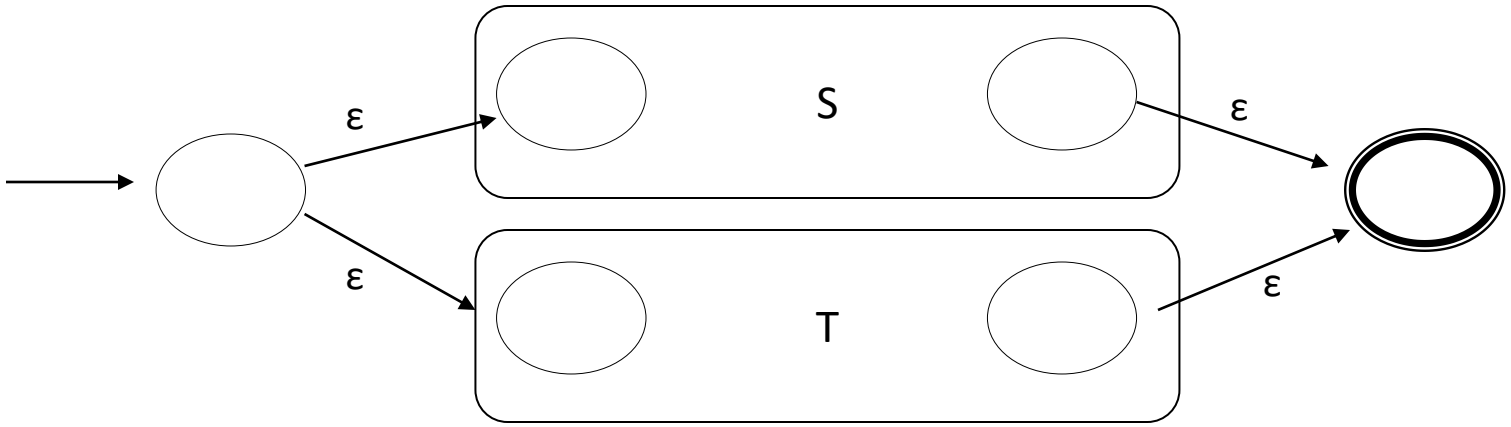


Kleene Theorem

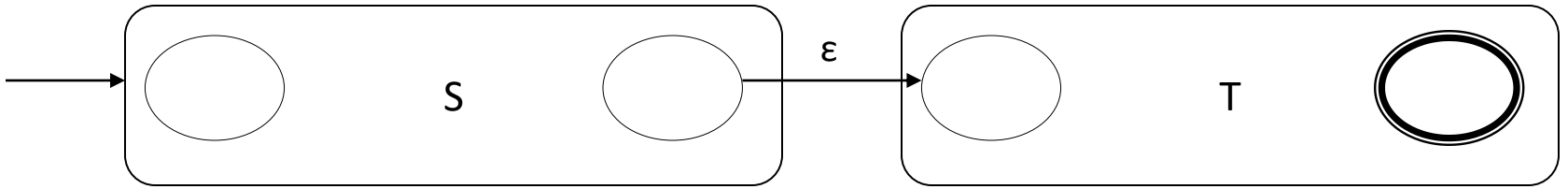
RE to ε-NFA construction

- Suppose ε-NFA$_1$ and ε-NFA$_2$ are the automata for R_1 and R_2
- Three operations to worry about: union $R_1 + R_2$, concatenation R_1R2), closure R_1*
- With ε-transitions, construction is straightforward
  - Union: create a new start state, with ε-transitions into the start states of ε-NFA$_1$ and ε-NFA$_2$; create a new final state, with ε-transitions from the two final states of ε-NFA$_1$ and ε-NFA$_2$
  - Concatenation: ε-transition from final state of ε-NFA$_1$ to the start state of ε-NFA$_2$
  - Closure: closure can be supported by an ε-transition from final to start state; need a few more ε-transitions (why?)

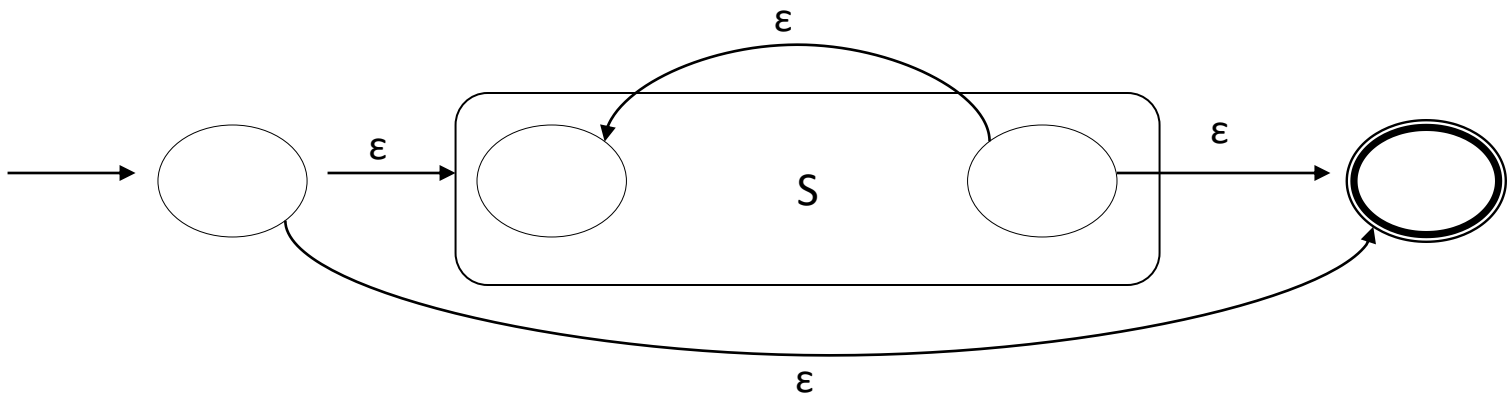
R=S+T



R=ST

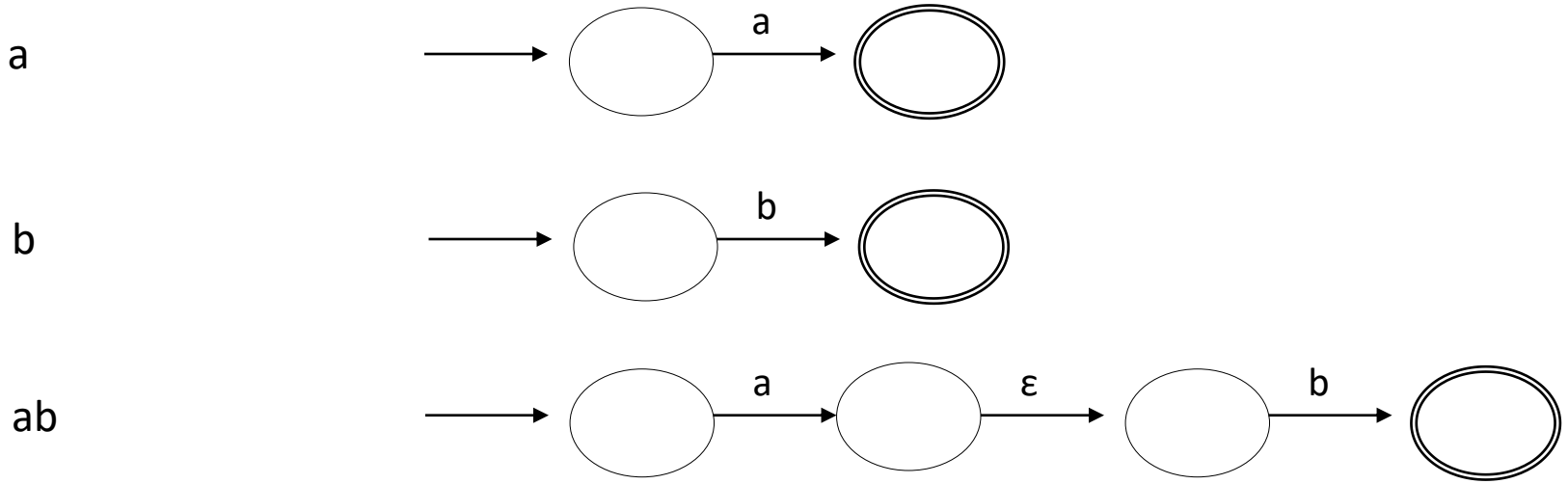


R=S*



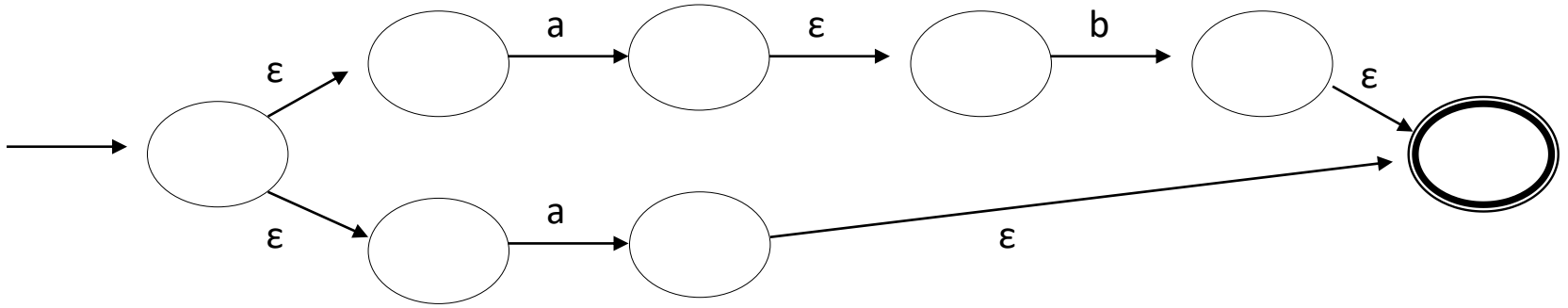
RE to ε-NFA Example

- Convert R= (ab+a)* to an NFA
  - We proceed in stages, starting from simple elements and working our way up

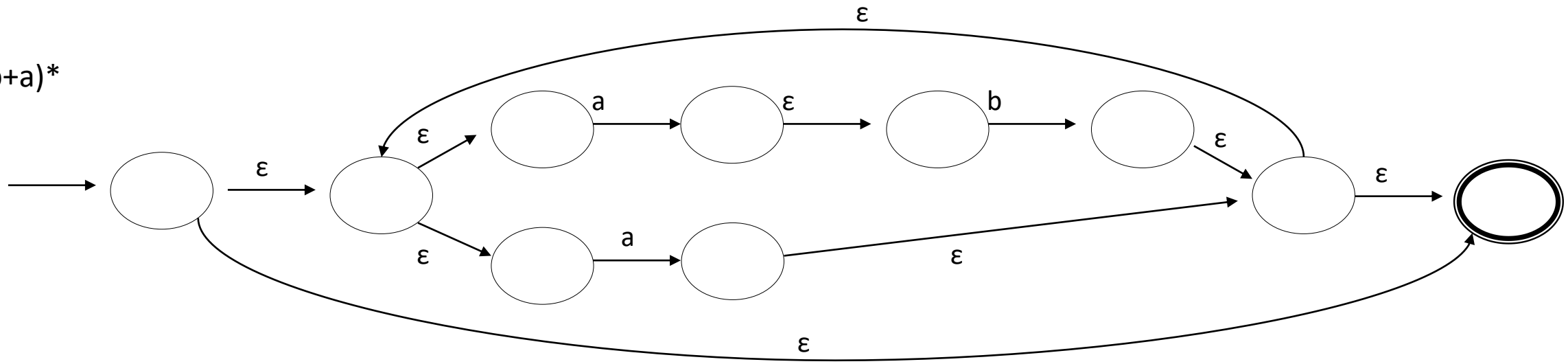


RE to ε-NFA Example (2)

ab+a



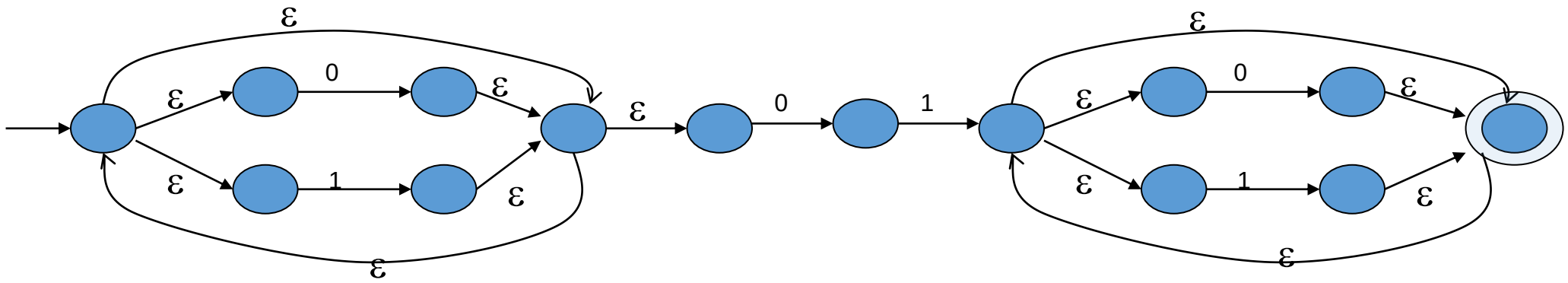
(ab+a)*



RE to ε-NFA construction

Example: (0+1)*01(0+1)*

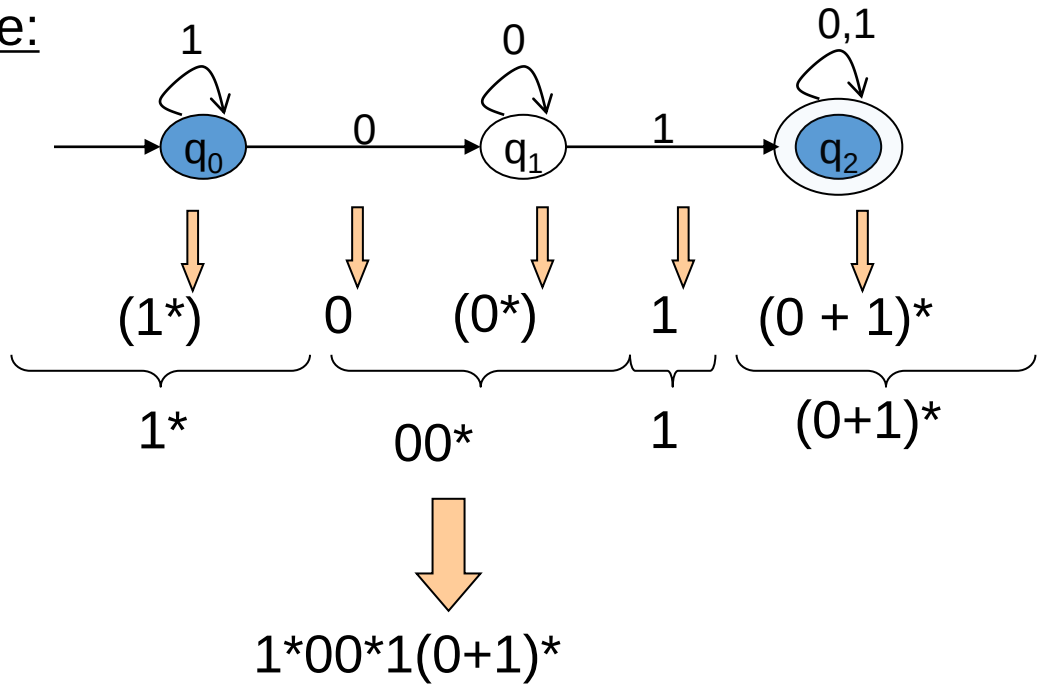
(0+1)* 01 (0+1)*



DFA to RE construction

Informally, trace all distinct paths (traversing cycles only once) from the start state to *each of the* final states and enumerate all the expressions along the way

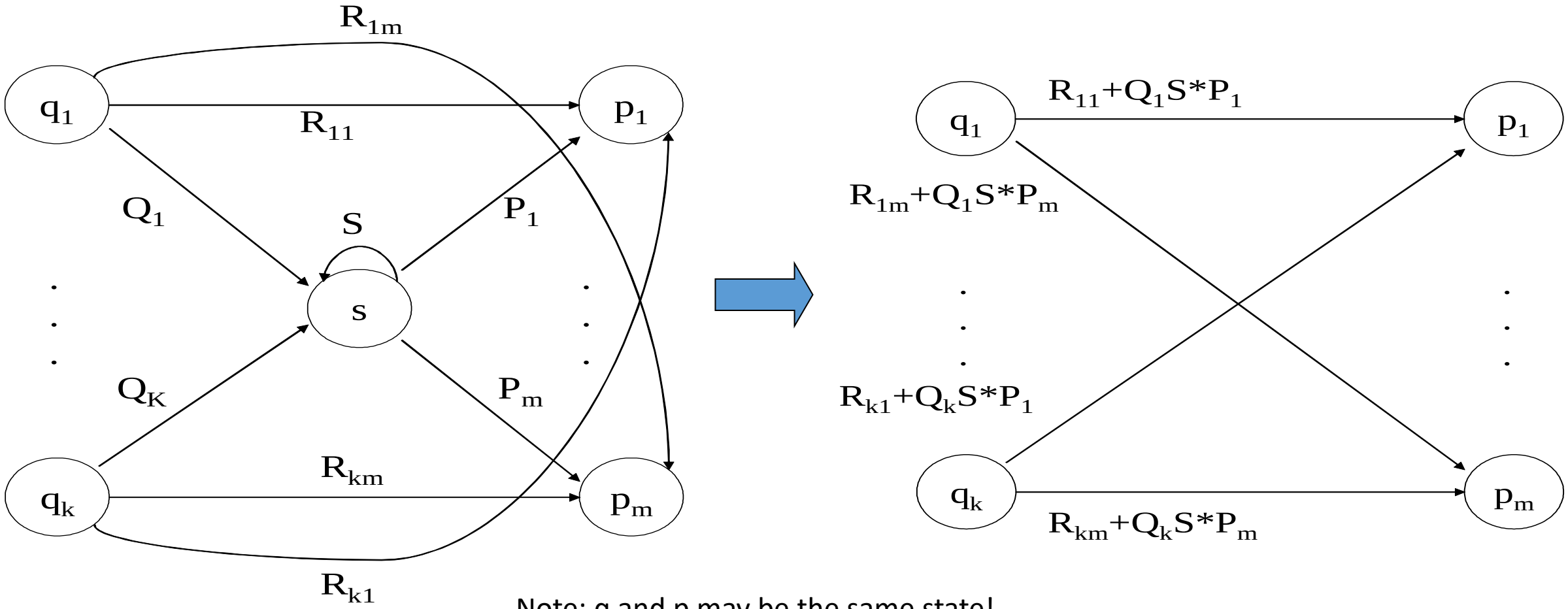
Example:



Q) What is the language?

State Elimination

- Consider the figure below, which shows a generic state s about to be eliminated. The labels on all edges are regular expressions.
- To remove s, we must make labels from each q_i to p_1 up to p_m that include the paths we could have made through s.



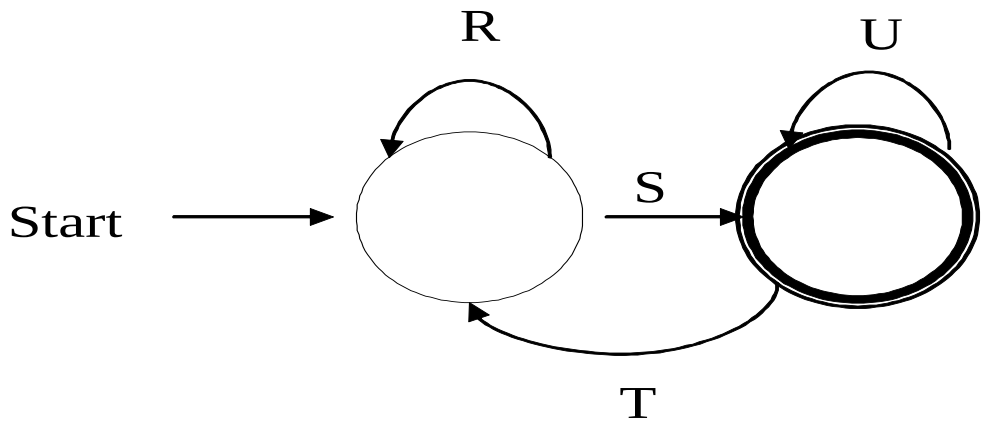
Note: q and p may be the same state!

DFA to RE via State Elimination (1)

1. Starting with intermediate states and then moving to accepting states, apply the state elimination process to produce an equivalent automaton with regular expression labels on the edges.
   - The result will be a one or two state automaton with a start state and accepting state.

DFA to RE State Elimination (2)

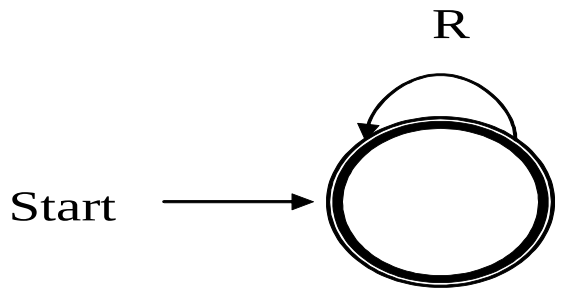
2. If the two states are different, we will have an automaton that looks like the following:



We can describe this automaton as: (R+SU*T)*SU*

DFA to RE State Elimination (3)

3. If the start state is also an accepting state, then we must also perform a state elimination from the original automaton that gets rid of every state but the start state. This leaves the following:

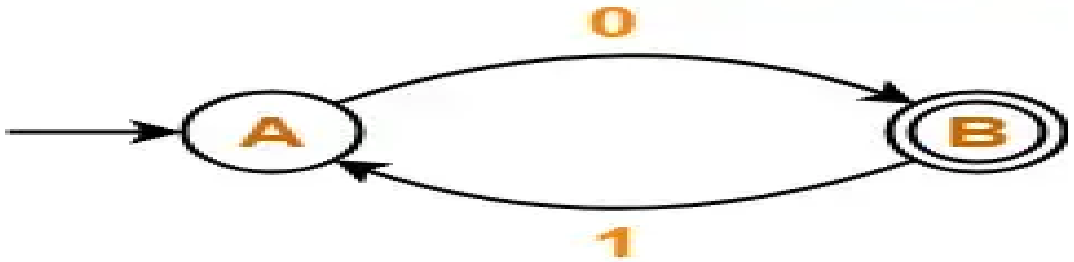


We can describe this automaton as simply R*.

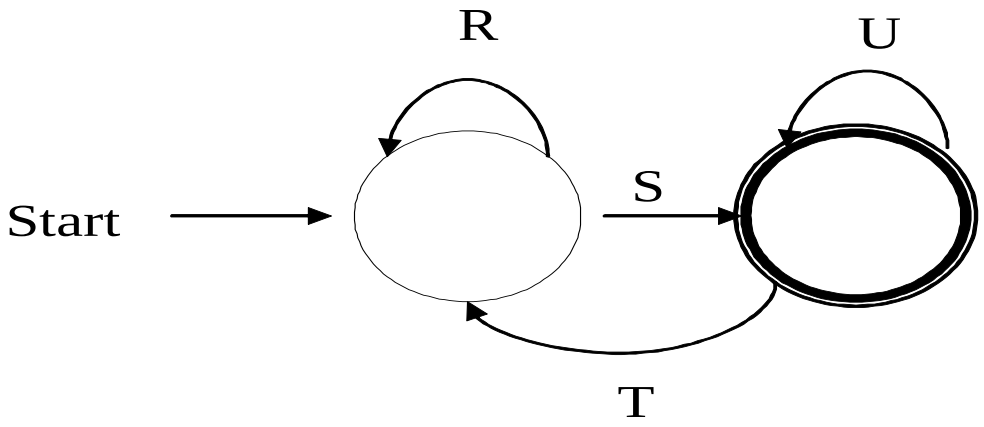
DFA to RE State Elimination (4)

4. If there are n accepting states, we must repeat the above steps for each accepting states to get n different regular expressions, R_1, R_2, … R_n. For each repeat we turn any other accepting state to non-accepting. The desired regular expression for the automaton is then the union of each of the n regular expressions: $R_1 \cup R_2 \ldots \cup R_N$

Example

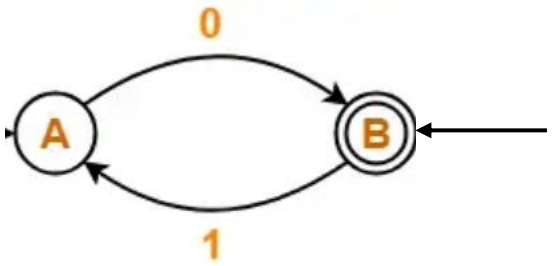


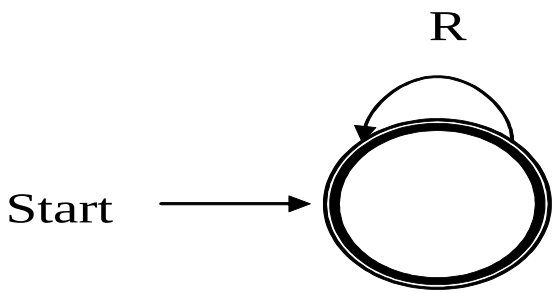
RE=(01)*0



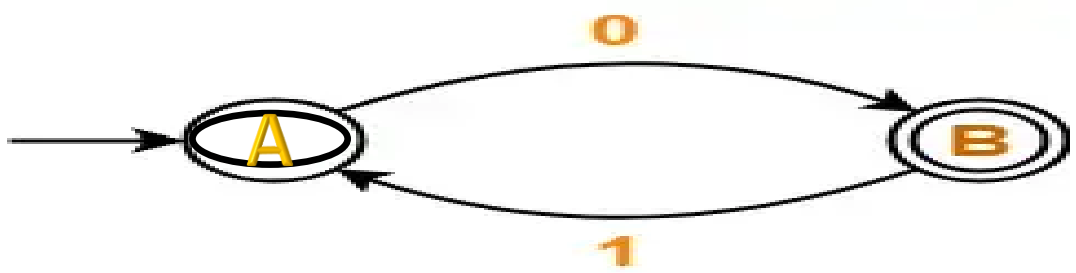
Example

RE=?



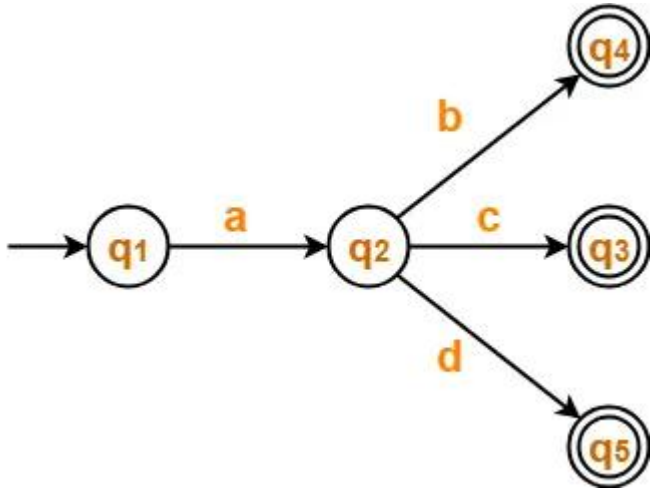


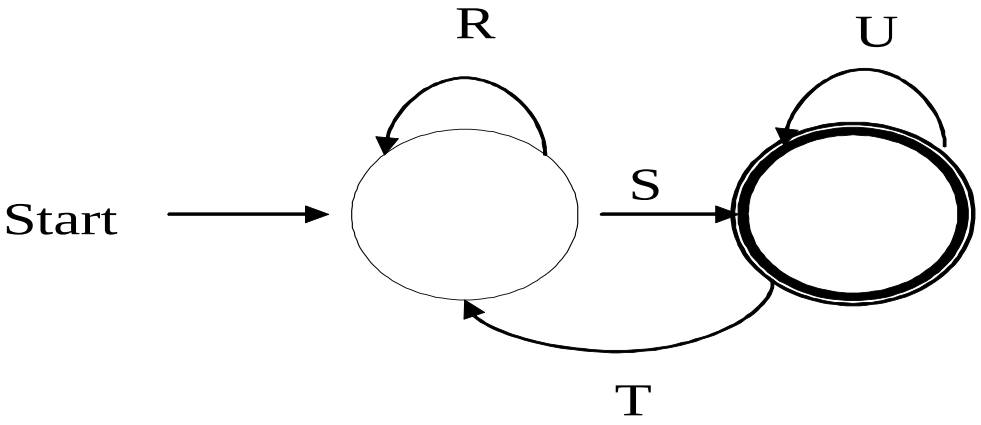
Example



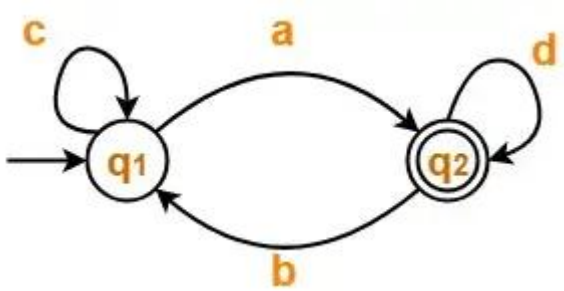
RE1=(01)*0

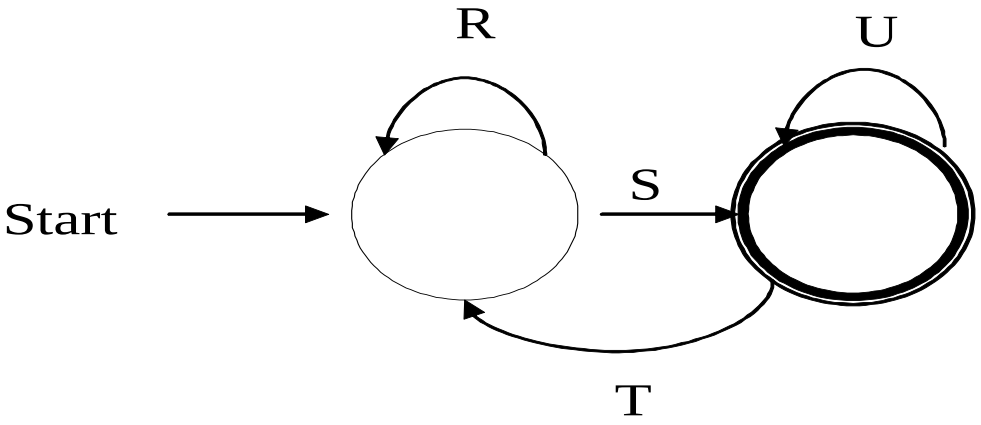
Example 2





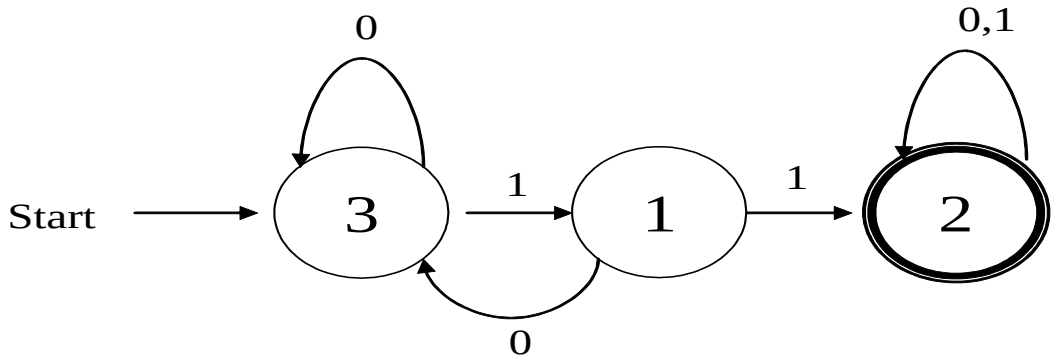
Example



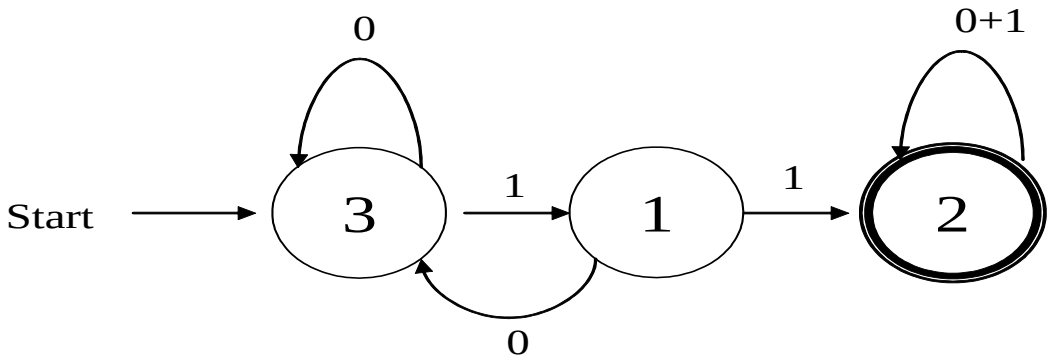


DFA→RE Example

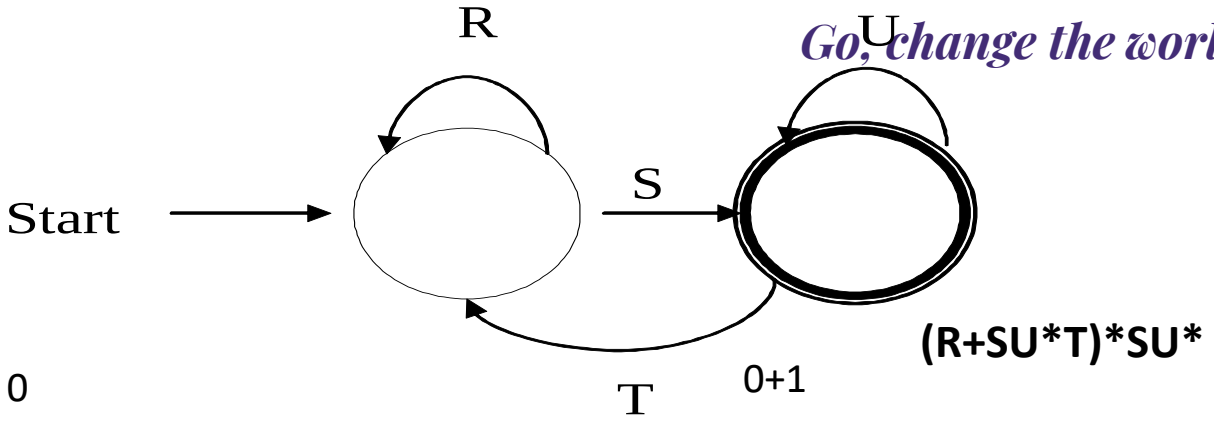
- Convert the following to a RE



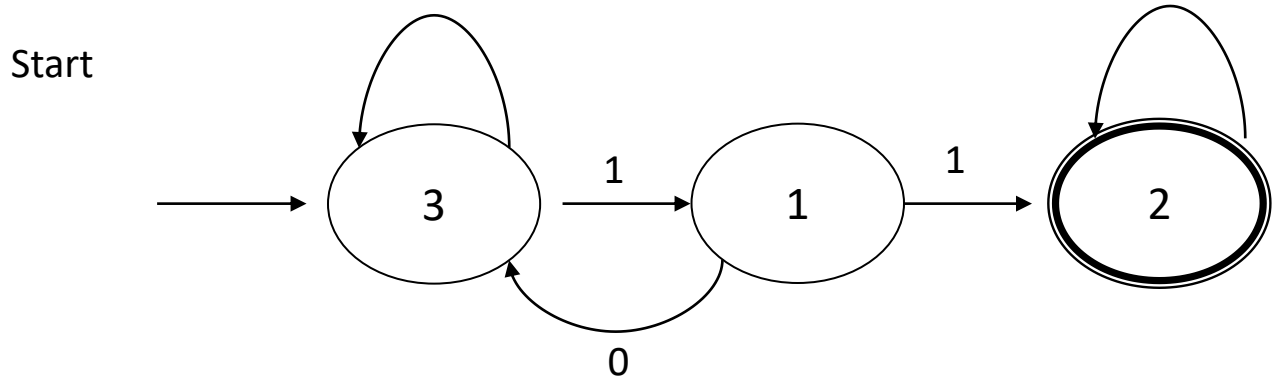
- First convert the edges to RE's:



DFA → RE Example

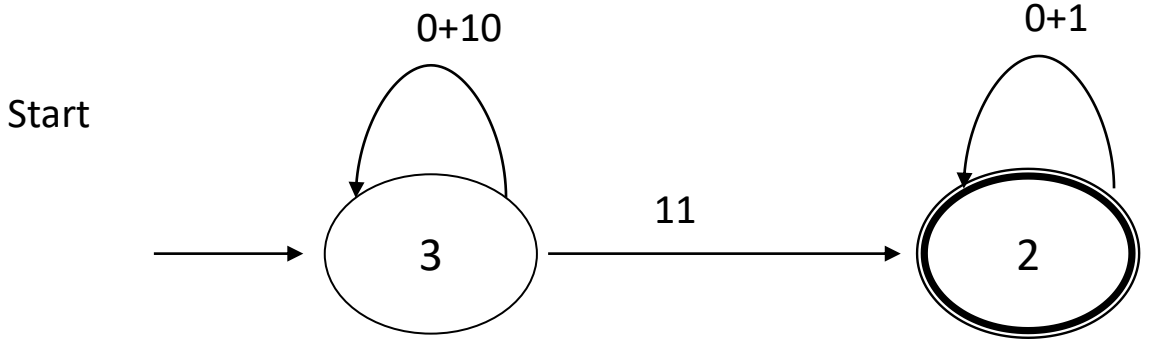


- Eliminate State 1:



- To:

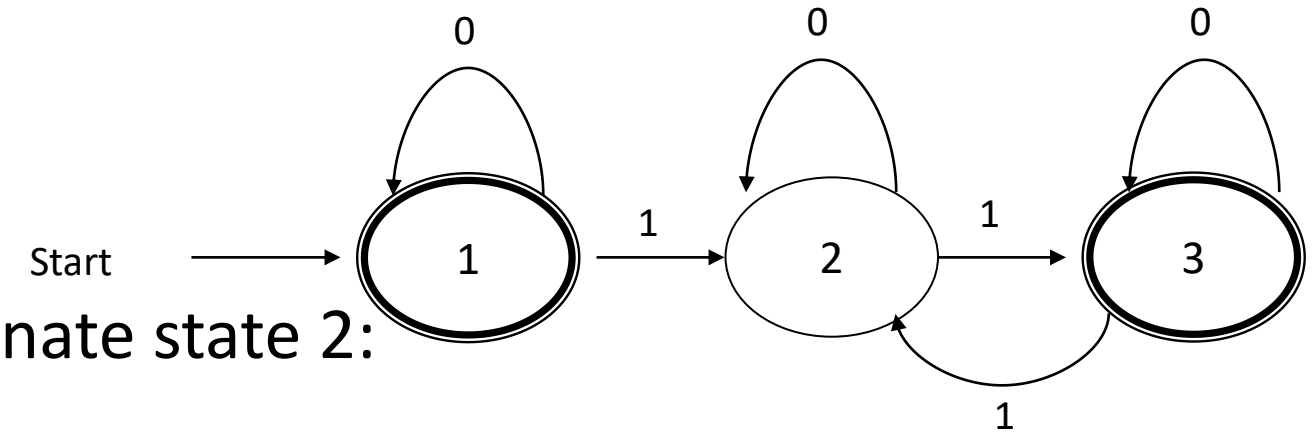
Note edge from 3→3



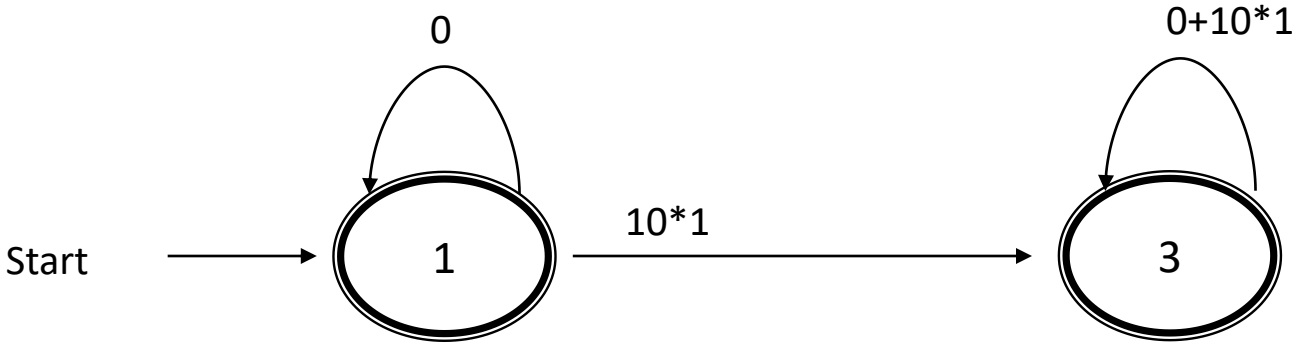
Answer: (0+10)*11(0+1)*

Second Example

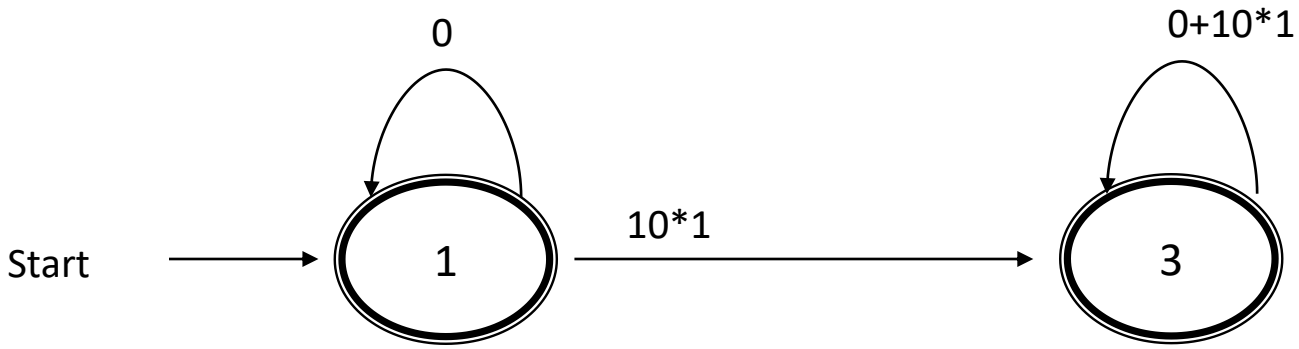
- Automata that accepts even number of 1's



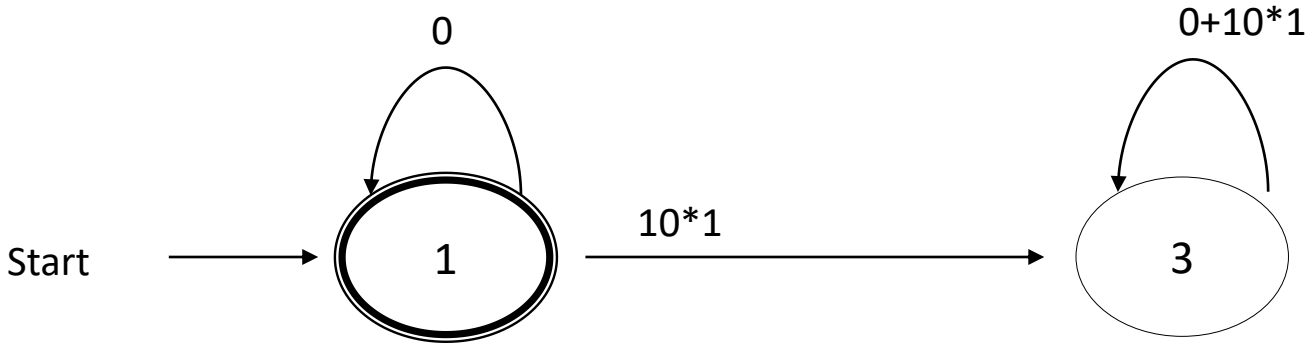
- Eliminate state 2:



Second Example (2)

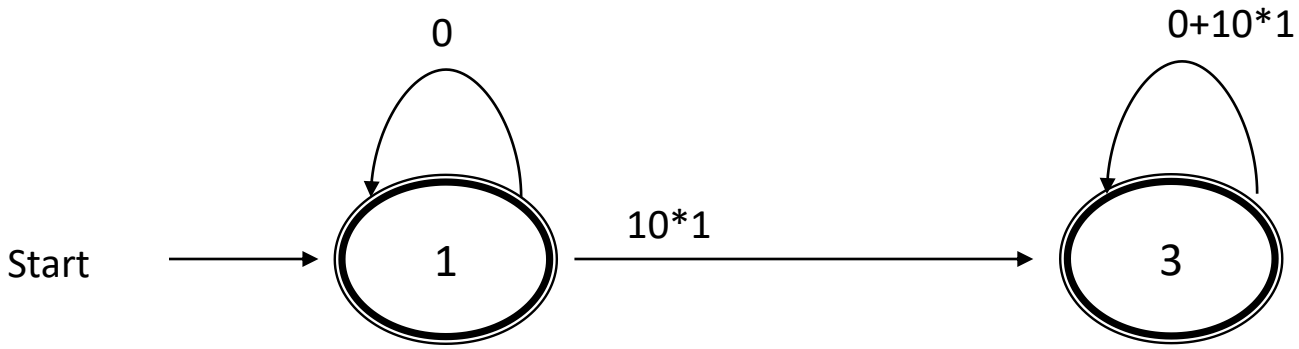


- Two accepting states, turn off state 3 first

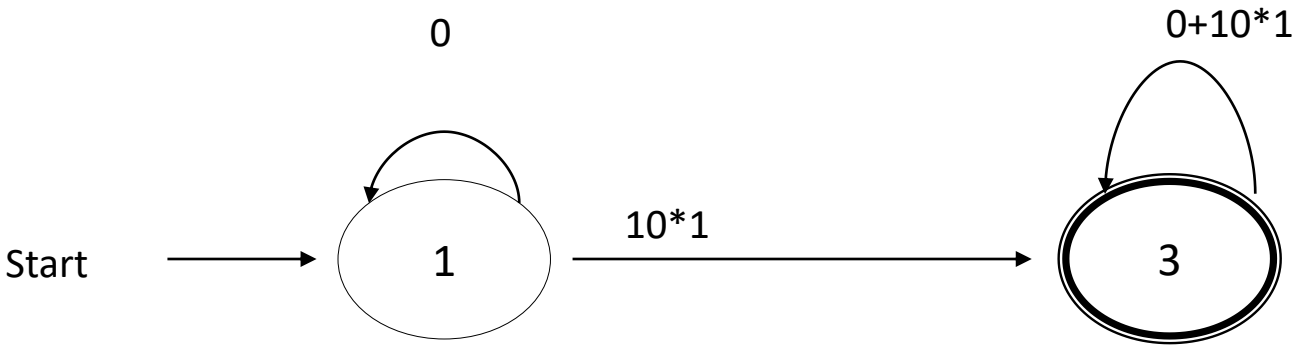


This is just 0*; can ignore going to state 3 since we would “die”

Second Example (3)



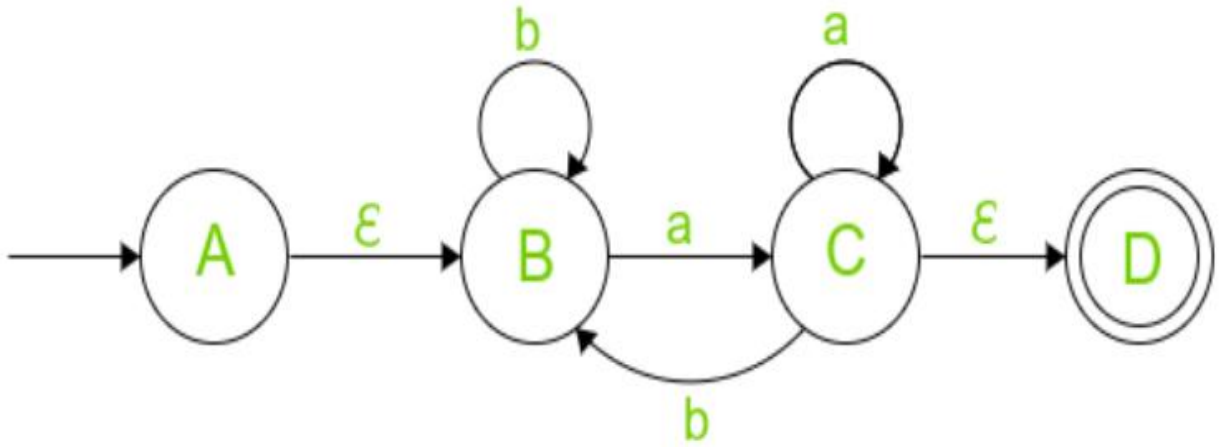
- Turn off state 1 second:

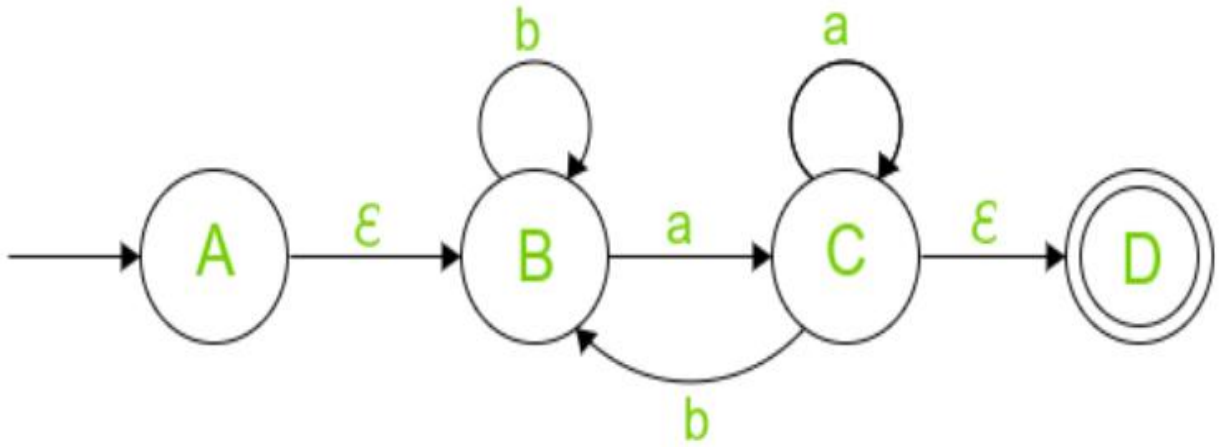


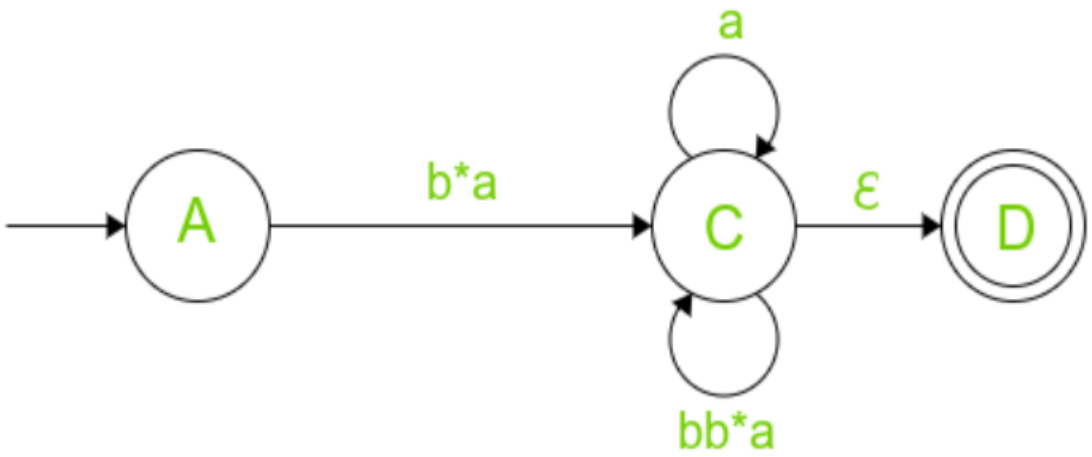
This is just 0*10*1(0+10*1)*

Combine from previous slide to get 0* + 0*10*1(0+10*1)*

Example 3







RE = b*a(bb*a +a)*

Algebraic Laws of Regular Expressions

- Commutative:
  - E+F = F+E
- Associative:
  - (E+F)+G = E+(F+G)
  - (EF)G = E(FG)
- Identity:
  - E+Φ = E
  - ε E = E ε = E
- Annihilator:
  - ΦE = EΦ = Φ

Algebraic Laws…

- Distributive:
  - E(F+G) = EF + EG
  - (F+G)E = FE+GE
- Idempotent: E + E = E
- Involving Kleene closures:
  - (E*)* = E*
  - Φ* = ε
  - ε* = ε
  - E^+ =EE*
  - E? = ε +E

True or False?

Let R and S be two regular expressions. Then:

1. ((R*)*)* = R* ?
2. (R+S)* = R* + S* ?
3. (RS + R)* RS = (RR*S)* ?

Applications of Regular Expressions

- **Regular expression in Unix**

[] , . , Classes of characters[:digit:] , [:alnum:],[:alpha:], |, ?, +, *, (),

R{5}➔RRRRR

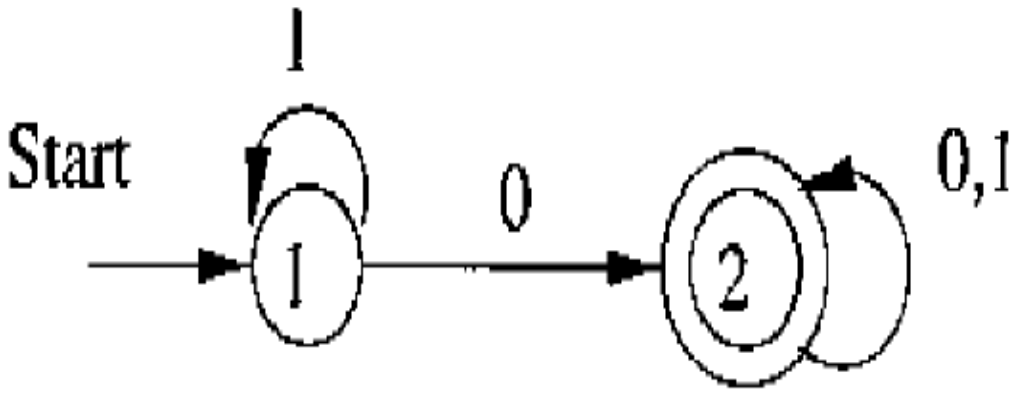
- **Lexical Analysis**
- **Finding Pattern in the text**

Kleen's Theorem

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^{*} R_{kj}^{(k-1)}$$

Path from state I to j that go through no state higher than k

Example- Convert DFA to RE



	By direct substitution	Simplified
$R_{11}^{(2)}$	$1^* + 1^*0(\epsilon + 0 + 1)^*\emptyset$	1^*
$R_{12}^{(2)}$	$1^*0 + 1^*0(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$1^*0(0+1)^*$
$R_{21}^{(2)}$	$\emptyset + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*\emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\epsilon + 0 + 1 + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$(0+1)^*$

	By direct substitution	Simplified
$R_{11}^{(1)}$	$\epsilon + 1 + (\epsilon + 1)(\epsilon + 1)^*(\epsilon + 1)$	1^*
$R_{12}^{(1)}$	$0 + (\epsilon + 1)(\epsilon + 1)^*0$	1^*0
$R_{21}^{(1)}$	$\emptyset + \emptyset(\epsilon + 1)^*(\epsilon + 1)$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon + 0 + 1 + \emptyset(\epsilon + 1)^*0$	$\epsilon + 0 + 1$

$R_{11}^{(0)}$	$\epsilon + 1$
$R_{12}^{(0)}$	0
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$(\epsilon + 0 + 1)$

Summary

- DFA –Definition, Transition diagrams & tables
- Regular language
- NFA- Definition, Transition diagrams & tables
- DFA vs. NFA
- NFA to DFA conversion using subset construction
- Equivalency of DFA & NFA
- Removal of redundant states and including dead states
- ε-transitions in NFA
- Text searching applications
- Simplification of DFAs
  - How to remove unreachable states?
  - How to identify and collapse equivalent states?
  - How to minimize a DFA?
  - How to tell whether two DFAs are equivalent?

Summary

- Regular expressions
- Equivalence to finite automata
- DFA to regular expression conversion
- Regular expression to ε-NFA conversion
- Algebraic laws of regular expressions
- Unix regular expressions and Lexical Analyzer