

MCA III SEMESTER  
CAMS-3003  
THEORY OF COMPUTATION

**Course Introduction**

# Text Books

- 1. Peter Linz.** An Introduction to Formal Languages and Automata, 6<sup>th</sup> Edition, Jones & Bartlett Learning, India.
- 2. Michael Sipser.** Introduction to the Theory of Computation 2<sup>nd</sup> Edition, Thomson Course Technology.
- 3. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman** Introduction to Automata Theory, Languages, and Computation 2<sup>nd</sup> Edition. Addison-Wesley.
- 4. John C. Martin.** Introduction to Languages and the Theory of Computation 3<sup>rd</sup> Edition. McGraw-Hill.
- 5. Harry R. Lewis & Christos H. Papadimitriou.** Elements of the Theory of Computation, 2<sup>nd</sup> Ed., PHI
- 6. K. L. P. Mishra.** Theory of Computer Science, 3<sup>rd</sup> Ed., Prentice Hall India

# Course Goals

**To investigate -**

- What do we exactly mean by computation?
- What are the models for computation?
- Are computers capable of computing anything?

# Course Goals

- **Computation**
  - Any type of calculation that includes both **arithmetical** and **non-arithmetical steps** and follows a well-defined model.
- **Models of computation**
  - FA, PDA, TM, etc.
- **Analyze power of Models**
  - Answer intractability questions:
    - **What computational problems can each model solve?**
  - Answer Time Complexity questions:
    - **How much time do we need to solve the problems?**

**Tractable Problem:** a problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial.

**Intractable Problem:** A problem that cannot be solved by a polynomial-time algorithm.

# Course Goals

- **Are computers capable of computing anything ?**
  - If computers are able to do everything-
    - *How can we claim ?*
  - If there are things computers can not do-
    - *What are they ? And why ?*
- Computers can't solve certain types of problems.
- **Halting Problem**
  - No computer can tell **IN GENERAL** “**Whether or not a given computer program stops after a finite amount of time on a given input**”.

# Theory of Computation (TOC)

- A branch of Theoretical Computer Science that deals with
  - “How **efficiently** problems can be solved on a model of computation, using an algorithm ?”.
- The field is divided into three major branches
  - **Automata theory and languages**
    - Automata theory is the study of abstract machines and automata, as well as the computational problems that can be solved using them.
  - **Computability theory**
    - It studies which problems are computationally solvable using different computation models.
    - It is concerned with whether a problem can be solved at all, given any finite but arbitrarily large amount of resources.
  - **Computational complexity theory**
    - Computational complexity theory focuses on classifying computational problems according to their resource usage and relating these classes to each other.
- The branches are linked by the question:
  - *“What are the fundamental capabilities and limitations of computers?”.*

# Course Topics

## Fundamental Concepts

Unsolvable Problems and Computability

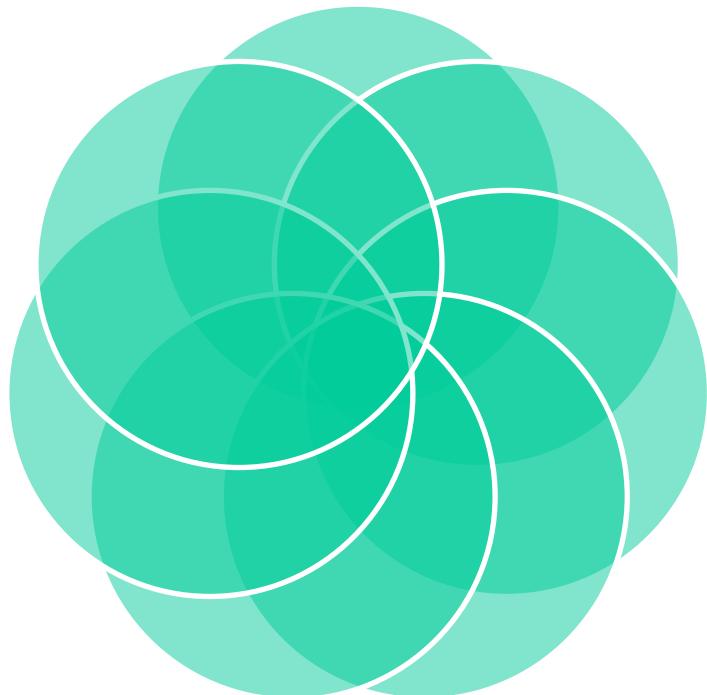
Turing Machines and Their Languages

CFL and Pushdown Automata

Finite Automata

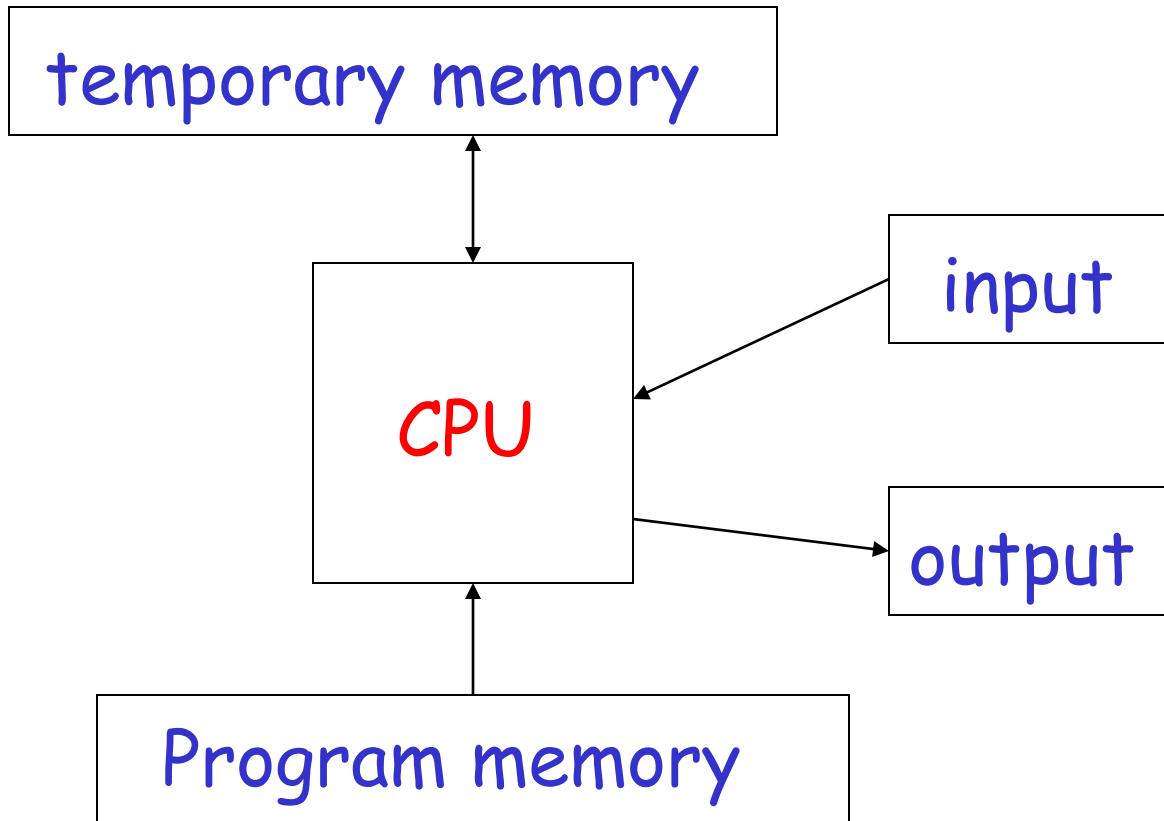
Regular Expressions

Regular and Non-regular Languages

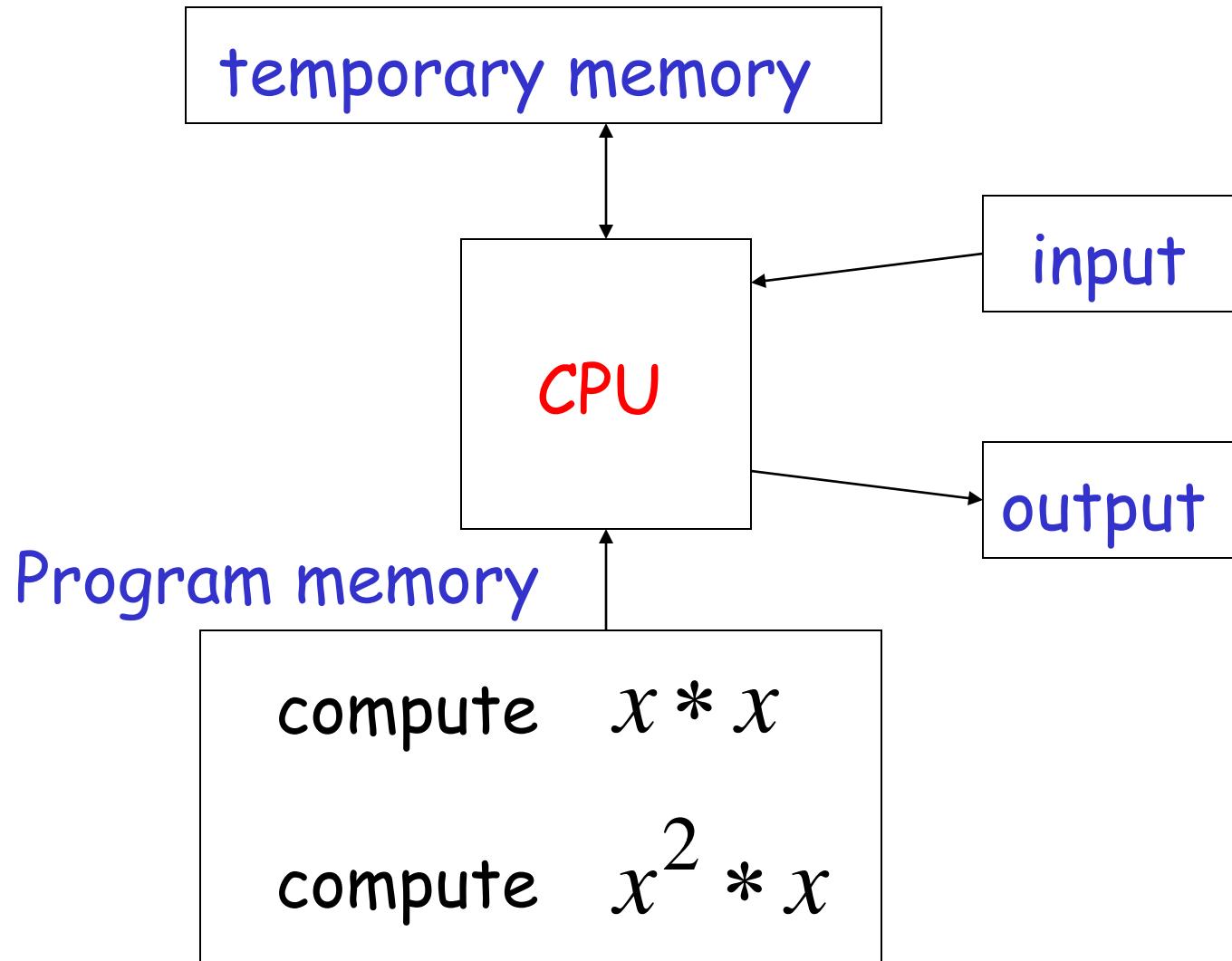


# Models of Computation

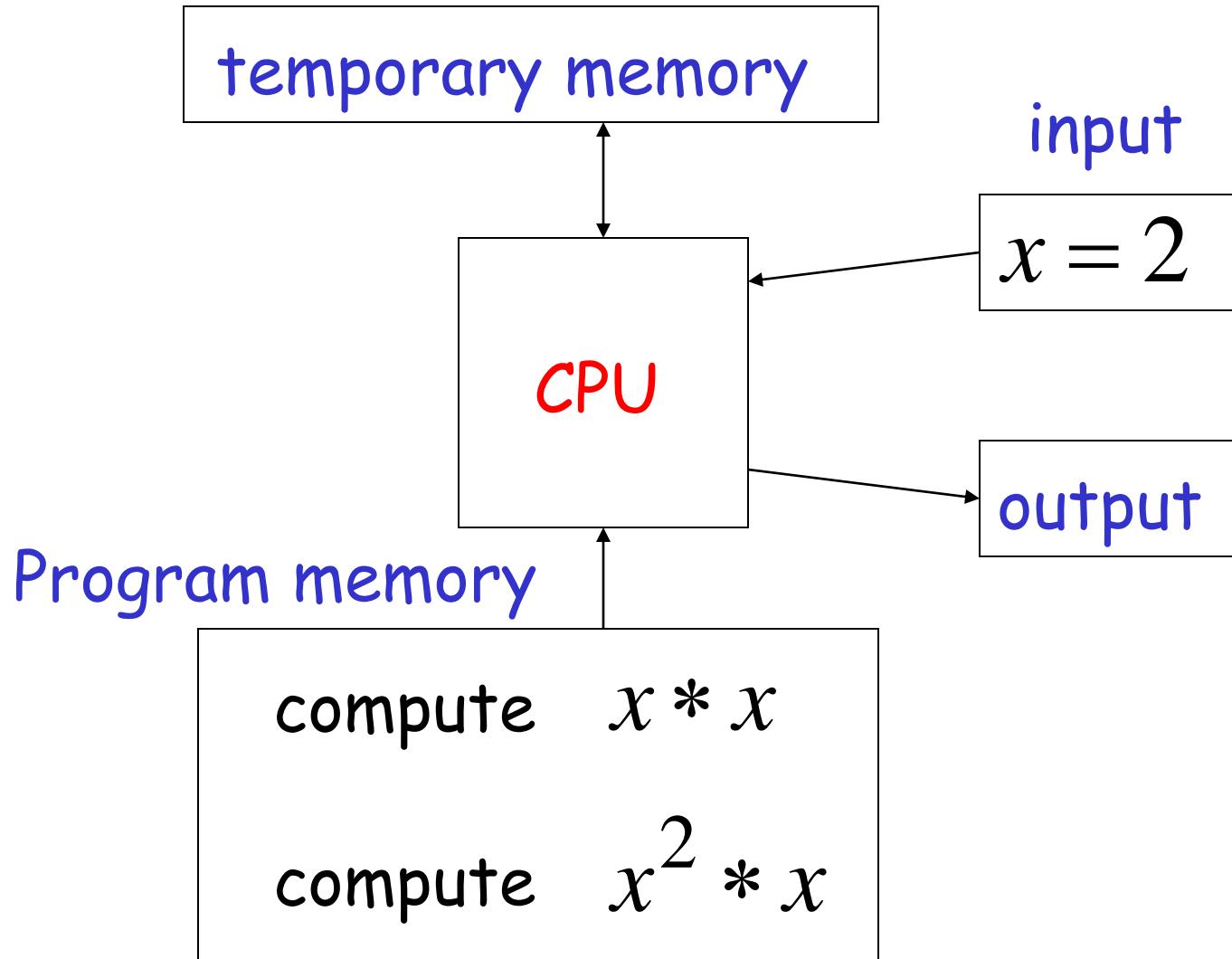
# A model of computation



Example:  $f(x) = x^3$



$$f(x) = x^3$$



temporary memory

$$f(x) = x^3$$

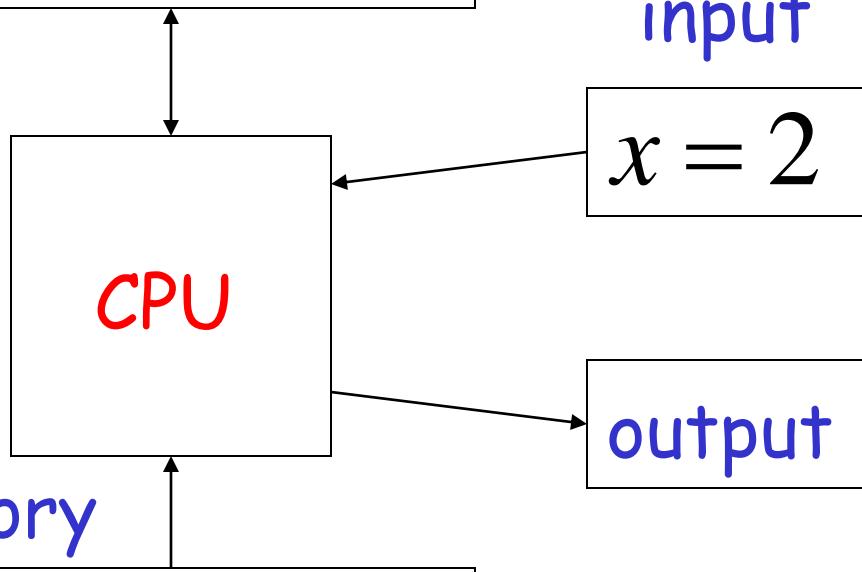
$$z = 2 * 2 = 4$$

$$f(x) = z * 2 = 8$$

Program memory

compute  $x * x$

compute  $x^2 * x$



temporary memory

$$f(x) = x^3$$

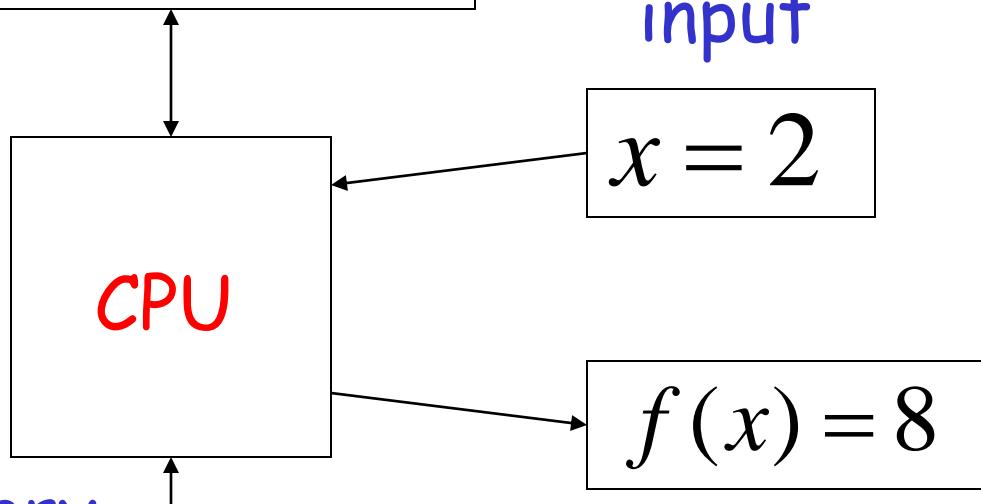
$$z = 2 * 2 = 4$$

$$f(x) = z * 2 = 8$$

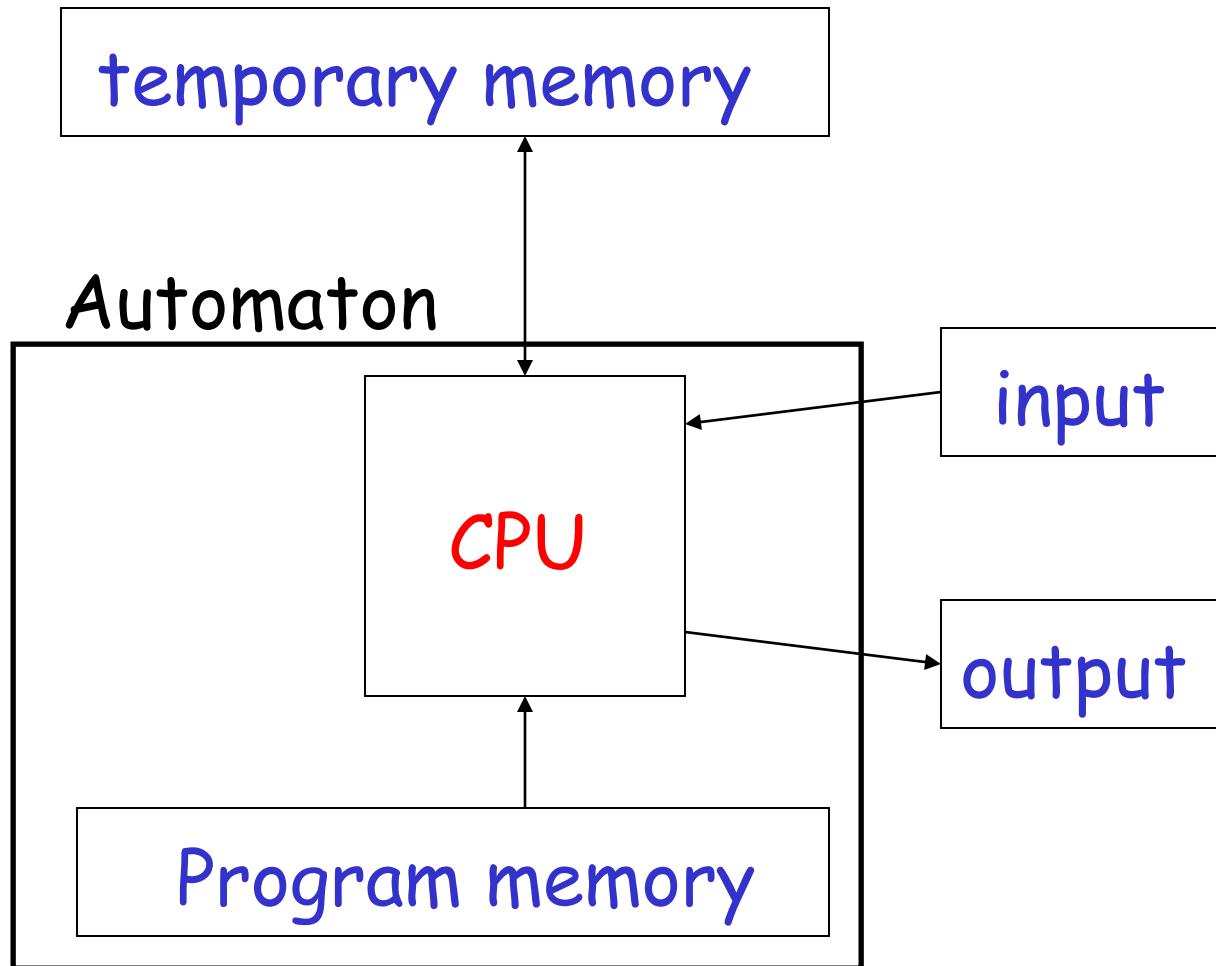
Program memory

compute  $x * x$

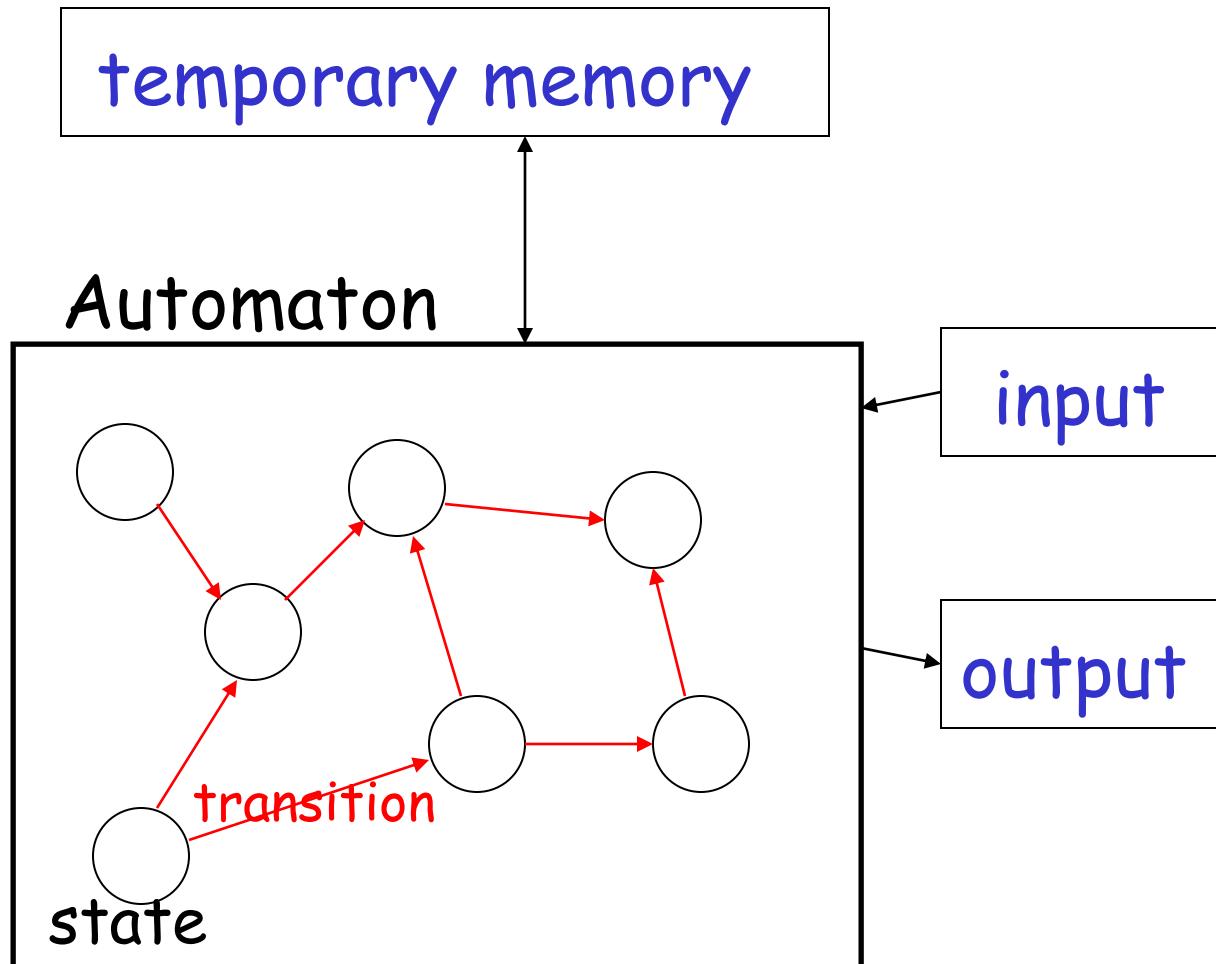
compute  $x^2 * x$



# Automaton



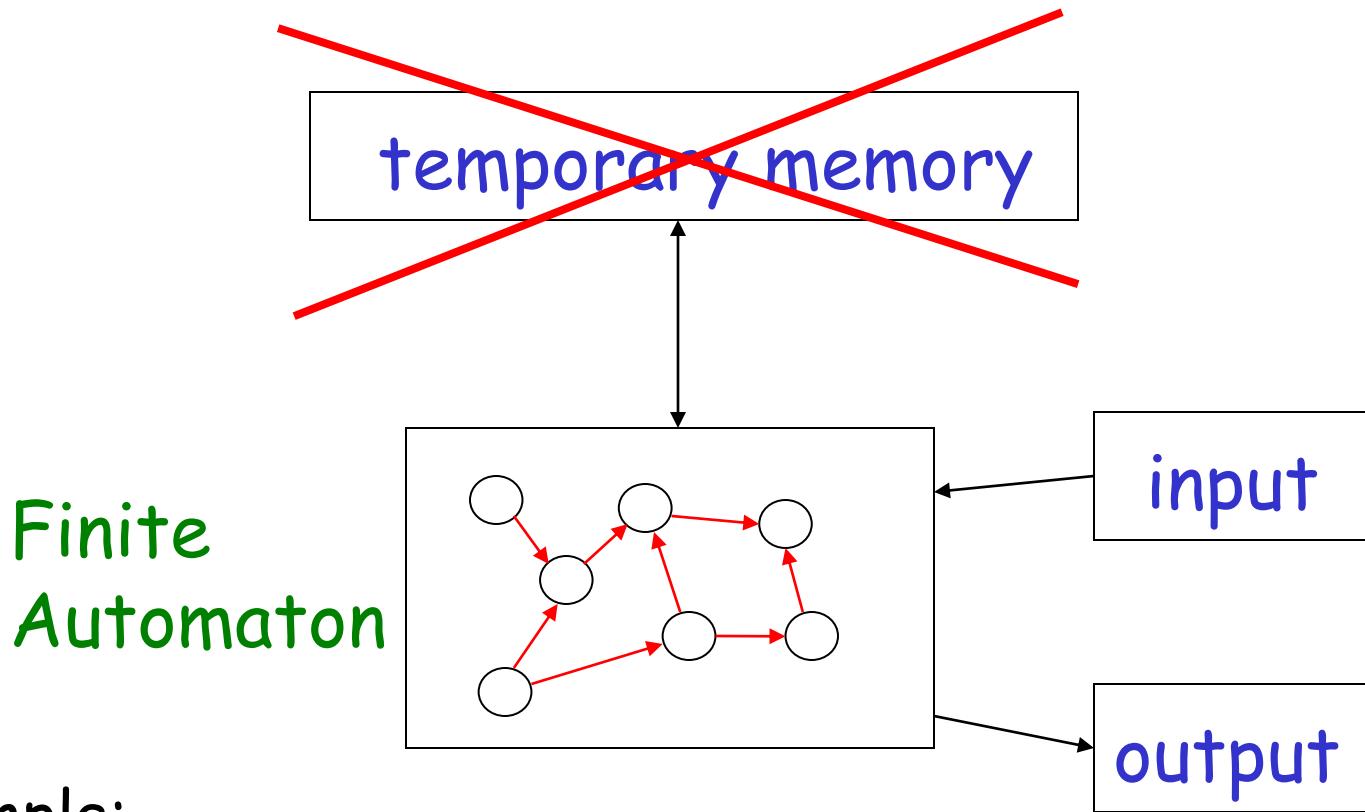
# Automaton



$\text{CPU} + \text{ProgramMem} = \text{States} + \text{Transitions}$

# Finite Automaton (Small Computing Power)

DFA, NFA, Mealy and Moore Machines



Example:

Vending Machines,

Elevators,

Lexical Analyzers

# Pushdown Automaton (Medium computing power)

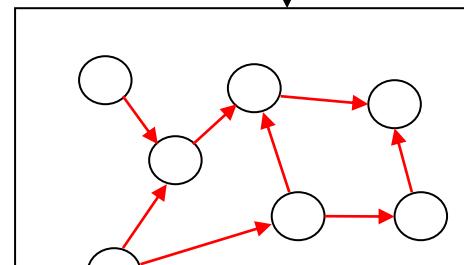
DPDA, NPDA

Temp.  
memory

Stack

Push, Pop

Pushdown  
Automaton



input

output

Example:

Parsers for Programming Languages

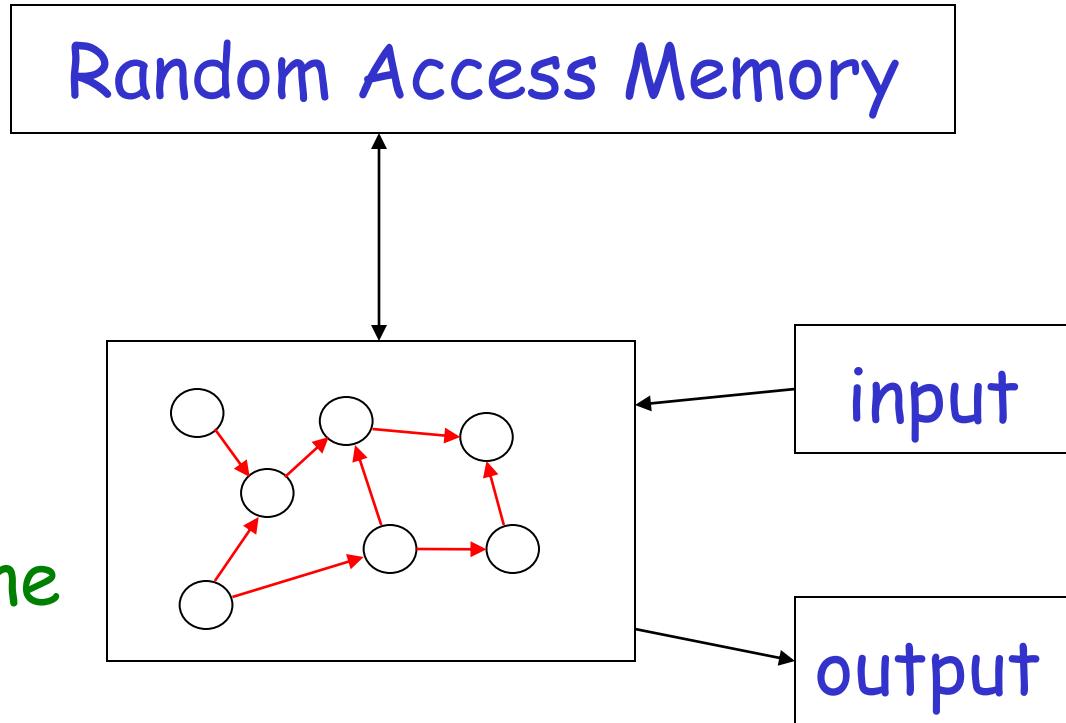
Speech Processing, NLP, etc.

# Turing Machine

(Highest known computing power)

Temp.  
memory

Turing  
Machine



Examples: Any solvable problem

# Different Kinds of Automata

Automata are distinguished by the temporary memory

- **Finite Automata:** no temporary memory
- **Pushdown Automata:** stack
- **Turing Machines:** random access memory
- **Memory affects computational power**
  - More flexible memory results in more powerful computational models.

# Powers of Automata

Simple  
problems

More complex  
problems

Hardest  
problems

Finite  
Automata

Pushdown  
Automata

Turing  
Machine

Less power



More power

Solve more

computational problems

Turing Machine is the most powerful known computational model

Question: can Turing Machines solve all computational problems?

Answer: NO

(there are unsolvable problems-)  
Halting Problem

# Applications of TOC

- Analysis of algorithms
- Complexity Theory
- Cryptography
- Speech Processing
- Compilers
- Circuit design
- Pattern Matching
- Software design

# History

- **1936:** Alan Turing invented the Turing machine, and proved that there exists an unsolvable problem.
- **1940:** Stored-program computers were built.
- **1943:** McCulloch and Pitts invented finite automata.
- **1956:** Kleene invented regular expressions and proved the equivalence of regular expression and finite automata.
- **1956:** Chomsky defined *Chomsky hierarchy*, which organized languages recognized by different automata into hierarchical classes.

# ..History

- **1959:** Rabin and Scott introduced *nondeterministic finite automata* and proved their equivalence to (deterministic) finite automata.
- **1950's:1960's** More works on languages, grammar, and compilers.
- **1965:** Hartmantis and Stearns defined *time complexity*, and Lewis, Hartmantis, and Stearns defined *space complexity*.
- **1971:** Cook showed the first *NP-complete problem*, the *satisfiability* problem.
- **1972:** Karp Showed many other NP-complete problems.

THANK YOU

# Theory of Computation

## **Mathematical Preliminaries**

- Sets, Relations, Functions
- Graphs
- Proof Techniques
- Languages
- Grammars

# Set

- **Set:** A set is a well defined collection of objects.
  - Order and repetition - **Irrelevant**
  - Empty/Finite/Infinite/Equal Sets.
- **Notations**
  - Given a set A and an object x, if x is an element of A, we write  $x \in A$
  - Otherwise, we write  $x \notin A$
- **Examples:**
  - $\{a, b, c, \dots, z\}$  stands for all the lower case letters of the English alphabet.
  - $\{2, 4, 6, \dots\}$  set of all even positive integers.
  - Explicit notation:  
 $S = \{i : i > 0, i \text{ is even}\}$
- **Whether the followings are set or not?**
  - The collection of the all-boys in our department. (Y/N)
  - The collection of the most talented boys in our department. (Y/N)
  - The collection of the best football players in the world. (Y/N)

# Set Representation & Cardinality

## ► By enumeration

- ▶ List all elements.
- ▶ **Only good for finite sets**
- ▶ Ex's.
  - ▶  $A = \{1, 2, 3, 4, 5\}$
  - ▶  $B = \{a, b, c, d, e\}$
  - ▶  $C = \{ \text{Joe, Jose, Sarah, Chen} \}$
  - ▶  $D = \{ \text{Black, Cat, White, Cat} \}$

## ► Set Cardinality

- Given a set A, the cardinality of A, denoted by  $|A|$ , is the number of elements in A.
- For finite set  $A = \{ 2, 5, 7 \}$ , the cardinality is  $|A| = 3$

## ► By Property

- ▶ Give property for set elements.
- ▶ **Good for infinite sets as well as for finite sets**
- ▶ Ex's.
  - ▶  $A = \{x: x \text{ is an integer}\}$
  - ▶  $B = \{y: y \text{ is a student at AMU}\}$
  - ▶  $C = \{z: z \text{ is a student in the CS department in AMU}\}$
  - ▶  $D = \{w: w \in A \text{ and } w \text{ is odd}\}$

# Set Operations/ Some Laws/Different Sets

$$A = \{ 1, 2, 3 \}$$

$$B = \{ 2, 3, 4, 5 \}$$

$$C = \{ 4, 5 \}$$

- Union
- Intersection
- Difference
- Complement
- DeMorgan's Law
- Subset/Proper Subset
- Disjoint Sets
- Power Set

**Typical Relations**

$$\emptyset = \{ \}$$

$$\overline{\emptyset} = \text{Universal Set}$$

$$s \cup \emptyset = s$$

$$s \cap \emptyset = \emptyset$$

$$s - \emptyset = s$$

$$\emptyset - s = \emptyset$$

# Powerset

- A powerset is a set of sets.
- $S = \{ a, b, c \}$
- Powerset of  $S$  = the set of all the subsets of  $S$
- $2^S = \{ \emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\} \}$
- Observation:  $|2^S| = 2^{|S|}$       ( $8 = 2^3$ )
  - Elements in power set  $2^S = 2^{\text{Elements in set } S}$

# Set Partition

- Given a nonempty set A, we say  $P = \{X_1, X_2, \dots, X_n\}$  is a partition of A if
  - Any two elements of P are disjoint.
  - None of the sets is empty.
  - $P \subseteq 2^A - \{\phi\}$
  - $\bigcup P = \bigcup_{X_i \in P} X_i = A$
- Ex's : Let  $S = \{a, b, c, d\}$
- P1 = {{a}, {b}, {c}, {d}}, P2 = {{a, b, c, d}}**
  - What partition of S has the fewest members?  $P = \{?\}$
  - What partition of S has the most members?  $P = \{?\}$
  - List all partitions of S with exactly two members?  $P = \{?\}$

$P3 = \{\{a, b\}, \{c, d\}\}, P4 = \{\{a, c\}, \{b, d\}\}$

**P5 = {{a}, {b}, {c}, {d}, {a, b}, {a, c}},**

# Ordered Pairs/Tuples

## Ordered Pairs

- Given any two objects  $a, b$ , let  $(a, b)$  denote the **ordered pair** of  $a$  and  $b$ .
- **The ordered pair of  $a$  and  $b$  tells that  $a$  and  $b$  are related under some relationship**
- The **differences** between  $(a, b)$  and  $\{a, b\}$ :
  - $(a, b) \neq (b, a)$ , while  $\{a, b\} = \{b, a\}$
- The **equivalence**
  - $(a, b) = (c, d)$  if and only if  $a = c, b = d$ .

## Ordered Tuples

- Given any  $n$  objects,  $a_1, a_2, \dots, a_n$ ,
- An  $n$ -ary ordered tuple.  $(a_1, a_2, \dots, a_n)$
- The objects may not be distinct.

# Cartesian Products/Relation

- Given two sets A and B, the **Cartesian Product** of A and B is

$$A \times B = \{(a, b) : a \in A \wedge b \in B\}$$

- Given  $n$  sets  $A_1, A_2, \dots, A_n$ , the  $n$ -fold Cartesian Product of these sets is

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \dots, a_n) : a_i \in A_i, 1 \leq i \leq n\}$$

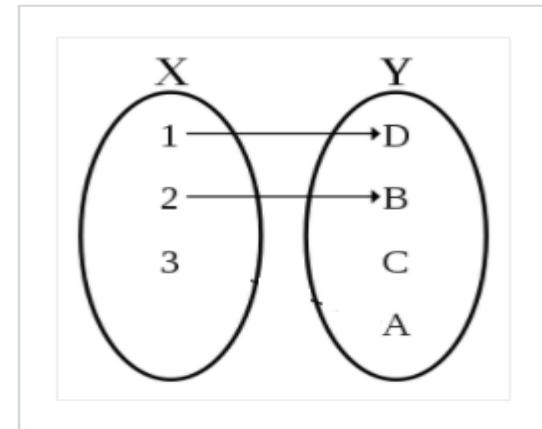
- A binary relation  $R : A \rightarrow B$  is defined as  $R \subseteq A \times B$

- A  $n$ -ary relation  $R : A_1 \times A_2 \times \cdots \times A_n \rightarrow B$  is defined as

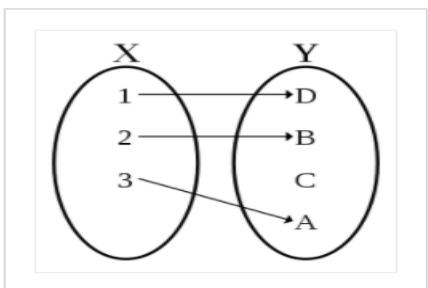
$$R \subseteq A_1 \times A_2 \times \cdots \times A_n \times B$$

# Functions

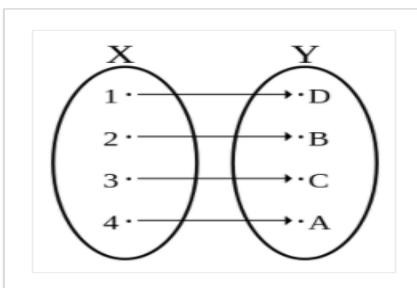
- A **function**  $f:A \rightarrow B$  is a binary relation on A and B such that
  - $\forall x \in A$ , **there is exactly one**  $y \in B$  **such that**  $(x, y) \in f$ .
  - A is called the domain.
- For any  $(x, y) \in f$ , we denote y as  $f(x)$ . (**Image** of x).
- The range of **f** is defined as
  - $f[A] = \{ y : y = f(x) \text{ for some } x \in A \subseteq B\}$
- Types of Functions
  - One-to-One (Injective)
  - Onto (Surjective)
  - Bijective



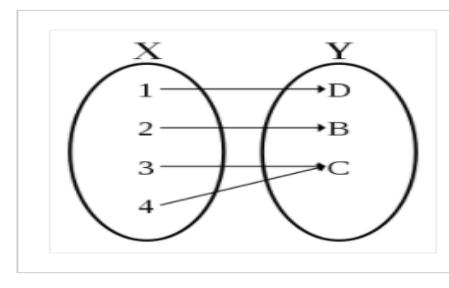
Function or not ?



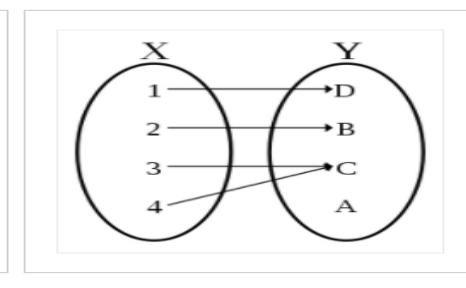
An injective non-surjective function  
(**injection**, not a **bijection**)



An injective surjective function  
(**bijection**)



A non-injective surjective function  
(**surjection**, not a **bijection**)



A non-injective non-surjective  
function (also not a **bijection**)

# Relation's Types

## Reflexive

- A relation  $R: A \rightarrow A$  is **reflexive**,
  - if for all  $a \in A, (a, a) \in R$ .
- Example- The product of  $x$  and  $y$  is even on even integers
- **Anti-Reflexive**—The product of  $x$  and  $y$  is even on odd integers.
- **Neither reflexive nor Anti-Reflexive** -The product of  $x$  and  $y$  is even on natural integers.

## Transitive

- A relation  $R: A \rightarrow A$  is **transitive** if  $(a, b)$  and  $(b, c)$  are in  $R$ , then  $(a, c)$  is also in  $R$ .
- Examples?
  - The ancestor relation over a group of people.
  - The less-than relation over integers.

# Relation's Types

- **Symmetric**

- A relation  $R: A \rightarrow A$  is **symmetric**
  - if for any  $(a, b) \in R, (b, a) \in R.$
- Example -Relationship among sisters
- Non-exemplar-Relationship among siblings consisting of a boy and girl each

- **Asymmetric**

- A relation  $R: A \rightarrow A$  is asymmetric
- if for any,  $(a, b) \in R, (b, a) \notin R.$
- Example- The parent relation over a group of people.

- **Antisymmetric**

- A relation  $R: A \rightarrow A$  is antisymmetric
- if for any  $(a, b) \in R, (b, a) \in R.$  then  $a=b$
- Example- The divisibility relation over the natural numbers

- **Equivalence Relation**

- A relation  $R: A \rightarrow A$  is equivalence if  $R$  is reflexive, symmetric and transitive.
- Example- For any natural number  $m,$  the modular relation  $\equiv_m$  is an equivalence relation on integers.

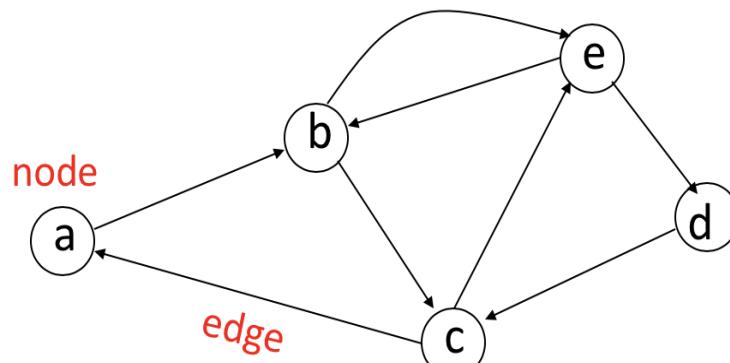
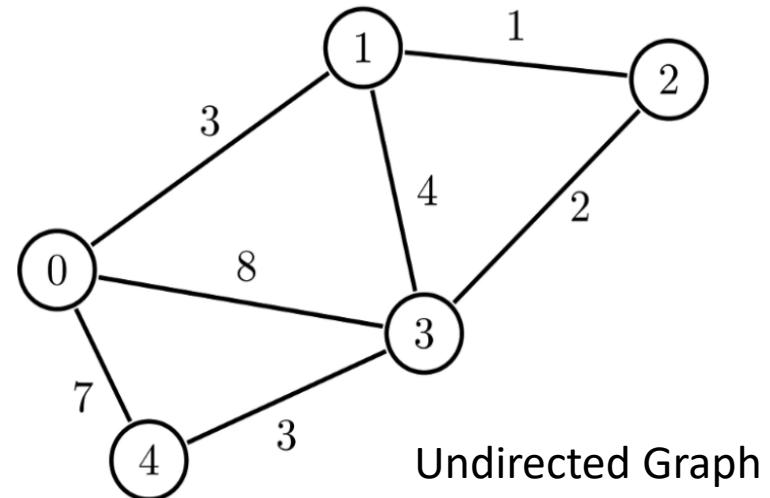
# Countable and Uncountable Sets

- **Countable**
  - A set S is countable if one of the followings is true
    - S is finite
    - S is infinite but there is a bi-jection  $f: S \rightarrow N$ , where N is the set of natural numbers.
  - In other words, S is countable, if elements in S are enumerable.
  - Example- Set of positive even Integers
    - $S = \{1, 2, 3, 4, 5, 6, 7, \dots\}$
- **Uncountable**
  - **Sets that are not countable?**
  - Example- The open interval of real numbers  $(0, 1)$
  - 0.000001 , 0.00001, 0.000011

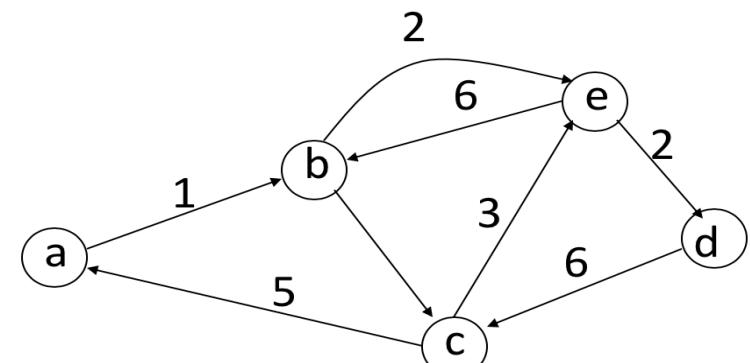
# GRAPHS

A graph is a construct consisting of two finite sets  $V$  (set of Vertices  $V$ ) and  $E$  (set of edges).

- Undirected
- Directed



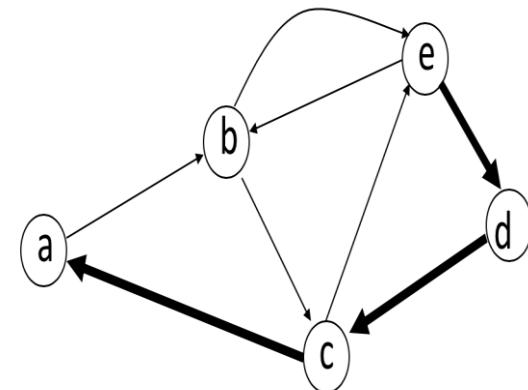
Directed Graph (Digraph)



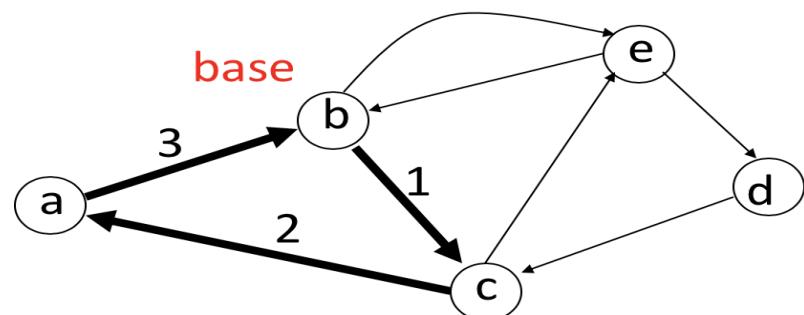
Labeled Graph

# Walk/Path/Cycle

- Walk : A sequence of adjacent edges
- A walk from e to a is (e, d), (d, c), (c, a)
- Length of the walk = Total number of traversed edges



- Path : A walk where no edge is repeated
- Path: (e, d), (d, c), (c, a)(a, b)(b, e)
- Cycle: A walk from a node (base) to itself with no repeated edges
- Simple cycle: A cycle in which only the base node is repeated



# PROOF TECHNIQUES

- **Direct Proof (Proof by Construction)**

- **Proof by Contrapositive**

- **Proof by Induction**

- **Proof by Contradiction**

## Statement

- The sum of any two consecutive numbers is odd.

- If  $a$  and  $b$  are consecutive integers, then the sum  $a + b$  is odd

- $P \rightarrow Q$

## Facts (Definitions)

- An integer number  $n$  is even if and only if there exists an integer  $k$  such that  $n = 2k$ .

- An integer number  $n$  is odd if and only if there exists an integer  $k$  such that  $n = 2k + 1$ .

- **Two integers  $a$  and  $b$  are consecutive if and only if  $b = a + 1$ .**

# Direct Proof (Proof by Construction)

## Method

- Assume that P is true.
- Use P to show that Q must be true.

## Statement ( $P \rightarrow Q$ )

- If a and b are consecutive integers, then the sum a + b is odd.

## Proof

- Assume that a and b are consecutive integers
- If a and b are consecutive integers,  $b = a + 1$
- So,  $a + b = a + a + 1 = 2a + 1$
- a is an integer, therefore, a + b is odd

### Definition:

An integer number n is odd if and only if there exists an integer k such that  $n = 2k + 1$ .

# Proof by Contrapositive

## Statement

- If  $a$  and  $b$  are consecutive integers, then the sum  $a + b$  is odd.

## Basis

- From first-order logic,  $P \Rightarrow Q$  is equivalent to  $\neg Q \Rightarrow \neg P$ .
- The second proposition is called the **contrapositive** of the first proposition.

## Method

- Assume  $\neg Q$  is true.
- Show that  $\neg P$  must be true.
- Observe that  $P \Rightarrow Q$  by contraposition.

# Proof by Contrapositive

## Statement

- If  $a$  and  $b$  are consecutive integers, then the sum  $a + b$  is odd ( $P \Rightarrow Q$ )

## Contrapositive Statement

- If the sum  $a + b$  is not odd, then  $a$  and  $b$  are not consecutive integers ( $\neg Q \Rightarrow \neg P$ ).

## Proof

- Assume that the sum of the integers  $a$  and  $b$  is not odd.
- There exists no integer  $k$  such that  $a + b = 2k + 1$ .
- Thus,  $a + b \neq k + (k + 1)$  for all integers  $k$ .
- Because  $k + 1$  is the successor of  $k$ , this implies that  $a$  and  $b$  cannot be consecutive integers.

# Proof by Induction-Example-1

## Method

- Inductive basis- Find  $P_1, P_2, \dots, P_b$  which are true.
- Inductive assumption - Let's assume  $P_1, P_2, \dots, P_k$  are true, for any  $k \geq b$ .
- Inductive step -Show that  $P_{k+1}$  is true.

Statement ( $P_i$ )

- If  $a$  and  $b$  are consecutive integers, then the sum  $a + b$  is odd. ( $a=i, b=i+1$  )

• **Proof.**

- (Step 1) Consider the proposition  $P_1$ .

$1 + 2 = 3$  is odd because there exist an integer  $x$  (1) such that  $2x + 1 = 3$ .

Thus,  $P_i$  is true when  $i = 1$ .

Consider the proposition  $P_2$ .

Consider the proposition  $P_3$ .

.....

# Proof by Induction-Example-1

Inductive basic:  $P_1, P_2, P_3, \dots, P_b$  are true

## (Step 2)

Assume that  $P_k$  is true for some  $k, k \geq b$ .

Thus,  $k + (k + 1)$  is odd.

## (Step 3)

Show that  $P_{k+1}$  is true.

$$P_{k+1} = k+1 + (k + 1+1) = 2k + 1 + 2 = P_k + 2.$$

**Observation: Adding two to any integer does not change integer's evenness or oddness.**

Since  $P_k$  is true and therefore,  $P_{k+1}$  is also true.

# Proof by induction-Example-2

**Theorem:** A binary tree of height  $h$  has at most  $2^h$  leaves.

Let  $L(i)$  be the maximum number of leaves of any binary tree at height  $i$ .

We want to show:  $L(i) \leq 2^i$

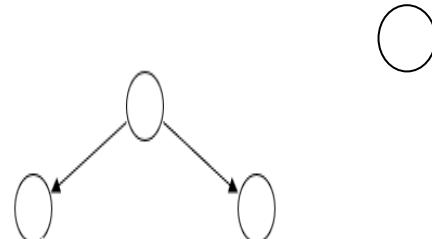
- Inductive basis

$$L(0) = 1 \quad (\text{the root node})$$

$$L(1) = 2$$

$$L(2) = 4$$

.....



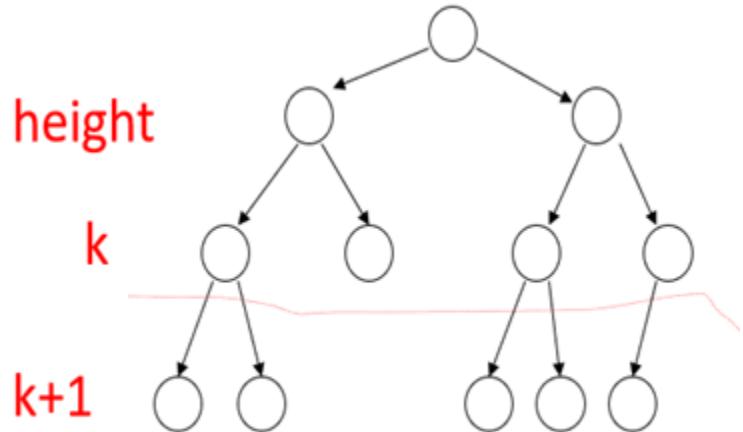
- Inductive hypothesis

Let's assume  $L(i) \leq 2^i$  for all  $i = 0, 1, \dots, k$

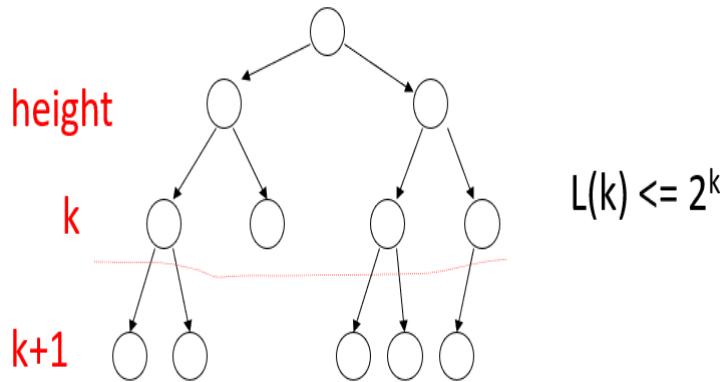
- Induction step

we need to show that  $L(k + 1) \leq 2^{k+1}$

# Proof by induction-Example-2



From Inductive hypothesis:  $L(k) \leq 2^k$



$$L(k+1) \leq 2 * L(k) \leq 2 * 2^k = 2^{k+1}$$

(we add at most two nodes for every leaf of level k)

# Proof by Contradiction

## Statement

$$P \Rightarrow Q$$

## Basis

Any proposition must be either true or false, but not both true and false at the same time.

## Method

- Assume that P is true.
- Assume that  $\neg Q$  is true.
- Use P and  $\neg Q$  to demonstrate a contradiction.

# Proof by Contradiction-Example-1

## Statement

- If a and b are consecutive integers, then the sum  $a + b$  is odd.

## Proof

- Assumption 1- a and b are consecutive integers.
- Assumption 2 - The sum  $a + b$  is not odd.
- Since, integers a and b are consecutive,
  - $a + b = a + a + 1 = 2a + 1$ . (**TRUE**)
- Since, sum  $a + b$  is not odd,
  - **There exists no integer k such that  $a + b = 2k + 1$ .** (**TRUE**)

Contradiction!!!

- **If we hold that a and b are consecutive then the sum  $a + b$  must be odd.**

# Proof by Contradiction-Example-2

Statement:  $\sqrt{2}$  is not rational

Proof: Assume  $\sqrt{2}$  it is rational

$$\sqrt{2} = n/m \quad n \text{ and } m \text{ have no common factors}$$

We will show that this is impossible

$$\sqrt{2} = n/m \longrightarrow 2m^2 = n^2$$

Therefore,  $n^2$  is even  $\longrightarrow$  n is even  
 $n = 2k$

$$2m^2 = 4k^2 \longrightarrow m^2 = 2k^2 \longrightarrow m \text{ is even}$$
$$m = 2p$$

Thus, m and n have common factor 2

**Contradiction!**

# Languages : Basics

- **Alphabet**

- An alphabet ( $\Sigma$ ) is a finite, non-empty set of symbols.
- Example:
  - {0,1} --- Binary alphabet.
  - {1, ..., 9,...} ----- Alphabet for natural numbers
  - {A, B, ..., Z, a, b, ..., z} ----- English alphabet.
  - {a, ..., z, A, ..., Z, 0,1, ..., 9, \_, } – Alphabet for C variables
  - The set of all ASCII characters
  - Etc.

- **String {#, \$, \*}**

- A string over an alphabet  $\Sigma$  is a finite sequence of symbols from the alphabet  $\Sigma$ .
- Example-
  - 0, 1, 11, 00, and 01101 ----- Strings over {0, 1}.
  - Cat, CAT, and, AND, compute ----- Strings over the English alphabet.
  - **a, b, ab, aa, ba, bb, aaa, aab, abaa, ----- Strings over {a, b}.**
  - \_abc, num, const1, const2 ----- Strings over alphabet for C variables
- **Note- Infinite strings exists over any alphabet  $\Sigma$**

# Language : Basics

- **Empty String**

- The string with **zero** occurrences of symbols from the alphabet  $\Sigma$
  - It is denoted by **e, ε or λ**.
  - A string without symbols over any alphabet.

- **String Length**

- **The length of a string w is the number of occurrences of symbols in w**
  - **Denoted by |w| or length(w)**
  - **Example- Let  $\Sigma = \{a, b, \dots, z, A, B, \dots, Z\}$ ,**
    - $\text{length(automata)} = 8$  ,  $\text{length(computation)} = 11$
    - $\text{length}(\lambda) = 0$ ,
  - **Example- Let  $\Sigma = \{0, 1\}$ ,**
    - $|1010| = 4$ , **|000000| =7**,  $|1010101010| =10$ ,  $|\lambda|=0$
  - **w(i), denotes the symbol in the ith position of a string w, for  $1 \leq i \leq \text{length}(w)$ .**

## Caution:

**Sets**

$\emptyset = \{\} \neq \{\lambda\}$

**Set size**

$|\{\ }| = |\emptyset| = 0$

**Set size**

$|\{\lambda\}| = 1$

**String length**

$|\lambda| = 0$

# String Operations

- **Substring**

- z is a substring of w if it is a sequence of consecutive symbols of w.
- Substrings of “Theory” are ‘T’, ‘Th’, ‘eo’ or ‘Theory’.

- **Prefix (Leading substring)**

- A prefix of a string w is any leading contiguous part of w.
- z is a prefix of string w if there is a string x such that  $w = zx$
- Prefix of “Theory” are ‘T’, ‘Th’, ‘Theo’
- Example- w = abbab
  - A =  $\{\lambda, a, ab, abb, abba, abbab\}$  is the set of all prefixes.

- **Suffix (Trailing substring)**

- A suffix of the string w is any trailing contiguous part of w.
- z is a suffix of w if there is a string x such that  $w = xz$ .
- Suffix of ‘love’ in “Ilovetheory” is ‘theory’
- Example- w = abbab
  - B =  $\{\lambda, b, ab, bab, bbab, abbab\}$  is the set of all suffixes.

# String Operations

## • Concatenation

- If  $x$  and  $y$  are two strings, then  $xy$  is obtained by appending the symbols of  $y$  at the right end of  $x$
- *Examples-*
  - $x = a_1a_2 \dots a_i, \quad y = b_1b_2 \dots b_j, \quad xy = a_1a_2 \dots a_i b_1b_2 \dots b_j$
  - $x = 01101, \quad y = 110, \quad xy = 01101110$
  - $x\lambda = \lambda x = x$
- Note: If  $u$  and  $v$  are strings, then the length of their concatenation is the sum of the individual lengths
  - $|uv| = |u| + |v|$

## • Power of a string $w$

- Given any natural number  $i$ ,
- $w^n$  stands for the string obtained by repeating (concatenating)  $w$   $n$  times.

$$w^i = \begin{cases} \lambda & \text{if } i = 0, \\ ww^{i-1} & \text{if } i > 0 \end{cases}$$

- Example-  $w = abab$ ; compute  $w^0, w^1, w^2, w^3, \dots$

# String Operations

## • Reverse

- The reverse of a string is obtained by writing the symbols in reverse order.
- The reversal of a string  $w$

$$w^R = \begin{cases} \lambda & \text{if } |w|=0, \\ u^R a & \text{if } w=au, \end{cases}$$

Where  $a \in \Sigma$  and  $u \in \Sigma^*$

- Property  $(xy)^R = y^R x^R$
- Examples-
  - $x = a_1 a_2 \dots a_i \dots a_n$        $x^R = a_i a_{i-1} \dots a_2 a_1$
  - $y = b_1 b_2 \dots b_j \dots b_n$        $y^R = b_j b_{j-1} \dots b_2 b_1$
  - $xy = a_1 a_2 \dots a_i b_1 b_2 \dots b_j \dots b_n$        $(xy)^R = b_j b_{j-1} \dots b_2 b_1 a_i a_{i-1} \dots a_2 a_1$
  - $x = 01101, x^R = 10110$

# Closure of Alphabet

## Kleen's Closure - $\Sigma^*$

- The set of strings obtained by concatenating **zero or more symbols** from alphabet  $\Sigma$  is denoted by  $\Sigma^*$ .
- The  $\Sigma^*$  consists of strings of every possible length.
- $\Sigma^* = \cup_{i=0}^{\infty} \Sigma^i = \cup_{i=0..\infty} \Sigma^i$
- Example-
  - Let  $\Sigma = \{0, 1\}$ .
  - $\Sigma^* = \{\lambda, \textcolor{violet}{0}, \textcolor{blue}{1}, \textcolor{red}{00}, \textcolor{blue}{01}, \textcolor{red}{10}, \textcolor{blue}{11}, \textcolor{brown}{000}, \textcolor{red}{001}, \textcolor{blue}{010}, \textcolor{red}{011}, \textcolor{brown}{100}, \textcolor{red}{101}, \textcolor{blue}{110}, \textcolor{red}{111}, \dots \}$ .

## Positive Closure - $\Sigma^+$

- The set of strings created by concatenating at least one symbol (1 or 2 or ...) from alphabet  $\Sigma$  is denoted by  $\Sigma^+$ .
- The  $\Sigma^+$  does not contain empty string  $\lambda$ .
- The  $\Sigma^+$  is infinite set since there is no limit on the length of the strings.
- $\Sigma^+ = \cup_{i=1}^{\infty} \Sigma^i = \cup_{i=0..\infty} \Sigma^i - \Sigma^0 = \cup_{i=0..\infty} \Sigma^i - \{\lambda\}$
- Example- Let  $\Sigma = \{0, 1\}$ .
  - $\Sigma^+ = \{\textcolor{violet}{0}, \textcolor{blue}{1}, \textcolor{red}{00}, \textcolor{blue}{01}, \textcolor{red}{10}, \textcolor{blue}{11}, \textcolor{brown}{000}, \textcolor{red}{001}, \textcolor{blue}{010}, \textcolor{red}{011}, \textcolor{brown}{100}, \textcolor{red}{101}, \textcolor{blue}{110}, \textcolor{red}{111}, \dots \}$ .

# Closure of Alphabet

- The following relations exist-

$$\Sigma^* = \Sigma^+ \cup \{\lambda\}$$

$$\Sigma^+ = \Sigma^* - \{\lambda\}$$

- Find  $\Sigma^*$  and  $\Sigma^+$ 
  - $\Sigma = \{a, b\}.$
  - $\Sigma = \{a, b, \dots, x, y, z, A, B, C, \dots, X, Y, Z\}.$
  - $\Sigma = \{0, 1, 2, \dots, 9\}$
  - $\Sigma = \{0, 1\}$

# Language

- Definition
  - A “Language” is a set of strings over an alphabet of symbols.
  - A language is a subset of  $\Sigma^*$
  - **A language is a system of communication** (which consists of a set of symbols that are used by the people of a particular country or region for talking or writing or establishing communication).
- Familiar languages
  - Natural languages like English, Hindi, etc.
    - English may be considered a **set of English words** or a **set of English sentences**, for which many grammar rules have been developed.
  - Programming languages like C, C++, Java, Python, etc.
    - Programming language may be considered a **set of valid identifiers** or a **set of valid statements** for which many rules (compilers) have been developed.
- **Observation:** Language = Alphabet + Some Rules (Grammar)

# Languages

- English alphabet  $\Sigma = \{a, b, \dots, x, y, z, A, B, C, \dots, X, Y, Z\}$ .
  - $\Sigma^* = \{\lambda, a, b, \dots, z, an, ball, cat, set, tes, \dots, A, \dots, Z, APPLE, \dots\} =$  The set of all English strings over the English alphabet.
  - $\Pi = \{x : x \text{ is valid English word}\} \subseteq \Sigma^*$
- The set of valid Arabic words over Arabic alphabet.
- $L = \{n \mid n \text{ is a prime number} > 20\} = \{23, 29, 31, 37, \dots\} \subseteq \{x : x \text{ is a prime number}\}$
- The set of strings with even numbers of 1's over  $\{0, 1\}$ 
  - $\Sigma^* = \{0, 1\}^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \dots\}$ .
  - $L = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\} = \{\lambda, 0, 00, 11, 000, 110, 101, 011, 0000, 1100, 1010, 1001, 0110, 0101, 0011, \dots\}$
  - $L \subseteq \Sigma^*$
- The set of strings consisting of  $n$  0's followed by  $n$  1's
  - $L_1 = \{\lambda, 01, 0011, 000111, \dots\}$  over  $\Sigma = \{0, 1\}$
- The set of strings with equal number of 0's and 1's
  - $L_2 = \{\lambda, 01, 10, 0011, 0101, 1010, 1001, 1100, \dots\}$  over  $\Sigma = \{0, 1\}$
- **Find the strings of language  $L_3 = \{a^n b^n : n \geq 0\}$  over  $\Sigma = \{a, b\}$ .**

# Language Examples

1. The empty language  $\emptyset$ .
2.  $\{\Lambda, a, aab\}$ , another finite language.
3. The language  $\text{Pal}$  of palindromes over  $\{a, b\}$  (strings such as  $aba$  or  $baab$  that are unchanged when the order of the symbols is reversed).
4.  $\{x \in \{a, b\}^* \mid n_a(x) > n_b(x)\}$ .
5.  $\{x \in \{a, b\}^* \mid |x| \geq 2 \text{ and } x \text{ begins and ends with } b\}$ .
6. The language of legal Java identifiers.
7. The language  $\text{Expr}$  of legal algebraic expressions involving the identifier  $a$ , the binary operations  $+$  and  $*$ , and parentheses. Some of the strings in the language are  $a$ ,  $a + a * a$ , and  $(a + a * (a + a))$ .
8. The language  $\text{Balanced}$  of balanced strings of parentheses (strings containing the occurrences of parentheses in some legal algebraic expression). Some elements are  $\Lambda$ ,  $0(0)$ , and  $((((0))))$ .
9. The language of numeric “literals” in Java, such as  $-41$ ,  $0.03$ , and  $5.0E-3$ .
10. The language of legal Java programs. Here the alphabet would include upper- and lowercase alphabetic symbols, numerical digits, blank spaces, and punctuation and other special symbols.

# Operations on Languages

- Let  $\Sigma$  be any finite alphabet  $\{a_1, a_2, \dots, a_n\}$
- Any subset  $L$  of  $\Sigma^*$  is called a **language over  $\Sigma$** .
- Language is defined as a **set**.
  - The set operations are applicable to languages.
    - Union, intersection, difference, complement.
  - Additional operations!
    - Reversal, Concatenation, Kleen's Closure (Kleen \*), Positive closure,  
...
- Example:
  - $L_1 = \{a, b, ab, ba\}$  over  $\Sigma = \{a, b\}$
  - $L_2 = \{a, b, aa, bb\}$  over  $\Sigma = \{a, b\}$

# Operations on Languages

- **Union**

- Let  $L_1$  and  $L_2$  be languages over an alphabet  $\Sigma$ .
- The **union** of  $L_1$  and  $L_2$ , denoted by  $L_1 \cup L_2 = \{x \mid x \text{ is in } L_1 \text{ or } L_2\}$
- Example:
  - $L_1 = \{x \in \{0,1\}^* \mid x \text{ begins with } 0\}$ ,  $L_2 = \{x \in \{0,1\}^* \mid x \text{ ends with } 0\}$
  - $L_1 \cup L_2 = \{x \in \{0,1\}^* \mid x \text{ begins or ends with } 0\}$   
 $= \{0, 01, 10, 010, \dots, 011111, 011110, 111110, \dots\}$

- **Intersection**

- Let  $L_1$  and  $L_2$  be languages over an alphabet  $\Sigma$ .
- The intersection of  $L_1$  and  $L_2$ , denoted by  $L_1 \cap L_2 = \{x \mid x \text{ is in } L_1 \text{ and } L_2\}$ .
- **Example-**
  - $L_1 = \{x \in \{0,1\}^* \mid x \text{ begins with } 0\}$ ,  $L_2 = \{x \in \{0,1\}^* \mid x \text{ ends with } 0\}$
  - $L_1 \cap L_2 = \{x \in \{0,1\}^* \mid x \text{ begins and ends with } 0\}$   
 $= \{0, 00, 010, 000, 0000, 0010, 0100, 0110, \dots\}$

# Operations on Languages

## • Difference

- Let  $L_1$  and  $L_2$  be languages over an alphabet  $\Sigma$ .
- The difference of  $L_1$  from  $L_2$ , denoted by  $L_1 - L_2 = \{ x \mid x \text{ is in } L_1 \text{ and } x \text{ is not in } L_2 \}$ .
- **Example-**

- $L_1 = \{ x \in \{0,1\}^* \mid x \text{ begins with } 0 \}$ ,  $L_2 = \{ x \in \{0,1\}^* \mid x \text{ ends with } 0 \}$
- $L_1 - L_2 = \{ x \in \{0,1\}^* \mid x \text{ begins with } 0 \text{ and does not end with } 0 \}$   
 $= \{01, 011, 001, 0001, 0011, 0101, 0111, \dots\}$

## • Complement

- Let  $L$  be a language over an alphabet  $\Sigma$ .
- The **complementation** of  $L$ , denoted by  $\bar{L} = \Sigma^* - L$
- **Example:** Let  $\Sigma = \{0, 1\}$  be the alphabet.
  - $L_e = \{\omega \in \Sigma^* \mid \text{the number of } 1's \text{ in } \omega \text{ is even}\}$ .
  - $\bar{L}_e = \{\omega \in \Sigma^* \mid \text{the number of } 1's \text{ in } \omega \text{ is not even}\}$   
 $= \{\omega \in \Sigma^* \mid \text{the number of } 1's \text{ in } \omega \text{ is odd}\}$ .

# Operations on Languages : Reversal

- Let  $L$  be a language over an alphabet  $\Sigma$ .
- The reversal of  $L$ , denoted by  $L^r = \{w^r \mid w \text{ is in } L\}$
- Example-1
  - $L = \{x \in \{0,1\}^* \mid x \text{ begins with } 0\}$
  - $L^r = \{x \in \{0,1\}^* \mid x \text{ ends with } 0\}$
- Example-2
  - $L = \{x \in \{0,1\}^* \mid x \text{ has } 00 \text{ as a substring}\}$
  - $L^r = \{x \in \{0,1\}^* \mid x \text{ has } 00 \text{ as a substring}\}$

# Operations on Languages :Concatenation

- Let  $L_1$  and  $L_2$  be languages over an alphabet  $\Sigma$ .
- The concatenation of  $L_1$  and  $L_2$ , denoted by  $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \text{ is in } L_1 \text{ and } w_2 \text{ is in } L_2\}$ .

## Example-1

- $L_1 = \{a, b\}$ ,  $L_2 = \{1, 11, 01\}$
- $L_1 L_2 = \{a1, a11, a01, b1, b11, b01\}$
- $L_2 L_1 = \{1a, 1b, 11a, 11b, 01a, 01b\}$

## Example-2

- $L_1 = \{x \in \{0,1\}^* \mid x \text{ begins with } 0\}$
- $L_2 = \{x \in \{0,1\}^* \mid x \text{ ends with } 0\}$
- $L_1 \cdot L_2 = \{x \in \{0,1\}^* \mid x \text{ begins and ends with } 0 \text{ and } \text{length}(x) \geq 2\}$
- $L_2 \cdot L_1 = \{x \in \{0,1\}^* \mid x \text{ has } 00 \text{ as a substring}\}$

# Operations on Languages

## Kleene's closure (Star closure) of a Language

- Let  $L$  be a language over an alphabet  $\Sigma$ . The Kleene's closure of  $L$ , is obtained by concatenating zero or more elements of  $L$ .

- The **Kleene's closure** of  $L$ , given by

$$L^* = \{x \mid \text{For an integer } n \geq 0, x = x_1 x_2 \dots x_n \text{ and } x_1, x_2, \dots, x_n \text{ are in } L\}$$

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \dots$$

- $L^*$  is different than  $\Sigma^*(?)$ .

- Example:

- $\Sigma = \{0,1\}$  and  $L_e = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\}$
- $\Sigma^* = \{\lambda, 0, 1, 00, 01, 10, 11, \dots\}$
- $L_e^* = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\}$

$$\Sigma^* \neq L_e^*$$

- Let  $\Sigma = \{a, b\}$  and  $L = \{a^n b^n : n \geq 0\}$

$$\Sigma^* = ?$$

$$L^* = ?$$

$$L^2 = \{e, ab, aabb, aaabbb, \dots\} \quad \{e, ab, aabb, aaabbb, \dots\} = \{e, ab, aabb, aaabbb, \dots, ab, abab, abaabb, abaaabbb, \dots\}$$

$$L^3 = L^2 L^1 = \{ \}$$

$$L^* = \{a^n b^n a^m b^m : n, m \geq 0\}$$

# Operations on Languages

## Positive Closure of Language

- Let  $L$  be a language over an alphabet  $\Sigma$ . The positive closure of  $L$ , is obtained by concatenating one or more elements of  $L$ .
- The **positive closure** of  $L$ , given by
$$L^+ = \{ x \mid \text{for an integer } n \geq 1, x = x_1x_2\dots x_n \text{ and } x_1, x_2, \dots, x_n \text{ are in } L \}$$
- That is,**  $L^+ = \bigcup_{i=1}^{\infty} L^i = L \cdot L^* = L^1 \cup L^2 \cup L^3 \dots \dots$
- $L^0 = \{ \lambda \}$
- $L^k =$  The set of strings obtained by concatenating  $k$  elements of  $L$

Example:

Let  $\Sigma = \{0, 1\}$  be the alphabet.,  $L_e = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\}$

$L_e^+ = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\}$

$$\Sigma^* \neq L_e^* = L_e^+$$

$L_e^* = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\}$

- Find the closures of the following languages
  - $L_1 = \{a, b\}, \quad L_2 = \{11\}$

# Observations: Language Closure

$$L^+ = L^* - \{\lambda\} ?$$

$$L^* = L^+ \cup \{\lambda\} ?$$

- Example 1:
  - $L = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\}$
  - $L^+ = \{\omega \in \Sigma^* \mid \text{the number of 1's in } \omega \text{ is even}\} = L_e^*$
- Example 2:
  - $L = \{1, 0\}$
  - $L^+ = \{1, 0, 00, 01, 10, 11, 000, \dots\}, L_e^* = \{\lambda, 1, 0, 00, 01, 10, 11, 000, \dots\}$

## Observations

1. *If  $L$  is a language that does not contain  $\lambda$ , then  $L^+$  is the language  $L^*$  without the null word  $\lambda$ .*
2. *If  $L$  is a language that does contain  $\lambda$ , then  $L^+ = L^*$*
3. *Likewise, if  $\Sigma$  is an alphabet, then  $\Sigma^+$  is  $\Sigma^*$  without the word  $\lambda$ .*

# Languages and Decision Problems

- **Problem**
  - Example: What are prime numbers > 20?
- **Decision problem**
  - Problem with a YES/NO answer
  - Given a positive integer  $n$ , is  $n$  a prime number > 20?
- **Language**
  - Example:  $\{n \mid n \text{ is a prime number} > 20\}$   
=  $\{23, 29, 31, 37, \dots\}$
- A problem is represented by a set of strings of the input whose answer for the corresponding problem is “YES”.
- A string is in a language = the answer of the corresponding problem for the string is “YES”

Grammars

**Mathematical mechanisms to describe the languages**

# Grammars

- English grammar tells us if a given combination of words is a valid sentence **syntactically** as well as **semantically**.
- **Examples:** **The mouse wrote a poem.**
  - From a syntax point of view, it is a valid sentence.
  - From a semantics point of view, It is **not valid.....perhaps in Disney land**
- **The syntax of sentence is concerned with its structure.**
- The **semantic of a sentence** is concerned with the **meaning** of the sentence.
- Natural languages (English, Hindi, French, Urdu, German, etc)
  - Very complex rules of syntax
  - Not necessarily well-defined
  - Ambiguous

# Formal Language/Grammar

## Formal grammar

- It is used to identify correct or incorrect strings of alphabets in a language.
- It is used to generate all possible strings over the alphabet that is syntactically correct in the language.
- It is used mostly in the syntactic analysis phase (parsing), particularly during the compilation.
- A grammar implies an algorithm that would generate all legal sentences of the language.
  - Often, it takes the form of a set of recursive definitions.

**Formal Language-**The set of all strings generated by a grammar.

- **Two key questions:**
  - Is a combination of words a valid sentence(string) in a formal language?
  - How can we generate valid sentences of a formal language?
- Formal languages provide models for both **natural languages and programming languages**.
  - Amazon Alexa
  - Siri (Apple)
  - Compilers

# Grammars

Grammars express languages

Example: The English language.

## Rule:

1. A sentence -----consists of a noun phrase followed by a predicate.
2. A noun phrase consists of an article and noun.
3. Etc.

## Grammar's Production Rules

$\langle \text{sentence} \rangle \rightarrow \langle \text{noun\_phrase} \rangle \langle \text{predicate} \rangle$

$\langle \text{noun\_phrase} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle$

$\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle$

$\langle \text{article} \rangle \rightarrow a$   
 $\langle \text{article} \rangle \rightarrow \text{the}$

$\langle \text{noun} \rangle \rightarrow \text{cat}$   
 $\langle \text{noun} \rangle \rightarrow \text{dog}$   
 $\langle \text{noun} \rangle \rightarrow \text{boy}$

$\langle \text{verb} \rangle \rightarrow \text{runs}$   
 $\langle \text{verb} \rangle \rightarrow \text{walks}$   
 $\langle \text{verb} \rangle \rightarrow \text{sleeps}$

# Derivation

A derivation of “**the dog walks**”:

$\langle \text{sentence} \rangle \Rightarrow \langle \text{noun\_phrase} \rangle \langle \text{predicate} \rangle$   
 $\Rightarrow \langle \text{noun\_phrase} \rangle \langle \text{verb} \rangle$   
 $\Rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle$   
 $\Rightarrow \text{the} \langle \text{noun} \rangle \langle \text{verb} \rangle$   
 $\Rightarrow \text{the dog} \langle \text{verb} \rangle$   
 $\Rightarrow \text{the dog walks}$

$\langle \text{sentence} \rangle \rightarrow \langle \text{noun\_phrase} \rangle \langle \text{predicate} \rangle$   
 $\langle \text{noun\_phrase} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle$   
 $\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle$

A derivation of “**a cat runs**”:

$\langle \text{sentence} \rangle \Rightarrow \langle \text{noun\_phrase} \rangle \langle \text{predicate} \rangle$   
 $\Rightarrow \langle \text{noun\_phrase} \rangle \langle \text{verb} \rangle$   
 $\Rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle$   
 $\Rightarrow a \langle \text{noun} \rangle \langle \text{verb} \rangle$   
 $\Rightarrow a \text{ cat} \langle \text{verb} \rangle$   
 $\Rightarrow a \text{ cat runs}$

$\langle \text{article} \rangle \rightarrow a$   
 $\langle \text{article} \rangle \rightarrow \text{the}$   
 $\langle \text{noun} \rangle \rightarrow \text{cat}$   
 $\langle \text{noun} \rangle \rightarrow \text{dog}$   
 $\langle \text{noun} \rangle \rightarrow \text{boy}$   
 $\langle \text{verb} \rangle \rightarrow \text{runs}$   
 $\langle \text{verb} \rangle \rightarrow \text{walks}$   
 $\langle \text{verb} \rangle \rightarrow \text{sleeps}$

# Derivation

- The following sentences can be derived-

- “the boy sleeps”
- “the boy runs”
- “the boy walks”
- “**the boy eats**” ????
- “**the boy eats pizza**” ????

$$\langle \text{sentence} \rangle \rightarrow \langle \text{noun\_phrase} \rangle \langle \text{predicate} \rangle$$
$$\langle \text{noun\_phrase} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle$$
$$\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle$$

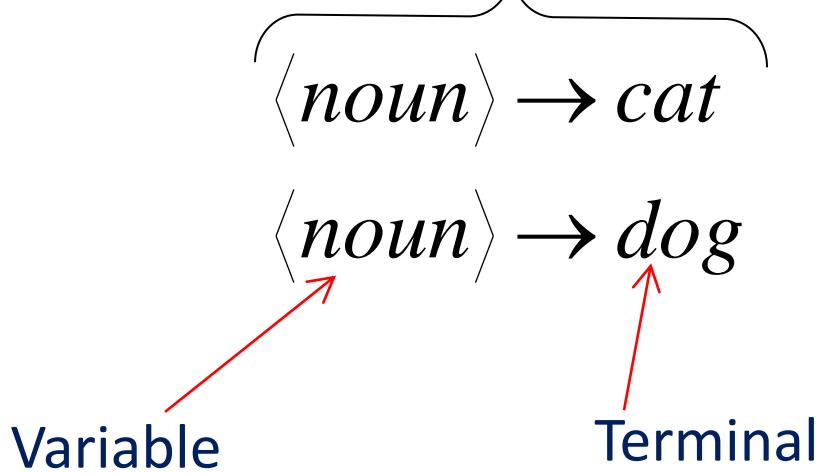
- The language of the grammar is the set of all strings (sentences) generated by the grammar.**

$L = L_G = \{ \text{“a cat runs”, “a cat walks”, “the cat runs”, “the cat walks”, “a dog runs”, “a dog walks”, “the dog runs”, “the dog walks”, “the boy sleeps”, ..... } \}$

$$\begin{aligned}\langle \text{article} \rangle &\rightarrow a \\ \langle \text{article} \rangle &\rightarrow \text{the}\end{aligned}$$
$$\begin{aligned}\langle \text{noun} \rangle &\rightarrow \text{cat} \\ \langle \text{noun} \rangle &\rightarrow \text{dog} \\ \langle \text{noun} \rangle &\rightarrow \text{boy}\end{aligned}$$
$$\begin{aligned}\langle \text{verb} \rangle &\rightarrow \text{runs} \\ \langle \text{verb} \rangle &\rightarrow \text{walks} \\ \langle \text{verb} \rangle &\rightarrow \text{sleeps}\end{aligned}$$

# Notation

## Production Rules



$\langle \text{sentence} \rangle \rightarrow \langle \text{noun\_phrase} \rangle \langle \text{predicate} \rangle$

$\langle \text{noun\_phrase} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle$

$\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle$

$\langle \text{article} \rangle \rightarrow \text{a}$

$\langle \text{article} \rangle \rightarrow \text{the}$

$\langle \text{noun} \rangle \rightarrow \text{cat}$

$\langle \text{noun} \rangle \rightarrow \text{dog}$

$\langle \text{noun} \rangle \rightarrow \text{boy}$

$\langle \text{verb} \rangle \rightarrow \text{runs}$

$\langle \text{verb} \rangle \rightarrow \text{walks}$

$\langle \text{verb} \rangle \rightarrow \text{sleeps}$

Variable = { $\langle \text{sentence} \rangle$ ,  $\langle \text{noun\_phrase} \rangle$ ,  $\langle \text{predicate} \rangle$ ,  $\langle \text{article} \rangle$ ,  $\langle \text{noun} \rangle$ ,  $\langle \text{verb} \rangle$  }

Terminal = {a, the, dog, cat, boy, runs, walks, sleeps}

# Grammar

A grammar G is defined as quadruple  $G = (V, T, S, P)$

- V --- **Finite and non-empty set of variables** (non-terminals)
- T --- Finite and non-empty set of terminal symbols
- S --- Start variable,  $S \in V$
- P --- Finite set of Production rules

## Note:

- V and T are disjoint sets

## Note:

- Every production rule must contain at least one non-terminal on its left side.
- Form of production rule

$$\begin{array}{ccc} x & \xrightarrow{\quad} & y \\ x \in (V \cup T)^* & & y \in (V \cup T)^* \end{array}$$

- Production rules-
  - Specify how to generate a string.
  - Specify how the grammar transforms one sentential form into another sentential form/string.
  - Define the language associated with grammar.

# Production Rules

- Production rule

$$P_1: \quad x \xrightarrow{\hspace{1cm}} y$$

$$w = uxv, z = u y v, \exists x, y, u, v \in (T \cup V)^*$$

- Production rule  $P_1$  is applicable to string  $w$ 
  - $x$  can be replaced with  $y$ , to form a new string

- This can be written as:  $w \xrightarrow{\hspace{1cm}} z$

w derives z

# Grammar-Example

Grammar:

$$\begin{array}{c} G \\ \downarrow \\ G = (V, T, S, P) \\ \uparrow \quad \uparrow \quad \uparrow \\ V = \{S\} \quad T = \{a, b\} \\ \quad \quad \quad P = \{S \rightarrow aSb, S \rightarrow \lambda\} \end{array}$$

What strings/sentences can be generated with this grammar?  
(What is the language generated by this grammar?)

Sentential Form:

- A finite sequence of variables and terminals.

Sentence/String:

- A finite sequence of terminals.

Derivation of string:

$$\begin{array}{ccccccc} S & \Rightarrow & aSb & \Rightarrow & aaSbb & \Rightarrow & aaaSbbb \\ & & \uparrow & & \uparrow & & \uparrow \\ & & \text{Sentential Forms} & & \text{Sentential Forms} & & \text{Sentence/String} \end{array}$$

# More Notation

We write:  $S \xrightarrow{*} aaabbb$

Instead of:  $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$

The \* indicates an unspecified number of steps (Zero or more steps).

If  $w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \dots \Rightarrow w_n$

Then, we can write  $w_1 \xrightarrow{*} w_n$

**By default:**  $w \xrightarrow{*} w$

# Example

## Grammar

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

## Derivation 1

$$\overset{*}{S} \Rightarrow aaSbb$$

$$\overset{*}{aaSbb} \Rightarrow aaaaaSbbbbbb$$

## Derivations

$$\overset{*}{S} \Rightarrow \lambda$$

\*

$$\overset{*}{S} \Rightarrow ab$$

\*

$$\overset{*}{S} \Rightarrow aabb$$

\*

$$\overset{*}{S} \Rightarrow aaabbb$$

# Another Grammar Example

Grammar  $G$ :

$$S \rightarrow Ab$$
$$A \rightarrow aAb$$
$$A \rightarrow \lambda$$

Derivations:

$$S \Rightarrow A b \Rightarrow$$
$$S \Rightarrow A b \Rightarrow a A b b \Rightarrow a b b$$
$$S \Rightarrow A b \Rightarrow a A b b \Rightarrow a a A b b b \Rightarrow a a b b b$$

Derivations:

$$S \Rightarrow A b \Rightarrow a A b b \Rightarrow a a A b b b \Rightarrow a a a A b b b b$$
$$\Rightarrow a a a a A b b b b b \Rightarrow a a a a b b b b b b$$

# Language of a Grammar

- Let  $G = (V, T, S, P)$  be a **grammar**. The language generated by  $G$  (or the language of  $G$ ) denoted by  $L(G)$ , is the set of all strings that can be derived from the starting variable  $S$ .

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$$

- That is,  $L(G)$  is simply the set of strings of terminals that can be derived from the start symbol.**

- Example-*  
 $G = (V, T, S, P); V = \{A, S\}, T = \{a, b\}, S$  is a start symbol  
 $P = \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\}$ .

$$T^* = \{e, a, b, aa, bb, ab, ba, aaa, aab, aba, abb, baa, bab, \dots\}$$

- The language of this grammar is given by  $L(G) = \{b, aaa\}$ 
  - we can derive  $aA$  from  $S$  using  $S \rightarrow aA$ , and then derive  $aaa$  using  $A \rightarrow aa$ .
  - We can also derive  $b$  using  $S \rightarrow b$ .

# Language of a Given Grammar: How to prove ?

- $L$  – A given language
  - $L(G)$  – Language of a grammar
- 
- Every string generated by  $G$  is in  $L$  i.e.,  $\textcolor{red}{L(G)} \subseteq L$ .
  - Every string in  $L$  can be generated by  $G$  i.e.,  $L \subseteq \textcolor{red}{L(G)}$ .

$$G = (V, T, S, P)$$

$$V = \{S\}$$

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

$$P = \{S \rightarrow aSb \mid \lambda\}$$

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

# Language of a Given Grammar

What's the language of the grammar with  
the productions?

$$S \rightarrow aSb$$
$$S \rightarrow \lambda$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

$$\begin{aligned} S &\Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \\ &\Rightarrow aaaaSbbbb \Rightarrow aaaabbbb \end{aligned}$$

$$L = \{a^n b^n : n \geq 0\}$$

Find languages of the following grammars-

- G1:  $S \rightarrow aaA \mid \lambda, A \rightarrow bS$  --  $L = \{(aab)^n : n \geq 0\}$
- G2:  $S \rightarrow Aa, A \rightarrow B, B \rightarrow Aa$  --  $L = \{ \}$
- G3:  $S \rightarrow a \mid b \mid aSa \mid bSb \mid \lambda$  ---  $L = \{ w \text{ is in } \{a, b\}^* : w \text{ is palindrome} \}$

# Equivalent Grammars

- Two grammars  $G_1$  and  $G_2$  are equivalent if they generate the same language, that is, if  $L(G_1) = L(G_2)$
- First grammar  $G_1 \quad S \rightarrow aSb \quad L(G1) = \{a^n b^n : n \geq 0\}$   
 $S \rightarrow \lambda$
- Second grammar  $G_2 \quad S \rightarrow aAb \mid \lambda \quad L(G2) = \{a^n b^n : n \geq 0\}$   
 $A \rightarrow aAb \mid \lambda$
- The language generated by both grammars is  $L = \{a^n b^n : n \geq 0\}$
- Grammars  $G_1$  and  $G_2$  are equivalent.

# Grammar for a given Language

- Procedure-
  - Write grammar rules to generate all strings in the language.
- Write production rules that generate the strings in the language.

1.  $L = \{w \in \{a\}^*\} = \{e, a, aa, aaa, aaaa, \dots\}$

**$S \rightarrow e \mid aS$**

2.  $L = \{w \in \{a, b\}^*\} = \{e, a, b, aa, ab, ba, bb, \dots\}$

**$S \rightarrow e \mid aS \mid bS$**

3.  $L = \{w \in \{a, b\}^*: w \text{ has exactly one } a\} = \{a, ab, ba, abb, bab, bba, \dots\}$

**$S \rightarrow XaX$**

**$X \rightarrow bX \mid e$**

4.  $L = \{w \in \{a, b\}^*: w \text{ has at least one } a\} = \{a, aab, bab, aa, aaaa, \dots, aaaabbbaaa, \dots\}$

**$S \rightarrow XaX, X \rightarrow aX \mid bX \mid e$**

5.  $L = \{w \in \{a, b\}^*: w \text{ has no more than three } a's\} = \{e, a, b, aab, ab, bba, babab, bbbb, \dots\}$

**$S \rightarrow e \mid XaX \mid XaXaX \mid XaXaXaX, X \rightarrow bX \mid e$**

6.  $L = \{w \in \{a, b\}^*: n_a(w) = n_b(w)\} = \{e, ab, aabb, ba, aabaabbb, aabbab, \dots\}$

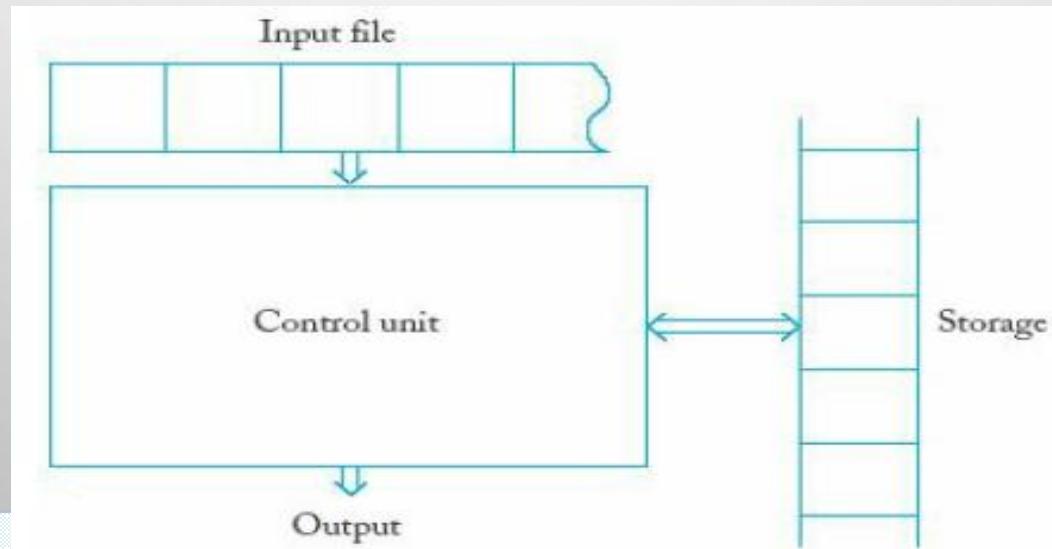
**$S \rightarrow e \mid aSb \mid bSa \mid SS$**

**Thank You**

# *AUTOMATA*

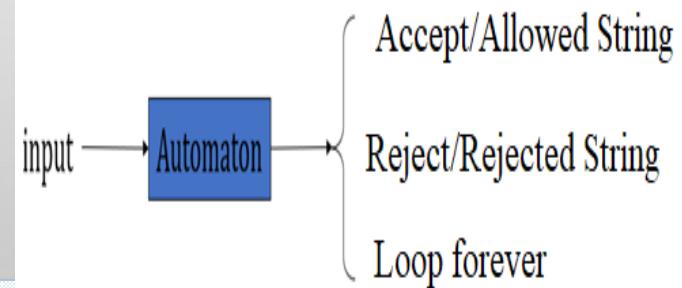
# AUTOMATA

- An automaton is an abstract model of a digital computer.
  - It can read input.
  - It **may have** a temporary storage device.
  - It can produce output.
  - It has a control unit
    - It can be in any one of a finite number of internal states.
    - It can also change the state in some defined manner.



# Categories

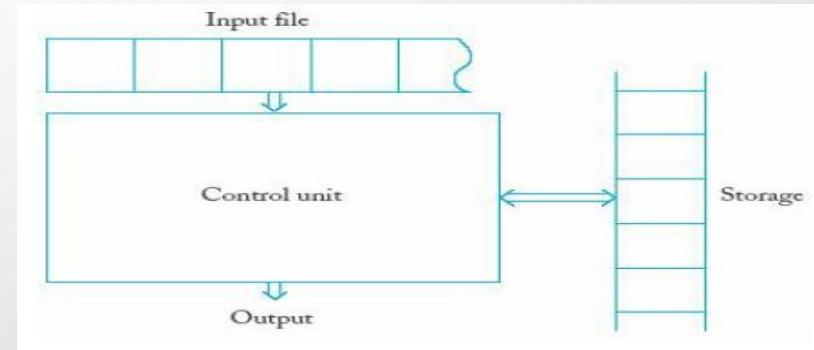
- Automata can be divided into the following categories based on the **availability of temporary memory**
  - **Finite Automata** No temporary memory
  - **Pushdown Automata** Stack
  - **Turing Machines** Random access memory
- Automata can be divided into the following categories based on the **output generated.**
  - **Acceptor:** An automaton whose output response is limited to a simple “yes” or “no” is known as an accepter.
  - **Transducer:** An automaton capable of producing strings of symbols as output, is known as a transducer.



# Categories

- **Deterministic Automata**

- A deterministic automaton is one in which each move is uniquely determined by the current configuration.
- **Current configuration- Internal state + Current input Symbol +Storages' content**

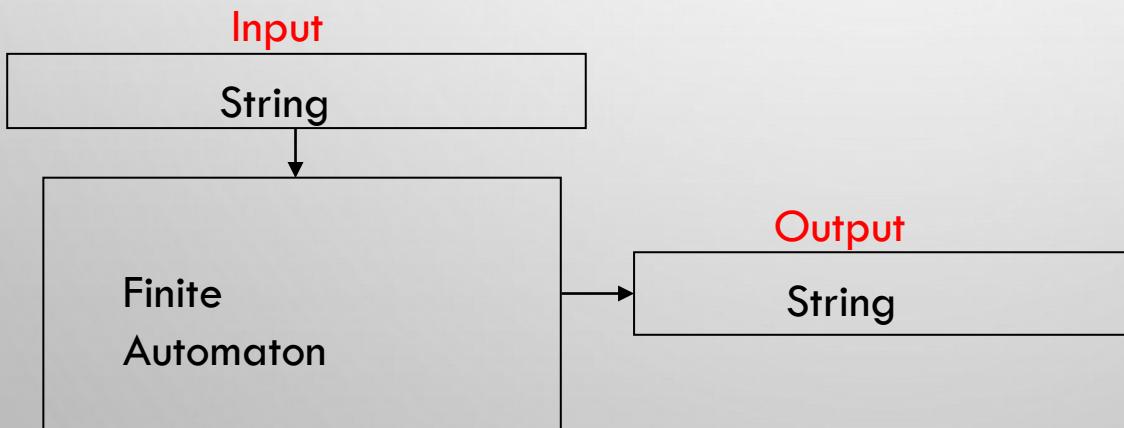


- **Non-deterministic automata**

- A non-deterministic automaton is one in which each move is **not** uniquely determined by the current configuration.

# FINITE AUTOMATA

- Finite automata are automata with
  - **No temporary memory**
  - Acceptor/Transducer
  - Deterministic/Non-deterministic
- A finite automaton has a finite number of states and **can remember only a limited number of things.**



- Acceptor
  - DFA
  - NFA
- Transducer
  - Moore M/c
  - Mealy M/c

# Deterministic Finite Acceptor (DFA)

- It is a simple model of computation.
  - It has no temporary storage.
  - It consists of a finite number of states.
  - It is **deterministic**.
    - Next move is uniquely determined by the current configuration.
  - It is an **accepter**.
    - The output response is limited to a simple “yes” or “no”.
  - It maintains the finite amount of information about what is seen so far (memory) by placing the control unit into a specific state.
    - A finite automaton has a finite number of states, and cannot remember more things for longer inputs.

# Deterministic Finite Acceptor (DFA)

A DFA is a quintuple  $M = (Q, \Sigma, \delta, q_0, F)$ , where:

- $Q$  is a finite set of internal states,
- $\Sigma$  is a finite set of symbols (input alphabet, without empty string),
- **$\delta: Q \times \Sigma \rightarrow Q$  is the transition function of  $M$**
- $q_0$  - initial state ( $q_0 \in Q$ )
- $F$  is a subset of  $Q$  and represents the set of final or accepting states of  $M$ .

**What do you mean by  $\delta$  ?**

# Finite Automata Representation

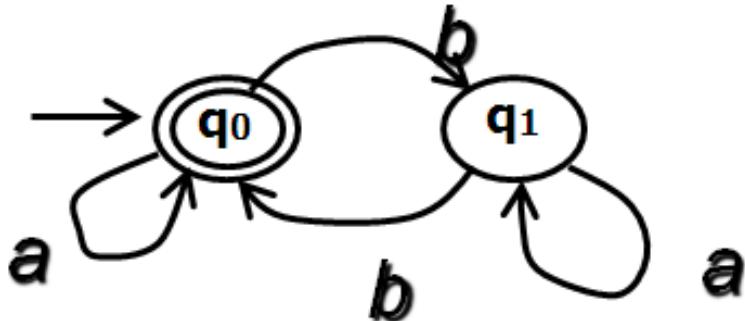
- **Representation 1: Transition graph/bubble diagram**

- **Good for visualizing behavior**

- **Representation 2: Transition table**

- **Good for computer simulation**

$$M = (Q, \Sigma, \delta, q_0, F)$$



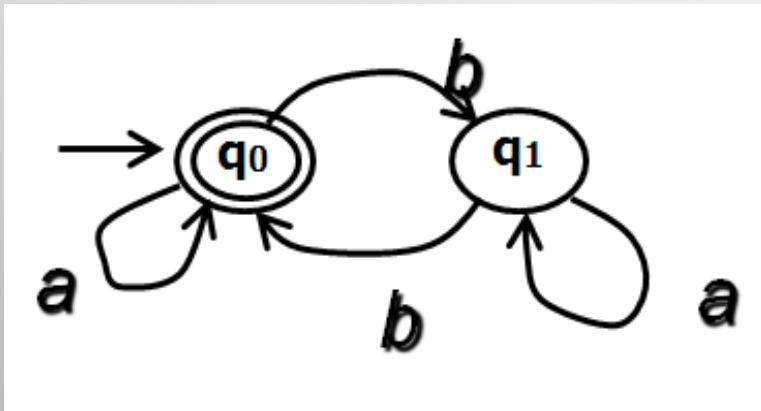
	<i>a</i>	<i>b</i>
<i>q</i> <sub>0</sub>	<i>q</i> <sub>0</sub>	<i>q</i> <sub>1</sub>
<i>q</i> <sub>1</sub>	<i>q</i> <sub>1</sub>	<i>q</i> <sub>0</sub>

- The circles represent states, with a double circle being an accepting state.
- The arcs represent the transitions of the transition function.
- The state with an incoming edge (not originating from any state) is an initial state.

$$M = (Q, \Sigma, \delta, q_0, F) = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_0\})$$

# DFA-Deterministic ?

- The transition function makes the automaton deterministic
  - At each moment in time, the automaton has exact instructions as to what will happen next.
  - **Current configuration- Internal state + Current input Symbol**



	<i>a</i>	<i>b</i>
$q_0$	$q_0$	$q_1$
$q_1$	$q_1$	$q_0$

# DFA FINITE INFORMATION . HOW?

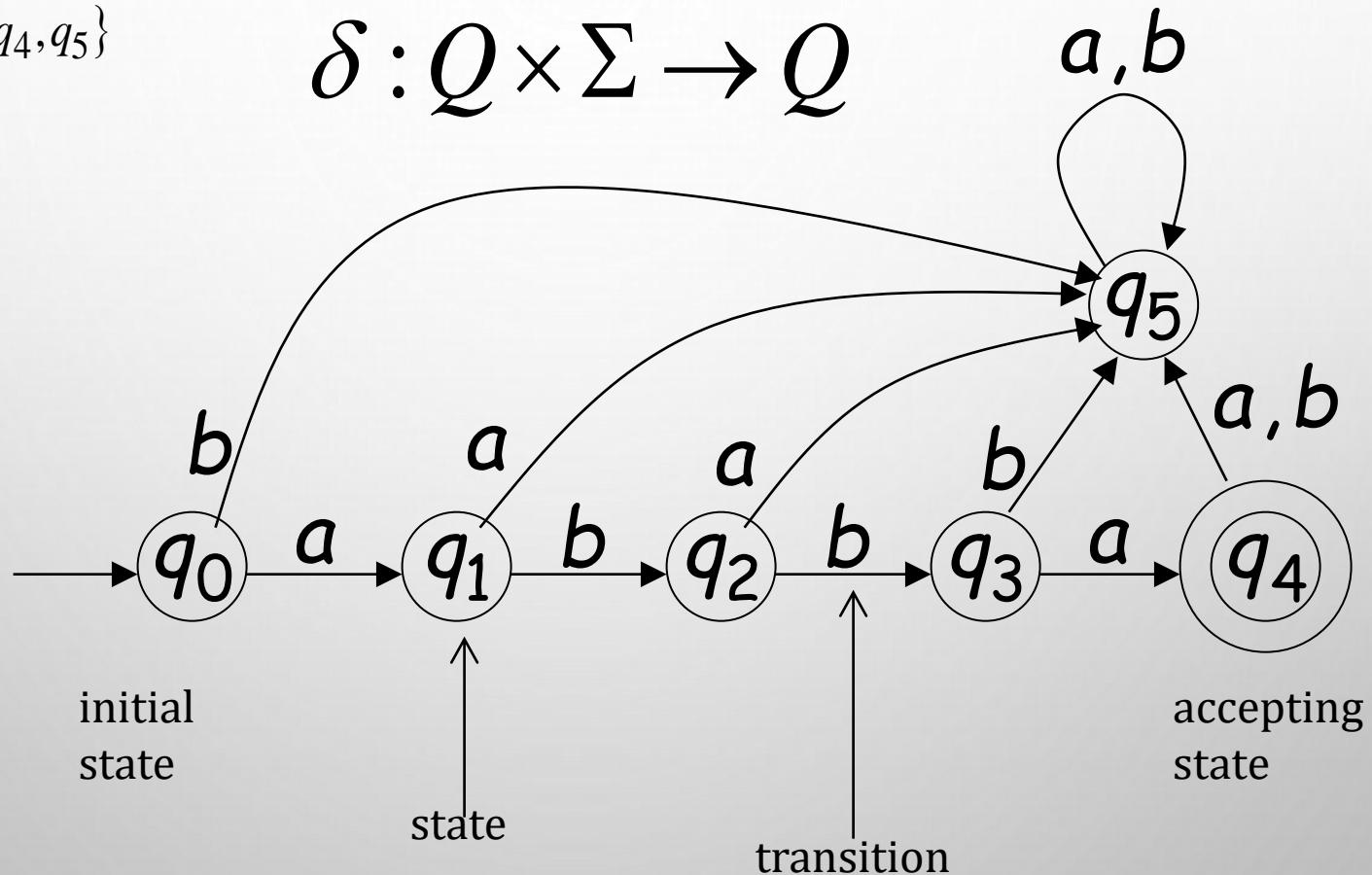
**DFA  $M = (Q, \Sigma, \delta, q_0, F)$**

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$

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

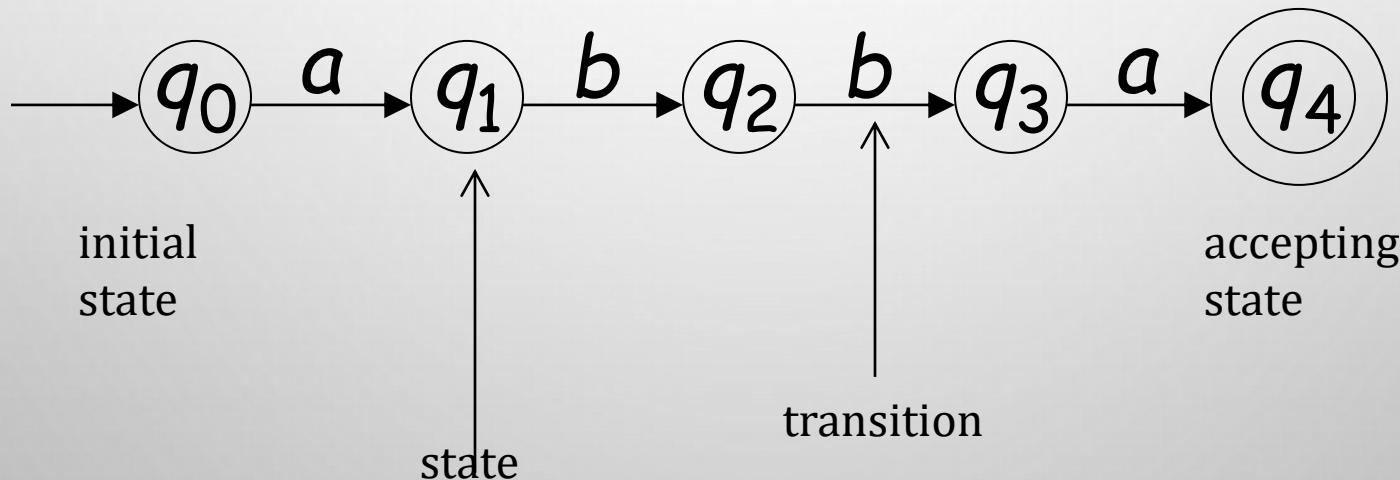
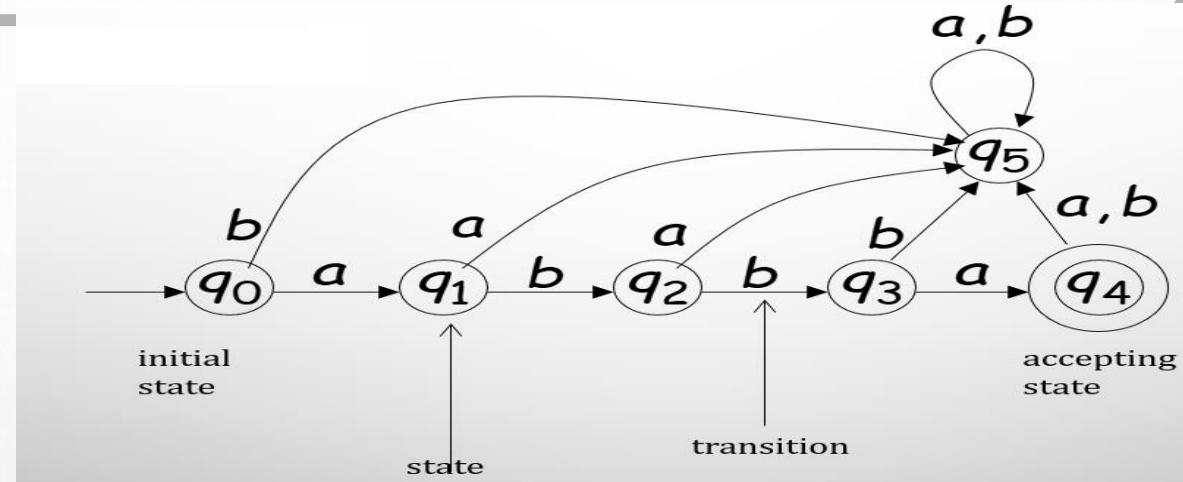
$$F = \{q_4\}$$

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



# HOW TO IDENTIFY A DFA?

$$M = (Q, \Sigma, \delta, q_0, F)$$



*Complete, incomplete, and NFA?*

# DFA-OPERATIONS

## ACCEPT/REJECT (A/R)

- The machine always starts in the initial state looking at the leftmost symbol.
- The transition function specifies, for each state and symbol, the next state the automaton will enter.
  - $\delta(q_1, b) = q_2$ , refers that if the automaton is in state  $q_1$  and the next symbol is  $b$ , then the state of the automaton becomes state  $q_2$ .
- The automaton stops after reading the rightmost symbol (i.e. Reading all symbols in the string).
  - If it stops in a non-final state, the string is REJECTED.
  - If it stops in a final state, the string is ACCEPTED.

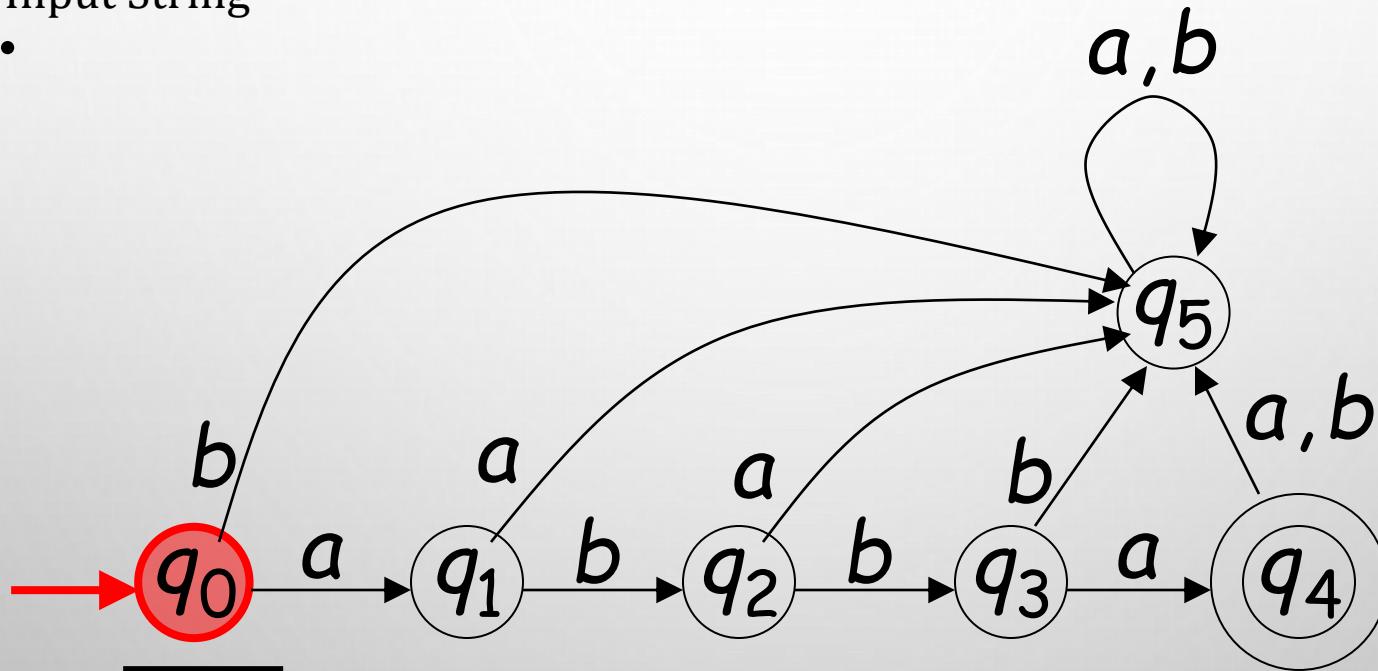
# (DFA)-INITIAL CONFIGURATION

↓

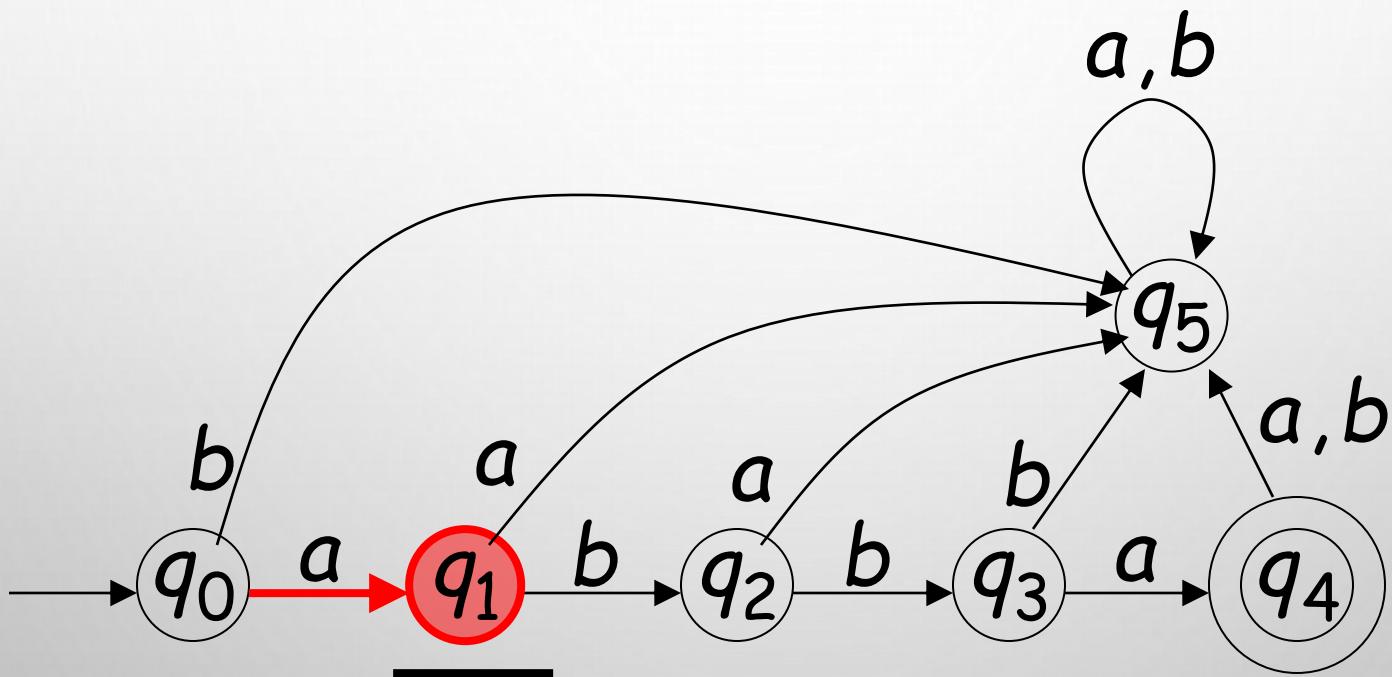
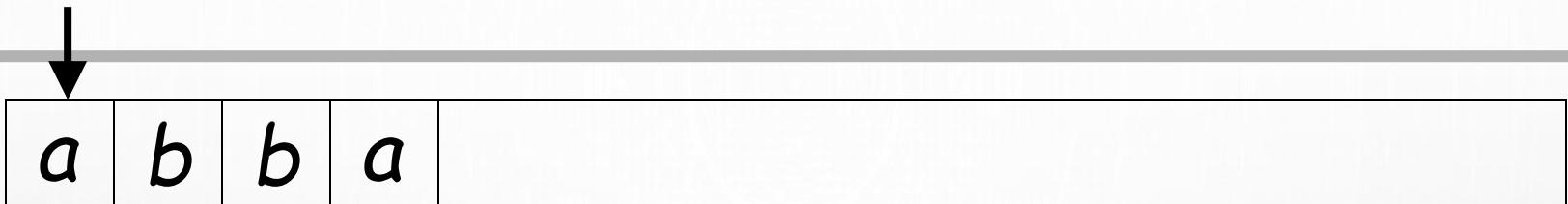
a	b	b	a	
---	---	---	---	--

Input String

.



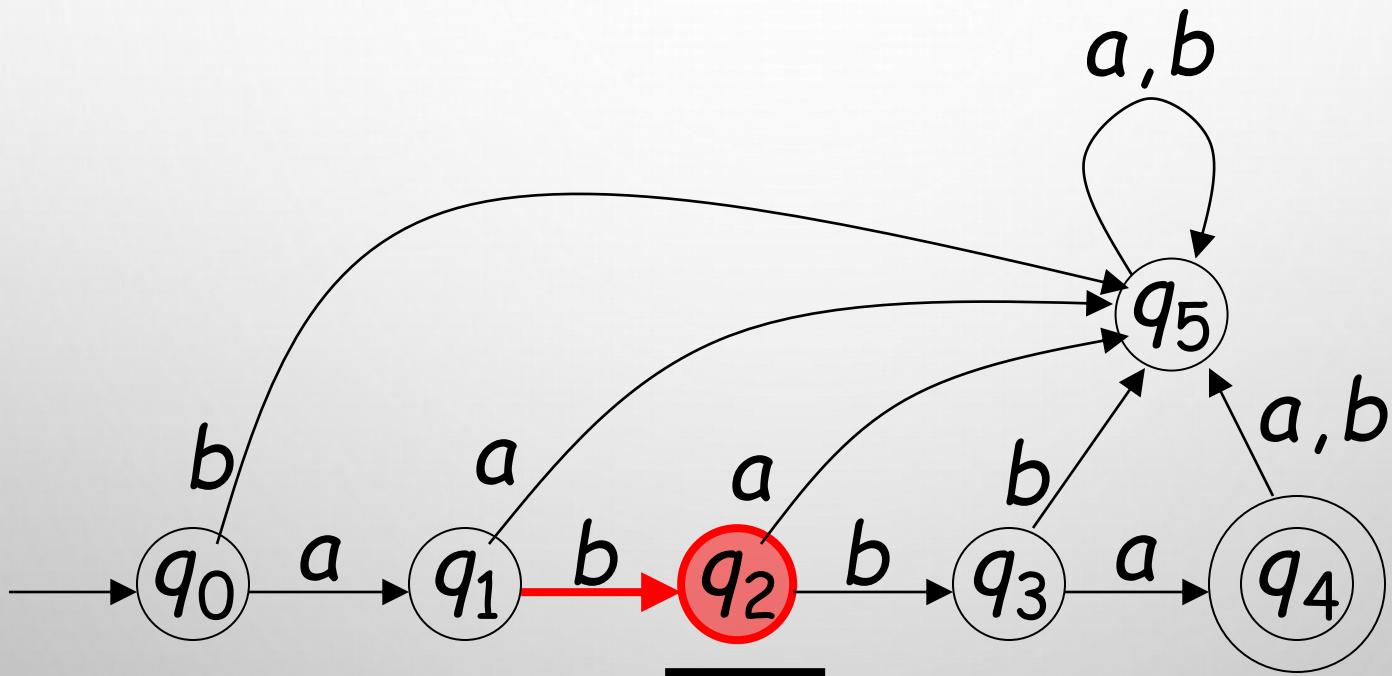
# READING THE INPUT



# NEXT TRANSITION



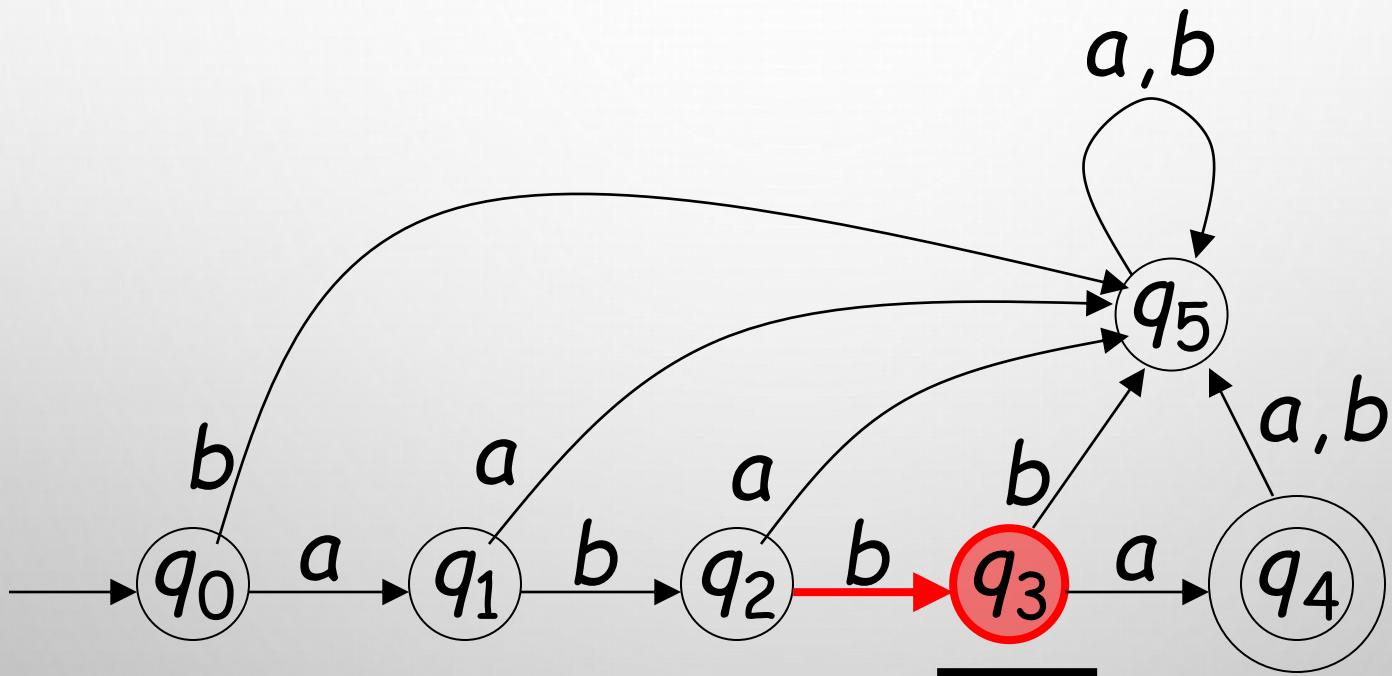
a	b	b	a	
---	---	---	---	--



# NEXT TRANSITION

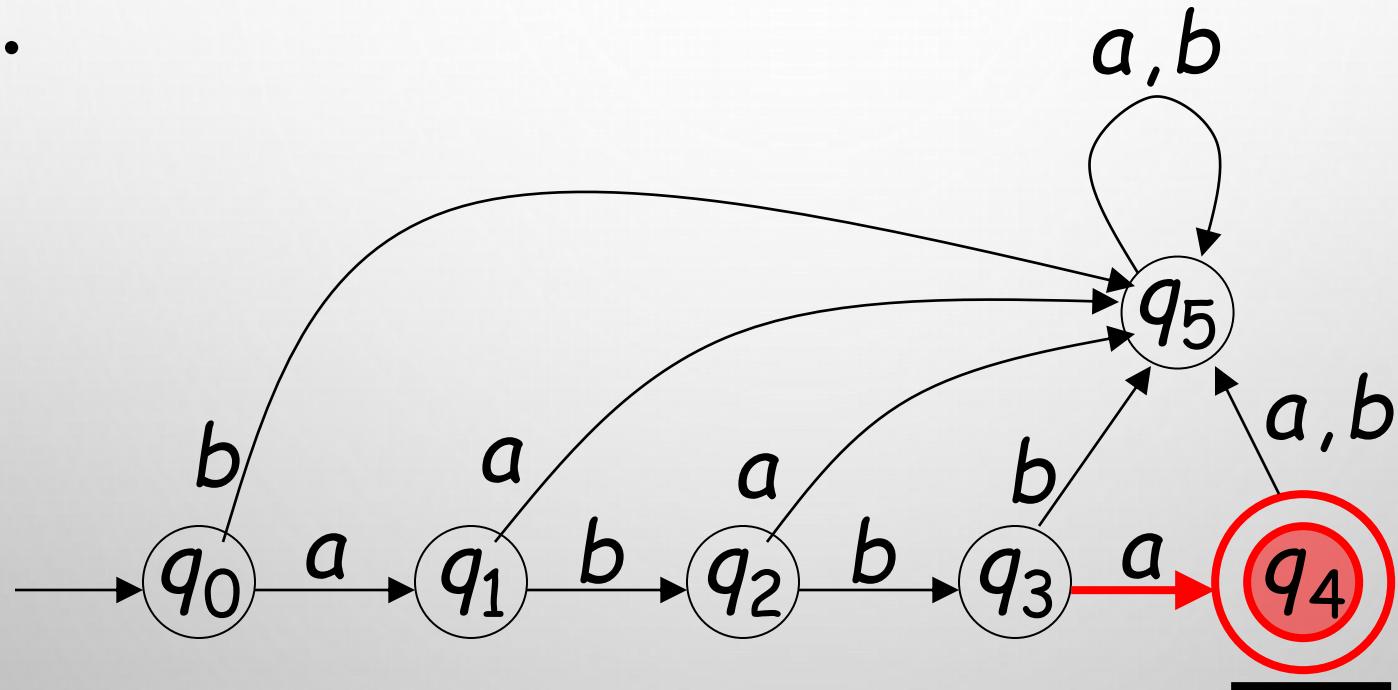


a	b	b	a	
---	---	---	---	--



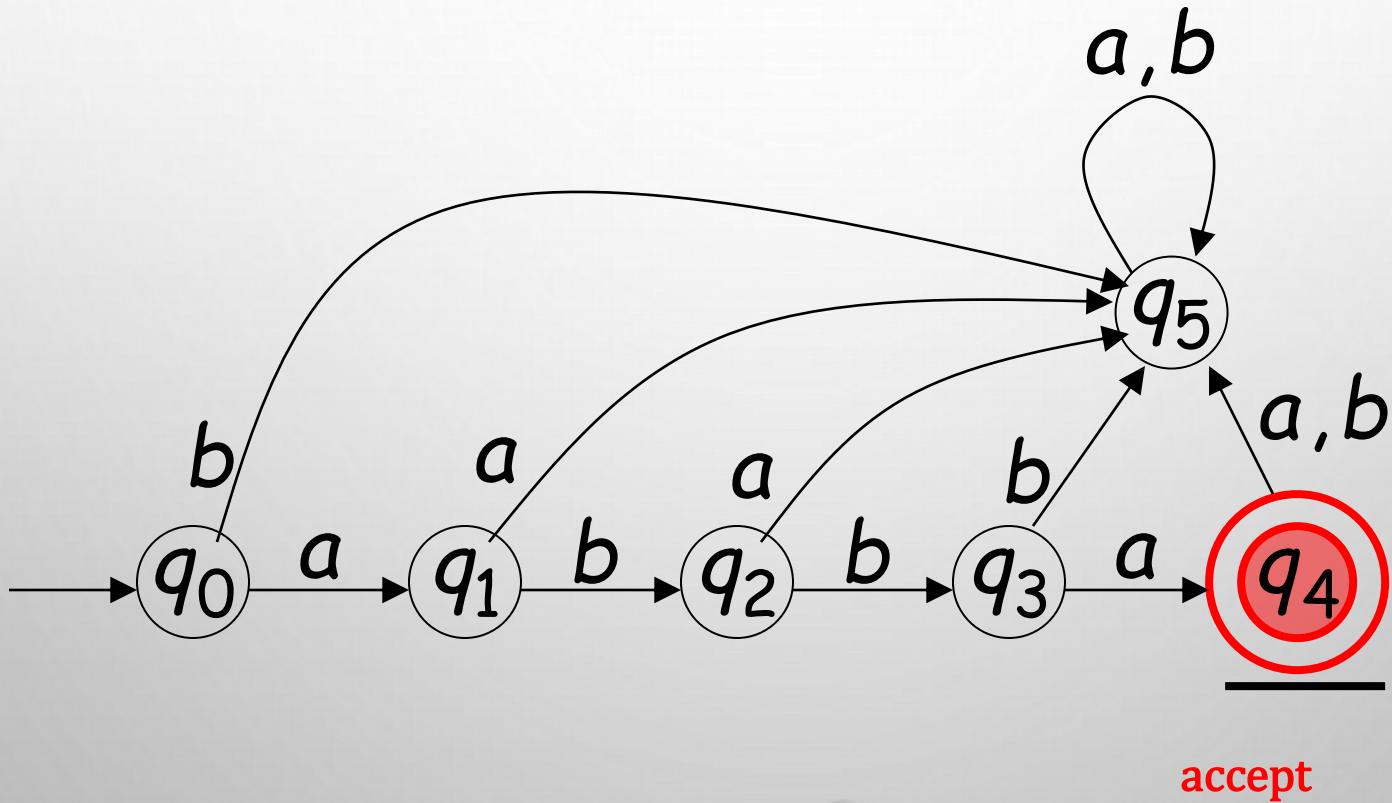
# LAST TRANSITION

a	b	b	a	
---	---	---	---	--



# Input finished

a	b	b	a	
---	---	---	---	--



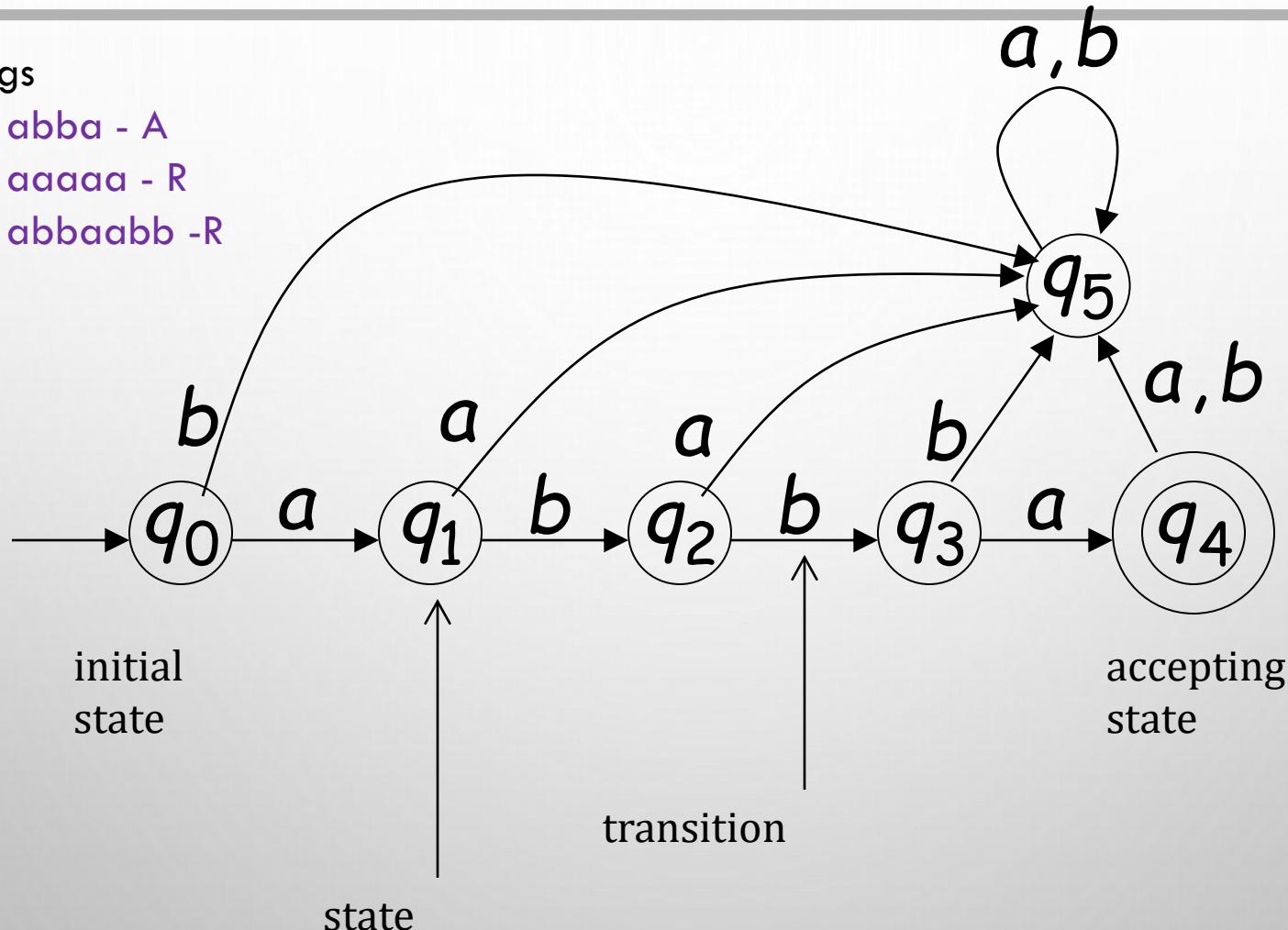
# DFA-EXAMPLE (ACCEPT/REJECT)

Strings

abba - A

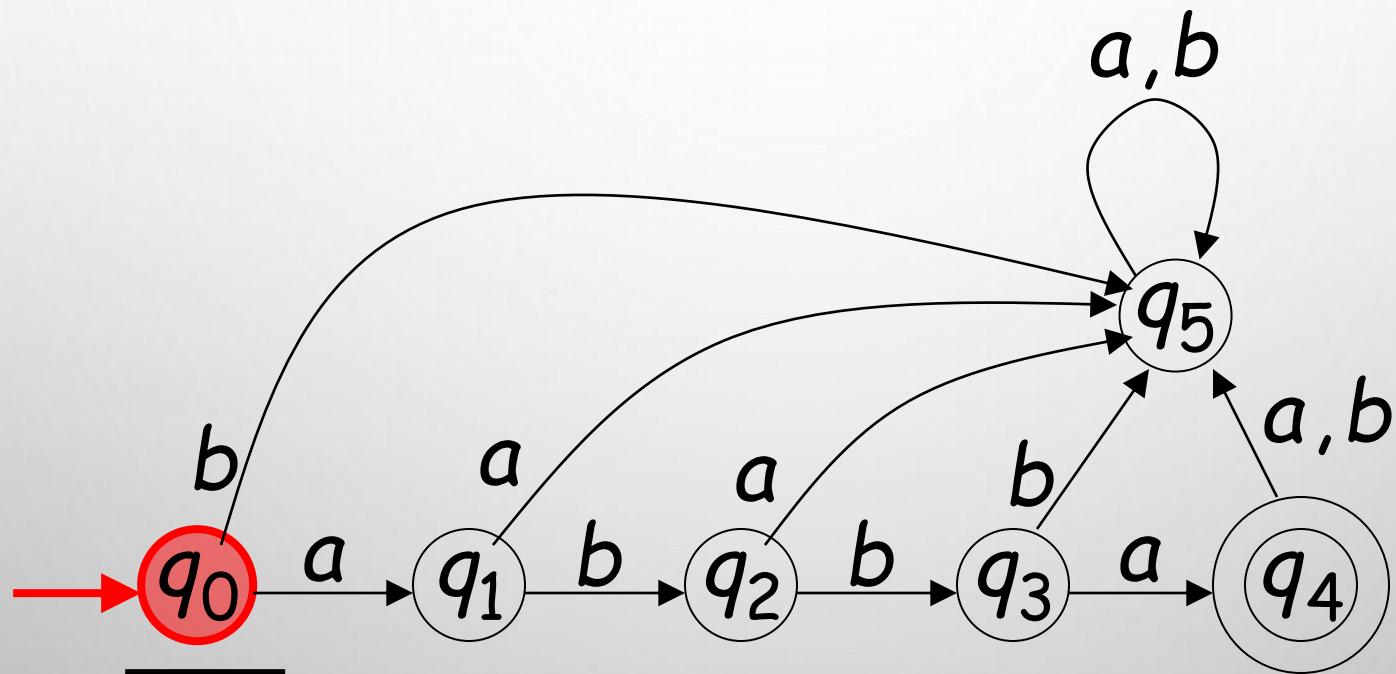
aaaaaa - R

abbaabb - R



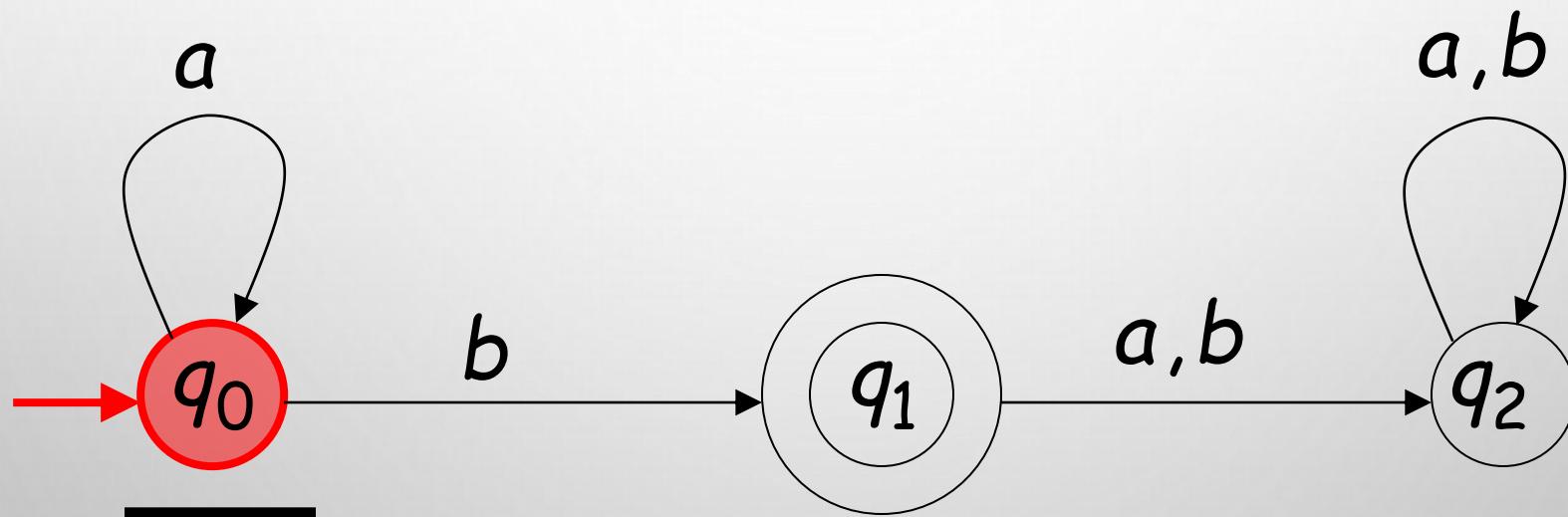
# DFA-Example-(Accept/Reject)

$\lambda$



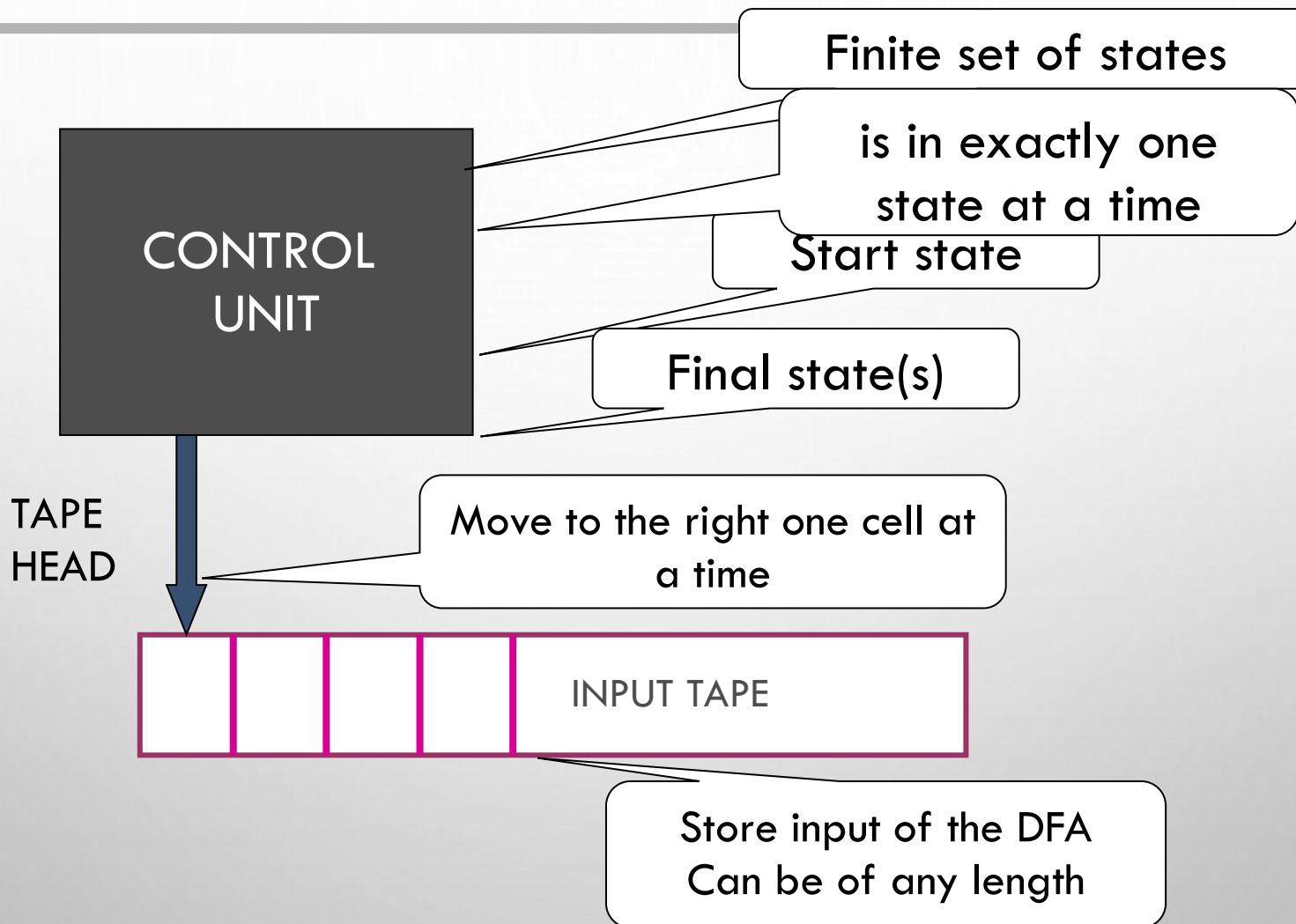
# DFA-Example (Accept/Reject)

a	a	b	
---	---	---	--



aab is Accepted

# DETERMINISTIC FINITE AUTOMATA/ACCEPTER



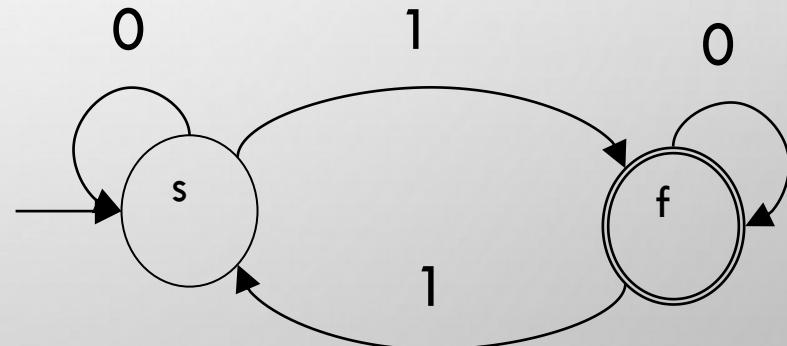
# CONFIGURATIONS

**DEFINITION:** LET  $M = (Q, \Sigma, \delta, S, F)$  BE A DETERMINISTIC FINITE AUTOMATON.

A CONFIGURATION OF  $M$  IS AN ELEMENT OF  $Q \times \Sigma^*$

## Configurations of $M$

- |                          |                     |
|--------------------------|---------------------|
| $(s, \varepsilon),$      | $(s, 00),$          |
| $(s, 01101),$            | $(f, \varepsilon),$ |
| $(f, 0011011001),$       |                     |
| $(f, 1111), (f, 011110)$ |                     |



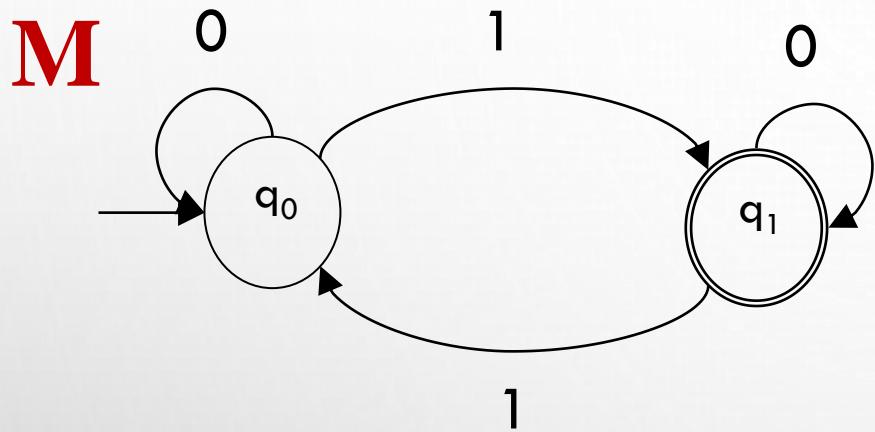
Transition Diagram

# YIELDING NEXT CONFIGURATION

For a DFA,  $M = (Q, \Sigma, \delta, s, F)$ ,  $(q_0, \omega_0)$  and  $(q_1, \omega_1)$  are the two (consecutive) configurations of  $M$ :

- We say  $(q_0, \omega_0)$  yields  $(q_1, \omega_1)$  in one step, .....denoted by  $(q_0, \omega_0) \vdash_M (q_1, \omega_1)$ , if  $\delta(q_0, a) = q_1$ , and  $\omega_0 = a \omega_1$ , for some  $a \in \Sigma$ .
- When a configuration yields another configuration, the first symbol of the string is read and discarded from the configuration.
- Once a symbol from an input string is read, it does not affect how the DFA works in the future.

# YIELDING NEXT CONFIGURATION



0 0 1 1 0 1  
↑ ↑ ↑ ↑ ↑ ↑

$(q_0, 001101)$   
 $\vdash_M (q_0, 01101)$   
 $\vdash_M (q_0, 1101)$   
 $\vdash_M (q_1, 101)$   
 $\vdash_M (q_0, 01)$   
 $\vdash_M (q_0, 1)$   
 $\vdash_M (q_1, \lambda)$

$(q_0, 0111) \vdash_M (q_0, 111)$   
 $\vdash_M (q_1, 11) \vdash_M (q_0, 1)$   
 $\vdash_M (q_1, \lambda)$

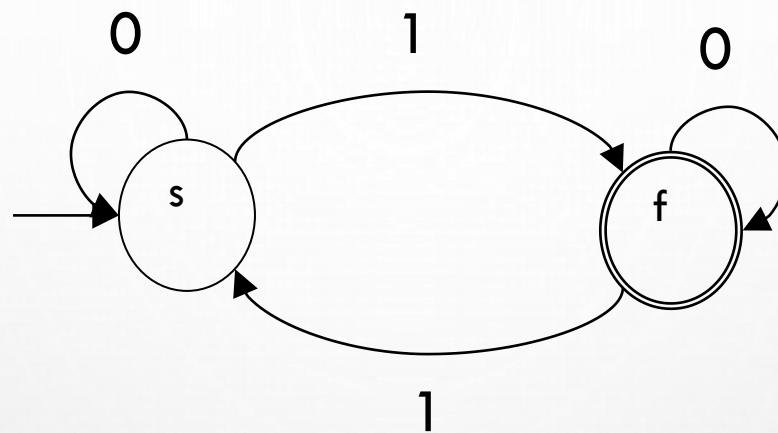
# YIELD IN ZERO STEP OR MORE STEPS

For a DFA,  $M = (Q, \Sigma, \delta, s, F)$ , with  $(q_0, \omega_0)$  and  $(q_1, \omega_1)$  be two configurations of  $M$ , the configuration  $(q_0, \omega_0)$  yields  $(q_1, \omega_1)$  in zero step or more, denoted by  $(q_0, \omega_0) \vdash_M^* (q_1, \omega_1)$ , if

- $q_0 = q_1$  and  $\omega_0 = \omega_1$ , or
- $(q_0, \omega_0) \vdash_M (q_2, \omega_2)$  and  $(q_2, \omega_2) \vdash_M^* (q_1, \omega_1)$  for some  $q_2$  and  $\omega_2$ .

**Note:** this definition is an inductive definition, and it is often used to prove properties of DFA's.

# YIELDING IN ZERO OR MORE STEPS



$(s, 001101)$

$\vdash^*_M (f, 101)$

$\vdash^*_M (s, 01)$

$\vdash^*_M (f, \varepsilon)$

$\vdash^*_M (f, \varepsilon)$

$(s, 01001)$

$\vdash^*_M (f, 001)$

$\vdash^*_M (f, 001)$

$\vdash^*_M (f, 1)$

$\vdash^*_M (s, \varepsilon)$

# EXTENDED TRANSITION FUNCTION(ETF)

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

$$\delta^*(q, w) = q'$$

It describes the resulting state after scanning string  $w$  starting from state  $q$ .

For any state  $q$ ,  $\delta^*(q, \lambda) = q$

For all  $q \in Q, w \in \Sigma^*, a \in \Sigma$   $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$

**$\delta: Q \times \Sigma \rightarrow Q$  is the transition function of M**  
 **$\delta(q, a) = q' \quad a \in \Sigma$**

# ETF EXAMPLE

$$\delta^*(q_0, ab) = q_2$$

$$\delta^*(q, \lambda) = q$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a)$$

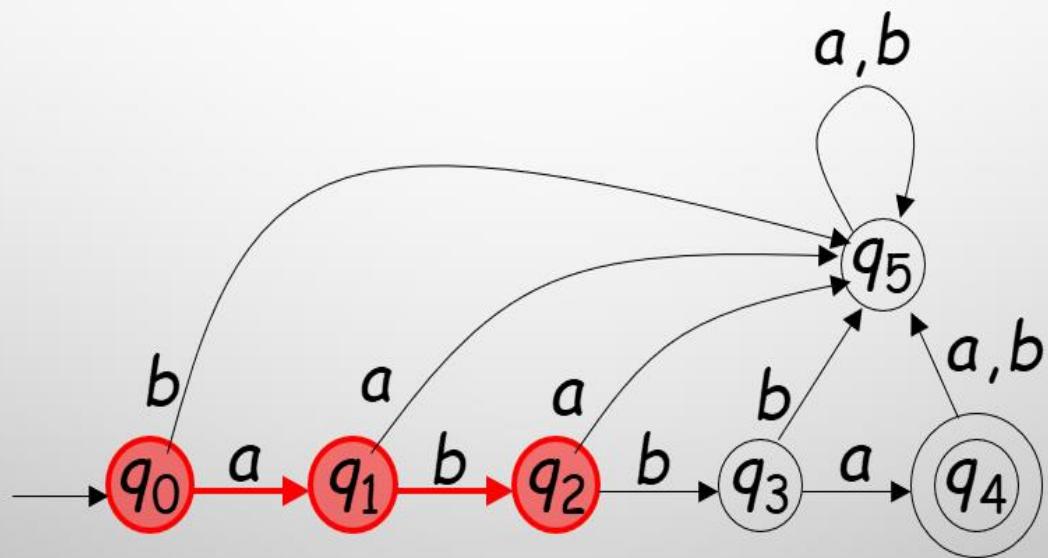
$$\begin{aligned}\delta^*(q_0, ab) &= \delta(\delta^*(q_0, a), b) = \delta(\delta(\delta^*(q_0, \lambda), a), b) \\ &= \delta(\delta(q_0, a), b) = \delta(q_1, b) = q_2\end{aligned}$$

Example-

abba

aabb

abbaaa



# ETF Observation

**Observation:** If there is a walk from  $q$  to  $q'$  with label  $\mathcal{W}$ , then

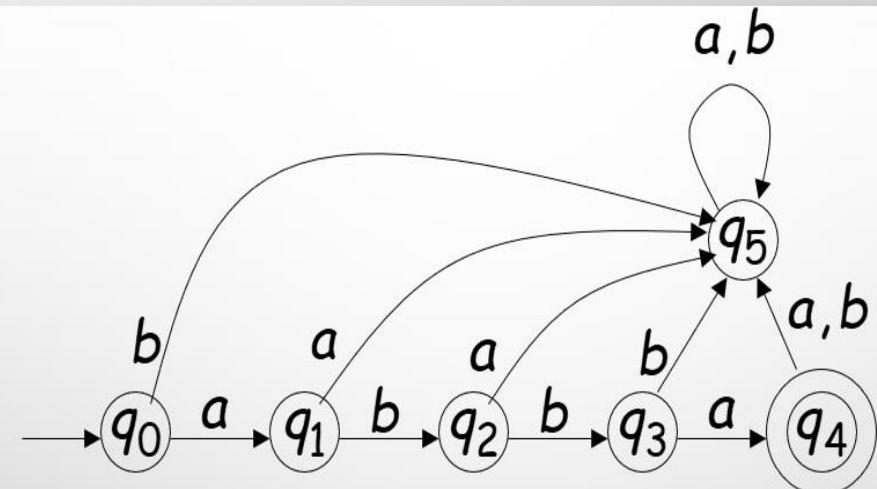
$$\delta^*(q, w) = q'$$



$$w = \sigma_1 \sigma_2 \Lambda \sigma_k$$

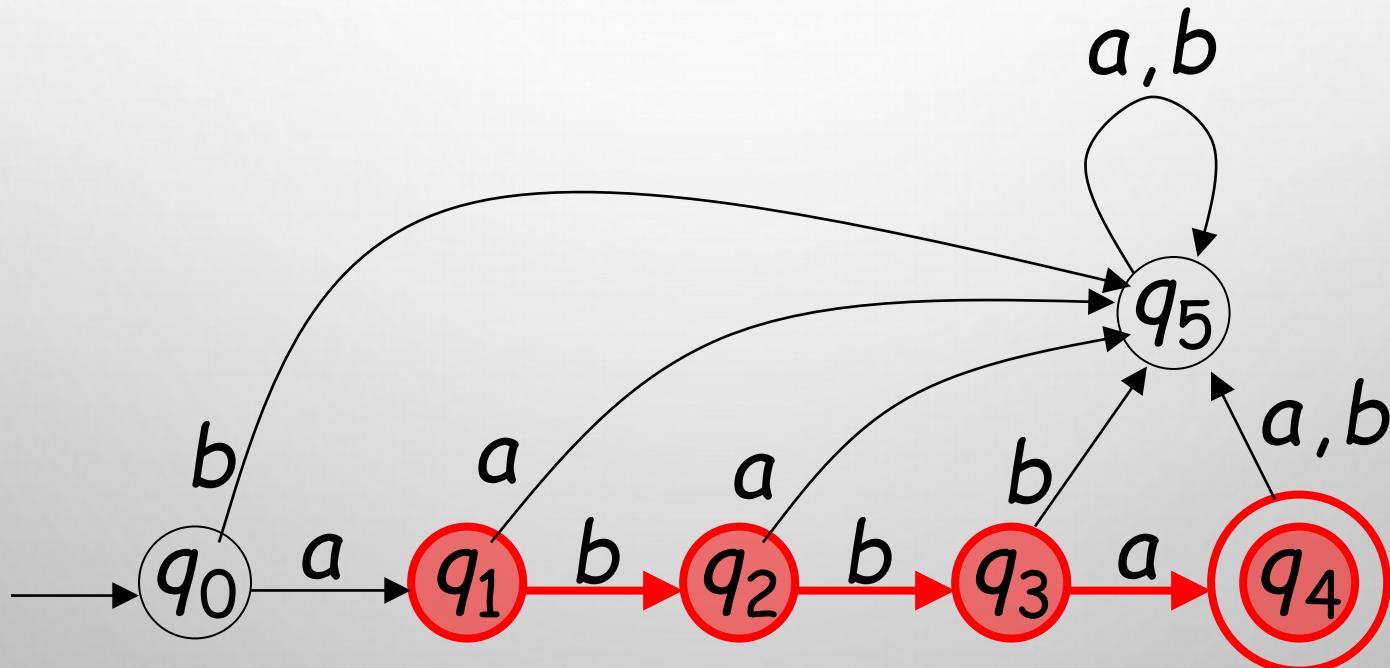


$$\delta^*(q_0, abbbbaa) = q_5$$



# ETF Example (**Caution(A/R)**)...

$$\delta^*(q_1, bba) = q_4$$



Does DFA accept or reject string bba? **NO**

# LANGUAGE OF A DFA

Let  $M = (Q, \Sigma, \delta, q_0, F)$  be a DFA.

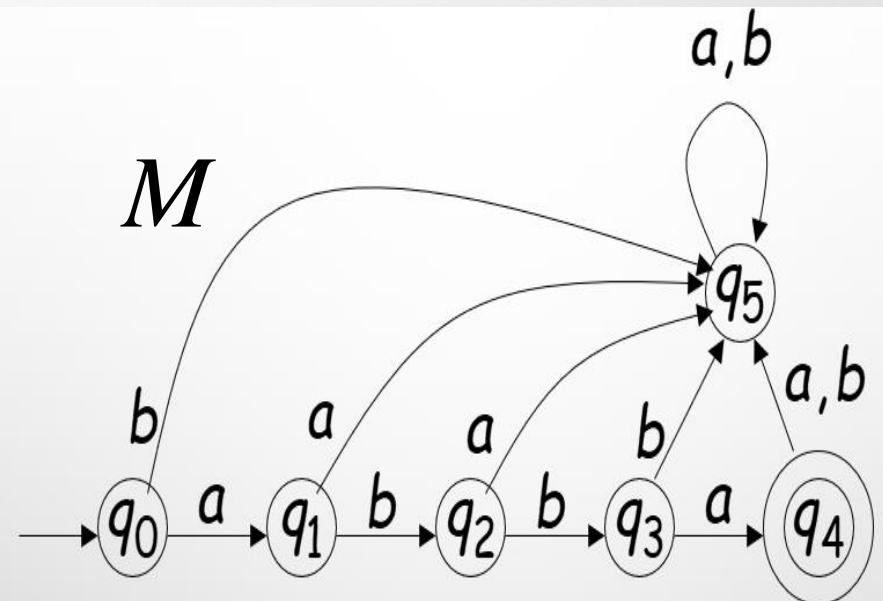
The language accepted by  $M$  is the set of all strings accepted by  $M$ .

$$L(M) = \{\omega \in \Sigma^* \mid (\mathbf{q}_0, \omega) \xrightarrow{*} (\mathbf{q}_f, \lambda) \text{ for some } \mathbf{q}_f \in F\}$$

or

$$L(M) = \{\omega \in \Sigma^* \mid \delta^*(\mathbf{q}_0, \omega) \in F\}$$

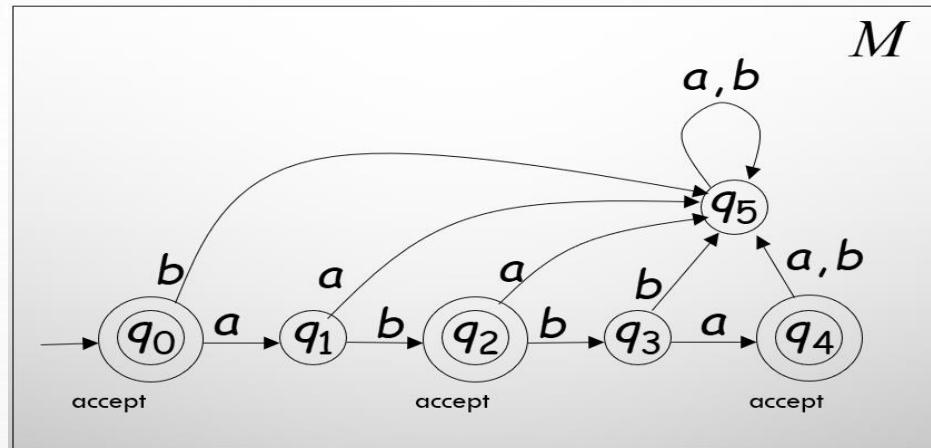
$$L(M) = \{abba\}$$



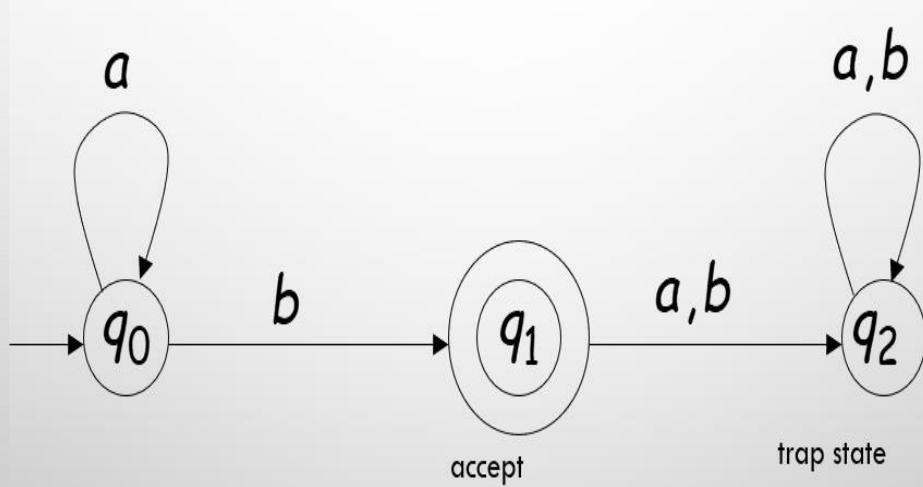
# LANGUAGE OF A DFA-EXAMPLES

$$L(M) = \{\lambda, ab, abba\}$$

**There is no transition with  $\lambda$  but it accepts  $\lambda$  as a string.**



$$L(M) = \{a^n b : n \geq 0\}$$



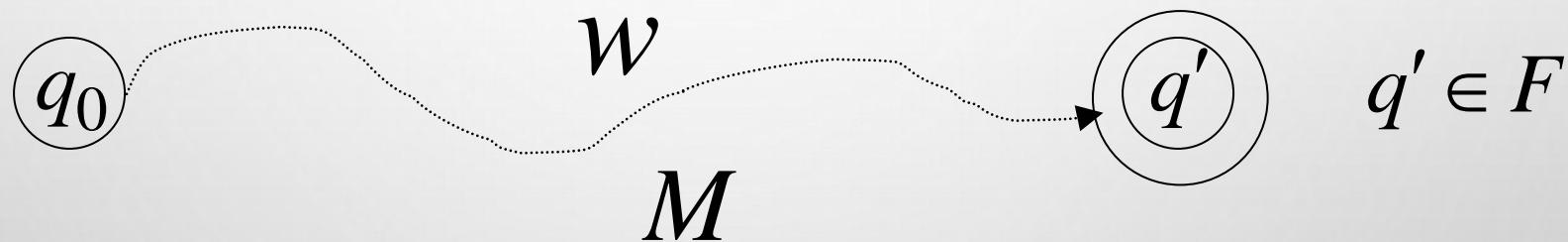
# OBSERVATION!!!

$$M = (Q, \Sigma, \delta, q_0, F) \quad L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$$

$L(M)$  is set of strings accepted by the DFA.

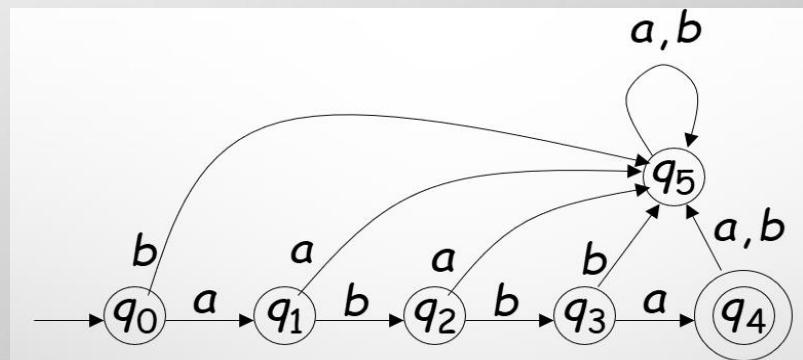
OR

$L(M)$  is the set of all strings  $w$  such that there exists a walk from  $q_0$  to  $q'$  with label  $w$ ,  $q' \in F$ .



**Observation:**  $q$  is a useful state if

- There exists a path from initial state  $q_0$  to state  $q$  and
- There exists a path from state  $q$  to  $q_f \in F$



$q_0, q_1, q_2, q_3$  and  $q_4$  are useful states.

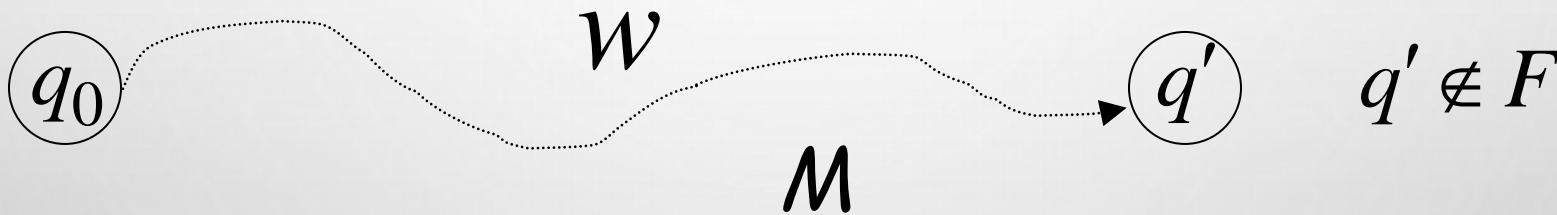
# OBSERVATION

LANGUAGE REJECTED BY DFA  $M$ :  $\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}$

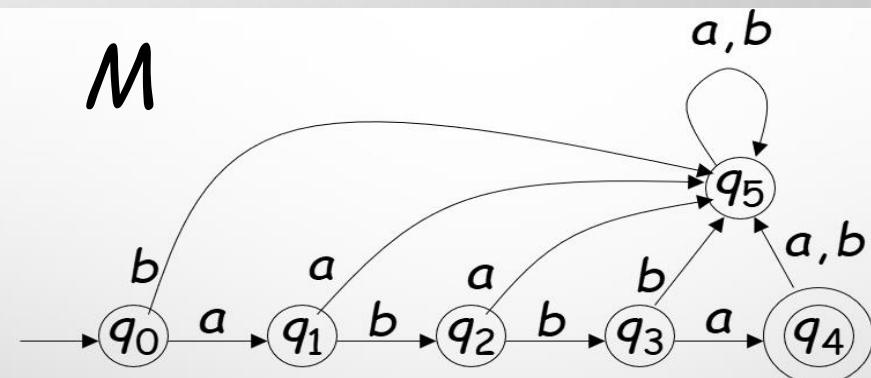
$\overline{L(M)}$  is set of strings rejected by the DFA.

OR

$\overline{L(M)}$  is the set of all strings  $w$  such that there exists a walk from  $q_0$  to  $q'$  with label  $w$  such that  $q' \notin F$ .



$\overline{L(M)}$  ?



# DESIGNING DFA

- DFA can be constructed for all finite languages but not necessarily for all infinite languages.
- Determine what a DFA need to memorize in order to recognize strings in the language.
  - Use the property of the strings in the language.
  - Find the regular pattern in the strings of the language.
- Determine how many states are required to memorize what we want.
  - Final state(s) memorizes the property of the strings in the language.
- Find out how the pattern DFA memorize is changed once the next input symbol is read.
  - From this change, we get the transition function.

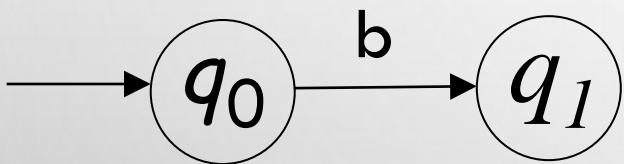
What about languages having “no common pattern” or “multiple pattern” ?

# Designing DFA

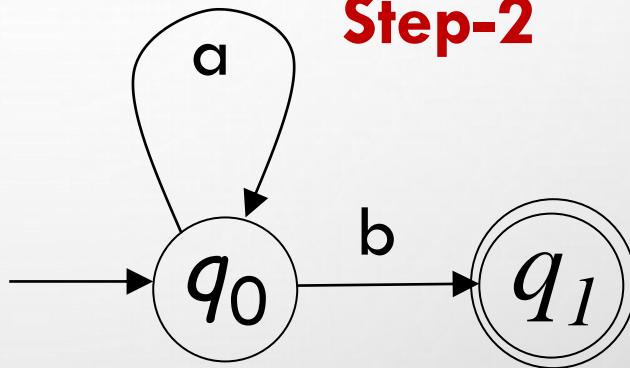
$L(M) = \{ \text{ all strings with any number of a followed by b } \}$

$$L(M) = \{a^n b : n \geq 0\} \quad \Sigma = \{a, b\}$$

**Step-1**



**Step-2**



**$\delta: Q \times \Sigma \rightarrow Q$  is the transition function of M**

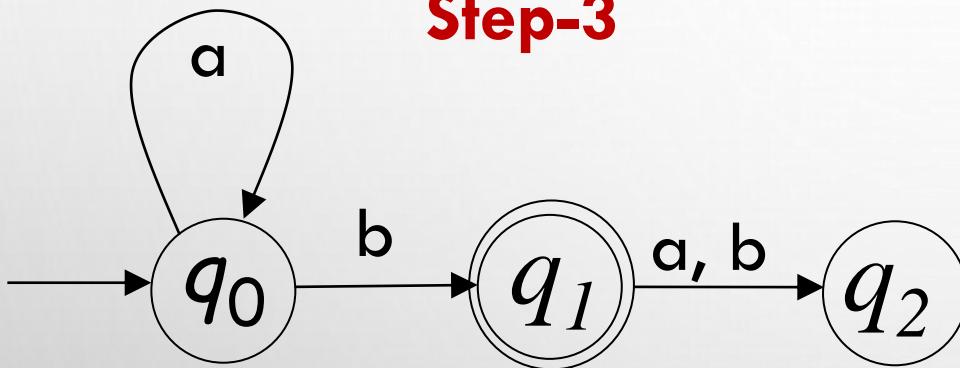
# Designing DFA

$L(M) = \{ \text{ all strings with any number of } a \text{ followed by } b \}$

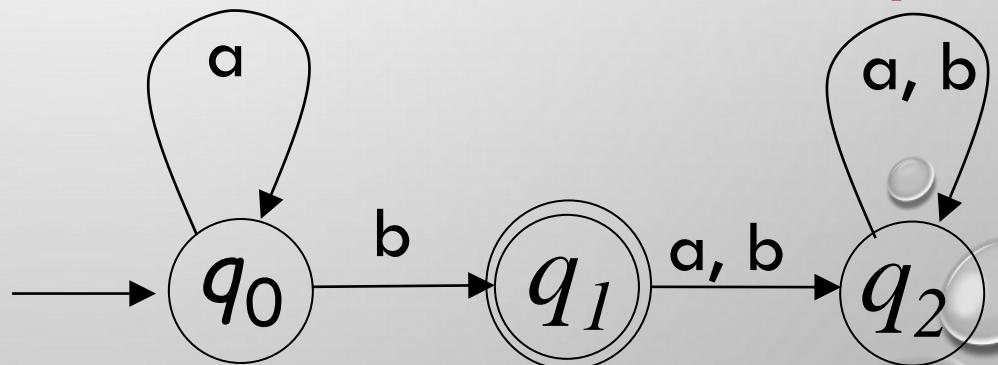
$$\Sigma = \{a, b\}$$

$$L(M) = \{a^n b : n \geq 0\}$$

**Step-3**



**Step-4**

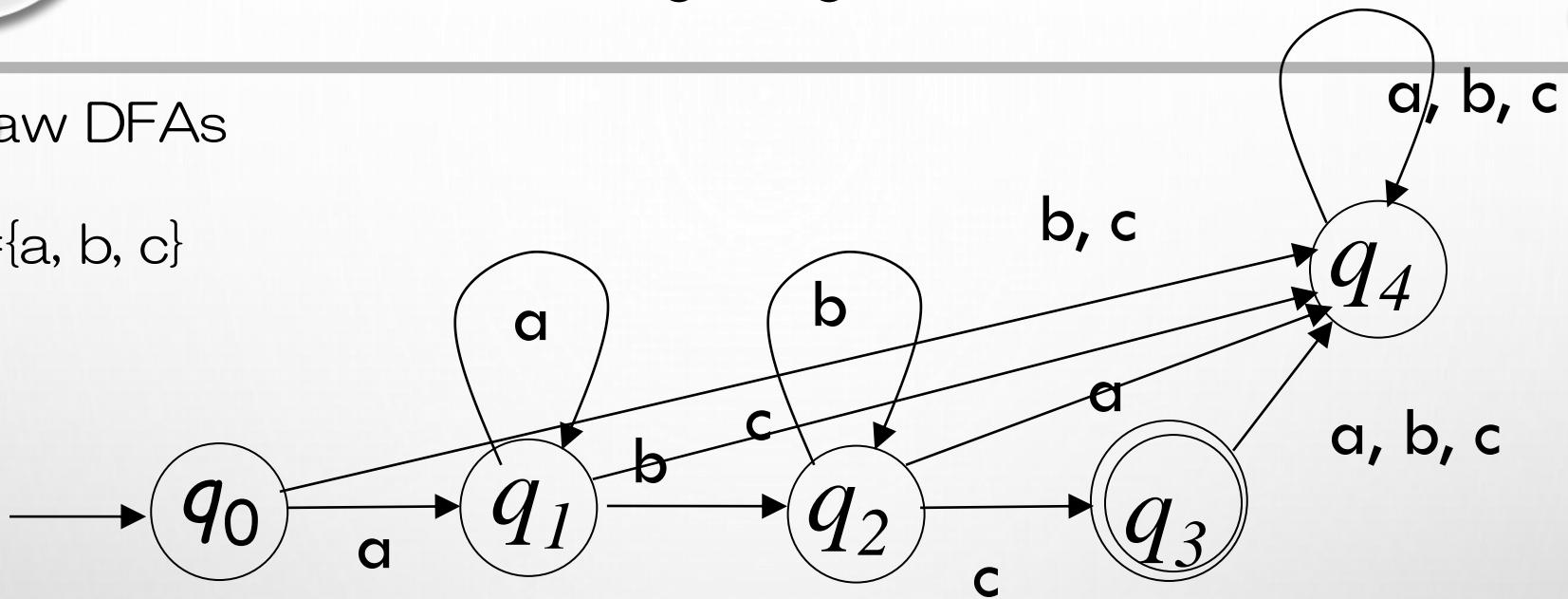


**$\delta: Q \times \Sigma \rightarrow Q$  is the transition function of M**

# Designing DFA

Draw DFAs

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



$$L1 = \{a^n b^m c : n \geq 1, m \geq 1\} = \{abc, aabbc, abbc, aabc, \}$$

$$L2 = \{a^n b^m c : n \geq 1, m \geq 0\}$$

$$L3 = \{a^n b^m c : n \geq 0, m \geq 1\}$$

$$L4 = \{a^n b^m c : n \geq 0, m \geq 0\}$$

# Designing DFA

- Draw DFAs,  $\Sigma = \{a, b, c, d\}$

$L_1 = \{a^l b^m c^n d : l \geq 1, n \geq 1, m \geq 1\}$

$L_2 = \{a^l b^m c^n d : l \geq 0, n \geq 0, m \geq 0\}$

$L_3 = \{a^l b^m c^n d : l \geq 0, n \geq 0, m \geq 1\}$

$L_4 = \{a^l b^m c^n d : l \geq 0, n \geq 1, m \geq 0\}$

$L_5 = \{a^l b^m c^n d : l \geq 0, n \geq 1, m \geq 1\}$

$L_6 = \{a^l b^m c^n d : l \geq 1, n \geq 0, m \geq 0\}$

$L_7 = \{a^l b^m c^n d : l \geq 1, n \geq 0, m \geq 1\}$

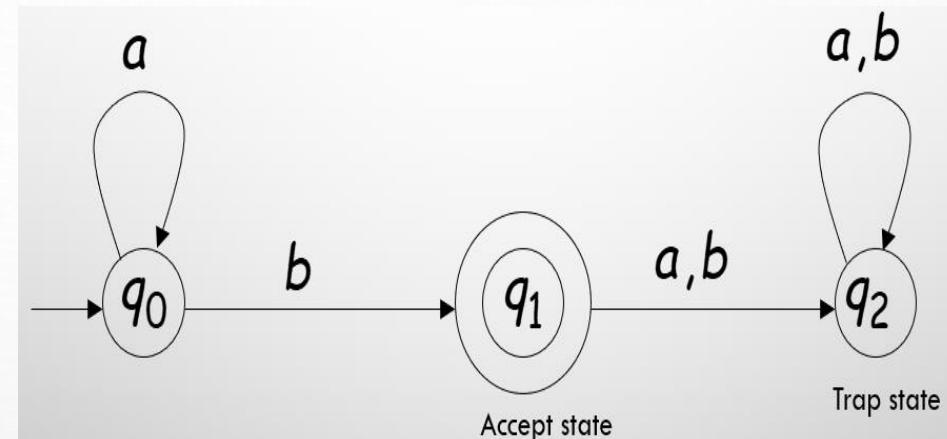
$L_8 = \{a^l b^m c^n d : l \geq 1, n \geq 1, m \geq 0\}$

# DESIGNING DFA

$L(M) = \{ \text{any number } a \text{ followed by single } b \}$

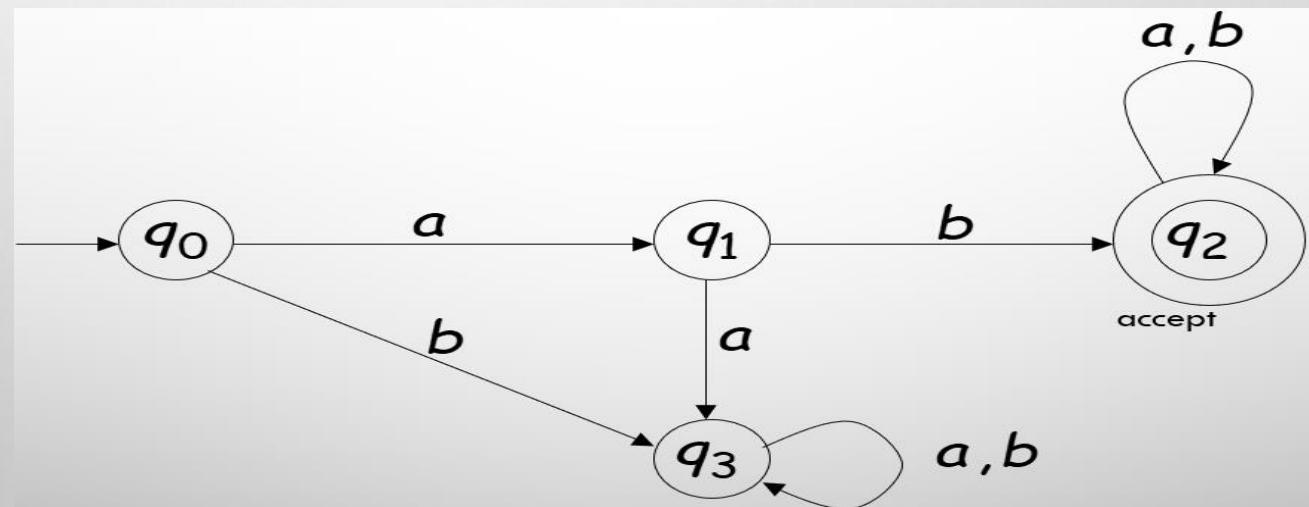
$$L(M) = \{a^n b : n \geq 0\}$$

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



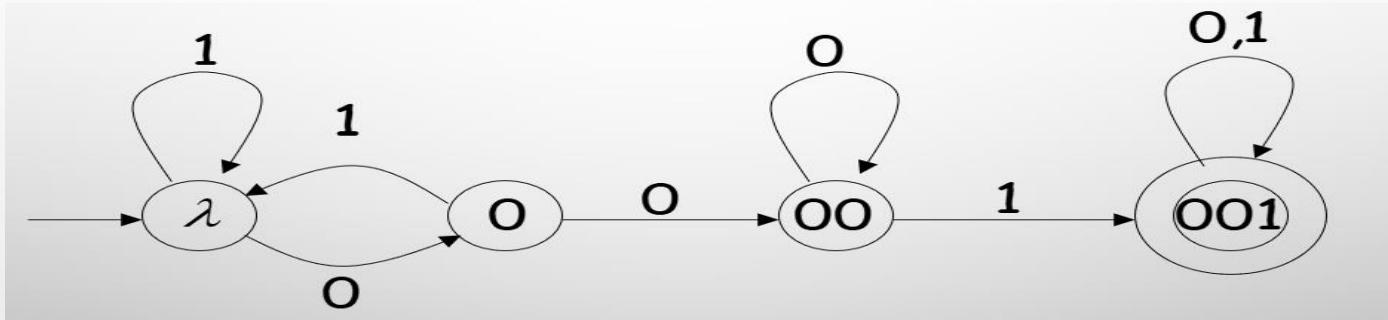
$L(M) = \{ \text{all strings with prefix ab} \}$

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

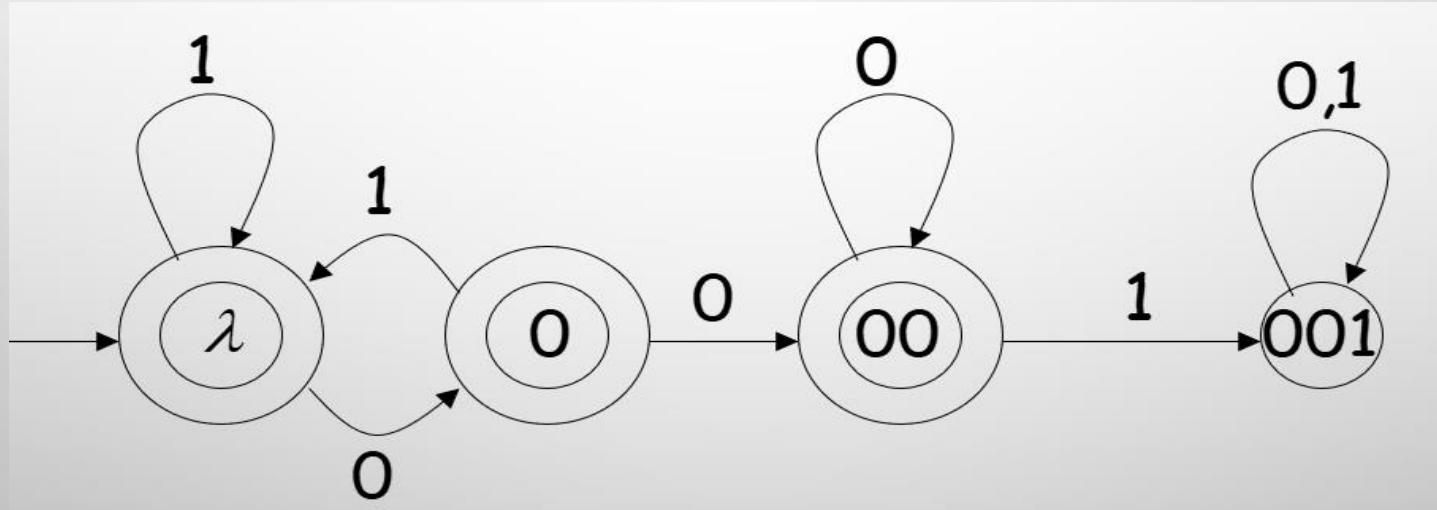


# Designing DFA

$\Sigma = \{0,1\}$ ,  $L_1 = \{ \text{ ALL STRINGS CONTAINING SUBSTRING } 001 \}$



$\Sigma = \{0,1\}$ ,  $L_2 = \{ \text{ all strings without substring } 001 \}$



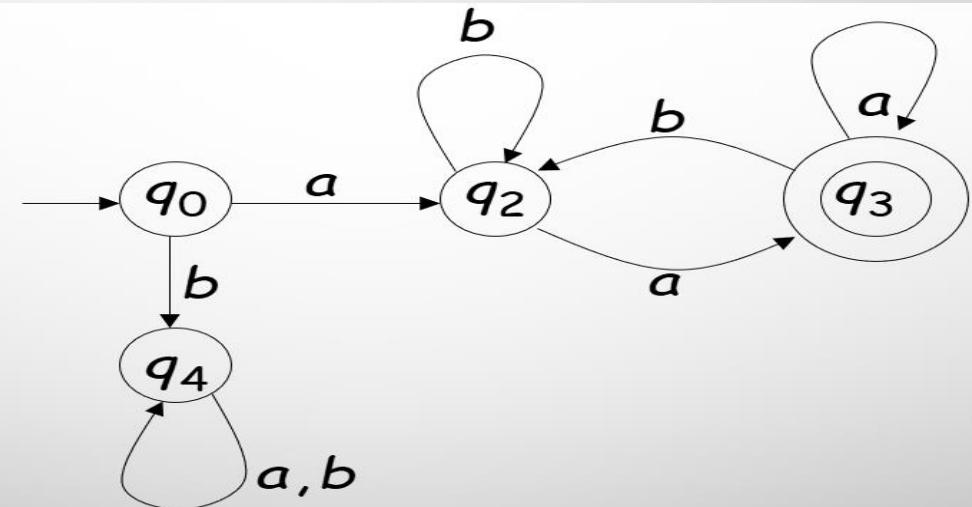
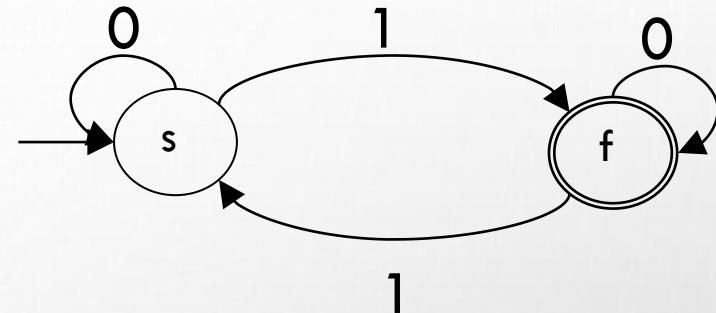
# DESIGNING DFA

$L(M) = \{x \in \{0,1\}^* \mid \text{The number of 1's in } x \text{ is odd}\} = \{1, 001, 11100, 111000, 010101\}$

S-even state

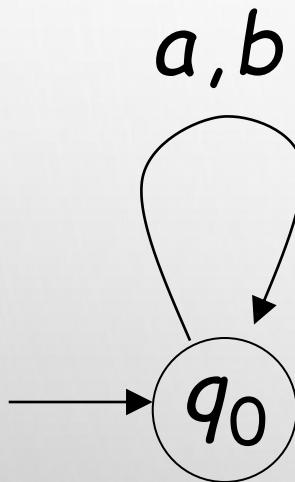
F-odd state

$L = \{awaw : w \in \{a,b\}^*\}$



# DFA: Typical Examples

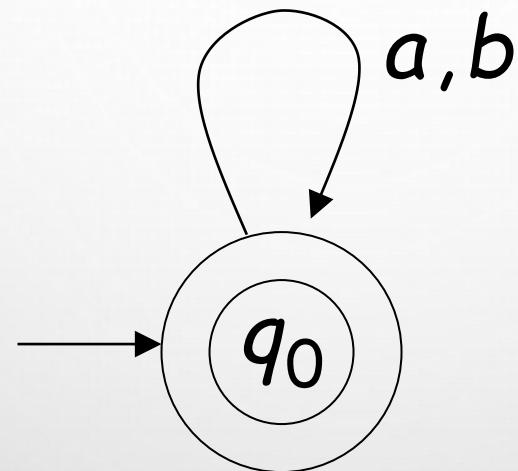
$$\Sigma = \{a, b\}$$



$$L(M) = \{ \}$$

Empty Language

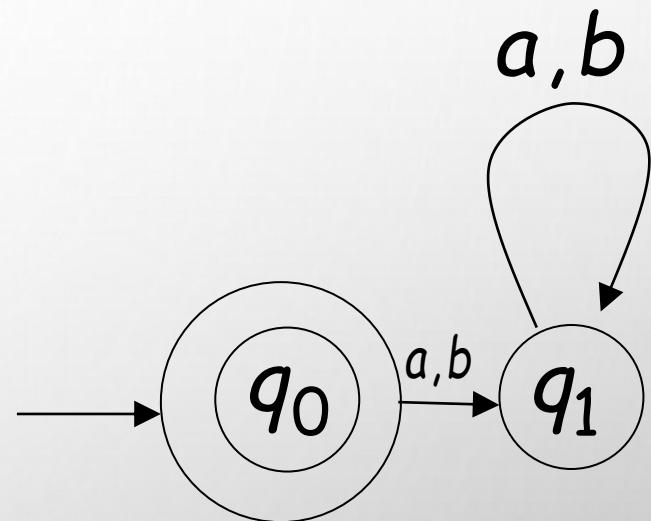
$$\Sigma = \{a, b\}$$



$$L(M) = \Sigma^*$$

All Strings

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



$$L(M) = \{\lambda\}$$

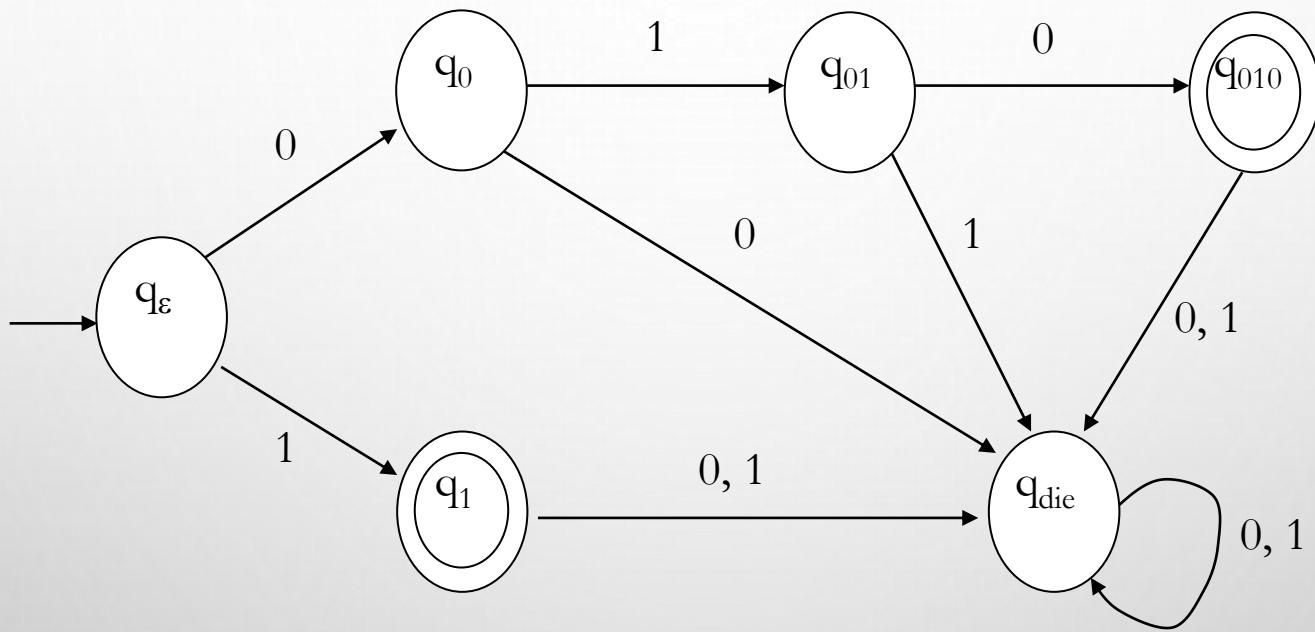
Language of the Empty String

# DESIGNING DFA....

- Construct a DFA that accepts the language

$$L = \{010, 1\}$$

$$(\Sigma = \{0, 1\})$$

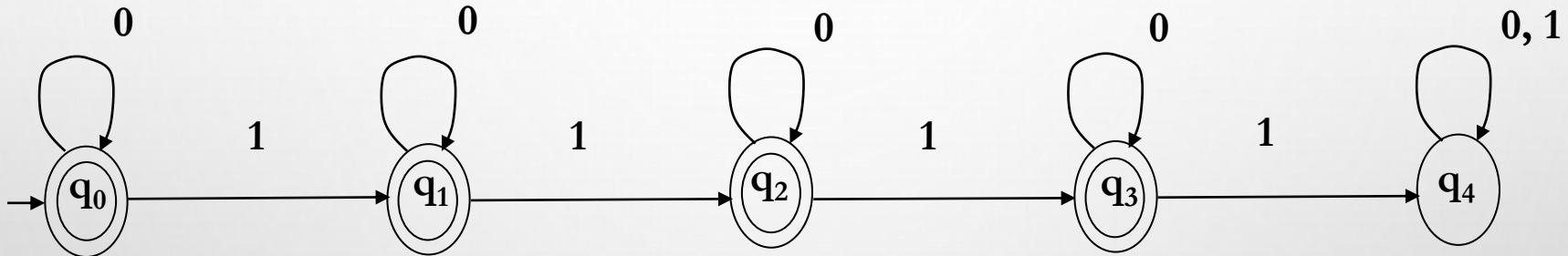


Note: DFA exists for all finite languages.

# DESIGNING DFA...

Construct a DFA over alphabet {0, 1} that accepts all strings with at most three 1s.

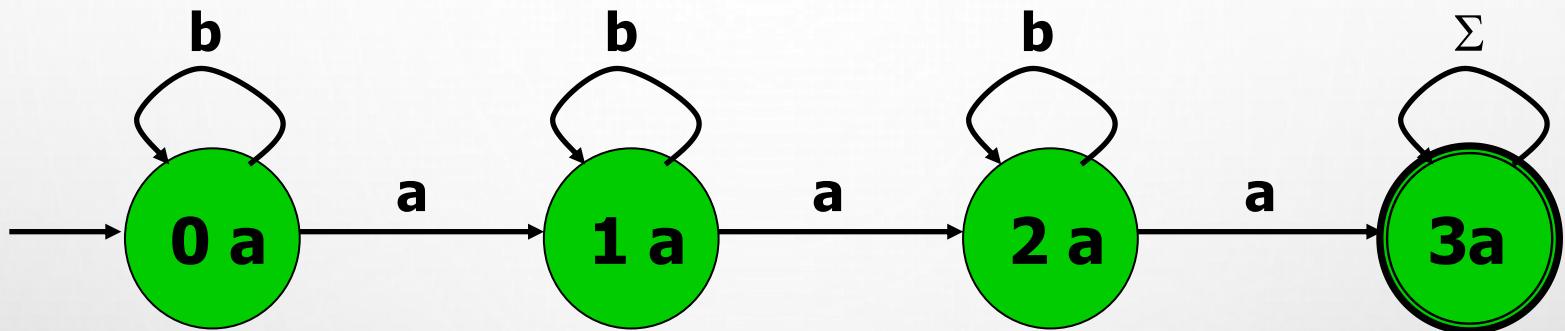
$$\{0^*\} \cup \{0^*10^*\} \cup \{0^*10^*10^*\} \cup \{0^*10^*10^*10^*\}$$



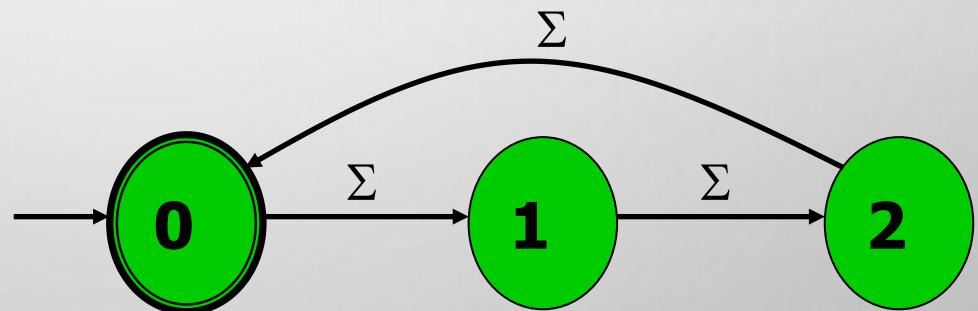
Construct a DFA over alphabet {0, 1} that accepts all strings with exactly three 1s.

# DESIGNING DFAS

Strings over  $\{a, b\}$  with at least 3 a's



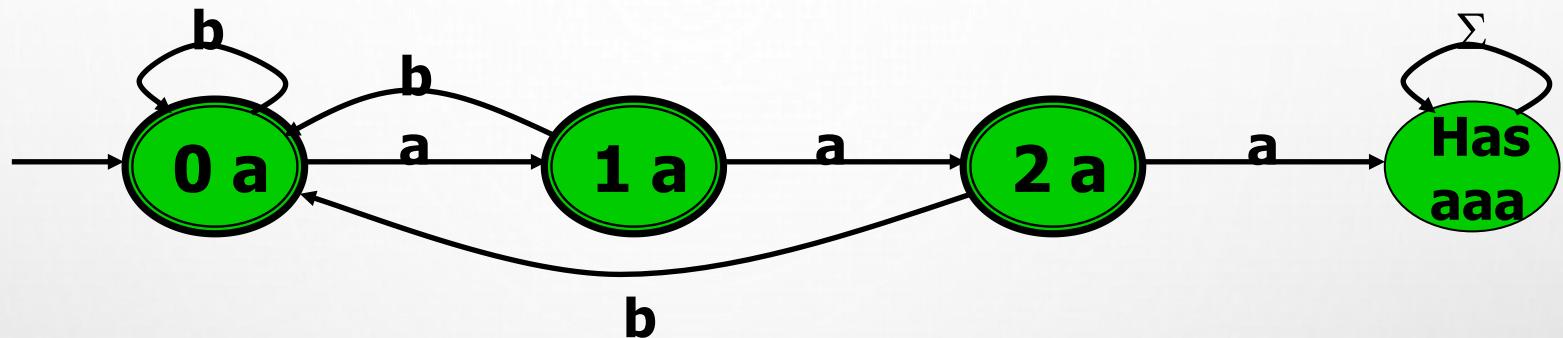
Strings over  $\{a, b\}$  with length mod 3 = 0



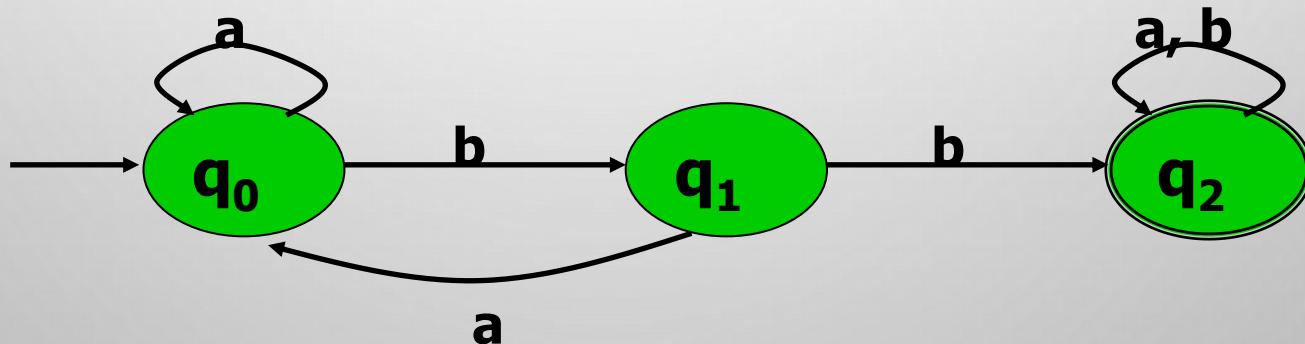
Strings over  $\{a, b\}$  with length not multiple of 3

# DESIGNING DFAS

Strings over  $\{a, b\}$  without 3 consecutive **a**'s

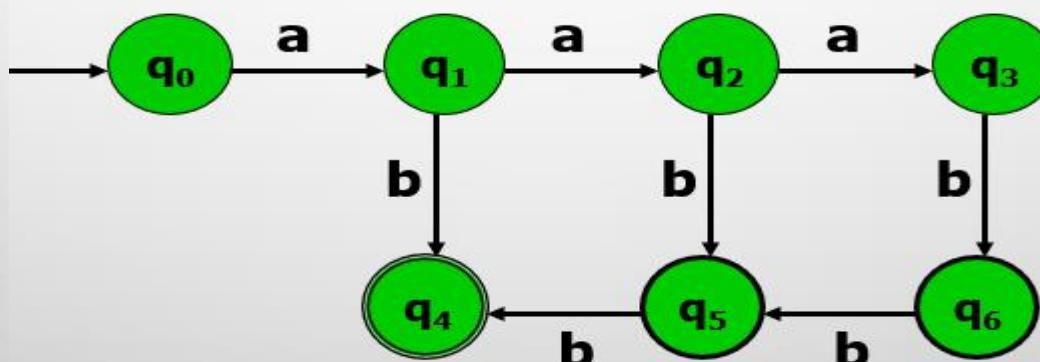


Strings over  $\{a, b\}$  that contain the substring **bb**



# DESIGNING DFAS

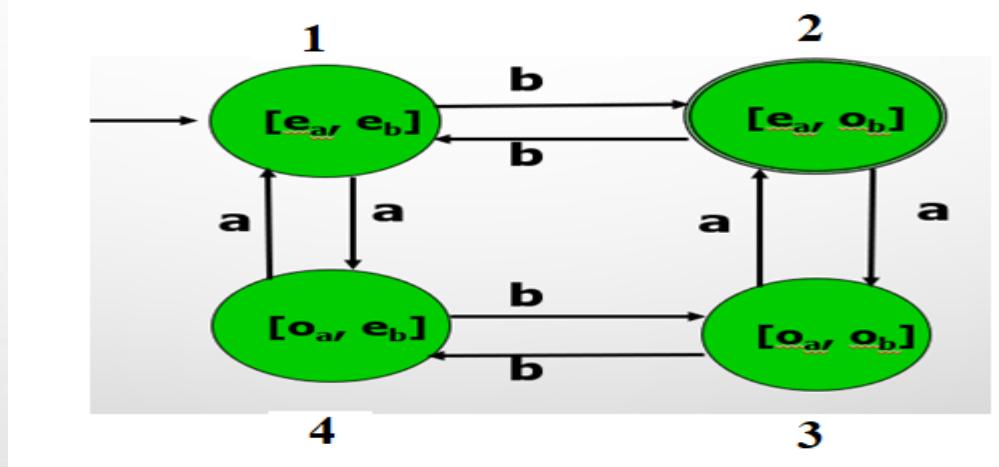
- We cannot construct DFA for the language  $L = \{a^n b^n : n \geq 0\}$ , because
  - It needs an infinite number of states
- But  $\{a^n b^n : 1 \leq n \leq 3\}$  is regular and its dfa is



This is NOT Complete DFA

# DESIGNING DFAS

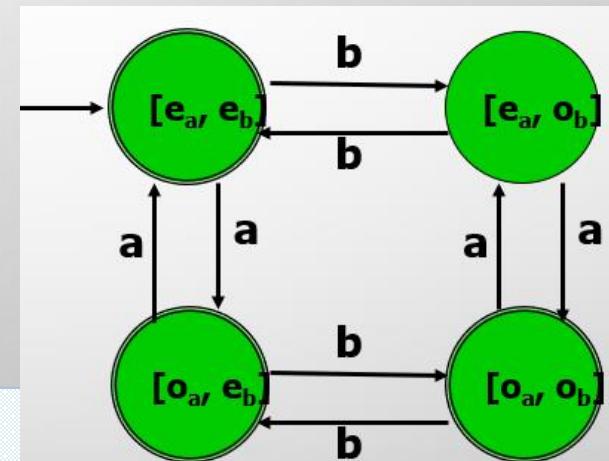
- Strings over  $\{a, b\}$  that contain an **even** number of **a's** and an **odd** number of **b's**



- A DFA  $M'$  that accepts all strings over  $\{a, b\}$  that **do not** contain an **even** number of **a's** **AND** an **odd** number of **b's** is shown below

$$L(M') = \{a, b\}^* - L(M) = \Sigma^* - L(M)$$

- Any string accepted by  $M$  is rejected by  $M'$  and vice versa

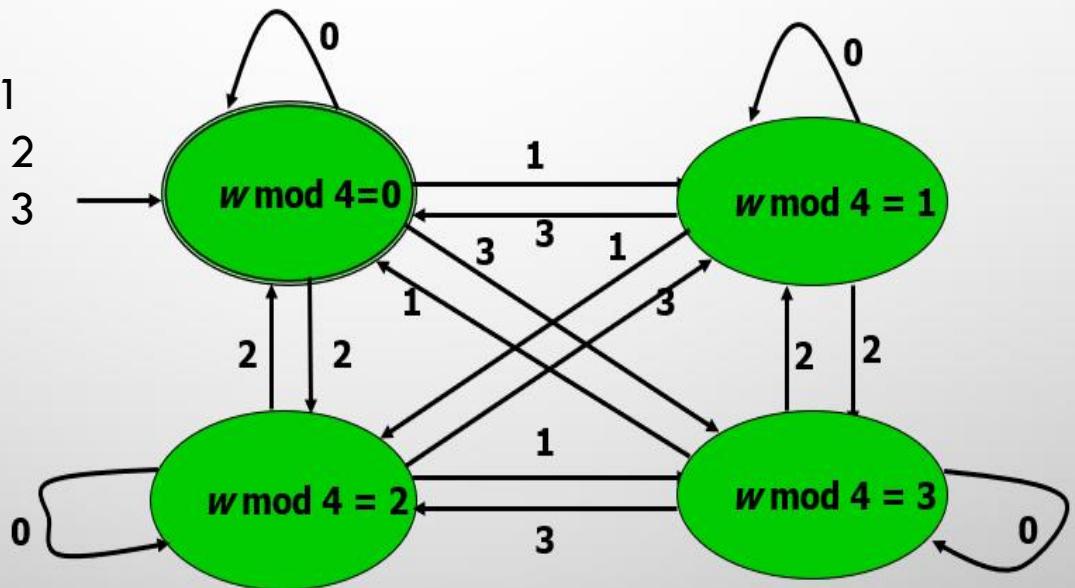


# Designing DFAs

- Let  $\Sigma = \{0, 1, 2, 3\}$ . A string in  $\Sigma^*$  is a sequence of integers from  $\Sigma$
- The DFA  $M$  such that **sum of elements** of a string is **divisible by 4**
- The string **12302** and **0130** should be accepted and **0111** rejected by  $M$

- multiple of 4  $\rightarrow w \bmod 4 = 0$
- multiple of 4 + 1  $\rightarrow w \bmod 4 = 1$
- multiple of 4 + 2  $\rightarrow w \bmod 4 = 2$
- multiple of 4 + 3  $\rightarrow w \bmod 4 = 3$

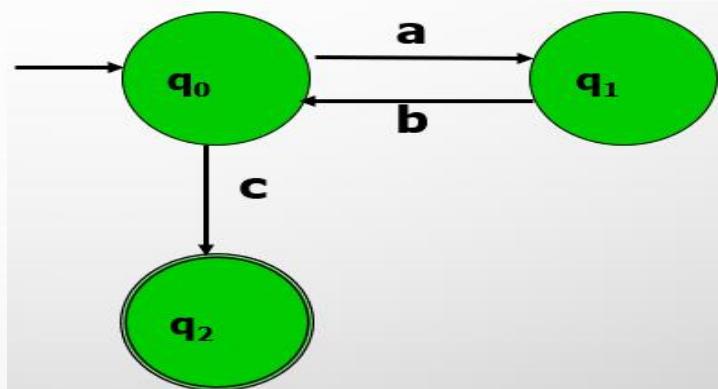
000002



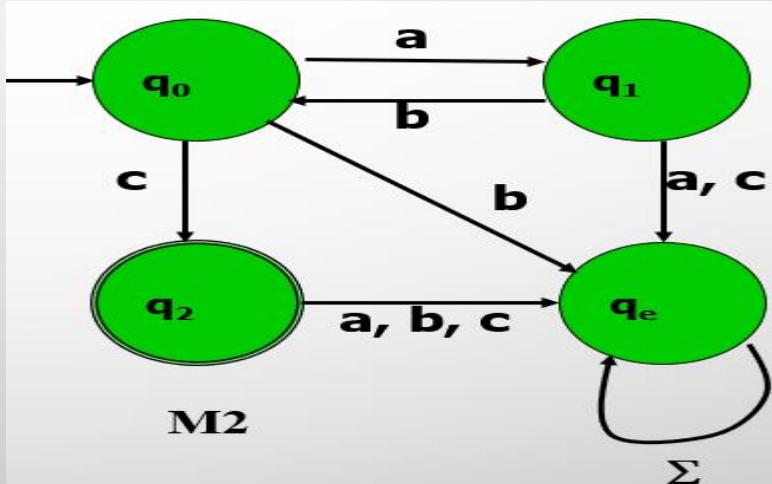
# Designing DFAs

- DFA **M1** accepts  $(ab)^*c$
- **M1** is *incomplete DFA*.
- The string **abcc** is *rejected* since **M1** is unable to process the final **c** from state **q<sub>2</sub>**

**M1**

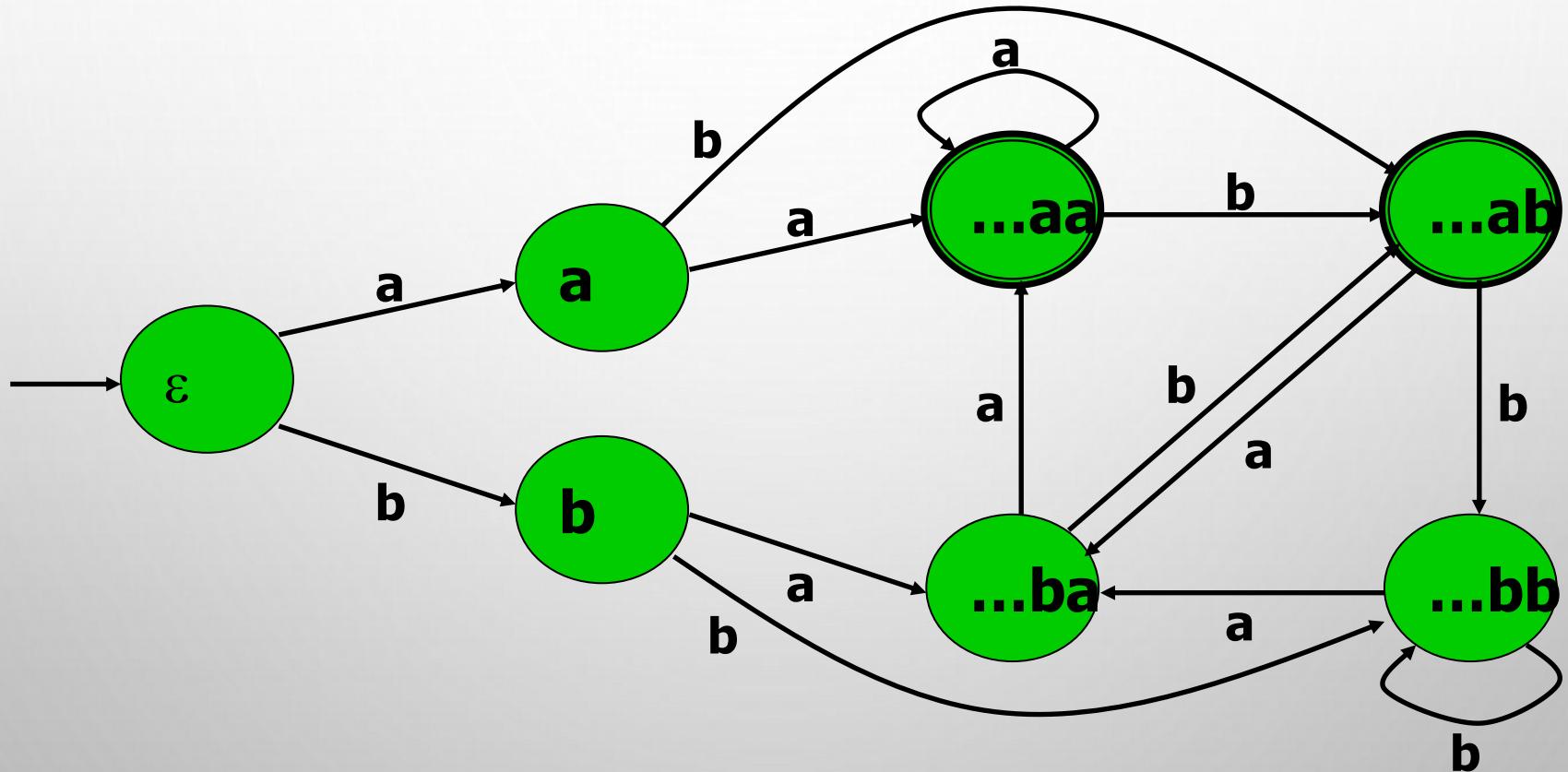


- **M2** accepts the same language as **M1**.
- **M2** is **complete DFA**.
- The state **q<sub>e</sub>** is the trap state.



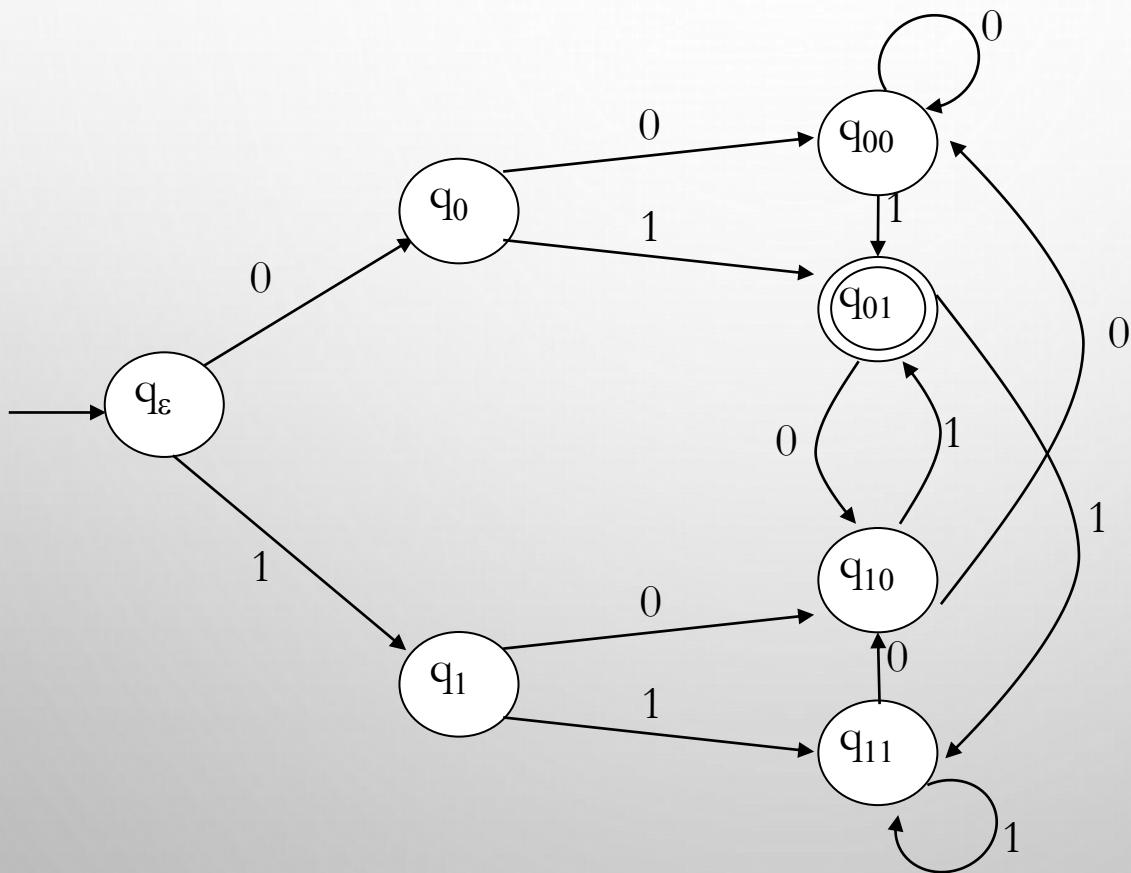
# DESIGNING DFAS

Strings over  $\{a, b\}$  with next-to-last symbol = a



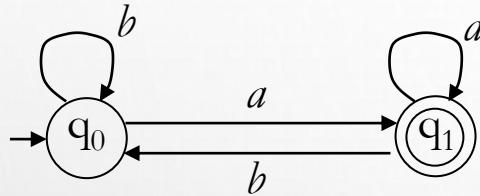
# DESIGNING DFA ...

- Construct a DFA over alphabet  $\{0, 1\}$  that accepts all strings that end in 01.

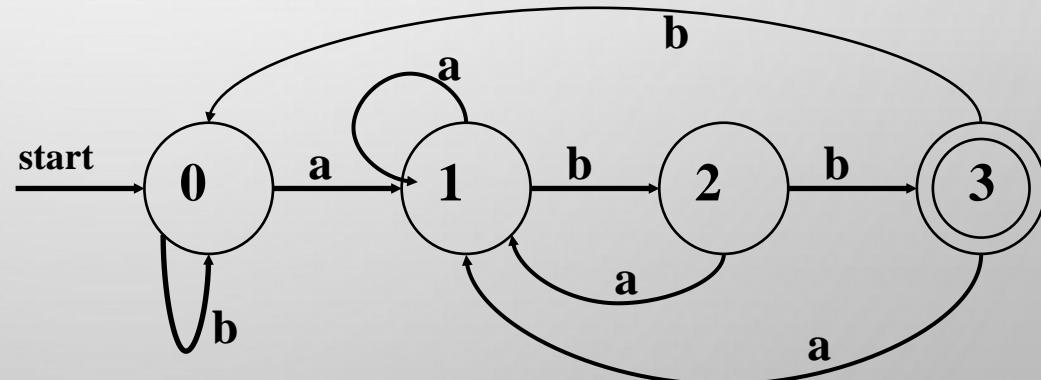
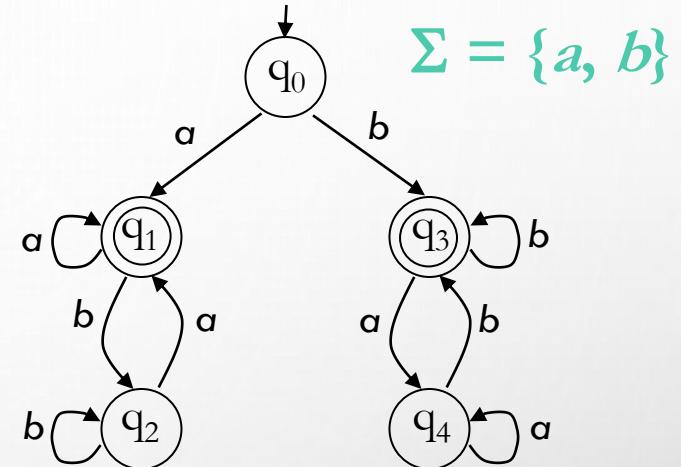
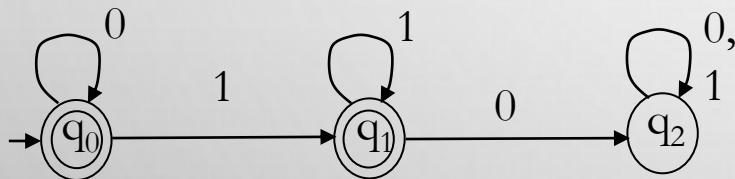


# Find Languages?

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



$$\Sigma = \{0, 1\}$$



# Design DFAs

- For  $\Sigma = \{0, 1\}$ , design DFAs for the following languages-
  - All strings ending in 00.
  - All strings with three consecutive 0's (not necessarily at the end).
  - All strings that contain exactly 4 "0"s.
  - All strings ending in "1101".
  - All strings containing exactly 4 "0"s and at least 2 "1"s.
  - All strings whose fourth symbol from the left end is a 1.
  - All strings that contain the substring 0101.
  - All strings that start with 0 and have an odd length or start with 1 and have an even length.
  - All strings that don't contain the substring 110.
  - All strings of length at most five.
  - All strings where every odd position is a 1.
  - All strings that contain an even number of 1's (Maximum 2 states).
  - All strings which do not contain the substring 10 (Maximum 3 states).
  - The set of strings that either begin or end (or both) with 01.
- For each of the following languages  $L_1, L_2, L_3$ , and  $L_4$  give a deterministic finite accepter (DFA). Consider the following languages over the alphabet  $\Sigma = \{a, b\}$ .
  - $L_1$ : All strings that contain at least three a's.
  - $L_2$ : All strings that contain at most one b.
  - $L_3$ : All strings that contain at least three a's but at most one b.
  - $L_4$ : All strings that contain no b's.

# **NONDETERMINISTIC FINITE ACCEPTER (NFA)**

# NONDETERMINISM

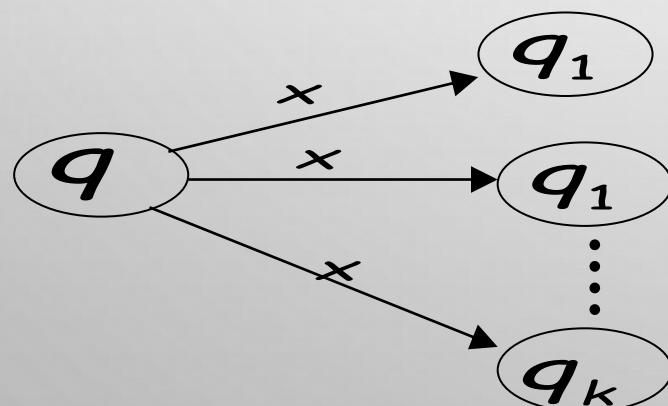
- For each input symbol and every state, the transition diagram may have
  - Zero or more transitions (None/one/more outgoing arrows)
- $\lambda$  is considered as one input symbol
  - $\lambda$ -transition allows moving from one state to another without consuming any input symbol  $a \in \Sigma$ .
- Non-determinism
  - Allows building automata in an easier way.
- Unlike a DFA M (where M can only be in one state at a time), an NFA can be in multiple states at one time.
  - It allows running processes simultaneously (parallelly).

# Formal definition of NFA

- A NFA is a quintuple  $(Q, \Sigma, \delta, q_0, F)$ , where:
  - $Q$  is a finite and non-empty set of internal states.
  - $\Sigma$  is a finite input alphabet.
  - $\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$  is the transition function.
  - $q_0 \in Q$  is the initial state.
  - $F \subseteq Q$  is the set of accepting states.

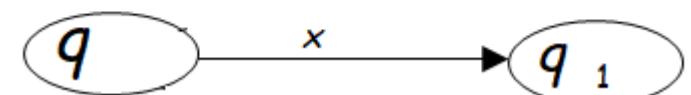
- *The input of  $\delta$  is an ordered pair of state and input symbol or  $\lambda$ .*
- *The output values of  $\delta$  may be a set of states.*

$$\delta(q, x) = \{q_1, q_2, \dots, q_k\}$$



DFA       $\delta : Q \times \Sigma \rightarrow Q$

$$\delta(q, x) = q_1$$

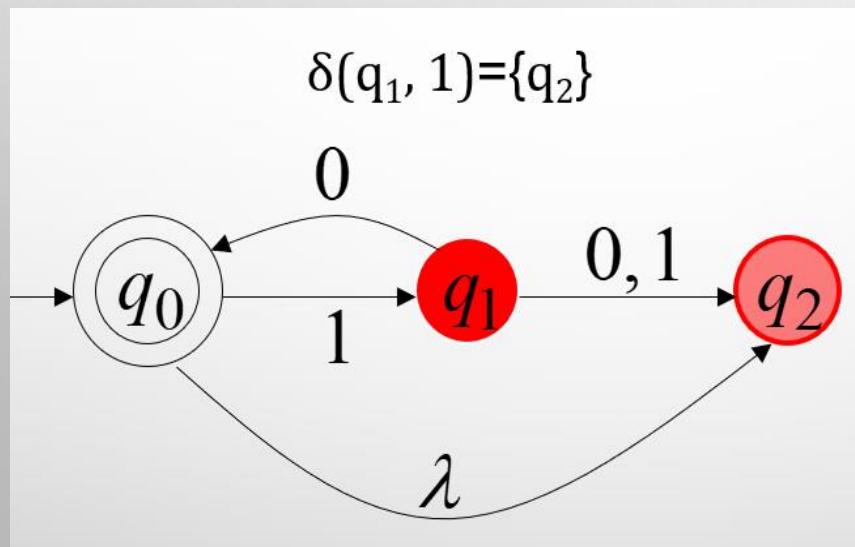
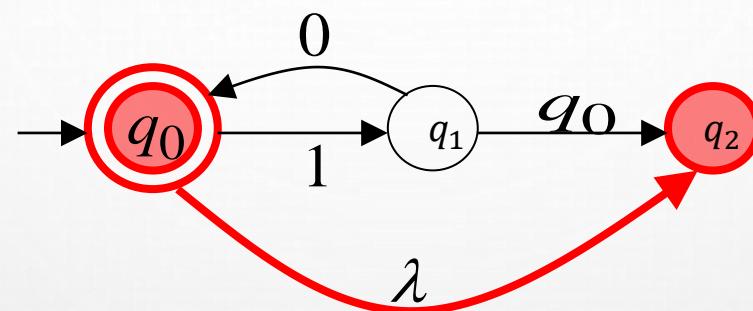
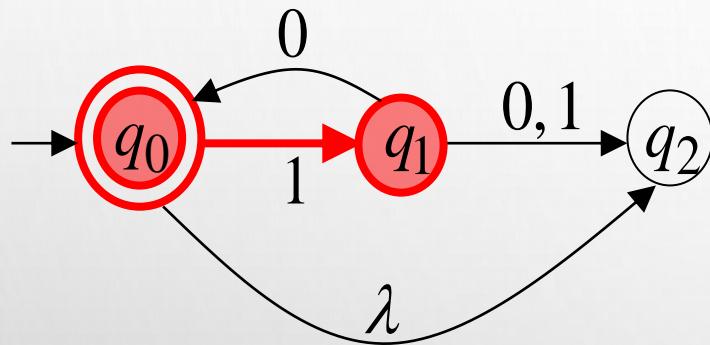


# NFA-Transition Function

$$\delta(q_0, 1) = \{q_1\}$$

$$\delta(q_0, \lambda) = \{q_0, q_2\}$$

$$\delta(q_0, 0) = \{ \}$$



$\delta(q_1, 1) = ??$   
 $\delta(q_1, \lambda) = ???$

$\delta(q_2, 1) = ??$   
 $\delta(q_2, \lambda) = ???$

# NFA

$$\delta(q_0, 1) = \{q_1\}$$

$$\delta(q_0, \lambda) = \{q_0, q_2\}$$

$$\delta(q_1, 0) = \{q_0, q_2\}$$

$$\delta(q_1, 1) = \{q_2\}$$

Transition Function

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$$

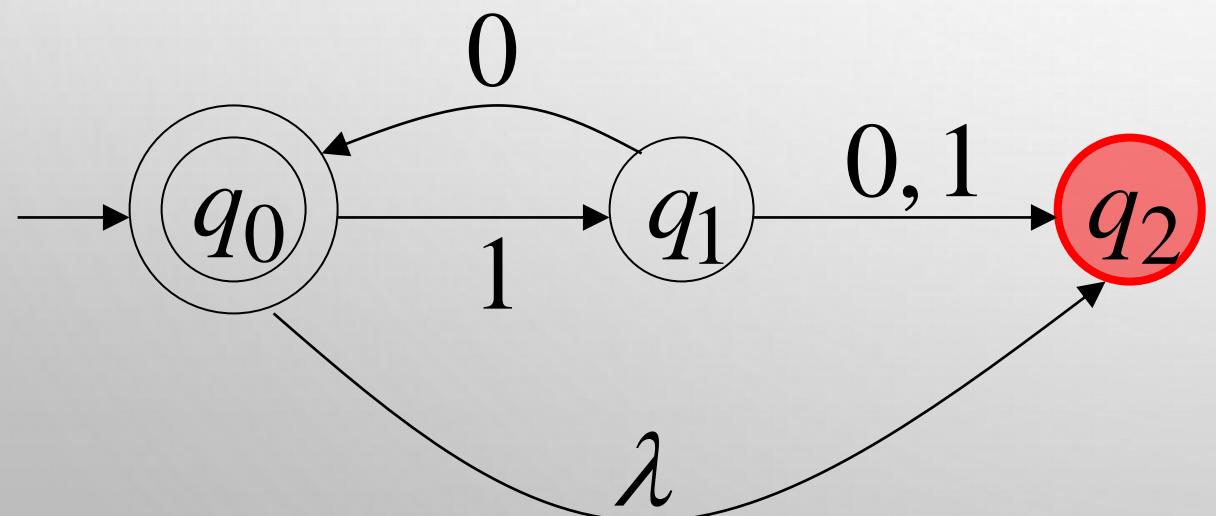
$$M = (Q, \Sigma, \delta, q_0, F)$$

$$Q = \{q_0, q_1, q_2\}$$

$$\Sigma = \{0, 1\}$$

$$q_0$$

$$F = \{q_0\}$$

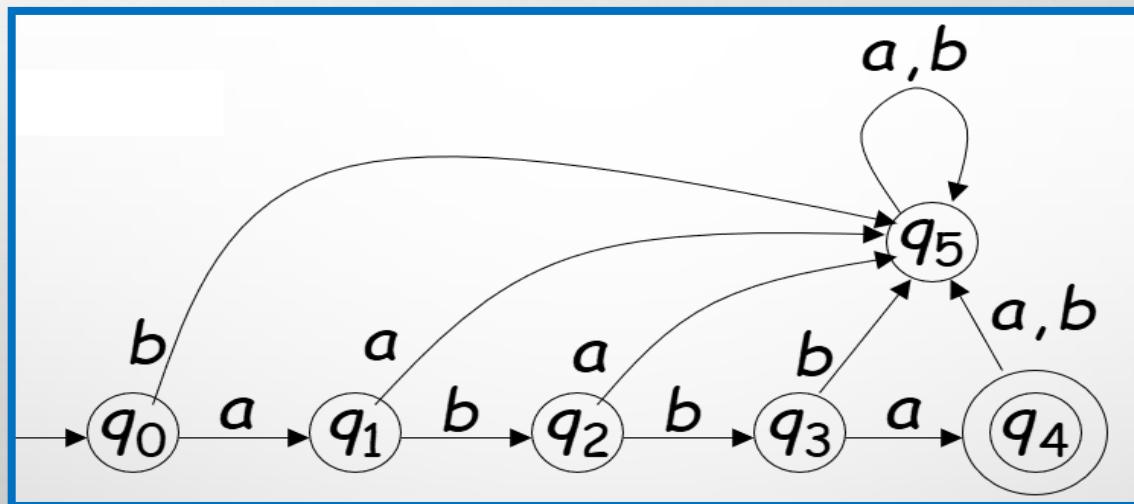
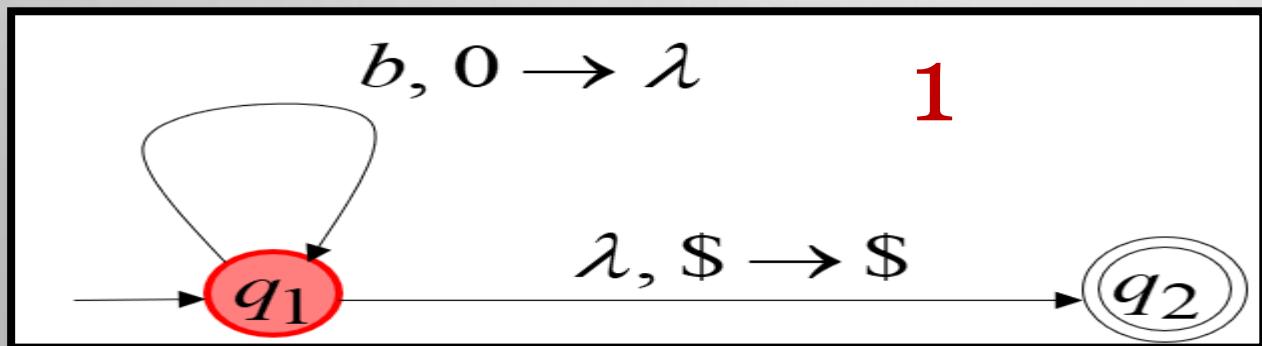
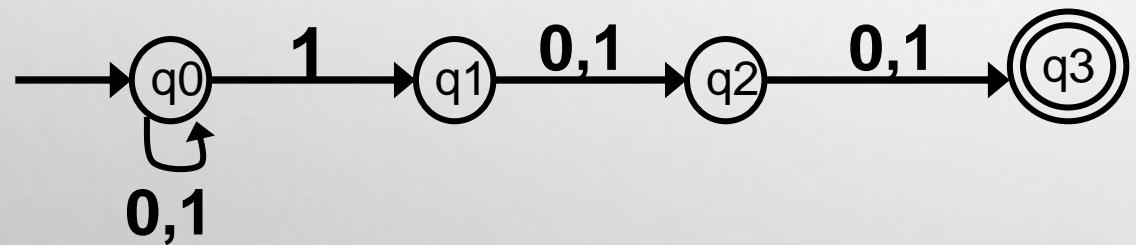
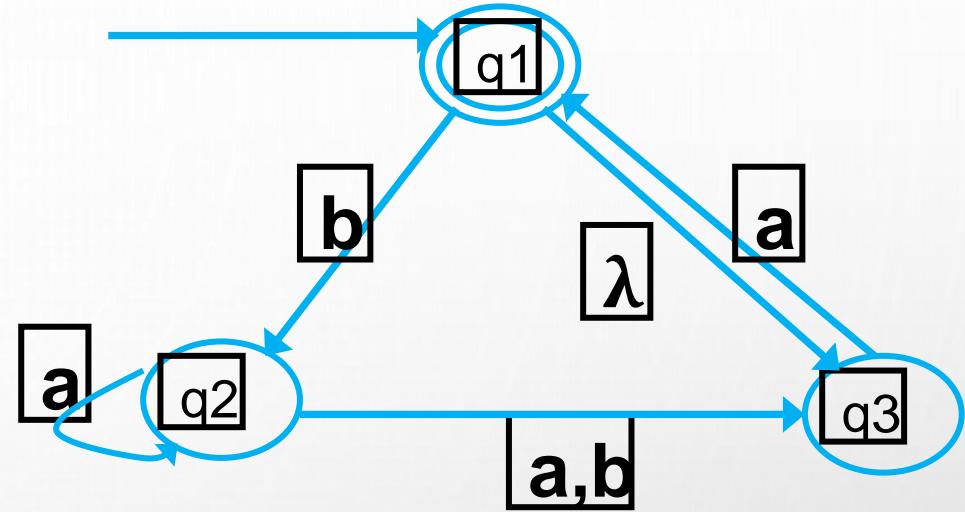
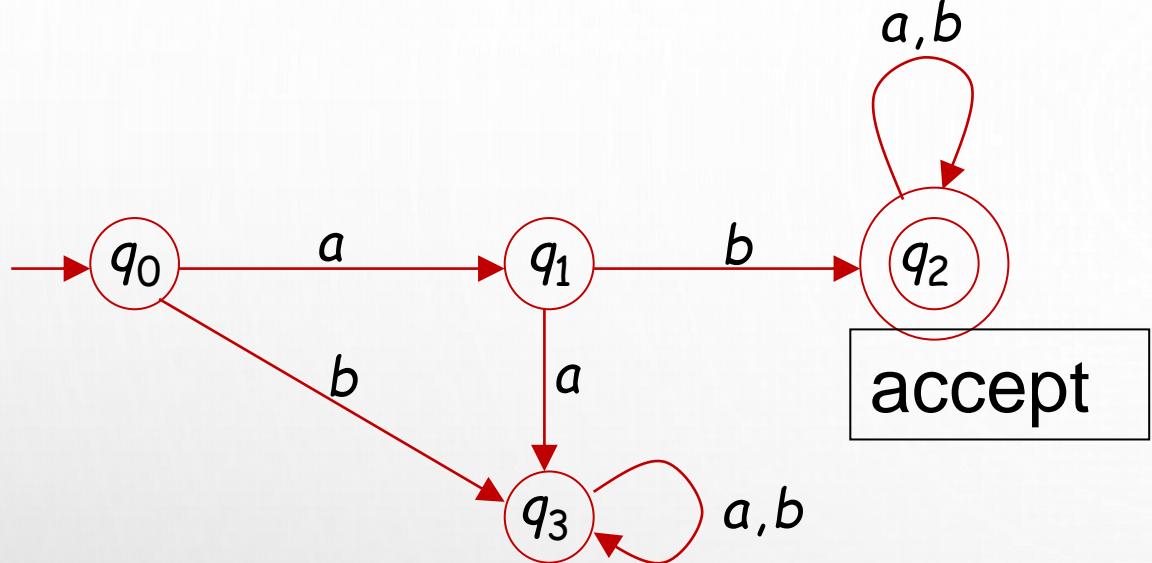


Transition Diagram

	0	1	$\lambda$
$q_0$	$\{\}$	$\{q_1\}$	$\{q_0, q_2\}$
$q_1$	$\{q_0, q_2\}$	$\{q_2\}$	$\{q_1\}$
$q_2$	$\{\}$	$\{\}$	$\{q_2\}$

Transition Table

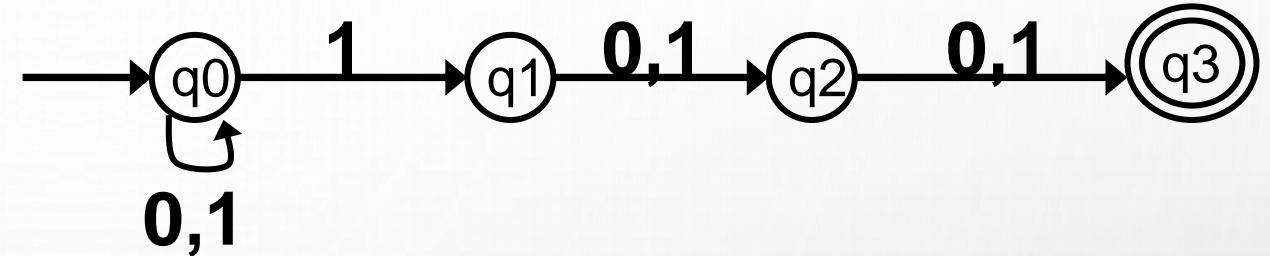
# IDENTIFY DFA/NFA ?



# NFA CONSTRUCTION- EASY

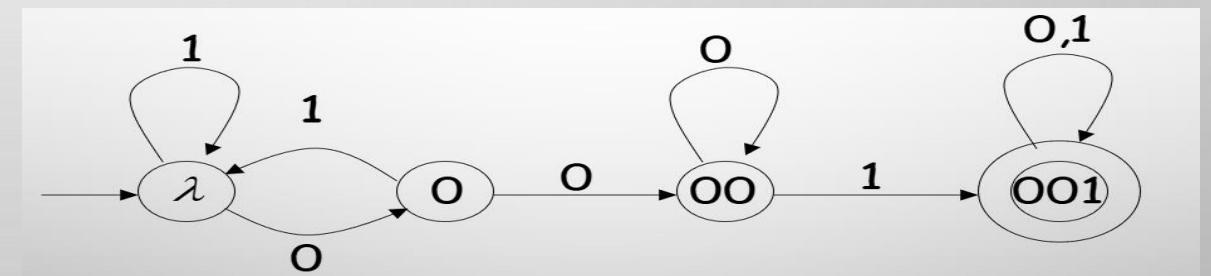
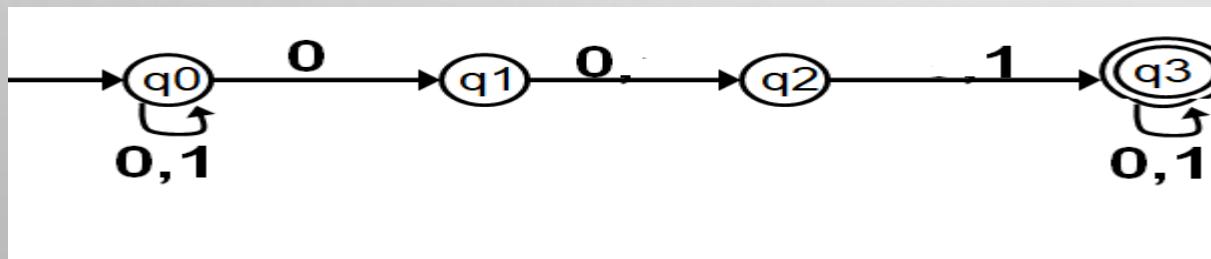
- Accept all strings with at least length three that contains a 1 in the third from the end  
 $\Sigma=\{0,1\}$ ,

The form of the strings -  $w_1 \mathbf{1} w_2 w_3$ ,  
 $w_1 \in \{0, 1\}^*$ ,  $w_2 \in \{0, 1\}$ ,  $w_3 \in \{0, 1\}$



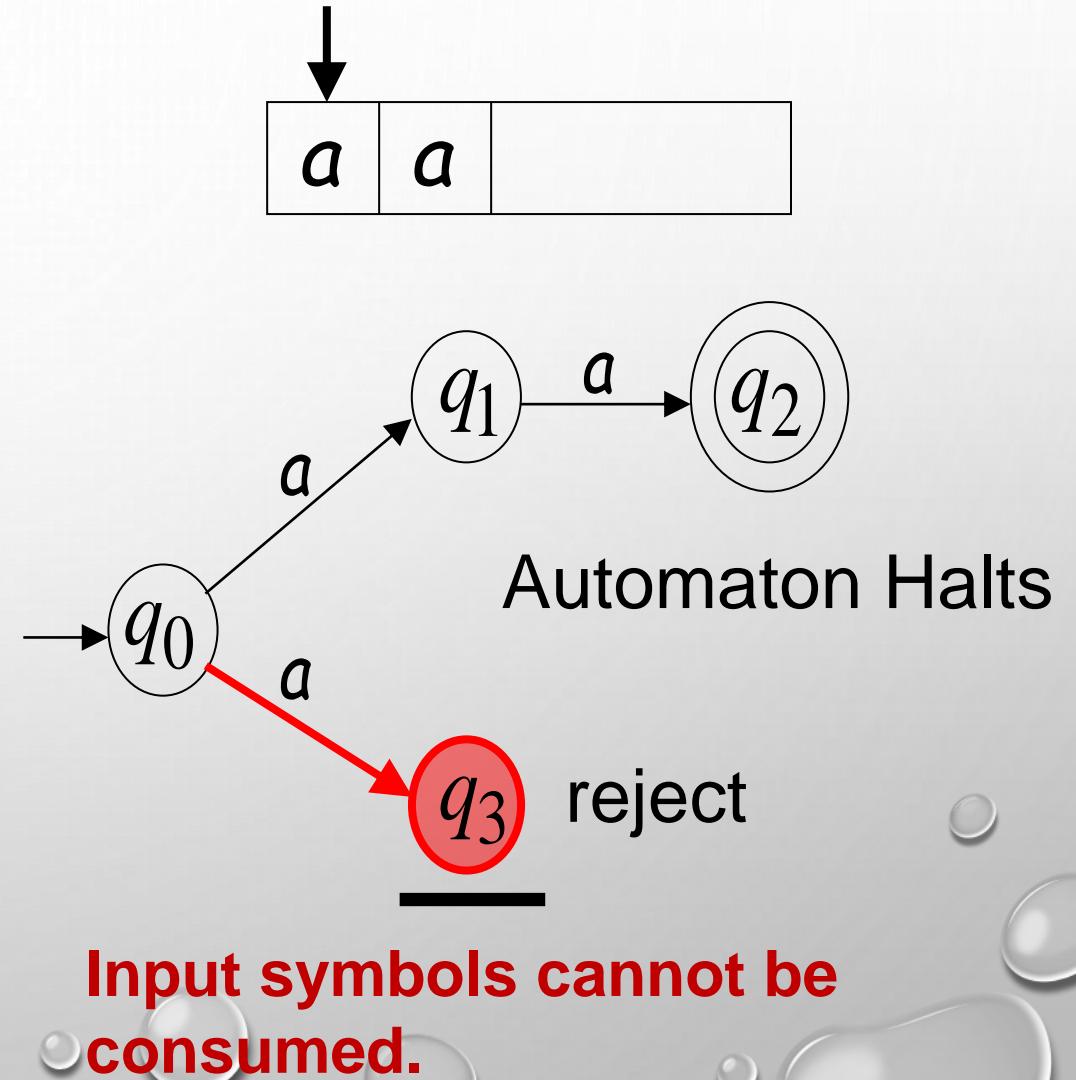
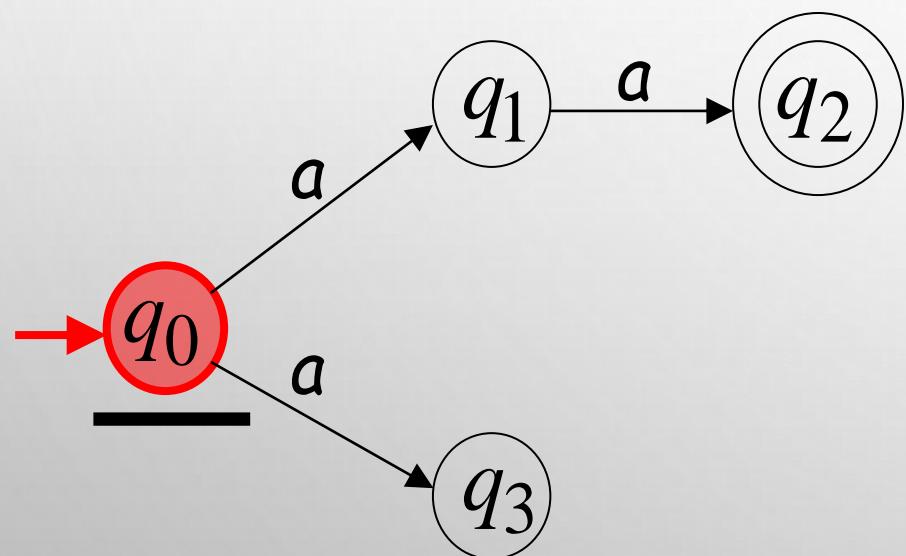
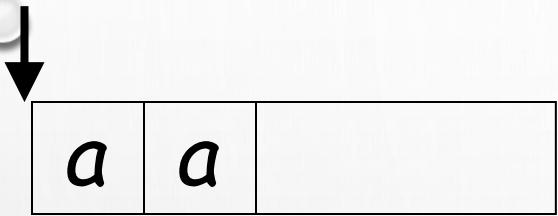
- What makes this difficult for a DFA?

- Accept all strings containing substring 001 = {00001, 000100, 00010100...}
- $\Sigma=\{0,1\}$ ,



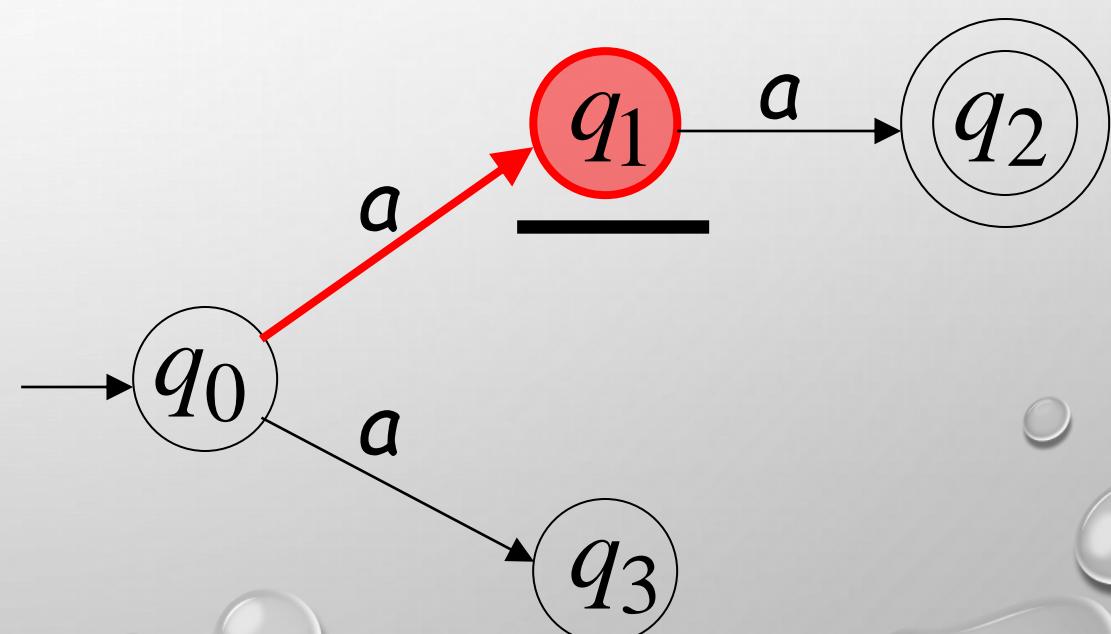
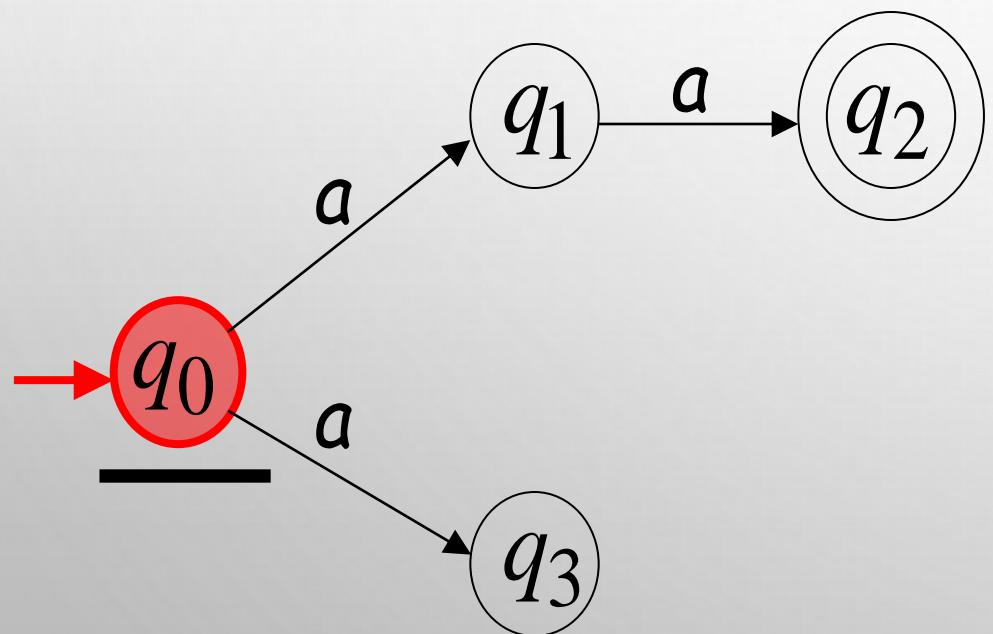
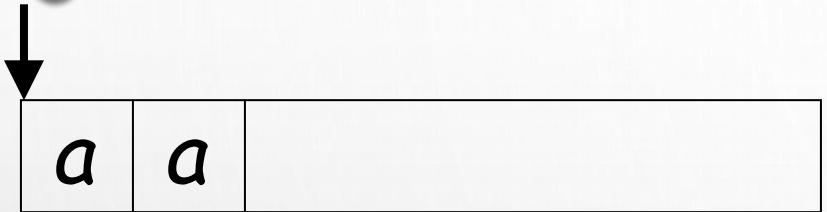
# STRING ACCEPTANCE/REJECTION BY NFA

## First Choice



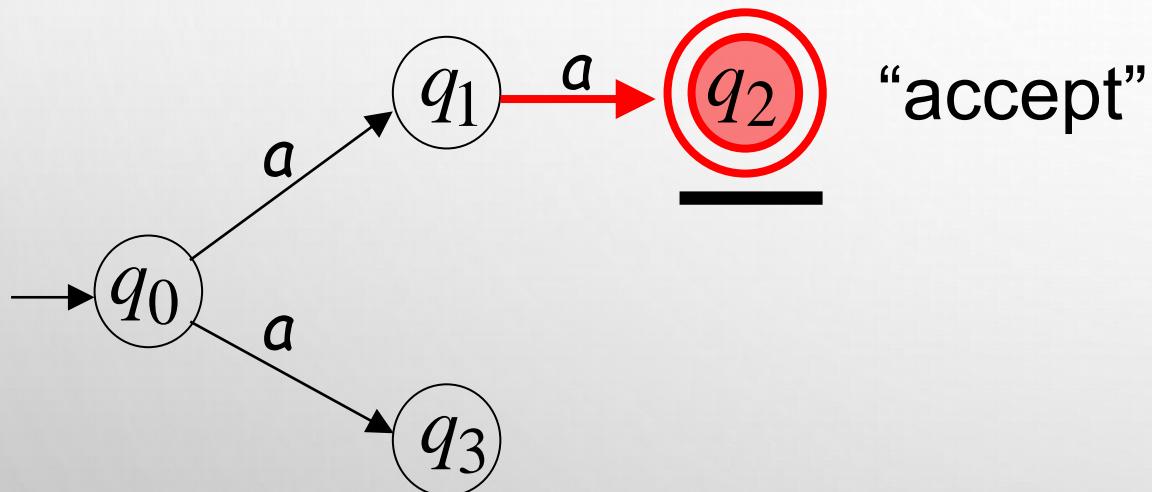
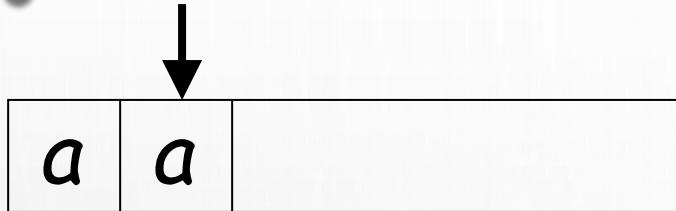
# STRING ACCEPTANCE/REJECTION BY NFA

## Second Choice



# STRING ACCEPTANCE/REJECTION BY NFA

## Second Choice



“accept”

All input symbols are consumed.

# String Acceptance/Rejection by NFA

## **Accept**

- If there is a **computation/path** of the NFA with label w which leads to **accepting state** -  
i.e.

**The automaton starts from the initial state.**

**and**

**All the input symbols of string are processed/read  
and**

**The automaton is in an accepting state.**

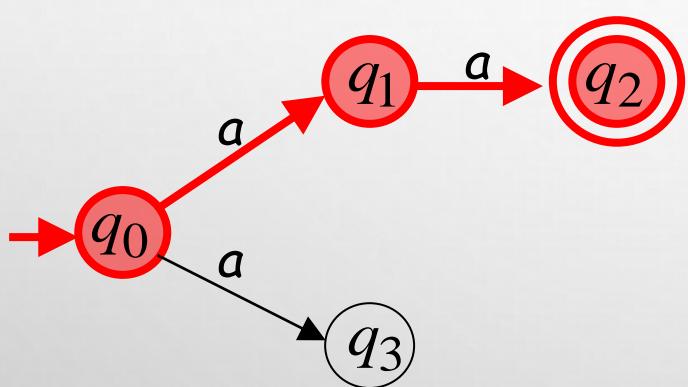
## **Reject**

- If there is **no computation/path** of the NFA with label w which leads to accepting state.

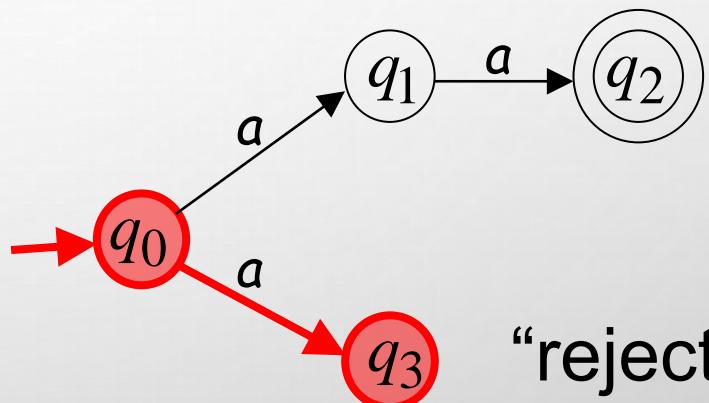
# String Acceptance by NFA

$aa$  is accepted by the NFA:

“accept”

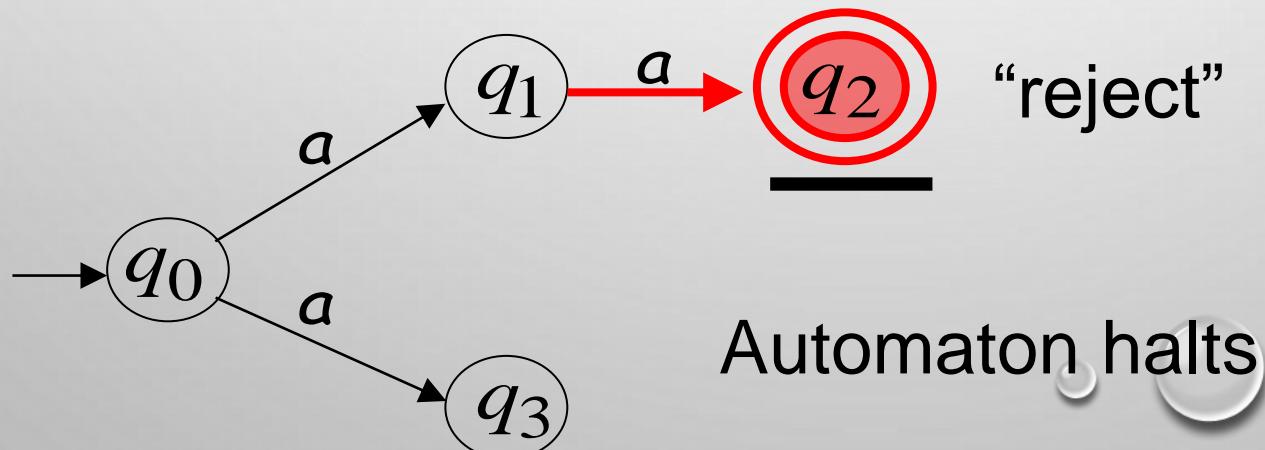
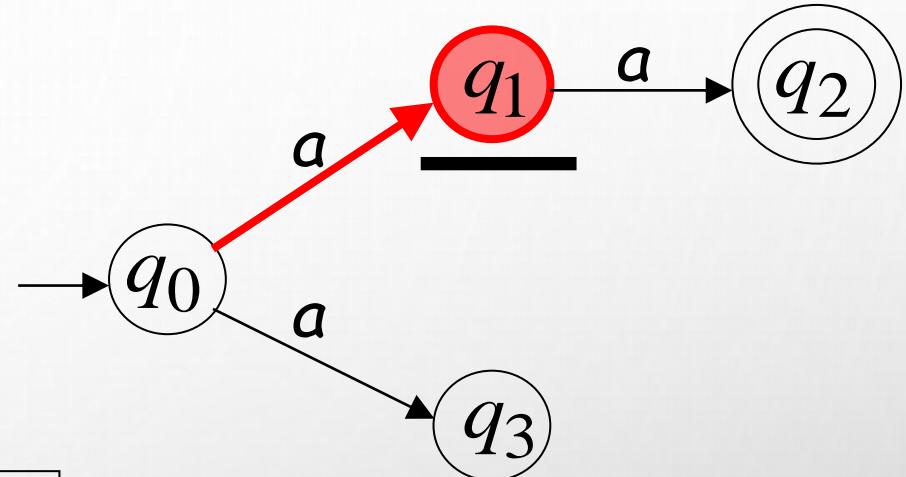
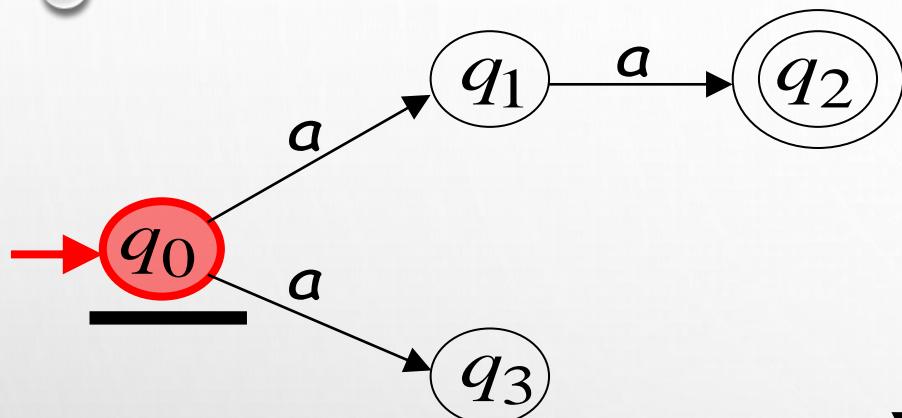


Because this computation accepts  $aa$ .



This computation is ignored.

# String Rejection by NFA: First Choice

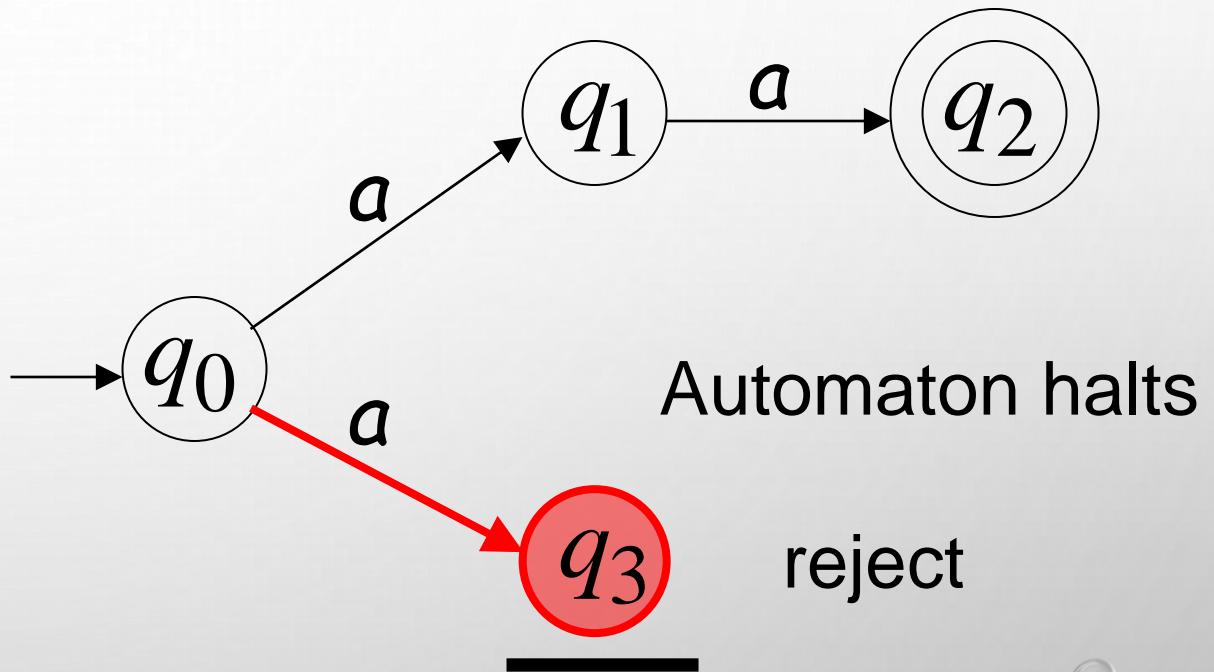
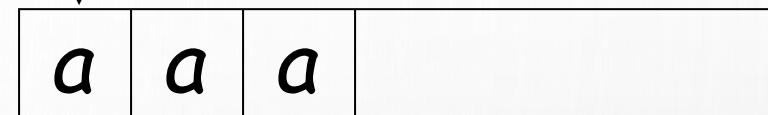
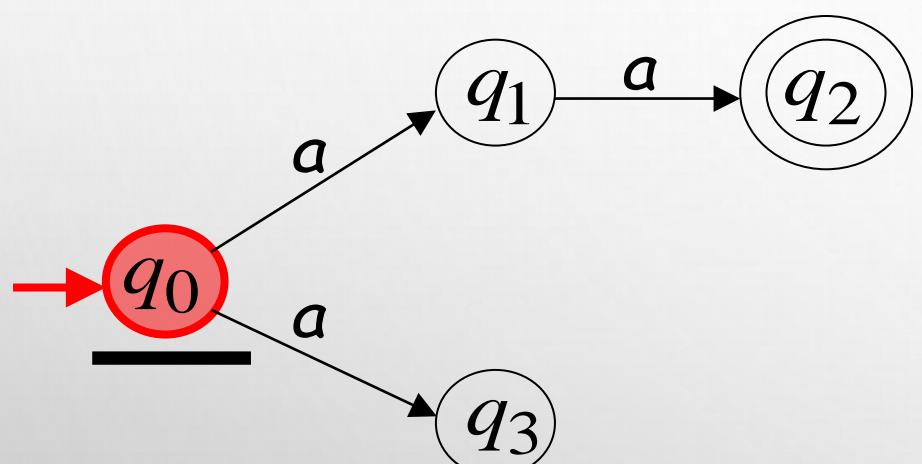


Automaton halts

“reject”

Input cannot be  
consumed.

# String Rejection by NFA: Second Choice



Input cannot be consumed.

Automaton halts  
reject

# NFA Rejection !

## An NFA rejects a string

- if there is no computation of the NFA that accepts the string.

**OR**

- for each computation, all the input symbols are consumed and the automaton is in a non-final state

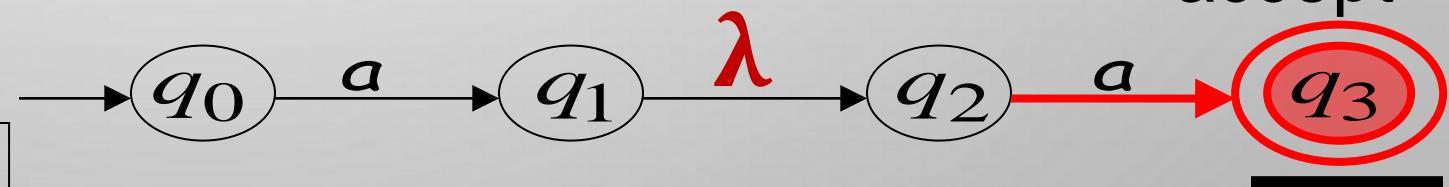
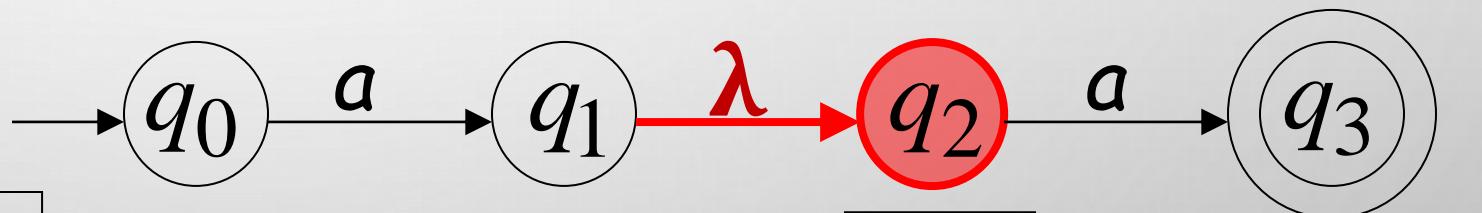
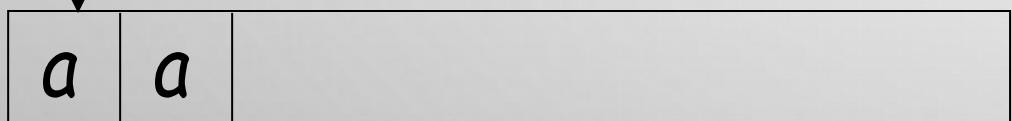
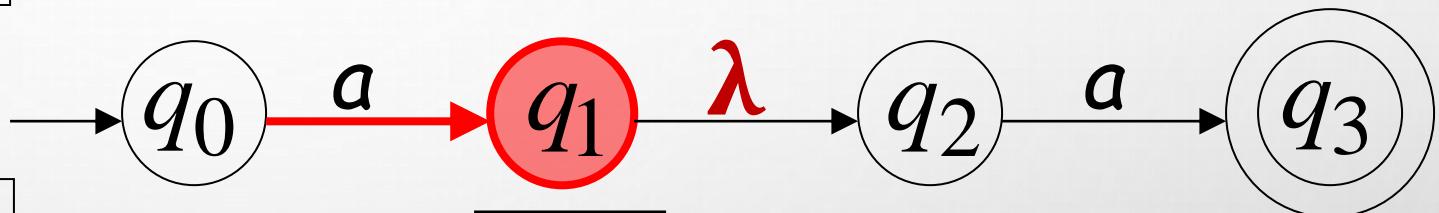
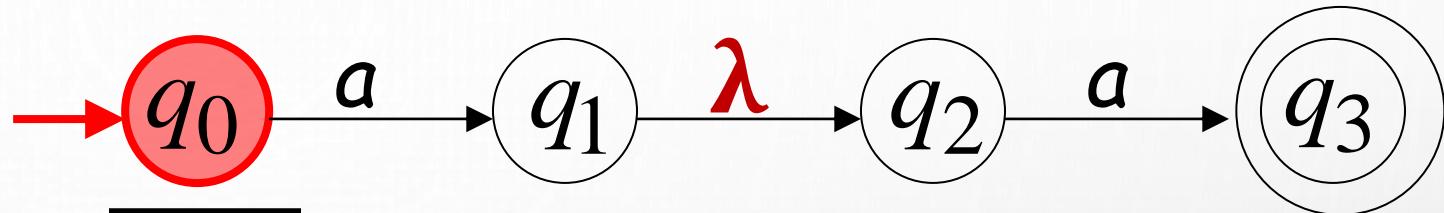
**OR**

- for each computation, some input symbols are consumed and there is no path for the next input symbol(the input cannot be consumed)
- The automaton may be in a final or non-final state.

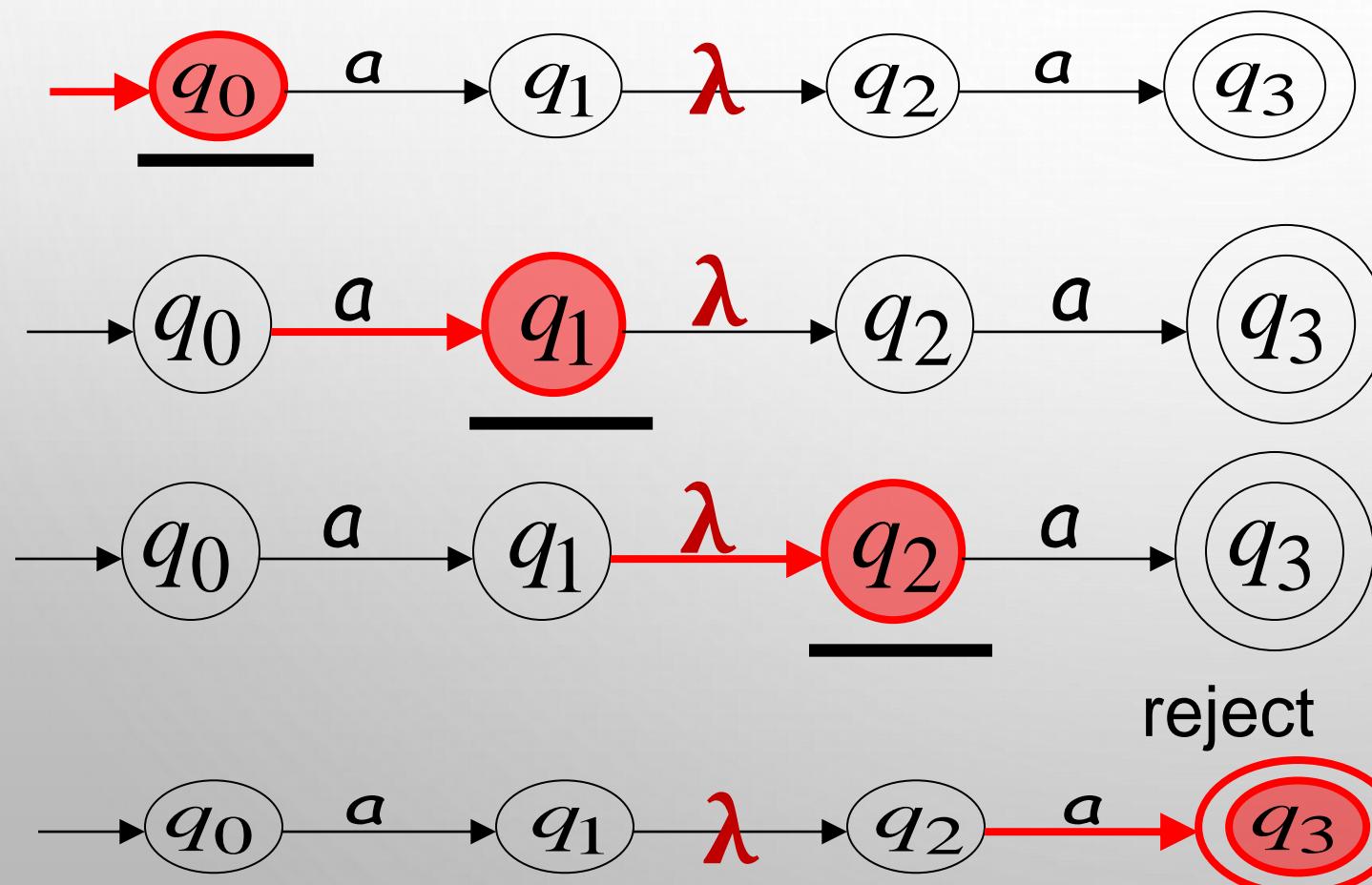
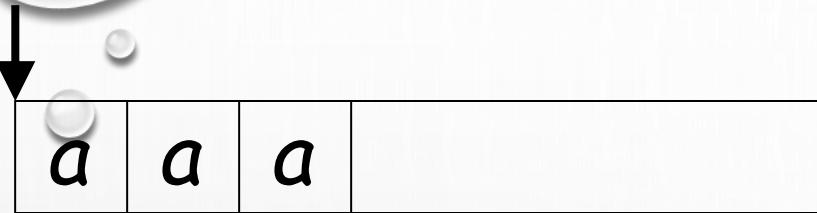
# $\lambda$ -Transition

- It could potentially occur at any stage – and allows the automaton to change state with no input.

- String aa is accepted.-



# $\lambda$ -Transition-Example



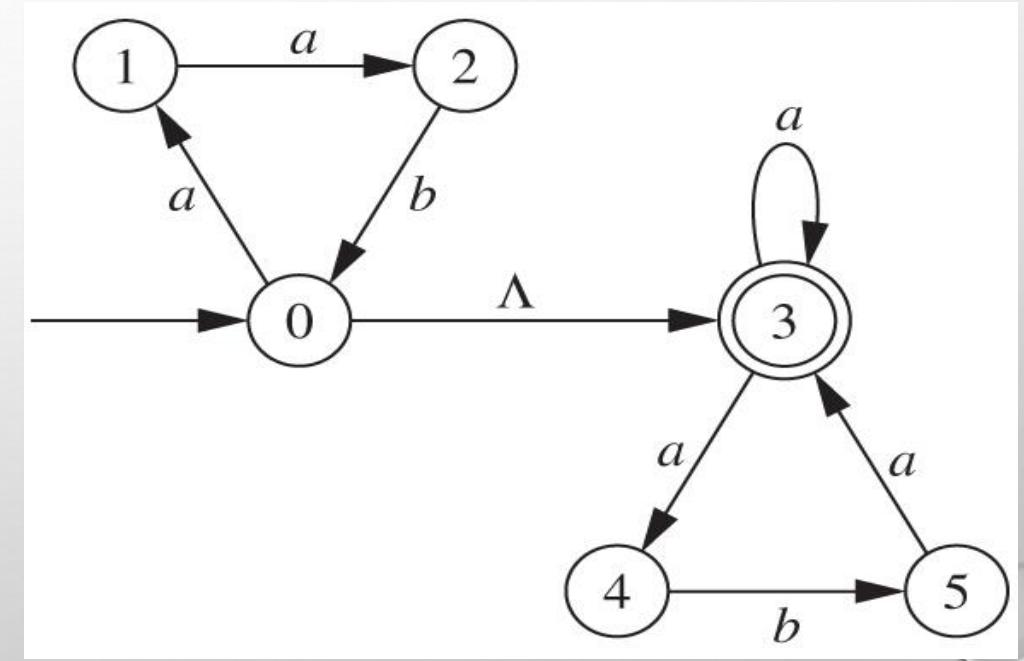
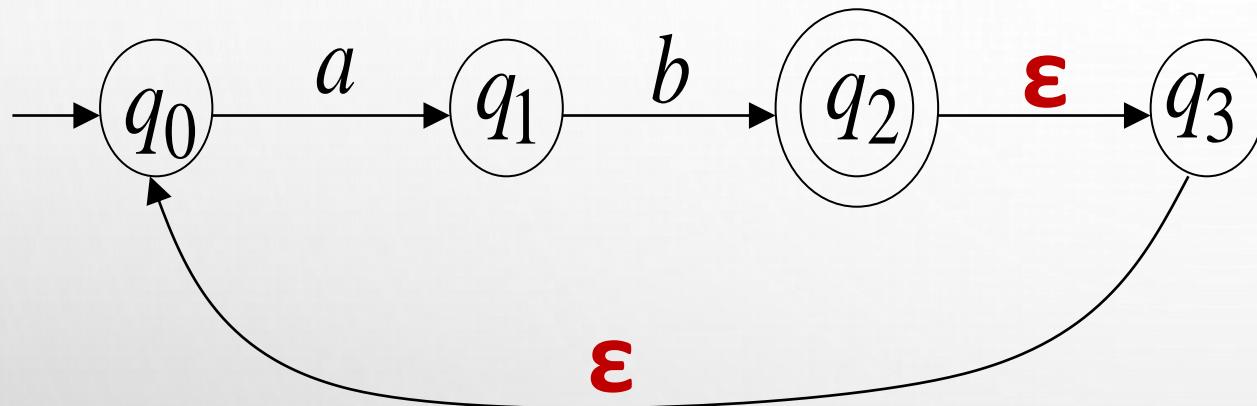
(read head on tape doesn't move)

reject

String aaa is rejected.

# Example-3

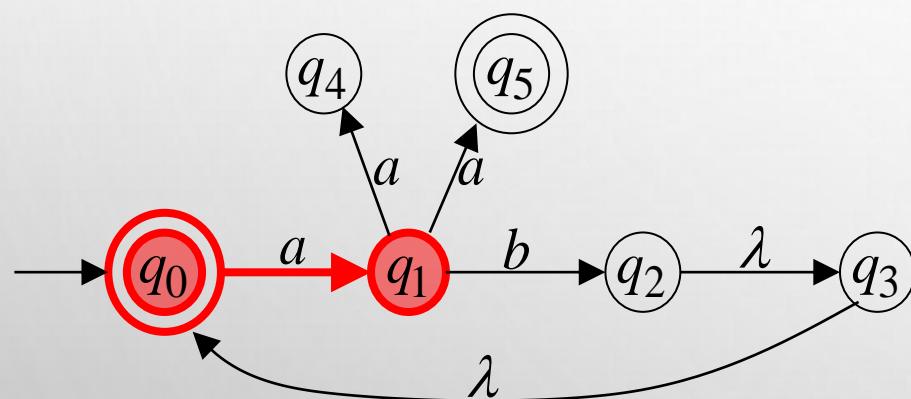
□ Find strings accepted/rejected by the NFA



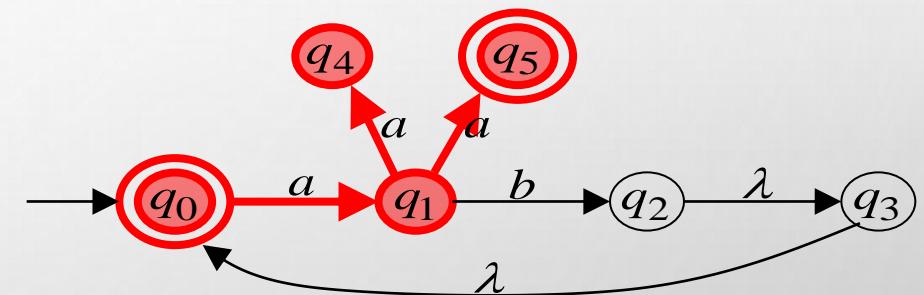
# EXTENDED TRANSITION FUNCTION

$$\delta^* : Q \times \Sigma^* \rightarrow 2^Q$$

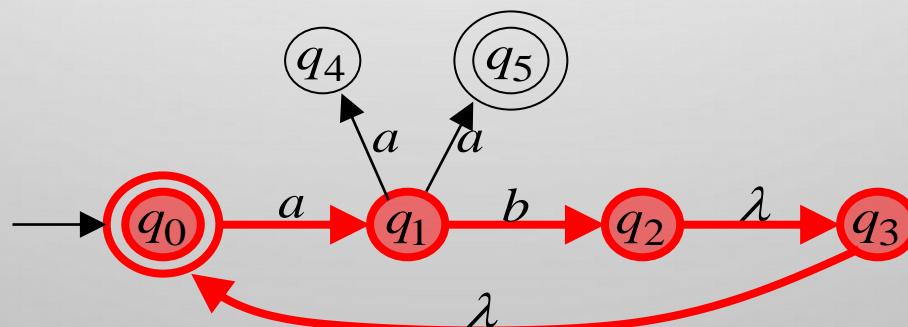
$$\delta^*(q_0, a) = \{q_1\}$$



$$\delta^*(q_0, aa) = \{q_4, q_5\}$$



$$\delta^*(q_0, ab) = \{q_2, q_3, q_0\}$$



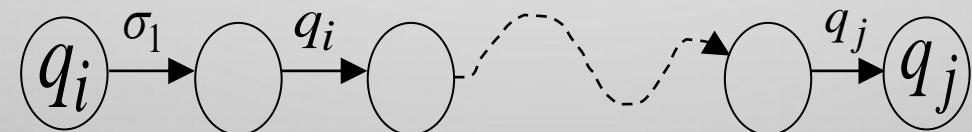
# ETF Special Case

For any state  $q$ ,  $q \in \delta^*(q, \lambda)$

In general,  $q_j \in \delta^*(q_i, w)$



$$w = \sigma_1 \sigma_2 \cdots \sigma_k$$

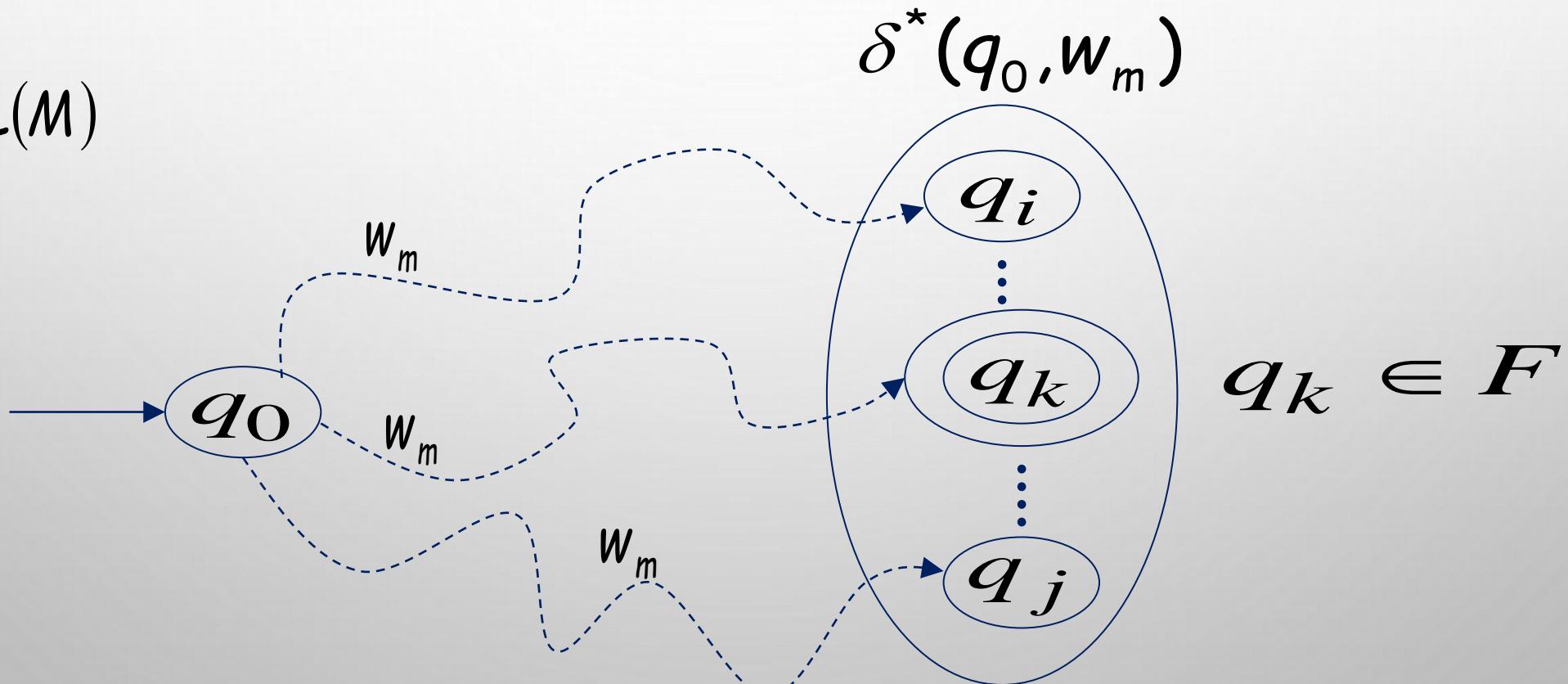


# Language of NFA

The language accepted by NFA  $M$  is defined as the set of strings accepted by NFA.

Formally,  $L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \emptyset\}$

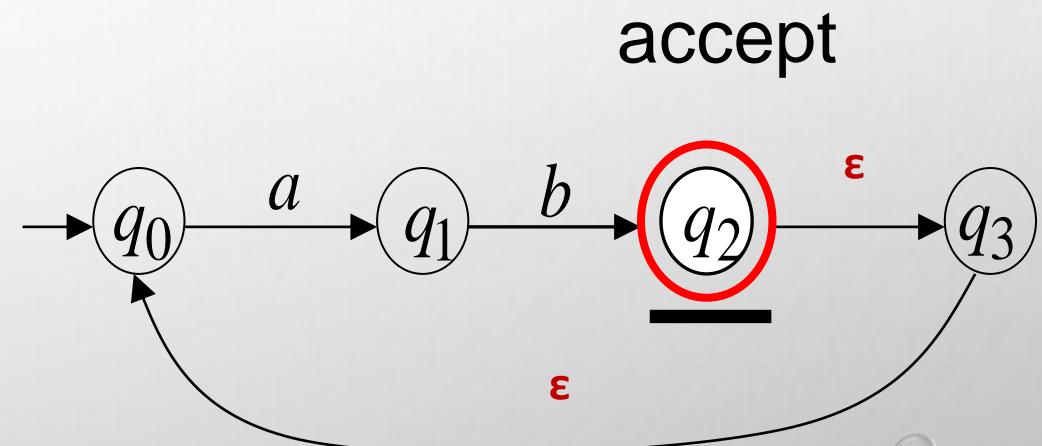
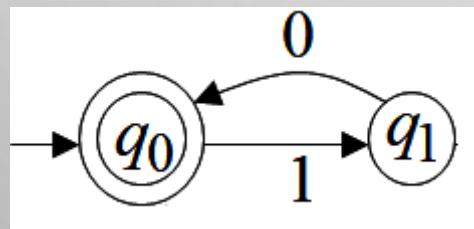
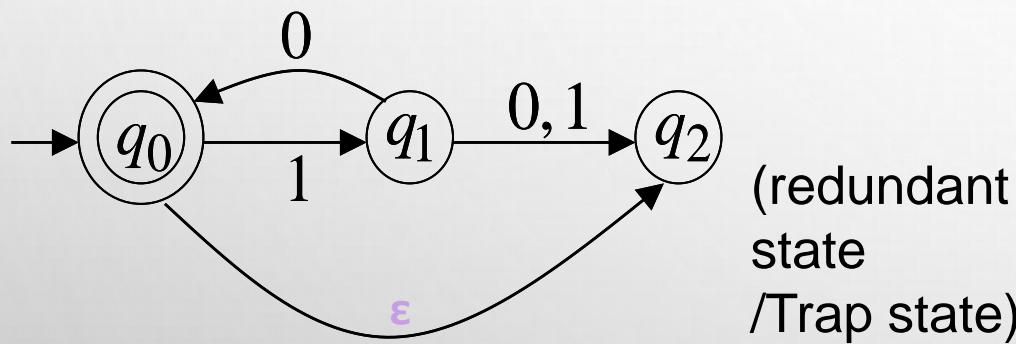
$$w_m \in L(M)$$



# Language of NFA

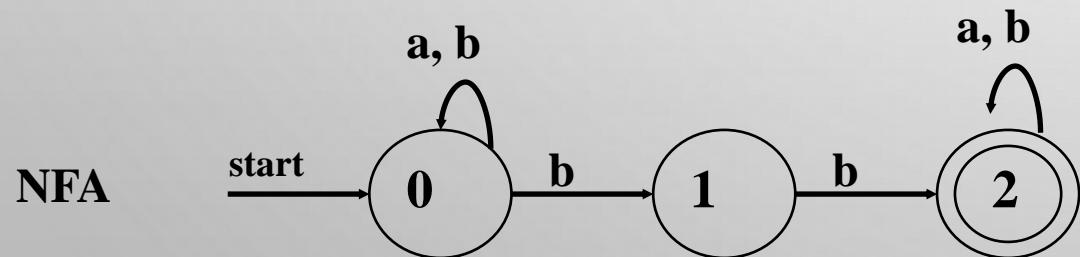
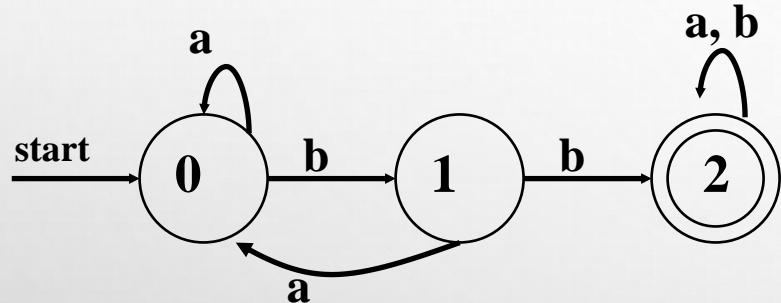
$$L(M) = \{\varepsilon, 10, 1010, 101010, \dots\} \\ = \{10\}^*$$

$$L = \{ab, abab, ababab, \dots\} \\ = \{ab\}^+$$

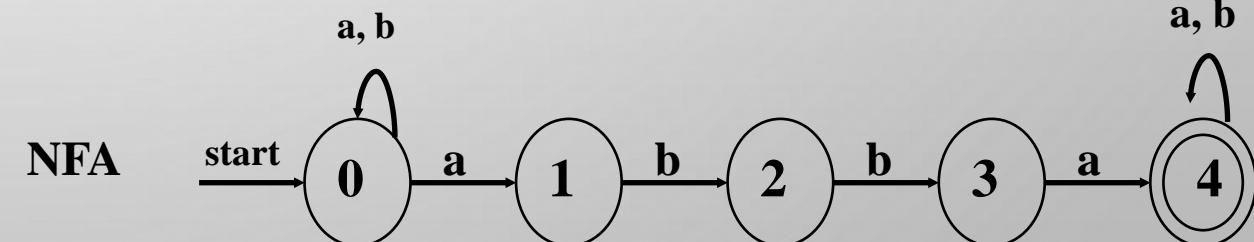
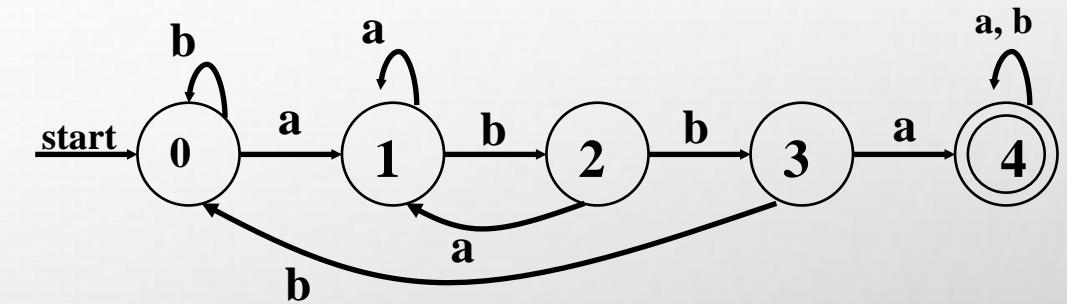


# Designing NFA

**w<sub>1</sub>bbw<sub>2</sub>,**    w<sub>1</sub>, w<sub>2</sub> ∈ {a, b}\*  
DFA

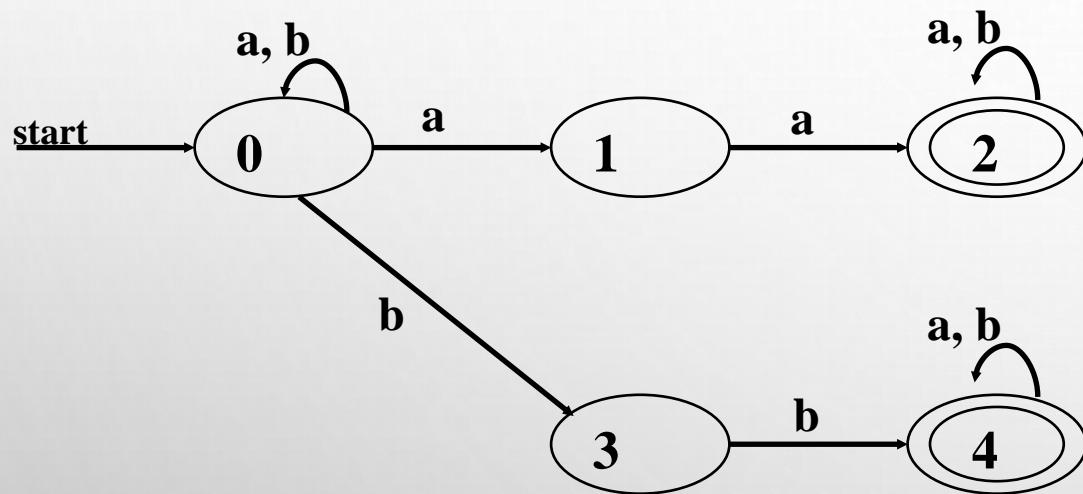


**w<sub>1</sub>abbaw<sub>2</sub>,**    w<sub>1</sub>, w<sub>2</sub> ∈ {a, b}\*  
DFA

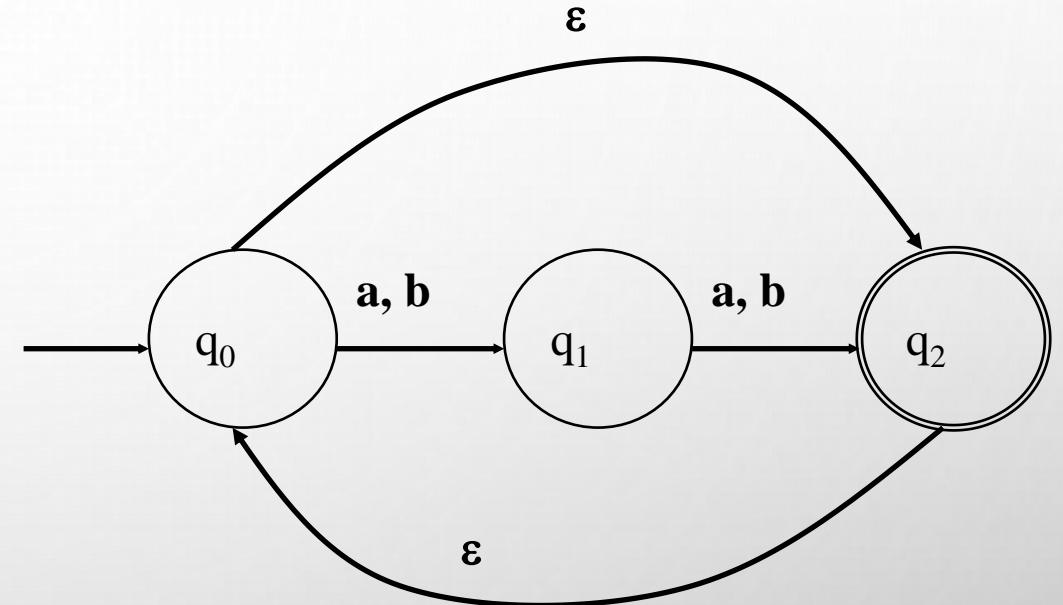


## Designing NFA

- An NFA that accepts string over  $\{a, b\}$  with substring **aa** or **bb**



An NFA accepts all strings over  $\{a, b\}$  of even length



- There are 2 distinct acceptance paths for the string **abaaaabb**

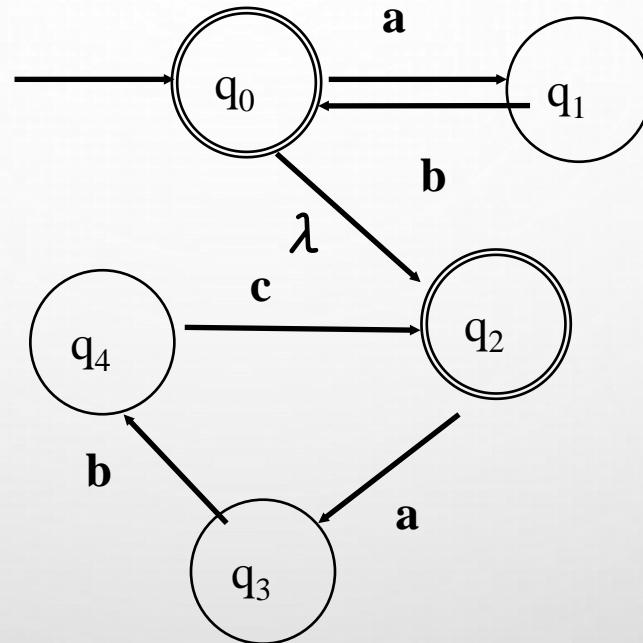
# Designing NFA

1.  $L(M) = \{abab^n : n \geq 0\} \cup \{aba^m : m \geq 0\}$ .
2.  $L(M) = \{abab^n : n \geq 0\} \cup \{aba^m : m \geq 0\}$  with no more than five states.
3.  $L(M) = \{abab^n : n \geq 0\} \cup \{abaa^m : m \geq 0\}$  with no more than five states.
4. Find an NFA with three states for  $L = \{a^n : n \geq 1\} \cup \{b^m a^k : m, k \geq 0\}$ .
5. Find an NFA with four states for  $L = \{a^n : n \geq 0\} \cup \{b^m a : m \geq 1\}$ .
6.  $L(M) = \{(ab)^n : n \geq 0\} \cup \{(abc)^m : m \geq 0\}$ .
7.  $L(M) = \{w : w \in \{ab, abc\}^*\}$ .
8. All strings such that the third symbol from the right end is a "0" over  $\{0, 1\}$ .
9. All strings that contain the substring 0101 over  $\{0, 1\}$ .
10. Construct a NFA for the language  $L = \{0^k : k \text{ is multiple of } 2 \text{ or } 3\}$ .
11. All strings containing exactly 4 "0"s or an even number of "1"s.
12. All strings that contain an even number of zeros or exactly two ones.
13. The set of strings over alphabet  $\{0, 1, 2 \dots 9\}$  such that the final digit has appeared before.
14. The set of strings over alphabet  $\{0, 1, 2 \dots 9\}$  such that the final digit has not appeared before.
15. **Construct NFA to define identifiers in C programming Language (Hint: Ullman-Compiler).**
16. **Construct NFA to identify relational operators (Hint: Ullman-Compiler).**
17. **Construct NFA to identify decimal numbers (Real, Integers, Scientific ) (Hint: Ullman-Compiler).**

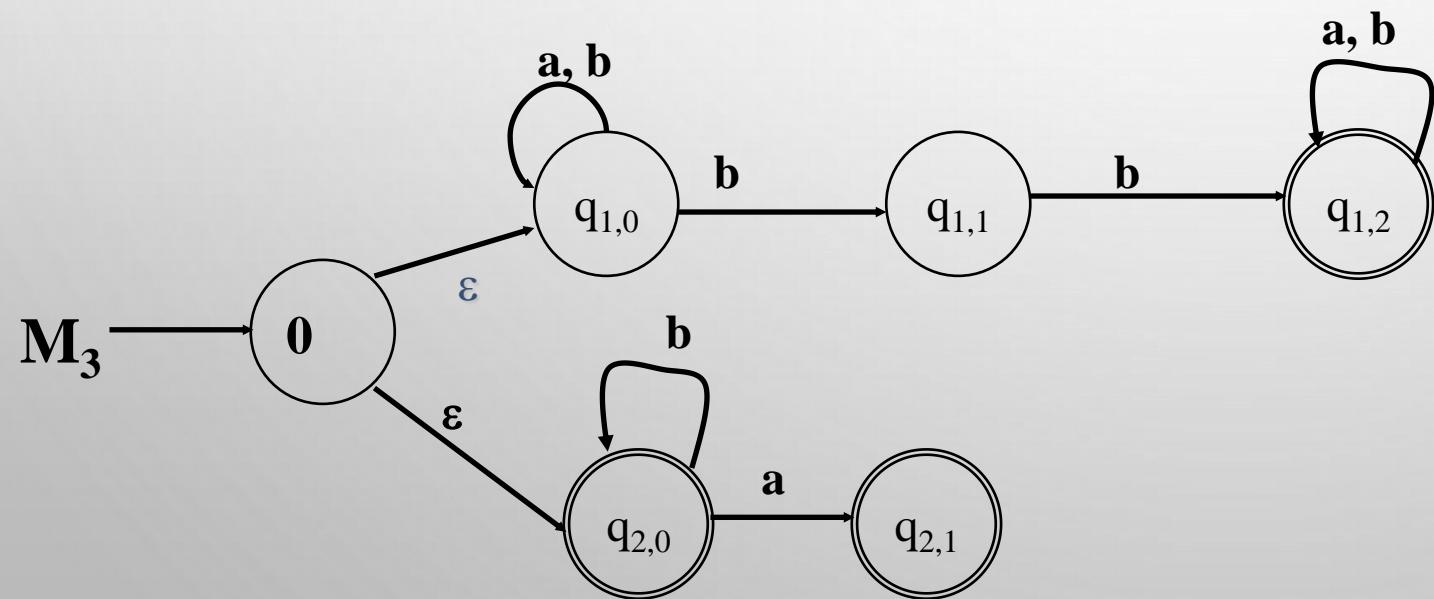
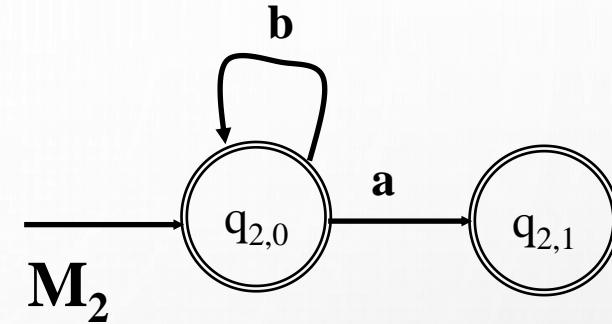
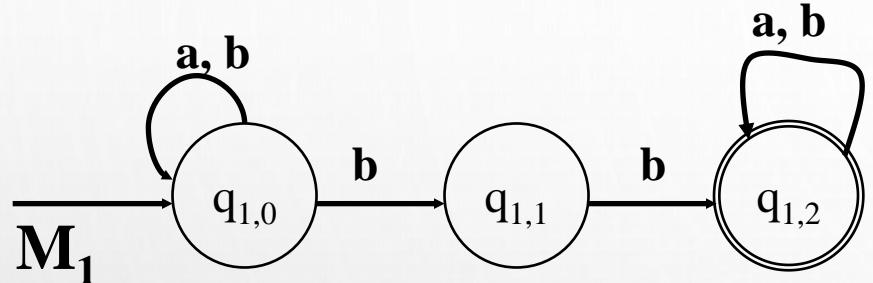
## Exercise

$$L(M) = \{(ab)^n : n \geq 0\} \cup \{(abc)^m : m \geq 0\}$$

Construct an NFA for the language  $L = \{0^k : k \text{ is multiple of 2 or 3}\}$ .



# FIND LANGUAGES ?



$M_3$  is union of  $M_1$  and  $M_2$ .

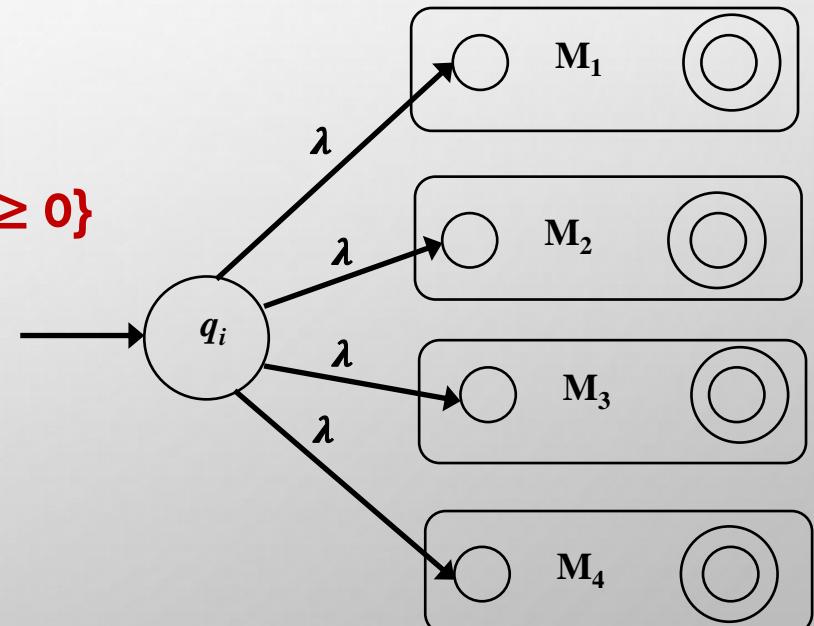
# MODIFIED NFA

- Consider an NFA with multiple initial states.
- $M = (Q, \Sigma, \delta, Q_0, F)$ , where:
  - $Q_0 \subseteq Q$  is the set of initial states.
- The language accepted by NFA  $M$  is defined as the set of strings accepted by NFA.

Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q, w) \cap F \neq \emptyset, \text{ for any } q \in Q_0\}$$

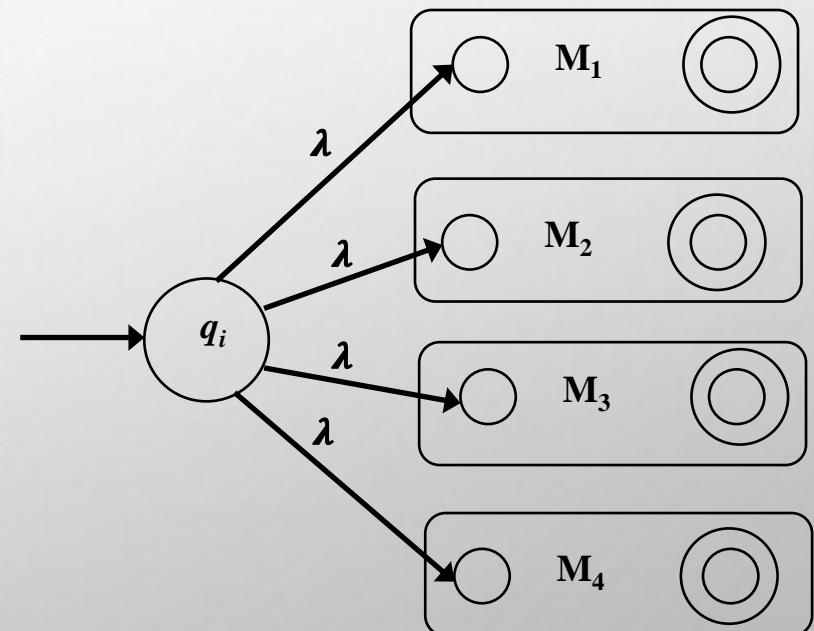
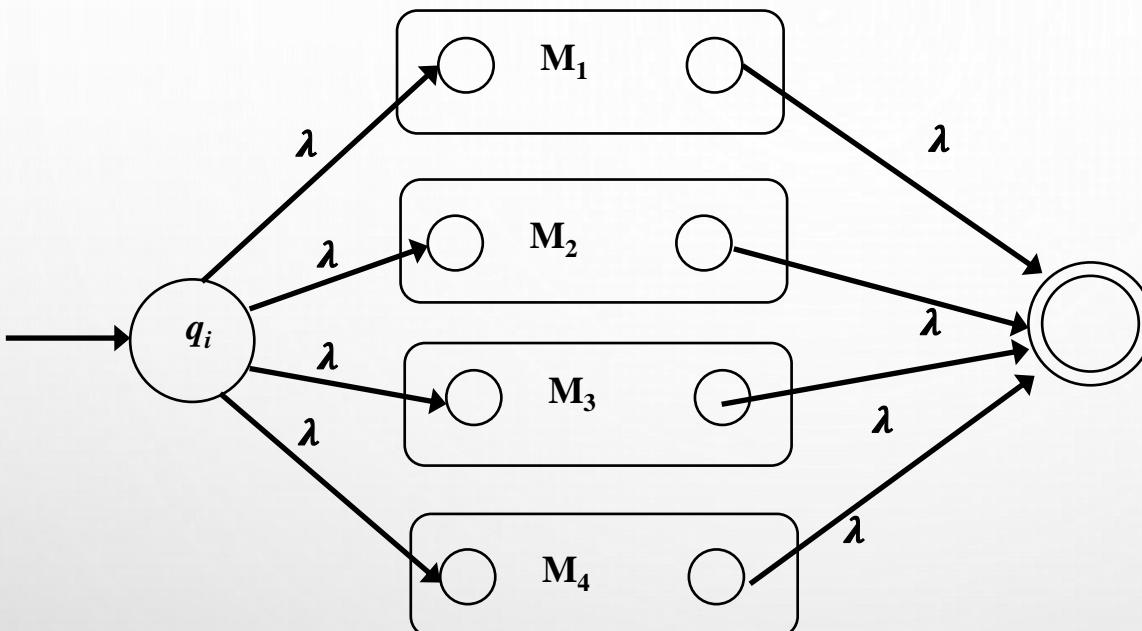
- **Construct an NFA with single initial states.**
  - $L = \{abab^n : n \geq 0\} \cup \{aba^m : m \geq 0\} \cup \{a^n : n \geq 1\} \cup \{b^m a^k : m, k \geq 0\}$
  - Introduce new initial state  $q'$
  - Add new transitions  $\delta(q', \lambda) = \{q\}$ , for all  $q \in Q_0$



## Exercise-

For a given NFA with multiple final states, construct an NFA with a single final state.

# MODIFIED NFA



**Exercise-**

For a given NFA with multiple final states, construct an NFA with a single final state.

# DFA vs. NFA

- |  |   |
|--|---|
| <ol style="list-style-type: none"><li>1. All transitions are deterministic.<ul style="list-style-type: none"><li>• Each transition leads to exactly one state.</li></ul></li><li>2. For each state, transition on all possible symbols (alphabet) should be defined.</li><li>3. Accepts input if the last state is the final state after processing the input string.</li><li>4. Harder to construct.</li><li>5. Practical implementation is feasible.</li></ol> | <ol style="list-style-type: none"><li>1. Some transitions could be non-deterministic.<ul style="list-style-type: none"><li>• A transition could lead to a set of states.</li></ul></li><li>2. Not all symbol transitions need to be defined explicitly (if undefined will go to a dead state).</li><li>3. Accepts input if at-least <i>one final state is reachable after processing the input string</i>.</li><li>4. Easier to construct in comparison to DFA.</li><li>5. Practical implementation should be deterministic (so needs conversion to DFA).</li></ol> |
|--|---|

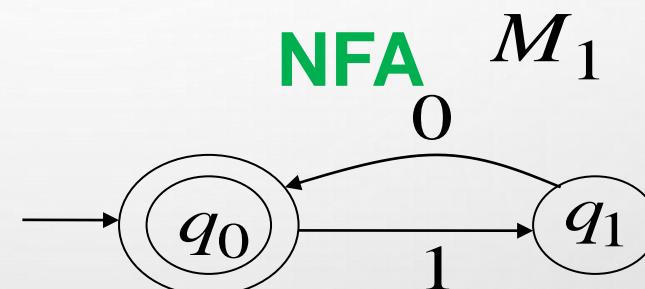
But, DFAs and NFAs are equivalent (in their power) !!

# Equivalent Machines

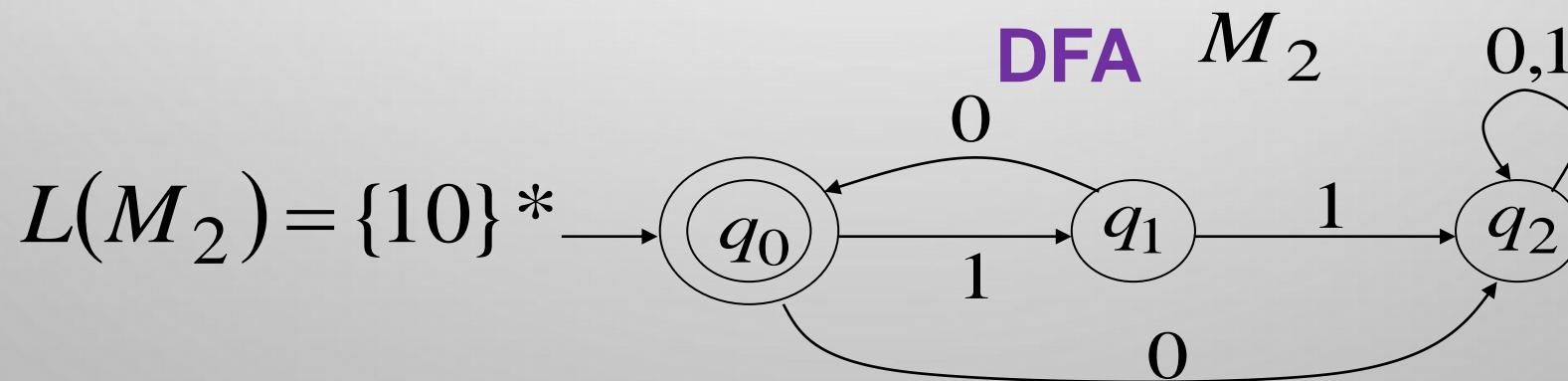
- Two finite accepters,  $M_1$  and  $M_2$ , are equivalent if both accept the same language.

$$L(M_1) = L(M_2)$$

$$L(M_1) = \{10\}^*$$

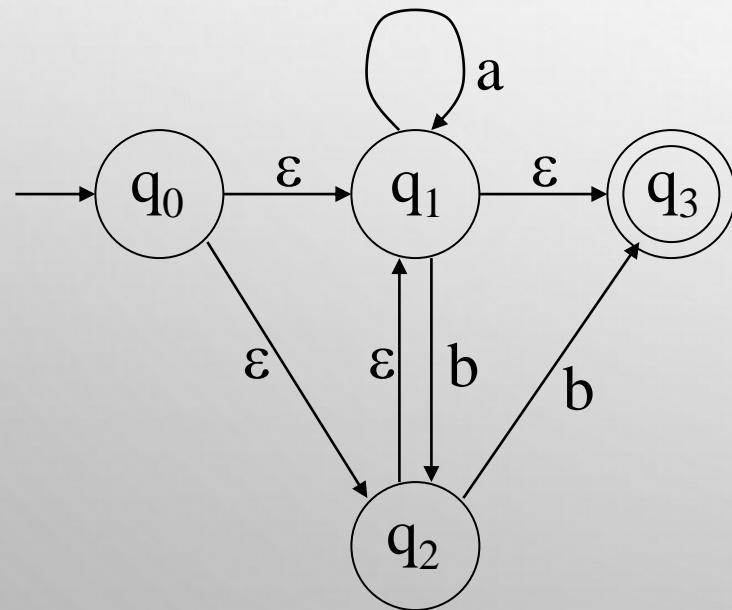


$$L(M_2) = \{10\}^*$$



# Removing Nondeterminism (Conversion from NFA to DFA)

- $\epsilon$ -NFA – A NFA with at least one  $\lambda$  transition.
- $\epsilon$ -Closure( $q$ ) =  $\delta^*(q, \lambda)$
- In an  $\epsilon$ -NFA, the  $\epsilon$ -closure( $q$ ) of a state  $q$  is the set of all states which are reachable from  $q$  by a path whose edges are labeled by  $\epsilon/\lambda$ .



$$\begin{aligned}\epsilon\text{-closure}(q_0) &= \{q_0, q_1, q_2, q_3\} \\ \epsilon\text{-closure}(q_1) &= \{q_1, q_3\} \\ \epsilon\text{-closure}(q_2) &= \{q_2, q_1, q_3\}\end{aligned}$$

# Removing Nondeterminism

**$\epsilon$ -NFA**  $N = (Q, \Sigma, \delta, q_0, F)$

$\delta^*(q, \lambda)$

=  **$\epsilon$ -closure(q)** for all  $q \in Q$

: set of  $\epsilon$ -NFA states that are reachable from  $q$  on  $\epsilon$ -transitions only.

$\delta^*(T, \lambda)$

=  **$\epsilon$ -closure(T)** for all  $T \subseteq Q$

:  $\epsilon$ -NFA states reachable from some state  $t \in T$  on  $\epsilon$ -transitions only

$\epsilon$ -closure( $T$ )

=  $\epsilon$ -closure( $t_1$ )  $\cup$   $\epsilon$ -closure( $t_2$ )  $\cup$   $\epsilon$ -closure( $t_3$ )  $\cup$ .....

**$move(T, a)$**

:  $T \subseteq Q, a \in \Sigma$

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

**$move(T, a)$**

=  $move(\{q_1, q_2, \dots, q_i\}, a)$

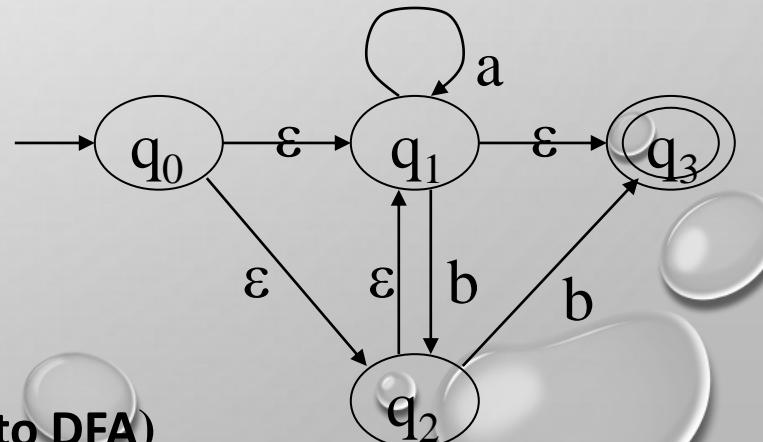
=  $\delta(q_1, a) \cup \delta(q_2, a) \cup \dots \cup \delta(q_i, a)$

$\epsilon$ -closure( $q_1$ ) = { $q_1, q_3$ }

$\epsilon$ -closure({ $q_1, q_2$ }) = { $q_1, q_3, q_2$ }

$move(\{q_1, q_2\}, b) = \delta(q_1, b) \cup \delta(q_2, b) = \{q_2, q_3\}$

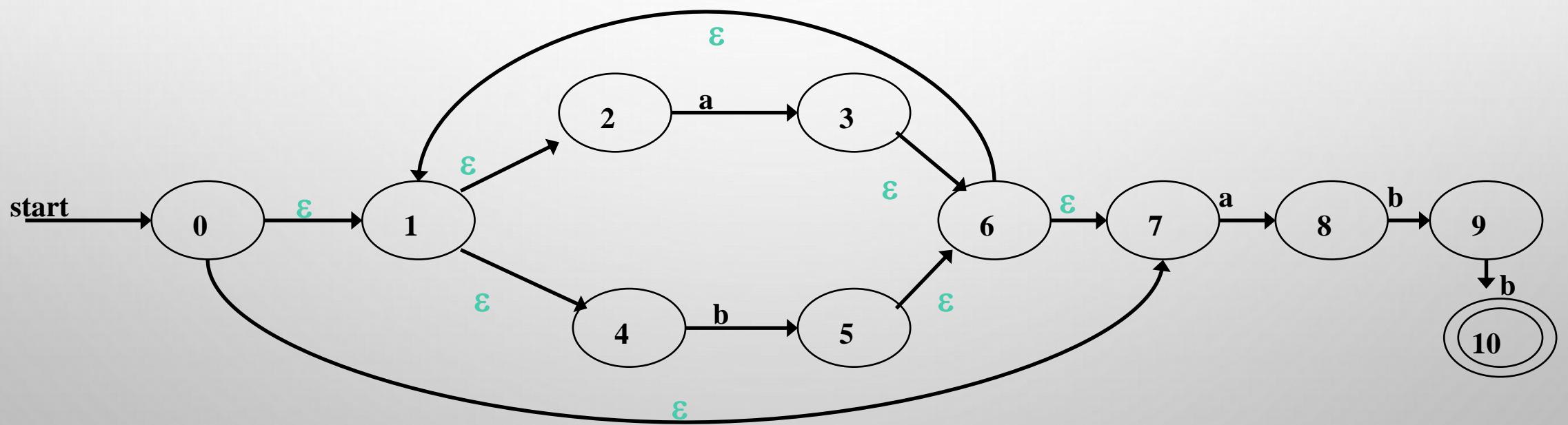
$move(\{q_0, q_1\}, b) = \delta(q_0, b) \cup \delta(q_1, b) = \{\} \cup \{q_2\} = \{q_2\}$



These 3 operations are utilized to facilitate the conversion process (from NFA to DFA)

# REMOVING NONDETERMINISM

- $\delta^*(0, \lambda) = \varepsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\}$  (all states reachable from 0 on  $\varepsilon$ -moves)
- $\varepsilon\text{-closure}(\{0, 6\}) = \varepsilon\text{-closure}(0) \cup \varepsilon\text{-closure}(6) = \{0, 1, 2, 4, 7\} \cup \{6, 1, 2, 4, 7\} = \{0, 1, 2, 4, 6, 7\}$
- $\text{move}(\{0, 1, 2, 4, 7\}, a) = \delta(0, a) \cup \delta(1, a) \cup \delta(2, a) \cup \delta(4, a) \cup \delta(7, a) = \{\} \cup \{\} \cup \{3\} \cup \{\} \cup \{8\} = \{3, 8\}$



# $\varepsilon$ -NFA $\rightarrow$ DFA with Subset Construction

$\varepsilon$ -NFA N  $\Rightarrow$  DFA M construction:

Given N = (Q,  $\Sigma$ ,  $\delta$ ,  $q_0$ , F)

**Construct M = (Q',  $\Sigma$ ,  $\delta'$ ,  $q'_0$ , F')**

$Q' \subseteq 2^Q$ , all the possible states of  $\varepsilon$ -NFA, M could be in.

- DFA M has a trap state **ERR**
  - Serves as the “error” state, when needed
  - $\delta'(\text{ERR}, a) = \text{ERR}, \forall a \in \Sigma$

**$q'_0 = \varepsilon\text{-closure}(q_0)$  – Initial state of DFA M**

Define transition for all new states and for all  $a \in \Sigma$ .

**$\delta'(q', a) = \varepsilon\text{-closure}(\text{move}(q, a))$  for all  $a \in \Sigma$**

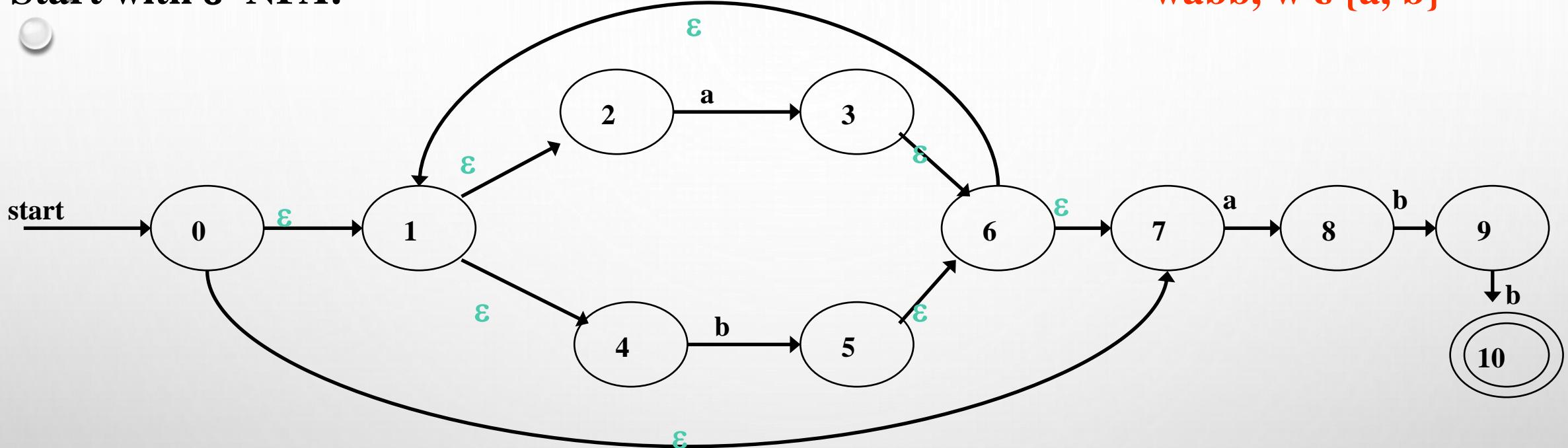
If there is no new state, stop.

**The final state of DFA is the one in which at-least one state is NFA's final state**

$F' = \{q' \in Q' \mid q' \cap F \neq \emptyset\}$

## SUBSET CONSTRUCTION (CONVERSION EXAMPLE-1)

Start with  $\epsilon$ -NFA:

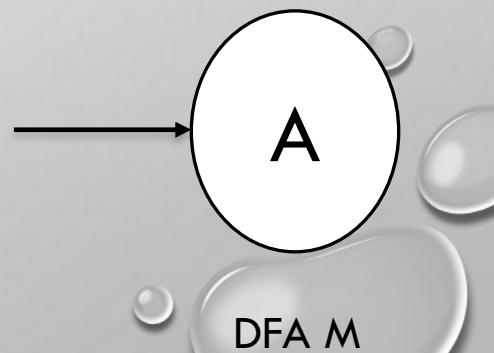


wabb, w  $\in \{a, b\}^*$

First we calculate:  $\epsilon$ -closure(0) (i.e., initial state)

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

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



DFA M

# CONVERSION EXAMPLE-1

2<sup>nd</sup>, we calculate : a :  $\delta_M(A, a) = \epsilon\text{-closure}(move(A, a))$   
 b :  $\delta_M(A, b) = \epsilon\text{-closure}(move(A, b))$

a :  $\epsilon\text{-closure}(move(A, a)) = \epsilon\text{-closure}(move(\{0, 1, 2, 4, 7\}, a))$   
 adds {3, 8} (since  $move(2, a) = 3$  and  $move(7, a) = 8$ )

From this we have :  $\epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$   
 (since  $3 \rightarrow 6 \rightarrow 1 \rightarrow 4$ ,  $6 \rightarrow 7$ , and  $1 \rightarrow 2$  all by  $\epsilon$ -moves)

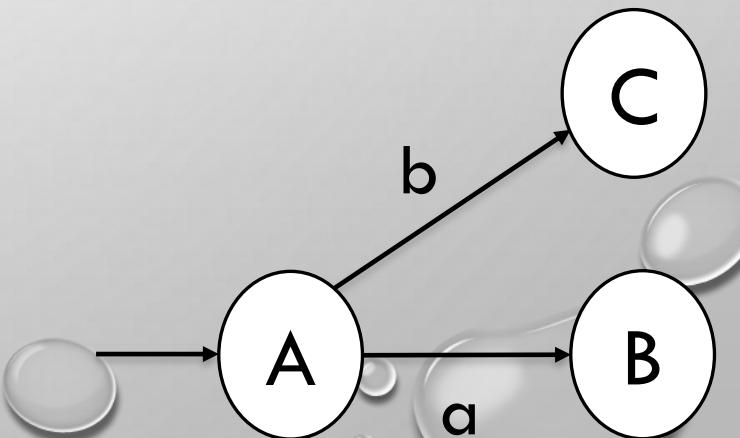
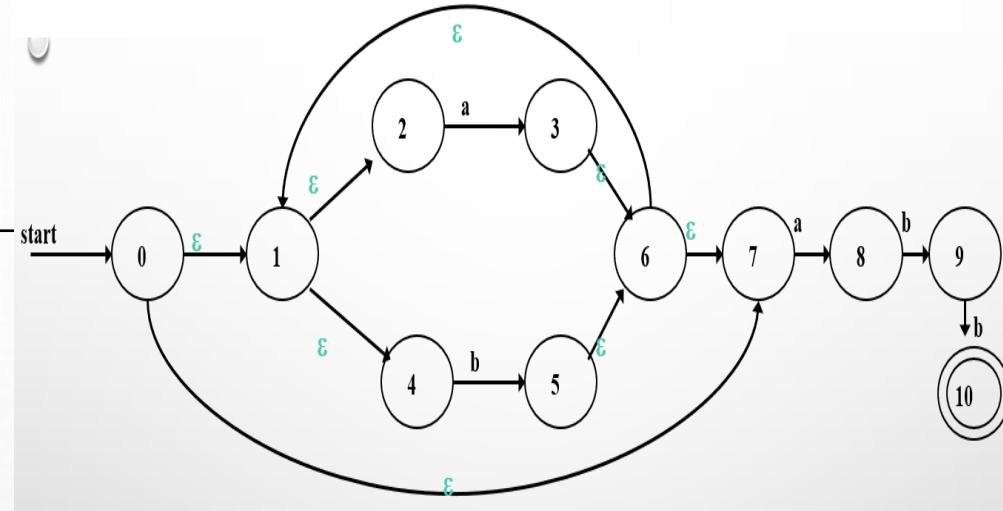
Let B = {1, 2, 3, 4, 6, 7, 8} be a new state in DFA M. Define  $\delta_M(A, a) = B$

b :  $\epsilon\text{-closure}(move(A, b)) = \epsilon\text{-closure}(move(\{0, 1, 2, 4, 7\}, b))$

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

From this we have :  $\epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\}$   
 (since  $5 \rightarrow 6 \rightarrow 1 \rightarrow 4$ ,  $6 \rightarrow 7$ , and  $1 \rightarrow 2$  all by  $\epsilon$ -moves)

Let C = {1, 2, 4, 5, 6, 7} be a new state in DFA M. Define  $\delta_M(A, b) = C$



# CONVERSION EXAMPLE-1

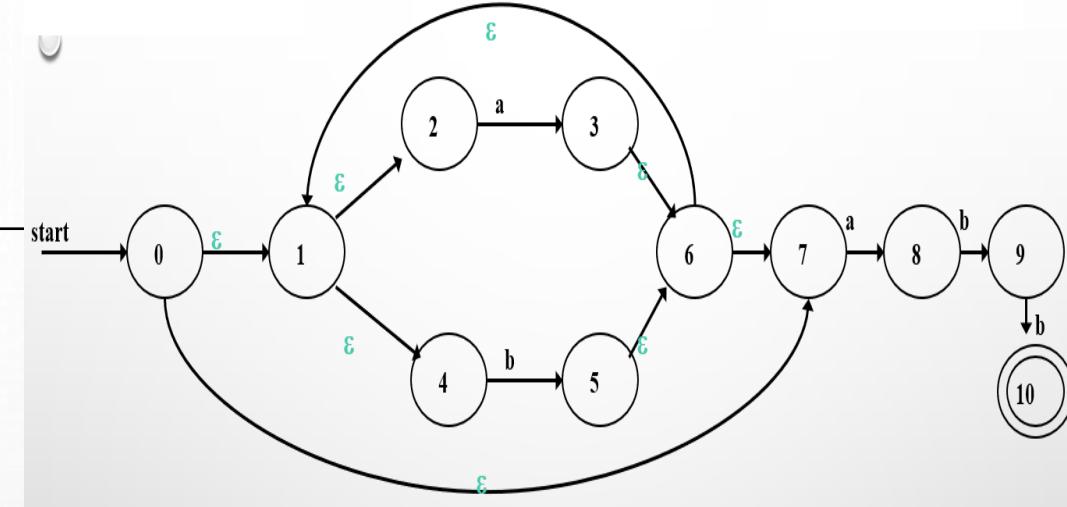
$\epsilon\text{-closure}(move(A, a)) = \epsilon\text{-closure}(\{3, 8\})$

$move(\{0, 1, 2, 4, 7\}, a) = \{3, 8\}$

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

From this we have :  $\epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$   
 (since  $3 \rightarrow 6 \rightarrow 1 \rightarrow 4$ ,  $6 \rightarrow 7$ , and  $1 \rightarrow 2$  all by  $\epsilon$ -moves)

Let  $B = \{1, 2, 3, 4, 6, 7, 8\}$  be a new state in DFA M. Define  $\delta_M(A, a) = B$

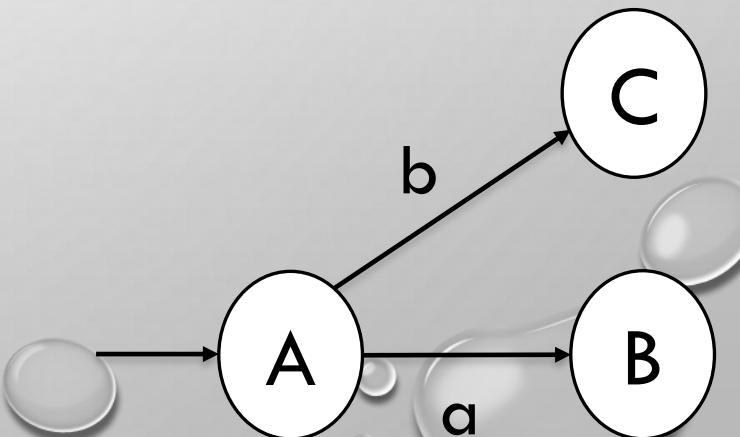


**b** :  $\epsilon\text{-closure}(move(A, b)) = \epsilon\text{-closure}(move(\{0,1,2,4,7\}, b))$

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

From this we have :  $\epsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\}$   
 (since  $5 \rightarrow 6 \rightarrow 1 \rightarrow 4$ ,  $6 \rightarrow 7$ , and  $1 \rightarrow 2$  all by  $\epsilon$ -moves)

Let  $C = \{1,2,4,5,6,7\}$  be a new state in DFA M. Define  $\delta_M(A, b) = C$



# CONVERSION EXAMPLE-1

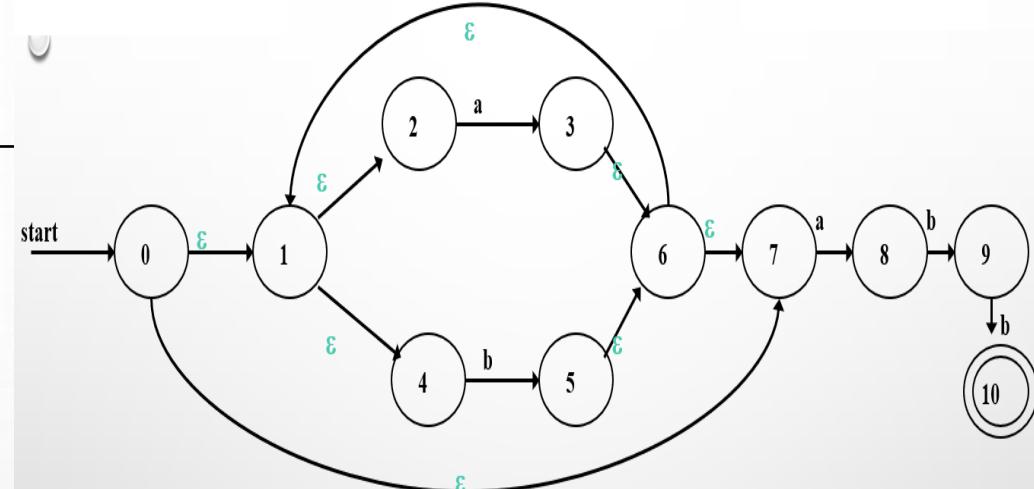
3<sup>rd</sup>, we calculate for states B on {a, b} i.e.  $\delta_M(B, a)$  and  $\delta_M(B, b)$

a :  $\epsilon\text{-closure}(move(B, a)) = \epsilon\text{-closure}(move(\{1,2,3,4,6,7,8\}, a))$   
 $= \{1,2,3,4,6,7,8\} = B$

Define  $\delta_M(B, a) = B$

b :  $\epsilon\text{-closure}(move(B, b)) = \epsilon\text{-closure}(move(\{1,2,3,4,6,7,8\}, b))$   
 $= \{1,2,4,5,6,7,9\} = D$

Define  $\delta_M(B, b) = D$



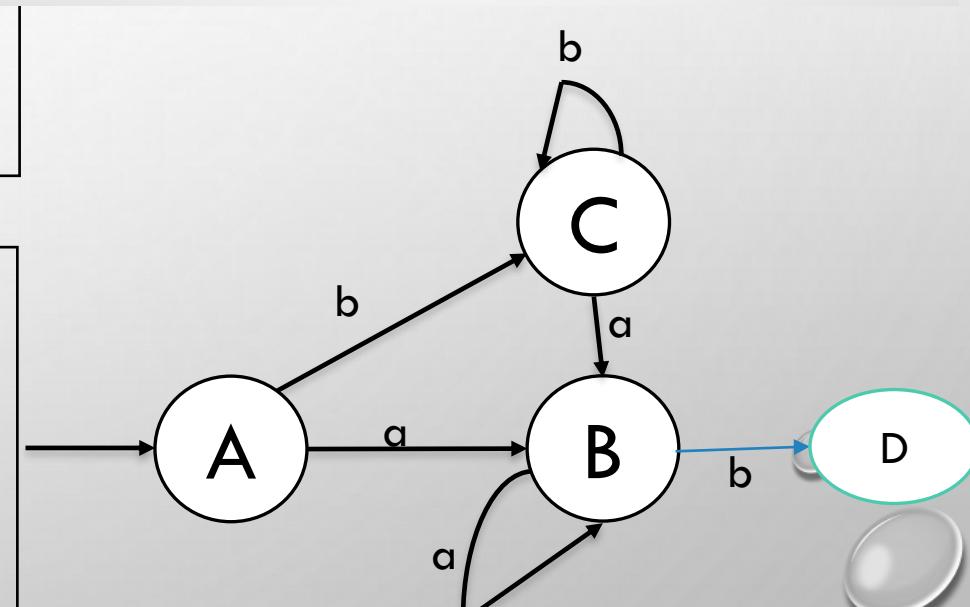
4<sup>th</sup>, we calculate for state C on {a, b} i.e.  $\delta_M(C, a)$  and  $\delta_M(C, b)$

a :  $\epsilon\text{-closure}(move(C, a)) = \epsilon\text{-closure}(move(\{1,2,4,5,6,7\}, a))$   
 $= \{1,2,3,4,6,7,8\} = B$

Define  $\delta_M(C, a) = B$

b :  $\epsilon\text{-closure}(move(C, b)) = \epsilon\text{-closure}(move(\{1,2,4,5,6,7\}, b))$   
 $= \{1,2,4,5,6,7\} = C$

Define  $\delta_M(C, b) = C$



# CONVERSION EXAMPLE-1

5<sup>th</sup>, we calculate for state D on {a, b} i.e.  $\delta_M(D, a)$  and  $\delta_M(D, b)$

a :  $\epsilon\text{-closure}(move(D, a)) = \epsilon\text{-closure}(move(\{1,2,4,5,6,7,9\}, a))$   
 $= \{1,2,3,4,6,7,8\} = B$

Define  $\delta_M(D, a) = B$

b :  $\epsilon\text{-closure}(move(D, b)) = \epsilon\text{-closure}(move(\{1,2,4,5,6,7,9\}, b))$   
 $\epsilon\text{-closure}(\{5, 10\}) = \{1,2,4,5,6,7,10\} = E$

Define  $\delta_M(D, b) = E$

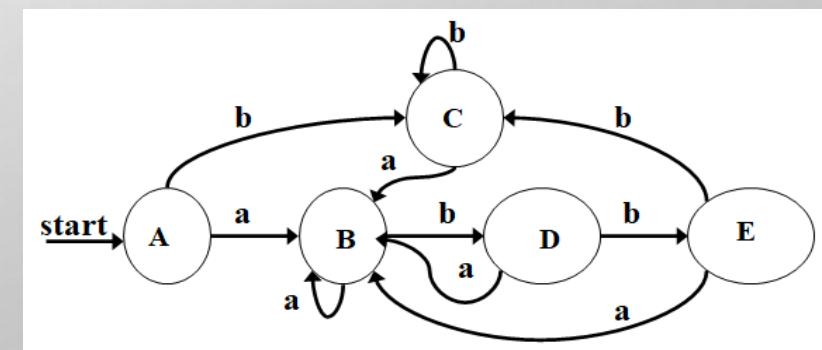
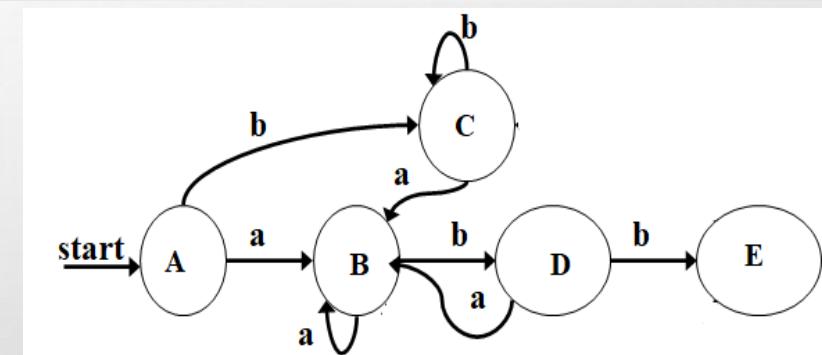
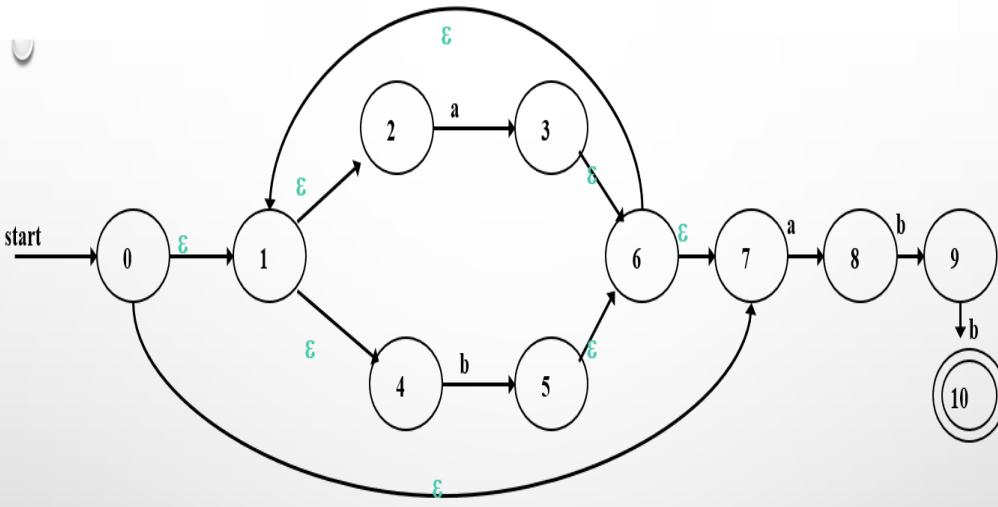
Finally, we calculate for state E on {a, b}

a :  $\epsilon\text{-closure}(move(E, a)) = \epsilon\text{-closure}(move(\{1,2,4,5,6,7,10\}, a))$   
 $= \epsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\} = B$

Define  $\delta_M(E, a) = B$

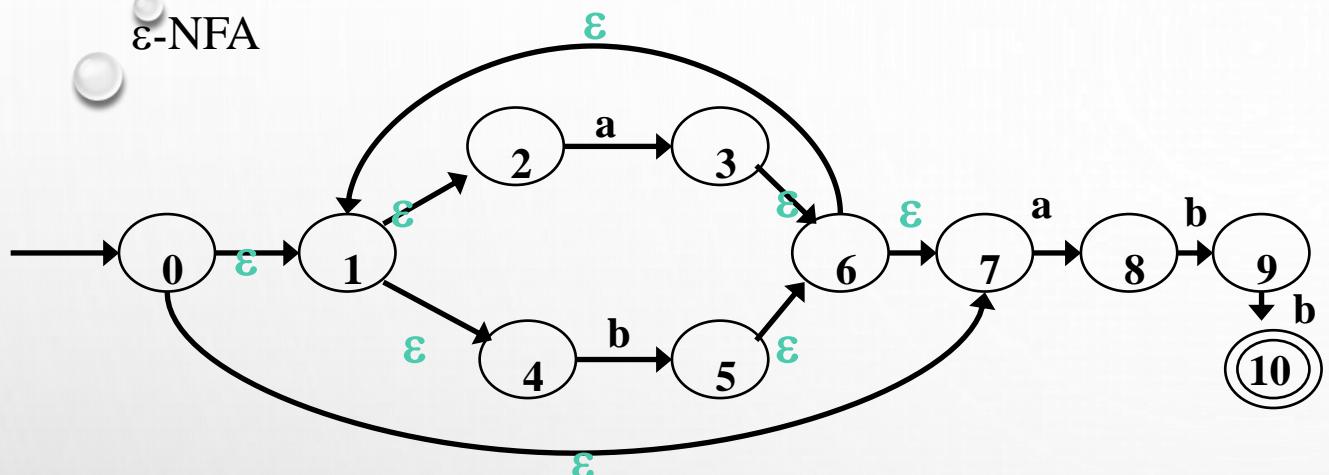
b :  $\epsilon\text{-closure}(move(E, b)) = \epsilon\text{-closure}(move(\{1,2,4,5,6,7,10\}, b))$   
 $= \{1,2,4,5,6,7\} = C$

Define  $\delta_M(E, b) = C$

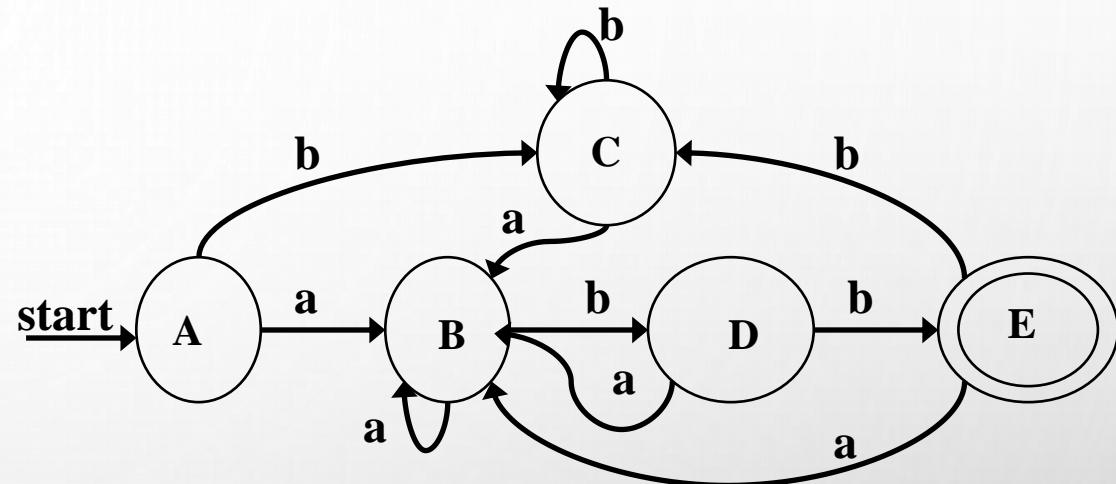


# CONVERSION EXAMPLE-1

$\epsilon$ -NFA



DFA



$$A = \{0, 1, 2, 4, 7\}$$

$$B = \{1, 2, 3, 4, 6, 7, 8\}$$

$$C = \{1, 2, 4, 5, 6, 7\}$$

$$D = \{1, 2, 4, 5, 6, 7, 9\}$$

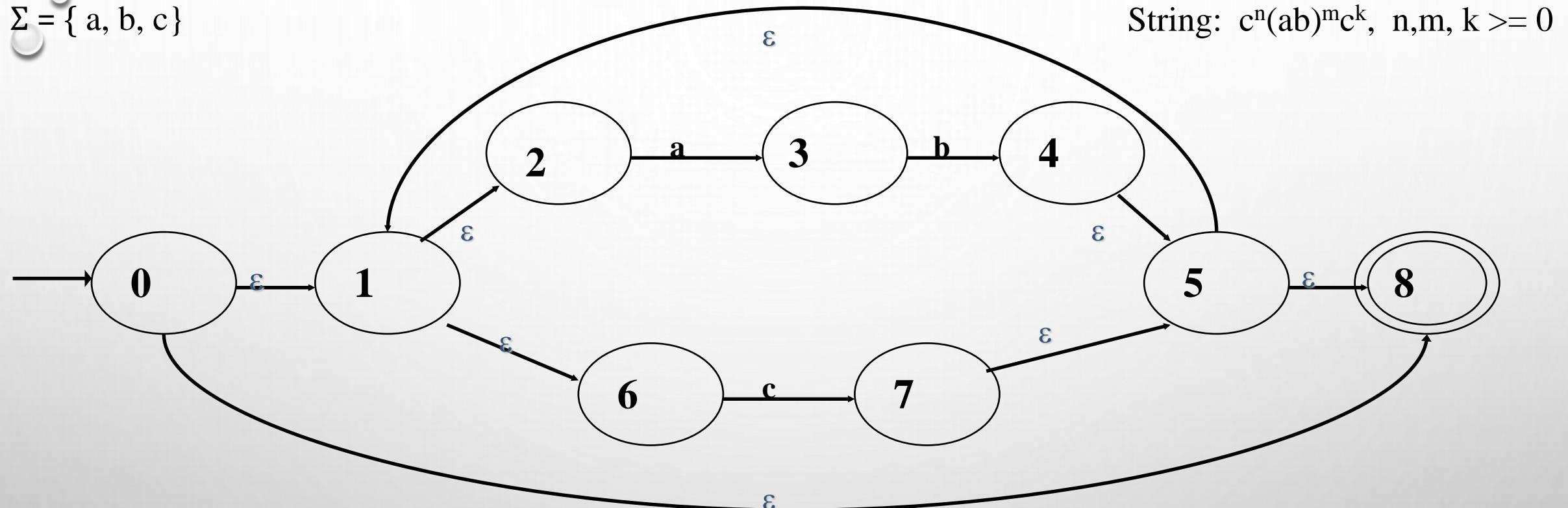
$$E = \{1, 2, 4, 5, 6, 7, 10\}$$

State

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

# CONVERTING NFA-E TO DFA –EXAMPLE-2

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



String:  $c^n(ab)^m c^k$ ,  $n, m, k \geq 0$

1<sup>st</sup> we calculate:  $\epsilon$ -closure(0)

$\epsilon$ -closure(0) = {0, 1, 2, 6, 8} (all states reachable from 0 on  $\epsilon$ -moves)

Let A={0, 1, 2, 6, 8} be a state of new DFA M

# CONVERSION EXAMPLE-2

2<sup>nd</sup>, we calculate : a :  $\epsilon$ -closure( $move(A, a)$ ) and  
 b :  $\epsilon$ -closure( $move(A, b)$ )  
 c :  $\epsilon$ -closure( $move(A, c)$ )

**a** :  $\epsilon$ -closure( $move(A, a)$ ) =  $\epsilon$ -closure( $move(\{0,1,2,6,8\}, a)$ )

adds {3} (since  $move(2, a) = 3$ )

From this we have :  $\epsilon$ -closure({3}) = {3} (since  $3 \rightarrow 3$  by  $\epsilon$ -moves)

Let B = {3} be a new state. Define  $\delta_M(A, a) = B$

**b** :  $\epsilon$ -closure( $move(A, b)$ ) =  $\epsilon$ -closure( $move(\{0,1,2,6,8\}, b)$ )

There is NO transition on b for all states 0,1,2,6 and 8

Define  $\delta_M(A, b) = \text{Reject}$

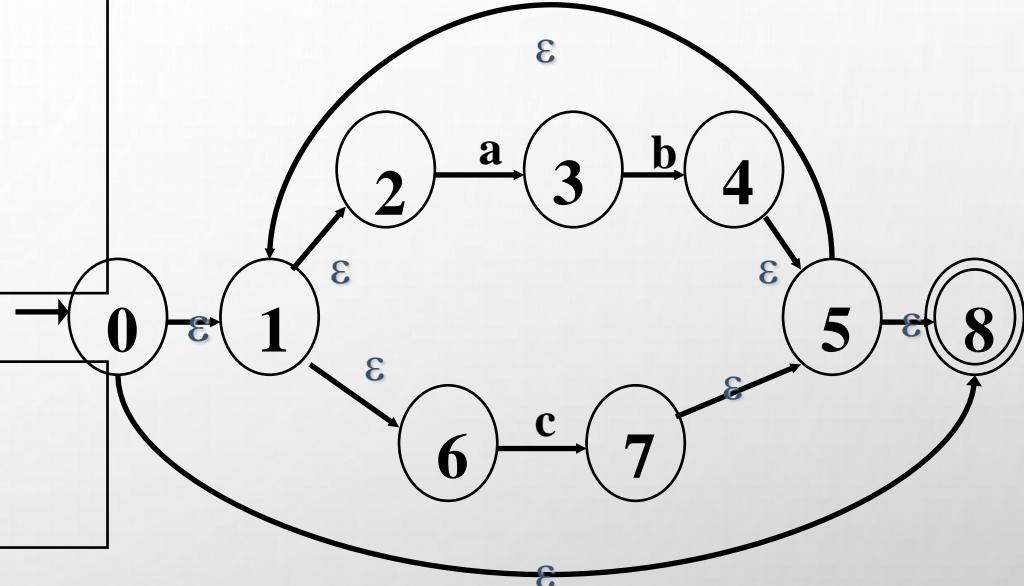
**c** :  $\epsilon$ -closure( $move(A, c)$ ) =  $\epsilon$ -closure( $move(\{0,1,2,6,8\}, c)$ )

adds {7} (since  $move(6, c) = 7$ )

From this we have :  $\epsilon$ -closure({7}) = {1,2,5,6,7,8}

(since  $7 \rightarrow 5 \rightarrow 8$ ,  $7 \rightarrow 5 \rightarrow 1 \rightarrow 2$ , and  $7 \rightarrow 5 \rightarrow 1 \rightarrow 6$  by  $\epsilon$ -moves)

Let C = {1,2,5,6,7,8} be a new state. Define  $\delta_M(A, c) = C$



# CONVERSION EXAMPLE-2

3<sup>rd</sup>, we calculate : a :  $\epsilon$ -closure(move(B, a)) and  
 b :  $\epsilon$ -closure(move(B, b))  
 c :  $\epsilon$ -closure(move(B, c))

**a** :  $\epsilon$ -closure(move(B, a)) =  $\epsilon$ -closure(move({3}, a))}

There is NO transition on **a** for state 3

Define  $\delta_M(B, a) = \text{Reject}$

**b** :  $\epsilon$ -closure(move(B, b)) =  $\epsilon$ -closure(move({3}, b))

adds {4} (since move(3, b) = 4)

From this we have :  $\epsilon$ -closure({4}) = {1,2,4,5,6,8}

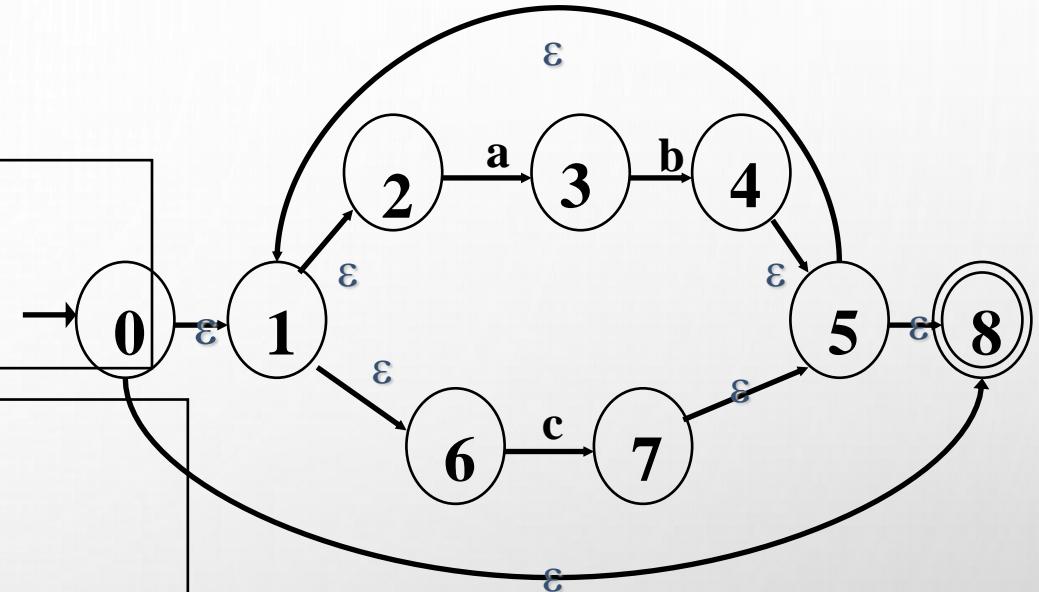
(since 4 → 5 → 8, 4 → 5 → 1 → 2, and 4 → 5 → 1 → 6 by  $\epsilon$ -moves)

Let D = {1,2,4,5,6,8} be a new state. Define  $\delta_M(B, b) = D$

**c** :  $\epsilon$ -closure(move(B, c)) =  $\epsilon$ -closure(move({3}, c))

There is NO transition on **c** for state 3

Define  $\delta_M(B, c) = \text{Reject}$



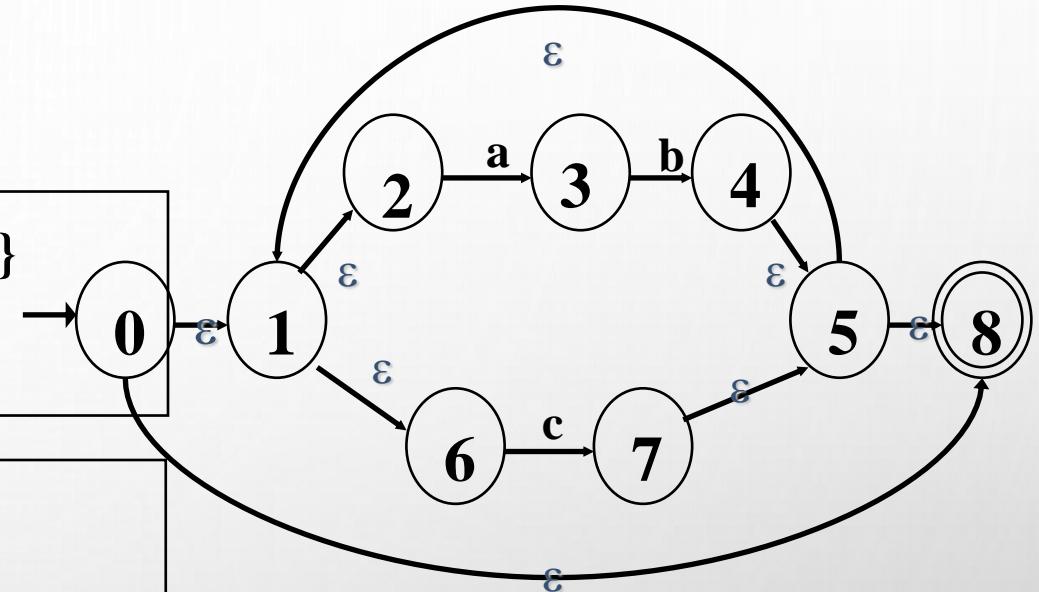
# CONVERSION EXAMPLE-2

4<sup>th</sup>, we calculate : a :  $\epsilon$ -closure( $move(C, a)$ ) and  
 b :  $\epsilon$ -closure( $move(C, b)$ )  
 c :  $\epsilon$ -closure( $move(C, c)$ )

**a** :  $\epsilon$ -closure( $move(C, a)$ ) =  $\epsilon$ -closure( $move(\{1,2,5,6,7,8\}, a)$ )  
 adds {3} (since  $move(2, a) = 3$ )  
 $\epsilon$ -closure({3}) = {3} Define  $\delta_M(C, a) = B$

**b** :  $\epsilon$ -closure( $move(C, b)$ ) =  $\epsilon$ -closure( $move(\{1,2,5,6,7,8\}, b)$ )  
 There is NO transition on **b** for all states 1,2,5,6, and 7  
 Define  $\delta_M(C, b) = \text{Reject}$

**c** :  $\epsilon$ -closure( $move(C, c)$ ) =  $\epsilon$ -closure( $move(\{1,2,5,6,7,8\}, c)$ )  
 adds {7} (since  $move(6, c) = 7$ )  
 From this we have :  $\epsilon$ -closure({7}) = {1,2,5,6,7,8}  
 (since  $7 \rightarrow 5 \rightarrow 8$ ,  $7 \rightarrow 5 \rightarrow 1 \rightarrow 2$ , and  $7 \rightarrow 5 \rightarrow 1 \rightarrow 6$  by  $\epsilon$ -moves)  
 Define  $\delta_M(C, c) = C$



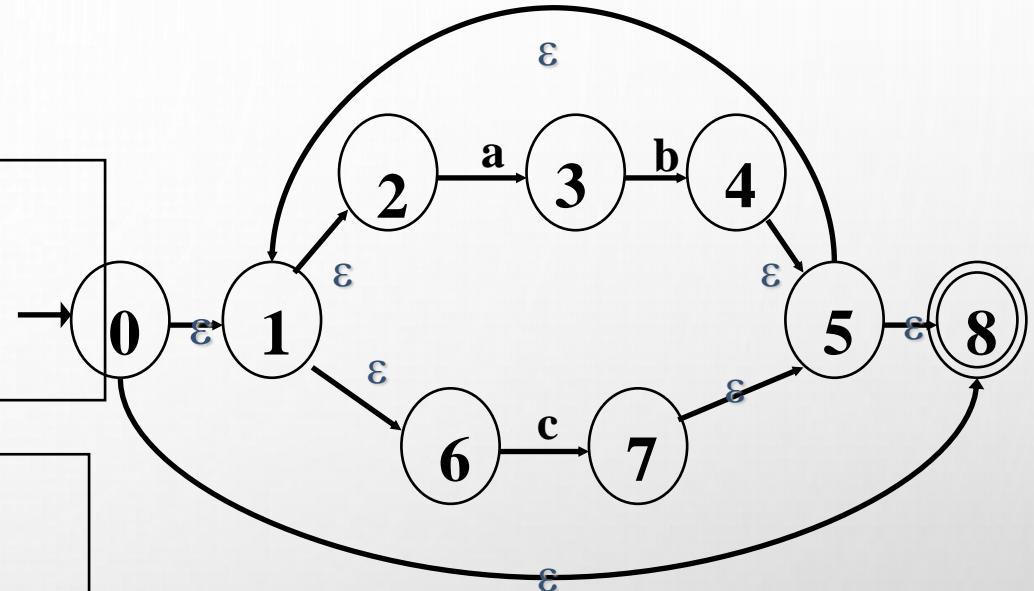
# CONVERSION EXAMPLE-2

5<sup>th</sup>, we calculate : a :  $\epsilon$ -closure( $move(D, a)$ ) and  
 b :  $\epsilon$ -closure( $move(D, b)$ )  
 c :  $\epsilon$ -closure( $move(D, c)$ )

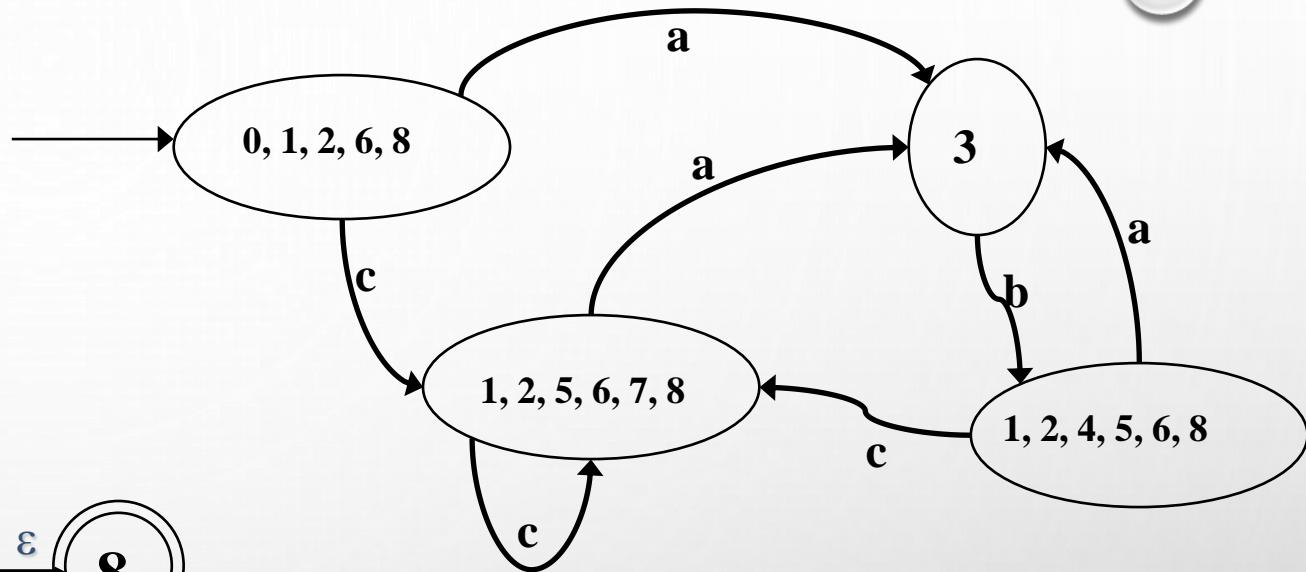
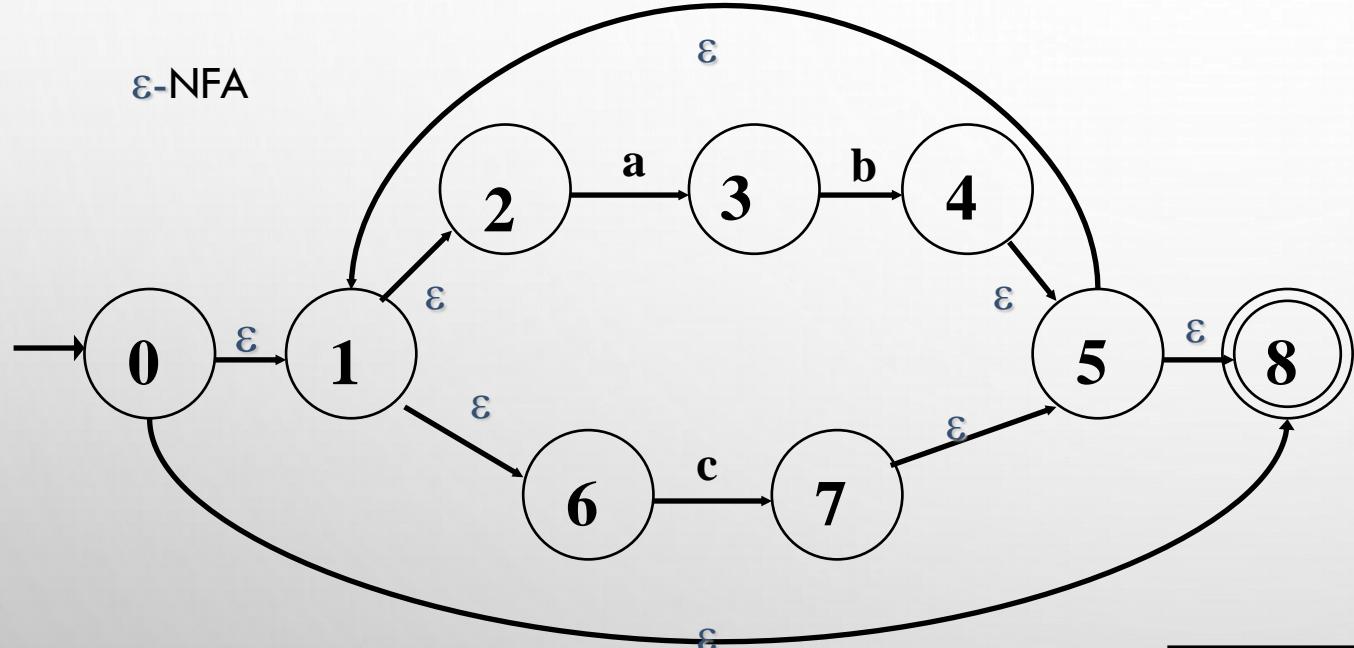
**a** :  $\epsilon$ -closure( $move(D, a)$ ) =  $\epsilon$ -closure( $move(\{1,2,4,5,6,8\}, a)$ )  
 adds {3} (since  $move(2, a) = 3$ )  
 $\epsilon$ -closure({3}) = {3} Define  $\delta_M(D, a) = B$

**b** :  $\epsilon$ -closure( $move(D, b)$ ) =  $\epsilon$ -closure( $move(\{1,2,4,5,6,8\}, b)$ )  
 There is NO transition on **b** for all states 1,2,4,5,6, and 8  
 Define  $\delta_M(D, b) = \text{Reject}$

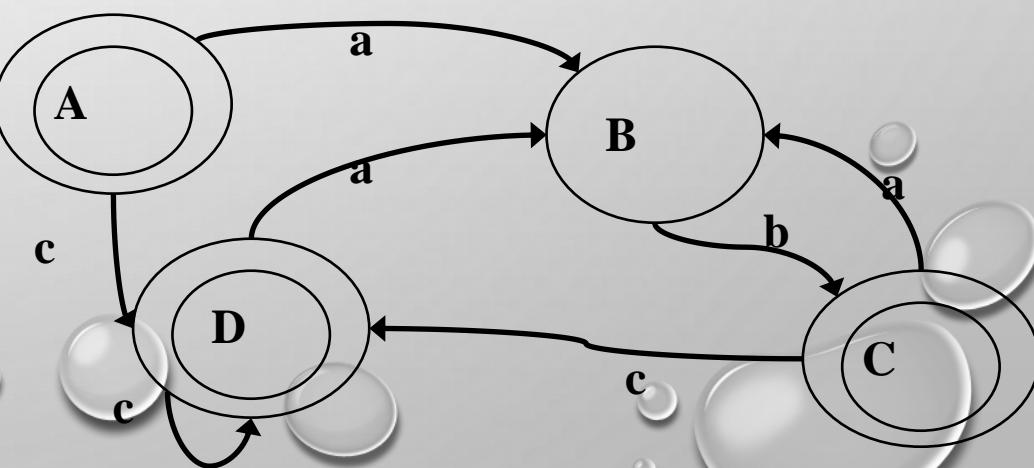
**c** :  $\epsilon$ -closure( $move(D, c)$ ) =  $\epsilon$ -closure( $move(\{1,2,4,5,6,8\}, c)$ )  
 adds {7} (since  $move(6, c) = 7$ )  
 From this we have :  $\epsilon$ -closure({7}) = {1,2,5,6,7,8}  
 (since  $7 \rightarrow 5 \rightarrow 8$ ,  $7 \rightarrow 5 \rightarrow 1 \rightarrow 2$ , and  $7 \rightarrow 5 \rightarrow 1 \rightarrow 6$  by  $\epsilon$ -moves)  
 Define  $\delta_M(D, c) = C$



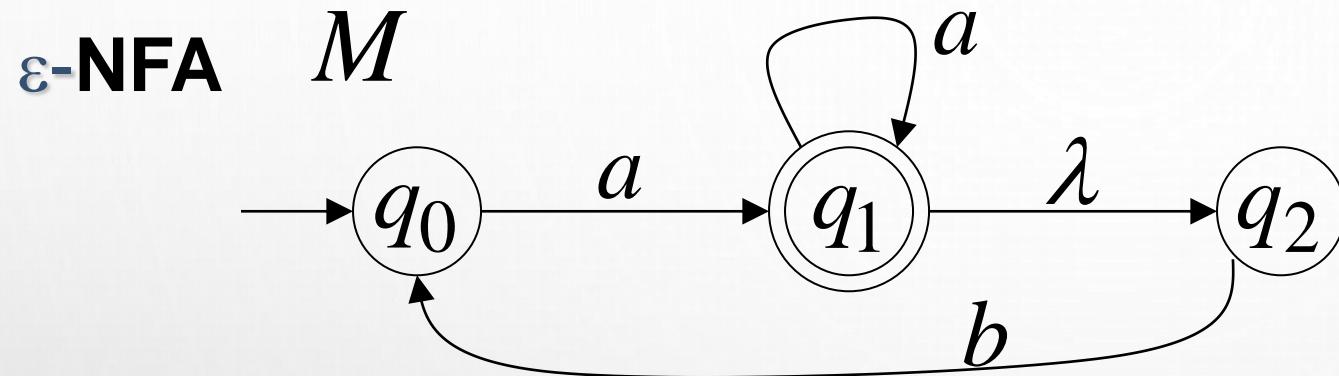
# THE RESULTING DFA M



FINAL States ?

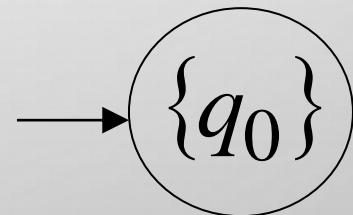


# CONVERSION: $\varepsilon$ -NFA TO DFA



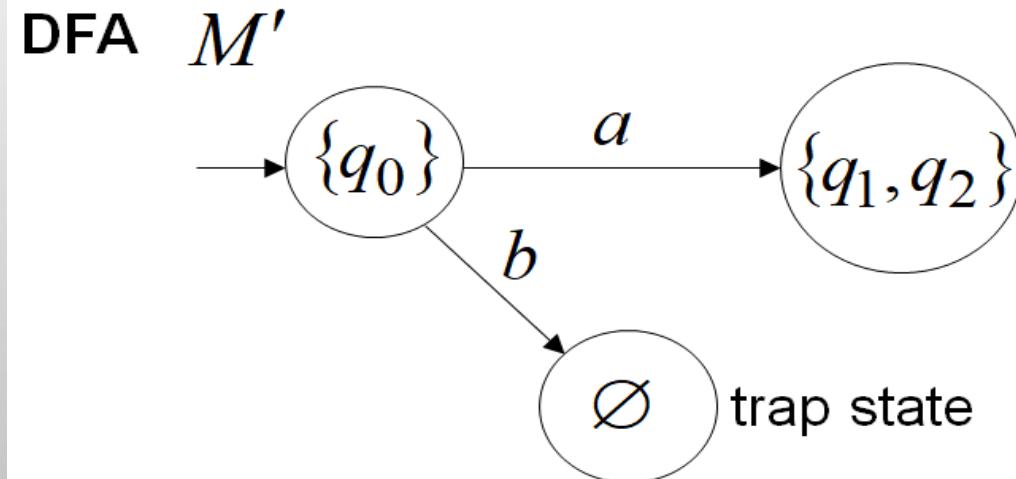
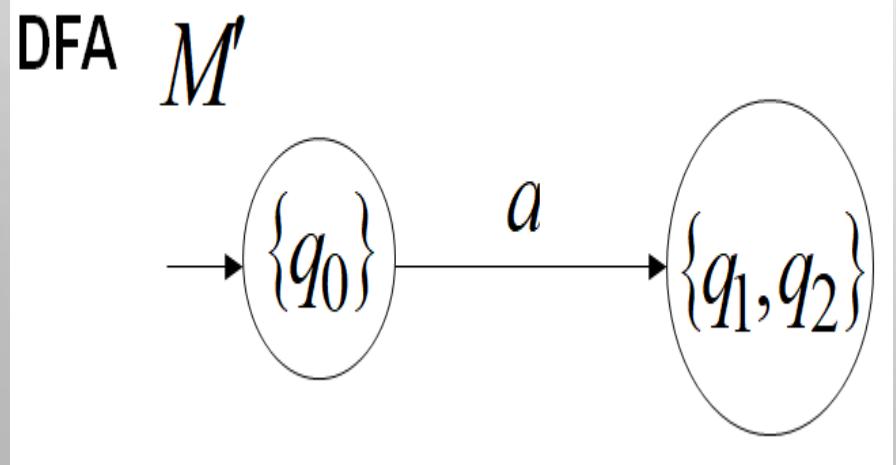
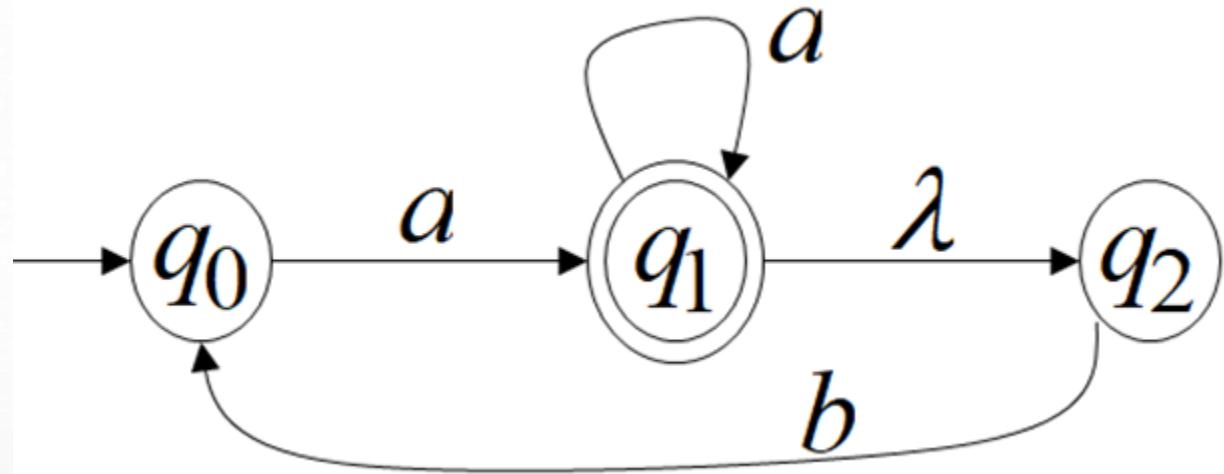
DFA  $M'$

$$\delta^*(q_0, \lambda) = \varepsilon\text{-closure}(q_0) = \{q_0\}$$



# Conversion: $\varepsilon$ -NFA to DFA..

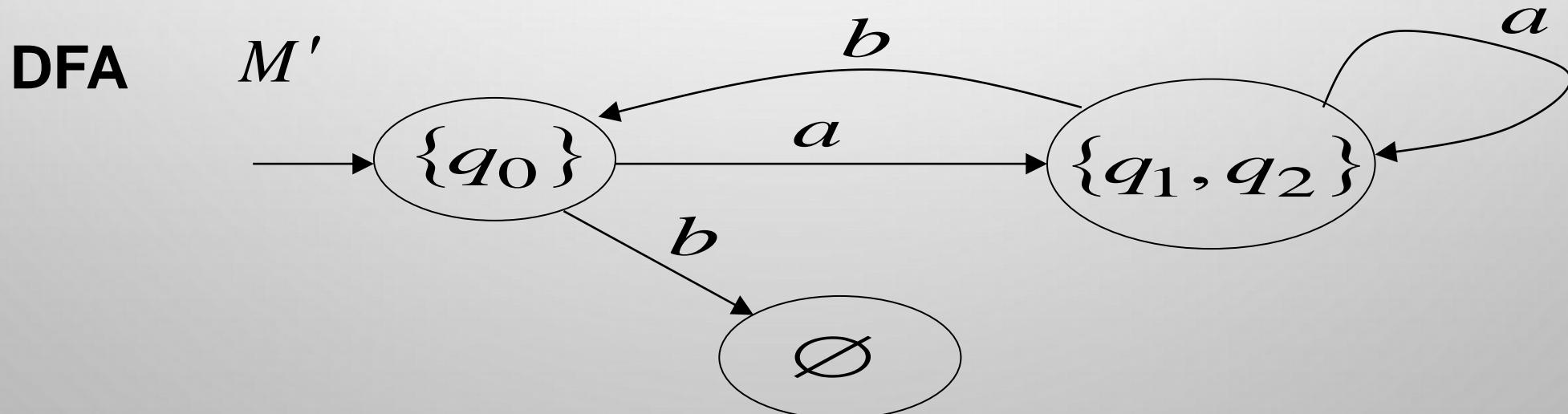
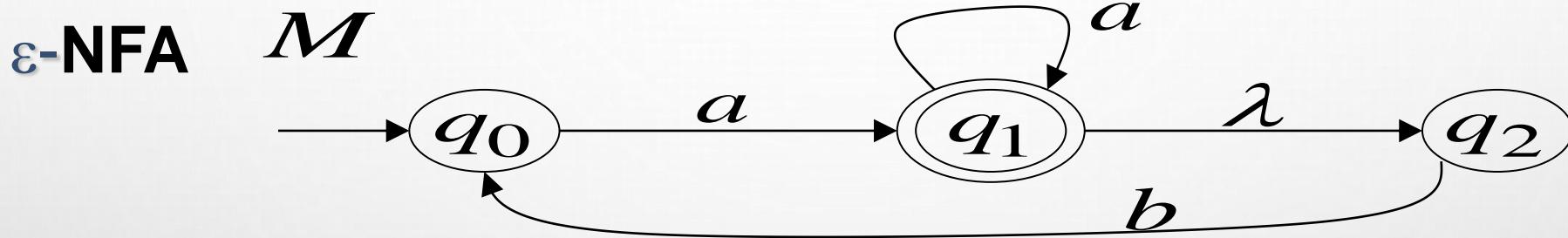
$$\begin{aligned}\delta^*(q_0, a) &= \varepsilon - \text{closure}(\text{move}(q_0, a)) = \{q_1, q_2\} \\ \delta^*(q_0, b) &= \emptyset\end{aligned}$$



# Conversion: $\epsilon$ -NFA to DFA.....

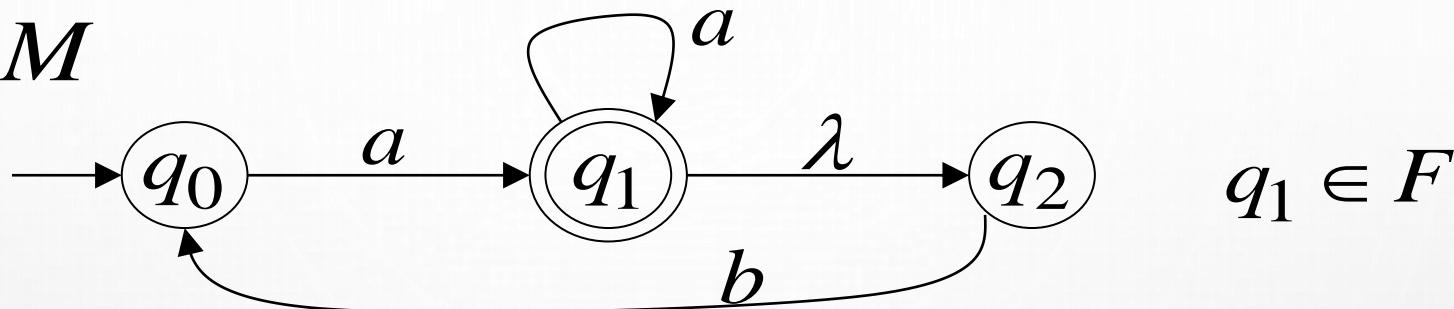
$$\delta^*(q_1, b) = \{q_0\} \quad \delta^*(q_2, b) = \{q_0\}$$

Union  $\{q_0\}$

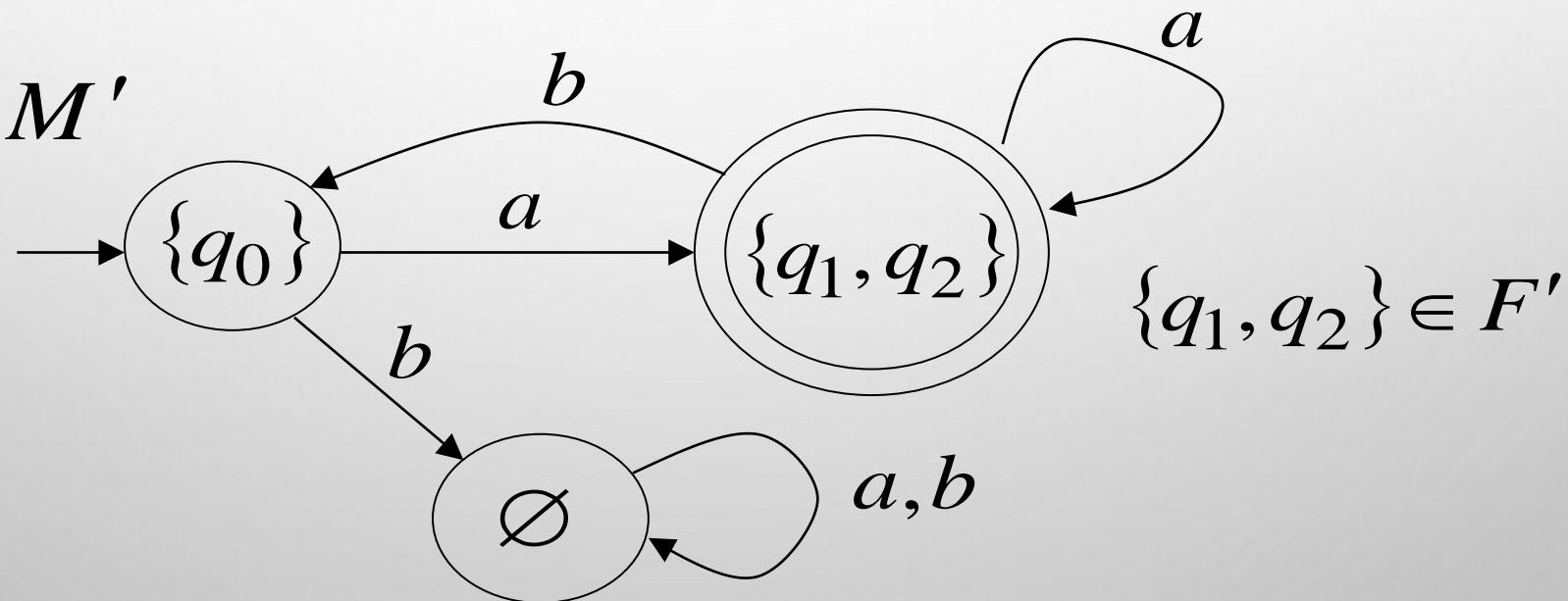


# Conversion: $\varepsilon$ -NFA to DFA

NFA  $M$



DFA  $M'$

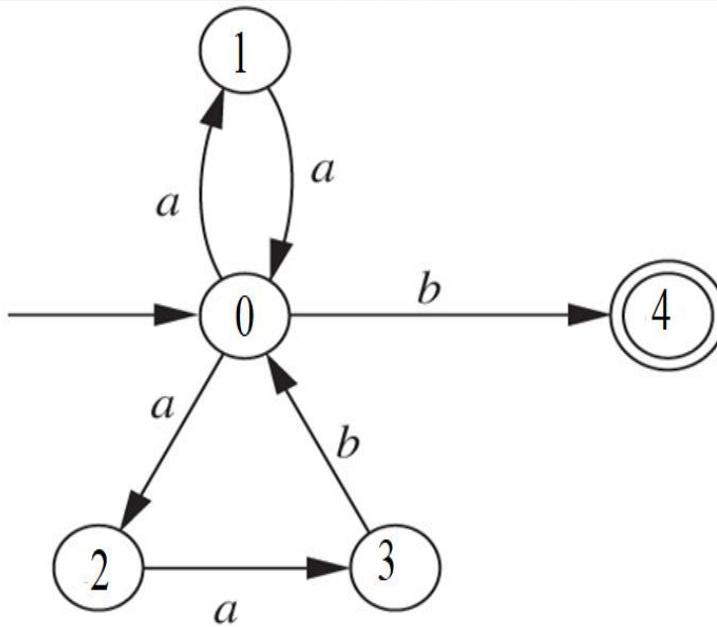


# NFA-to-DFA: No- $\lambda$ transition Subset Construction

1. The subset construction gives us  $2^5 = 32$  states; including 'not reachable states'.
2. Consider the 'reachable states from 0' (subsets of q) and defines transitions for all states.

1. Use transition table/transition diagram to convert an NFA to DFA

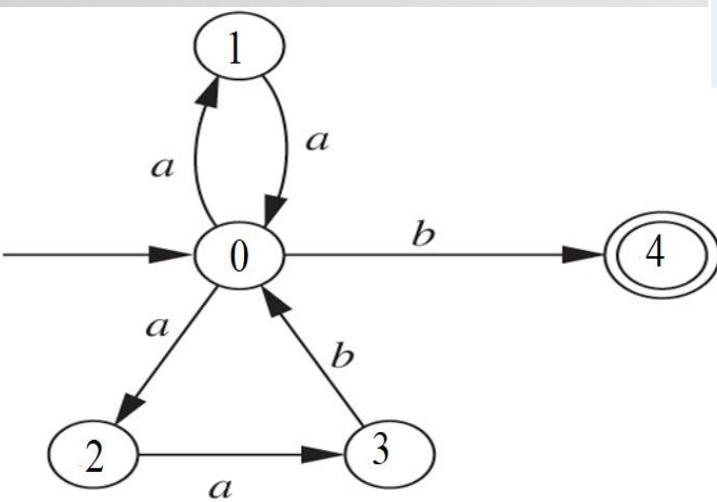
	a	b
0	{1, 2}	{4}
1	{0}	$\emptyset$
2	{3}	$\emptyset$
3	$\emptyset$	{0}
4	$\emptyset$	$\emptyset$



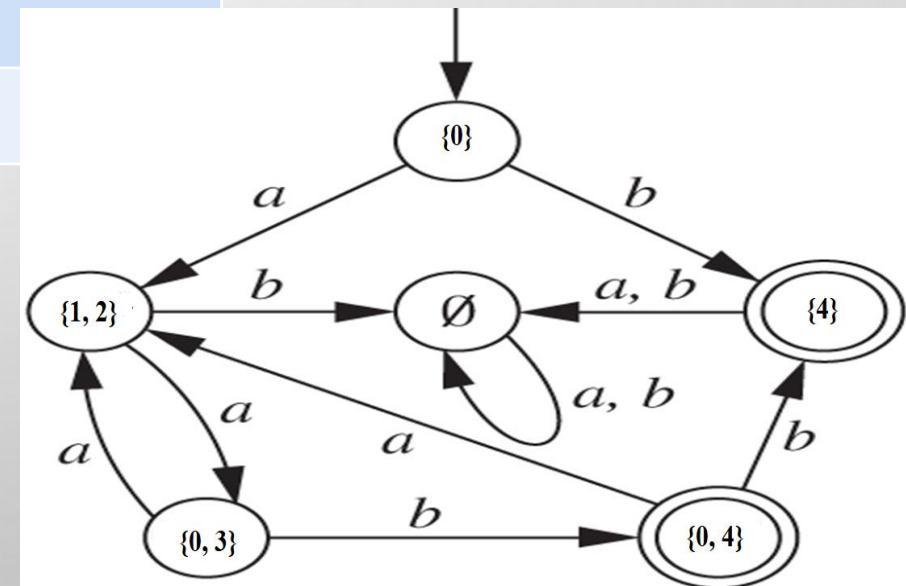
# NFA-to-DFA: No- $\lambda$ transition Subset Construction

1. Use transition table/transition diagram to convert an NFA to DFA

	a	b
0	{1, 2}	{4}
1	{0}	$\emptyset$
2	{3}	$\emptyset$
3	$\emptyset$	{0}
4	$\emptyset$	$\emptyset$

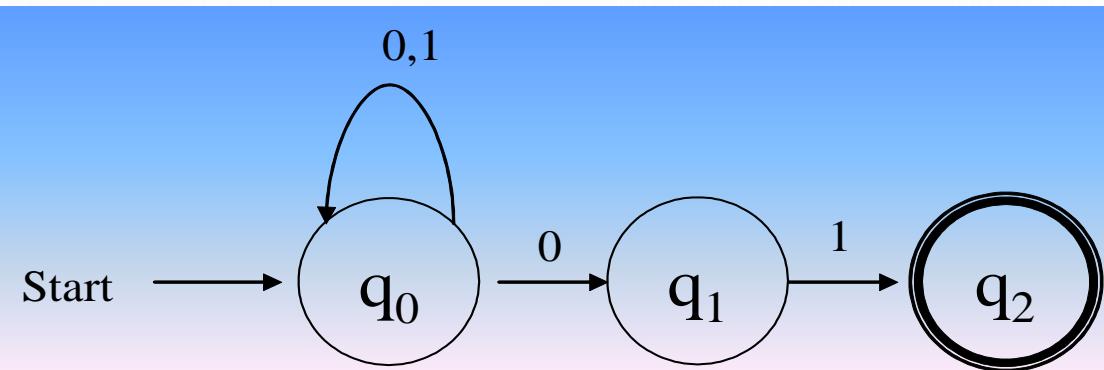


	a	b
$\rightarrow\{0\}$	{1, 2}	{4}
{1, 2}	{0, 3}	{ }
{0, 3}	{1, 2}	{0, 4}
*{0, 4}	{1, 2}	{4}
*{4}	{ }	{ }
{ }	{ }	{ }



# NFA-to-DFA: No- $\lambda$ transition

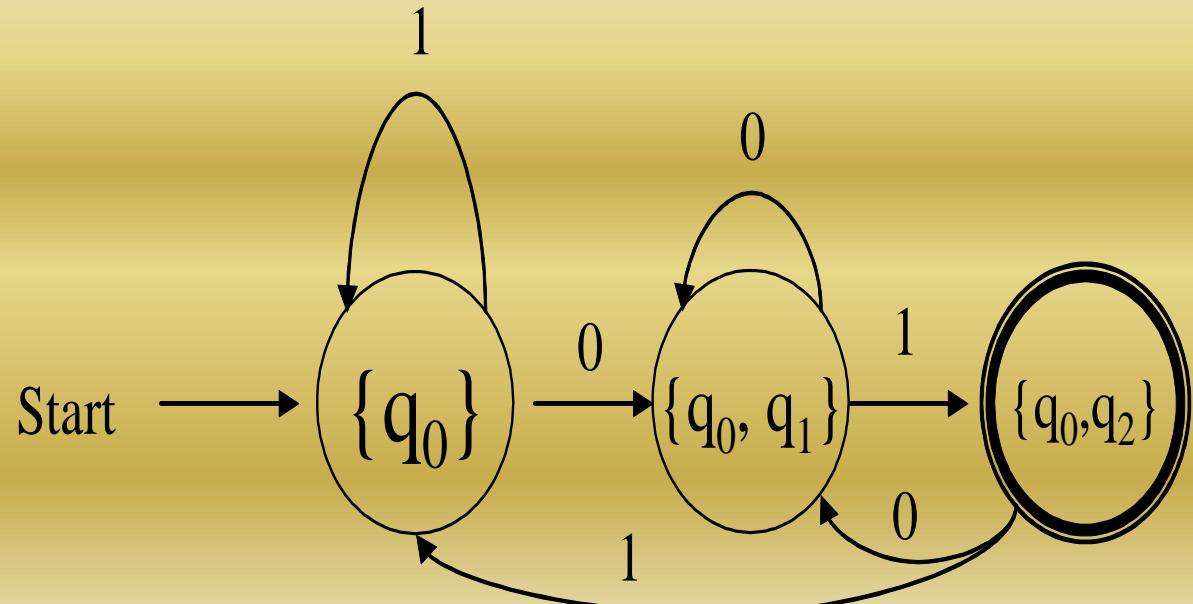
## Subset Construction



NFA

	0	1
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

DFA



# Convert the following NFAs to DFAs

Exercise-1

	a	b	$\lambda$
$q_0$	$\{q_0, q_1\}$	$\emptyset$	$\{q_2\}$
$q_1$	$\emptyset$	$\{q_1, q_2\}$	$\emptyset$
$q_2$	$\{q_2\}$	$\emptyset$	$\emptyset$

Exercise-2

	a	b	$\lambda$
$q_0$	$\{q_0, q_1\}$	$\emptyset$	$\{q_2\}$
$q_1$	$\emptyset$	$\{q_1, q_2\}$	$\{q_1, q_2\}$
$q_2$	$\emptyset$	$\emptyset$	$\emptyset$

Exercise-3

	a	b
$q_0$	$\{q_0, q_1\}$	$\emptyset$
$q_1$	$\emptyset$	$\{q_1, q_2\}$
$q_2$	$\{q_2\}$	$\emptyset$

# $\epsilon$ -NFA → NFA

Supplementary Material :  
Not in syllabus

$\epsilon$ -NFA N ⇒ NFA M construction:

Given N = (Q, Σ, δ, q<sub>0</sub>, F)

Construct M = (Q, Σ, δ', q<sub>0</sub>, F)

For all q ∈ Q and for all a ∈ Σ

$$\delta'(q, a) = \epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q), a))$$

For all q ∈ Q and for all a ∈ Σ

Add a transition from q to all reachable states in  $\epsilon\text{-closure}(\delta(\epsilon\text{-closure}(q), a))$   
with label a.

Remove all  $\lambda$  transitions

Remove all disconnected state (Q<sub>D</sub> - Set of all disconnected/unreachable States), if any.

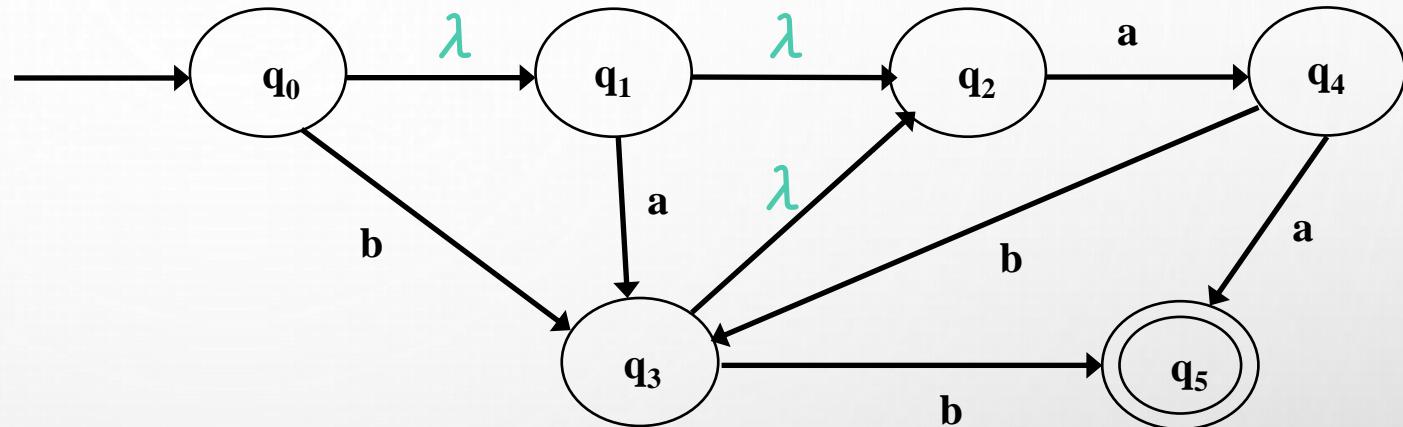
$$Q = Q - Q_D$$

# $\epsilon$ -NFA $\rightarrow$ NFA: EXAMPLE

Supplementary Material :  
Not in syllabus

Given  $\epsilon$ -NFA  $N = (Q, \Sigma, \delta, q_0, \{q_5\})$

	a	b	$\lambda$
$q_0$	$\phi$	$q_3$	$q_1$
$q_1$	$q_3$	$\phi$	$q_2$
$q_2$	$q_4$	$\phi$	$\phi$
$q_3$	$\phi$	$q_5$	$q_2$
$q_4$	$q_5$	$q_3$	$\phi$
$q_5$	$\phi$	$\phi$	$\phi$



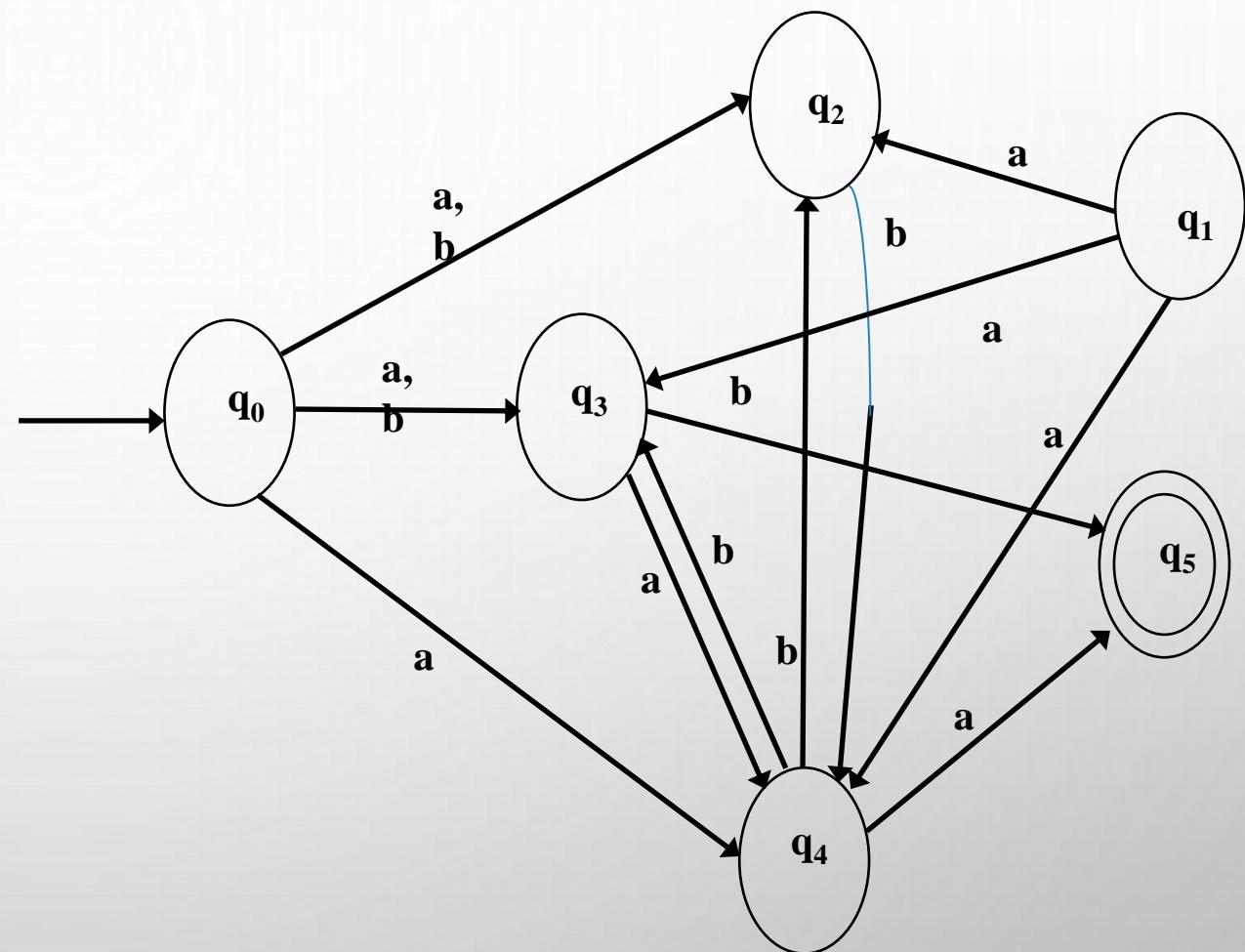
$$\begin{aligned}
 \delta'(q_0, a) &= \text{closure}(\delta(\text{closure}(q_0), a)) \\
 &= \text{closure}(\delta(\{q_0, q_1, q_2\}, a)) \\
 &= \text{closure}(\{q_3, q_4\}) \\
 &= \{q_2, q_3, q_4\}
 \end{aligned}$$

$$\begin{aligned}
 \delta'(q_0, b) &= \text{closure}(\delta(\text{closure}(q_0), b)) \\
 &= \text{closure}(\delta(\{q_0, q_1, q_2\}, b)) \\
 &= \text{closure}(\{q_3\}) \\
 &= \{q_2, q_3\}
 \end{aligned}$$

# $\epsilon$ -NFA $\rightarrow$ NFA: EXAMPLE

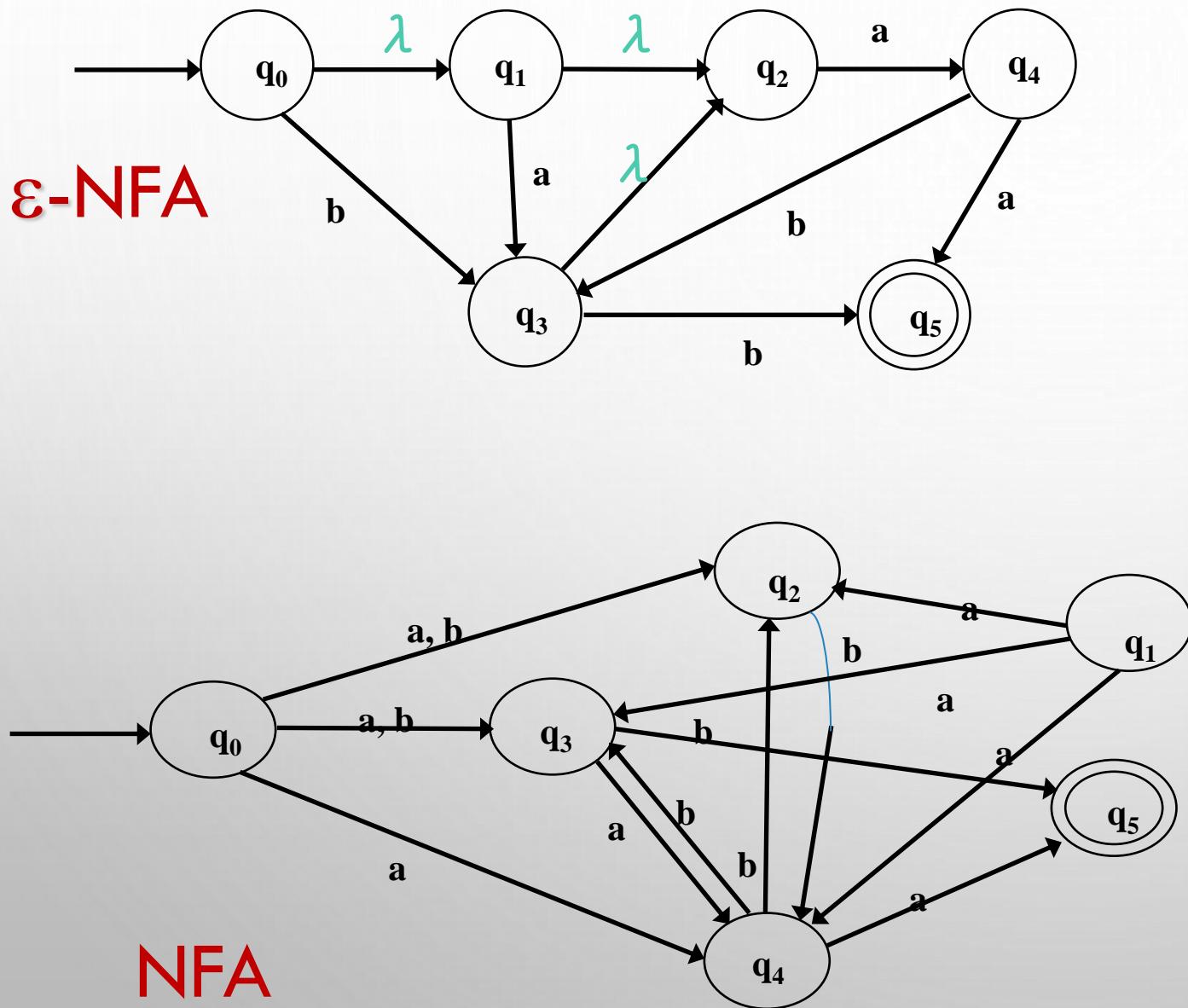
Supplementary Material :  
Not in syllabus

	a	b
q <sub>0</sub>	{q <sub>2</sub> , q <sub>3</sub> , q <sub>4</sub> }	{q <sub>2</sub> , q <sub>3</sub> }
q <sub>1</sub>	{q <sub>2</sub> , q <sub>3</sub> , q <sub>4</sub> }	$\phi$
q <sub>2</sub>	q <sub>4</sub>	$\phi$
q <sub>3</sub>	q <sub>4</sub>	q <sub>5</sub>
q <sub>4</sub>	q <sub>5</sub>	{q <sub>2</sub> , q <sub>3</sub> }
q <sub>5</sub>	$\phi$	$\phi$

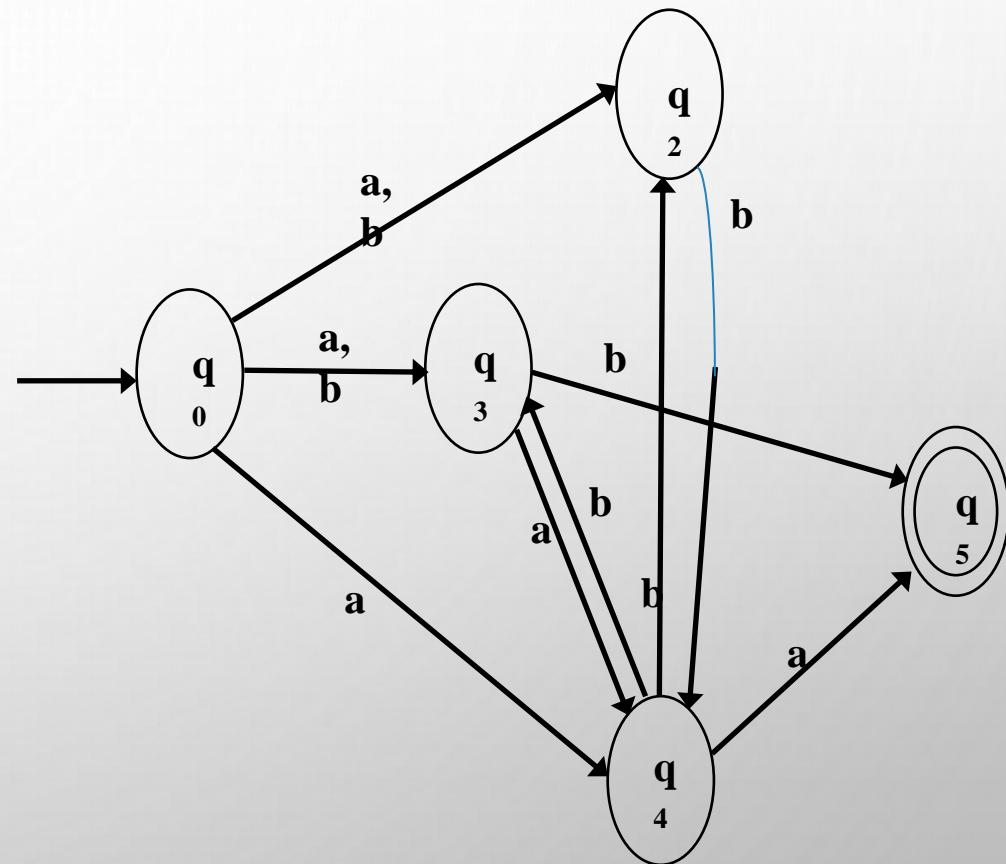


# $\epsilon$ -NFA $\rightarrow$ NFA: EXAMPLE

Supplementary Material :  
Not in syllabus



- $q_1$  is useless state and can be removed.



# Exercise: Remove $\lambda$ Transitions

Supplementary Material :  
Not in syllabus

	a	b	$\lambda$
$q_0$	$\{q_0, q_1\}$	$\phi$	$\{q_2\}$
$q_1$	$\phi$	$\{q_1, q_2\}$	$\phi$
$q_2$	$\{q_2\}$	$\phi$	$\phi$

$q_0$  is initial state.

$\{q_2\}$  is the set of final states.

	a	b	$\lambda$
$q_0$	$\{q_0, q_1\}$	$\phi$	$\{q_2\}$
$q_1$	$\phi$	$\{q_1, q_2\}$	$\{q_1, q_2\}$
$q_2$	$\phi$	$\phi$	$\phi$

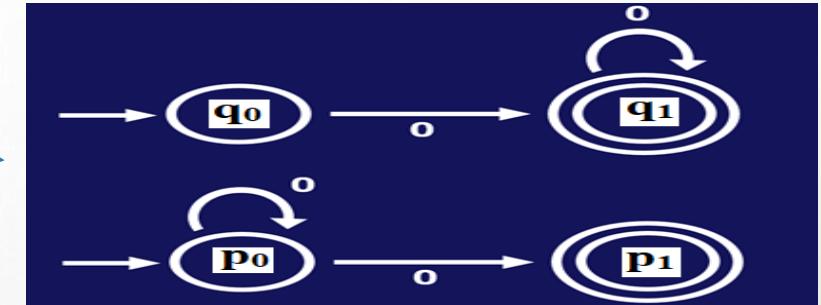
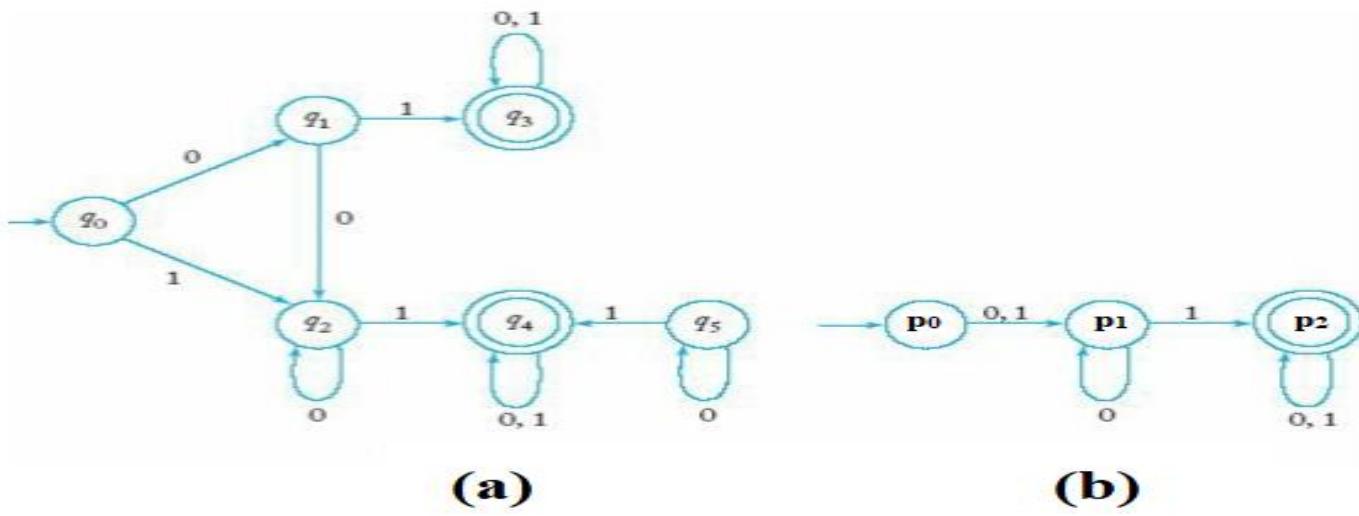
$q_0$  is initial state.

$\{q_2\}$  is the set of final states.

# Minimizing the number of states in a DFA

- Any DFA accepts a **unique language** (regular language).
- **Theorem:** For every regular language L, a **unique minimal DFA M** exists such that  $L = L(M)$ .
  - It is not true for NFAs.
  - Two minimal NFAs for the same language
- For **a given language**, there exist many DFA's-
  - Different DFAs **for the same language** have a different number of states.
  - Considerable difference in the number of states of such equivalent DFAs.

$$L = \{w_1 0^n 1 w_2 : n \geq 0, w_1 \in \{0, 1\}, w_2 \in \{0, 1\}^*\}$$



- The DFA (a) consists of inaccessible/unreachable/useless states. ( $q_5$  plays no role).
  - The inaccessible states can be removed along with all transitions without any effect.
- $q_1$  and  $q_2$  are images of each other (Why).
- $q_3$  and  $q_4$  are images of each other.

# Minimizing the number of states in a DFA

- Minimize a DFA
  - “Minimal”? - Minimum Number Of States.
  - Find an equivalent DFA that has the least possible number of states (minimum DFA).
- “Unique”?
  - Unique up to renaming of states.
  - i.e., has the same shape. *isomorphic*.
- Why Minimization?
  - To write a program (e.g., A compiler) or to design hardware using DFA, then there may be a significant benefit in minimizing it before implementing it.

# Indistinguishable States: Key Idea

- Let  $M$  be a DFA.
- For all  $w \in \Sigma^*$ , two states  $p$  and  $q$  are called *indistinguishable* if
  - $\delta^*(p, w) \in F$  implies  $\delta^*(q, w) \in F$
  - Or
  - $\delta^*(p, w) \notin F$  implies  $\delta^*(q, w) \notin F$
- $p \equiv q$ .
- The “*indistinguishable*” relation is an “*equivalent*” relation.

- Let  $M$  be a DFA.

For states 1 and 2,  $w = a$ ,  $w = b$

$$\begin{aligned}\delta^*(1, a) &= 2 \notin F \text{ and } \delta^*(2, a) = 6 \notin F \\ \delta^*(1, b) &= 7 \notin F \text{ and } \delta^*(2, b) = 3 \notin F\end{aligned}$$

It means states 1 and 2 are *indistinguishable* for all strings of length one.

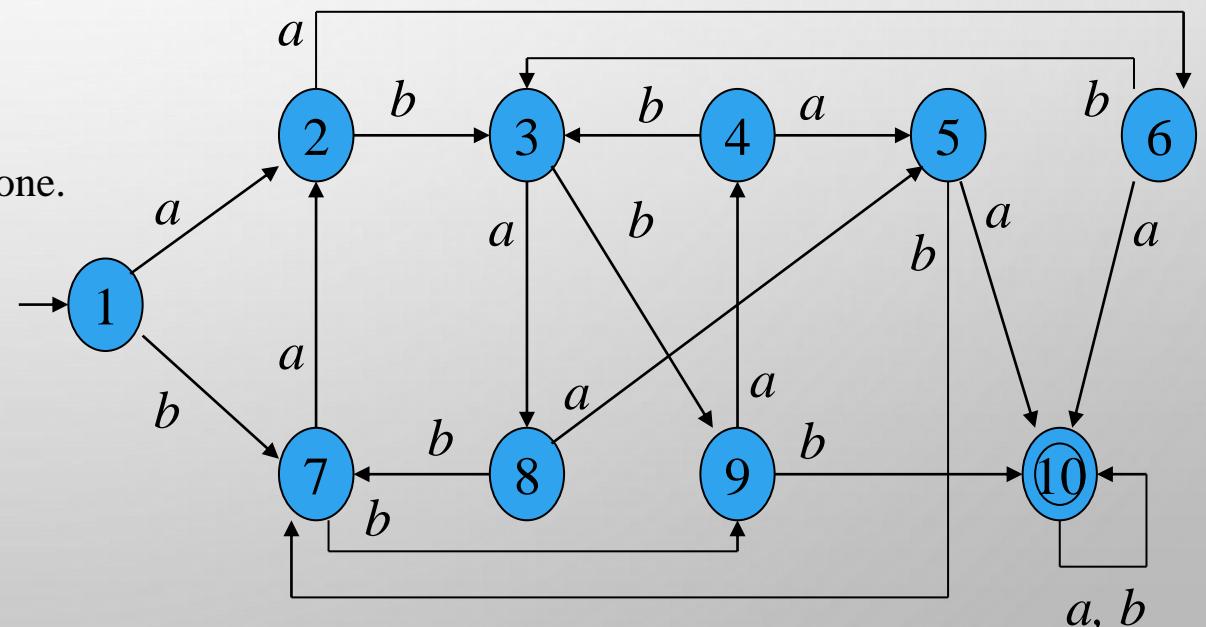
For states 1 and 2 and  $w = aa$

$$\delta^*(1, aa) = 6 \notin F \text{ and } \delta^*(2, aa) = 10 \in F$$

It means states 1 and 2 are **not indistinguishable** for  $w = aa \in \Sigma$ .

Check all pairs for  $w \in \Sigma$

$$\begin{aligned}(1,2), (1,3), (1,4), (1,5), (1,6), (1,7), (1,8), (1,9), (1,10) \\ (2,3), (2,4), \dots, (2,10)\end{aligned}$$



# DISTINGUISHABLE STATES

- Let  $M$  be a DFA. If there exists some string  $w \in \Sigma^*$  such that

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

- Then, the states  $p$  and  $q$  are said to be distinguishable by a string  $w$ .

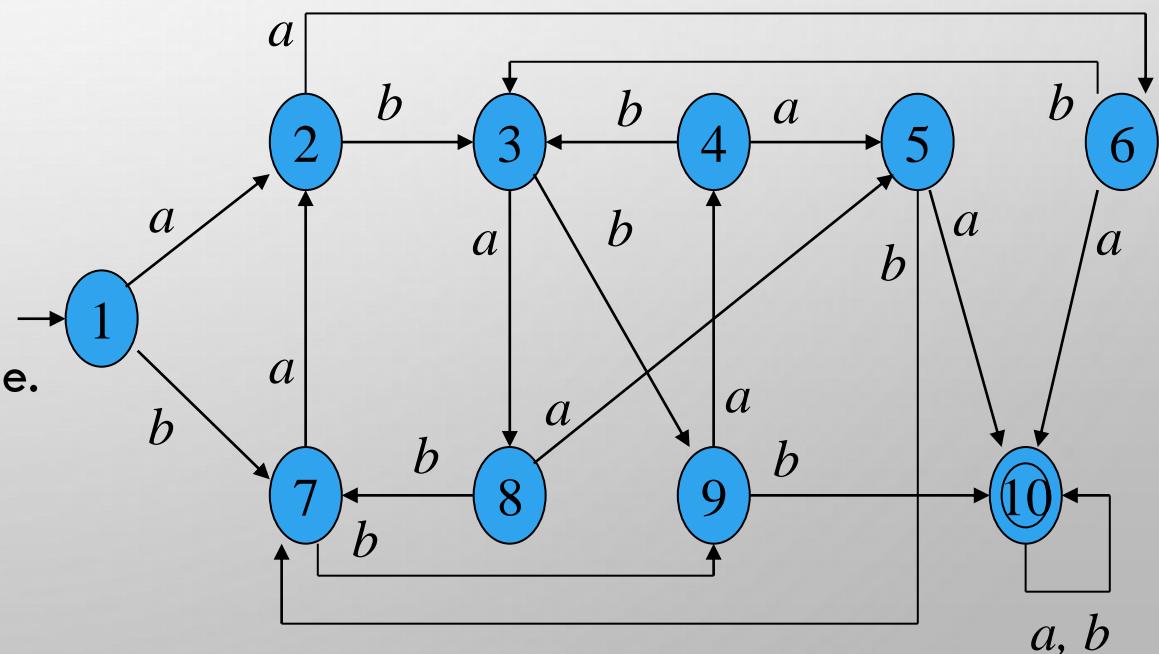
- The “distinguishable” relation is also known as the “not equivalent” relation.

For states 4 and 5

$$\delta^*(4, a) = 5 \notin F \text{ and } \delta^*(5, a) = 10 \in F$$

it means states 4 and 5 are distinguishable for  $w = a \in \Sigma$ .

i.e. states 4 and 5 are distinguishable for strings of length one.



# Indistinguishable/Distinguishable Relation

- **Indistinguishable (“ $\equiv$ ”) is an equivalence relation.**

- $p \equiv p$  (reflexive)
- $p \equiv q \Rightarrow q \equiv p$  (symmetric)
- $p \equiv q$  and  $q \equiv r \Rightarrow p \equiv r$  (transitive)

- Let  $M$  be a DFA.

- Two states  $p$  and  $q$  are called *indistinguishable* if

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

OR

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

For all  $w \in \Sigma^*$ .

- $p \equiv q$ .

- **The distinguishable relation is not an equivalence relation.**

- If  $p$  and  $q$  are distinguishable and  $q$  and  $r$  are also distinguishable
  - Then  $p$  and  $r$  are not necessarily distinguishable.

- Let  $M$  be a DFA.

- If there exists some string  $w \in \Sigma^*$  such that

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

OR

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

- Then the states  $p$  and  $q$  are said to be distinguishable by a string  $w$ .

# Identify equivalence classes using the relation $\equiv_n$

- Let  $p$  and  $q$  be states.

Define the relation  $P \equiv_n q$

- To mean that  $p$  and  $q$  are not distinguishable (indistinguishable) by any string  $w$  with  $|w| \leq n$ . That is, **no sequence of up to  $n$  moves** will distinguish  $p$  and  $q$ .
    - This, too, is an equivalence relation.
  - Clearly,  $p \equiv_0 q$  if and only if  $p, q \in f$  or  $p, q \notin f$ .
  - The following are also obvious:
    - For all  $n \geq 0$ , if  $p \equiv q$ , then  $p \equiv_n q$ .
    - For all  $n \geq 0$ , if  $p \equiv_n q$  then  $p \equiv q$ .
    - For all  $n (\geq 1)$ , if  $p \equiv_n q$ , then  $p \equiv_{n-1} q$ .
      - But it is not the case that if  $p \equiv_{n-1} q$ , then  $p \equiv_n q$ .
- Theorem:** For any two states  $p$  and  $q$  and any  $n \geq 1$ ,  $p \equiv_n q$  if and only if
    - $P \equiv_{n-1} q$ , and
    - For all  $a \in \Sigma$ ,  $\delta(p, a) \equiv_{n-1} \delta(q, a)$ .
  - This theorem characterizes the relation  $\equiv_n$  entirely in terms of the relation  $\equiv_{n-1}$  and the function  $\delta$ .
  - It leads to an algorithm to determine which states are indistinguishable.

# Minimizing a DFA

- The problem is determining “**Which DFA states are distinguishable and indistinguishable ?**”.
- To minimize a DFA,
  - **Remove unreachable (inaccessible) states**
    - States which are not reachable from the initial state (s).
    - States which are also not connected with final states.
  - **Identify equivalence classes of indistinguishable states**
    - Replace the equivalence class with a single state.

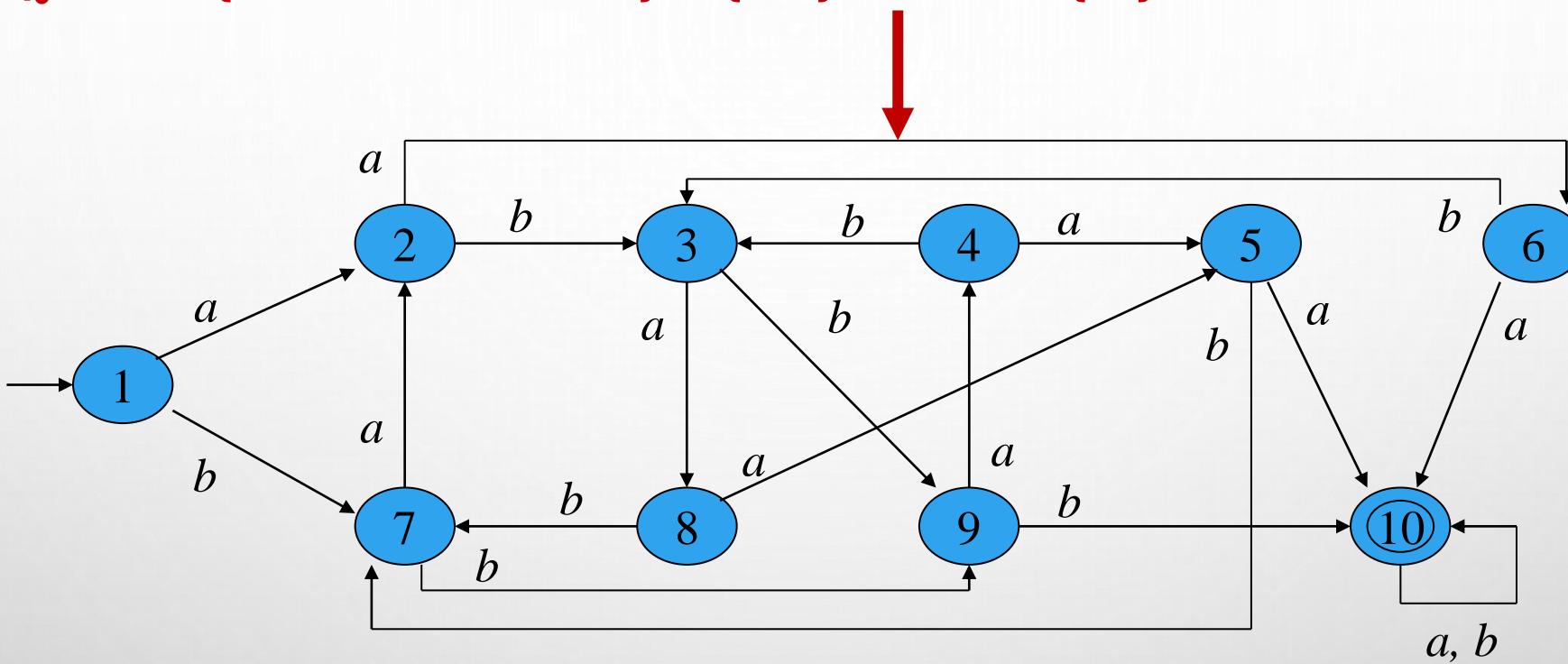
# The Minimization Algorithm

Given a DFA  $M = (Q, \Sigma, \delta, q_0, F)$ ; find the minimum DFA  $M'$

1. Remove inaccessible/unreachable states.
  2. Initialize the partition;  $\pi_0 = \{F, Q-F\}$
  3. For  $n = 1, 2, \dots,$ 
    - Compute the equivalence class  $\pi_k$  using  $\pi_{k-1}$ .
      - For all sets  $S \in \pi_{k-1}$ 
        - For all  $q_1, q_2 \in S$ 
          - For all  $a \in \Sigma$ 
            - If  $\delta(q_1, a)$  and  $\delta(q_2, a)$  make a transition to the states of the **same set** of  $\pi_{k-1}$ 
              - $q_1$  and  $q_2$  are  $k+1$  indistinguishable.
              - Otherwise,  $q_1$  and  $q_2$  are  $k+1$  distinguishable. Then, split the set  $S$
      - Repeat step 3 until, for some  $n$ ,  $\pi_{n-1} = \pi_n$
4. Replace each equivalence class of states with a single state.
  - The initial state will be a set consisting of the initial state.
  - The final state will be a set consisting of at least one final state.
- This must eventually happen.
  - Why?

# DFA minimization: Example-1

$$M = (Q, \Sigma, \delta, q_0, F) = (\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}, \{a, b\}, \delta, 1, \{10\})$$



Step 1: Remove useless states.

Step 2: Initialize the partition  $\pi_0$

$$\pi_0 = \{Q - F, F\}$$

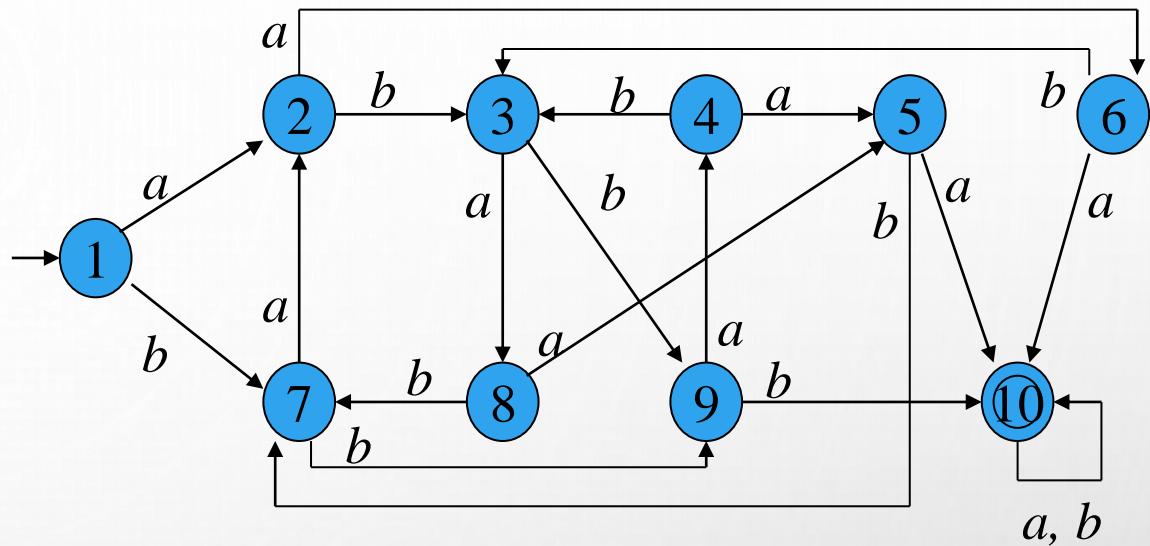
$$\pi_0 = \{\{1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{10\}\}$$

# DFA minimization-Example-1:Construct $\pi_1$ partition

$$\pi_0 = \{\{1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{10\}\}$$

$\delta$	1	2	3	4	5	6	7	8	9	10
$a$	2	6	8	5	10	10	2	5	4	10
$b$	7	3	9	3	7	3	9	7	10	10

- a-transitions
  - Apply the “a-transitions” to each class, the next states are  $\{2, 6, 8, 5, \textcolor{red}{10, 10}, 2, 5, 4\}, \{10\}$
  - The **a-transitions indicate** that 5 and 6 are in a different class than 1, 2, 3, 4, 7, 8, and 9.
  - The **a-transitions split**  $\{\{1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{10\}\}$  into **{1, 2, 3, 4, 7, 8, 9}, {5, 6}** and **{10}** i.e.
- $\{\{1, 2, 3, 4, 7, 8, 9\}, \{\textcolor{red}{5, 6}\}, \{10\}\}$
- b-transitions
  - Apply the “b-transitions” to each class, the next states are  $\{7, 3, 9, 3, 7, 3, 9, 7, \textcolor{red}{10}\}, \{10\}$
  - The **b-transitions indicate** that 9 is in a different class than 1, 2, 3, 4, 5, 6, 7, and 8.
  - The **b-transitions further split**  $\{\{1, 2, 3, 4, 7, 8, 9\}, \{\textcolor{red}{5, 6}\}, \{10\}\}$  into **{1, 2, 3, 4, 7, 8}, {5, 6}, {9}, {10}** i.e.
- The  $\Pi_1$  partition is  $\Pi_1 = \{\{1, 2, 3, 4, 7, 8\}, \{\textcolor{red}{5, 6}\}, \{\textcolor{red}{9}\}, \{10\}\}$
- $\pi_1 \neq \pi_0$ . compute  $\pi_2$



# EXAMPLE-1: THE $\Pi_2$ PARTITION

$$\pi_1 = \{ \{1, 2, 3, 4, 7, 8\}, \{5, 6\}, \{9\}, \{10\} \}$$

- apply the  $a$ -transitions to these classes:

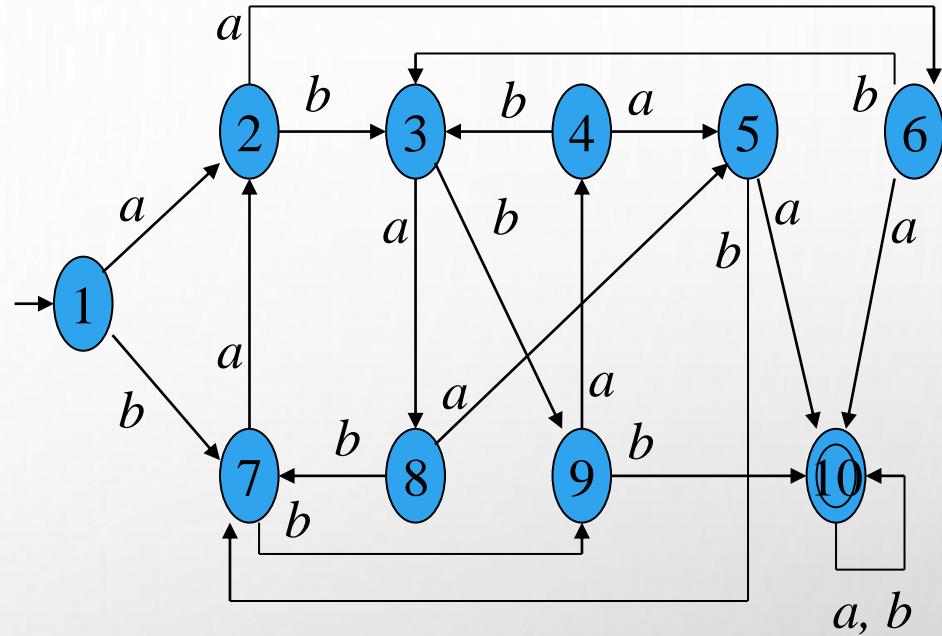
$$\{2, \textcolor{red}{6}, 8, \textcolor{red}{5}, 2, \textcolor{red}{5}\}, \{10, 10\}, \{4\}, \{10\}$$

- the  $a$ -transitions split  $\{1, 2, 3, 4, 7, 8\}$  into **{1, 3, 7} and {2, 4, 8}**.

- apply the  $b$ -transitions to these classes:

$$\{7, 3, \textcolor{red}{9}, 3, \textcolor{red}{9}, 7\}, \{7, 3\}, \{10\}, \{10\}$$

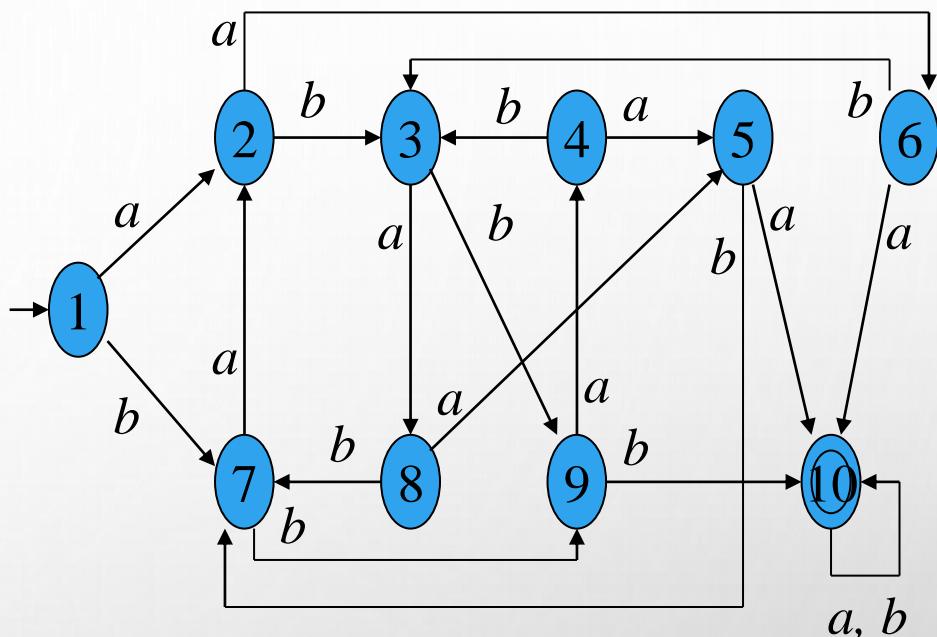
- the  $b$ -transitions split further into **{1}, {2, 4, 8} and {3, 7}**.



- $\pi_2 = \{\{1\}, \{3, 7\}, \{2, 4, 8\}, \{5, 6\}, \{9\}, \{10\}\}$
- $\pi_1 = \{ \{1, 2, 3, 4, 7, 8\}, \{5, 6\}, \{9\}, \{10\} \}$
- $\pi_2 \neq \pi_1$ . compute  $\pi_3$

# EXAMPLE-1: THE $\Pi_3$ PARTITION

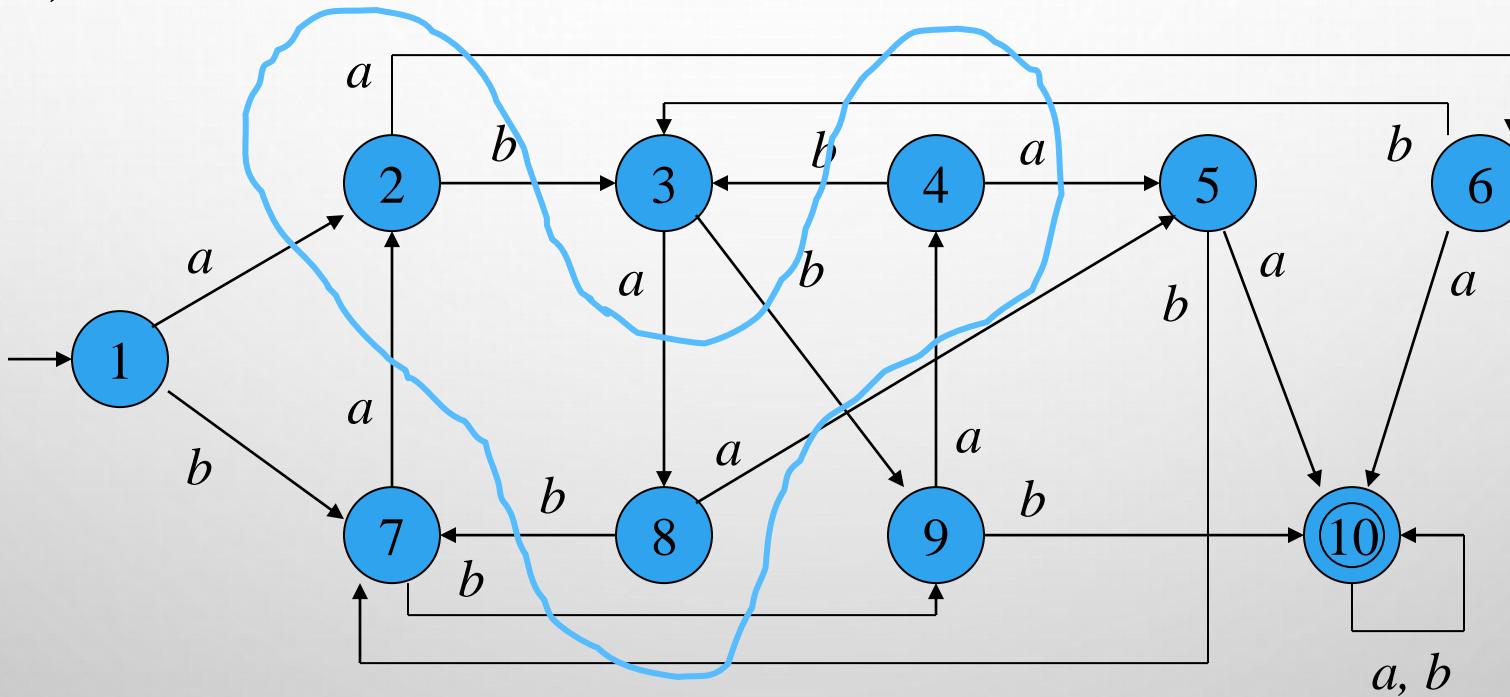
- $\pi_2 = \{\{1\}, \{3, 7\}, \{2, 4, 8\}, \{5, 6\}, \{9\}, \{10\}\}$
- apply the  $a$ -transitions to these classes:  
 $\{2\}, \{8, 2\}, \{6, 5, 5\}, \{10, 10\}, \{4\}, \{10\}.$
- apply the  $b$ -transitions to these classes:  
 $\{7\}, \{9, 9\}, \{3, 3, 7\}, \{7, 3\}, \{10\}, \{10\}.$
- no further split
  - it produces no new classes.
  - it means  $\pi_3 = \pi_2$ . stop.



# DFA MINIMIZATION: EXAMPLE-1

$$\pi_3 = \{\{1\}, \{2, 4, 8\}, \{3, 7\}, \{5, 6\}, \{9\}, \{10\}\}.$$

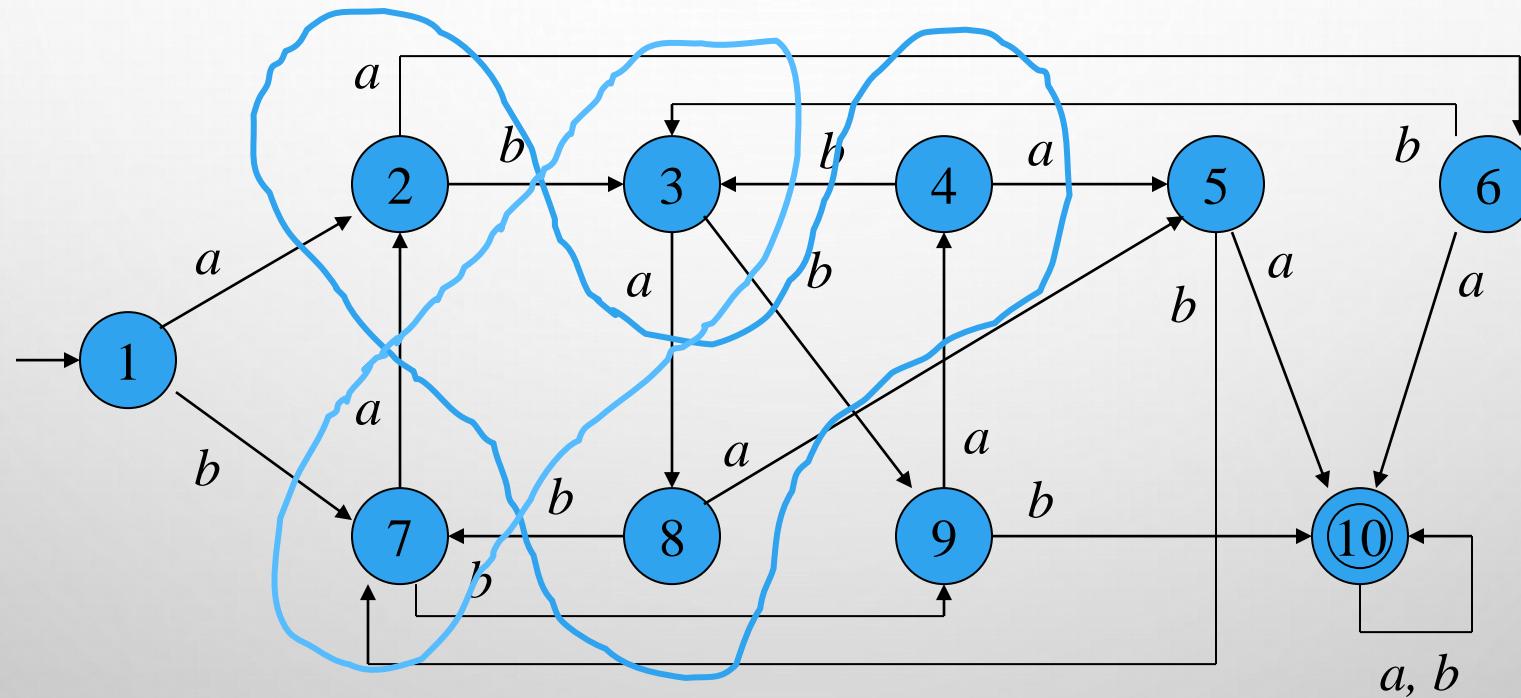
- Replace each equivalence class of states with a single state.
- Merge states 2, 4, and 8 into one state.



# DFA MINIMIZATION: EXAMPLE-1

$$\pi_3 = \{ \{1\}, \{2, 4, 8\}, \{3, 7\}, \{5, 6\}, \{9\}, \{10\} \}.$$

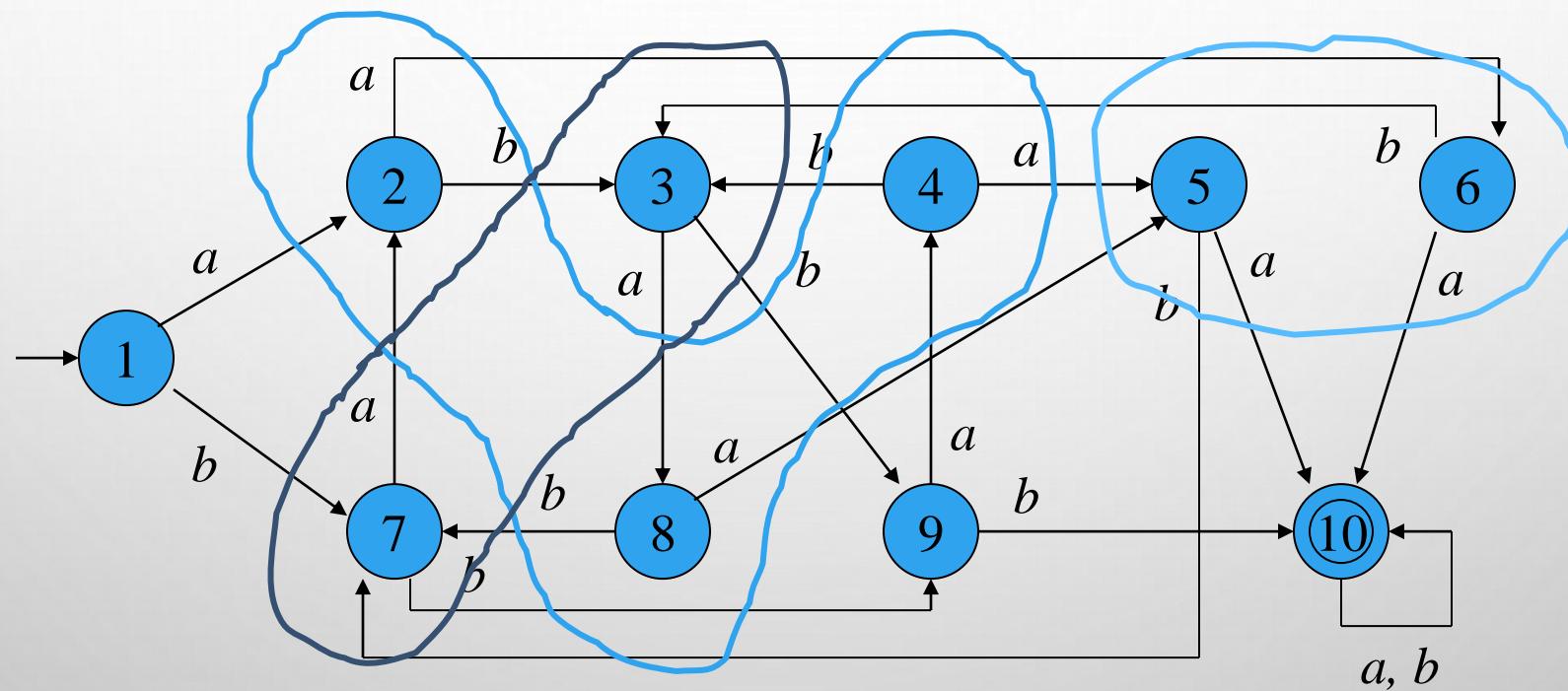
- Merge states 3 and 7 into one state.



# DFA MINIMIZATION: EXAMPLE-1

$$\pi_3 = \{ \{1\}, \{2, 4, 8\}, \{3, 7\}, \{5, 6\}, \{9\}, \{10\} \}.$$

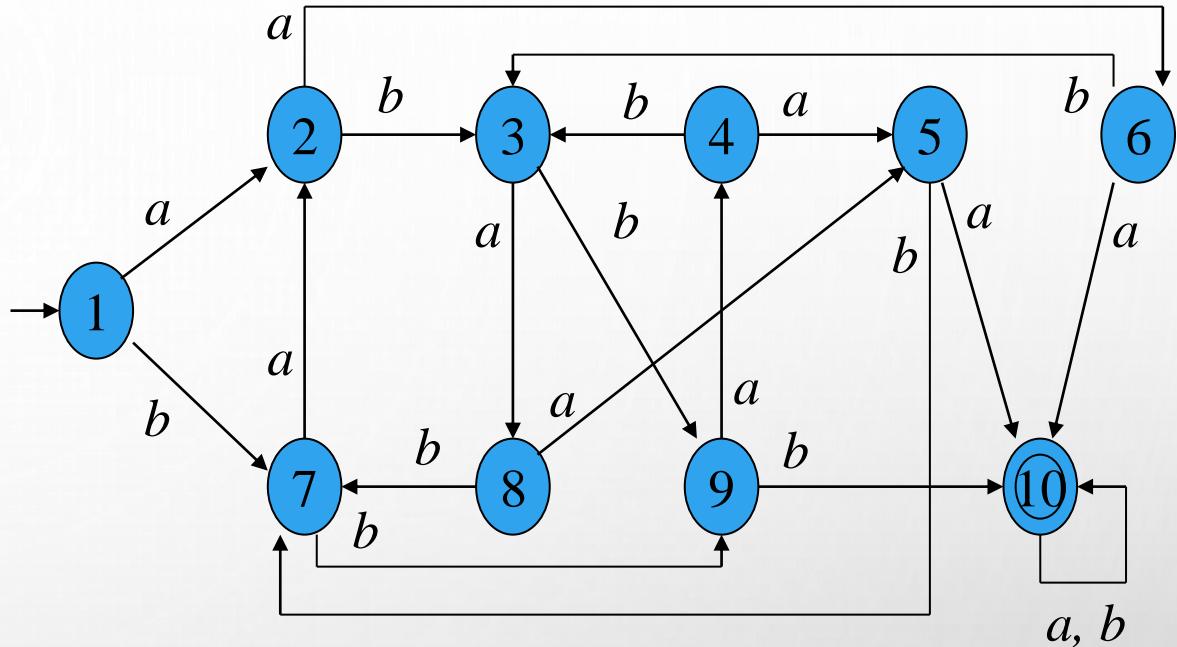
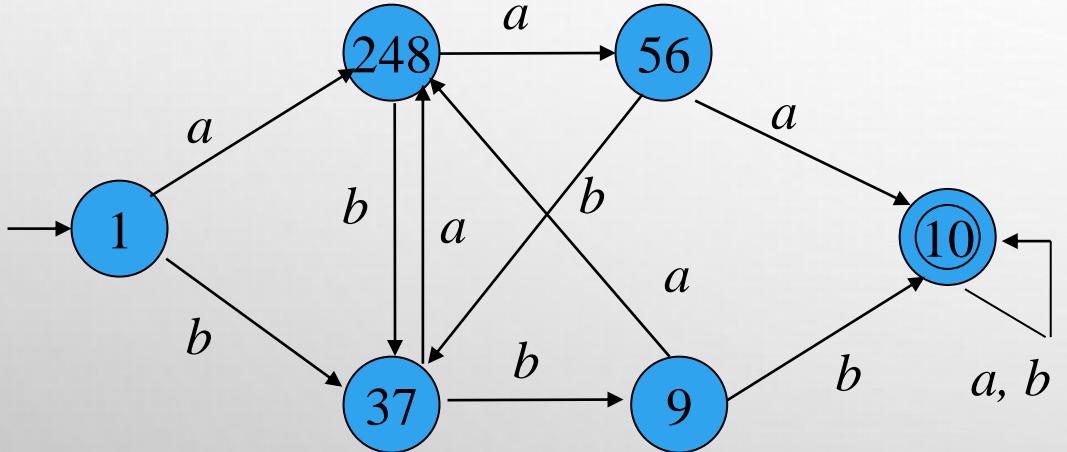
- Merge states 5 and 6 into one state.



# DFA MINIMIZATION: EXAMPLE-1

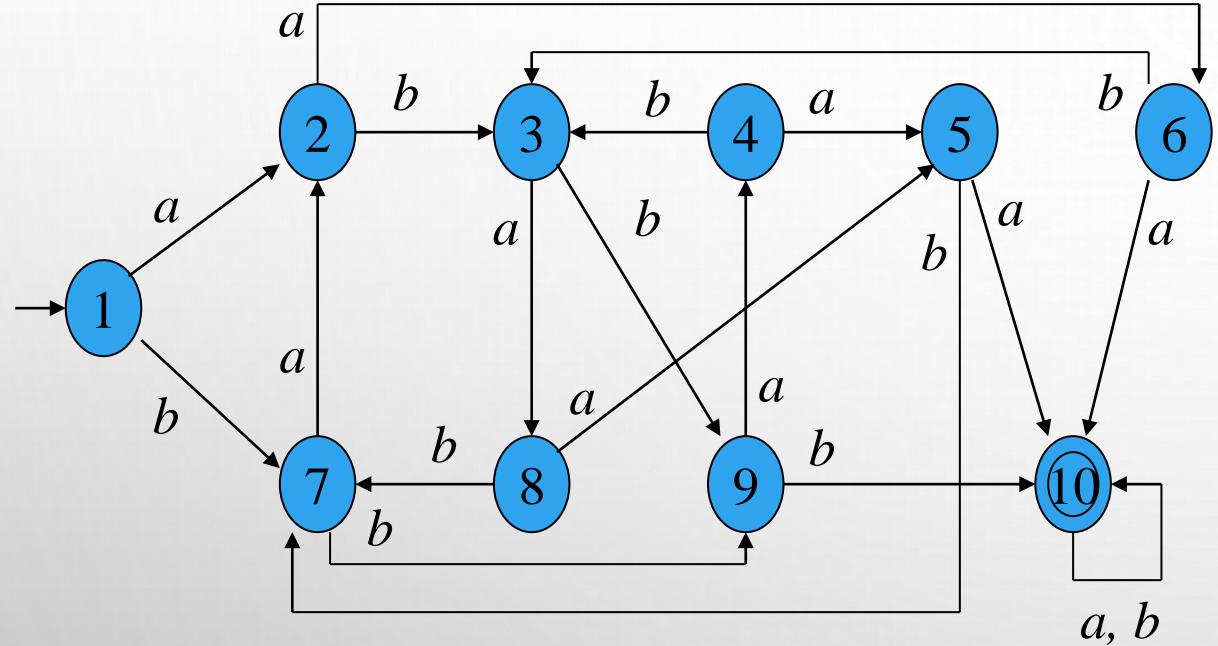
$$\pi_3 = \{\{1\}, \{2, 4, 8\}, \{3, 7\}, \{5, 6\}, \{9\}, \{10\}\}.$$

- initial state = {1}
- final state = { {10} }
- replace each of these classes with a single state.



# DFA Minimization: What about transitions?

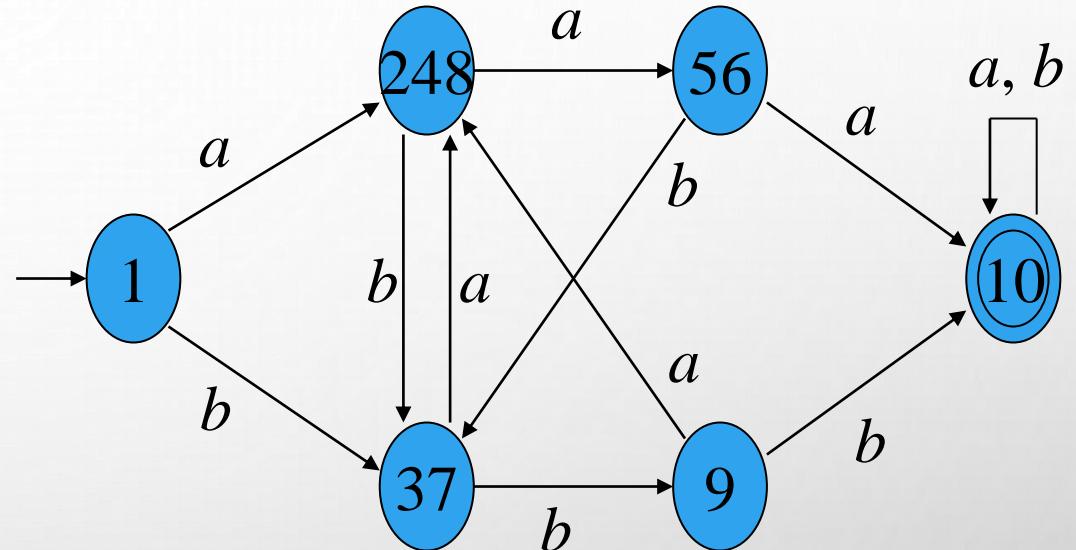
**Are they lost? No**



Given DFA  $M$

$$M = (Q, \Sigma, \delta, q_0, F)$$

$$M = ( \{1,2,3,4,5,6,7,8,9, 10\}, \{a, b\}, \delta, 1, \{10\} )$$



Minimum DFA  $M'$

$$M' = (Q', \Sigma, \delta, q'_0, F')$$

$$Q' = \{ \{1\}, \{2,4,8\}, \{3,7\}, \{5,6\}, \{9\}, \{10\} \}$$

# DFA MINIMIZATION: CORRECTNESS

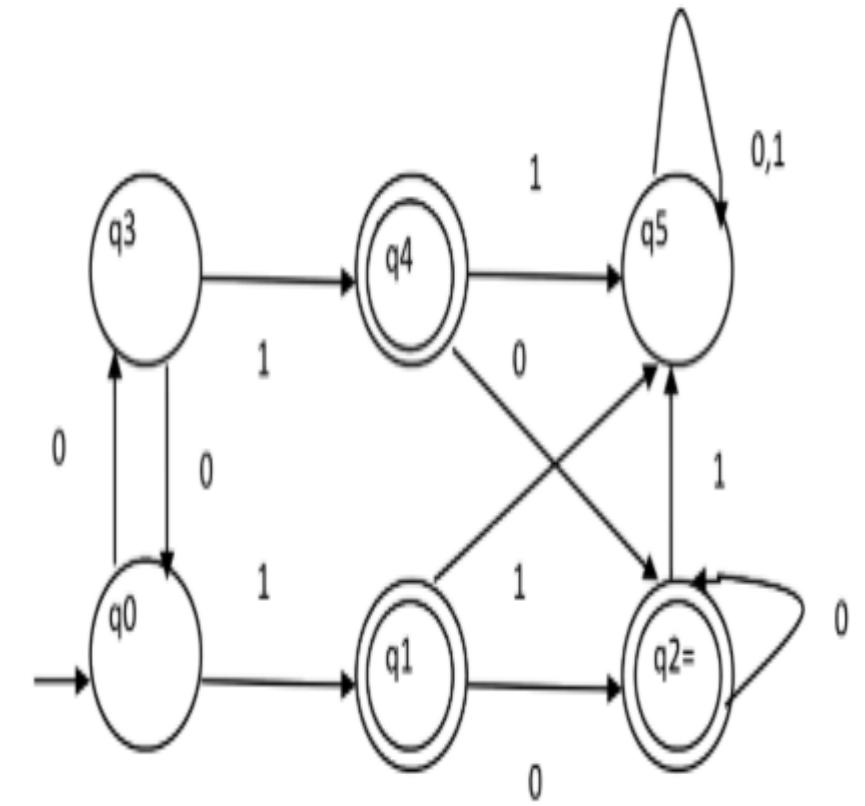
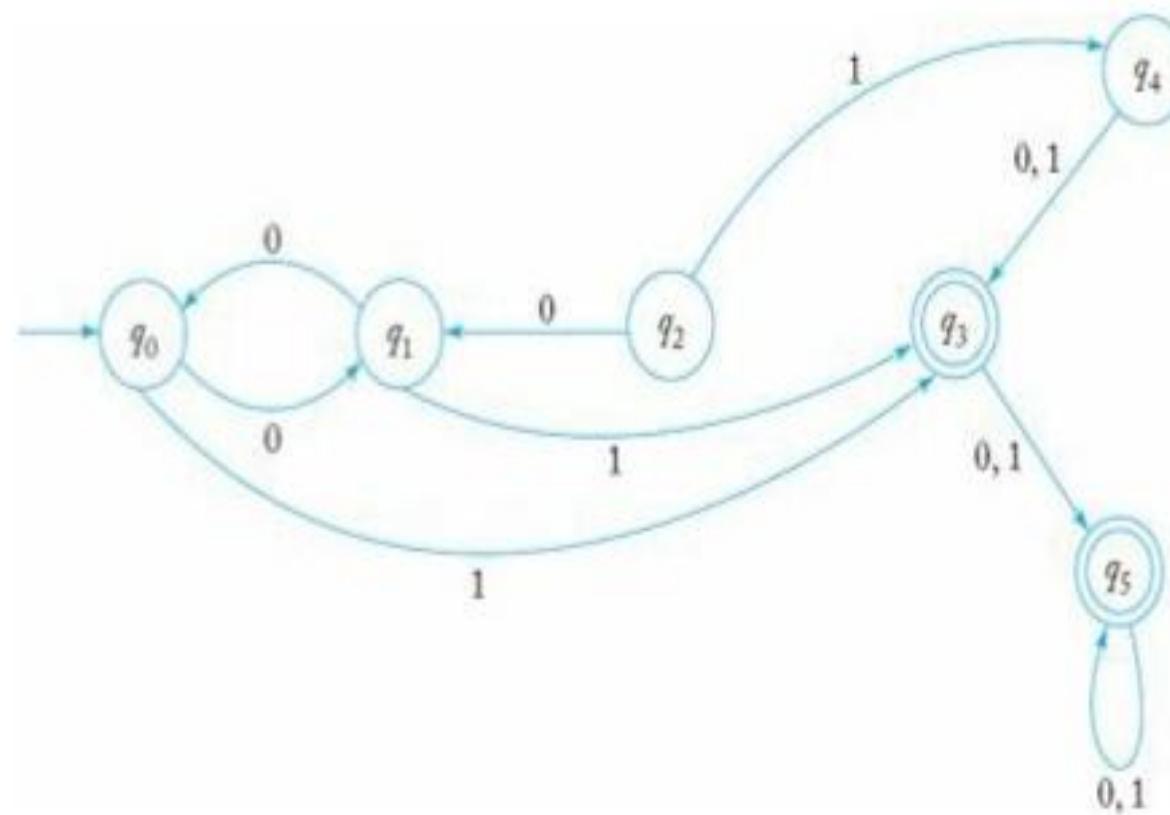
Why is the new DFA no larger than the old DFA?

- Only removing OR merges states
- Never introduces new states.

Why is the new DFA equivalent to the old DFA?

- Only identify states that provably have same behavior.
- Could prove for string  $x \in L(M) \leftrightarrow x \in L(M')$  using indistinguishability or distinguishability.

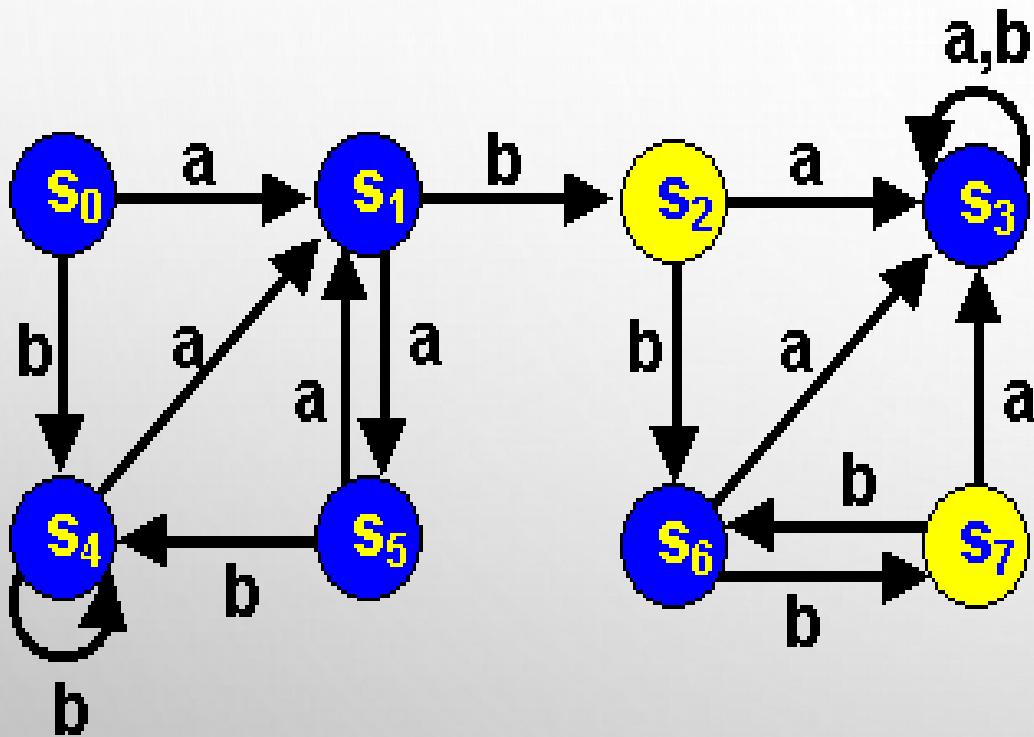
# FIND THE MINIMUM DFA



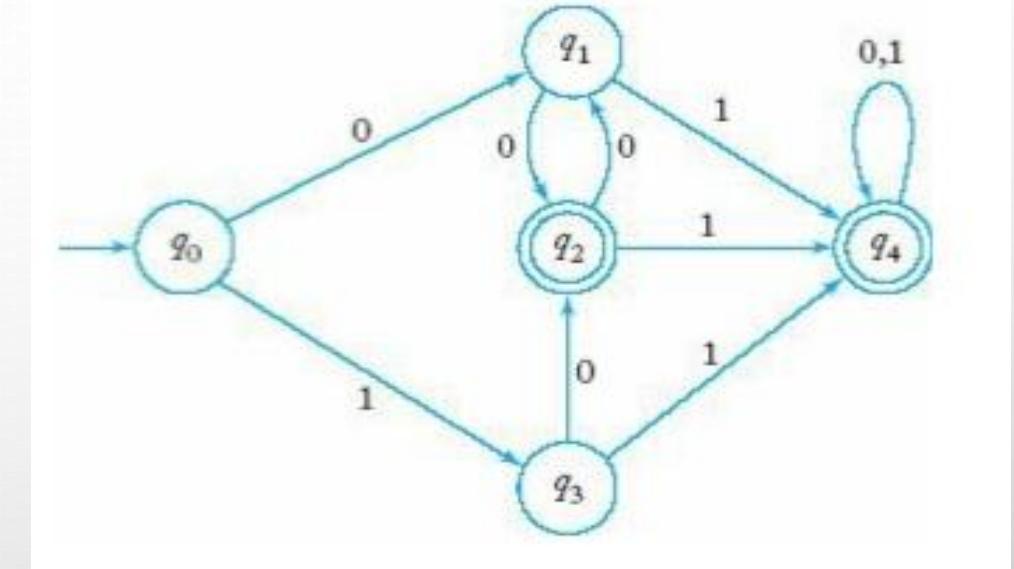
- $q_0$  is initial state.
- $q_3$  and  $q_5$  are final states.

- $q_0$  is initial state.
- $q_1$ ,  $q_2$  and  $q_4$  are final states.

# FIND THE MINIMUM DFA



- $S_0$  is initial state.
- $S_2$  and  $S_7$  are final states.



- $q_0$  is initial state.
- $q_2$  and  $q_4$  are final states.

# Regular Expression & Regular Languages

# Regular Language

- A language L is known as **regular if and only if** it is recognized by a finite accepter (FA).
  - **Language L is regular if and only if** it is recognized by a **DFA**. (??)
  - **Language L is regular if and only if** it is recognized by an **NFA**. (??)
- A language L is known as **regular if and only if** it is described by a **regular expression (RE)**.
- A language L is recognized by a FA **if and only if** L is described by a **regular expression**.
- FA recognize exactly the **regular languages**.
- **Regular expressions** describe exactly the **regular languages**.

**How to show that a given language is regular ?**

# Regular Expression

- A regular expression consists of strings of symbols from some alphabet  $\Sigma$ , parentheses  $()$ , and three operators  $+$ ,  $.$  and  $*$ .
- Let  $\Sigma$  be a given alphabet. Then,
  - $\phi$ ,  $\lambda$  and  $a \in \Sigma$  are all regular expressions. These are known as primitive regular expressions.
  - **Recursive Definition:**
    - If  $r_1$  and  $r_2$  are regular expressions (REs), then the following expressions are also regular
      - $r_1 + r_2$  OR  $r_1 | r_2 \rightarrow (r_1 \text{ or } r_2)$
      - $r_1.r_2$  OR  $r_1r_2 \rightarrow (r_1 \text{ followed by } r_2)$
      - $r_1^* \rightarrow (r_1 \text{ repeated zero or more times})$
      - $(r_1)$
- A string of symbols is a regular expression if and only if it can be derived from primitive regular expressions by finite applications of recursive definition.

# Valid Regular Expressions: Example

- Let  $\Sigma = \{a, b, c\}$  be a given alphabet.

1.  $\phi, \lambda, a, b, c$
2.  $a^*, b^*, c^*$
3.  $a.b, b.a, a+b, (a.b)^*, (b.a)^*, (a+b)^*$ 
  1.  $a + b$  is equivalent to  $b+a$
  2.  $a.b$  is not equivalent to  $b.a$
4.  $(a + b.c)^*$
5.  $(c+\phi)$

$r1 = a, r2 = b \rightarrow r3 = a.b \rightarrow r4 = (a.b)^*$

- Let  $\Sigma$  be a given alphabet. Then,
  - $\phi, \lambda$  and  $a \in \Sigma$  are all regular expressions. These are known as primitive regular expressions.
  - Recursive Definition:**
    - If  $r_1$  and  $r_2$  are regular expressions (REs), then the following expressions are also regular:
      1.  $r_1 + r_2$  OR  $r_1 | r_2 \rightarrow (r_1 \text{ or } r_2)$
      2.  $r_1.r_2$  OR  $r_1 r_2 \rightarrow (r_1 \text{ followed by } r_2)$
      3.  $r_1^* \rightarrow (r_1 \text{ repeated zero or more times})$
      4.  $(r_1)$

Why ?

# Invalid Regular Expressions: Example

- Let  $\Sigma = \{a, b, c\}$

1. d, e
2. \*a, \*b\*, +a\*, .b\*
3. +a.b, \*b.a, .\*a+b, (++a.b)\*
4. (..\*b.a\*\*\*)\*, (++a++b\*\*)\*
5. (+a + b.c)\*
6. (c+ϕ\*+\*)

- Let  $\Sigma$  be a given alphabet. Then,
  - $\emptyset, \lambda$  and  $a \in \Sigma$  are all regular expressions. These are known as primitive regular expressions.
  - Recursive Definition:**
    - If  $r_1$  and  $r_2$  are regular expressions (REs), then the following expressions are also regular:
      1.  $r_1 + r_2$  OR  $r_1 | r_2 \rightarrow (r_1 \text{ or } r_2)$
      2.  $r_1.r_2$  OR  $r_1r_2 \rightarrow (r_1 \text{ followed by } r_2)$
      3.  $r_1^* \rightarrow (r_1 \text{ repeated zero or more times})$
      4.  $(r_1)$

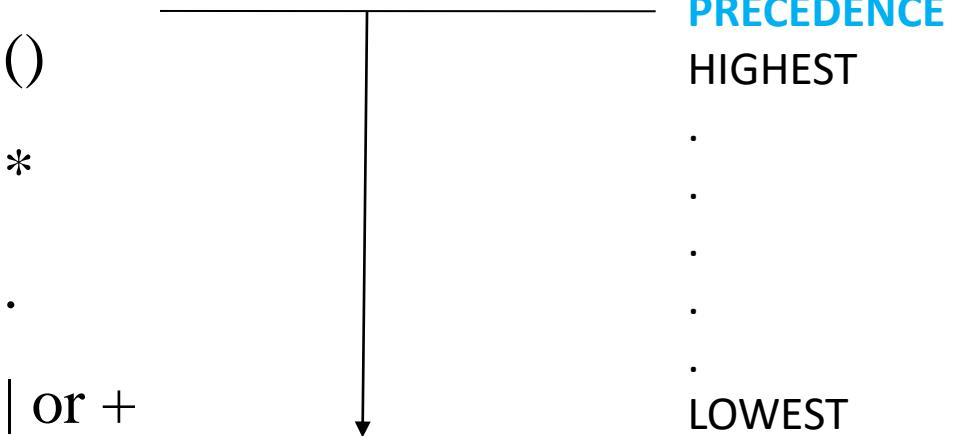
Why ?

# Rules for REs

If  $r$ ,  $s$  and  $t$  are RE over  $\Sigma$ , then

1.  $r + \emptyset = \emptyset + r = r$
2.  $r \cdot \emptyset = \emptyset \cdot r = \emptyset$
3.  $\emptyset^* = \lambda$
4.  $r + \lambda = \lambda + r = r$
5.  $r \cdot \lambda = \lambda \cdot r = r$
6.  $\lambda^* = \lambda$
7.  $(r + \lambda)^+ = r^*$
8.  $r + s = s + r$
9.  $r \cdot (s + t) = r \cdot s + r \cdot t$
10.  $r \cdot (s \cdot t) = (r \cdot s) \cdot t$
11.  $r^+ = r \cdot r^*$
12.  $r^* = r^*(r + \lambda) = r^* \cdot r^* = (r^*)^*$
13.  $(r^*s^*)^* = (r + s)^*$

## Rules for Specifying Regular Expressions:



$\lambda$  and  $\emptyset$  represent empty string.

- Parentheses in regular expressions can be omitted when the order of evaluation is clear.

$$\begin{aligned}((0+1)^*) &= (0+1)^* \neq 0+1^* \\ ((0^*)+(1^*)) &= 0^* + 1^*\end{aligned}$$

- For concatenation,  $\cdot$  can be omitted.

$r \cdot r \cdot r \dots r$  is denoted by  $r^n$ .

**n times**

## Simple Examples over $\Sigma = \{0,1\}$

1.  $\{\alpha \in \Sigma^* \mid \alpha \text{ does not contain 1's}\} = \{e, 0, 00, 000, 0000, \dots\}$ 
  - $0^*$
2.  $\{\alpha \in \Sigma^* \mid \alpha \text{ contains 1's only}\} = \{1, 11, 111, 1111, \dots\}$ 
  - $1 \cdot (1^*)$  (which can be denoted by  $(1^+)$ )
3.  $\{\alpha \in \Sigma^* \mid \alpha \text{ contains only 0's or only 1's}\} = \{0, 00, 000, 0000, \dots, 1, 11, 111, 1111, \dots\}$ 
  - $(00^*) + (11^*) \rightarrow 0^+ + 1^+$
4.  $\Sigma^* = \{0, 1\}^* = \{e, 0, 1, 01, 10, 00, \dots\}$ 
  - $(0+1)^*$       **Note:  $0^* + 1^* \neq (0+1)^*$**
5. **Strings of even length**,  $L = \{00, 01, 10, 11\}^* = \{e, 00, 11, 01, 10, 0000, 0010, 1111, \dots\}$ 
  - $(00+01+10+11)^*$  or  $((0+1)(0+1))^*$
6. **Strings of length 6**,  $L = \{\alpha \in \Sigma^* \mid \text{the length of } \alpha \text{ is 6}\} = \{000000, 000001, 000010, 000011, 000100, \dots\}$ 
  - $000000 + \dots + 111111$
  - $(0+1)(0+1)(0+1)(0+1)(0+1)(0+1) = (0+1)^6$
7. **Strings of length 6 or less**,  $L = \{\alpha \in \Sigma^* \mid \text{the length of } \alpha \text{ is less than or equal to 6}\}$ 
  - $\lambda + 0 + 1 + 00 + 01 + 10 + 11 + \dots + 111111$
  - $(0+1 + \lambda)^6$

# Examples over $\Sigma = \{0,1\}$

- All strings of 1s and 0s  
 $(0 \mid 1)^*$
- All strings of 1s and 0s beginning with a 1  
 $1(0 \mid 1)^*$
- All strings containing two or more 0s  
 $(1|0)^*0(1|0)^*0(1|0)^*$
- $\{\alpha \in \Sigma^* \mid \alpha \text{ is a binary number divisible by } 4\}$   
 $(0+1)^*00$
- All strings ending with "011"  
 $(0 \mid 1)^*011$
- All strings with at least one "0" and one "1", and no "0" after "1"  
 $1^*0(1+0)^*11^*$
- $\{\alpha \in \Sigma^* \mid \alpha \text{ does not contain } 11\}$   
 $(0+10)^* (1+ \lambda)$
- $\{\alpha \in \Sigma^* \mid \alpha \text{ contains odd number of } 1's\}$   
 $0^*(10^*10^*)^*10^*$
- $\{\alpha \in \Sigma^* \mid \text{any two } 0's \text{ in } \alpha \text{ are separated by three } 1's\}$   
 $1^*(0111)^*01^* + 1^*$
- All strings containing an even number of 0s  
 $(1^*01^*01^*)^* \mid 1^*$
- All strings starting with "1" and containing no "00"  
 $(1 \mid 10)^*$
- All strings with no "0" after "1"  
 $0^*1^*$

# Regular Expressions: Exercise

- Construct a RE over  $\Sigma=\{0,1\}$  such that
  - The set of all strings ending with “00”
  - The set of all strings with 3 consecutive 0’s
  - **The set of all strings not containing 101 as a sub-string**
  - **It does not contain any string with two consecutive “0”’s**
  - **It has no prefix with two or more “0”’s than “1” nor two or more “1”’s than “0”**
  - **The set of all strings beginning with “1”, which when interpreted as a binary no., is divisible by 5**
  - **The set of all strings with a “1” at the 5th position from the right**
- Construct a RE for the set  $\{a^n b^m : n \geq 3, m \text{ is even}\}$ .
- Construct a RE for the set  $\{a^n b^m : n \geq 4, m \leq 3\}$ .
- Construct a RE for the set  $\{w : |w| \bmod 3 = 0\}$  over  $\{a, b\}$
- Construct a RE for the set  $\{w : |w| \bmod 3 = 1\}$  over  $\{a,b\}$

# Regular Languages

- What languages do the following RE represent?

- $((0 \mid 1)(0 \mid 1))^* \mid ((0 \mid 1)(0 \mid 1)(0 \mid 1))^*$

- Each RE has an equivalent regular language (RL).

- A language L is regular iff there is a regular expression r such that  $L = L(r)$ .

- The language  $L(r)$  denoted by any regular expression r is defined by the following rules.

- **$\Phi$  is a regular expression.  $L(\Phi) = \{\} = \Phi$**
- **$\lambda$  is a regular expression.  $L(\lambda) = \{\lambda\}$**
- **$a \in \Sigma$  are all regular expressions.  $L(a) = \{a\}$**

Let  $\Sigma$  be a given alphabet. Then,

- $\phi, \lambda$  and  $a \in \Sigma$  are all regular expressions. These are known as primitive regular expressions.
- **Recursive Definition:**
  - If  $r_1$  and  $r_2$  are regular expressions (REs), then the following expressions are also regular:
    1.  $r_1 + r_2$  OR  $r_1 \mid r_2 \rightarrow (r_1 \text{ or } r_2)$
    2.  $r_1 \cdot r_2$  OR  $r_1 r_2 \rightarrow (r_1 \text{ followed by } r_2)$
    3.  $r_1^* \rightarrow (r_1 \text{ repeated zero or more times})$
    4.  $(r_1)$

# Regular Languages: Cont..

- If  $r_1$  and  $r_2$  are regular expressions (REs).

- $r_1 + r_2$  is R.E., then

$$L(r_1 + r_2) = L(r_1) \cup L(r_2)$$

$$L(r_1) \cup L(r_2) = \{w \mid w \in L(r_1) \text{ or } w \in L(r_2)\}$$

- $r_1 \cdot r_2$  is R.E., then

$$L(r_1 \cdot r_2) = L(r_1) L(r_2)$$

$$L(r_1) \cdot L(r_2) = \{w_1 \cdot w_2 : w_1 \in L(r_1) \text{ and } w_2 \in L(r_2)\}$$

- $r_1^*$  is R.E., then

$$L(r_1^*) = (L(r_1))^*$$

$$(L(r_1))^* = L(r_1)^0 \cup L(r_1)^1 \cup L(r_1)^2 \cup L(r_1)^3 \cup \dots$$

- $(r_1)$  is R.E., then

$$L((r_1)) = L(r_1)$$

# Regular Languages

- $r_1 = 0, r_2 = 1$  over  $\{0, 1\}$ 
  - $r_1 + r_2$  is R.E., then  $L(r_1 + r_2) = L(0) \cup L(1) = \{0\} \cup \{1\} = \{0, 1\}$
  - $r_1.r_2$  is R.E., then  $L(r_1.r_2) = L(0).L(1) = \{0\}\{1\} = \{01\}$
  - $r_1^*$  is R.E., then  $L(r_1^*) = L(0^*) = (L(0))^* = (\{0\})^* = \{\epsilon, 0, 00, 000, \dots\}$
  - $(r_1)$  is R.E., then  $L((r_1)) = L((0)) = L(0) = \{0\}$

# Regular Expression to Regular Language

- $\Phi$  is a regular expression.  $L(\Phi) = \{\} = \emptyset$
- $\lambda$  is a regular expression.  $L(\lambda) = \{\lambda\}$
- $a \in \Sigma$  are all regular expressions.  $L(a) = \{a\}$

**Regular Expression:**  $(a + b) \cdot a^*$

$$L((a + b) \cdot a^*) = L((a + b)) L(a^*)$$

$$= L(a + b) L(a^*)$$

$$= (L(a) \cup L(b)) (L(a))^*$$

$$= (\{a\} \cup \{b\}) (\{a\})^*$$

$$= \{a, b\} \{\lambda, a, aa, aaa, \dots\}$$

$$= \{a, aa, aaa, \dots, b, ba, baa, \dots\}$$

$r_1 + r_2$  is R.E., then  $L(r_1 + r_2) = L(r_1) \cup L(r_2)$

$$L(r_1) \cup L(r_2) = \{w \mid w \in L(r_1) \text{ or } w \in L(r_2)\}$$

$r_1 \cdot r_2$  is R.E., then  $L(r_1 \cdot r_2) = L(r_1) L(r_2)$

$$L(r_1) \cdot L(r_2) = \{w_1 w_2 : w_1 \in L(r_1) \text{ and } w_2 \in L(r_2)\}$$

$r_1^*$  is R.E., then  $L(r_1^*) = (L(r_1))^*$

$$(L(r_1))^* = L(r_1)^0 \cup L(r_1)^1 \cup L(r_1)^2 \cup L(r_1)^3 \cup \dots$$

$(r_1)$  is R.E., then  $L((r_1)) = L(r_1)$

# RE & RL: Example

1.  $01^*$ 
  - $\{0, 01, 011, 0111, \dots\}$
2.  $(01^*)(01)$ 
  - $\{001, 0101, 01101, 011101, \dots\}$
3.  $(0 \mid 1)^*$ 
  - $\{0, 1, 00, 01, 10, 11, \dots\}$  i.e., all strings of 0 and 1
4.  $\lambda^*$  is RE, then the language
  - $L(\lambda^*) = \{\lambda\}^* = \{\lambda\}$
5.  $\phi^*$  is RE, then the language
  - $L(\phi^*) = \{\phi\}^* = \{\}$
6.  $0^*$  is RE, then the language
  - $L(0^*) = \{0\}^* = \{\lambda, 0, 00, 000, 0000, \dots\}$
7.  $(0+1).(00+11)$  is RE, then the language
  - $L((0+1).(00+11)) = \{0, 1\}\{00, 11\} = \{000, 011, 100, 111\}$
8.  $(10+01)^*$  is RE, then the language
  - $L((10+01)^*) = \{10, 01\}^* = \{\lambda, 10, 1010, 101010, \dots, 01, 0101, 010101, \dots, 1001, 100101, 10010101, \dots, 0110, 011010, 01101010, \dots\}$

## RE to RL

$$r = (a+b)^*(a+bb)$$
$$L(r) = \{a, bb, aa, abb, ba, bbb, \dots\}$$

$$r = (aa)^*(bb)^*b$$
$$L(r) = \{a^{2n}b^{2m}b : n, m \geq 0\}$$

$$r = (0+1)^* 00 (0+1)^*$$
$$L(r) = \{ \text{ all strings containing substring } 00 \}$$

# RE & RL: Example

- Let  $L$  be a language over  $\{a, b\}$ , each string in  $L$  contains the substring  $bb$ 
  - $L = \{a, b\}^* \{bb\} \{a, b\}^*$
- $L$  is regular language (RL). Why?
  - $\{a\}$  and  $\{b\}$  are RLs
  - $\{a, b\}$  is RL
  - $\{a, b\}^*$  is RL
  - $\{b\}\{b\} = \{bb\}$  is also RL
  - Then  $L = \{a, b\}^* \{bb\} \{a, b\}^*$  is RL

# EQUIVALENT REs

- Two regular expressions **r** and **s** are equivalent ( $r=s$ ), **if and only if** r and s represent/generate the same language.
- Example-:
  - $r = a|b, s = b|a \rightarrow r = s$  Why?
  - Since  $L(r) = L(s) = \{a, b\}$
- RE  $= (a)|((b)^*(c))$  is equivalent to  $a + b^*c$

# EQUIVALENT REs

- Examples,
  - $(a^*b^*)^* = (a+b)^*$
  - $(a+b)^*ab(a+b)^* + b^*a^* = (a+b)^*$
- First equality rather clear.
- For the second equality, note that  $(a+b)^*$  denotes strings over a and b, that a string either contains  $ab$  or it doesn't; the first half of the left-hand expression describes the strings that contain the substring  $ab$  and the second half describes those that don't; the  $+$  says "take the union".

# Regular Expression to NFA

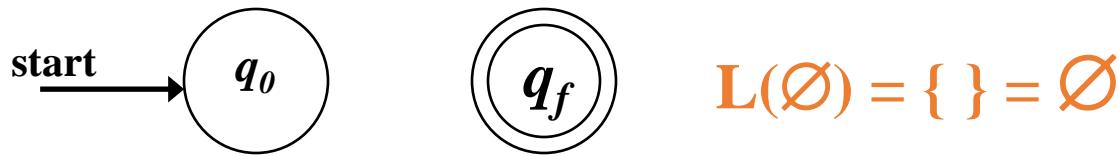
# Regular Expression to NFA

- A language  $L$  is known as **regular if and only if** it is recognized by a finite accepter (FA).
  - **Language  $L$  is regular if and only if** it is recognized by a **DFA**. (??)
  - **Language  $L$  is regular if and only if** it is recognized by an **NFA**. (??)
- A language  $L$  is called **regular** if and only if there exists some DFA  $M$  such that  $L = L(M)$ .
- Since a DFA has an equivalent NFA, then
  - A language  $L$  is called **regular** if and only if there exists some NFA  $N$  such that  $L = L(N)$ .
- If we have a RE  $r$ , we can construct an NFA that accept  $L(r)$ .

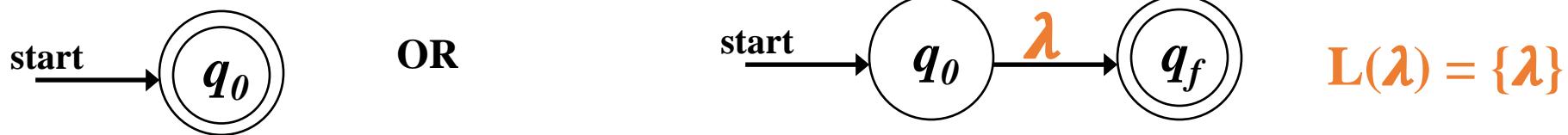
# Connection Between RE & FA

## NFAs for Primitive Regular Expression

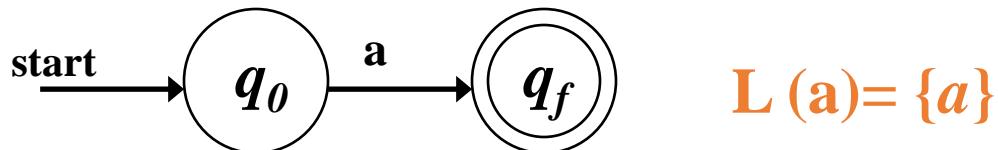
1. For regular expression  $\emptyset$ , construct NFA



2. For regular expression  $\lambda$ , construct NFA



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



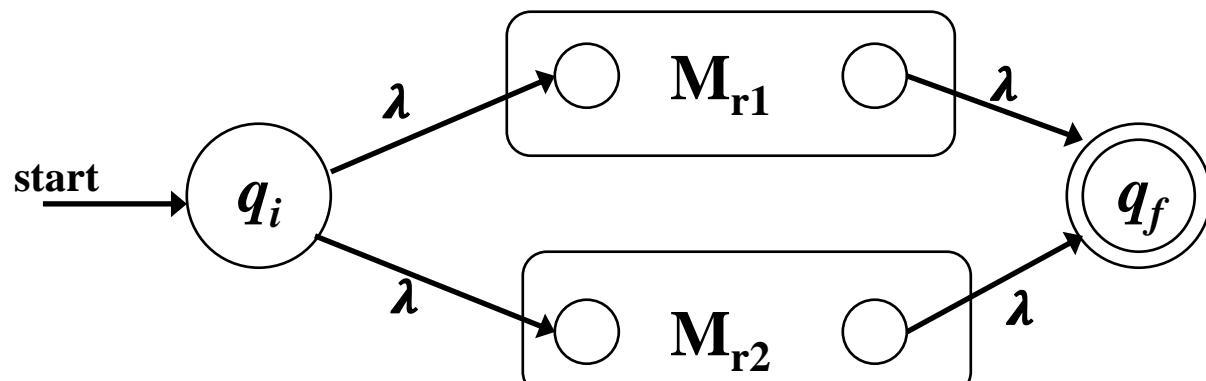
# Connection Between RE & FA

If  $r_1$  and  $r_2$  are regular expressions,  $M_{r_1}$  and  $M_{r_2}$  are their NFAs.

Then,  $r_1 + r_2$  has NFA.

Convert  $M_{r_1}$  into NFA with single final state.

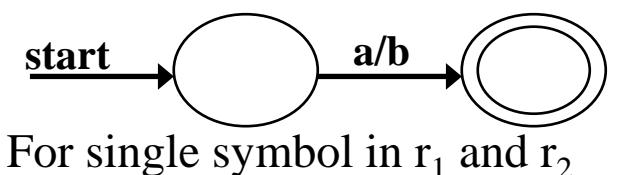
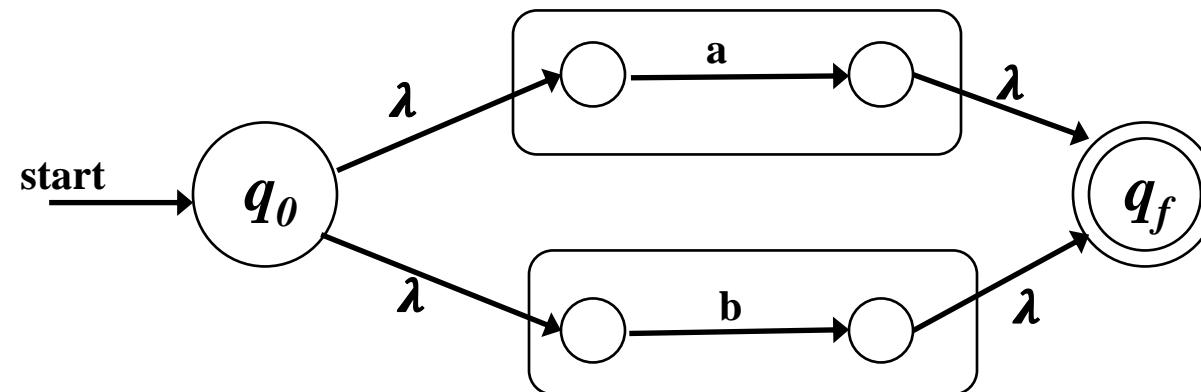
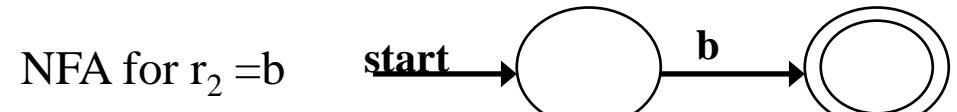
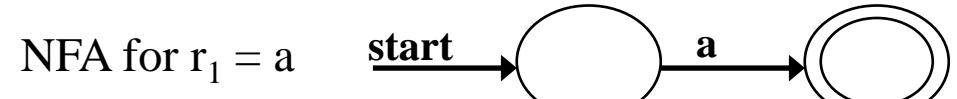
Convert  $M_{r_2}$  into NFA with single final state.



$$L(r_1 + r_2) = L(M_{r_1}) \cup L(M_{r_2})$$

where  $q_i$  and  $q_f$  are new initial/final states, and  $\lambda$ -moves are introduced from  $q_i$  to the old start states of  $M_{r_1}$  and  $M_{r_2}$  as well as from all of their final states to  $q_f$ .

$r_1 = a$  and  $r_2 = b$  are regular expressions,  
Construct NFA for  $r_1 + r_2$ .



# Connection Between RE & FA

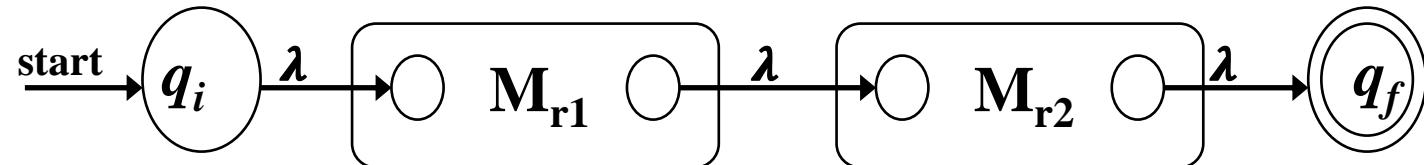
If  $r_1$  and  $r_2$  are regular expressions,  $M_{r_1}$  and  $M_{r_2}$  are their NFAs.

Then,  $r_1.r_2$  has NFA.

Convert  $M_{r_1}$  into NFA with single final state.

Convert  $M_{r_2}$  into NFA with single final state.

$$L(r_1.r_2) = L(M_{r_1}).L(M_{r_2})$$



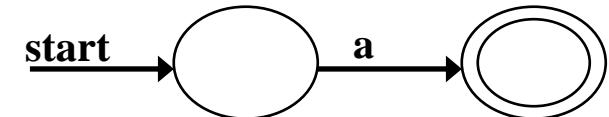
where  $q_i$  is the new initial state of  $M_{r_1}$  and  $q_f$  is the new final state of  $M_{r_2}$ .

$\lambda$ -move is introduced from final state of  $M_{r_1}$  to initial state of  $M_{r_2}$ .

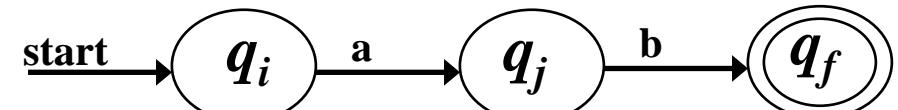
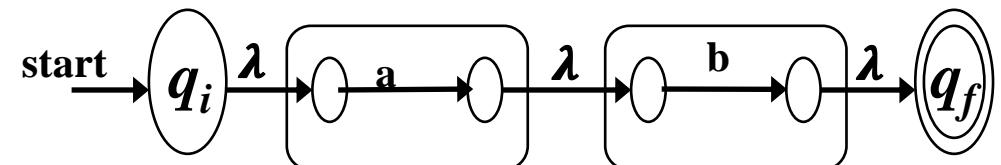
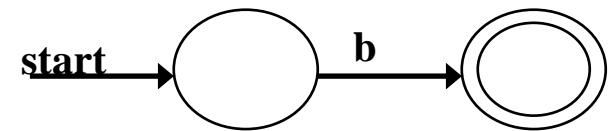
$r_1 = a$  and  $r_2 = b$  are regular expressions

Construct NFA for  $r_1.r_2$ .

NFA for  $r_1 = a$



NFA for  $r_2 = b$



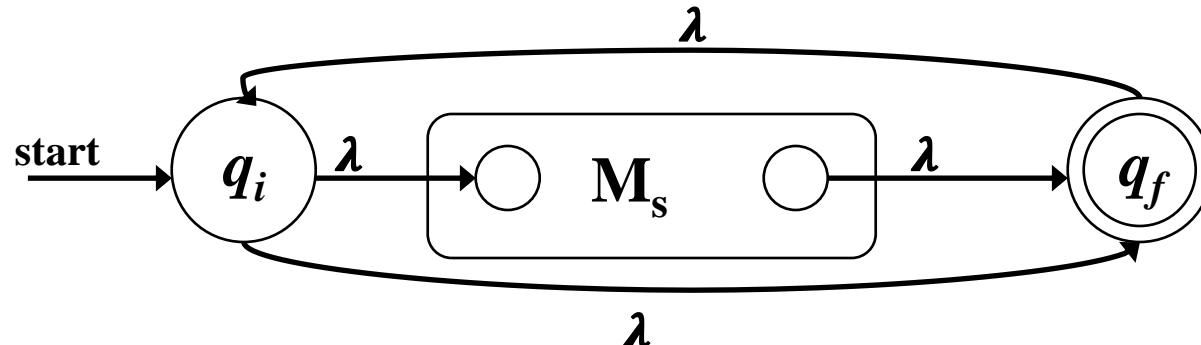
For single symbol in  $r_1$  and  $r_2$

# Connection Between RE & FA

If  $r_1$  is a regular expression and  $M_{r_1}$  its NFA,

$(r_1)^*$  (Kleene star) has NFA:

$$L((r_1)^*) = (L(r_1))^*$$



where :  $q_i$  is new start state and  $q_f$  is new final state

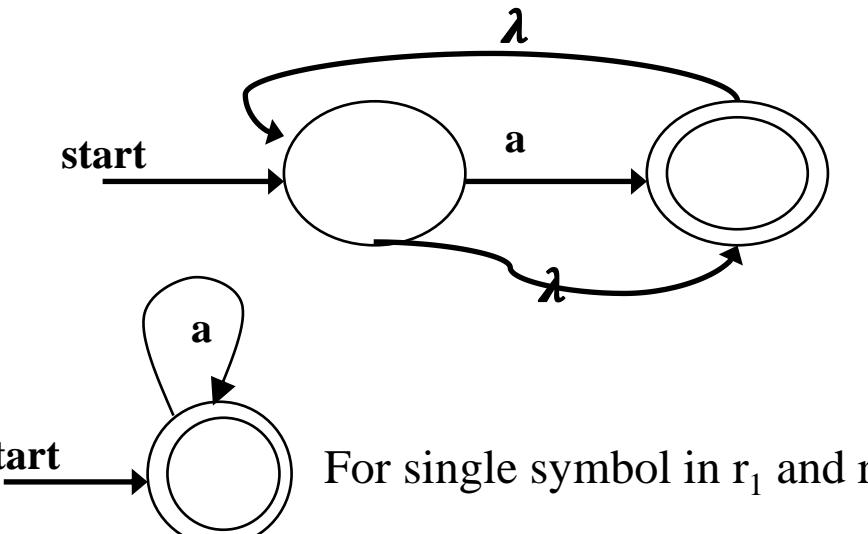
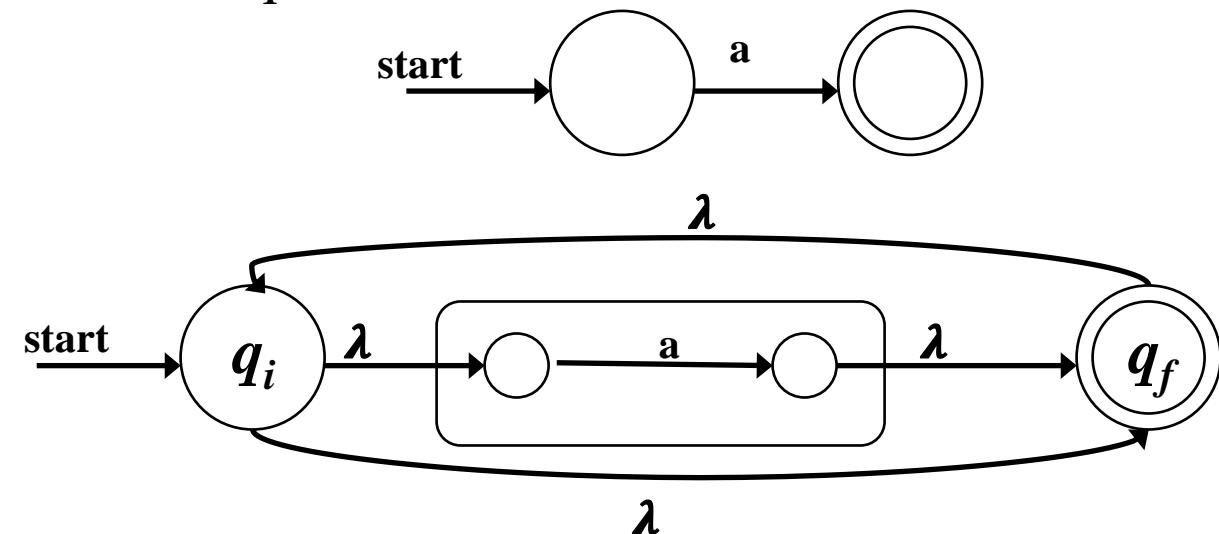
$\lambda$ -move  $q_i$  to  $q_f$  (to accept null string)

$\lambda$ -moves  $q_i$  to old start, old final(s) to  $q_f$

$\lambda$ -moves old final(s) to  $q_f$

$\lambda$ -move old final to old start (WHY? Repetition)

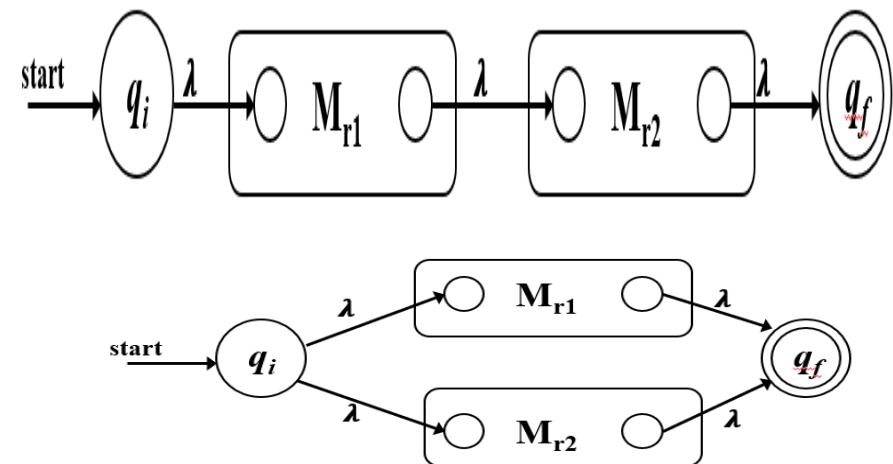
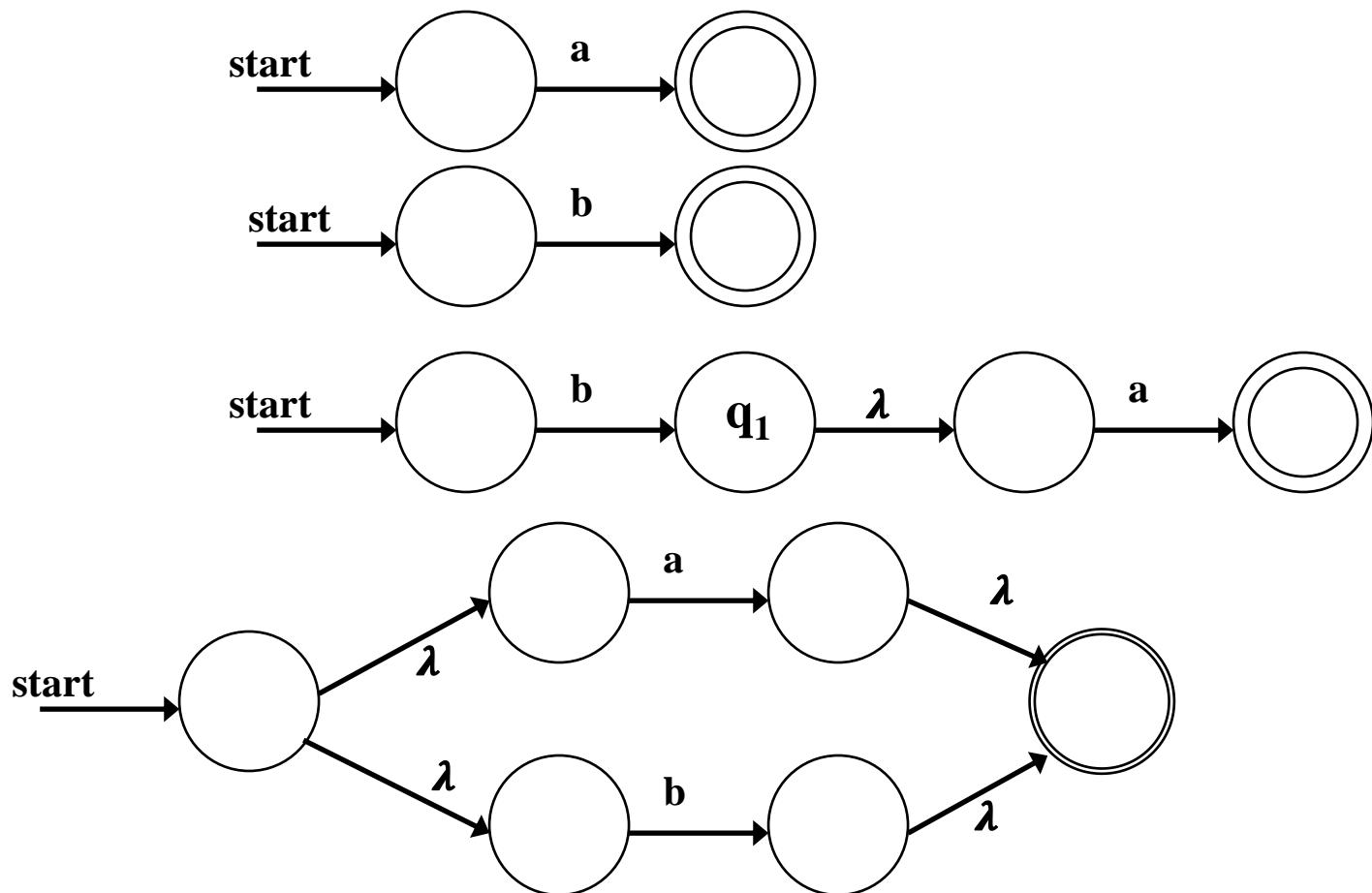
$r_1 = a$  is a regular expression. Construct NFA for  $(r_1)^*$



For single symbol in  $r_1$  and  $r_2$

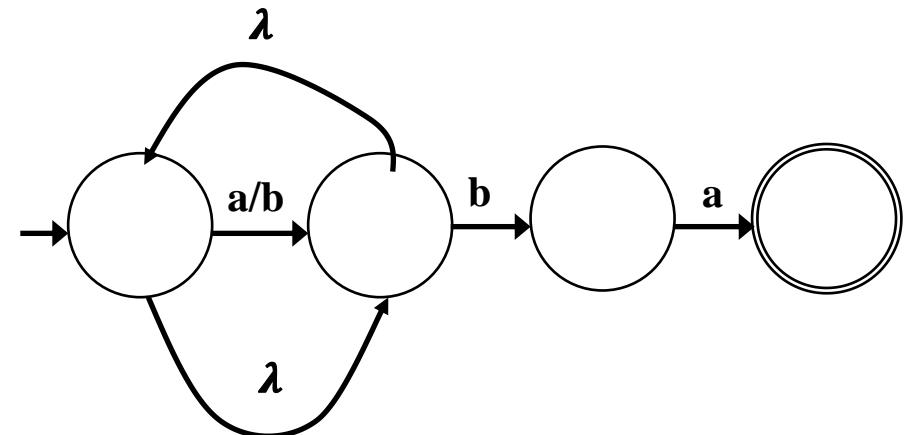
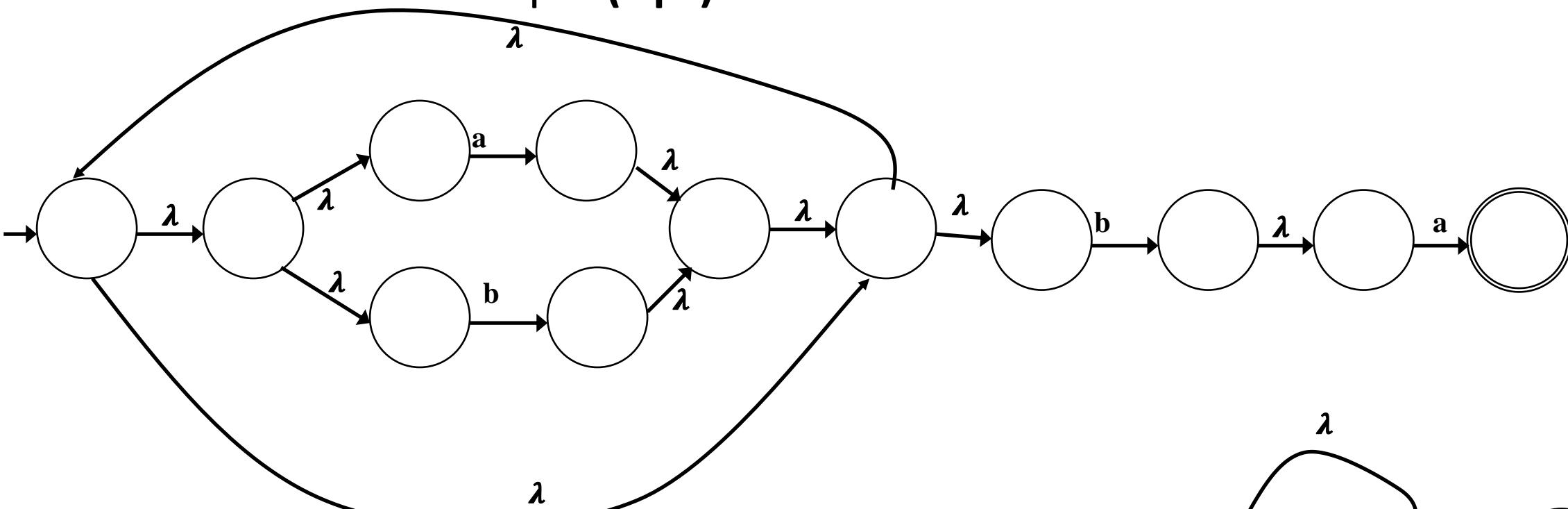
# Example-1

- Build an NFA- $\epsilon$  that accepts  $r_1 = (a|b)^*ba$
- The RE  $r_1$  consists of a, b, ba and  $a|b$



# Example-1

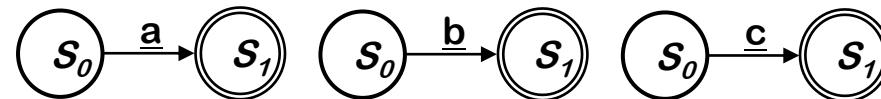
- Build an NFA that accepts  $(a|b)^*ba$



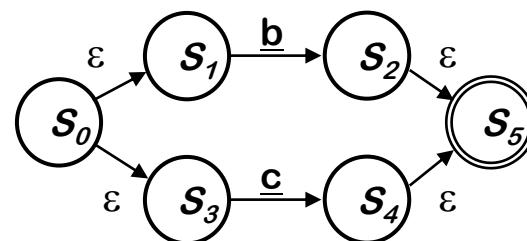
# Example-2

R.E.  $a(b|c)^*$

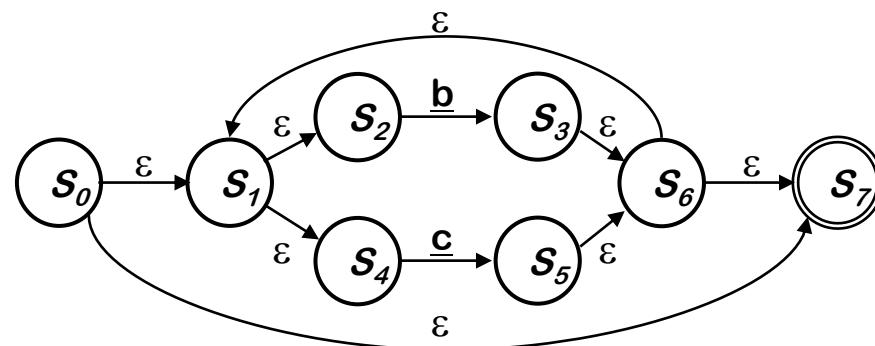
1.  $\underline{a}$ ,  $\underline{b}$ , &  $\underline{c}$



2.  $\underline{b} \mid \underline{c}$

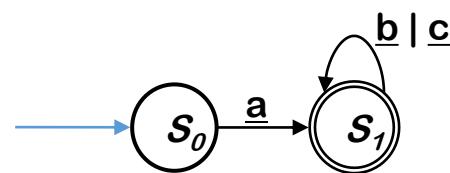
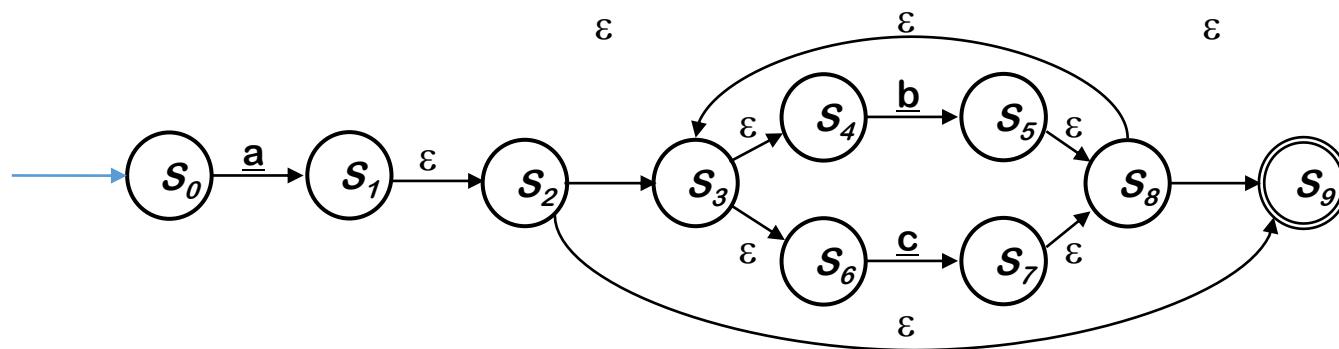


3.  $(\underline{b} \mid \underline{c})^*$



# Example-2

4.  $\underline{a} (\underline{b} \mid \underline{c})^*$

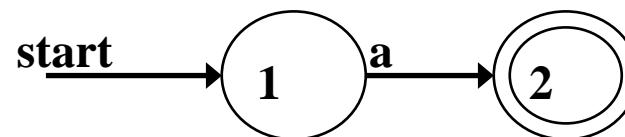


# Example-3

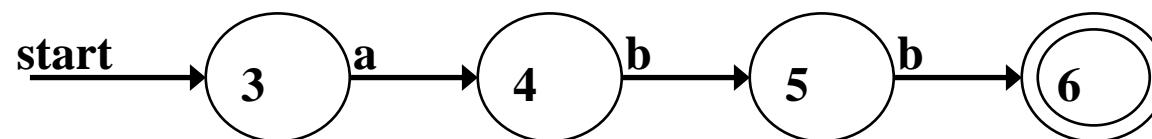
NFA for :  $a \mid abb \mid a^*b^+$

NFA's :

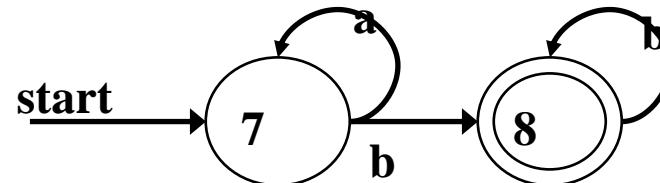
a



abb

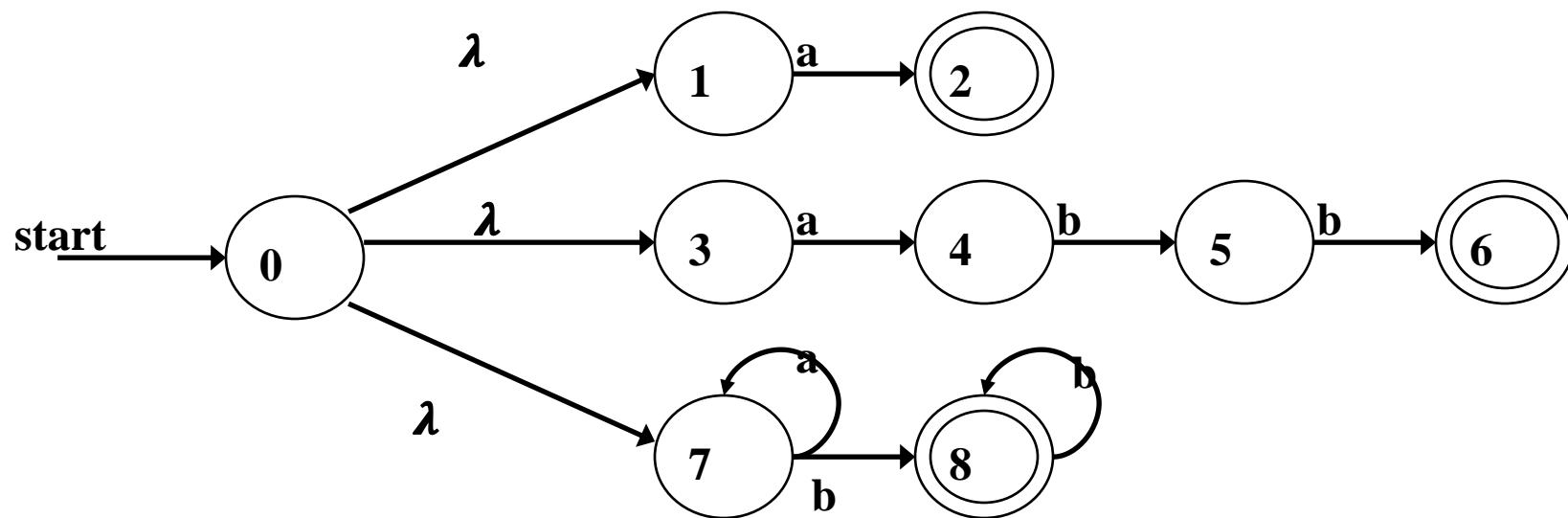


$a^*b^+$



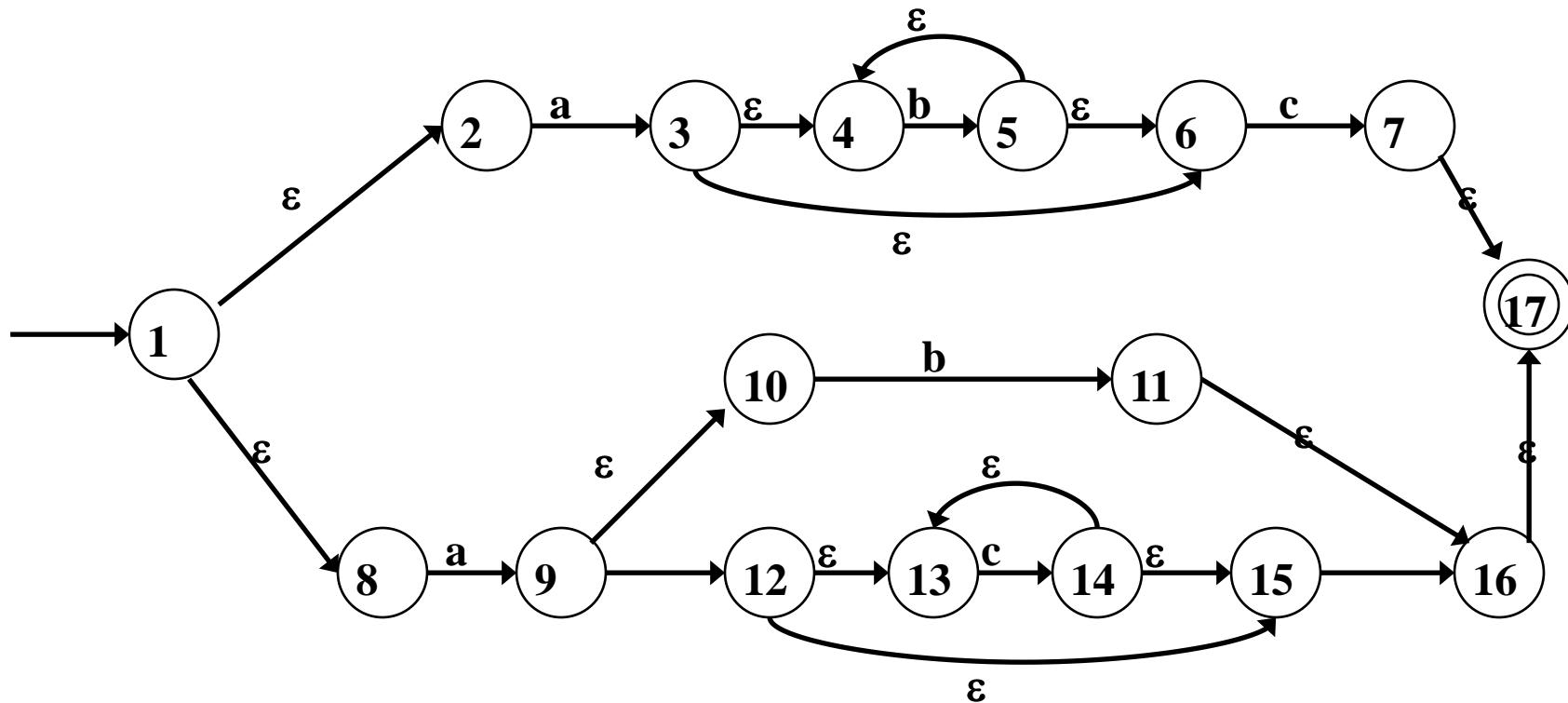
# Example-3

NFA for :  $a \mid abb \mid a^*b^+$



# Example-4

Regular Expression:  $(ab^*c) \mid (a(b|c^*))$



- $\{\alpha \in \Sigma^* \mid \alpha \text{ is a binary number divisible by 4}\}$ 
  - $(0+1)^*00$
  
- $\{\alpha \in \Sigma^* \mid \alpha \text{ does not contain } 11\}$ 
  - $(0+10)^* (1+\lambda)$
  
- $\{\alpha \in \Sigma^* \mid \alpha \text{ contains an odd number of 1's}\}$ 
  - $0^*(10^*10^*)^*10^*$
  
- $\{\alpha \in \Sigma^* \mid \text{any two 0's in } \alpha \text{ are separated by three 1's}\}$ 
  - $1^*(0111)^*01^* + 1^*$
  
- Strings over the alphabet  $\{a, b\}$  with exactly three b's
  - $a^*ba^*ba^*ba^*$
  
- Strings over the alphabet  $\{a, b, c\}$  containing (at least once) bc
  - $(a|b|c)^*bc(a|b|c)^*$
  
- Strings of even length,  $L=\{00,01,10,11\}^*$ 
  - $(00+01+10+11)^*$
  
- Strings of length 6,  $L=\{\alpha \in \Sigma^* \mid \text{the length of } \alpha \text{ is 6}\}$ 
  - $000000 + \dots + 111111 = (0+1)(0+1)(0+1)(0+1)(0+1)(0+1) = (0+1)^6$
  
- Strings of length 6 or less,  $L=\{\alpha \in \Sigma^* \mid \text{the length of } \alpha \text{ is less than or equal to 6}\}$ 
  - $\lambda + 0+1+00+01+10+11\dots+111111 = (0+1+\lambda)^6$

# Construct FAs for RE over $\Sigma = \{a, b\}$

- Construct NFA for the language  $L(ab^*aa + bba^*ab)$
- Construct NFA for the language  $L( (a + b)^*b(a + bb)^* )$
- Construct NFA for the set  $\{a^n b^m : n \geq 3, m \text{ is even}\}$ .
- Construct NFA for the set  $\{a^n b^m : n \geq 4, m \leq 3\}$ .
- Construct NFA for the set  $\{w : |w| \bmod 3 = 0\}$ .
- Construct NFA for the set  $\{w : |w| \bmod 3 = 1\}$
- Construct DFA for the language  $L( ab^*a^* ) \cup L( (ab)^*ba )$
- **Construct DFA for the language  $L( ab^*a^* ) \cap L( (ab)^*ba )$**
- **Find the minimal DFA for the language  $L( a^*bb ) \cap L( ab^*ba )$**

} After properties of  
Regular Languages  
Or Lecture 6

# NFA to Regular Expression

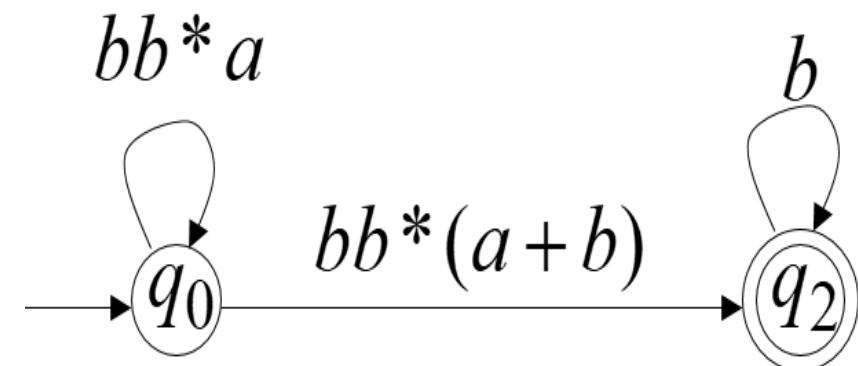
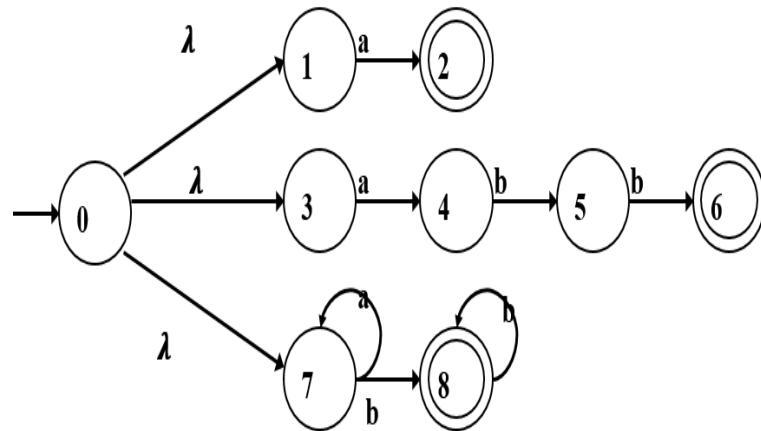
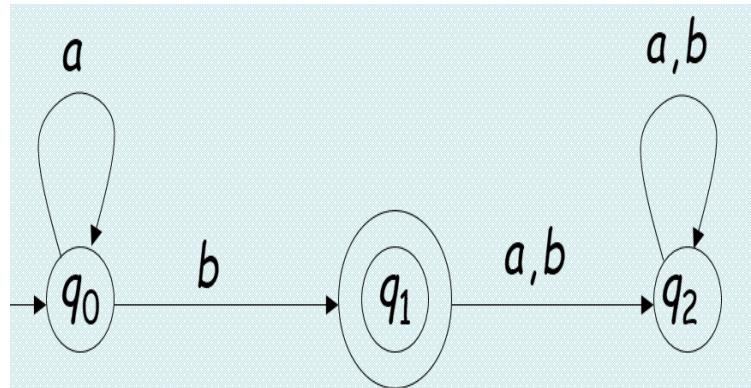
# NFA to RE

- If an NFA accepts  $L$ , then  $L$  is represented by some *regular expression*.
  - A regular expression for an NFA consists of labels of all the walks from the initial state ( $q_0$ ) to the final state (s)  $q_f$ .
  - The computation of labels of all the walks is not too difficult, but it becomes complicated due to the existence of cycles (The cycles can be traversed arbitrarily, in any order).
- Rules-
  - $r + \phi = r$
  - $r. \phi = \phi.r = \phi$
  - $\phi^* = \lambda$

# Generalized Transition Graph

- A **generalized transition graph (GTG or Expression graph)** is like a transition diagram but **it may have *regular expressions* as labels on arcs**

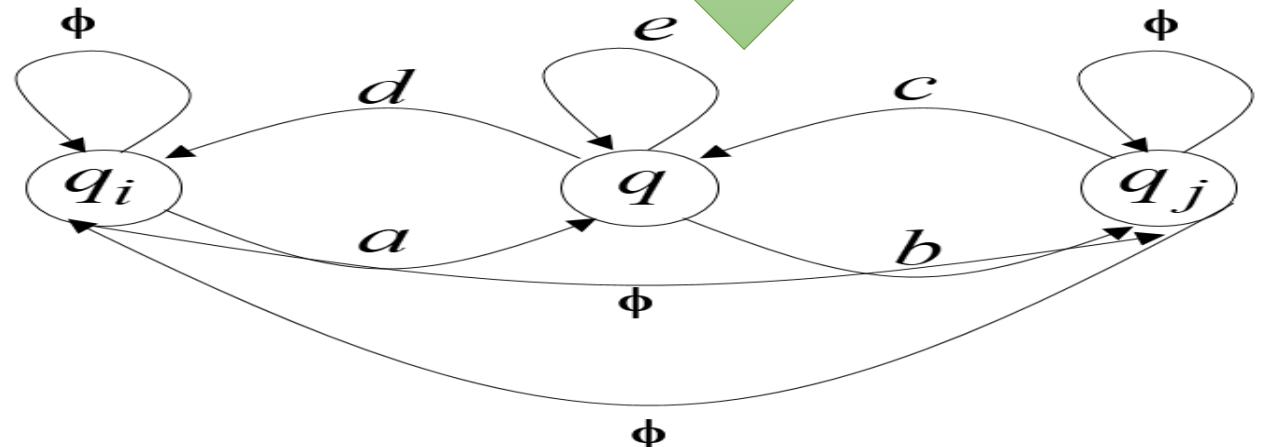
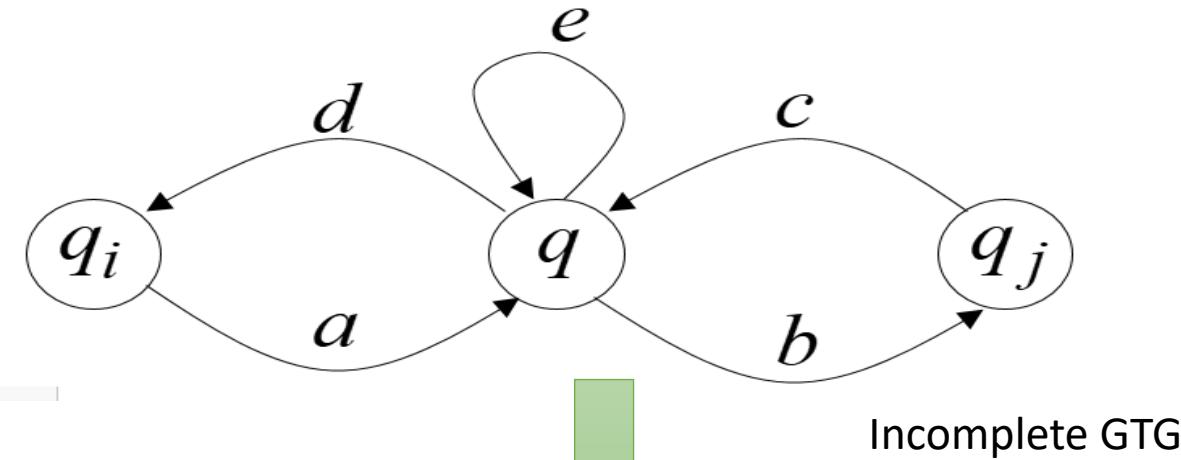
- An  $\varepsilon$ -NFA is a GTG.
- An NFA is a GTG.
- A DFA is a GTG.



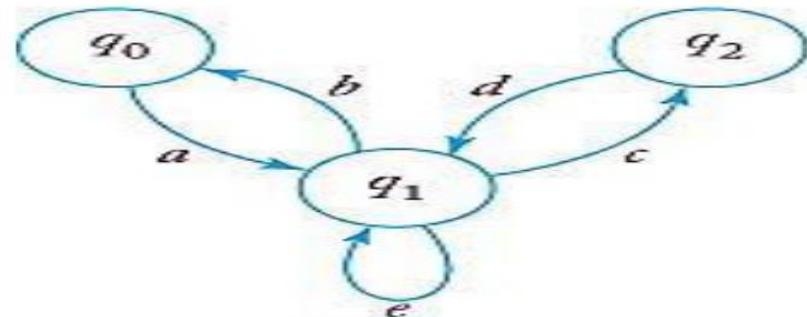
# Complete GTG

- A complete GTG is a graph in which **all edges are present**.
- A complete GTG with  $|V|$  vertices has exactly  $|V^2|$  edges.

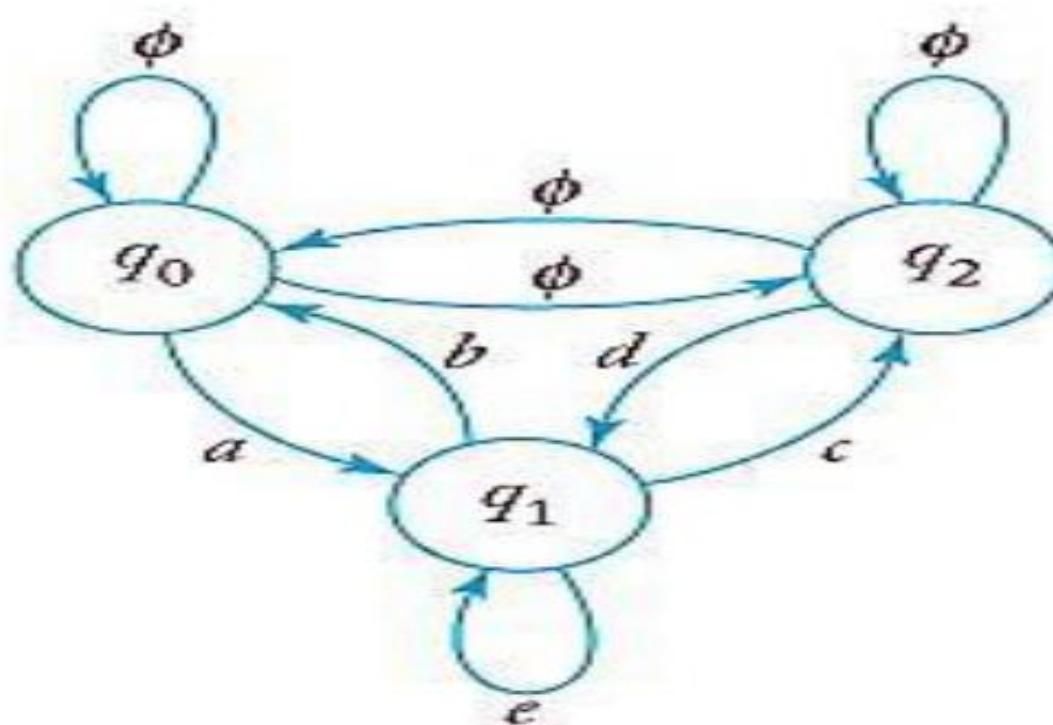
- If a GTG has some missing edges
  - Add edge with the label  $\phi$ .



# Complete GTG



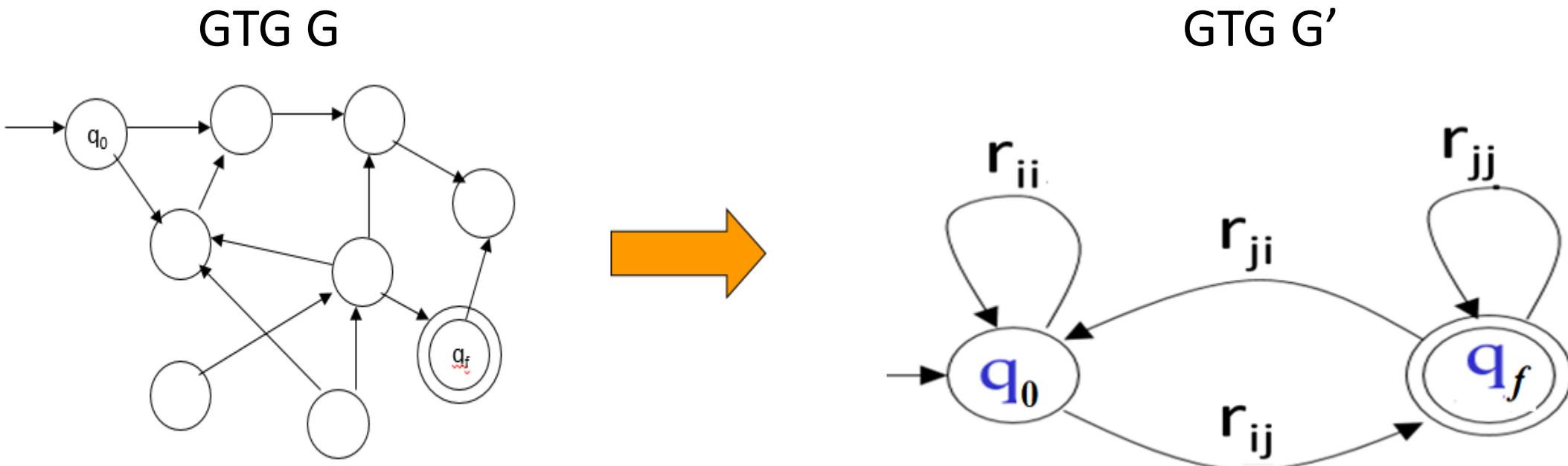
Incomplete GTG



Complete GTG

# GTG Reduction to Regular Expression

- A GTG G can be reduced to a **GTG G' with just two states (Initial and final states)**
  - The GTG G' can be reduced to an R.E.,
  - The arc label corresponds to the regular expression representing it.



# RE for GTG

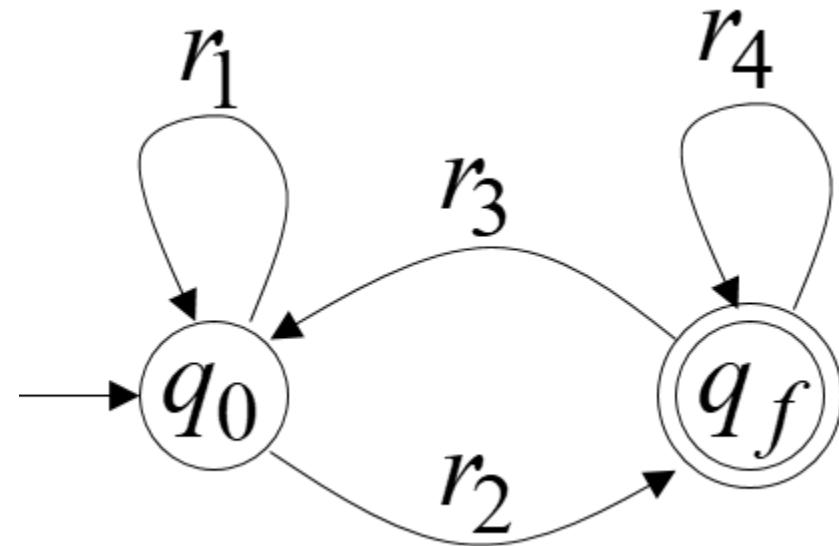
- For two-state **complete GTG**, the Regular Expression is given as.

$$r = (r_1)^* r_2 (r_4 + r_3(r_1)^* r_2)^*$$

$$A = (r_1)^*$$

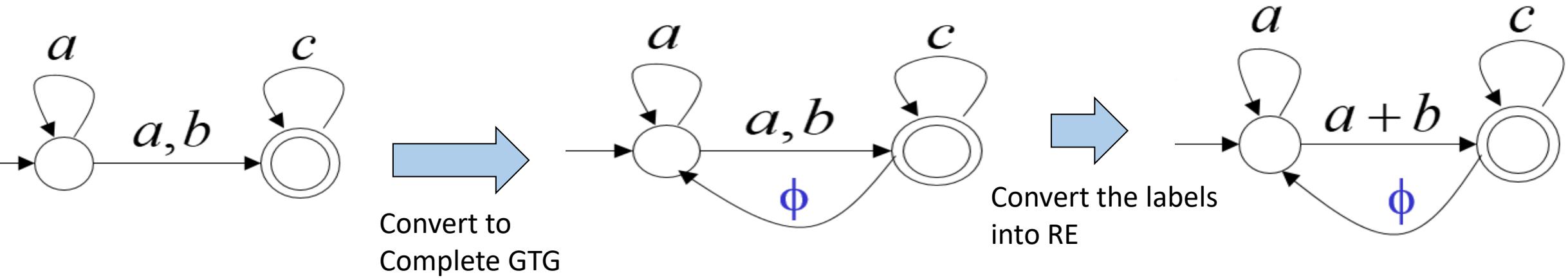
$$B = r_4 + r_3(r_1)^* r_2$$

$$r = Ar_2B^*$$

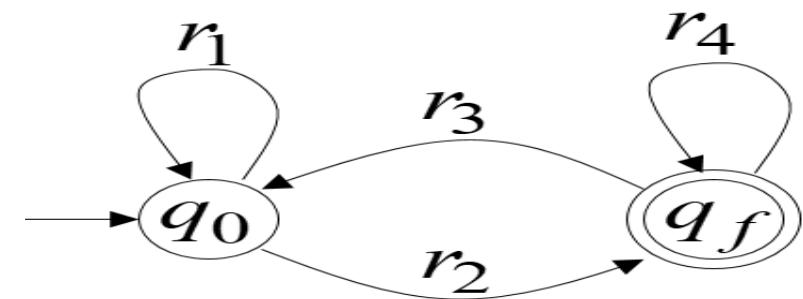


- This regular expression  $r$  covers all possible paths from the initial state to the final state.
  - First path  $q_0$  to  $q_f$  (Self Loop + Direct path from  $q_0$  to  $q_f$ )  $\rightarrow A = (r_1)^* r_2$   
**OR**
  - Second path  $q_0$  to  $q_f$  (Self Loop + Direct path from  $q_0$  to  $q_f$  + Indirect path ( $q_f$  to  $q_0$  to  $q_f$ ))

# Example



$$\begin{aligned}
 r &= (r_1)^* r_2 (r_4 + r_3(r_1)^* r_2)^* \\
 r &= (a)^*(a+b) ( c + \phi(a)^*(a+b) )^* \\
 r &= (a)^*(a+b) ( c + \phi )^* \\
 r &= a^*(a|b) c^*
 \end{aligned}$$

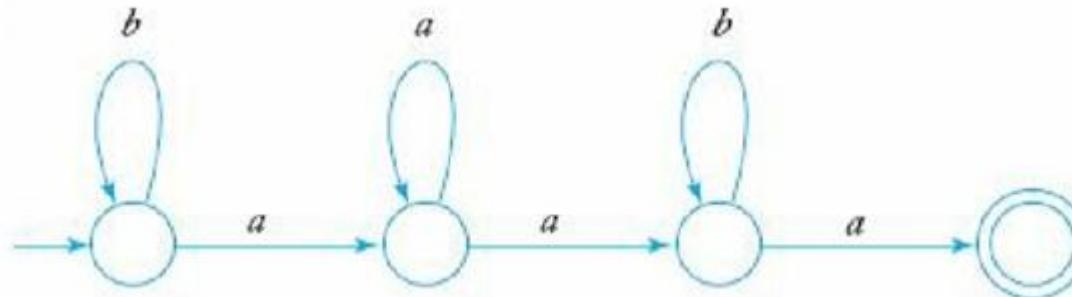


$$r = (r_1)^* r_2 (r_4 + r_3(r_1)^* r_2)^*$$

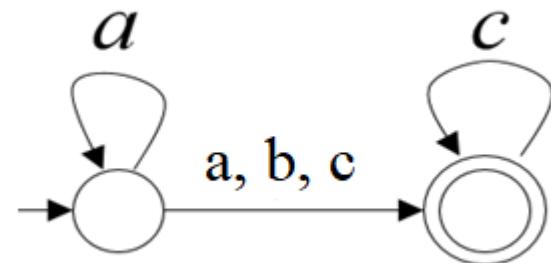
Rules-

$$\begin{aligned}
 r + \phi &= r \\
 r \cdot \phi &= \phi \cdot r = \phi \\
 \phi^* &= \lambda
 \end{aligned}$$

# Find RE for NFAs



$b^*aa^*ab^*a$



# Linear Grammars

A grammar  $G = (V, T, S, P)$  is said to be linear if all productions have **at most one variable at the right side and have exactly one variable on the left-side.**

Example:

$$S \xrightarrow{\text{G1}} aSb$$

$$S \xrightarrow{\lambda}$$

$$S \xrightarrow{\text{G2}} Ab$$

$$A \xrightarrow{\lambda}$$

$$S \xrightarrow{\text{G3}} SS$$

$$S \xrightarrow{\lambda}$$

$$S \xrightarrow{\text{G4}} A$$

$$A \xrightarrow{\lambda} aB \mid \lambda$$

$$A \xrightarrow{\lambda}$$

$$S \xrightarrow{\lambda} aSb$$

$$S \xrightarrow{\lambda} bSa$$

$$B \xrightarrow{\lambda} Ab$$

# Right- & Left-Linear Grammars

- Right-Linear Grammar

A grammar  $G = (V, T, S, P)$  is said to be right-linear if all productions are of the form

$$A \rightarrow xB$$

or

$$A \rightarrow x$$

Where

$A, B \in V$  and  $x \in T^*$

## Example 1

$$S \rightarrow abS$$

$$S \rightarrow a$$

## Left-Linear Grammar

A grammar  $G = (V, T, S, P)$  is said to be left-linear if all productions are of the form

$$A \rightarrow Bx$$

or

$$A \rightarrow x$$

Where

$A, B \in V$  and  $x \in T^*$

## Example 2

$$S \rightarrow Aab$$

$$A \rightarrow Aab \mid B$$

$$B \rightarrow a$$

# Regular Grammars

A **regular grammar** is one that is **either right-linear or left-linear grammar.**

Examples:

 $G_1$ 

$$S \rightarrow abS$$

$$S \rightarrow a$$

 $G_2$ 

$$S \rightarrow Aab$$

$$A \rightarrow Aab \mid B$$

$$B \rightarrow a$$

# Example

**G<sub>1</sub>**

$$\begin{aligned} S &\rightarrow aaA \mid Abb \\ A &\rightarrow aB \\ B &\rightarrow b \end{aligned}$$

**G<sub>2</sub>**

$$\begin{aligned} S &\rightarrow Aabc \\ A &\rightarrow abBc \\ B &\rightarrow c \end{aligned}$$

**G<sub>3</sub>**

$$\begin{aligned} S &\rightarrow S_1ab \\ S_1 &\rightarrow S_1ab \mid S_2 \\ S_2 &\rightarrow c \end{aligned}$$

**Linear, Left Linear  
Regular**

**G<sub>4</sub>**

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow aB \mid \lambda \\ B &\rightarrow Ab \end{aligned}$$

**Linear only**

**G<sub>5</sub>**

$$\begin{aligned} S &\rightarrow abS_1 \\ S_1 &\rightarrow abS_1 \mid S_2 \\ S_2 &\rightarrow c \end{aligned}$$

**Linear, Right Linear,  
Regular**

**G<sub>6</sub>**

$$\begin{aligned} S &\rightarrow S_1 \\ S_1 &\rightarrow S_2 \\ S_2 &\rightarrow a \mid b \mid c \end{aligned}$$

**Linear, Right Linear, Left Linear  
Regular**

# Right-Linear Grammar To NFA

Grammar G is right-linear.

$$G = (V, T, S, P) = (\{S, A, B\}, \{a, b\}, S, P)$$

Example:

$$S \rightarrow aA \mid B$$

$$A \rightarrow aa \mid B$$

$$B \rightarrow b \mid B \mid a$$

# Right-Linear Grammar To NFA

Construct NFA M

- Define one state for each grammar variable.
- **Define one final state (Extra).**

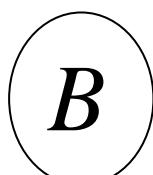
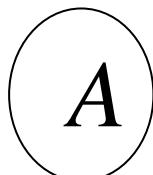
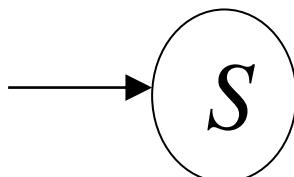
$$M = (Q, \{a, b\}, \delta, S, \{V_F\})$$

$$Q = \{S, A, B, V_F\}$$

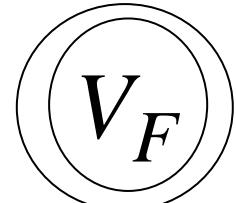
$$S \rightarrow aA \mid B$$

$$A \rightarrow aa \mid B$$

$$B \rightarrow b \mid B \mid a$$



special  
final state

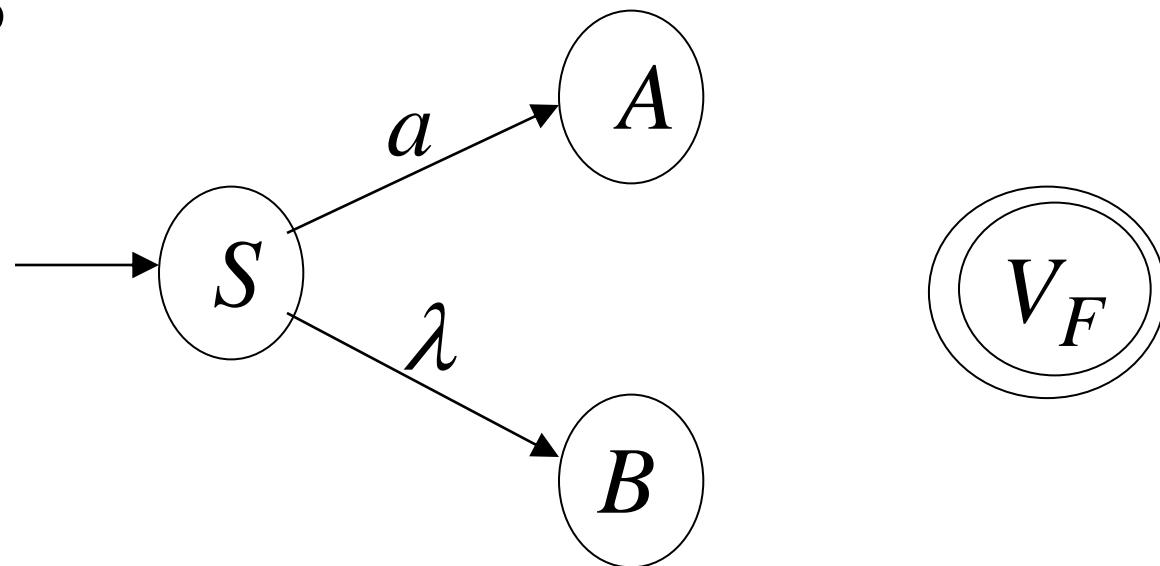


# Right-Linear Grammar To NFA

Add edges for each production:

$$Q = \{S, A, B, V_F\}$$

$$S \rightarrow aA \mid B$$

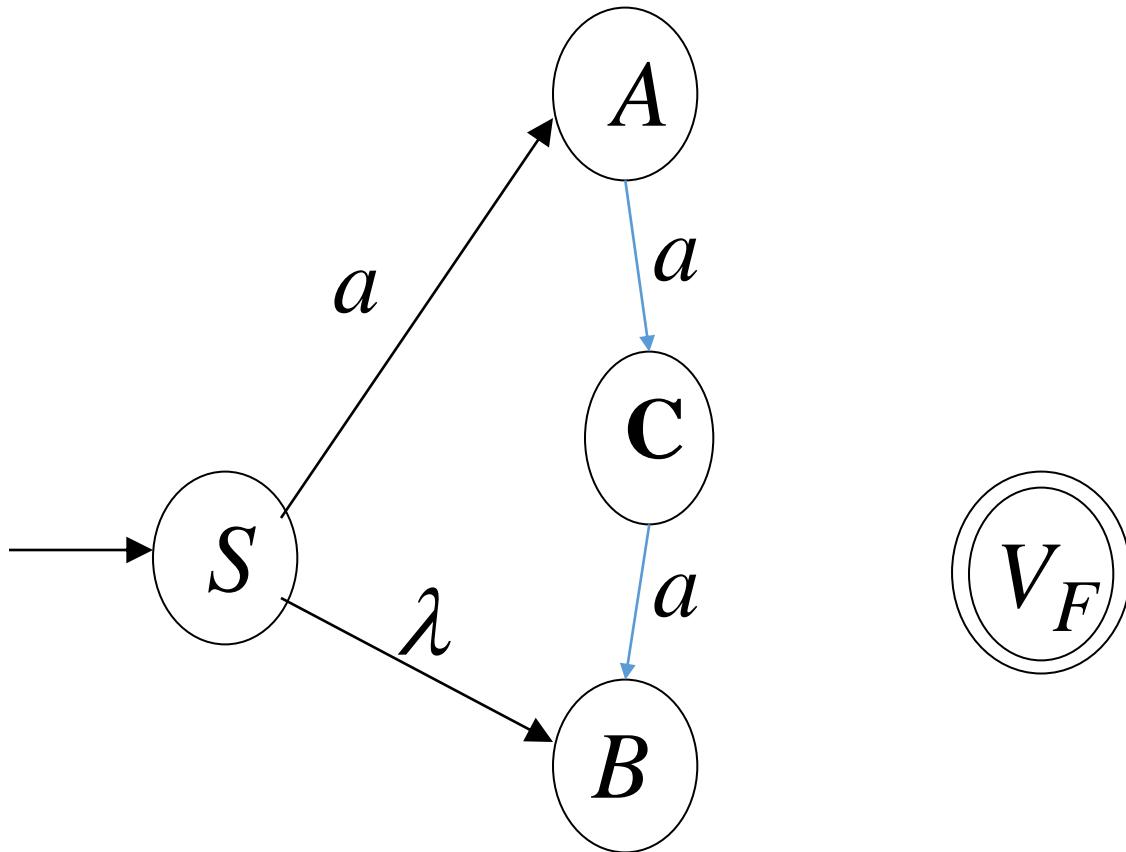


# Right-Linear Grammar To NFA

$$S \rightarrow aA \mid B$$

$$A \rightarrow aa \mid B$$

$$Q = \{S, A, B, \textcolor{blue}{C}, V_F\}$$



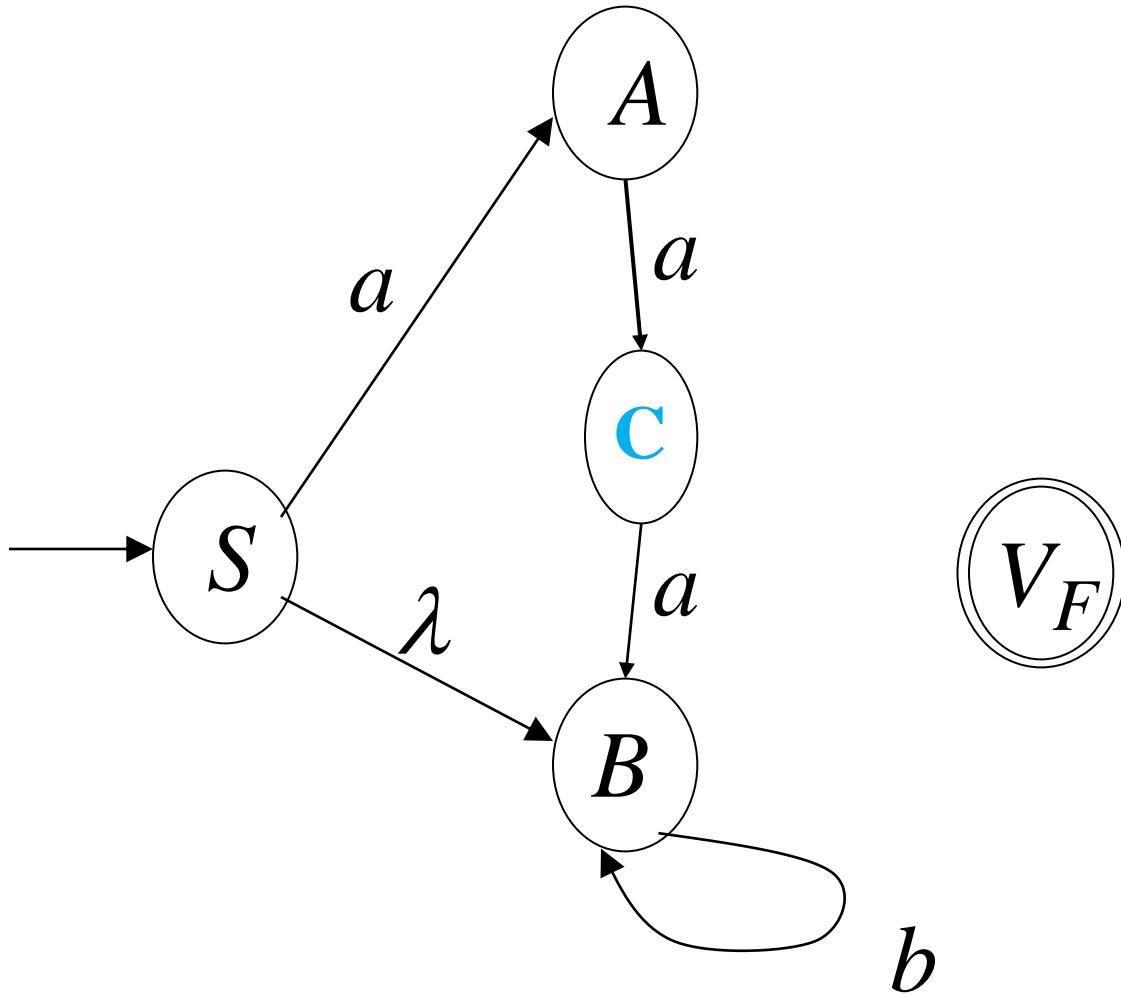
# Right-Linear Grammar To NFA

$$S \rightarrow aA \mid B$$

$$A \rightarrow aa \mid B$$

$$B \rightarrow bB$$

$$Q = \{S, A, B, C, V_F\}$$

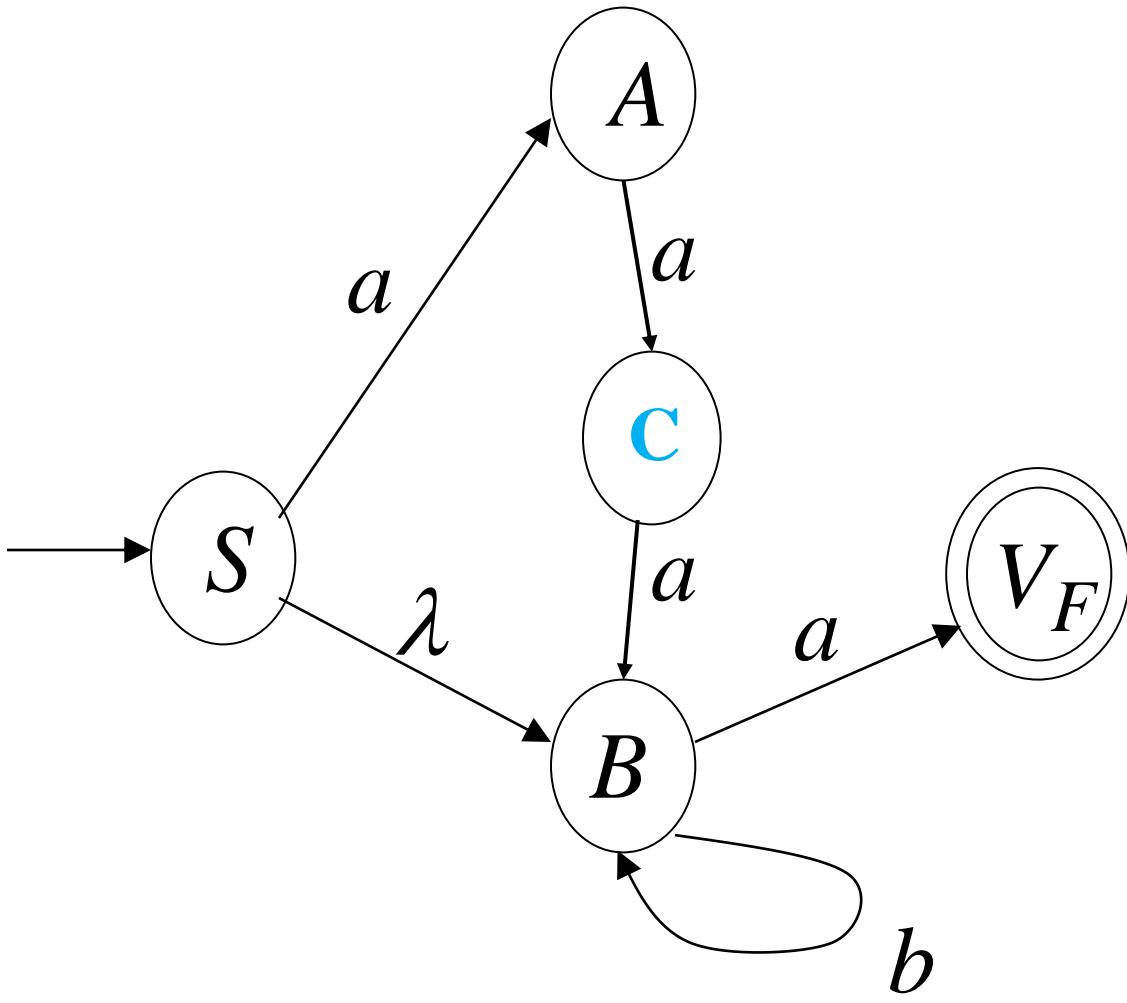


# Right-Linear Grammar To NFA

$$S \rightarrow aA \mid B$$

$$A \rightarrow aa \mid B$$

$$B \rightarrow bB \mid a$$



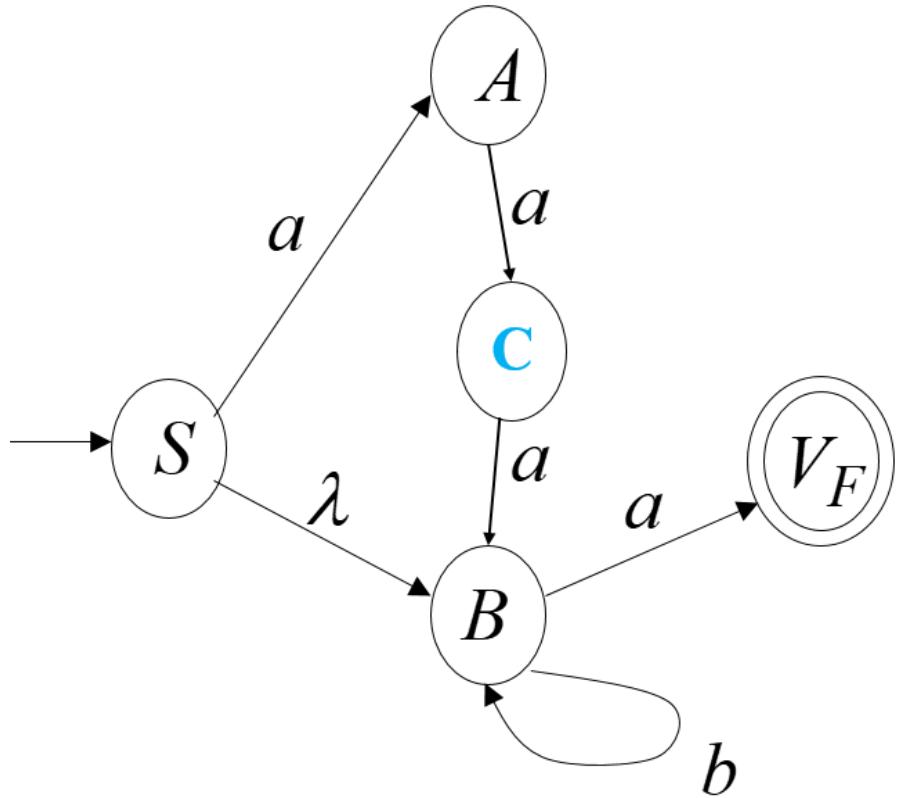
$$M = (Q, \Sigma, \delta, S, F)$$

$$Q = \{S, A, B, C, V_F\}$$

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

$$F = \{V_F\}$$

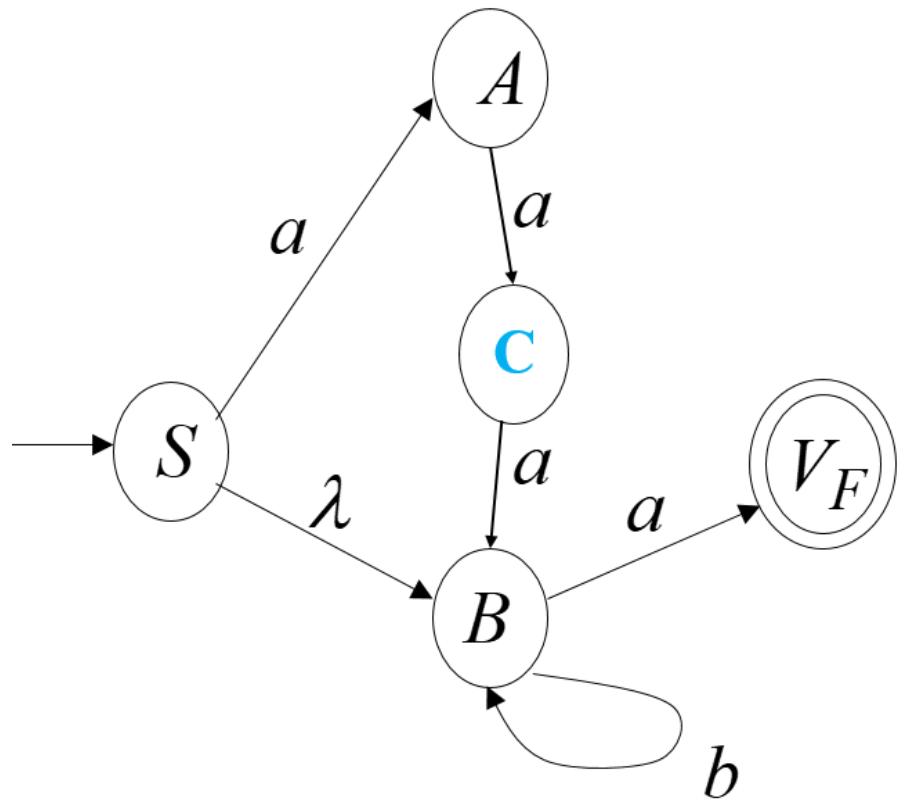
# Right-Linear Grammar To NFA



$\delta^*(S, aaaba)$   
=  $\delta(\delta^*(S, aaab), a)$   
=  $\delta(\delta(\delta^*(S, aaa), b), a)$   
=  $\delta(\delta(\delta(\delta^*(S, aa), a), b), a)$   
=  $\delta(\delta(\delta(\delta(\delta^*(S, a), a), a), b), a)$   
=  $\delta(\delta(\delta(\delta(\delta(\delta^*(S, \lambda), a), a), a), b), a)$   
=  $\delta(\delta(\delta(\delta(\delta(S, a), a), a), b), a)$   
=  $\delta(\delta(\delta(\delta(A, a), a), b), a)$   
=  $\delta(\delta(\delta(C, a), b), a)$   
=  $\delta(\delta(B, b), a)$   
=  $\delta(B, a) = V_F$

$S \Rightarrow aA \Rightarrow aaaB \Rightarrow aaabB \Rightarrow aaaba$

NFA  $M$



Grammar  $G$

$$S \rightarrow aA \mid B$$

$$A \rightarrow aa \ B$$

$$B \rightarrow bB \mid a$$

$$L(M) = L(G) = L( \text{aaab}^*a + b^*a )$$

## Find NFA

$L = \{w \in \{a, b\}^*: w \text{ ends with the pattern } aaaa\}$ .

- Regular grammar:  $G = (V, T, S, P)$ , where
- $V = \{S, B, C, D\}$ ,  $T = \{a, b\}$ ,
- $P = \{S \rightarrow aS, S \rightarrow bS, S \rightarrow aB, B \rightarrow aC, C \rightarrow aD, D \rightarrow a\}$

## Exercise-1

$L = \{w \in \{a, b\}^*: |w| \text{ is even}\}$

Regular grammar:  $G = (V, \Sigma, S, P)$ , where

$V = \{S, T\}$ ,  $\Sigma = \{a, b\}$ ,

$P = \{S \rightarrow \epsilon, S \rightarrow aT, S \rightarrow bT, T \rightarrow aS, T \rightarrow bS\}$

## Exercise-2

## Exercise-3

$S \rightarrow \epsilon$	$A \rightarrow bA$	$C \rightarrow aC$
$S \rightarrow aB$	$A \rightarrow cA$	$C \rightarrow bC$
$S \rightarrow aC$	$A \rightarrow \epsilon$	$C \rightarrow \epsilon$
$S \rightarrow bA$	$B \rightarrow aB$	
$S \rightarrow bC$	$B \rightarrow cB$	
$S \rightarrow cA$	$B \rightarrow \epsilon$	
$S \rightarrow cB$		

# Find NFA

## Exercise-1

- Regular grammar:  $G = (V, T, S, P)$ , where
- $V = \{S, B, C, D\}$ ,  $T = \{a, b\}$ ,
- $P = \{S \rightarrow Sa, S \rightarrow Sb, S \rightarrow Ba, B \rightarrow Ca, C \rightarrow Da, D \rightarrow a\}$

Regular grammar:  $G = (V, \Sigma, S, P)$ , where

$V = \{S, T\}$ ,  $\Sigma = \{a, b\}$ ,

$P = \{S \rightarrow \epsilon, S \rightarrow Ta, S \rightarrow Tb, T \rightarrow Sa, T \rightarrow Sb\}$

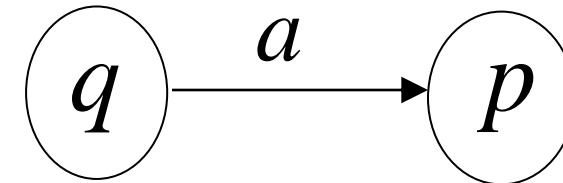
## Exercise-2

## Exercise-3

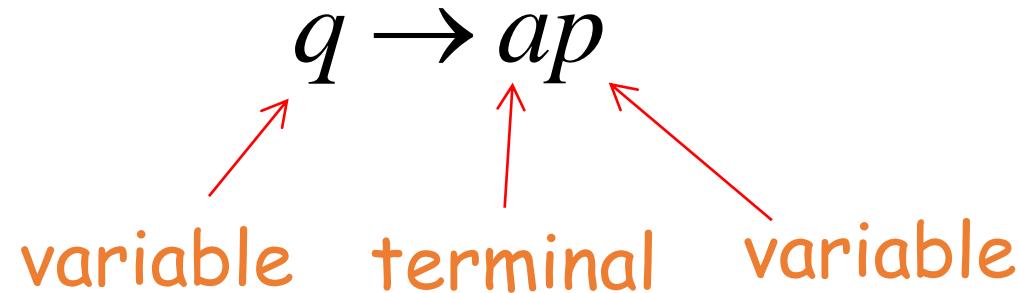
$S \rightarrow \epsilon$	$A \rightarrow Ab$	$C \rightarrow Ca$
$S \rightarrow Ba$	$A \rightarrow Ac$	$C \rightarrow Cb$
$S \rightarrow Ca$	$A \rightarrow \epsilon$	$C \rightarrow \epsilon$
$S \rightarrow Ab$	$B \rightarrow Ba$	
$S \rightarrow Cb$	$B \rightarrow Bc$	
$S \rightarrow Ac$	$B \rightarrow \epsilon$	
$S \rightarrow Bc$		

# NFA-to-Regular Grammar (Right-Linear Grammar): Rules

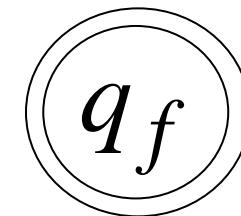
For any transition:



Add production:



For any final state:



Add production:

$$q_f \rightarrow \lambda$$

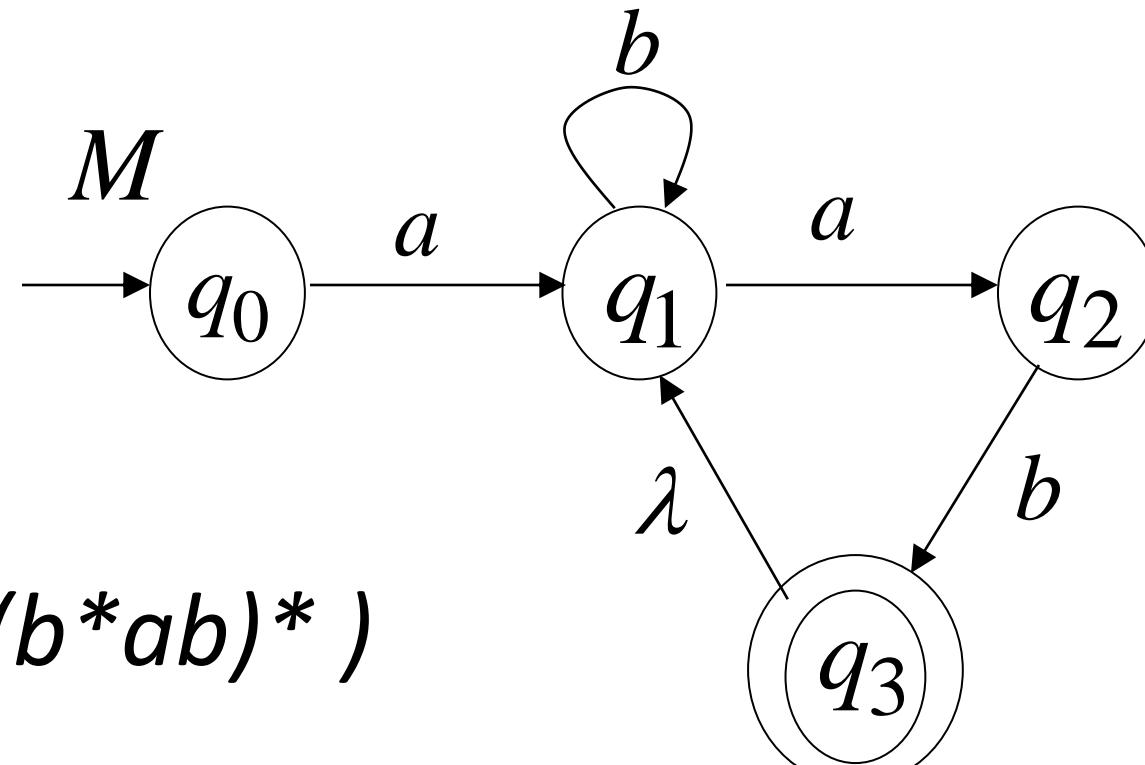
# NFA-to-Regular Grammar

Since  $L$  is regular

there is an NFA  $M$  such that

$$L = L(M)$$

Example:



$$L = L( ab^*ab(b^*ab)^* )$$

$$L = L(M)$$

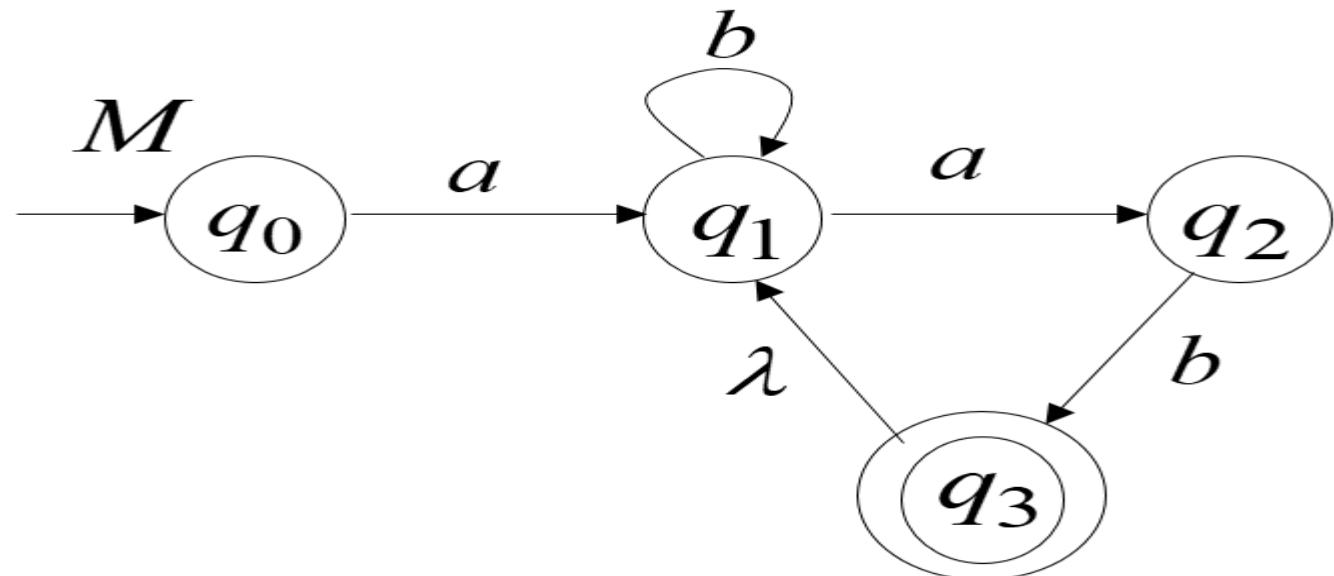
# NFA-to-RG

Convert  $M$  to a right-linear grammar  $G$

$$q_0 \rightarrow aq_1$$

$$q_1 \rightarrow bq_1$$

$$q_1 \rightarrow aq_2$$



# NFA-to-RG

Convert  $M$  to a right-linear grammar  $G$

$$q_0 \rightarrow aq_1$$

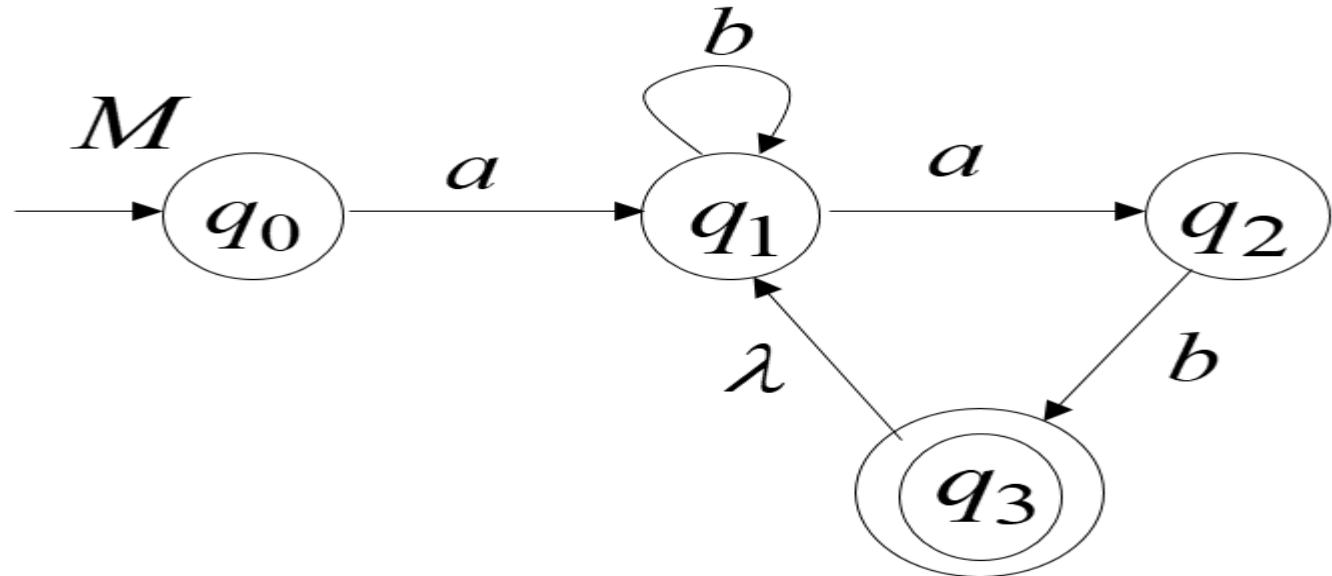
$$q_1 \rightarrow bq_1$$

$$q_1 \rightarrow aq_2$$

$$q_2 \rightarrow bq_3$$

$$q_3 \rightarrow q_1$$

$$q_3 \rightarrow \lambda$$



right-linear grammar  
(Regular Grammar)

$$q_0 \rightarrow aq_1$$

$$G \quad q_1 \rightarrow bq_1$$

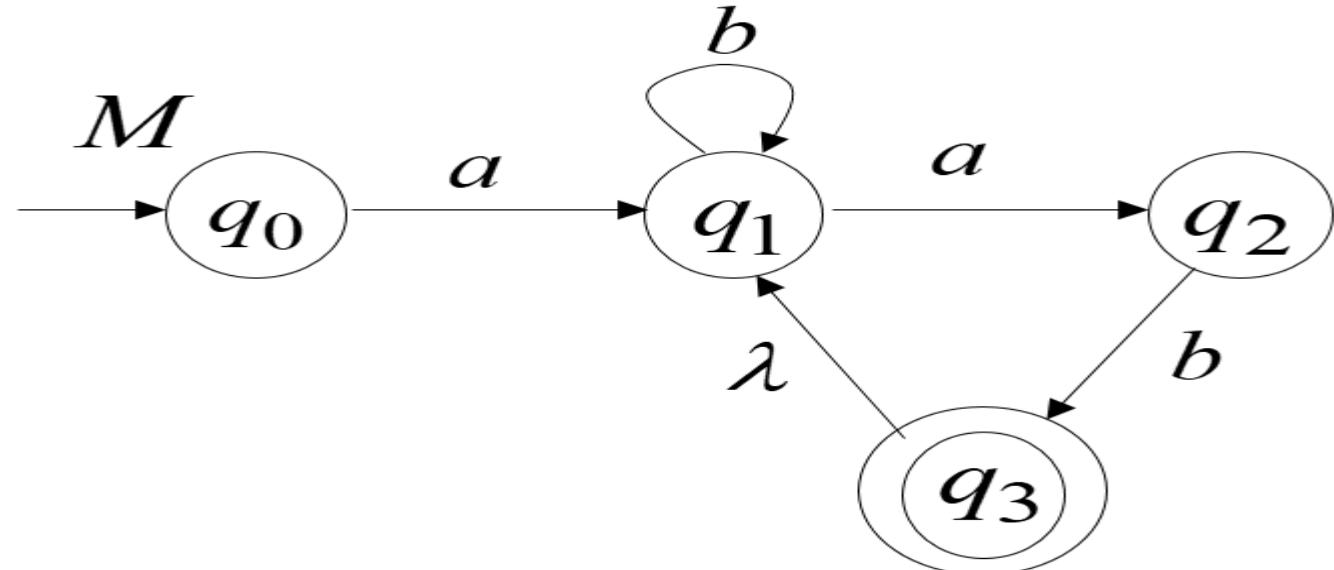
$$q_1 \rightarrow aq_2$$

$$q_2 \rightarrow bq_3$$

$$q_3 \rightarrow q_1$$

$$q_3 \rightarrow \lambda$$

## NFA-to-RG

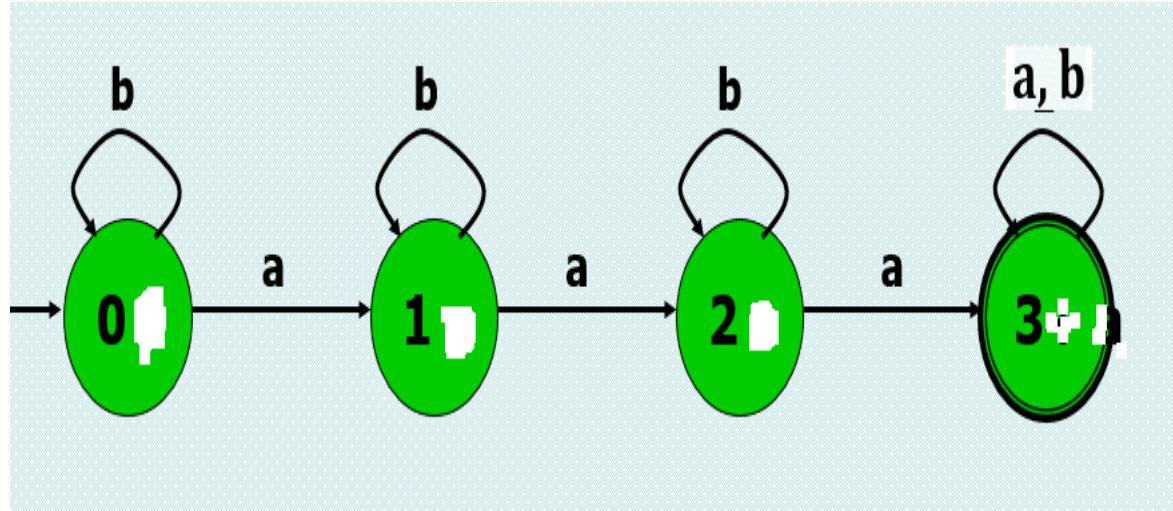
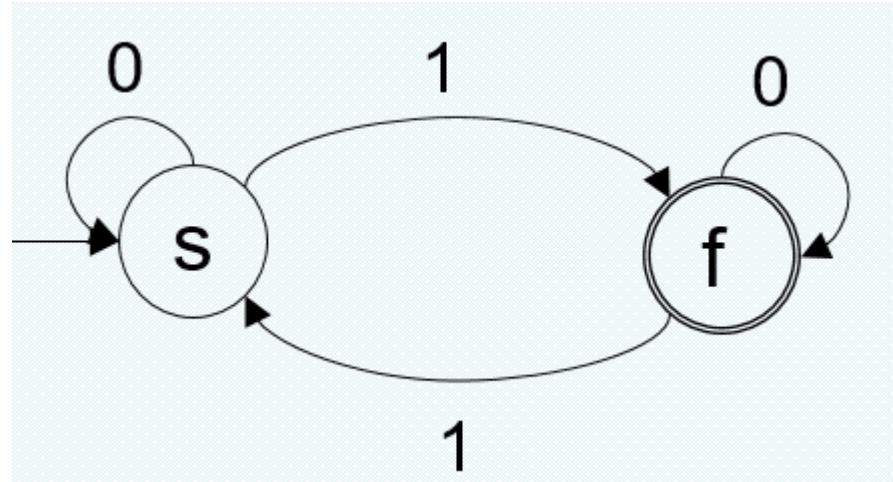
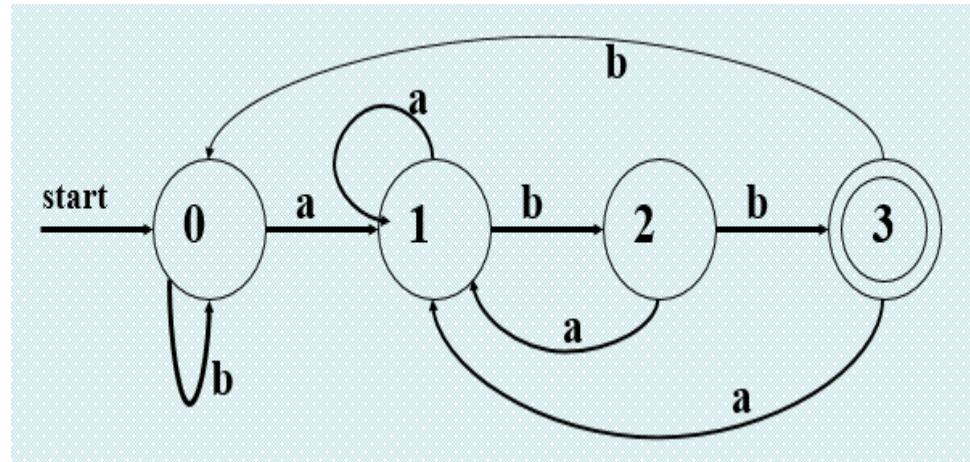


$$L(M) = L( ab^*ab(b^*ab)^* )$$

$$L(G) = L( ab^*ab(b^*ab)^* )$$

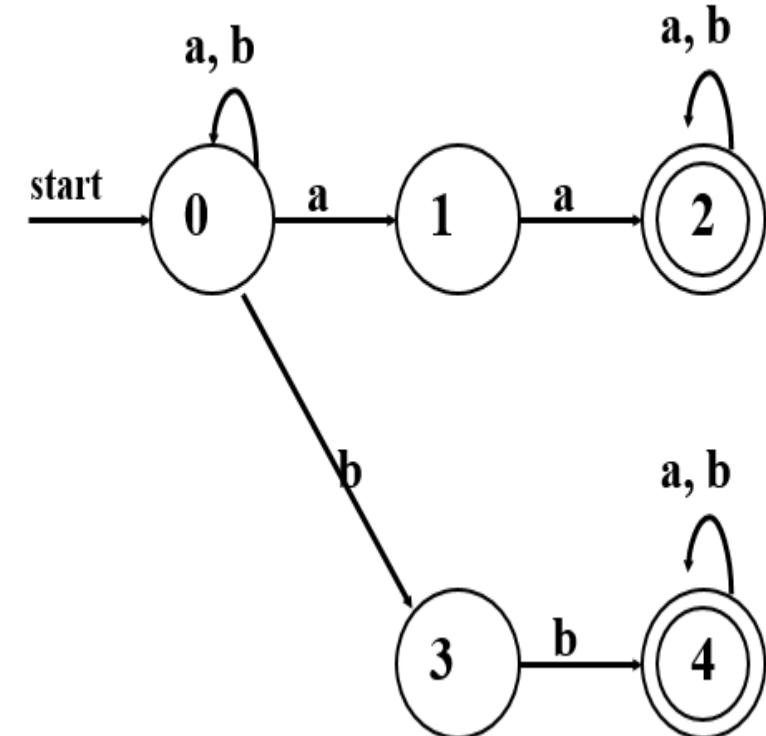
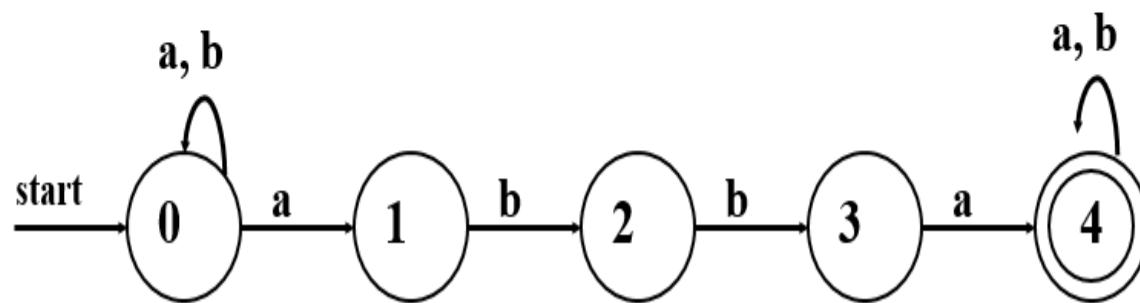
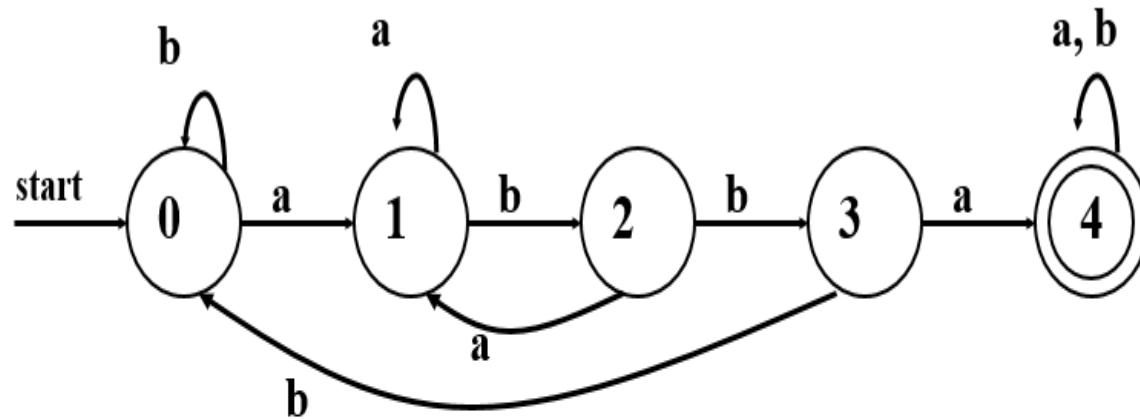
$$L(G) = L(M) = L$$

# Find Regular Grammar(Right-Linear)



$s \rightarrow 0s$   
 $s \rightarrow 1f$   
 $f \rightarrow 0f$   
 $f \rightarrow 1s$   
 $f \rightarrow e$

# Find Regular Grammar(Right-Linear)

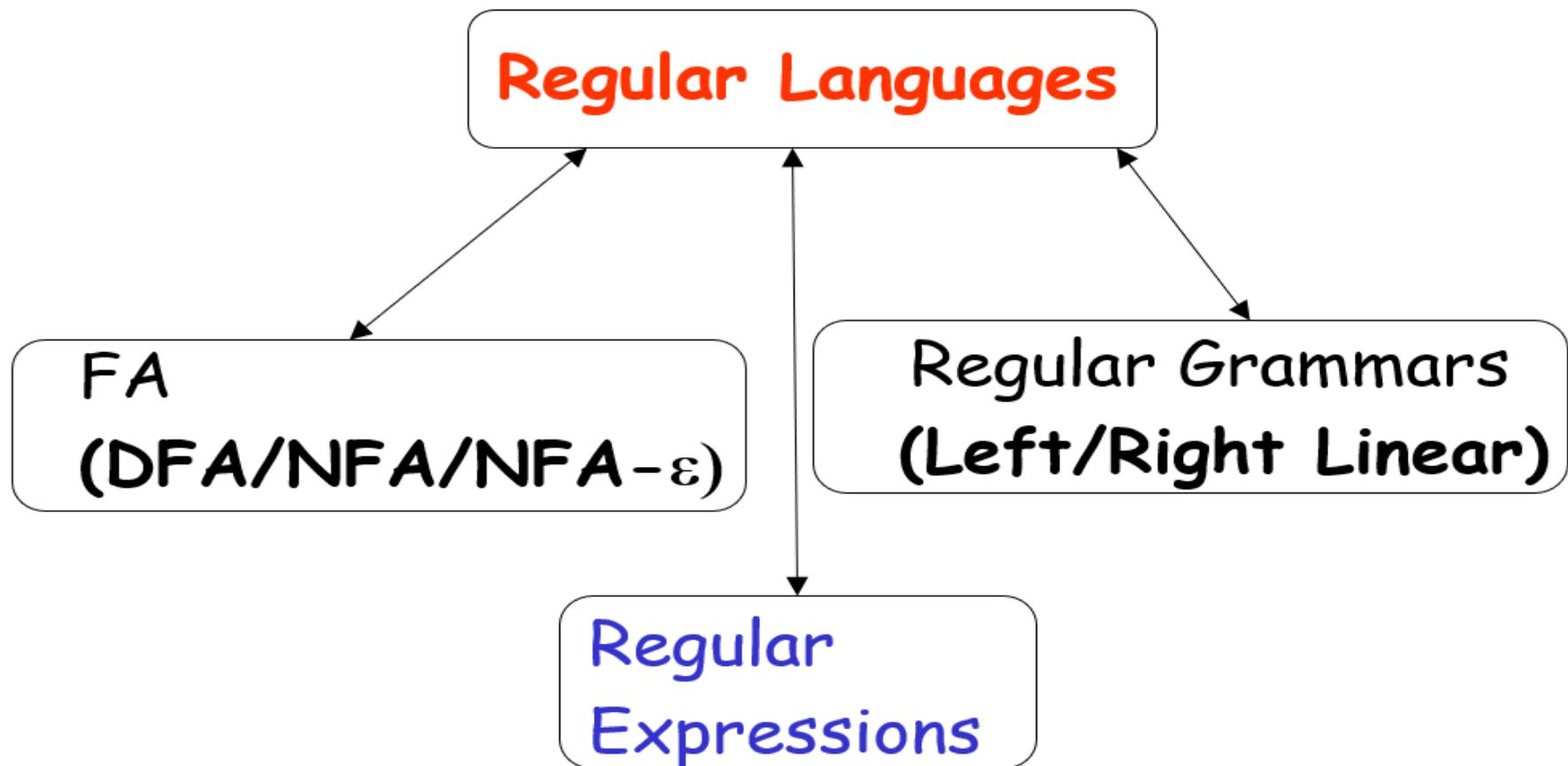


# Properties of Regular Languages

# Standard Representations of Regular Languages

A language  $L$  over an alphabet  $\Sigma$  is **regular iff**

- It is accepted by FA (DFA, NFA, or NFA- $\epsilon$ ).
- It is described by a regular expression.
- It is generated by regular grammar (Left-linear or Right-linear).



# Closure Properties

- Recall a closure property is a statement that a certain operation on languages, when applied to languages in a class, produces a result that is also in that class.

Given two regular languages  $L_1$  and  $L_2$ , **is their union is also regular ?**

If it is true for all regular languages, then family of regular languages is closed under union.

- For regular languages, we can use any of its representations to prove a closure property.

# Properties of RLs

For regular languages  $L_1$  and  $L_2$ ,  
we will prove that:

Union:  $L_1 \cup L_2$

Concatenation:  $L_1 L_2$

Star:  $L_1^*$

Reversal:  $L_1^R$

Complement:  $\overline{L_1}$

Intersection:  $L_1 \cap L_2$

Difference:  $L_1 - L_2$



Are regular  
Languages

It means the family of Regular languages are  
**closed under above operations.**

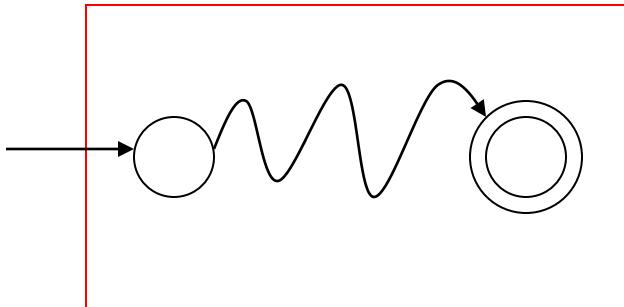
# Example

Regular language  $L_1$    Regular language  $L_2$

$$L(M_1) = L_1$$

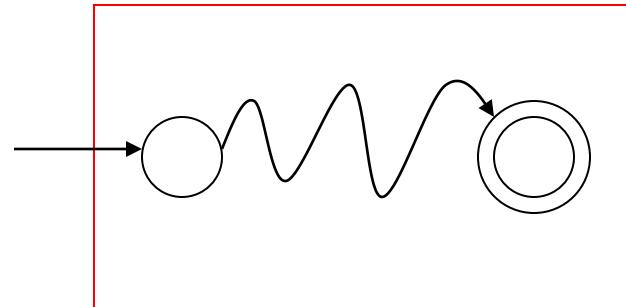
$$L(M_2) = L_2$$

NFA  $M_1$



Single final state

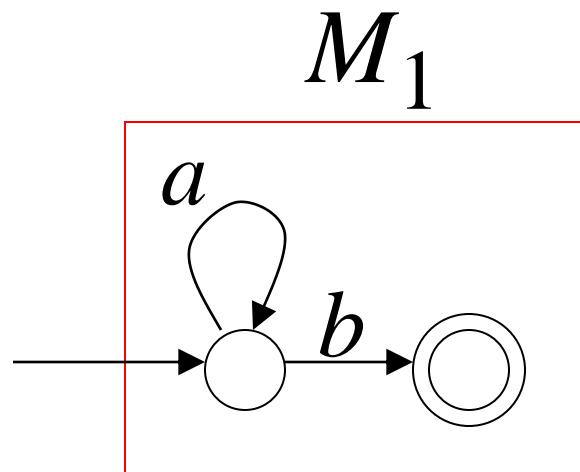
NFA  $M_2$



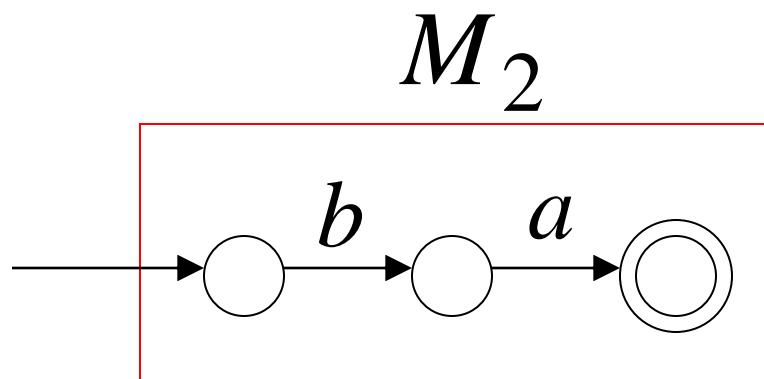
Single final state

# Example

$$L_1 = \{ a^n b : n \geq 0 \}$$

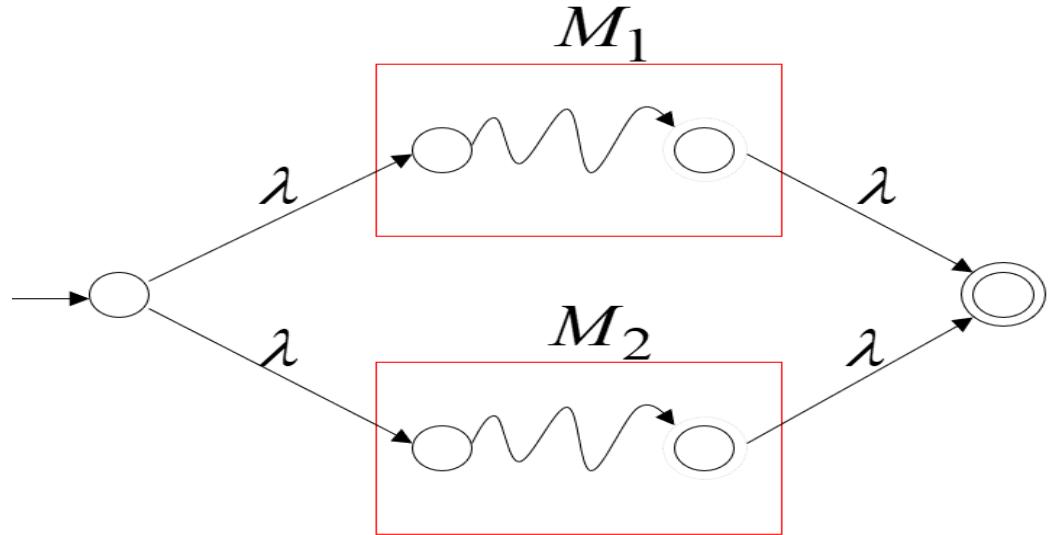


$$L_2 = \{ ba \}$$



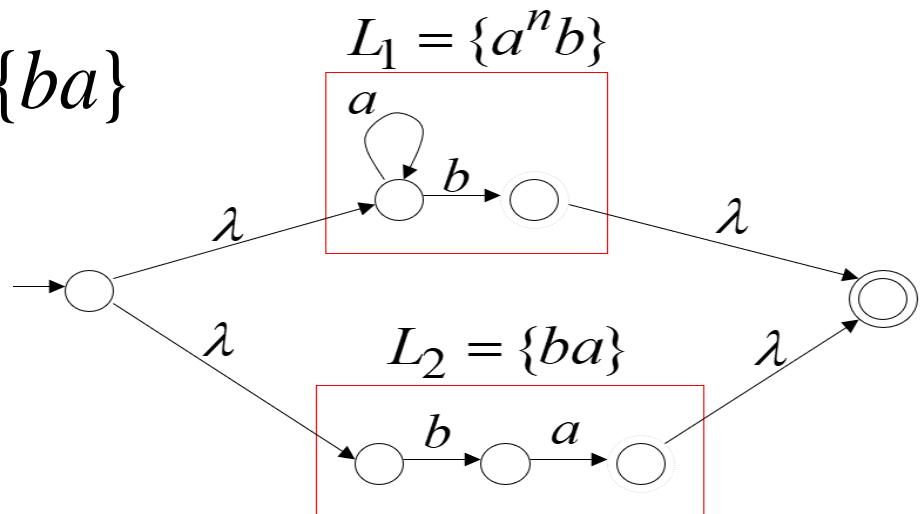
# Union

NFA for  $L_1 \cup L_2$



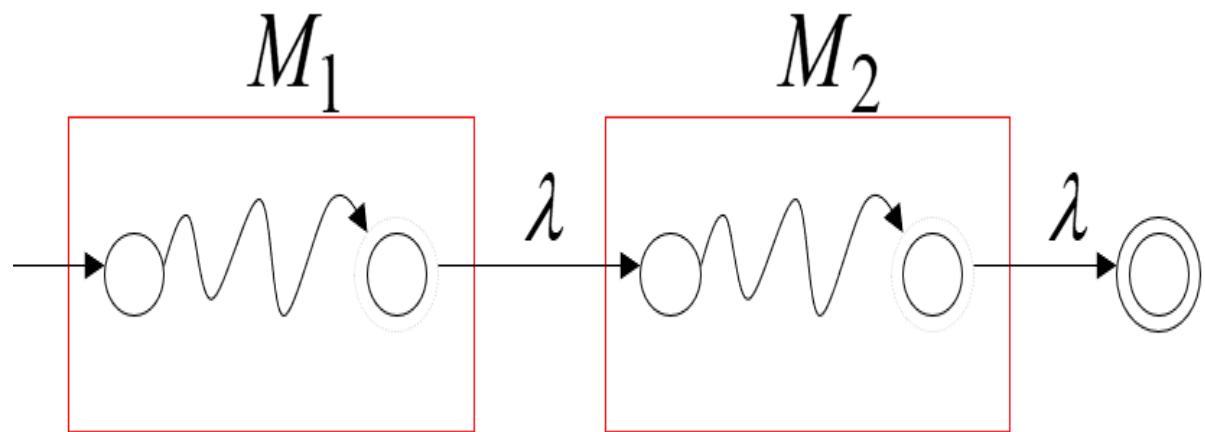
Example: NFA for

$$L_1 \cup L_2 = \{a^n b\} \cup \{ba\}$$



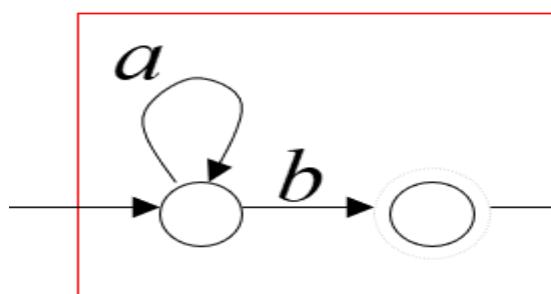
# Concatenation

NFA for  $L_1 L_2$

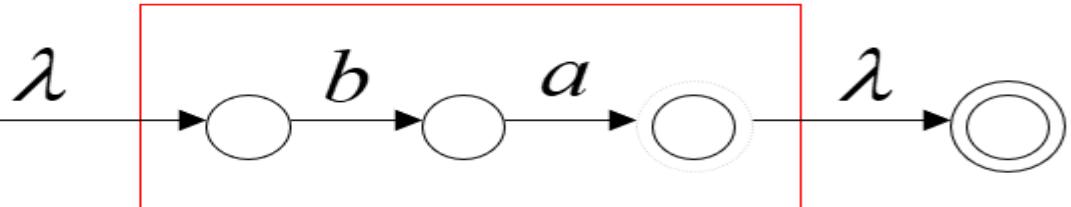


NFA for  $L_1 L_2 = \{a^n b\} \{ba\} = \{a^n bba\} \quad n \geq 0$

$$L_1 = \{a^n b\}$$

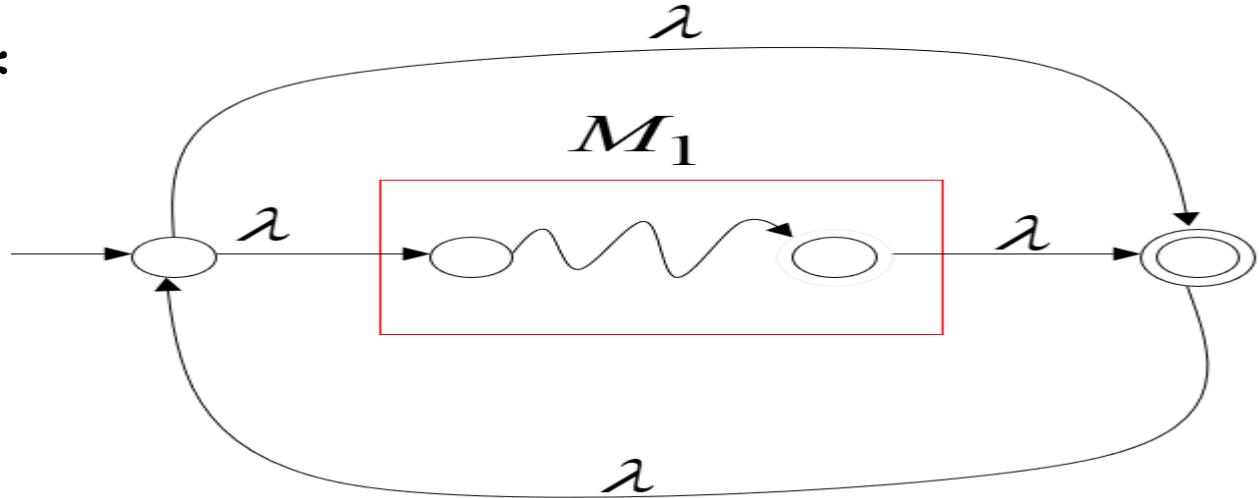


$$L_2 = \{ba\}$$



# Star Operation

NFA for  $L_1^*$

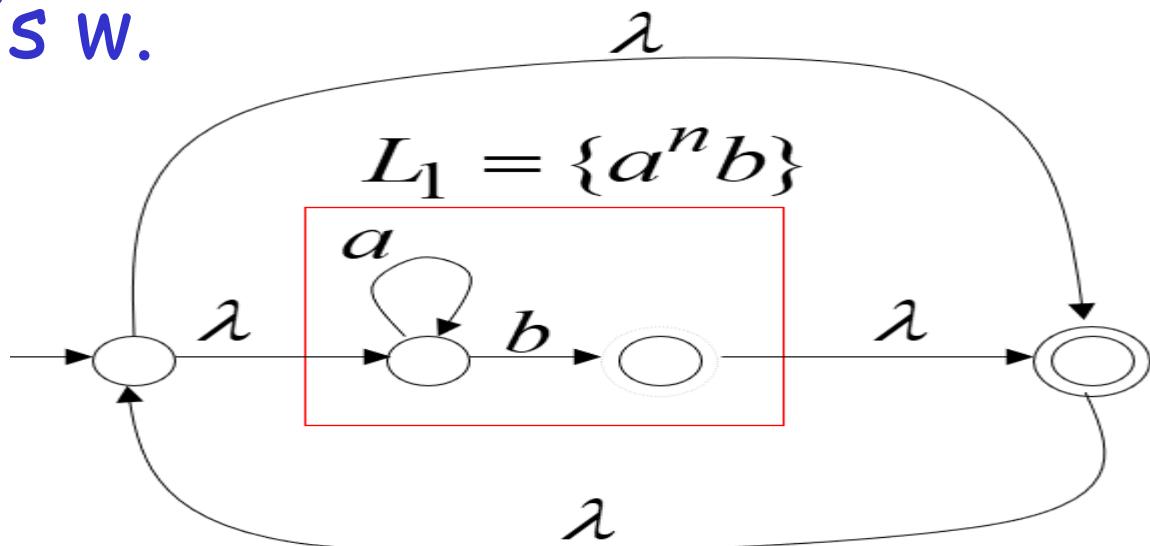


NFA for  $L_1^* = \{a^n b\}^*$

The string in  $L_1^*$  is w.

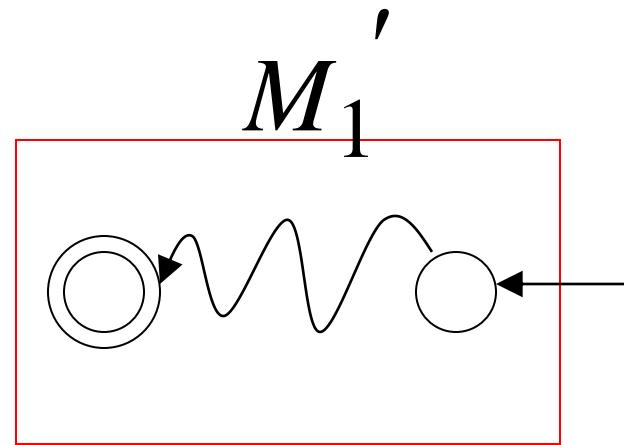
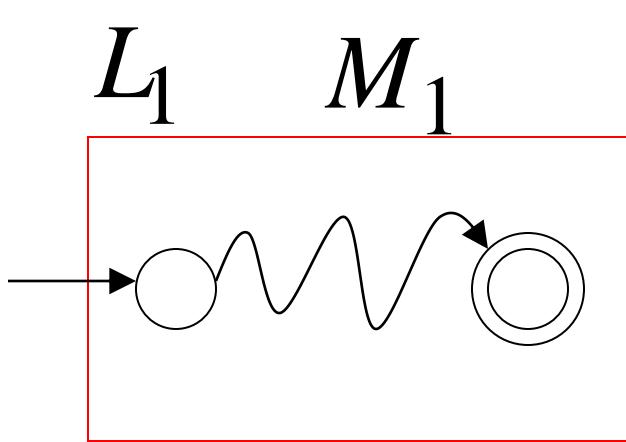
$$w = w_1 w_2 \Delta w_k$$

$$w_i \in L_1$$



# Reversal

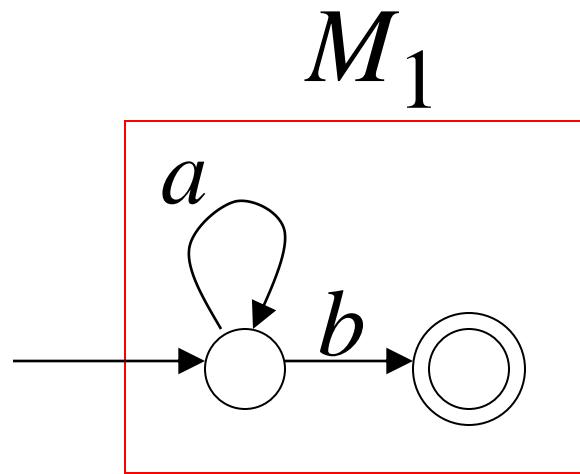
NFA for  $L_1^R$



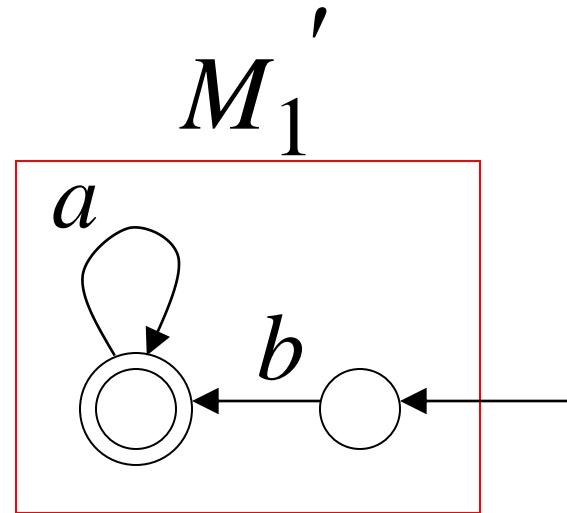
1. Reverse all transitions
2. Mark initial state as final state and vice versa

# Example

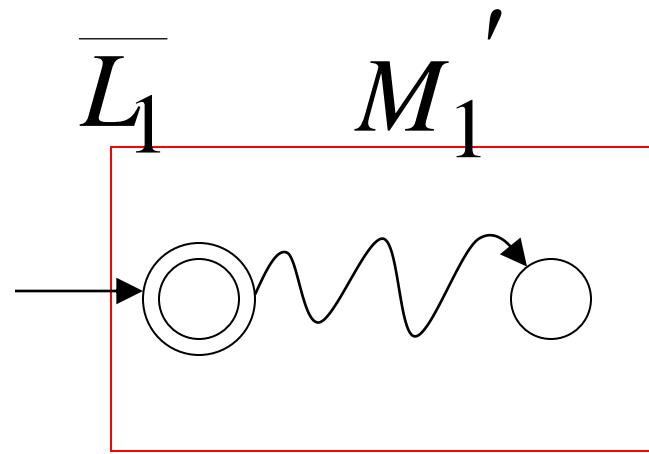
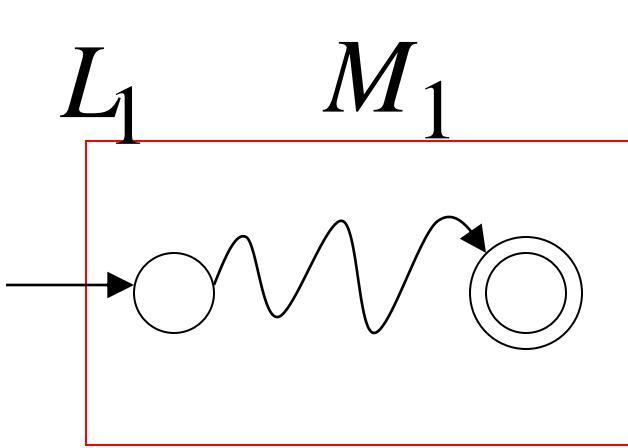
$$L_1 = \{a^n b\}$$



$${L_1}^R = \{ba^n\}$$



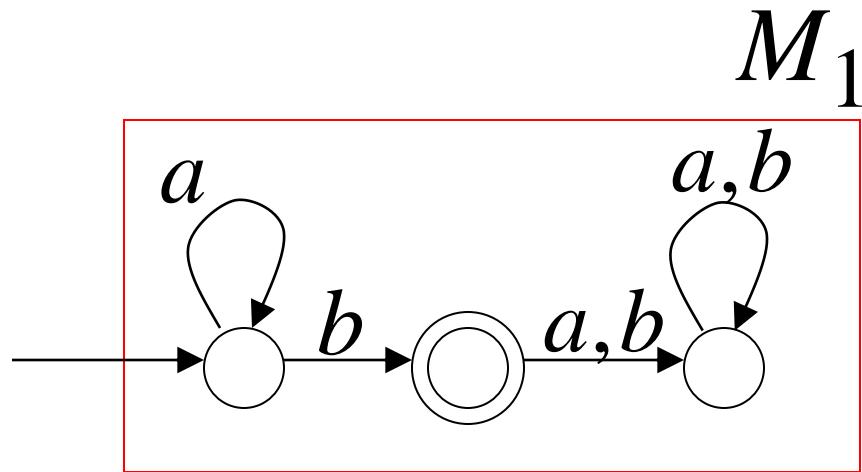
# Complement



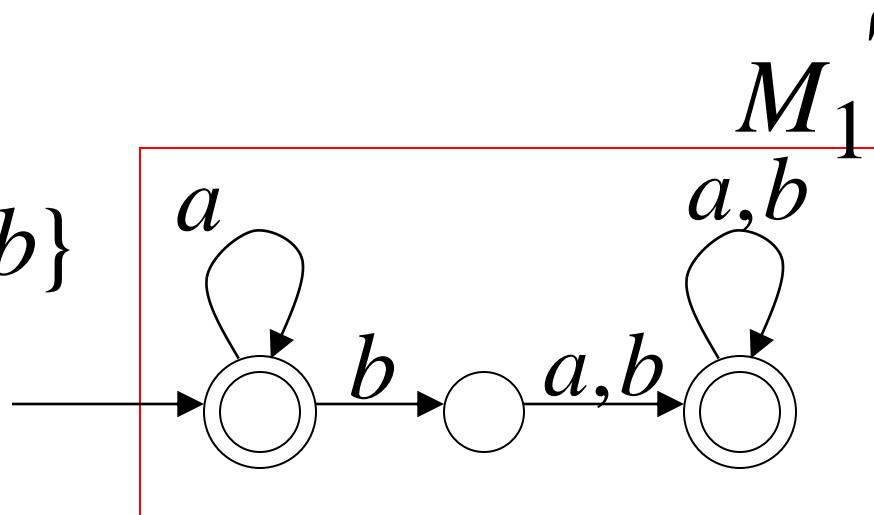
1. Take the **DFA** that accepts  $L_1$
2. Make final states non-final, and vice-versa.

# Example

$$L_1 = \{a^n b\}$$



$$\overline{L_1} = \{a,b\}^* - \{a^n b\}$$



# Intersection

DeMorgan's Law:  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

$L_1, L_2$  regular

  $\overline{L_1}, \overline{L_2}$  regular

  $\overline{\overline{L_1} \cup \overline{L_2}}$  regular

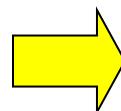
  $\overline{\overline{L_1} \cup \overline{L_2}}$  regular

  $L_1 \cap L_2$  regular

# Example

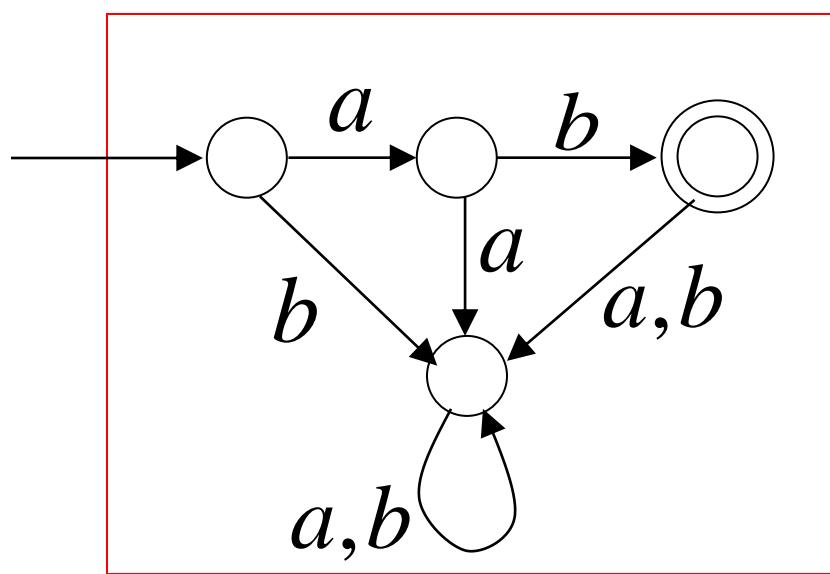
$L_1 = \{a^n b\}$  regular

$L_2 = \{ab, ba\}$  regular



$L_1 \cap L_2 = \{ab\}$

regular



# Intersection (Product)

$L_1 = L(M_1)$ ,  $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$ ,  $Q = \{q_0, q_1, \dots, q_m\}$

$L_2 = L(M_2)$ ,  $M_2 = (P, \Sigma, \delta_2, p_0, F_2)$ ,  $P = \{p_0, p_1, \dots, p_n\}$

$M_1$  and  $M_2$  are DFAs.

Define new automation,  $M' = (Q', \Sigma, \delta', (q_0, p_0), F')$

$$Q' = Q \times P = \{(q_i, p_j) : q_i \in Q, p_j \in P\}$$

$$Q' = Q \times P = \{A, B\} \times \{C, D\} = \{(A, C), (A, D), (B, C), (B, D)\}$$

Define transition function  $\delta'$  s.t.

$$\delta'(q_i, p_j, a) = (q_k, p_l)$$

If  $\delta_1(q_i, a) = q_k$

AND  $\delta_2(p_j, a) = p_l$

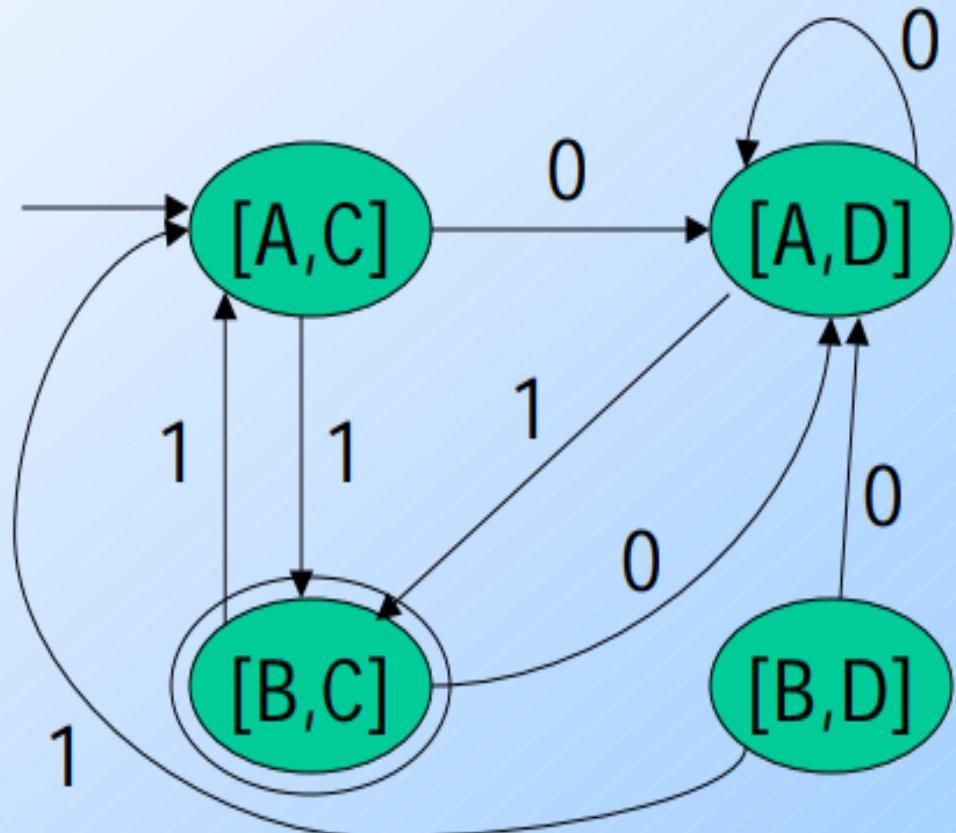
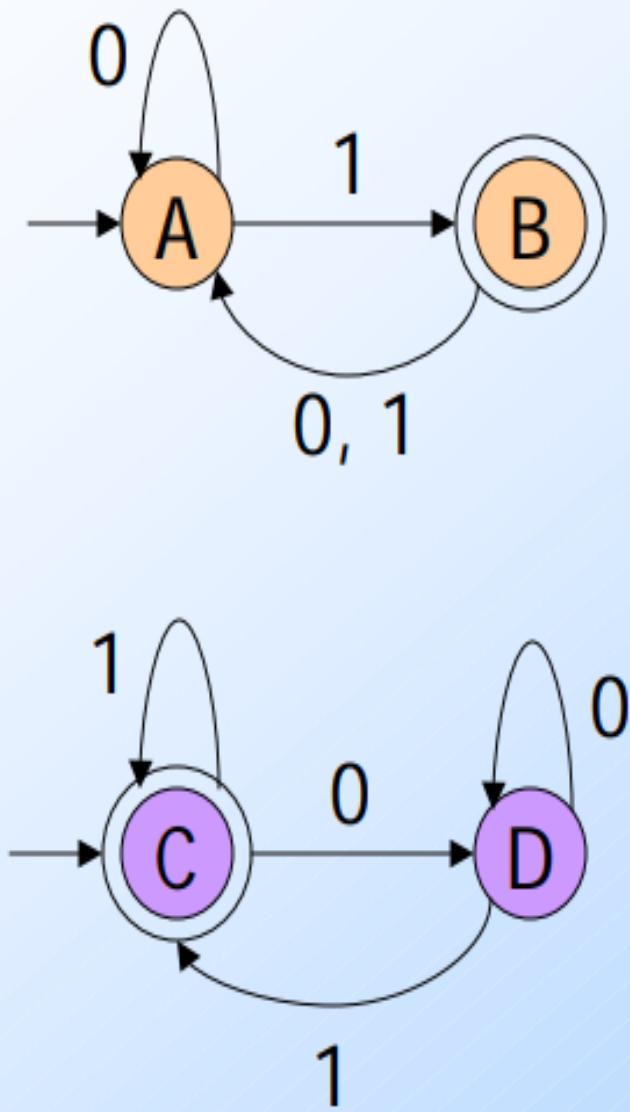
Define transition for all states.

The initial state is  $(q_0, p_0)$ .

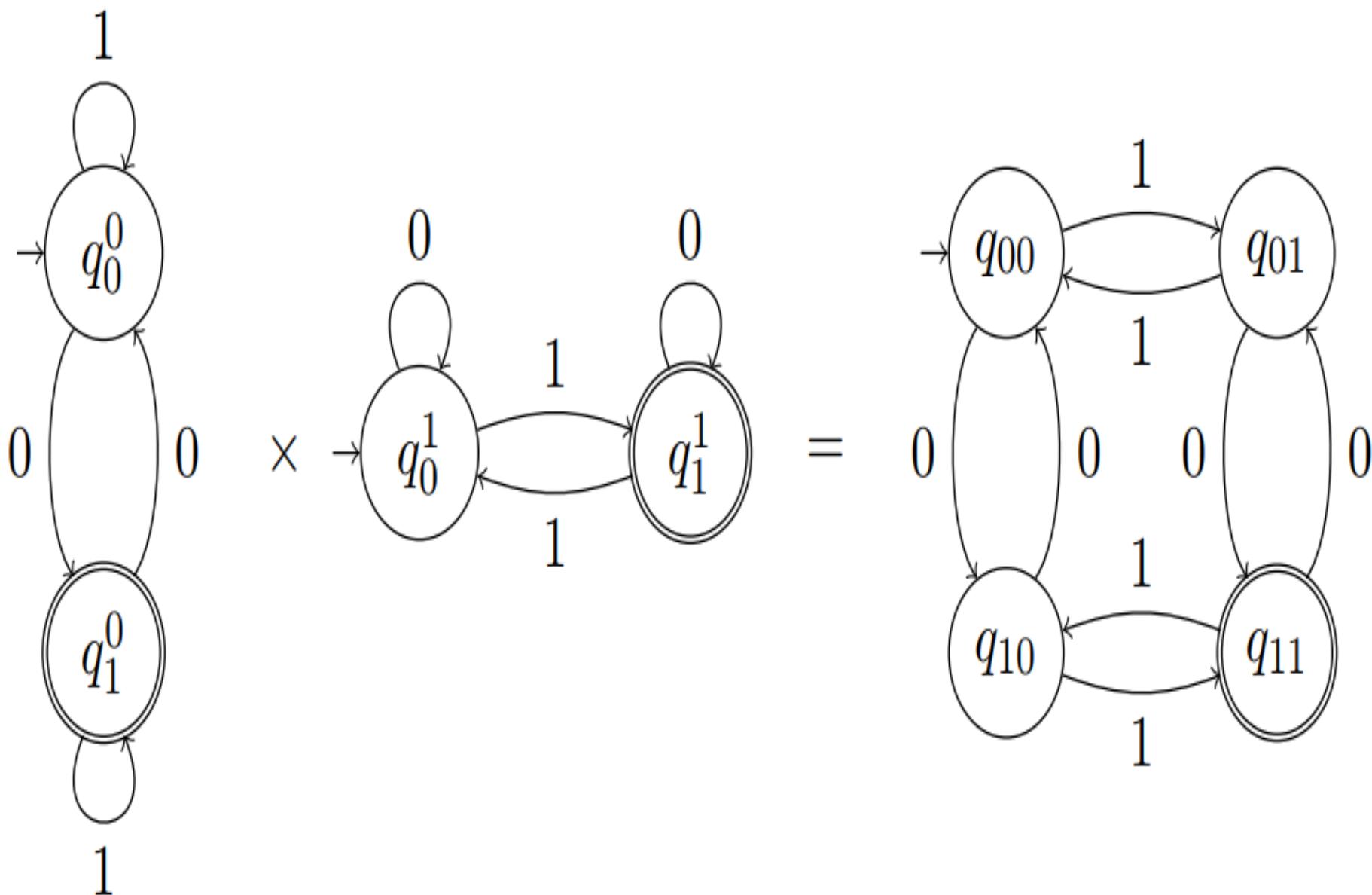
$F'$  is the set of all states s.t.  $(q_i, p_j)$  s.t.

$$q_i \in F_1, p_j \in F_2$$

# Example-1

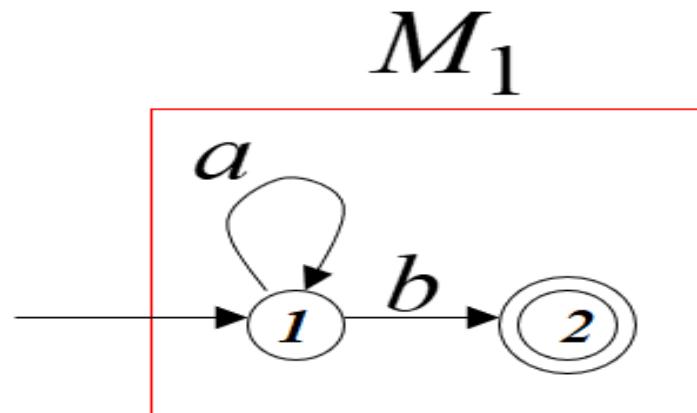


# Example-2

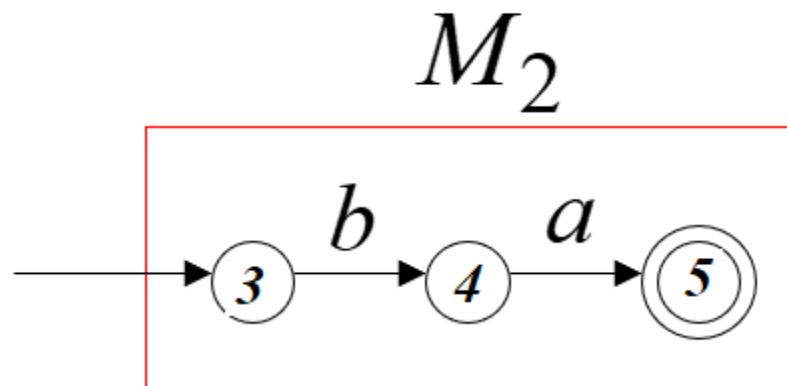


# Find product

$$n \geq 0$$
$$L_1 = \{a^n b\}$$



$$L_2 = \{ba\}$$



# DIFFERENCE

$L_1 = L(M_1)$ ,  $M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$ ,  $Q = \{q_0, q_1, \dots, q_{m_1}\}$

$L_2 = L(M_2)$ ,  $M_2 = (P, \Sigma, \delta_2, p_0, F_2)$ ,  $P = \{p_0, p_1, \dots, p_{n_2}\}$

$M_1$  and  $M_2$  are DFAs

$$L_1 - L_2 = L_1 \cap \overline{L_2}$$

$L_1, L_2$  regular

  $L_1, \overline{L_2}$  regular

  $L_1 \cap \overline{L_2}$  regular

# DIFFERENCE

If  $L_1$  and  $L_2$  are regular languages, So is  $L_1 - L_2$

$L_1 - L_2$  consists of strings in  $L_1$  but not  $L_2$  i.e.,

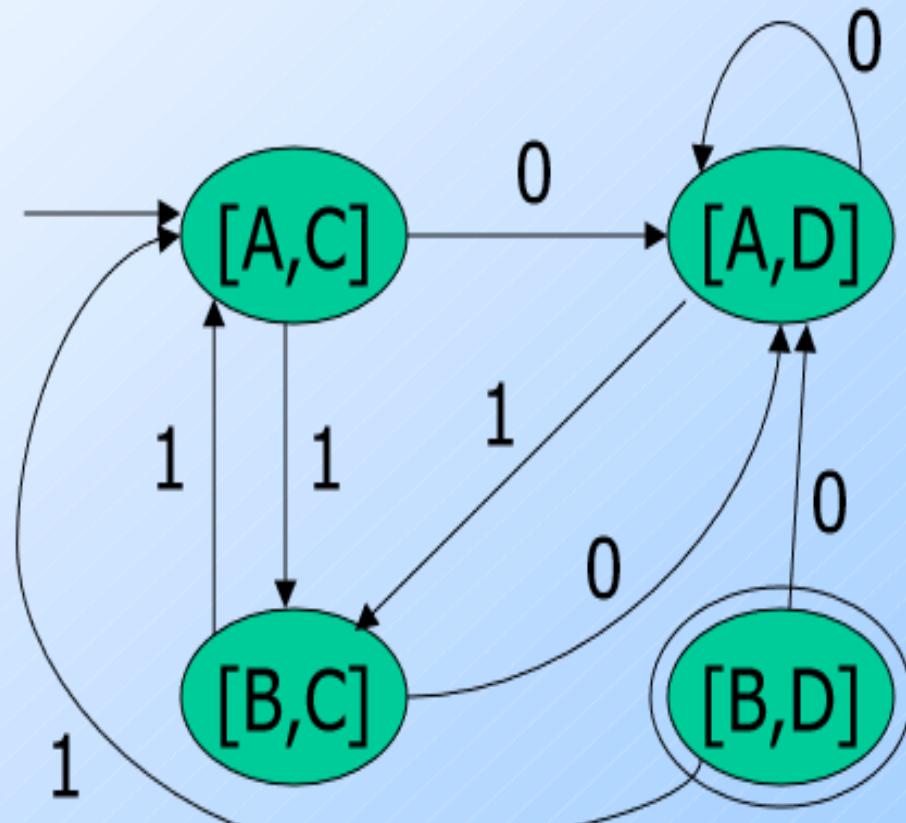
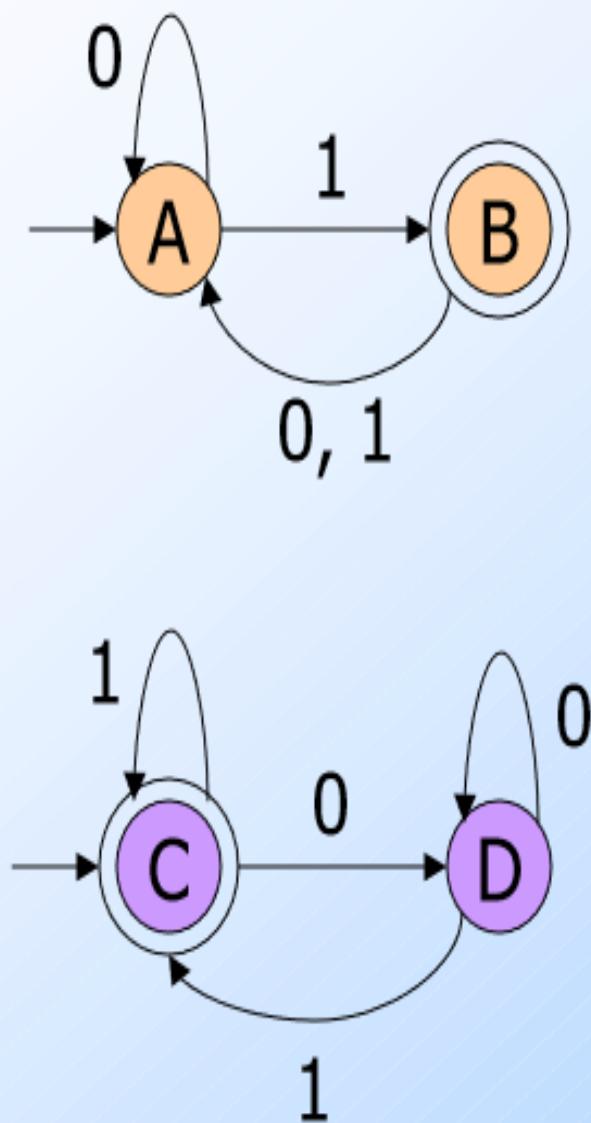
$$L_1 - L_2 = L_1 \cap \overline{L_2}$$

Let  $M_1$  and  $M_2$  be DFA's whose languages are  $L_1$  and  $L_2$ , respectively.

Construct  $C$ , the product automaton of  $M_1$  and  $M_2$ .

Make the final states of  $C$  be the pairs where  $M_1$ -state is final but  $M_2$ -state is not.

# Example



Notice: difference  
is the empty language

# Exercise

$$L_1 = L((a+b)a^*)$$

$$L_2 = L(baa^*)$$

For above given languages  $L_1$  and  $L_2$ , construct the FAs for.

- Union
- Star closure
- Concatenation
- Complement
- Reversal
- Intersection
- Difference

# Exercise

$L_1 = \{w \in \{0, 1\}^*: w \text{ consists of } 0's \text{ in multiple of } 3\}$

$L_2 = \{w \in \{0, 1\}^*: w \text{ consists of odd number of } 0's\}$

For given languages  $L_1$  and  $L_2$ , find FA for

- Union
- Star closure
- Concatenation
- Complement
- Reversal
- Intersection
- Difference

# Elementary Questions

about

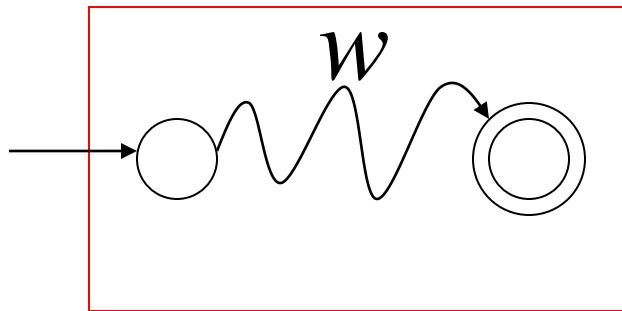
# Regular Languages

# Membership Question

**Question-1** Given regular language  $L$  and string  $w$  how can we check if  $w \in L$ ?

**Answer:** Take the DFA that accepts  $L$  and check if  $w$  is accepted

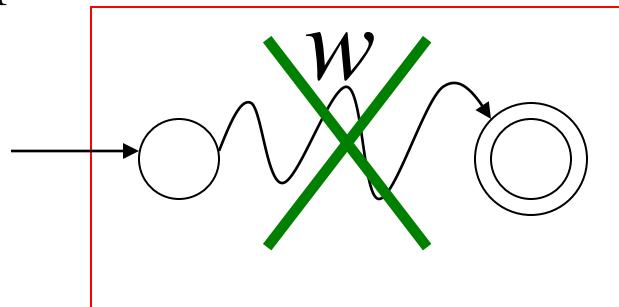
DFA



$w \in L$

If there is a path from initial state to final state with labeled  $w$ ; Accepted

DFA



$w \notin L$

If there is **no path** from initial state to final state with labeled  $w$ ; Rejected

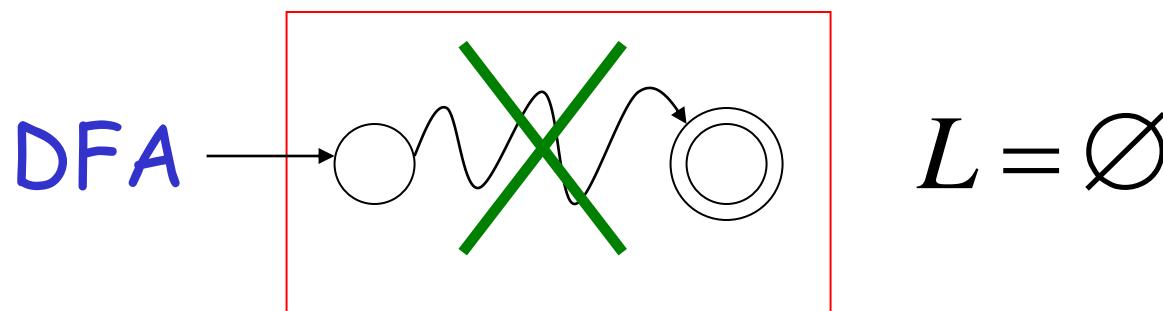
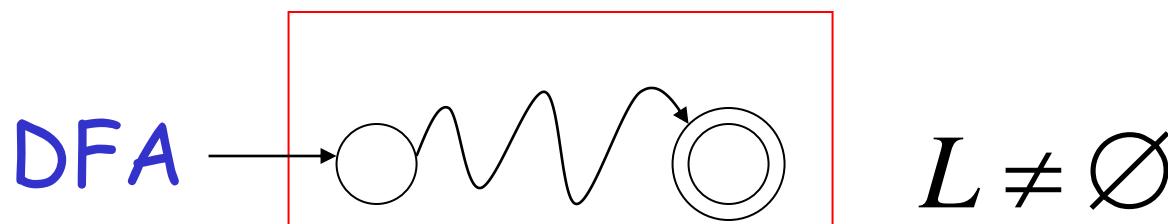
Given regular language  $L$

**Question-2** how can we check  
if  $L$  is empty: ( $L = \emptyset$ )?

Take the DFA that accepts  $L$

**Answer:**

Check if there is any path from  
the initial state to a final state



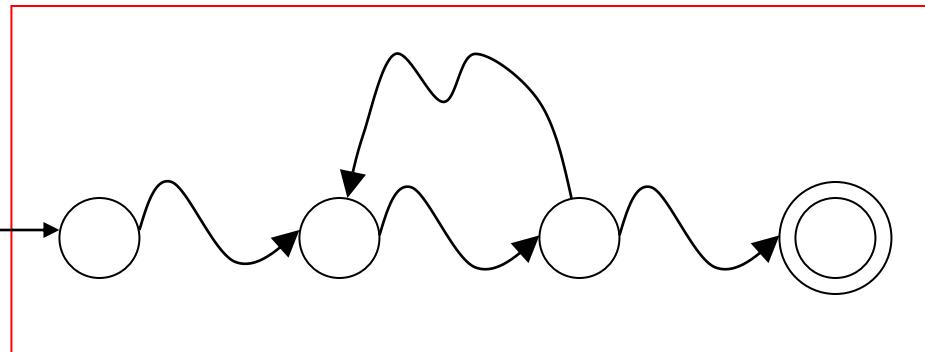
Given regular language  $L$

**Question-4** how can we check  
if  $L$  is finite?

Take the DFA that accepts  $L$

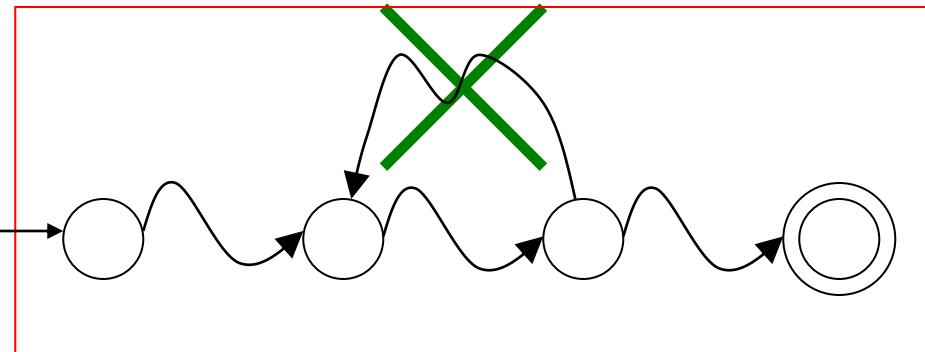
**Answer:** Check if there is **a walk with cycle**  
from the initial state to a final state

DFA



$L$  is infinite

DFA



$L$  is finite

# Non-regular languages

# Regular Languages

Every finite language is regular.

Some infinite languages are regular

$$L_1 = \{a^n b : n \geq 0\} \rightarrow aaab$$

**Regularity:**

The language is regular if, in processing any string, the information that has to be remembered at any stage is strictly limited.

Some infinite languages are not regular

$$L_2 = \{a^n b^n : n \geq 0\} \rightarrow aaabbb$$

How can we prove that a language  $L$   
is regular?

Prove that there is DFA  $M$  that accepts  $L$

Prove that there is RE  $r$  that generates  $L$

Prove that there is RG  $G$  that generates  $L$

Non-regular languages

$$\{a^n b^n : n \geq 0\}$$

$$\{vv^R : v \in \{a,b\}^*\}$$

Regular languages

$$a^*b$$

$$b^*c + a$$

$$b + c(a+b)^*$$

etc...

How can we prove that a language  $L$  is not regular?

Prove that there is no DFA that accepts  $L$

**Problem:** this is not easy to prove

**Solution:** the Pumping Lemma !!!

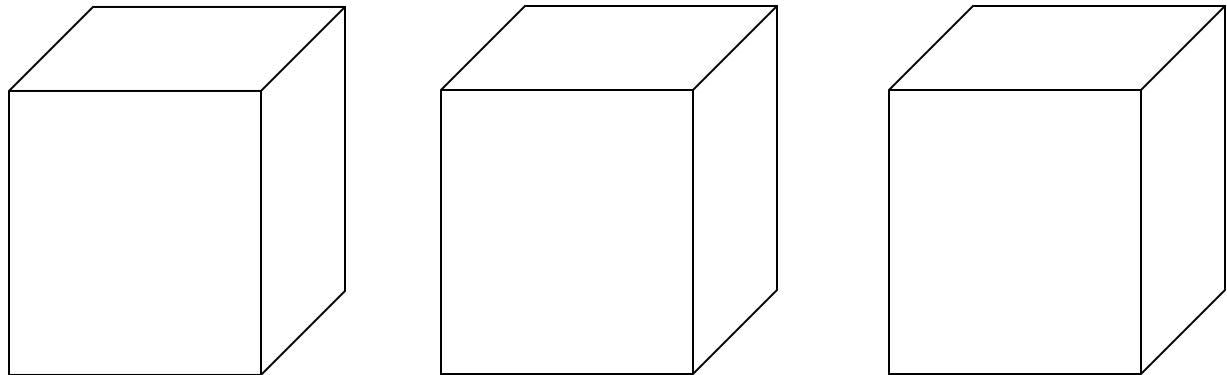
The pumping Lemma uses pigeonhole principle.

# The Pigeonhole Principle

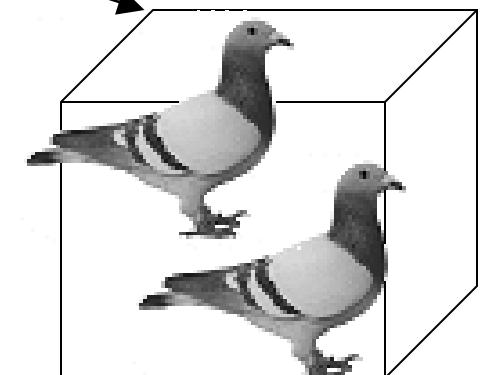
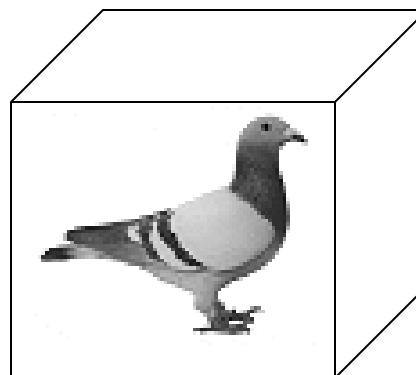
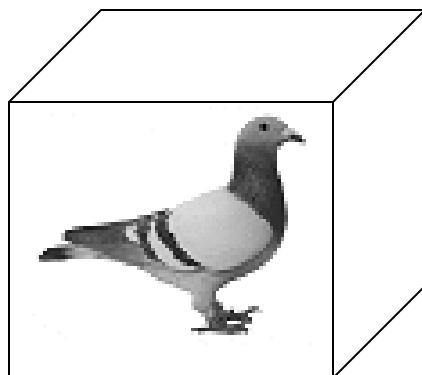
4 pigeons



3 pigeonholes



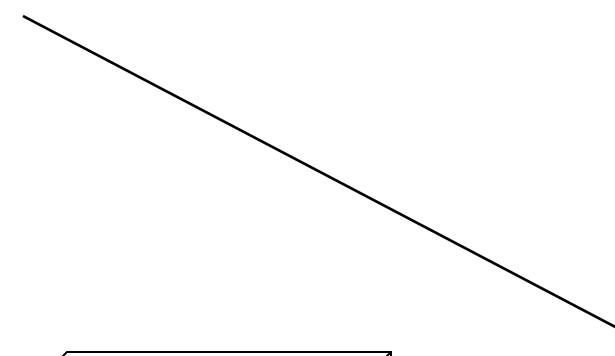
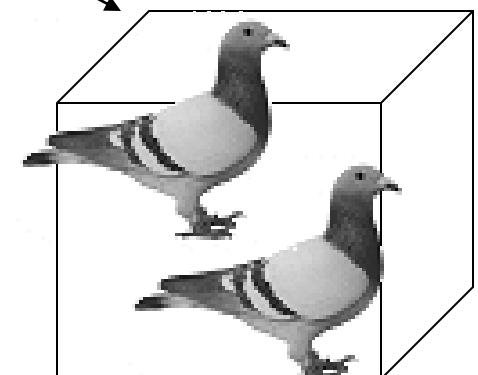
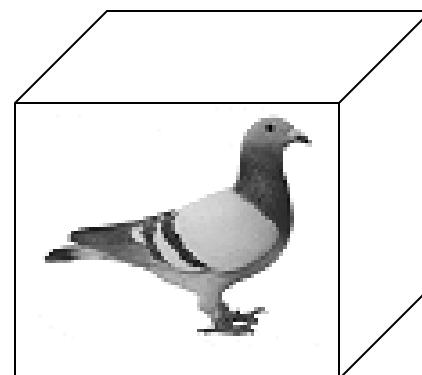
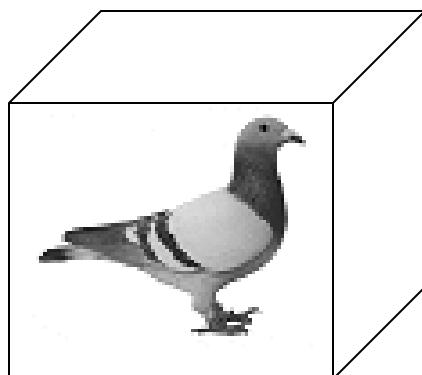
A pigeonhole must contain at least two pigeons



# The Pigeonhole Principle

If we put  $n$  objects into  $m$  boxes and if  $n > m$

- then at least one box must have more than one object in it.



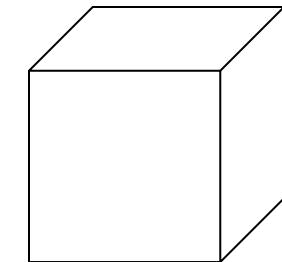
In other words for a string  $w$ :



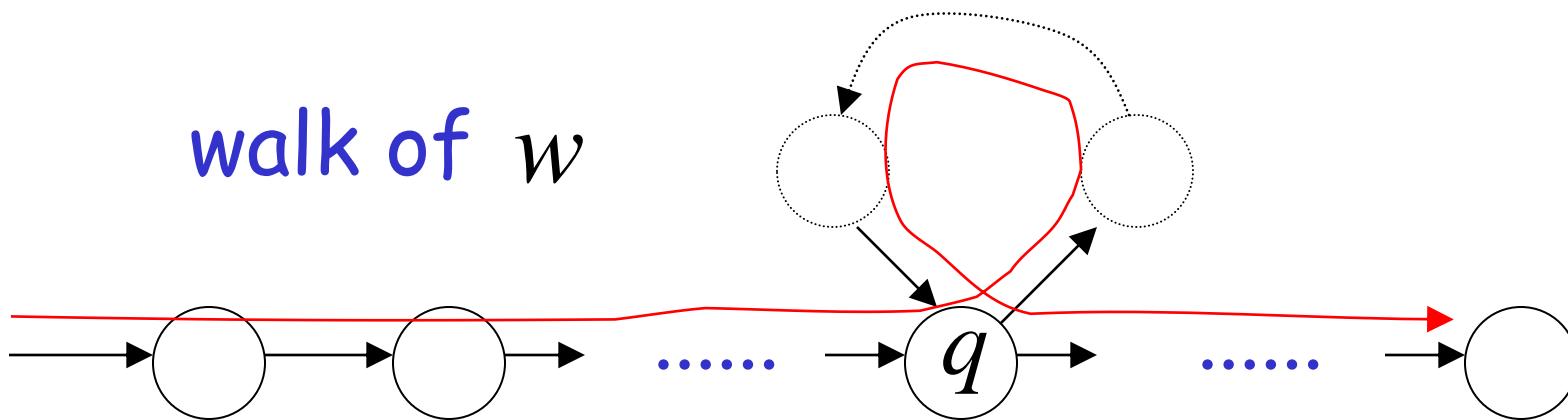
transitions are pigeons



states are pigeonholes



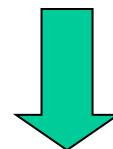
walk of  $w$



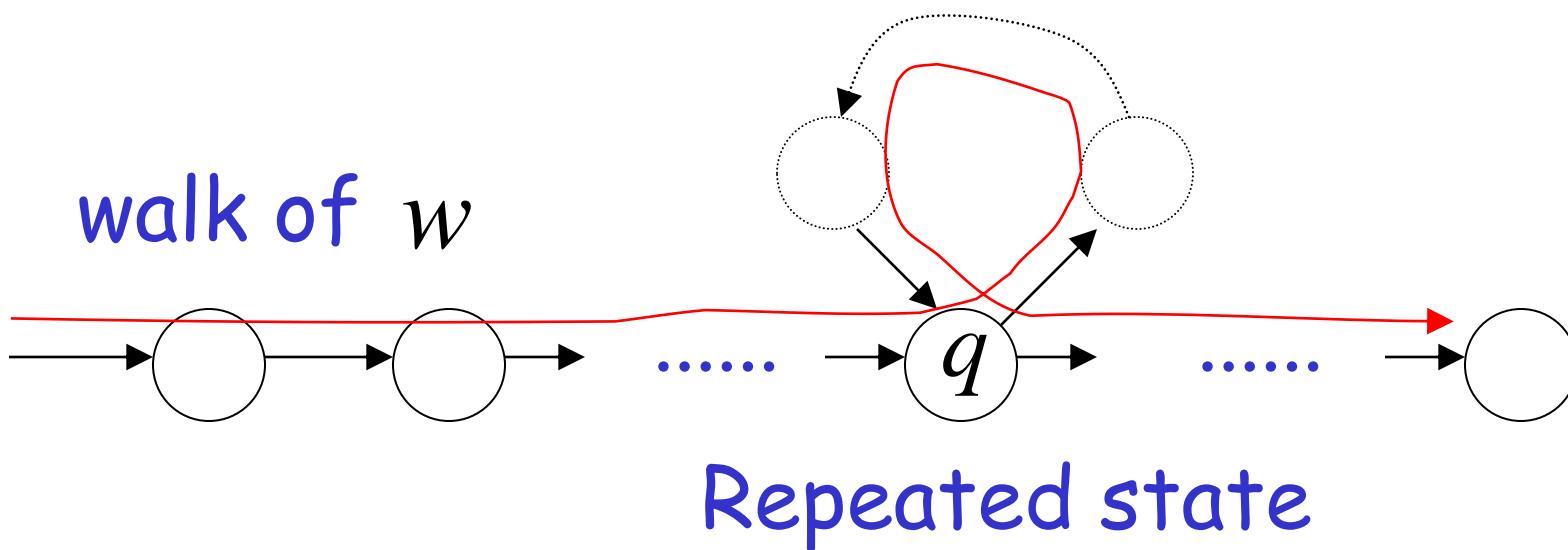
Repeated state

In general, for any DFA:

String  $w$  has length  $\geq$  number of states



A state  $q$  must be repeated in the walk of  $w$



# The Pigeonhole Principle and DFAs

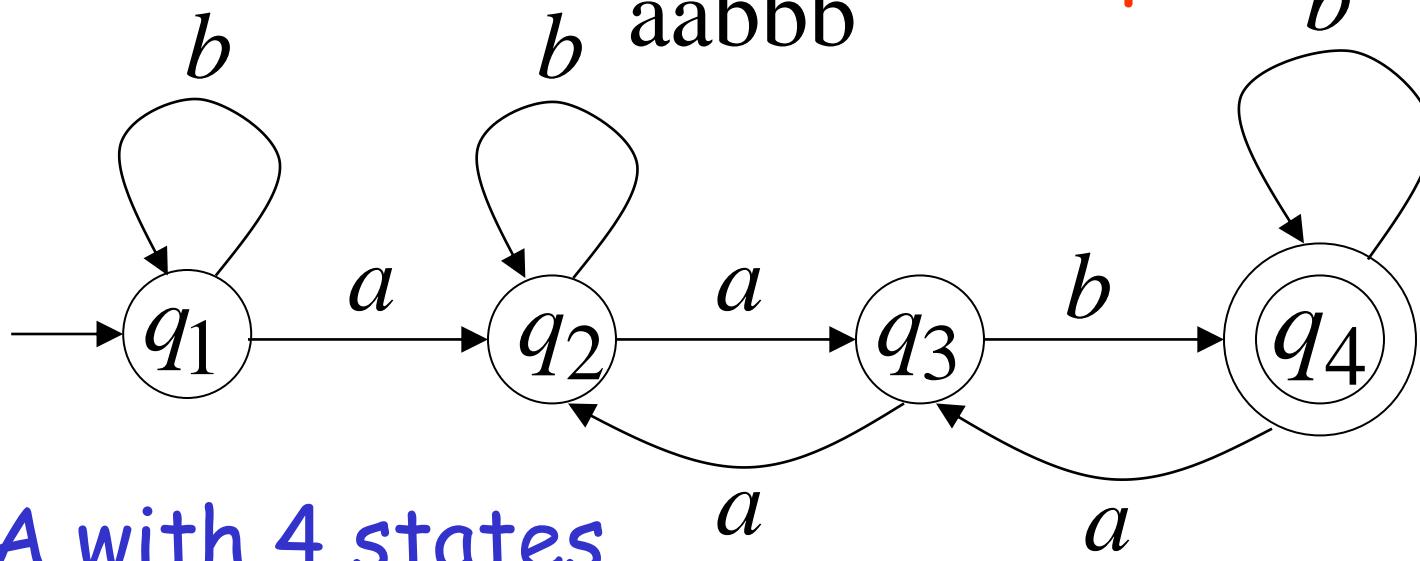
In walks of strings: aab

abab

aabb

aabb

a state  
is repeated



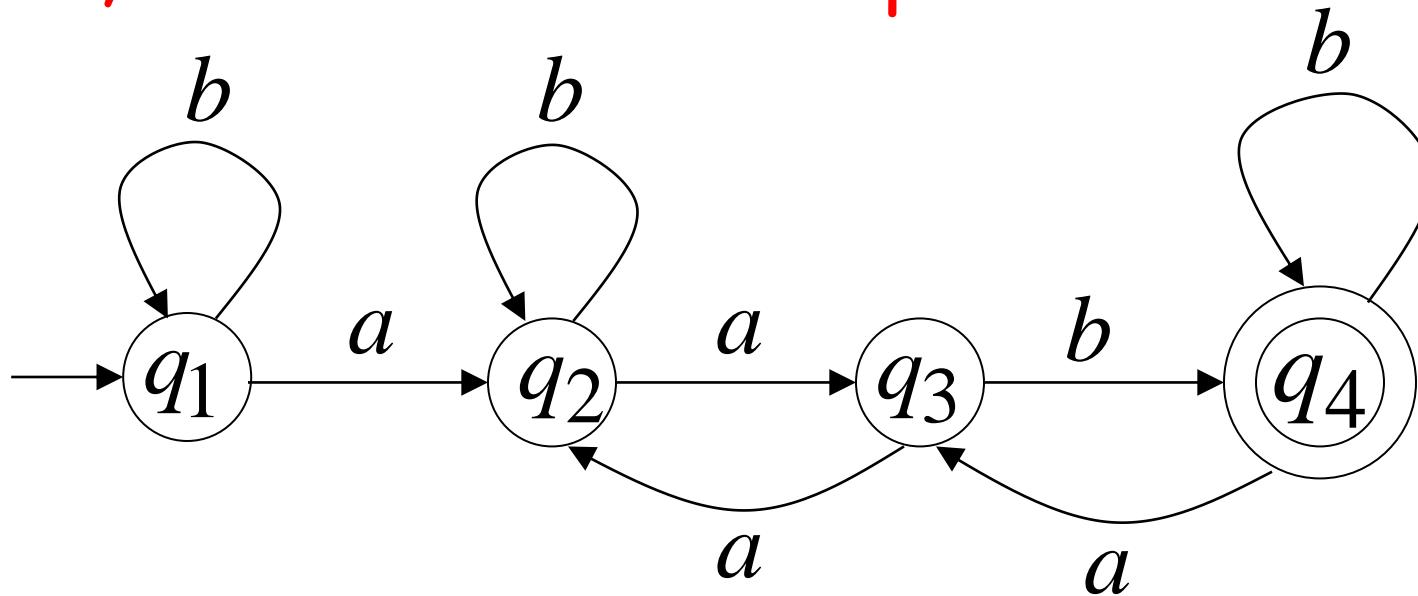
In string length  $< 4$

no state is repeated

If string  $w$  has length  $|w| \geq 4$ :

Then the transitions of string  $w$   
are more than the states of the DFA

Thus, a state must be repeated



# The Pumping Lemma:

- Assume that given a **infinite**  $L$  language is **regular**
- There exists an integer  $m$  for any string

$w \in L$  with length  $|w| \geq m$

Then, we can decompose  $w = x y z$

with  $|x y| \leq m$  and  $|y| \geq 1$

such that:  $x y^i z \in L \quad i = 0, 1, 2, \dots$

# Applications of the Pumping Lemma

Prove that the language  $L = \{a^n b^n : n \geq 0\}$   
is not regular

**Proof:** Since  $L$  is infinite.  
we can apply the Pumping Lemma

Assume for contradiction that  $L$  is a regular language

Let  $m$  be the integer in the Pumping Lemma

Pick a string  $w$  such that:  $w \in L$

length  $|w| \geq m$

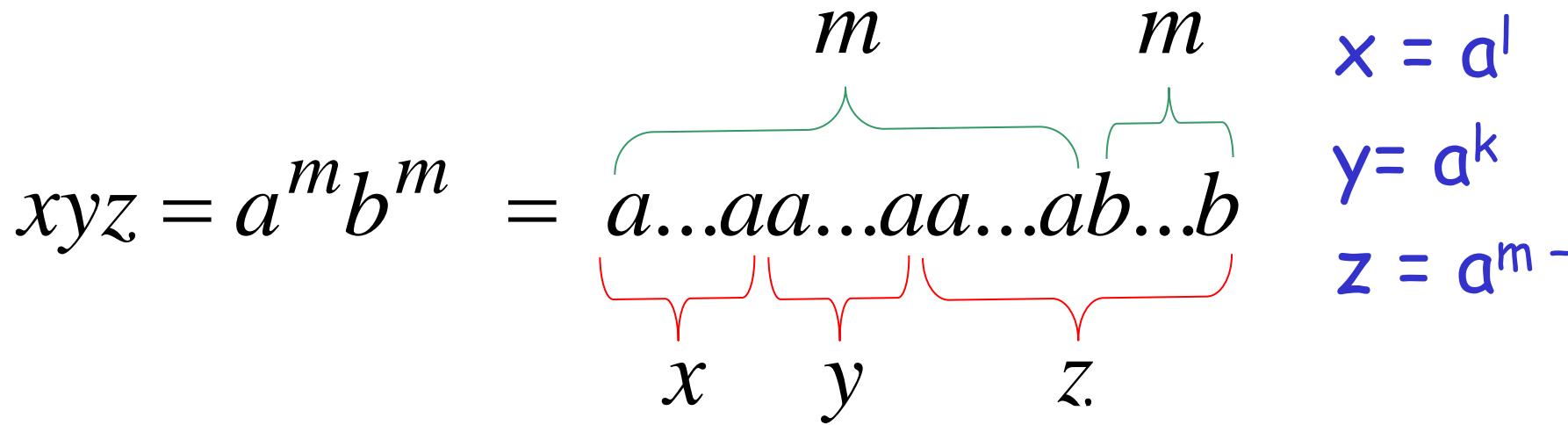
We pick  $w = a^m b^m$

Write:  $a^m b^m = x \ y \ z$

From the Pumping Lemma

it must be given that length

$$|x \ y| \leq m, \ |y| \geq 1$$



Thus:  $y = a^k, \ k \geq 1$

$$x \ y \ z = a^m b^m \quad y = a^k, \quad k \geq 1$$

From the Pumping Lemma:  $x \ y^i \ z \in L$   
 $i = 0, 1, 2, \dots$

Thus:  $x \ y^2 \ z \in L$

$xy^2z = \underbrace{a \dots aa \dots aa \dots aa}_{m+k} \dots ab \dots b \in L$

Diagram illustrating the decomposition of  $xy^2z$  into  $x, y, y, z$ :  
The string is divided into four segments by red brackets below:  
- Segment  $x$  covers the first  $m$  'a's.  
- Segment  $y$  covers the first 'a'.  
- Segment  $y$  covers the second 'a'.  
- Segment  $z$  covers the remaining part of the string, starting with 'b'.  
A green bracket above the string groups the first  $m+k$  'a's, and another green bracket groups the last  $m$  'a's.

Therefore:  $a^{m+k} b^m \in L$

$$a^{m+k}b^m \in L \quad k \geq 1$$

---

**BUT:**  $L = \{a^n b^n : n \geq 0\}$



$a^{m+k}b^m \notin L$  **CONTRADICTION!**

Therefore: Our assumption that  $L$  is a regular language is not true

**Conclusion:**  $L$  is not a regular language

Prove that the language  $L = \{vv^R : v \in \Sigma^*\}$   
is not regular  $\Sigma = \{a,b\}$

**Proof:** Since  $L$  is infinite  
we can apply the Pumping Lemma

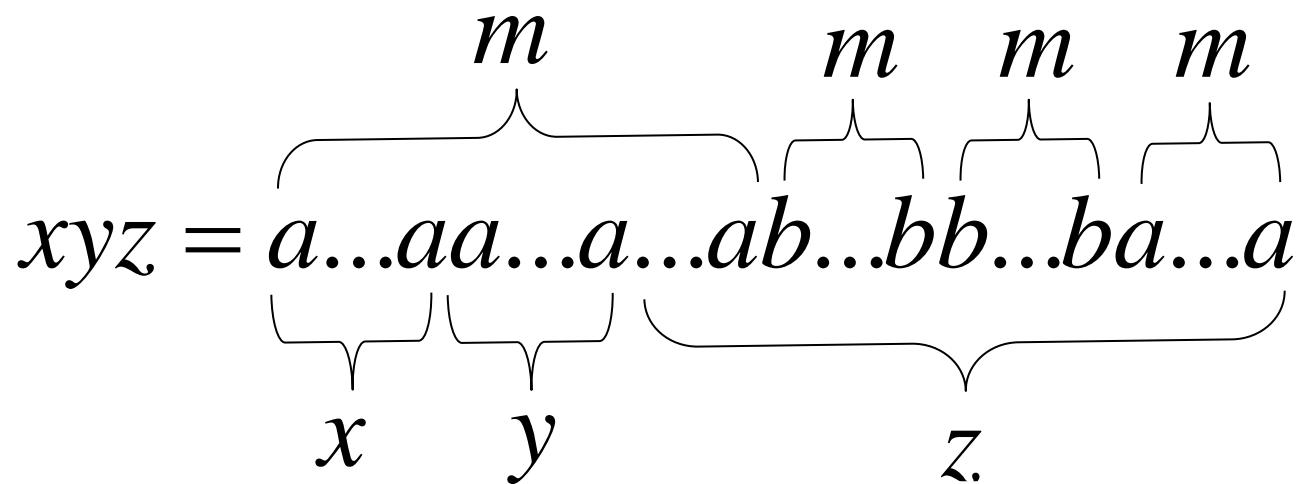
Assume for contradiction  
that  $L$  is a regular language

Let  $m$  be the integer in the Pumping Lemma  
Pick a string  $w$  such that:  $w \in L$  and  $|w| \geq m$   
We pick  $w = a^m b^m b^m a^m$

Write  $a^m b^m b^m a^m = x y z$

From the Pumping Lemma

it must be that length  $|x y| \leq m$ ,  $|y| \geq 1$



Thus:  $y = a^k$ ,  $k \geq 1$

$$x \ y \ z = a^m b^m b^m a^m \quad y = a^k, \quad k \geq 1$$

From the Pumping Lemma:  $x \ y^i \ z \in L$

$$i = 0, 1, 2, \dots$$

Thus:  $x \ y^2 \ z \in L$

$$xy^2z = \overbrace{a \dots aa \dots aa \dots a \dots ab \dots bb \dots ba \dots a}^{m+k} \in L$$

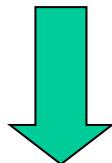
$m+k$   
 $m \quad m \quad m$   
 $x \quad y \quad y \quad z.$

Thus:  $a^{m+k} b^m b^m a^m \in L$

$$a^{m+k} b^m b^m a^m \in L \quad k \geq 1$$

---

**BUT:**  $L = \{vv^R : v \in \Sigma^*\}$



$a^{m+k} b^m b^m a^m \notin L$  **CONTRADICTION!**

Therefore: Our assumption that  $L$  is a regular language is not true

**Conclusion:**  $L$  is not a regular language

# Exercise

Prove that the following languages are not regular.

- $L = \{a^n b^l a^k : k \geq n+l\}$
- $L = \{a^n b^n : n \geq 0\} \cup \{a^n b^{n+1} : n \geq 0\} \cup \{a^n b^{n+1} : n \geq 0\}$
- $L = \{a^{n!} : n \geq 1\}$

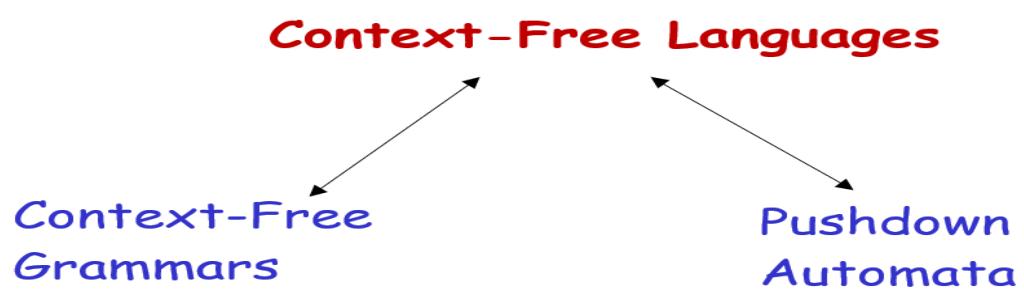
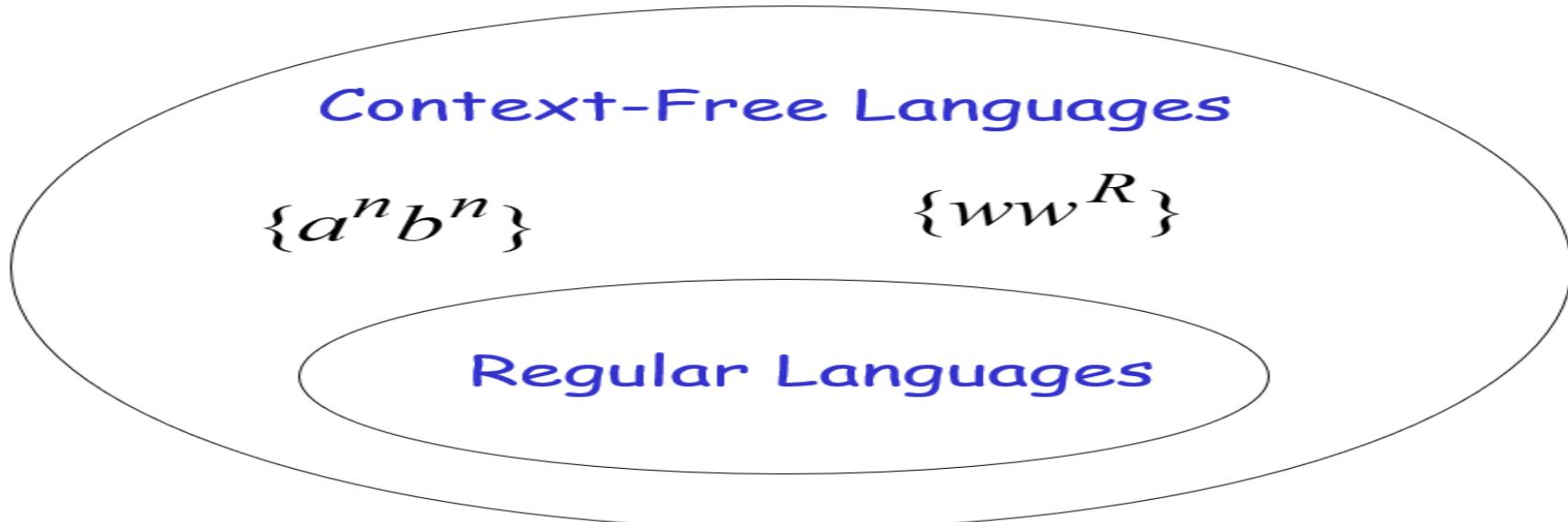
Are the following languages are regular ?

- $L = \{w \in \{a, b, c\}^*: |w| = 3n_a(w)\}$
- $L = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^*, w_1 \neq w_2\}$

# Non-regular Languages

- Construct NFA for real numbers (Compiler by Aho & Ullman).
- Construct NFA for identifiers (Compiler by Aho & Ullman).
- Construct NFA for comparison operators (Compiler by Aho & Ullman).
- Construct NFA for valid arithmetic expression with parenthesis (Is it possible ?). **NO**
  - Example:  $(a + (b - c * (d + e * f - g)) + h * i)$
  - $L = \{(n)^n : n \geq 0\}$
- Regular languages are effective in describing certain simple patterns.
- Example-  $L = \{(n)^n : n \geq 0\}$  is not regular.

# Context-Free Languages



# Context-Free Grammar (CFG)

A CFG G is defined as quadruple

$$G = (V, T, S, P)$$

V --- Finite and non-empty set of variables (non-terminals)

T --- Finite and non-empty set of terminal symbols

S --- Start variable,  $S \in V$

P --- Finite set of Production rules

All productions in P have the form  $A \rightarrow x$

$$A \in V$$

$$x \in (V \cup T)^*$$

Note:

- V and T are disjoint sets

Imposing a restriction  
on left-hand side but  
permitting anything on  
right side

# Definition: Context-Free Grammars

Grammar  $G = (V, T, S, P)$

Set of Variables

Set of Terminal  
symbols

Start  
variable

Set of Productions

The form of production

$$A \rightarrow x$$

Variable

String of variables  
and terminals

# Example-1

A context-free grammar  $G$  :

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow \lambda \end{aligned}$$

A derivation:  $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

Another derivation:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

## Example-2

A context-free grammar  $G$ :

$$S \rightarrow aSa$$

$$S \rightarrow bSb$$

$$S \rightarrow \lambda$$

A derivation:  $S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$

Another derivation:

$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$

$$L(G) = \{ww^R : w \in \{a,b\}^*\}$$

## Example-3

A context-free grammar  $G$ :  $S \rightarrow aSb$

$$S \rightarrow SS$$

$$S \rightarrow \lambda$$

A derivation:

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow ab$$

A derivation:

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$$

$$S \rightarrow aSb$$

$$S \rightarrow SS$$

$$S \rightarrow \lambda$$

$$\begin{aligned} L(G) = \{ w & : n_a(w) = n_b(w), \\ & \text{and } n_a(v) \geq n_b(v) \\ & \text{in any prefix } v \} \end{aligned}$$

Describes  
matched  
parentheses:

( ) (( )) (( ))

# Applications of CFG

- It is useful for nested structures, or parenthesized expressions in programming languages.
- It is used for parsing.
- It has many applications in Natural Language processing.

# Applications: Natural Languages CFG Representation

**SENTENCE** → NOUN-PHRASE VERB-PHRASE

**NOUN-PHRASE** → CMPLX-NOUN

**NOUN-PHRASE** → CMPLX-NOUN PREP-PHRASE

**VERB-PHRASE** → CMPLX-VERB

**VERB-PHRASE** → CMPLX-VERB PREP-PHRASE

**PREP-PHRASE** → PREP CMPLX-NOUN

**CMPLX-NOUN** → ARTICLE NOUN

**CMPLX-VERB** → VERB NOUN-PHRASE

**CMPLX-VERB** → VERB

**ARTICLE** → a

**ARTICLE** → the

**NOUN** → boy

**NOUN** → girl

**NOUN** → knife

**VERB** → kills

**VERB** → touches

**VERB** → sees

**PREP** → with

**Variables:** SENTENCE, NOUN-PHRASE, ...

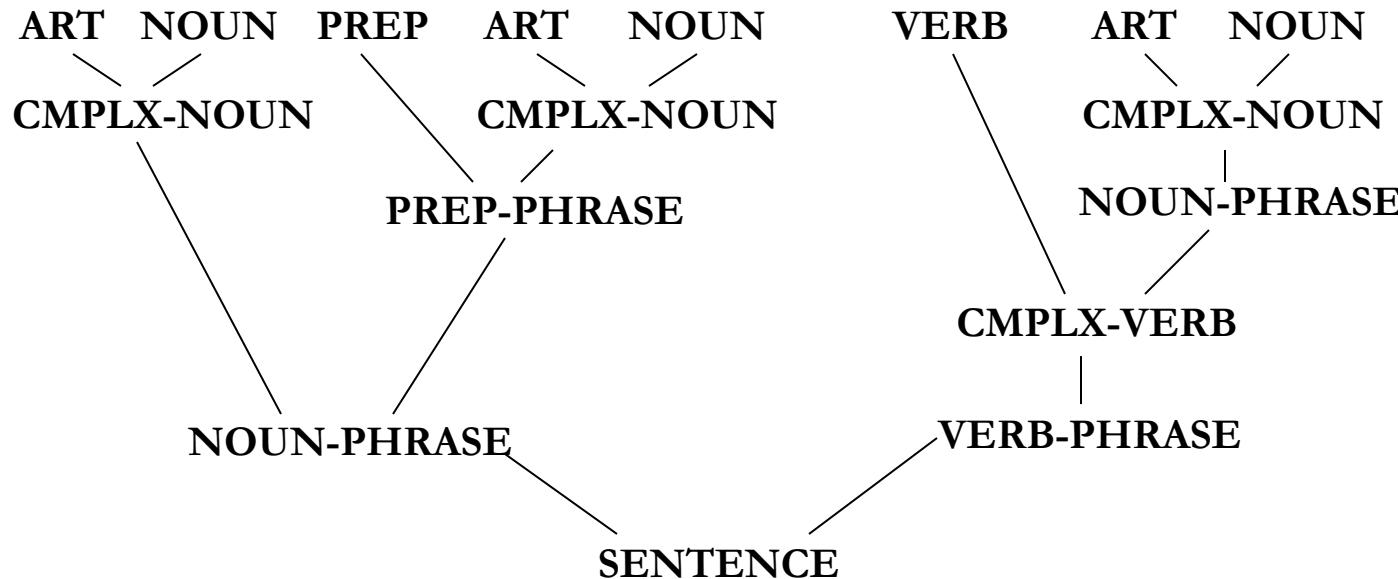
**Terminals:** a, the, boy, girl, knife, kills, touches, sees, with

**Start Variable:** SENTENCE

# Applications: Natural Languages

Context-free grammar were first used for natural languages.

a girl with a knife kills the boy



**SENTENCE** → **NOUN-PHRASE** **VERB-PHRASE**

**NOUN-PHRASE** → **CMPLX-NOUN**

**NOUN-PHRASE** → **CMPLX-NOUN** **PREP-PHRASE**

**VERB-PHRASE** → **CMPLX-VERB**

**VERB-PHRASE** → **CMPLX-VERB** **PREP-PHRASE**

**PREP-PHRASE** → **PREP** **CMPLX-NOUN**

**CMPLX-NOUN** → **ARTICLE** **NOUN**

**CMPLX-VERB** → **VERB** **NOUN-PHRASE**

**CMPLX-VERB** → **VERB**

**ARTICLE** → **a**

**ARTICLE** → **the**

**NOUN** → **boy**

**NOUN** → **girl**

**NOUN** → **knife**

**VERB** → **kills**

**VERB** → **touches**

**VERB** → **sees**

**PREP** → **with**

# Definition: Context-Free Languages

A language  $L$  is context-free

if and only if there is a context-free grammar

$G = (V, T, S, P)$  with  $L = L(G)$

\*

$L(G) = \{ w : S \Rightarrow w, w \in T^* \}$

# Regular Grammar v/s CFG

A **regular grammar** is one that is either right-linear or left-linear grammar.

Every regular grammar is **context-free grammar**.

Every context-free grammar **is not** regular.

It means - The family of regular languages is a proper subset of the family of context-free languages.

# Linear Grammar v/s CFG

# FIND Language

$$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSa \mid bSb \mid \epsilon\})$$

For  $aabbba$ , typical derivation might be:

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbba$$

$$S \Rightarrow bSb \Rightarrow bbSbb \Rightarrow bbaSabb \Rightarrow bbaabb$$

Grammar generates language

$$L(G) = \{ww^R : w \in \{a, b\}^*\}$$

# Context-Free Language: Example

What is the language of this grammar?

$$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow ab\})$$

$$L(G) = \{a^n b^n \mid n \geq 1\}$$

What is the language of this grammar?

$$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb, S \rightarrow \epsilon\})$$

$$L(G) = \{a^n b^n \mid n \geq 0\}$$

# Context-Free Language: Example

What is the language of this grammar?

$$G = (\{S, A\}, \{a, b\}, S, \{S \rightarrow aSa \mid aAa, \quad A \rightarrow bA \mid b\})$$

$$\begin{aligned} S \rightarrow aSa &\rightarrow aaSaa \rightarrow aaaSaaa \rightarrow aaaaAaaaa \rightarrow aaaabAaaaa \rightarrow \\ &aaaabbAaaaa \rightarrow aaaabbbaaaa \end{aligned}$$

$$L(G) = \{a^n b^m a^n \mid n \geq 1, m \geq 1\}$$

What is the language of this grammar?

$$\begin{aligned} G_1 = (\{S, A, B\}, \{a, b\}, S, \{S \rightarrow AB \\ A \rightarrow aA \mid a \\ B \rightarrow bB \mid \varepsilon\}) \end{aligned}$$

$$S \rightarrow AB \rightarrow aAB \rightarrow aaB \rightarrow aa$$

$$\begin{aligned} G_2 = (\{S, B\}, \{a, b\}, S, \{S \rightarrow aS \mid aB \\ B \rightarrow bB \mid \varepsilon\}) \end{aligned}$$

$$L(G) = \{a^+ b^*\} = \{a^n b^m \mid n \geq 1, m \geq 0\}$$

# Context-Free Language: Example

What is the language of this grammar?

$$G = (\{S, B\}, \{a, b, c\}, S, \{S \rightarrow abScB \mid \epsilon, B \rightarrow bB \mid b\})$$

$$L = \{(ab)^n(cb^m)^n \mid n \geq 0, m \geq 1\}$$

What is the grammar of this language?

$$L = \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

$$G = (\{S, A\}, \{a, b, c, d\}, S, \{S \rightarrow aSd \mid aAd, A \rightarrow bAc \mid bc\})$$

What is the grammar of this language?

$$L = \{a^n b^m c^m d^{2n} \mid n \geq 0, m \geq 1\}$$

$$G = (\{S, A\}, \{a, b, c, d\}, S, \{S \rightarrow aSdd \mid A, A \rightarrow bAc \mid bc\})$$

# Context-Free Language: Example

What is the grammar of this language?

$$L = \{a^*ba^*ba^*\}$$

$$G_1 = (\{S, A\}, \{a, b\}, S, \{S \rightarrow AbAbA$$

$$A \rightarrow aA \mid \varepsilon\})$$

$$G_2 = (\{S, A, C\}, \{a, b\}, S, \{S \rightarrow aS \mid bA$$

$$A \rightarrow aA \mid bC\})$$

$$C \rightarrow aC \mid \varepsilon$$

# Context-Free Language: Example

What is the grammar of this language?

A language over  $\{a, b\}$  with at least 2 b's

$$G_1 = (\{S, A\}, \{a, b\}, S, \{S \rightarrow AbAbA \\ A \rightarrow aA \mid bA \mid \varepsilon\})$$

$$G_2 = (\{S, A, C\}, \{a, b\}, S, \{S \rightarrow aS \mid bA \\ A \rightarrow aA \mid bC \\ C \rightarrow aC \mid bC \mid \varepsilon\})$$

# Context-Free Grammar: Example

Write a CFG for the following Languages

$$L_1 = \{wcw^R \mid w \in \{a, b\}^*\}$$

$$m=1$$

$$S \rightarrow aSa \mid bSb \mid c$$

$$n = 0, 1, 2, 3, 4$$

$$L_2 = \{a^n b^m \mid n \leq m+3, m, n \geq 0\}$$

$$S \rightarrow e \mid aA \mid aaA \mid aaaA$$

$$S \rightarrow aaaA \mid aaA \mid aA \mid A \mid \lambda$$

$$A \rightarrow aAb \mid B$$

$$A \rightarrow aAb \mid B$$

$$B \rightarrow Bb \mid \lambda$$

$$B \rightarrow e \mid Bb$$

$$L_3 = \{a^n b^m \mid n \geq m+3, m, n \geq 0\}$$

$$S \rightarrow aA \rightarrow aaaaaAbbb$$

$$S \rightarrow aaaA$$

$$A \rightarrow aAb \mid aA \mid \lambda$$

$$aaaaaaBbbbbbb \rightarrow a^5Bb^9$$

$$n = 5, m = 10$$

# Context-Free Grammar: Example

The following languages are NOT CFLs

$$L_1 = \{ww \mid w \in \{a, b\}^*\}$$

$$L_2 = \{a^n b^m c^n d^m \mid n \geq 1, m \geq 1\}$$

$$S \rightarrow XY$$

$$X \rightarrow aXb \mid ab$$

$$Y \rightarrow cYd \mid cd$$

$$L = \{a^n b^n c^m d^m\}$$

$$L_3 = \{a^n b^n c^n \mid n \geq 0\}$$

$$G = (\{S\}, \{a, b\}, S, \{S \rightarrow aSb \mid bSa \mid SS \mid \epsilon\})$$

is this grammar context-free?

yes, there is a single variable on the left hand side

# Write Context-Free Grammars

$$L = \{a^n b^{2n} : n \geq 0\}$$

$$L = \{a^n b^n . a^m b^m : n \geq 0, m \geq 0\}$$

$$L = \{a^n b^m : n \neq m\}$$

$$L = \{a^n b^m : n \neq 2m\}$$

$$L = \{a^n b^m : 2n \leq m \leq 2m\}$$

$$L = \{a^n b^m : n \leq m+3, m, n \geq 0\}$$

$$L = \{a^n b^m c^k : k = m+n, m, n, k \geq 0\}$$

$$L = \{a^n b^m c^k : k = 2m+n, m, n, k \geq 0\}$$

$$L = \{a^n b^n c^k : k \geq 0\}$$

# CFG Derivation

It generates strings by repeated replacement of non-terminals with string of terminals and non-terminals.

Write down start symbol (non-terminal).

Replace a non-terminal with the right-hand-side of production rule.

Repeat above until no more non-terminals.

A sequence of substitutions to obtain a string is called derivation.

# Derivation: Example

A context-free grammar  $G$  :

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow SS \\ S &\rightarrow \lambda \end{aligned}$$

A derivation:

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow ab$$

Another derivation:

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$$

Another derivation:

$$S \xrightarrow{} SS \xrightarrow{} SaSb \xrightarrow{} Sab \xrightarrow{} aSbab \xrightarrow{} abab$$

# Derivation Order

$$1. \ S \rightarrow AB$$

$$2. \ A \rightarrow aaA$$

$$4. \ B \rightarrow Bb$$

$$3. \ A \rightarrow \lambda$$

$$5. \ B \rightarrow \lambda$$

Leftmost derivation:

$$\begin{array}{ccccc} 1 & & 2 & & 3 \\ S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB & \xrightarrow{4} & \xrightarrow{5} & \xrightarrow{4} & \end{array}$$

$aab$

Rightmost derivation:

$$\begin{array}{ccccc} 1 & & 4 & & 5 \\ S \Rightarrow AB \Rightarrow ABb \Rightarrow Ab & \xrightarrow{2} & \xrightarrow{3} & \xrightarrow{2} & \end{array}$$

$aab$

$$S \rightarrow aAB$$

$$A \rightarrow bBb$$

$$B \rightarrow A | \lambda$$

Leftmost derivation:

$$\begin{aligned} S &\Rightarrow aAB \Rightarrow abBbB \Rightarrow abAbB \Rightarrow abbBbbB \\ &\Rightarrow abbbbB \Rightarrow abbbb \end{aligned}$$

Rightmost derivation:

$$\begin{aligned} S &\Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abAb \\ &\Rightarrow abbBbb \Rightarrow abbbb \end{aligned}$$

# Leftmost, Rightmost Derivations

**Definition.** A derivation is said to be **left-most derivation** if in each step the **leftmost variable** in the sentential form is replaced.

**Definition.** A derivation is said to be **right-most derivation** if in each step the **rightmost variable** in the sentential form is replaced.

# DERIVATION Tree/Parse Tree

- Alternative way for derivation of strings.
- Independent of order

Beginning with the root (labeled with the start symbol) and ending in leaves that are terminals, a derivation tree shows how each variable is replaced in the derivation.

# DERIVATION Tree/Parse Tree

A derivation tree for a CFG  $G = (V, T, S, P)$  is an ordered tree with labels on the nodes, as follows:

- The root is labeled with  $S$ .
- Every leaf is labeled by a terminal or  $\lambda$ .
- Leaves labeled by  $\lambda$  have no siblings.
- Every internal node is labeled by a variable.
- If a node is labeled  $A$  and has children  $a_1, \dots, a_n$  from left to right, then  $P$  must contain production of the form.

$$A \rightarrow a_1 \dots a_n$$

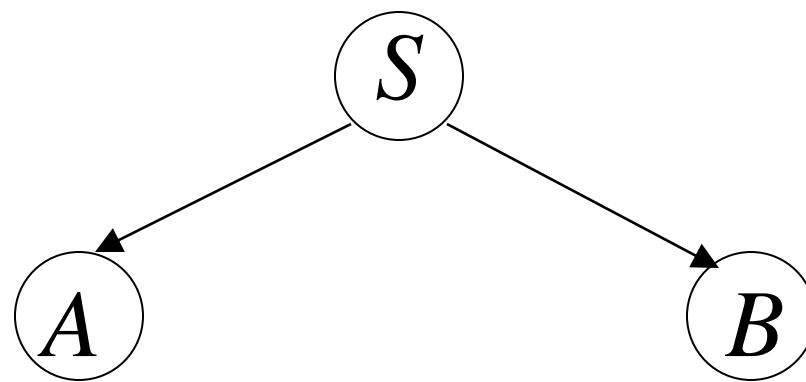
$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

w =aab

$$S \Rightarrow AB$$



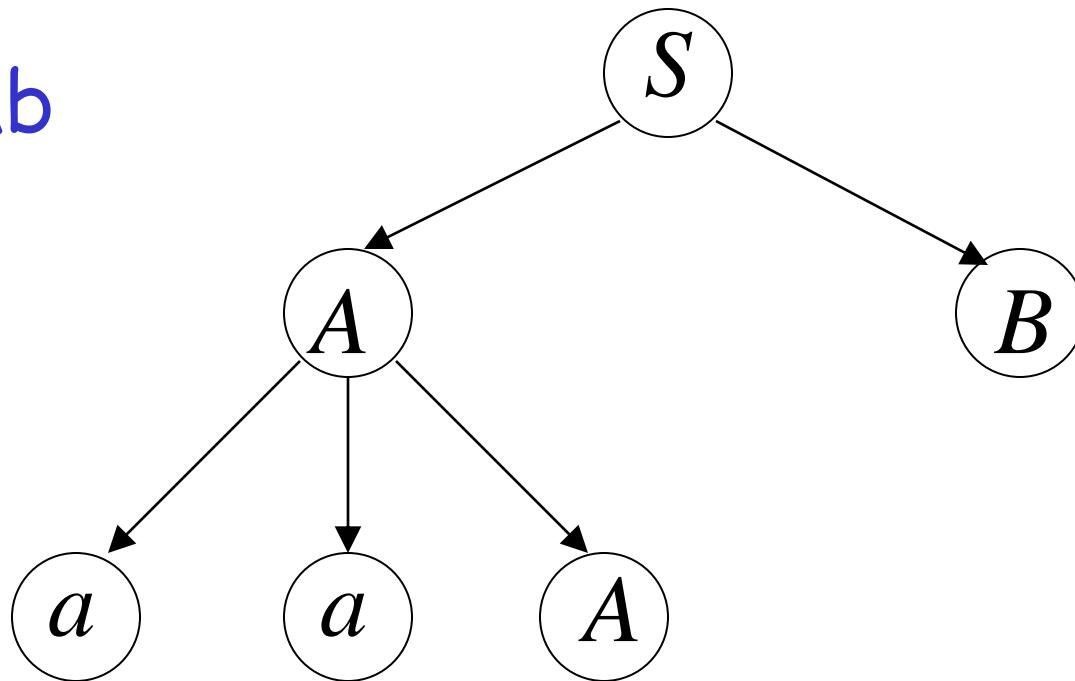
$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB$$

w =aab



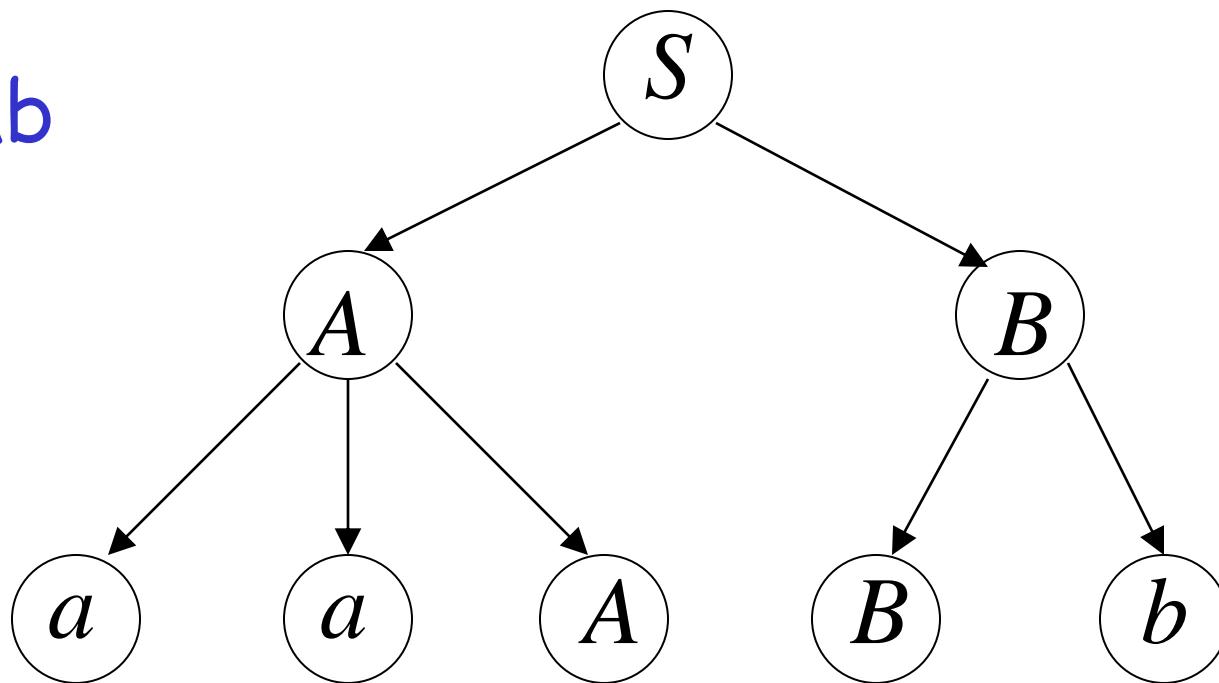
$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb$$

w =aab

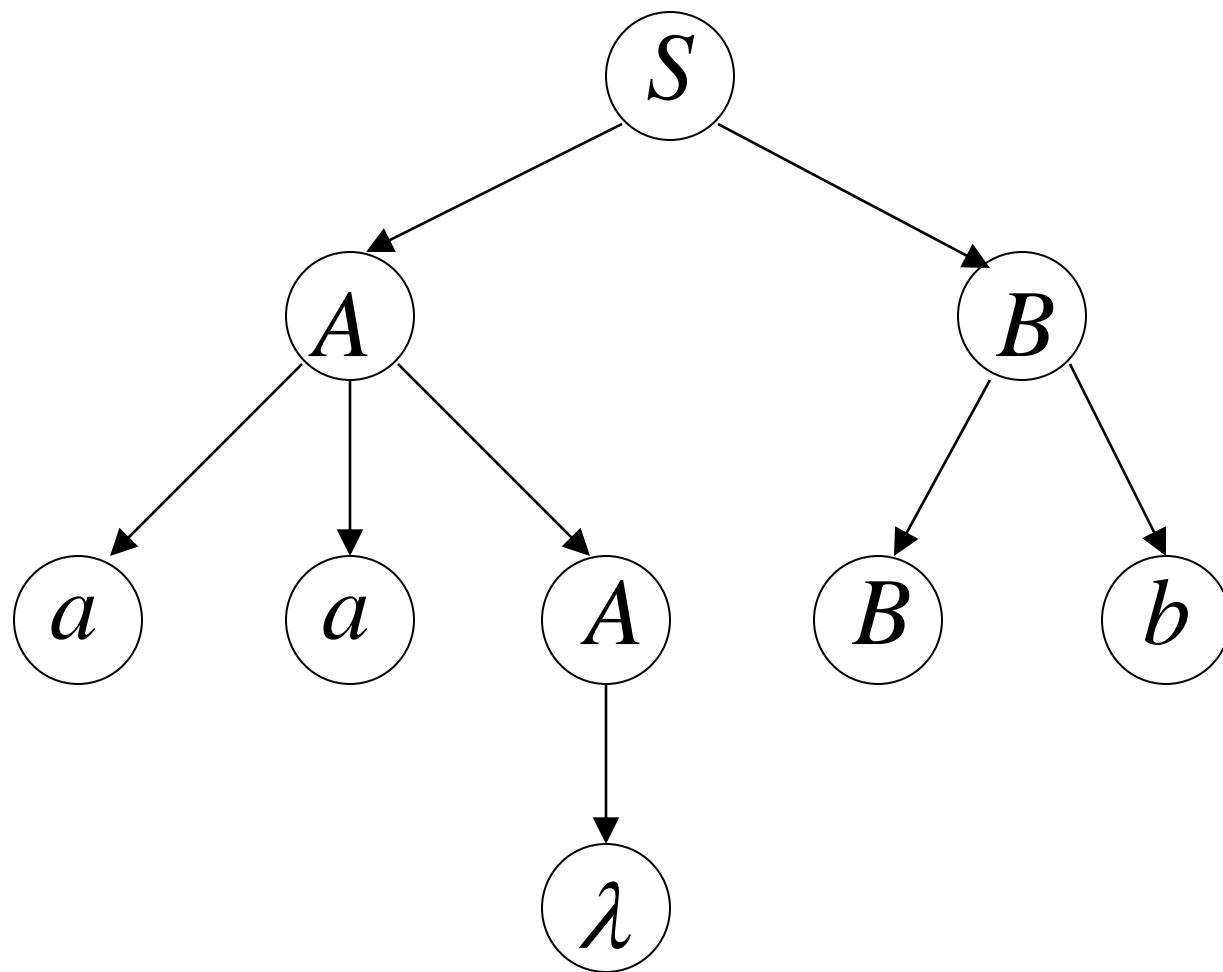


$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb$$



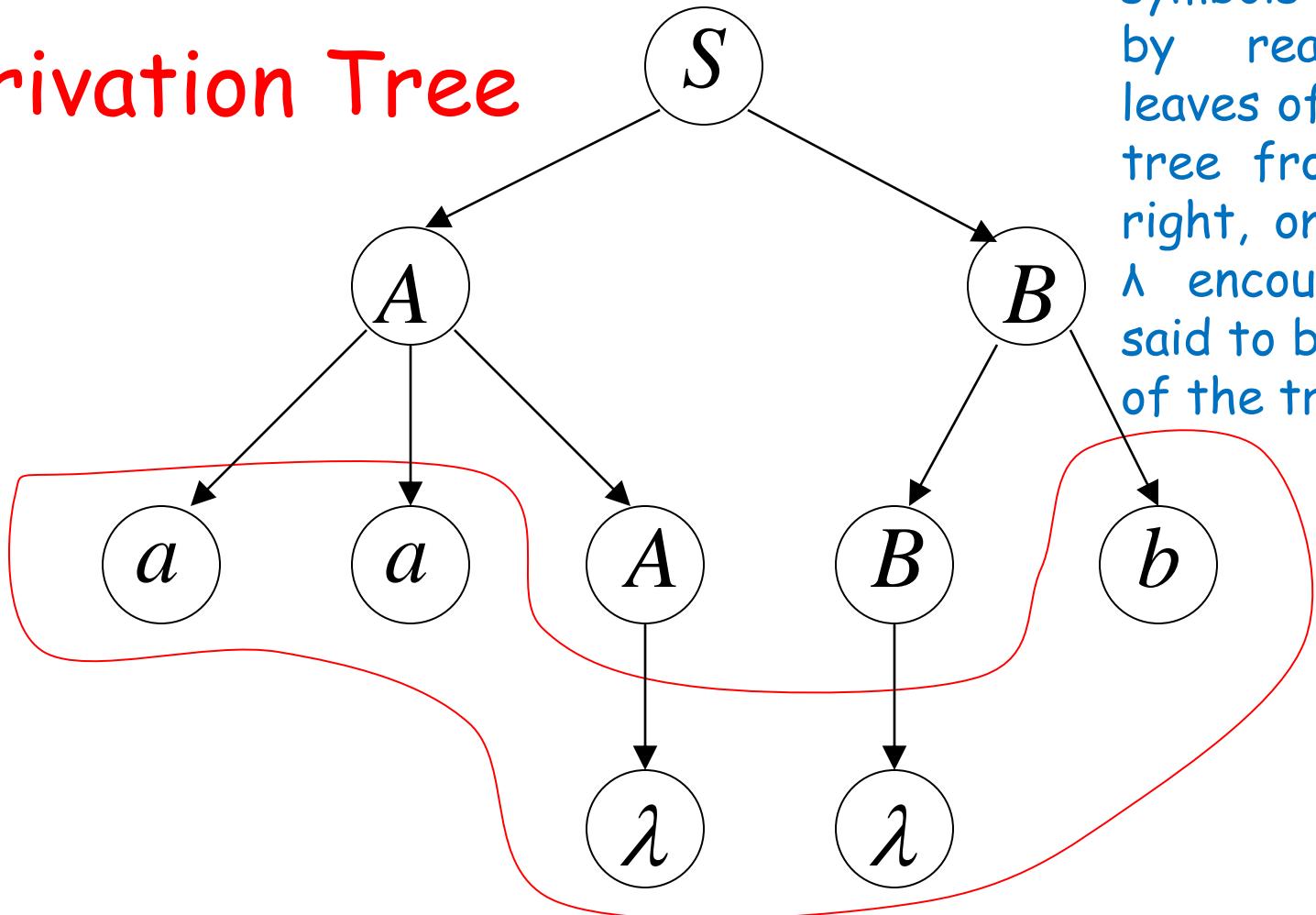
$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

## Derivation Tree



The string of the symbols obtained by reading the leaves of the parse tree from left to right, omitting any  $\lambda$  encountered, is said to be the yield of the tree.

yield

$aa\lambda\lambda b$

$= aab$

# Derivation tree exists for every string

Let  $G = (V, T, S, P)$  be a CFG.

For every  $w$  in  $L(G)$ , there exists a derivation tree of  $G$  whose yield is  $w$ .

Conversely,

The yield of any derivation tree of  $G$  is in  $L(G)$ .

Similarly, for every  $w$  in  $L(G)$ , there exists a leftmost/rightmost derivation.

# Parsing & Ambiguity

# Parsing

Given a grammar  $G = (V, T, S, P)$ .

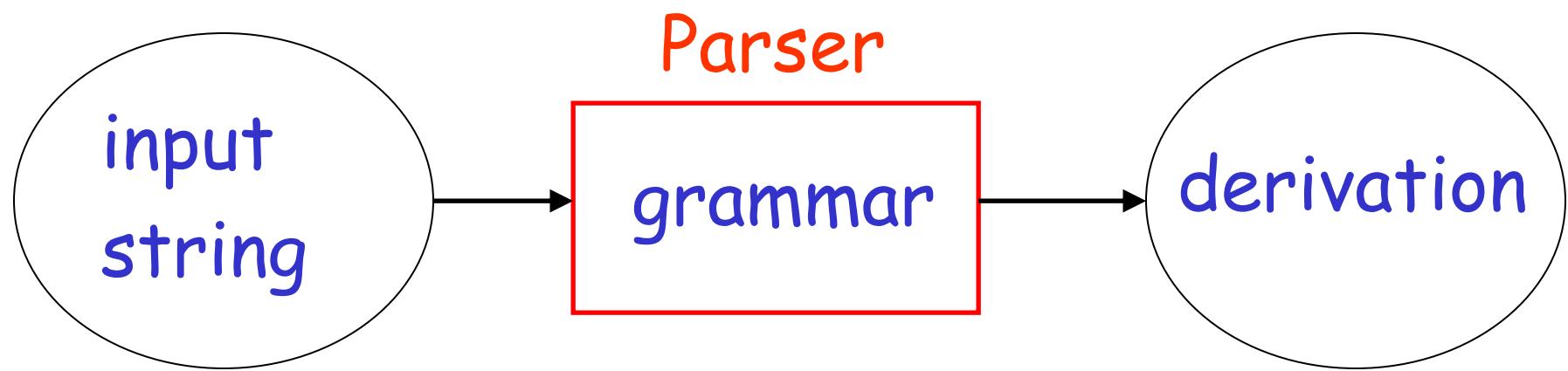
Given a string  $w$

We are interested to know whether or not  $w$  is in  $L(G)$ .

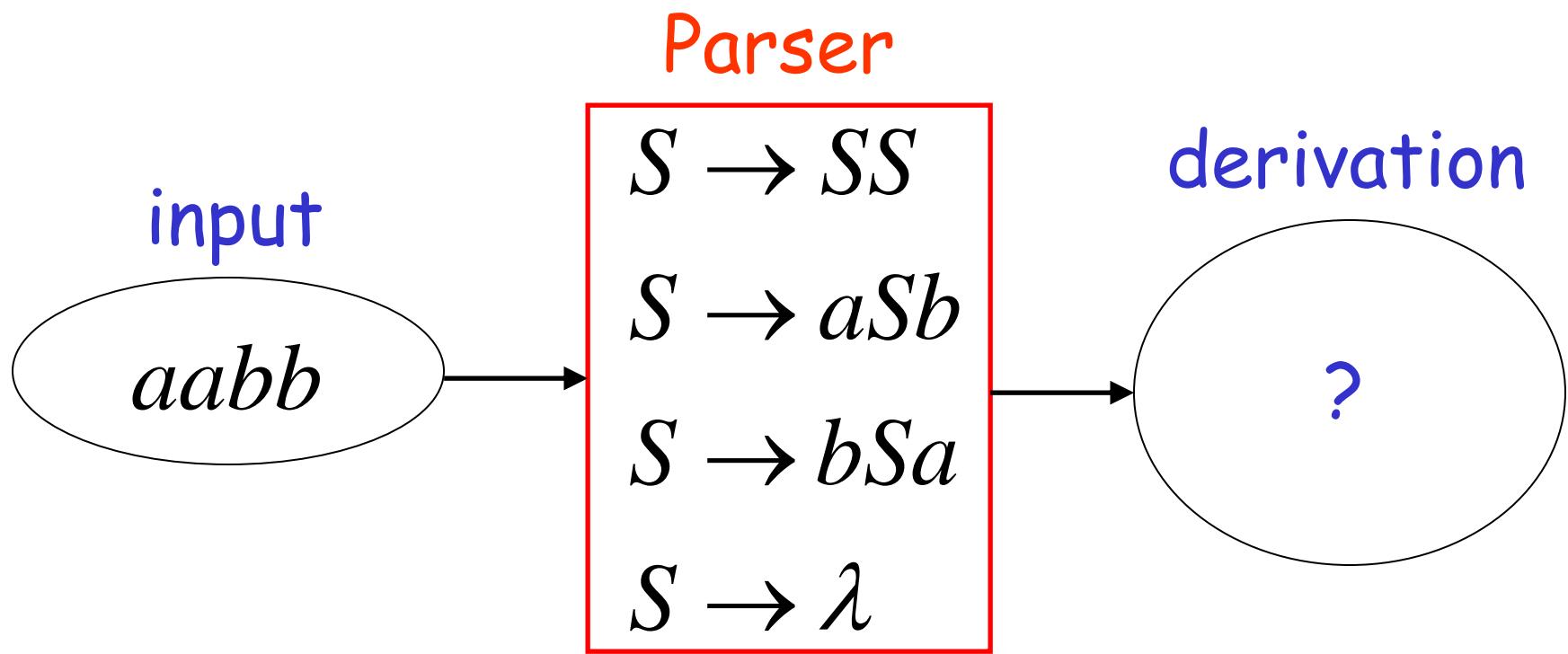
If so, we are interested to find a derivation.

An algorithm that can tell us whether  $w$  is in  $L(G)$  is a membership algorithm.

The term parsing describes finding a sequence of production by which  $w$  in  $L(G)$  is derived.



## Example:



# Exhaustive Search

$$S \rightarrow SS \mid aSb \mid bSa \mid \lambda$$

Phase 1:

$$S \Rightarrow SS$$

Find derivation of

$$S \Rightarrow aSb$$

$aabb$

~~$$S \Rightarrow bSa$$~~

~~$$S \Rightarrow \lambda$$~~

All possible derivations of length 1

Phase 2  $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$

$S \Rightarrow SS \Rightarrow SSS$

$S \Rightarrow SS \Rightarrow aSbS$

$aabb$

Phase 1

$S \Rightarrow SS$

$S \Rightarrow SS \Rightarrow S$

$S \Rightarrow aSb$

$S \Rightarrow aSb \Rightarrow aSSb$

$S \Rightarrow aSb \Rightarrow aaSbb$

$\cancel{S \Rightarrow aSb \Rightarrow abSab}$

$\cancel{S \Rightarrow aSb \Rightarrow ab}$

$$S \rightarrow SS \mid aSb \mid bSa \mid \lambda$$

Phase 2

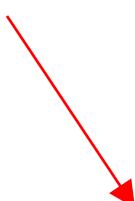
$$S \Rightarrow SS \Rightarrow SSS$$

$$S \Rightarrow SS \Rightarrow aSbS \qquad \qquad \qquad aabb$$

$$S \Rightarrow SS \Rightarrow S$$

$$S \Rightarrow aSb \Rightarrow aSSb$$

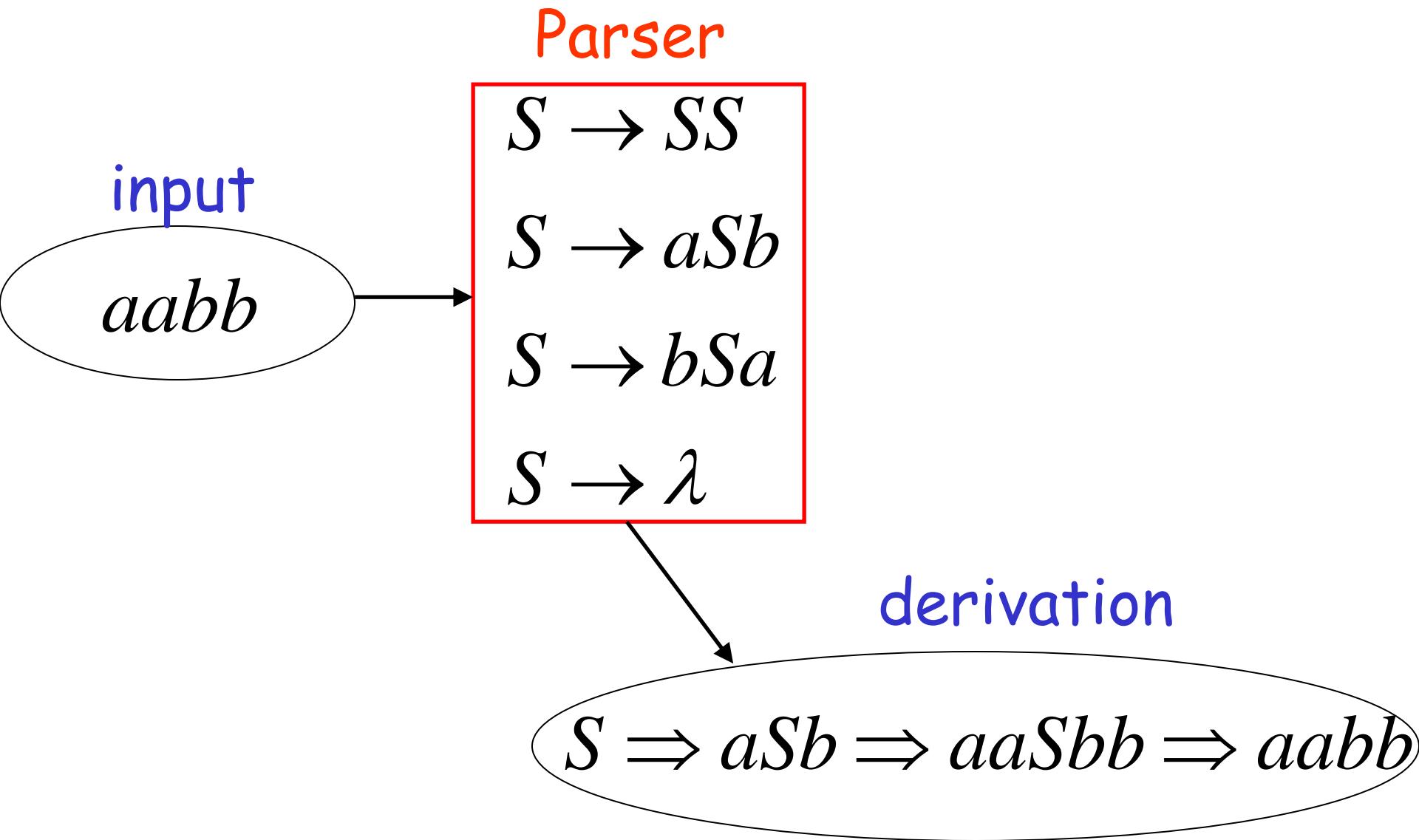
$$S \Rightarrow aSb \Rightarrow aaSbb$$



Phase 3

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

# Final result of exhaustive search (top-down parsing)

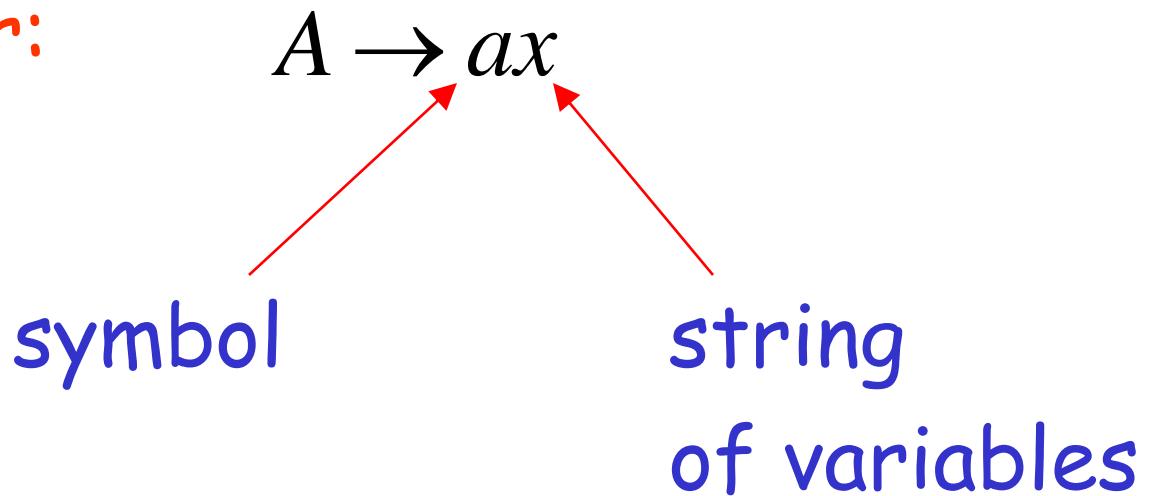


# Issues with Exhaustive search

- Tedious; Not efficient.
  - Generates too many sentential forms.
  - Depends on the grammars
- The search may be infinite;
  - $W = abb$   $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$
  - Null production  $S \longrightarrow \lambda$

There exist faster algorithms  
for specialized grammars

S-grammar:



Pair  $(A, a)$  appears once

## *S*-grammar example:

$$S \rightarrow aS$$

$$S \rightarrow bSS$$

$$S \rightarrow c$$

Each string has a unique derivation

$$S \rightarrow aS \rightarrow abSS \rightarrow abcS \rightarrow abcc$$

$$S \Rightarrow aS \Rightarrow abSS \Rightarrow abcS \Rightarrow abcc$$

For S-grammars:

In the exhaustive search parsing  
there is only one choice in each phase

Time for a phase: 1

Total time for parsing string  $w$ :  $|w|$

For general context-free grammars:

There exists a parsing algorithm  
that parses a string  $|w|$   
in time  $|w|^3$

# Ambiguity

CFG Issue: ambiguous grammar!

A string is derived ambiguously if it has at least two different derivation trees or leftmost/rightmost derivations.

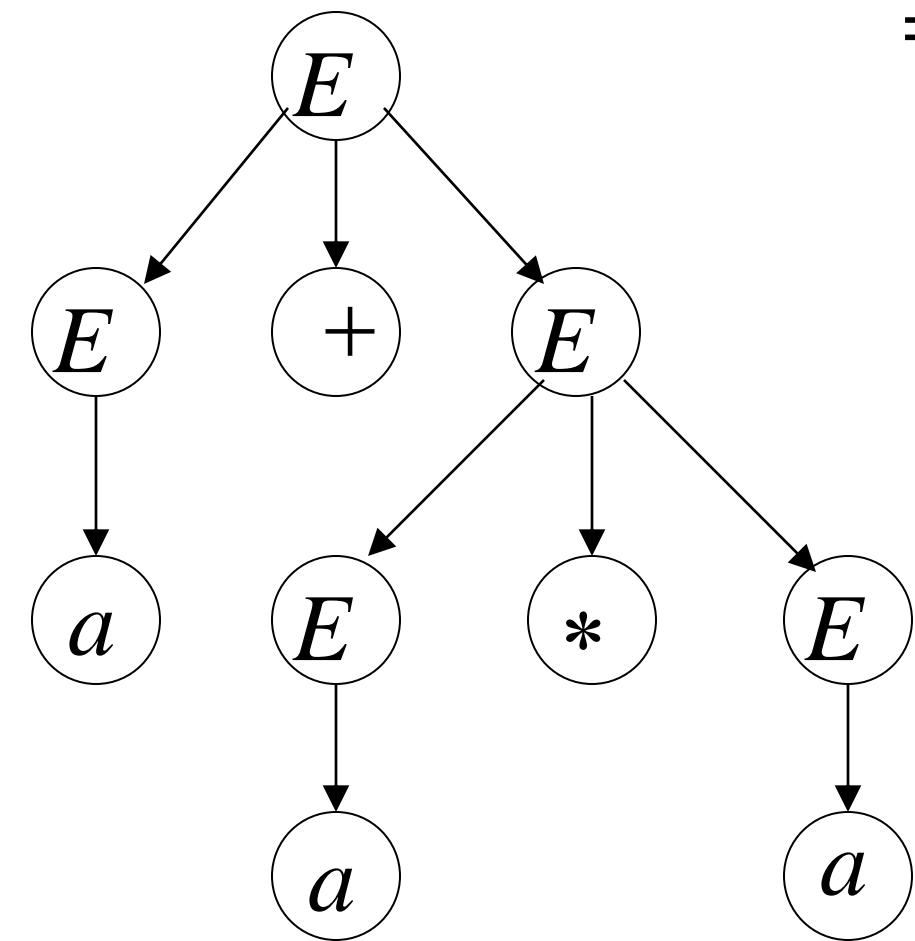
A context-free grammar is ambiguous if its language contains at least one ambiguously derived string.

Ambiguity is undesirable and some CFLS are inherently ambiguous.

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$a + a * a$

$$\begin{aligned} E &\Rightarrow \textcolor{red}{E + E} \Rightarrow \textcolor{green}{a + E} \Rightarrow a + \textcolor{blue}{E * E} \\ &\Rightarrow a + a * E \Rightarrow a + a * a \end{aligned}$$



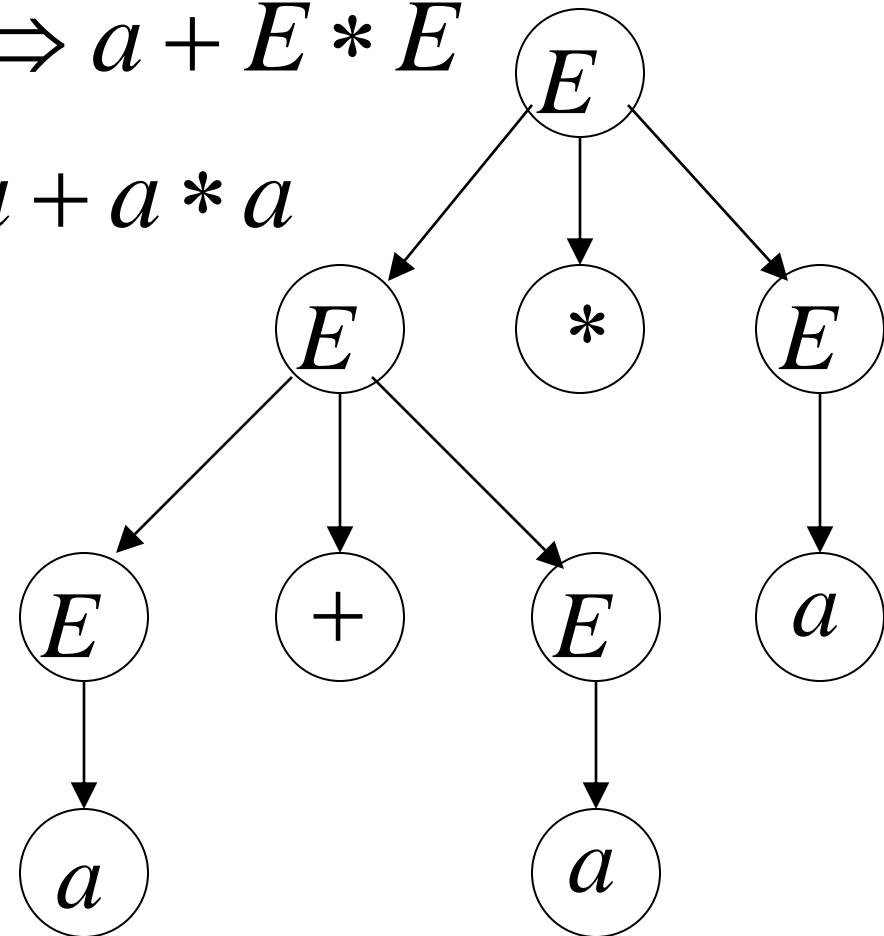
leftmost derivation  
&  
Derivation Tree

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$a + a * a$

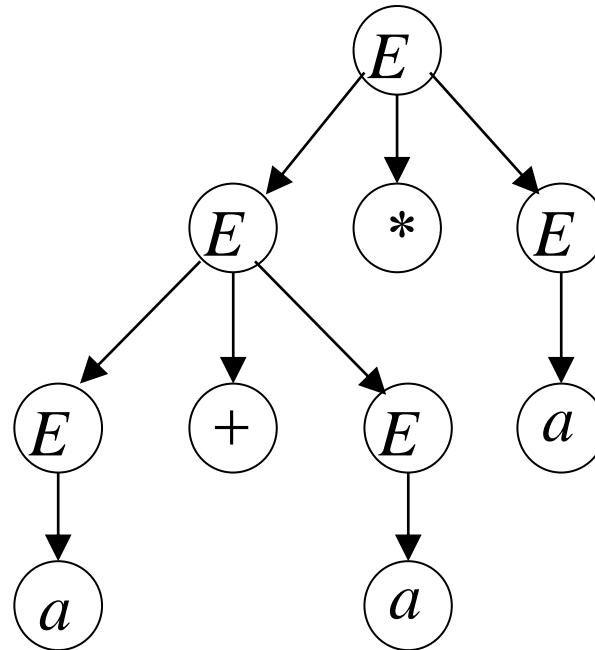
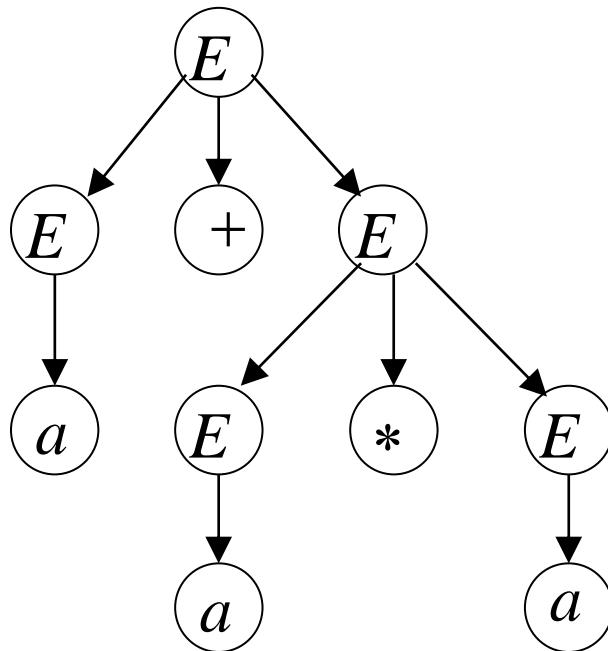
$$\begin{aligned} E &\Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \\ &\Rightarrow a + a * E \Rightarrow a + a * a \end{aligned}$$

Another leftmost  
derivation  
&  
derivation tree



The grammar  $E \rightarrow E + E \mid E * E \mid (E) \mid a$   
is ambiguous:

string  $a + a * a$  has two derivation trees



The grammar  $E \rightarrow E + E \mid E * E \mid (E) \mid a$   
is ambiguous:

string  $a + a * a$  has two leftmost derivations

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E$$

$$\Rightarrow a + a * E \Rightarrow a + a * a$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E$$

$$\Rightarrow a + a * E \Rightarrow a + a * a$$

# Definition:

A context-free grammar  $G$  is **ambiguous**

if some string  $w \in L(G)$  has:

two or more derivation trees

Or

two or more leftmost derivations

Or

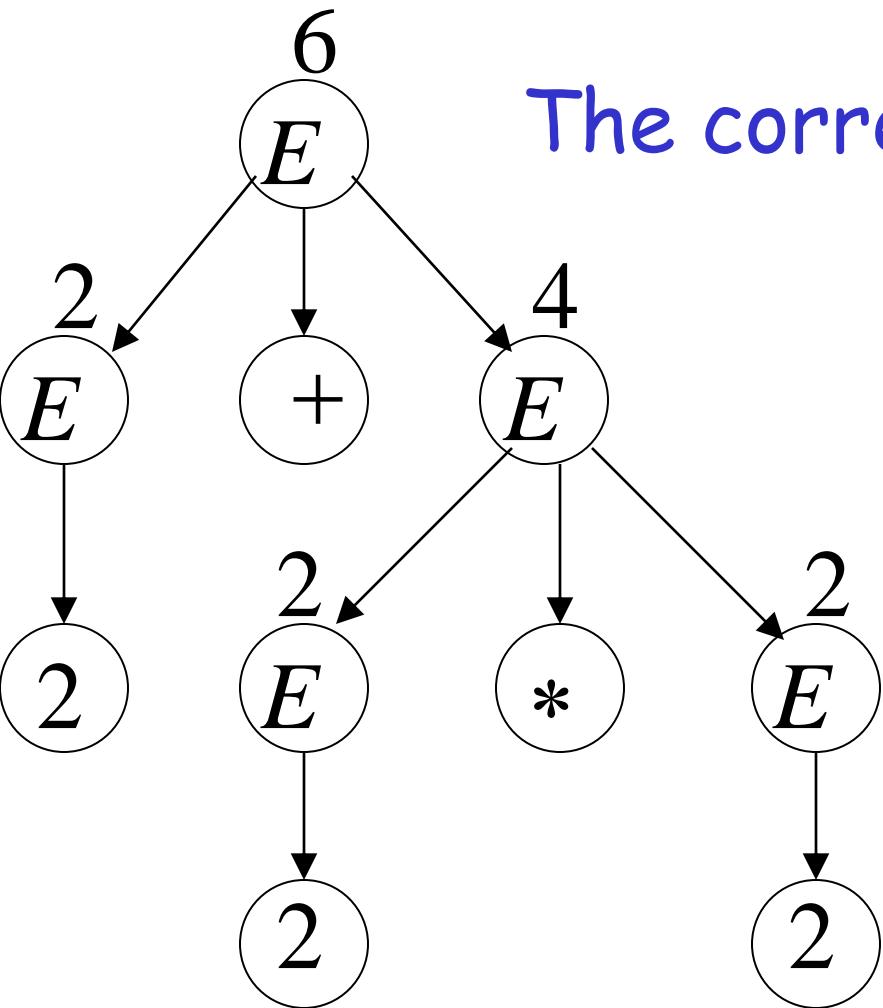
two or more rightmost derivations

# Why do we care about ambiguity?

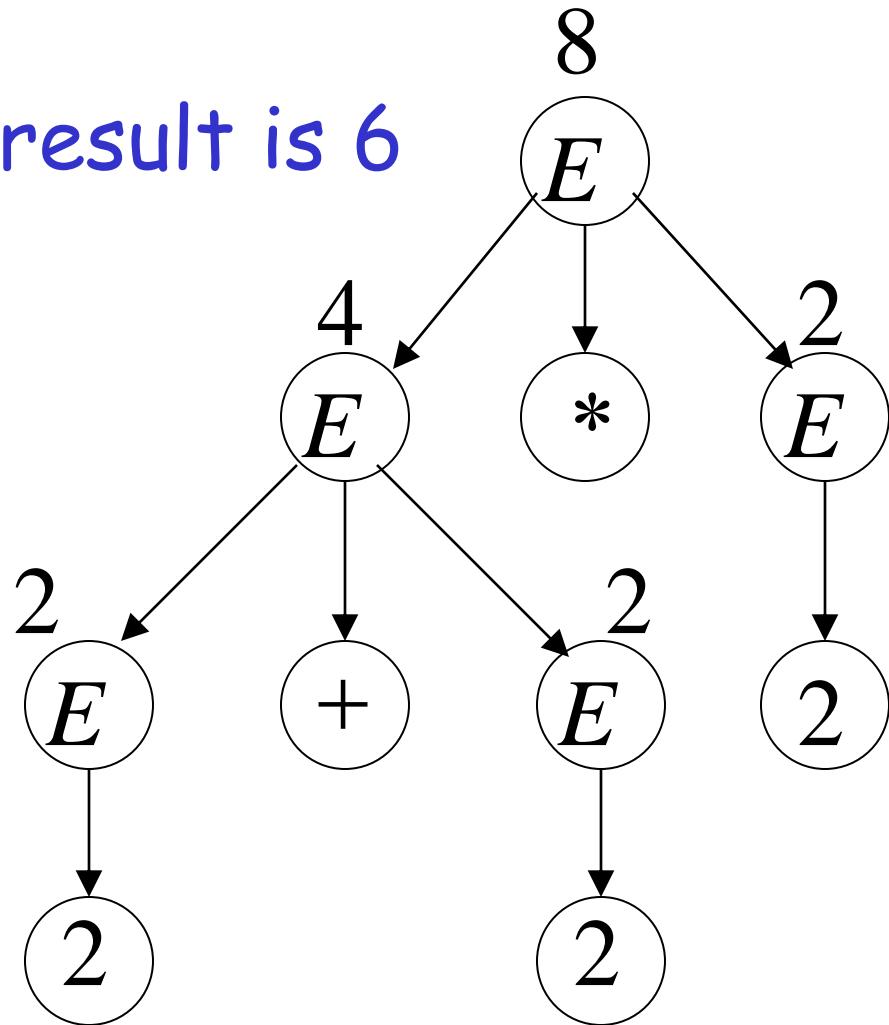
$$a + a * a$$

take  $a = 2$

$$2 + 2 * 2 = 6 \quad 2 + 2 * 2 = 8$$



The correct result is 6



# Simplifications of Context-Free Grammars

# A Substitution Rule

$$S \rightarrow aB$$
$$A \rightarrow aaA$$
$$A \rightarrow abBc$$
$$B \rightarrow aA$$
$$B \rightarrow b$$

Substitute  
 $B \rightarrow b$

Equivalent grammar

$$S \rightarrow aB | ab$$
$$A \rightarrow aaA$$
$$A \rightarrow abBc | abbc$$
$$B \rightarrow aA$$

# A Substitution Rule

$$\begin{aligned}S &\rightarrow \mathbf{aB} | ab \\A &\rightarrow aaA \\A &\rightarrow \mathbf{abBc} | abbc \\B &\rightarrow aA\end{aligned}$$

Substitute

$$B \rightarrow aA$$
$$\begin{aligned}\cancel{S &\rightarrow a\mathbf{B} | ab | \mathbf{aaA}} \\A &\rightarrow aaA \\A &\rightarrow \cancel{\mathbf{abBc}} | \mathbf{abbc} | abaAc\end{aligned}$$

Equivalent  
grammar

In general:

$$A \rightarrow xBz$$

$$B \rightarrow y_1$$

A and B are  
different variables

Substitute

$$B \rightarrow y_1$$

$$A \rightarrow xBz \mid xy_1z$$

equivalent  
grammar

# Nullable Variables

$\lambda$  – production :  $A \rightarrow \lambda$

Nullable Variable: Any variable  $A$  for which the following derivation is possible, is known as nullable.  $A \Rightarrow^* K \Rightarrow^* \lambda$

$$\begin{aligned} S &\rightarrow ACA \\ A &\rightarrow aAa \mid C \\ C &\rightarrow cC \mid \epsilon \end{aligned}$$

$$\begin{aligned} C, A, \text{ and } S \text{ are nullable} \\ A \rightarrow C \rightarrow e \\ S \rightarrow ACA \rightarrow CCA \rightarrow CCC \rightarrow e \end{aligned}$$

A grammar without nullable variables is called **non-contracting**.

A grammar may generate a language not containing  $\lambda$ , yet have some  $\lambda$ -productions or nullable variables.

In such cases, the  $\lambda$ -productions can be removed.

# Procedure: Removing Nullable Variables

Find  $V_N$  -set of Nullable variables

- If there is a production  $A \rightarrow \lambda$ , then  $A$  is nullable. Put  $A$  into  $V_N$ .
- Repeat the following until no further variables are added to  $V_N$ .
  - For all productions,

$$B \rightarrow A_1 A_2 A_3 A_4 A_5 \dots A_n$$

Where  $A_1, A_2, \dots, A_n$  are in  $V_N$

Add  $B$  to  $V_N$

# Example : Removing Nullable Variables

The set of nullable non-terminals of the grammar

$$S \rightarrow ACA$$

$$A \rightarrow aAa \mid B \mid C$$

$$B \rightarrow bB \mid b$$

$$C \rightarrow cC \mid \epsilon$$

is  $\{S, A, C\}$

$C$  is nullable

since  $C \rightarrow \epsilon$  and hence  $C \Rightarrow^* \epsilon \rightarrow V_N = \{C\}$

$A$  is nullable

since  $A \rightarrow C$ , and  $C$  is nullable,  $A$  is also nullable.  $\rightarrow$

$$V_N = \{C, A\}$$

$S$  is nullable

since  $S \rightarrow ACA$ , and  $A$  and  $C$  are nullable.  $V_N = \{S, C, A\}$

Find nullable non-terminals.

$$S \rightarrow aS \mid SS \mid bA$$

$$A \rightarrow BB$$

$$B \rightarrow CC \mid ab \mid aAbC$$

$$C \rightarrow \epsilon$$

$$V_N = \{C, B, A\}$$

$$S \rightarrow bA \rightarrow b$$

$$A \rightarrow BB \rightarrow CC \quad CC \rightarrow e$$

$$B \rightarrow CC \rightarrow e$$

# Removing $\varepsilon$ -Productions

If  $\varepsilon \notin L(G)$ , we can eliminate all productions  $A \rightarrow \varepsilon$

For every  $B$  referring to  $A$ :

$$\begin{array}{l} B \rightarrow \alpha A \beta \mid \dots \\ A \rightarrow \varepsilon \mid \dots \end{array} \quad \longrightarrow \quad \begin{array}{l} B \rightarrow \alpha \beta \mid \alpha A \beta \mid \dots \\ A \rightarrow \dots \end{array}$$

For example, if  $B \rightarrow \varepsilon$  and  $A \rightarrow BABa$

Then after eliminating the rule  $B \rightarrow \varepsilon$ , new rules for  $A$  will be added

$$A \rightarrow BABa$$

$$A \rightarrow ABa$$

$$A \rightarrow BAa$$

$$A \rightarrow Aa$$

## Removing $\varepsilon$ -Productions

Let  $G$  be

$$S \rightarrow SaB \mid aB$$

$$B \rightarrow bB \mid \varepsilon$$

After removing  $\varepsilon$ -productions, the new grammar will be

$$S \rightarrow SaB \mid Sa \mid aB \mid a$$

$$B \rightarrow bB \mid b$$

The removal of  $\varepsilon$ -productions increases the number of rules but reduces the length of derivations.

# Removing $\varepsilon$ -Productions

$G$

$$S \rightarrow ACA$$

$$A \rightarrow aAa \mid B \mid C$$

$$B \rightarrow bB \mid b$$

$$C \rightarrow cC \mid \varepsilon$$

$$V_N = \{C, A, S\}$$

The equivalent essentially non-contracting grammar  $G_L$  is

$$G_L: S \rightarrow ACA \mid CA \mid AA \mid AC \mid A \mid C \mid \varepsilon$$

$$A \rightarrow aAa \mid aa \mid B \mid C$$

$$B \rightarrow bB \mid b$$

$$C \rightarrow cC \mid c$$

Since  $S \Rightarrow^* \varepsilon$  in  $G$ , the rule  $S \rightarrow \varepsilon$  is allowed in  $G_L$ , but all other  $\varepsilon$ -productions are replaced

A grammar satisfying these conditions is called **essentially non-contracting** (only start symbol is nullable)

# Removing $\varepsilon$ -Productions

Let  $G$  be

$$S \rightarrow aS \mid SS \mid bA$$

$$A \rightarrow BB$$

$$B \rightarrow ab \mid aAbC \mid aAb \mid CC$$

$$C \rightarrow \varepsilon$$

We eliminate  $C \rightarrow \varepsilon$  by replacing:

$$B \rightarrow CC \text{ into } B \rightarrow CC, B \rightarrow C, \text{ and } B \rightarrow \varepsilon$$

$$B \rightarrow aAbC \text{ into } B \rightarrow aAbC \text{ and } B \rightarrow aAb$$

Since  $C \rightarrow \varepsilon$  is only  $C$  production  
only  $B \rightarrow \varepsilon$  and  $B \rightarrow aAb$  retained.

The new grammar:

$$S \rightarrow aS \mid SS \mid bA$$

$$A \rightarrow BB$$

$$B \rightarrow \varepsilon \mid ab \mid aAb \mid aAbC$$

# Removing $\varepsilon$ -Productions

The new grammar:

$$S \rightarrow aS \mid SS \mid bA$$

$$A \rightarrow BB$$

$$B \rightarrow \varepsilon \mid ab \mid aAb \mid aAbC$$

We eliminate  $B \rightarrow \varepsilon$  by replacing

$$A \rightarrow BB \text{ into } A \rightarrow BB, A \rightarrow B, \text{ and } A \rightarrow \varepsilon$$

Since there are other  $B$  productions, these are all retained

The new grammar:

$$S \rightarrow aS \mid SS \mid bA$$

$$A \rightarrow BB \mid B \mid \varepsilon$$

$$B \rightarrow ab \mid aAb \mid aAbC$$

# Removing $\varepsilon$ -Productions

The new grammar:

$$\begin{aligned} S &\rightarrow aS \mid SS \mid bA \\ A &\rightarrow BB \mid B \mid \varepsilon \\ B &\rightarrow ab \mid aAb \mid aAbC \end{aligned}$$

Finally we eliminate  $A \rightarrow \varepsilon$  by replacing

$$\begin{aligned} B \rightarrow aAb &\quad \text{into } B \rightarrow aAb, B \rightarrow ab \\ B \rightarrow aAbC &\quad \text{into } B \rightarrow aAbC, B \rightarrow abc \\ S \rightarrow bA &\quad \text{into } S \rightarrow bA \mid b \end{aligned}$$

The final CFG is:

$$\begin{aligned} S &\rightarrow aS \mid SS \mid bA \mid b \\ A &\rightarrow BB \mid B \\ B &\rightarrow ab \mid aAb \mid aAbC \mid aAb \end{aligned}$$

# Example: Eliminating $\epsilon$ -Productions

$S \rightarrow ABC, A \rightarrow aA \mid \epsilon, B \rightarrow bB \mid \epsilon, C \rightarrow \epsilon$

$V_N = \{A, B, C, S\}$

$A, B, C$ , and  $S$  are all nullable variables.

New grammar:

$S \rightarrow ABC \mid AB \mid AC \mid BC \mid A \mid B \mid C \mid \epsilon$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

# Removing $\varepsilon$ -Productions

$S \rightarrow ABaC$

$A \rightarrow BC$

$B \rightarrow b \mid \lambda$

$C \rightarrow D \mid \lambda$

$D \rightarrow d$

# Unit-Productions

Unit Production:  $A \rightarrow B$

(a single variable in both sides)

Observation:  $A \rightarrow A$

Is removed immediately

# Procedure: Removing of Unit Production Rules

Step-1: Draw dependency Graph

For all unit-production rules ( $C \rightarrow D$ ) in  $G$

Draw an edge from  $C$  to  $D$

Step-2: Find reachable variables

If there is a path between two nodes  $A$  and  $B$  in dependency graph,

$$A \xrightarrow{*} B \quad (1.1)$$

Step-3: Put all non-unit production of  $P$  in  $P'$  of new grammar ( $G'$ ). Next, for all  $A$  and  $B$  satisfying (1.1)

If  $B \rightarrow y_1 | y_2 | \dots | y_n$  in  $G$ .  $y_i$  is not a single variable.

Then, add

$$A \rightarrow y_1 | y_2 | \dots | y_n \text{ in } G'$$

# Example: Removing of Unit Rules

$G_L$ :

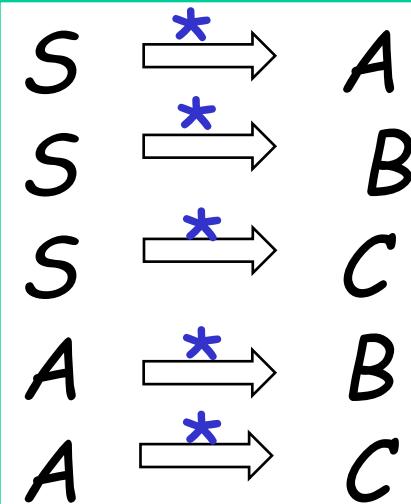
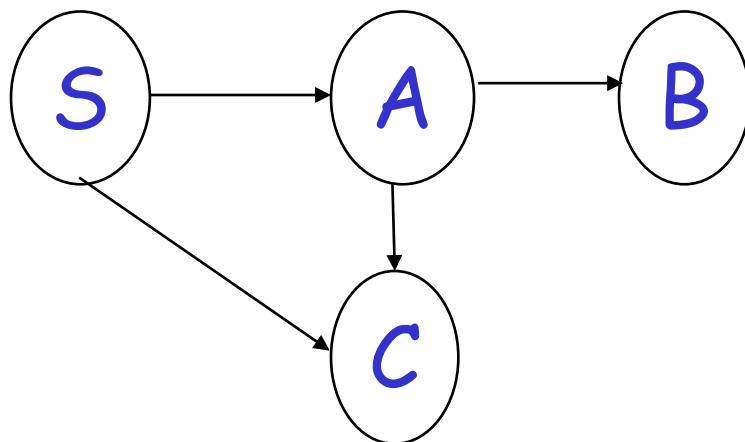
$S \rightarrow ACA \mid CA \mid AA \mid AC \mid A \mid C \mid \epsilon$

$A \rightarrow aAa \mid aa \mid B \mid C$

$B \rightarrow bB \mid b$

$C \rightarrow cC \mid c$

## Draw Dependency Graph ( $A \rightarrow C$ )



# Example: Removing of Unit Rules

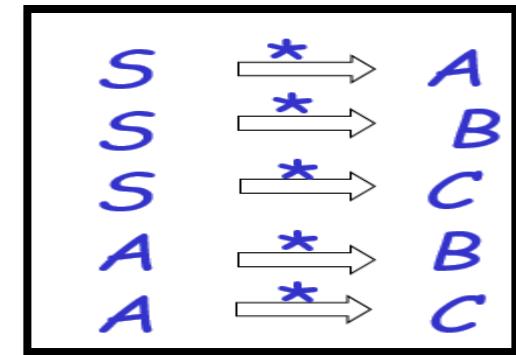
$G_L$ :

$$S \rightarrow ACA | CA | AA | AC | A | C | \epsilon$$

$$A \rightarrow aAa | aa | B | C$$

$$B \rightarrow bB | b$$

$$C \rightarrow cC | c$$



The new equivalent grammar (without unit rules)

$$G_C: S \rightarrow ACA | CA | AA | AC | \epsilon$$

$$aAa | aa | bB | b | cc | c$$

$$A \rightarrow aAa | aa | bB | b | cc | c$$

$$B \rightarrow bB | b$$

$$C \rightarrow cc | c$$

# Exercise: Remove Unit Production Rules

$G1$

$S \rightarrow aA$

$A \rightarrow a$

$A \rightarrow B$

$B \rightarrow A$

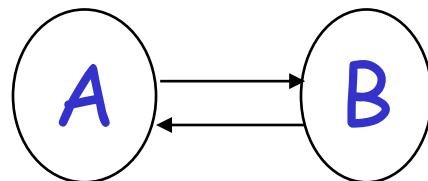
$B \rightarrow bb$

$G2$

$S \rightarrow Aa \mid B$

$B \rightarrow A \mid bb$

$A \rightarrow a \mid bc \mid B$



$A^* \rightarrow B$

$B^* \rightarrow A$

$G'$

$S \rightarrow aA$

$A \rightarrow a \mid bb$

$B \rightarrow bb \mid a$

# Remove Unit Production Rules

Original Grammar

$$S \rightarrow T \mid S + T$$

$$T \rightarrow F \mid F^* T$$

$$F \rightarrow a \mid (S)$$

After removing Unit production rules

$$S \rightarrow a \mid (S) \mid F^* T \mid S + T$$

$$T \rightarrow a \mid (S) \mid F^* T$$

$$F \rightarrow a \mid (S)$$

# Useless Productions

A variable  $A$  is useful (it occurs in at least one derivation.) if:

It is **reachable**: occurs in a sentential form  $S \Rightarrow^* \alpha A \beta$

It is **live**: generates a terminal string  $A \Rightarrow^* w \in T^*$

A variable  $A$  is useless if:

A does not occur in any sentential form

It cannot be reached from the start symbol

OR

A does not generate any string of terminals.

It cannot derive a terminal string

A variable is useful if it occurs in at least one sentence  $w \in L(G)$

Any production rule consisting of useless variable is a **useless production**.

# Removing Useless Productions

To eliminate useless symbols:

First: Find the set TERM that contains all variables that derive a terminal string

$$A \Rightarrow^* w, \text{ where } w \in T^*$$

Variables NOT in TERM are useless, they cannot contribute to generate strings in  $L(G)$

Second: Find the set REACH that contains all variables  $A$  that are reachable from  $S$

$$S \Rightarrow^* \alpha A \beta$$

Variables NOT in REACH are useless, they cannot contribute to generate strings in  $L(G)$

# Removing Useless Productions

First: Find the set TERM that contains all non-terminals that derive a terminal string

$$A \Rightarrow^* w, \text{ where } w \in T^*$$

Initialize TERM = {} --Empty set

1. Add all variables  $A \in V$  in TERM if there is a production rule

$$A \rightarrow x_1 x_2 \dots x_n \text{ For all } x_i \in T$$

2. Repeat the following step until no more variables are added to TERM

For  $A \in V$  for which P has a production of the form

$$A \rightarrow x_1 x_2 \dots x_n \text{ For all } x_i \in \text{TERM} \cup T$$

Add A to TERM

Take  $P_1$  as all the productions in P whose symbols are all in TERM

# Removing Useless Productions

Example Grammar:

$$S \rightarrow aS \mid A \mid C$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

$$C \rightarrow aCb$$

**First:** find all variables that can produce strings with only terminals

$$S \rightarrow aS \mid A \mid C$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

$$C \rightarrow aCb$$

TERM

Round 1:  $\{A, B\}$

$$S \rightarrow A$$

Round 2:  $\{A, B, S\}$

Keep only the variables  
that produce terminal symbols:  $\{A, B, S\}$   
(the rest variables are useless)

$$S \rightarrow aS \mid A \mid \cancel{C}$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

$$\cancel{C \rightarrow aCb}$$



$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

Remove useless productions

# Removing Useless Productions

Second: Find the set REACH that contains all non-terminals  $A$  that are reachable from  $S$

$$S \Rightarrow^* \alpha A \beta$$

Step-1: Draw dependency Graph

For all production rules ( $C \rightarrow xDy$ ) in  $G$  ( $C, D$  are Variables)

Draw an edge from  $C$  to  $D$

Step-2: Find reachable variables

If there is a path between two nodes  $S$  and  $A$  in the dependency graph, then variable  $A$  is reachable from  $S$ .

Step-3: Put all production in  $P'$  of new grammar ( $G'$ ) such that  $A$  is reachable from  $S$  and keep only those productions having useful variables and terminals.

# Removing Useless Productions

Example Grammar:

$$S \rightarrow aS \mid A \mid C$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

$$C \rightarrow aCb$$

$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

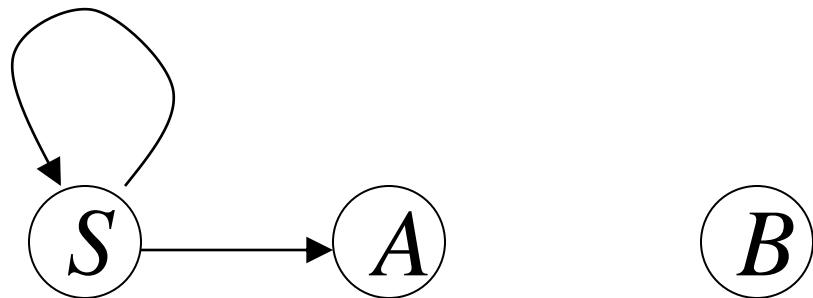
**Second:** Find all variables  
reachable from  $S$

Use a Dependency Graph

$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

$$B \rightarrow aa$$



not  
reachable

B is useless.

Keep only the variables  
reachable from S

(the rest variables are useless)

Final Grammar

$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$



$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

~~$B \rightarrow aa$~~

Remove useless productions

# Removing Useless Productions

$$G = (\{S, A, B, C, D, E, F\}, \{a, b, c\}, S, P)$$

$$\begin{aligned} G: S &\rightarrow AC \mid BS \mid B \\ A &\rightarrow aA \mid aF \\ B &\rightarrow CF \mid b \\ C &\rightarrow CC \mid D \\ D &\rightarrow aD \mid BD \mid C \\ E &\rightarrow aA \mid BSA \\ F &\rightarrow bB \mid b \end{aligned}$$

$$R-1: TERM = \{\}$$

$$R-2: TERM = \{B, F\}$$

$$R-3: TERM = \{B, F, S, A, E\}$$

# Removing Useless Productions

$$TERM = \{F, B, S, A, E\}$$

C and D do not belong to TERM, so all rules containing C and D are removed

The new grammar is

$$\begin{array}{ll} G_T: & S \rightarrow BS \mid B \\ & A \rightarrow aA \mid aF \\ & B \rightarrow b \\ & E \rightarrow aA \mid BSA \\ & F \rightarrow bB \mid b \end{array}$$

OLD Grammar

$$\begin{array}{ll} G: S \rightarrow AC \mid BS \mid B \\ & A \rightarrow aA \mid aF \\ & B \rightarrow CF \mid b \\ & C \rightarrow cC \mid D \\ & D \rightarrow aD \mid BD \mid C \\ & E \rightarrow aA \mid BSA \\ & F \rightarrow bB \mid b \end{array}$$

# Removing Useless Productions

$G_T$ :

$$S \rightarrow BS \mid B$$

$$A \rightarrow aA \mid aF$$

$$B \rightarrow b$$

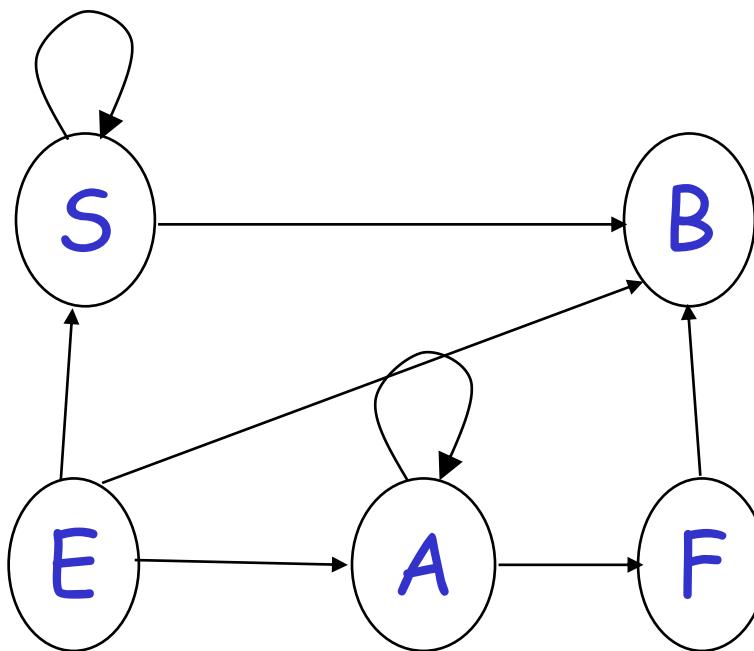
$$E \rightarrow aA \mid BSA$$

$$F \rightarrow bB \mid b$$

$A$ ,  $E$ , and  $F$  can not be reached from  $S$ .

$A$ ,  $E$  and  $F$  are useless.

so all rules containing  $A$ ,  $E$  and  $F$  are removed



Dependency  
Graph

# Removing Useless Productions

The new grammar is

$$G_U: \quad S \rightarrow BS \mid B \\ B \rightarrow b$$

$$L(G_U) = b^+$$

$$G_U = (\{S, B\}, \{b\}, S, P)$$

<i>OLD Grammar</i>	$G: S \rightarrow AC \mid BS \mid B$
	$A \rightarrow aA \mid aF$
	$B \rightarrow CF \mid b$
	$C \rightarrow cC \mid D$
	$D \rightarrow aD \mid BD \mid C$
	$E \rightarrow aA \mid BSA$
	$F \rightarrow bB \mid b$

# Exercise: Removing Useless Productions

$S \rightarrow AB \mid CD \mid ADF \mid CF \mid EA$

$A \rightarrow abA \mid ab$

$B \rightarrow bB \mid aD \mid BF \mid aF$

$C \rightarrow cB \mid EC \mid Ab$

$D \rightarrow bB \mid FFB$

$E \rightarrow bC \mid AB$

$F \rightarrow abbF \mid baF \mid bD \mid BB$

$G \rightarrow EbE \mid CE \mid ba$

# Simplifications of CFGs

**Step 1:** Remove Nullable Variables

Introduces Unit Productions

**Step 2:** Remove Unit-Productions

**Step 3:** Remove Useless Variables

# Simplify the CFG

*G1*

$$S \rightarrow AB \mid CD \mid ADF \mid CF \mid EA$$

$$A \rightarrow abA \mid ab$$

$$B \rightarrow bB \mid aD \mid BF \mid aF$$

$$C \rightarrow cB \mid EC \mid Ab$$

$$D \rightarrow bB \mid FFB$$

$$E \rightarrow bC \mid AB$$

$$F \rightarrow abbF \mid baF \mid bD \mid BB$$

$$G \rightarrow EbE \mid CE \mid ba$$

*G2*

$$S \rightarrow aA \mid aBB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow bB \mid bbC$$

$$C \rightarrow B$$

# Normal Forms for Context-free Grammars

# Chomsky Normal Form (CNF)

A context-free grammar (CFG)  $G$  is in Chomsky Normal Form if all productions are of the form.

$$A \rightarrow BC$$

variable

or

$$A \rightarrow a$$

terminal

Where  
and

$$\begin{aligned} A, B, C &\in V \\ a &\in T \end{aligned}$$

## Examples:

$$S \rightarrow AS$$
$$S \rightarrow a$$
$$A \rightarrow SA$$
$$A \rightarrow b$$

Chomsky  
Normal Form

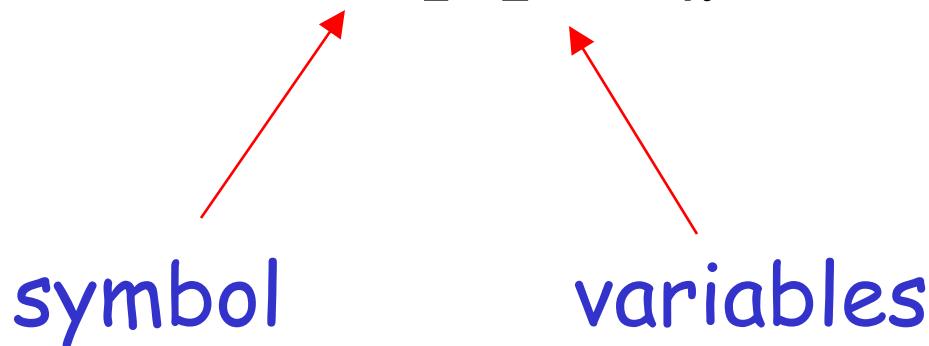
$$S \rightarrow AS$$
$$S \rightarrow AAS$$
$$A \rightarrow SA$$
$$A \rightarrow aa$$

Not Chomsky  
Normal Form

# Greibach Normal Form

A context-free grammar (CFG)  $G$  is in Greibach Normal Form (GNF) if all productions are of the form.

$$A \rightarrow a V_1 V_2 \cdots V_k \quad k \geq 0$$



Where  
and

$$A, V_1, V_2, \dots, V_k \in V$$
$$a \in T$$

## Examples:

$S \rightarrow cAB$

$A \rightarrow aA \mid bB \mid b$

$B \rightarrow b$

Greibach  
Normal Form

$S \rightarrow abSb$

$S \rightarrow aa$

Not Greibach  
Normal Form

# Pushdown Automata (PDA)

# Regular Languages (Review)

- Every Regular language has
  - regular expression / regular grammar / Finite Automata
- Every regular language RL is **Context-Free Language.**
- But some CFL are not regular:

The language  $L = \{a^n b^n : n \geq 1\}$  has CFG  $S \rightarrow aSb \mid ab$

The language  $\{ww^R : w \in \{a, b\}^*\}$  has CFG  $S \rightarrow aSa \mid bSb \mid a \mid b \mid \epsilon$

# Context-Free Languages (Review)

A grammar  $G = (V, T, S, P)$  is CFG if all production rules have the form

$$A \rightarrow x, \text{ where } A \in V, \text{ and } x \in (V \cup T)^*$$

A language  $L$  is CFL iff there is a CFG  $G$  such that  $L = L(G)$

All regular languages, and some non-regular languages, can be generated by CFGs

The family of regular languages is a proper subset of the family of context-free languages.

# Stack Memory

The language  $L = \{wcw^R : w \in \{a, b\}^*\}$  is CFL  
but not RL

We can not have a DFA for  $L$

Problem is **memory**, DFA cannot remember left hand substring

What kind of memory do we need to be able to recognize strings in  $L$ ?

Answer: a **stack**

# Stack Memory

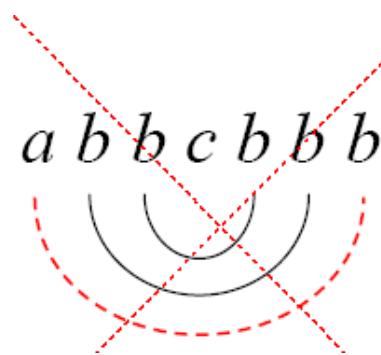
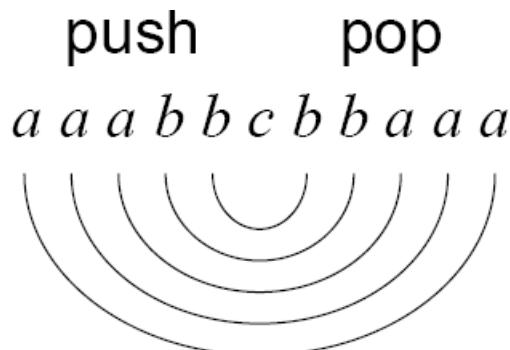
Example:  $u = aaabb \ c \ bbaaa \in L$

We push the first part of the string onto the stack and

after the  $c$  is encountered

start popping characters from the stack and  
matching them with each character.

if everything matches, this string  $u \in L$



# Stack Memory

We can also use a stack for counting out equal numbers of a's and b's.

Example:

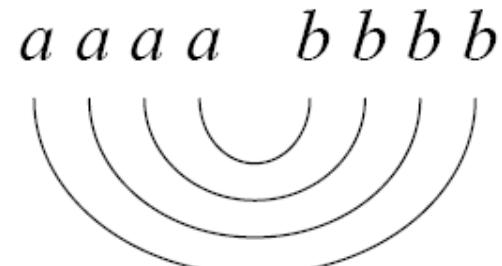
$$L = \{a^n b^n : n \geq 0\}$$

$$w = aaaabbbbb \in L$$

Push the a's onto the stack, then pop an a off and match it with each b.

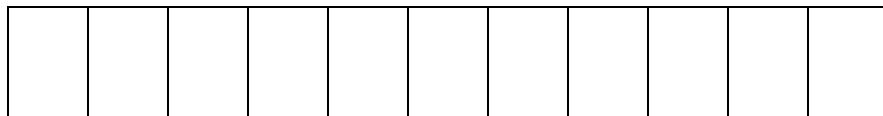
If we finish processing the string successfully (and there are no more a's on our stack), then the string belongs to  $L$ .

$w = aaabbbbb$  Reject

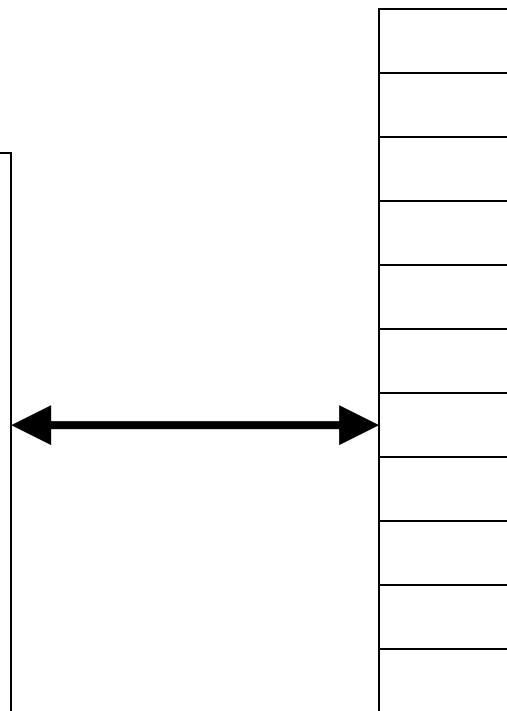
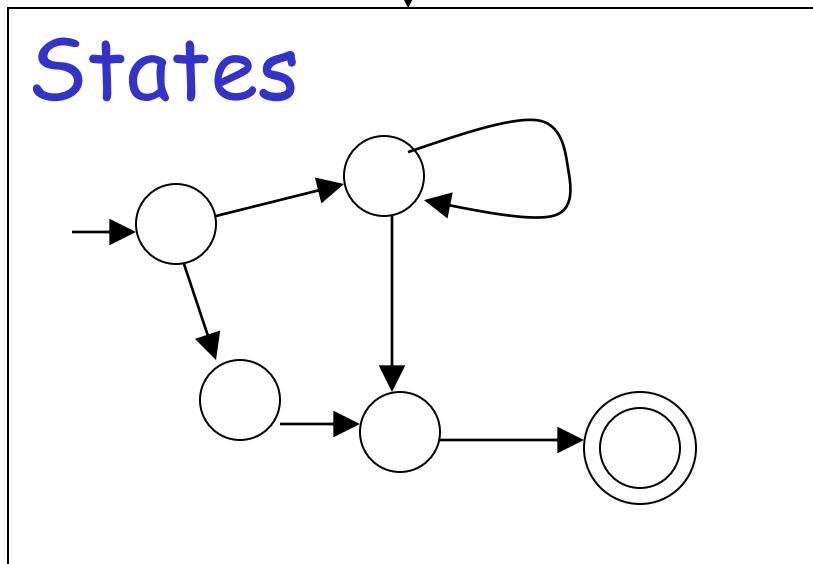


# Pushdown Automaton -- PDA

Input String



Stack



# Nondeterministic Push-Down Automata

A language is **context free** iff some Nondeterministic Pushdown Automata (NPDA) recognizes (accepts) it.

Intuition: **NPDA = NFA + one stack for memory.**

Stack remembers info about previous part of string

E.g.,  $a^n b^n$

Deterministic Pushdown Automata (DPDA) can accept some but not all of the CFLs.

Thus, there is no longer an equivalence between the deterministic and nondeterministic versions,  
i.e., languages recognized by DPDA are a proper subset of context-free languages.

# NPDA

A NPDA is a seven-tuple  $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$   
where

$Q$  finite set of states

$\Sigma$  finite set of input alphabet

$\Gamma$  finite set of stack alphabet  $\Sigma \subseteq \Gamma$

$\delta$  transition function

$Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$

$q_0$  start state

$q_0 \in Q$

$z$  initial stack symbol

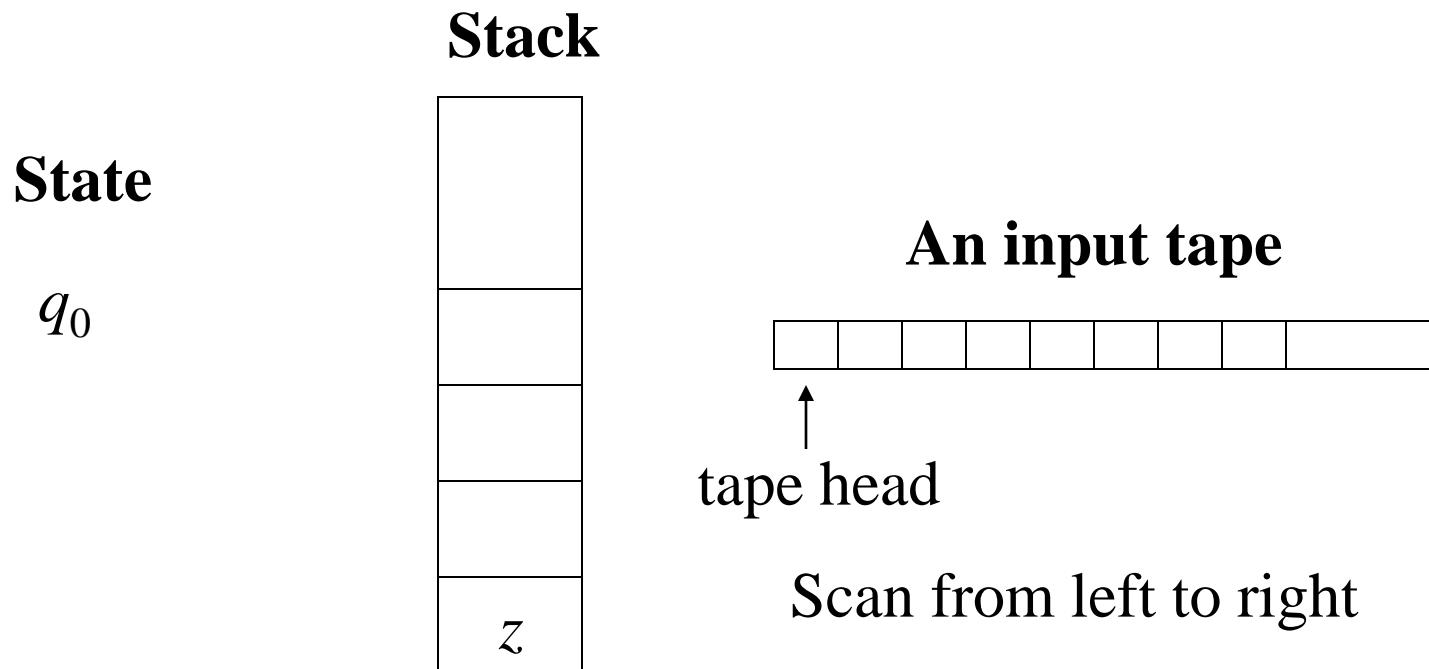
$z \in \Gamma$

$F$  set of final states

$F \subseteq Q$

# NPDA

There are three things in an NPDA:

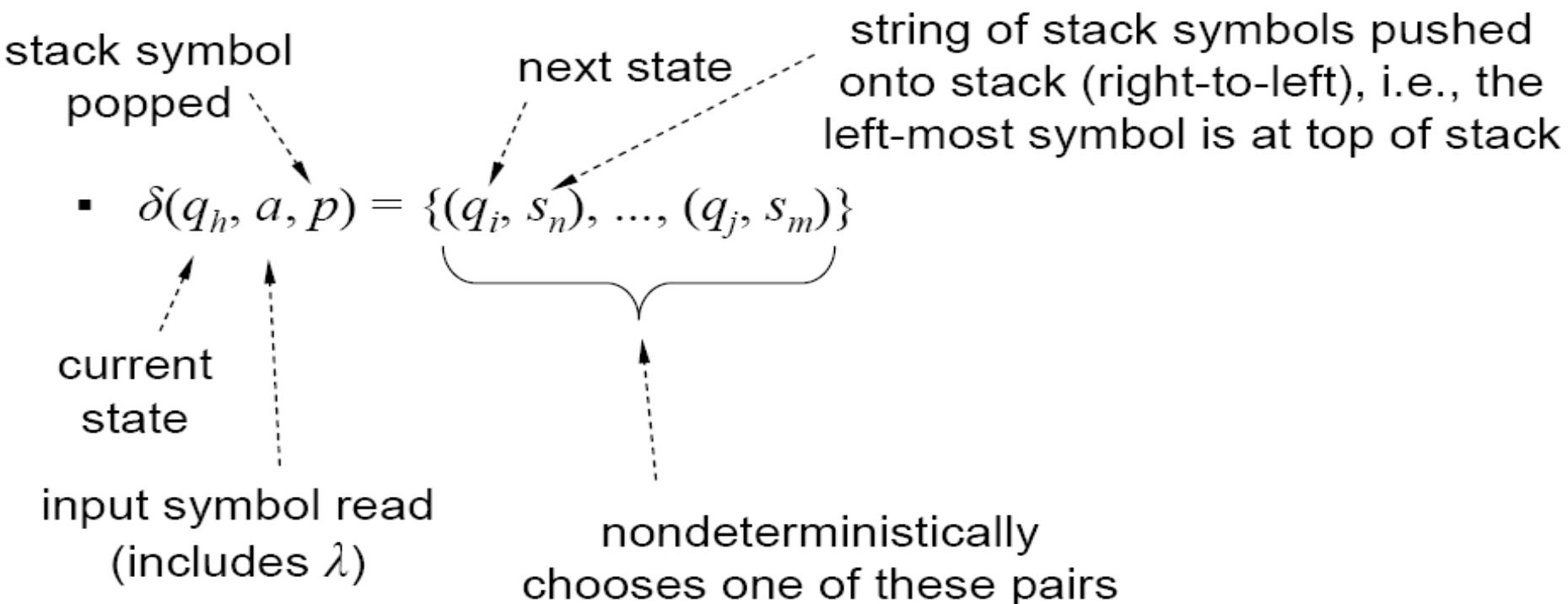


# NPDA : Transition Function

The transition function deserves further explanation

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$$

**INPUT**   **OUTPUT**



# NPDA : Transition Function

In an NPDA, we read an input symbol in a state, but

we also need to know what is on the stack before

- we can decide what is new state.
  - When moving to the new state, we also need to decide what to do with the stack.

# NPDA : Transition Function

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$$

The second argument of  $\delta$  is  $\varepsilon$ .

It means the transition is possible without consuming the input symbol.

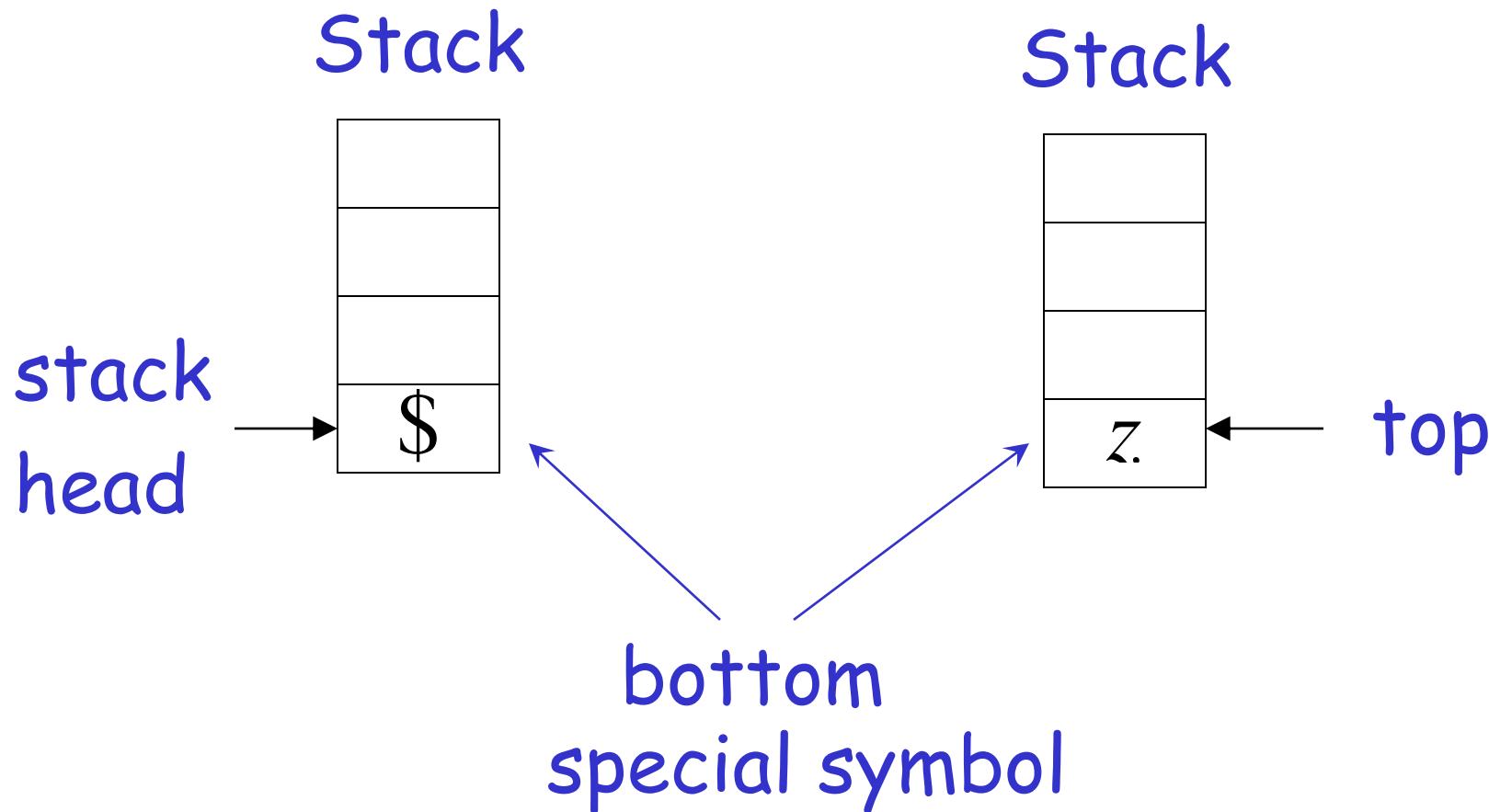
The third argument of  $\delta$  is from  $\Gamma$ .

It means the transition is not possible if the stack is empty.

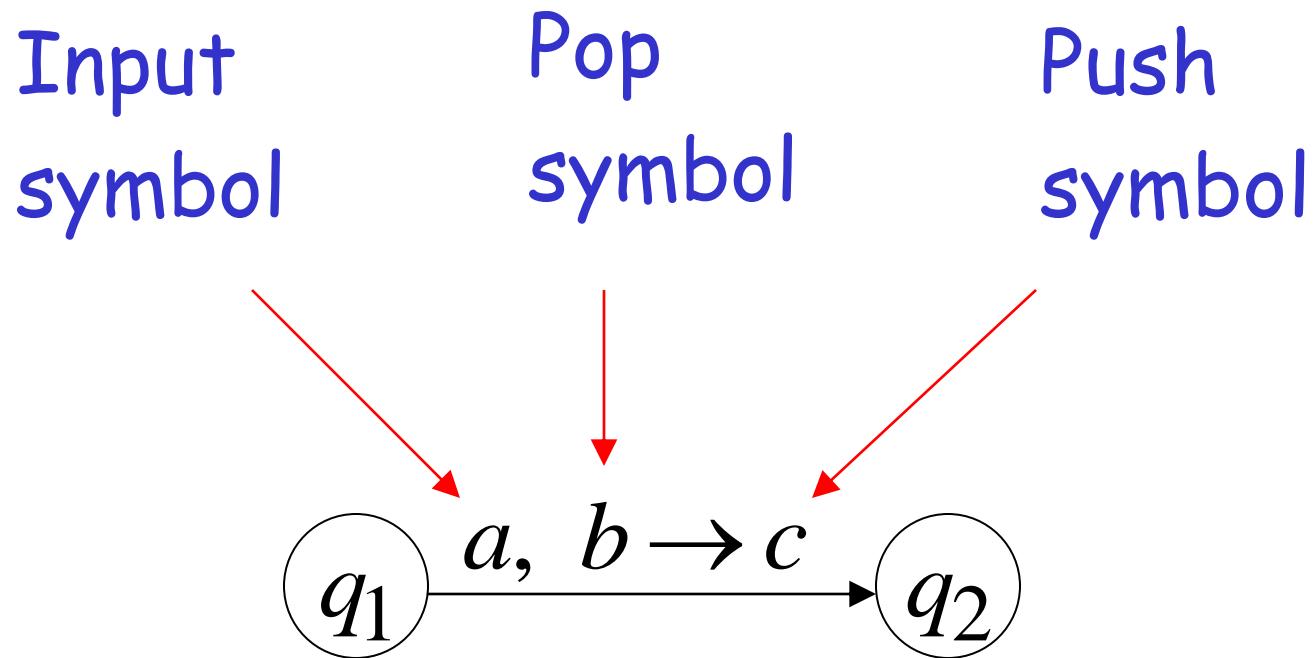
Finally, the finite subset is necessary because  $Q \times \Gamma^*$  is an infinite set.

# Initial Stack Symbol

*Initially, It is assumed that stack consists of start symbol (z|\$|#|0)*

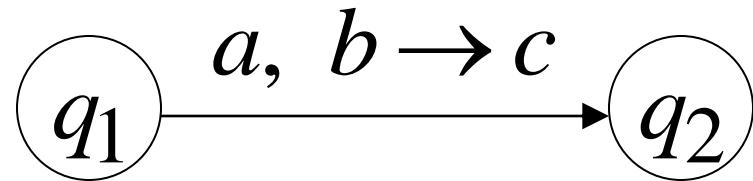


# The Transition



$$\delta(q_1, a, b) = \{ (q_2, c) \}$$

$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$



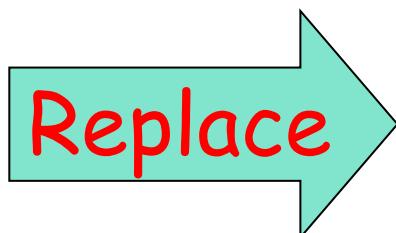
$$\delta(q_1, a, b) = \{ (q_2, c) \}$$



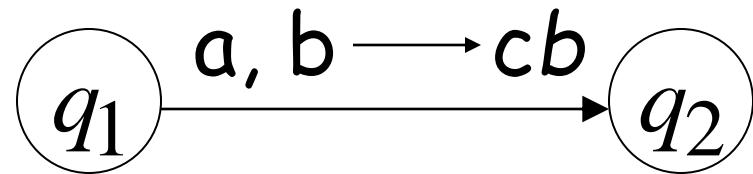
stack

b
h
e
\$

top



c
h
e
\$



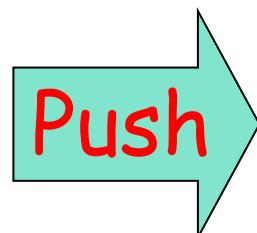
$$\delta(q_1, a, b) = \{(q_2, cb)\}$$



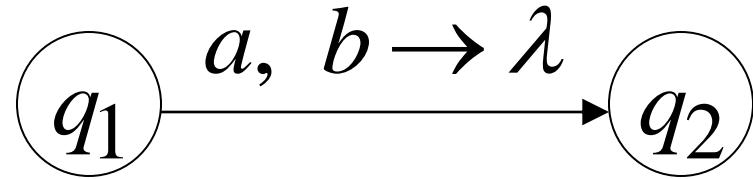
stack

$b$
$h$
$e$
$\$$

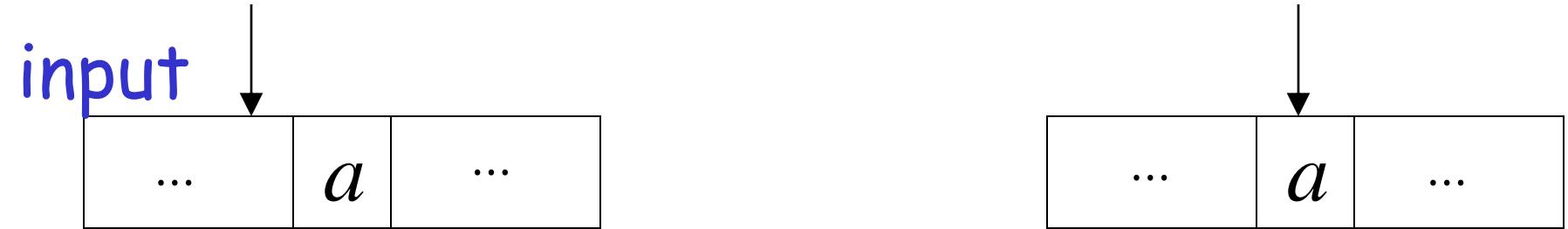
top



$c$
$b$
$h$
$e$
$\$$



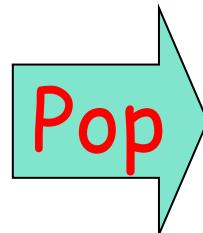
$$\delta(q_1, a, b) = \{(q_2, \lambda)\}$$



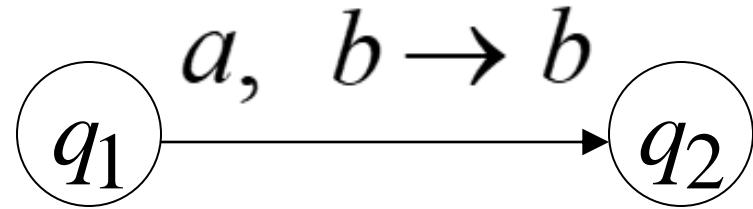
stack

$b$
$h$
$e$
$\$$

top



$h$
$e$
$\$$



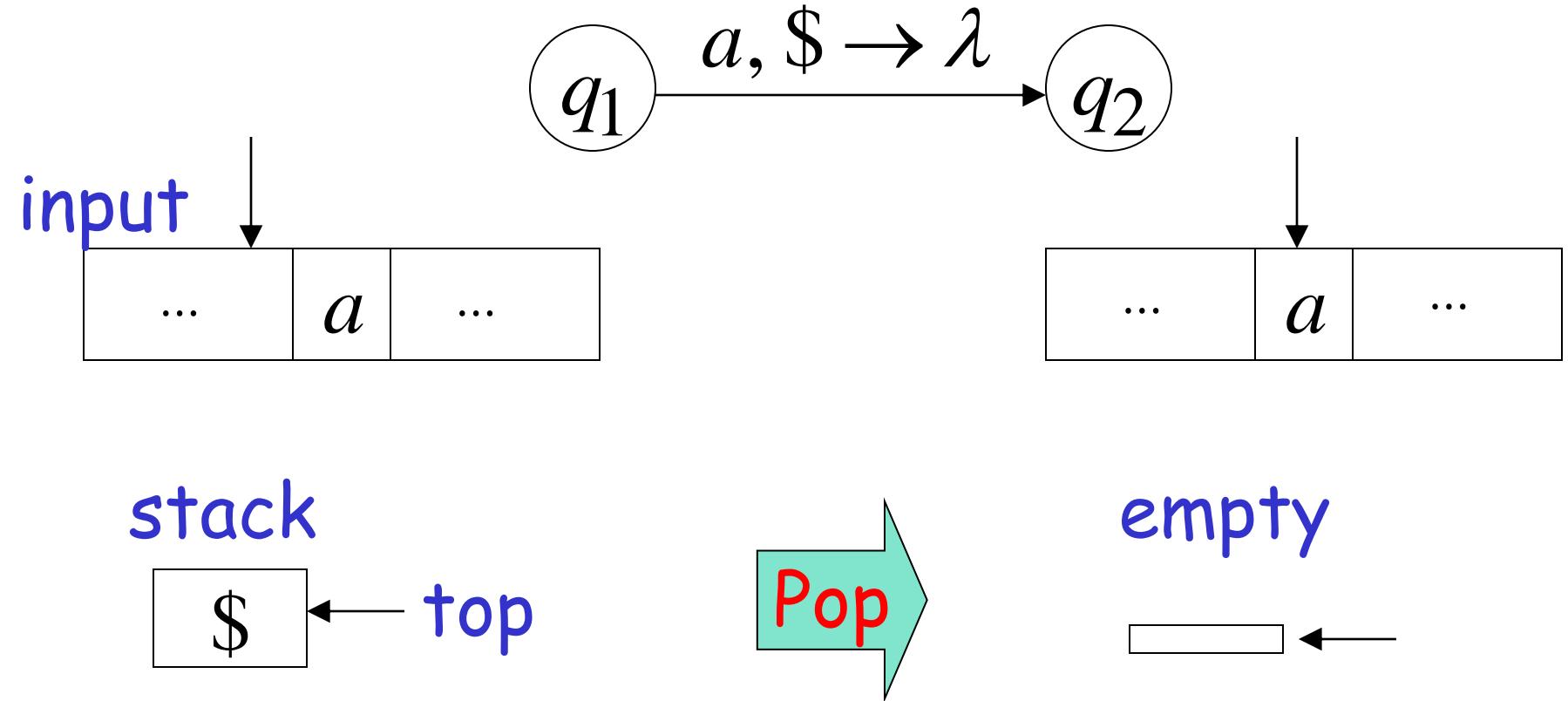
$$\delta(q_1, a, b) = \{(q_2, b)\}$$



<i>b</i>
<i>h</i>
<i>e</i>
\$

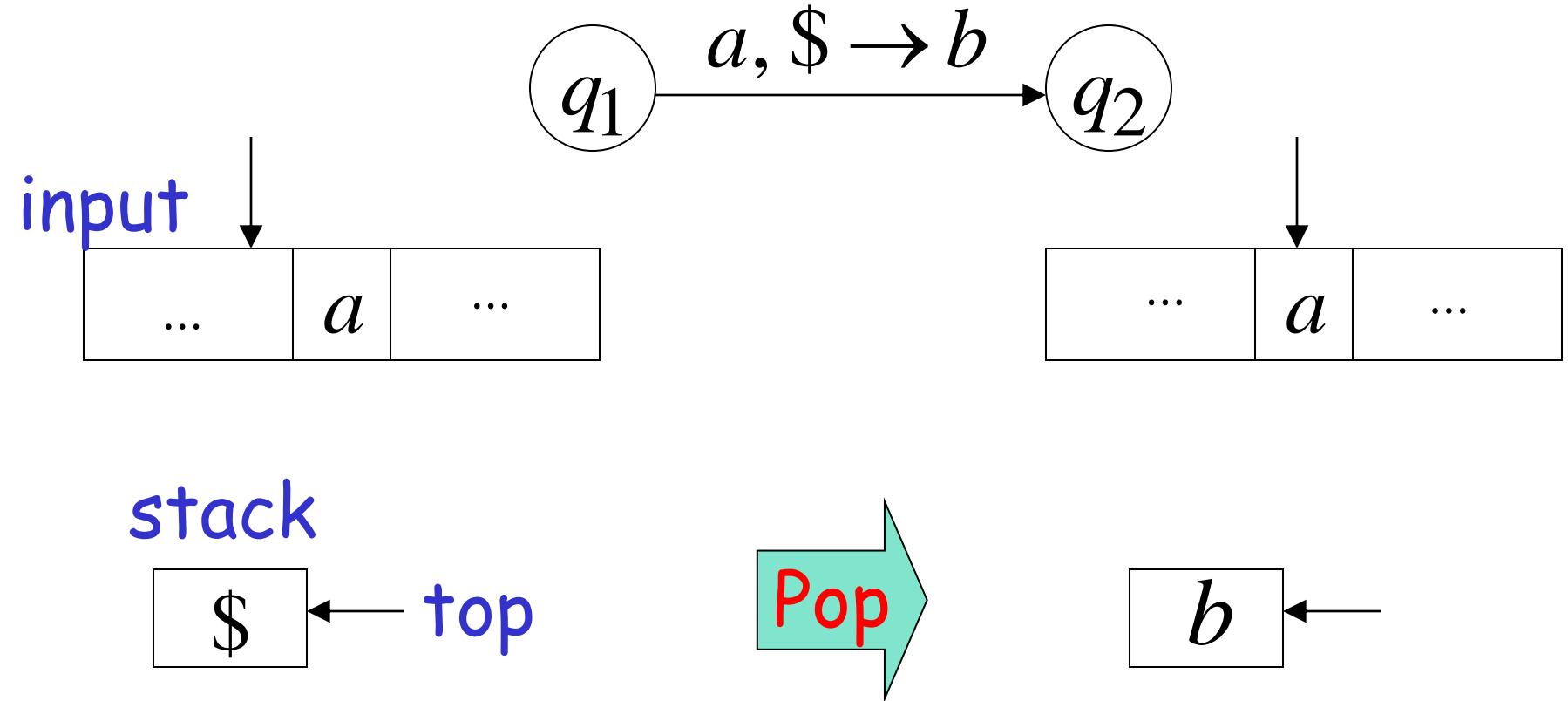
<i>b</i>
<i>h</i>
<i>e</i>
\$

# A Possible Transition

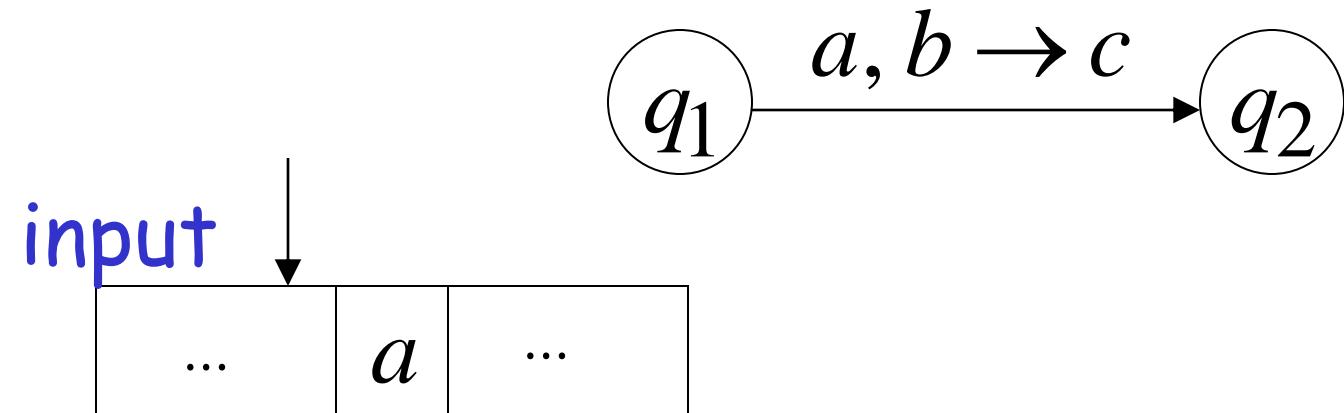


$$\delta(q_1, a, \$) = \{ (q_2, \lambda) \}$$

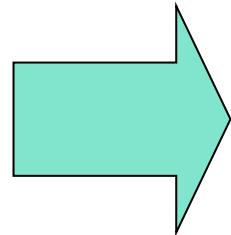
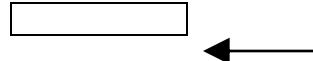
# A Possible Transition



# A Bad Transition



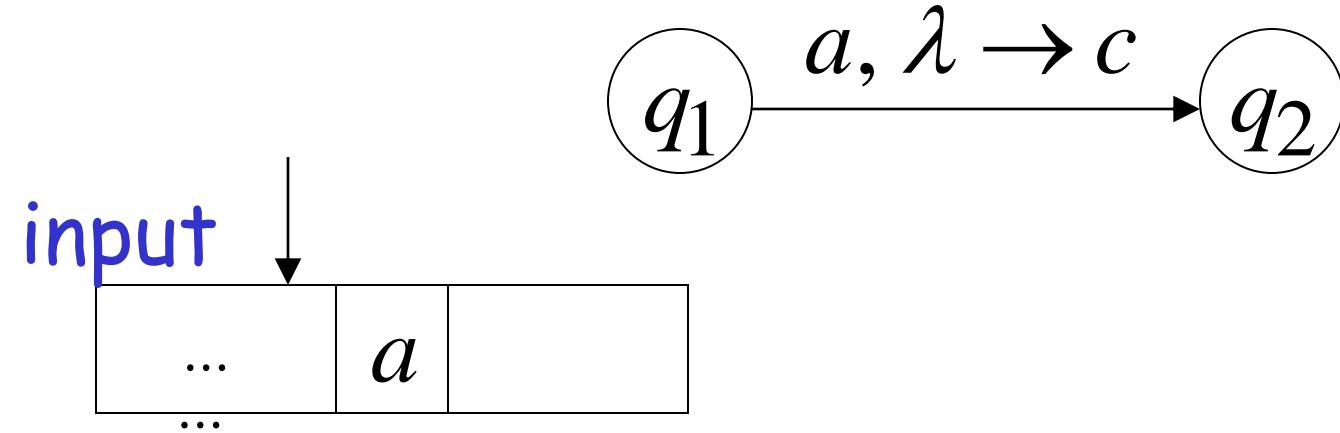
Empty stack



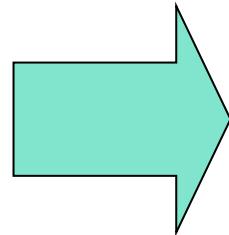
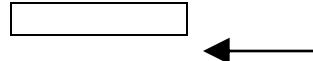
HALT

The automaton **Halts** in state  $q_1$  and **Rejects** the input string

# A Bad Transition



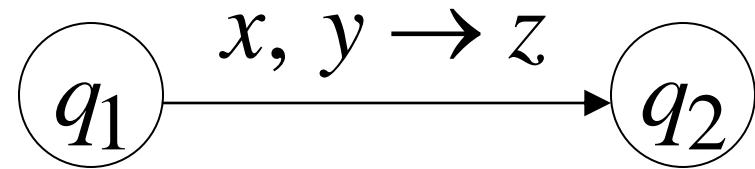
Empty stack



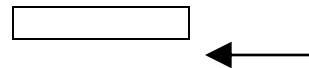
HALT

The automaton **Halts** in state  $q_1$  and **Rejects** the input string

No transition is allowed to be followed  
When the stack is empty

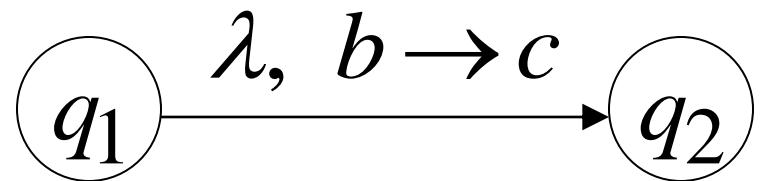
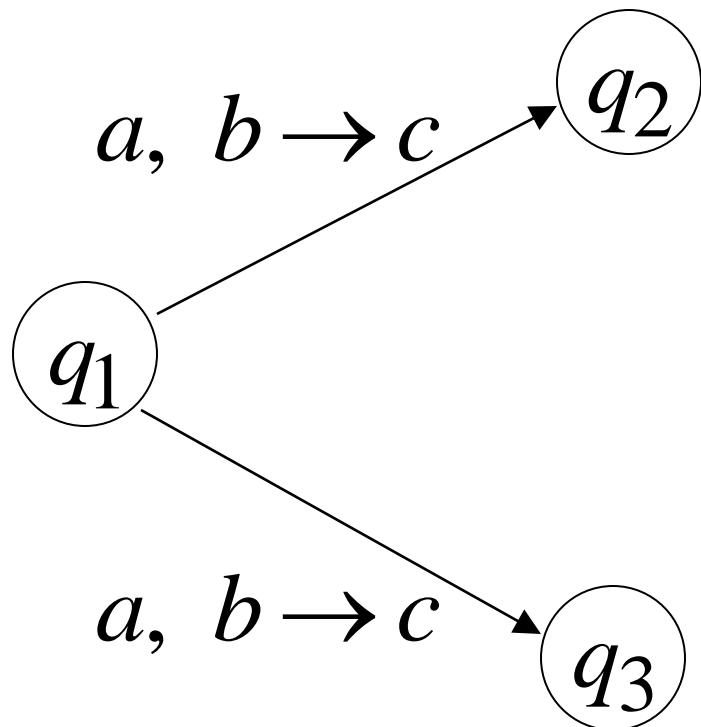


Empty stack



# Non-Determinism

$$\delta(q_1, a, b) = \{(q_2, c), (q_3, d)\}$$



$\lambda$  – transition

These are allowed transitions in a  
Non-deterministic PDA (NPDA)

# Construct NPDA for $L = \{a^n b^n : n \geq 0\}$

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F),$$

where

$$Q = \{q_0, q_1, q_2, q_3\}$$

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

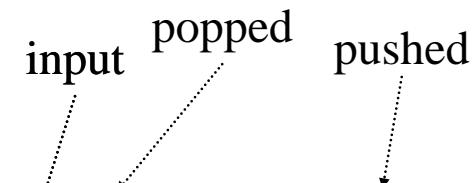
$$\Gamma = \{Z, a\}$$

$$\delta$$

$q_0$  is the start state

$Z$  is the initial stack symbol

$$F = \{q_3\}$$



$$\delta(q_0, \varepsilon, Z) = \{(q_3, \varepsilon)\}$$

$$\delta(q_0, a, Z) = \{(q_1, aZ)\}$$

$$\delta(q_1, a, a) = \{(q_1, aa)\}$$

$$\delta(q_1, b, a) = \{(q_2, \varepsilon)\}$$

$$\delta(q_2, b, a) = \{(q_2, \varepsilon)\}$$

$$\delta(q_2, \varepsilon, Z) = \{(q_3, \varepsilon)\}$$

*aaabbb*

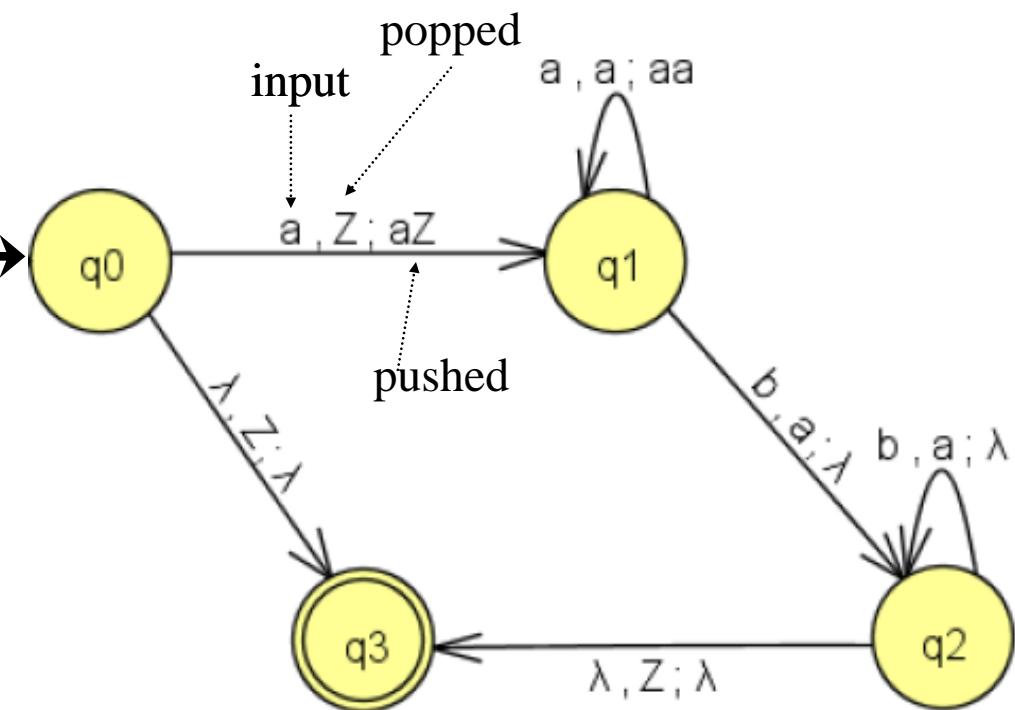
*Stack - \$aaa*

*Stack after  
first b - \$*

# Construct NPDA

$$L = \{a^n b^n : n \geq 0\}$$

$\{e, ab, aabb, aaabbb, \dots\}$   
bbaa



w is given string  
state=q0, sym =w[0] , top =Z  
While no more symbols in w

If state=q0, sym = $\lambda$ , top =Z  
state = q3, top =  $\lambda$

elif state=q0, sym =a, top =Z  
state = q1, top = aZ

elif state=q1, sym =a, top =aZ  
state = q1, top = aa

elif state=q1, sym =b, top =aa  
state = q2, top =  $\lambda$

elif state=q2, sym =b, top = $\lambda$   
state = q2, top =  $\lambda$

elif state=q2, sym = $\lambda$ , top = $\lambda$   
state = q3, top =  $\lambda$

else  
break

If state = q3; w = $\lambda$ ; Accepted  
Else ; Rejected

state=q2, sym =  $\lambda$ , top = $\lambda$

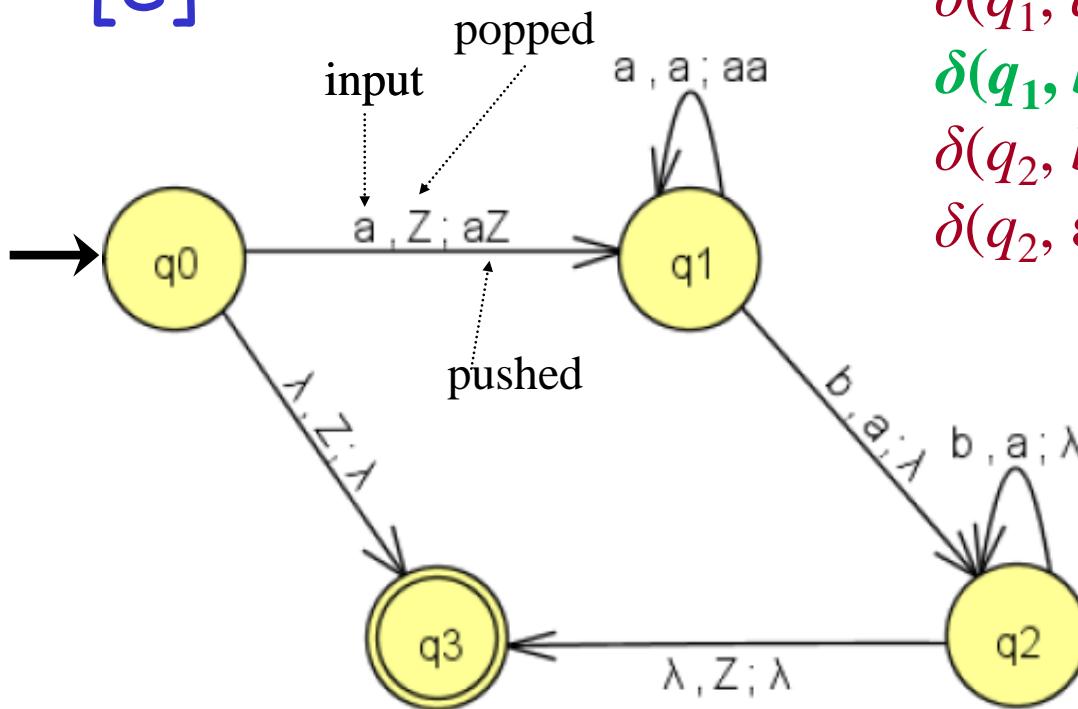
# Construct NPDA

$w = aaabb$

$w[0] = e$

State =  $q_3$

Stack = [e]



$$\delta(q_0, \varepsilon, Z) = \{(q_3, \varepsilon)\}$$

$$\delta(q_0, a, Z) = \{(q_1, aZ)\}$$

$$\delta(q_1, a, a) = \{(q_1, aa)\}$$

$$\delta(q_1, b, a) = \{(q_2, \varepsilon)\}$$

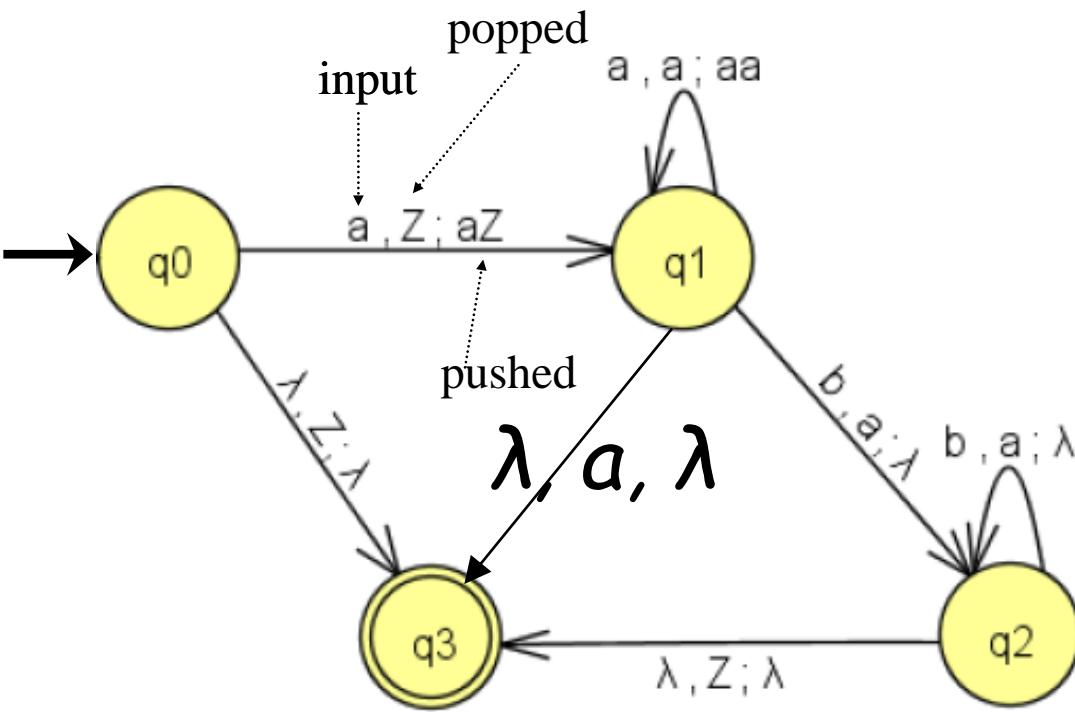
$$\delta(q_2, b, a) = \{(q_2, \varepsilon)\}$$

$$\delta(q_2, \varepsilon, Z) = \{(q_3, \varepsilon)\}$$

# Construct NPDA

Language:  $L = \{a^n b^n : n \geq 0\} \cup \{a\}$

Add one more transition for  $\{a\}$



# NPDA

A NPDA configuration (**Instantaneous Description**) is represented by

$[q_n, u, \alpha]$  where

$q_n$  :current state

$u$  :unprocessed input (Total)

$\alpha$  :current stack content (Total)

if  $\delta(q_n, a, A) = \{(q_m, B)\}$

then  $[q_n, au, A\alpha] \vdash [q_m, u, B\alpha]$

The notation  $[q_n, u, \alpha] \vdash [q_m, v, \beta]$  indicates that configuration  $[q_m, v, \beta]$  is obtained from  $[q_n, u, \alpha]$  by a single transition of the NPDA

# NPDA

The notation  $[q_n, u, \alpha] \xrightarrow{*} [q_m, v, \beta]$  indicates that configuration  $[q_m, v, \beta]$  is obtained from  $[q_n, u, \alpha]$  by zero or more transitions of the NPDA

A computation of a NPDA is a sequence of transitions beginning with start state.

# Acceptance

A string is accepted if there is a computation such that:

All input symbols are consumed

AND

The last state is a final state

At the end of the computation,  
we do not care about the stack contents

# Acceptance

Language:  $L = \{a^n b^n : n \geq 0\}$

The computation generated by the input string  $w = aaabbbb$  is

$\vdash [q_0, aaabbbb, Z]$

$\vdash [q_1, abbb, aaZ]$

$\vdash [q_2, bb, aaZ]$

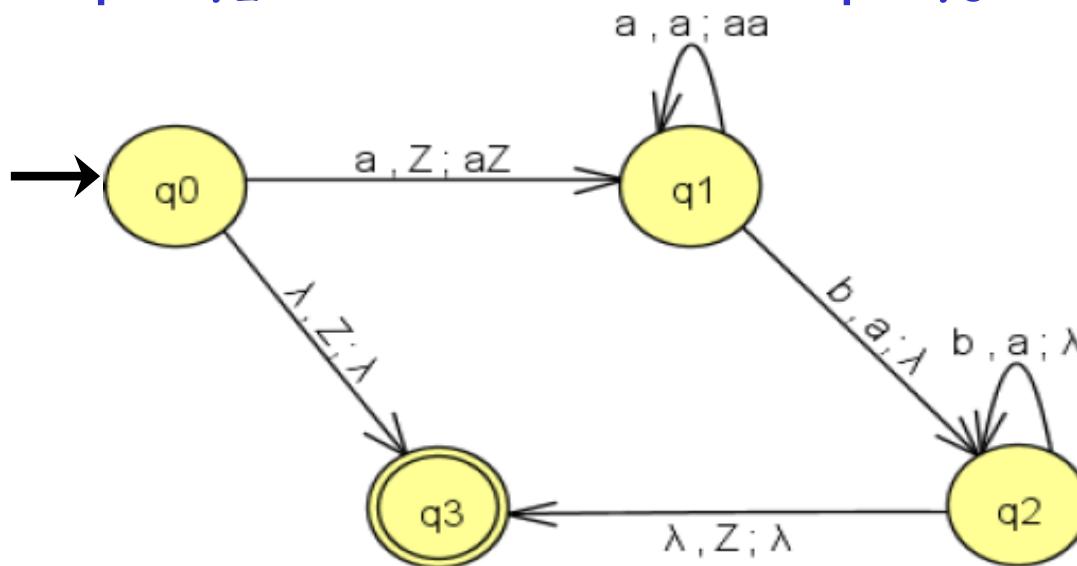
$\vdash [q_2, \varepsilon, Z]$

$\vdash [q_1, aabbbb, aZ]$

$\vdash [q_1, bbb, aaaZ]$

$\vdash [q_2, b, aZ]$

$\vdash [q_3, \varepsilon, \varepsilon]$



state	string	stack
$q_0$	$aaabbbb$	$Z$
$q_1$	$aabbbb$	$aZ$
$q_1$	$abbb$	$aaZ$
$q_1$	$bbb$	$aaaZ$
$q_2$	$bb$	$aaZ$
$q_2$	$b$	$aZ$
$q_2$	$\lambda$	$Z$
$q_3$	$\lambda$	$\lambda$

# Rejection

A string is rejected  
if in **every computation** with the string:

The input cannot be consumed

**OR**

The input is consumed and the last state is  
not a final state

**OR**

The stack head moves below the bottom of  
the stack.

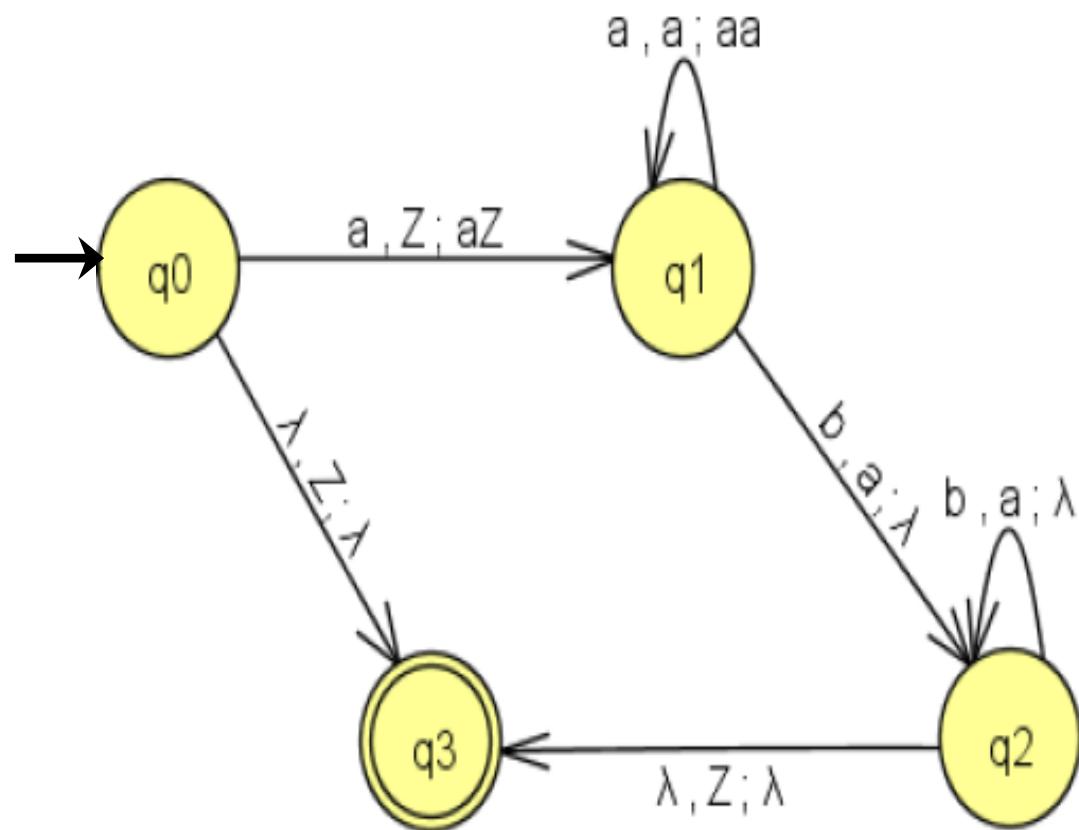
# Rejection

Language:  $L = \{a^n b^n : n \geq 0\}$

The computation generated by the input string  $w = \textcolor{green}{aabbbb}$  is

- $[q_0, \textcolor{green}{aabbbb}, Z] \quad \vdash [q_1, abbb, aZ]$
- $\vdash [q_1, bbb, aaZ] \quad \vdash [q_2, bb, aZ]$
- $\vdash [q_2, b, Z]$

Reject  $w = \text{aabbbb}$



# Rejection

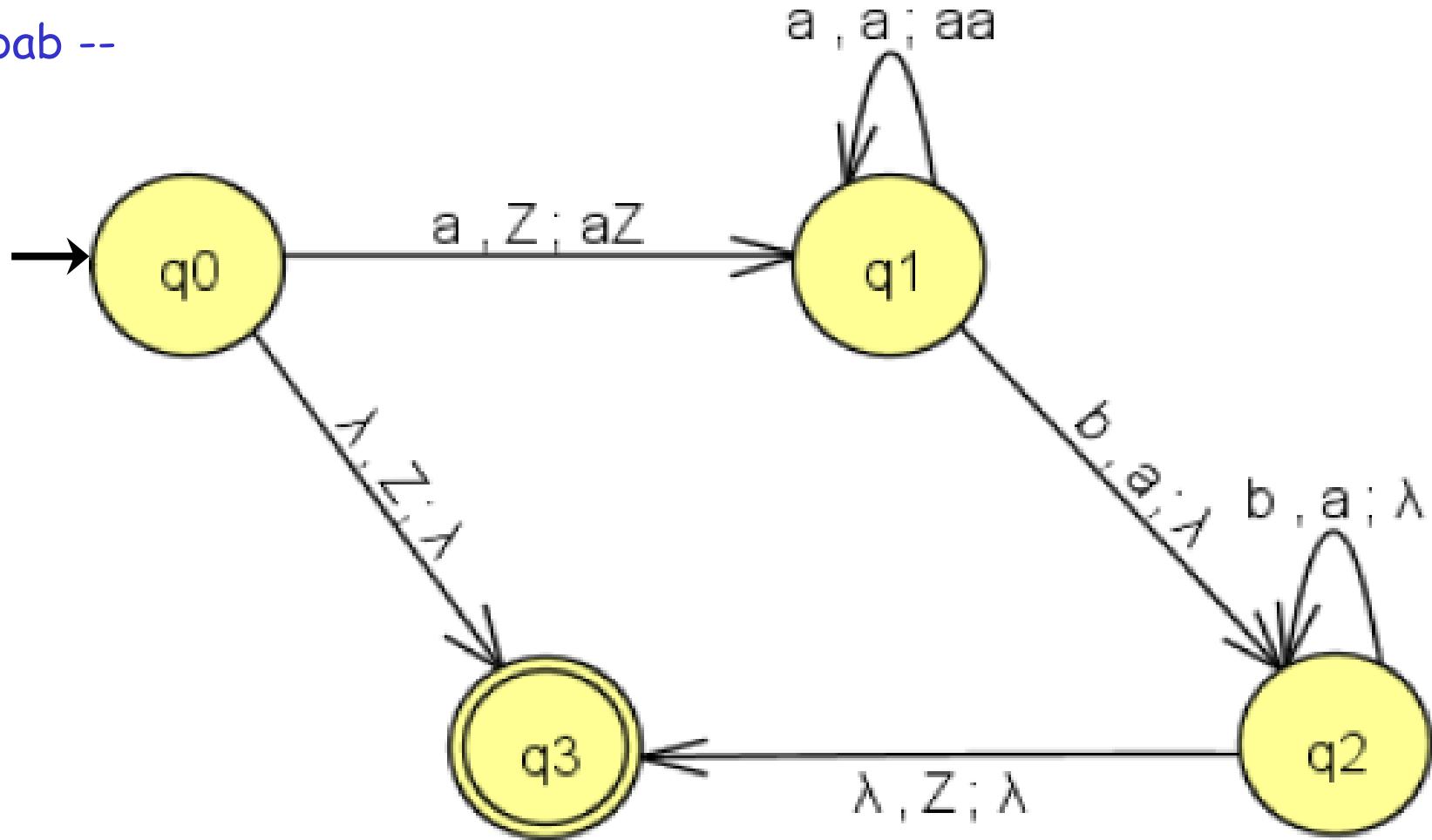
$w = aaaaabbbbb$  -Accept

$w = aabbaa$  --

$w = bbaa$  -

$w = abbaa$  --

$w = abab$  --



# Language of NPDA

The language accepted by NPDA  $M$  is

$$L(M) = \{w \in \Sigma^* : [q_0, w, z] \vdash^* [p, \varepsilon, u] \text{ with } p \in F, u \in \Gamma^*\}$$

# Construct NPDA

Language:  $L = \{wcw^R : w \in \{a, b\}^+ \}$

$b, a \rightarrow ba$

$b, b \rightarrow bb$

$a, b \rightarrow ab$

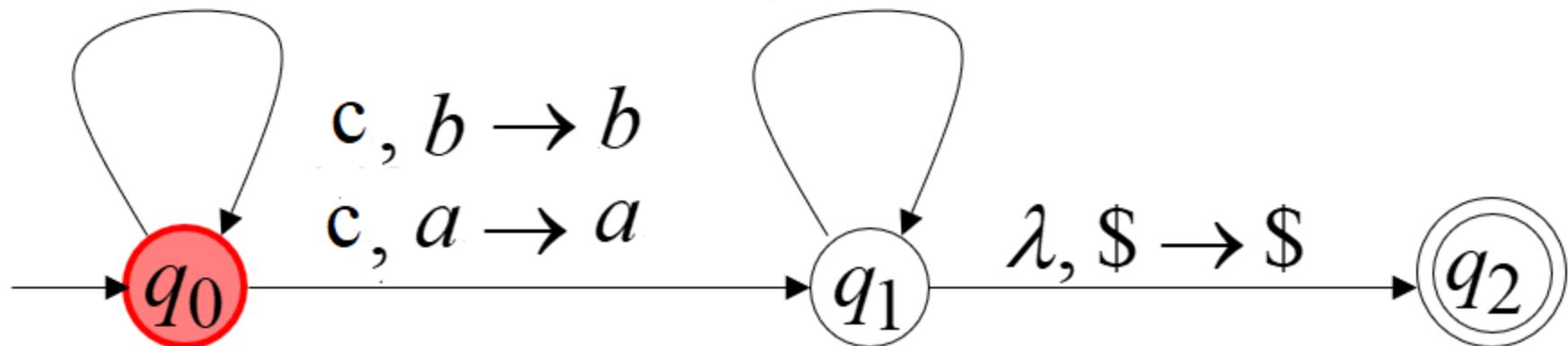
$a, a \rightarrow aa$

$a, \$ \rightarrow a\$$

$a, a \rightarrow \lambda$

$b, \$ \rightarrow b\$$

$b, b \rightarrow \lambda$



# Construct NPDA

Language:  $L = \{ww^R : w \in \{a, b\}^+ \}$

$b, a \rightarrow ba$

$b, b \rightarrow bb$

$a, b \rightarrow ab$

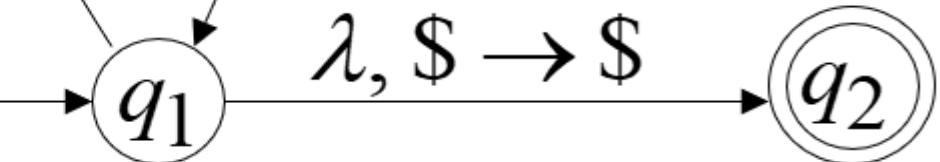
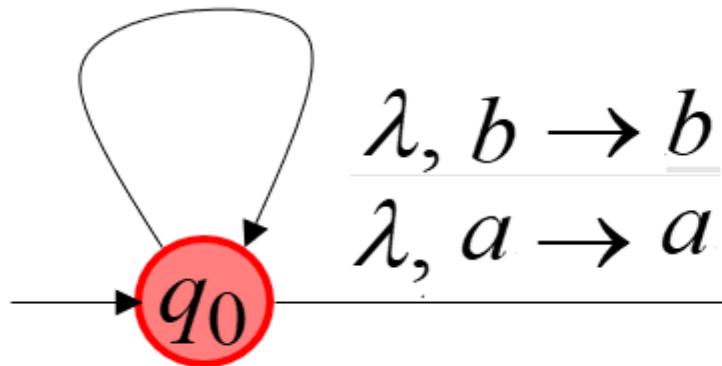
$a, a \rightarrow aa$

$a, \$ \rightarrow a\$$

$a, a \rightarrow \lambda$

$b, \$ \rightarrow b\$$

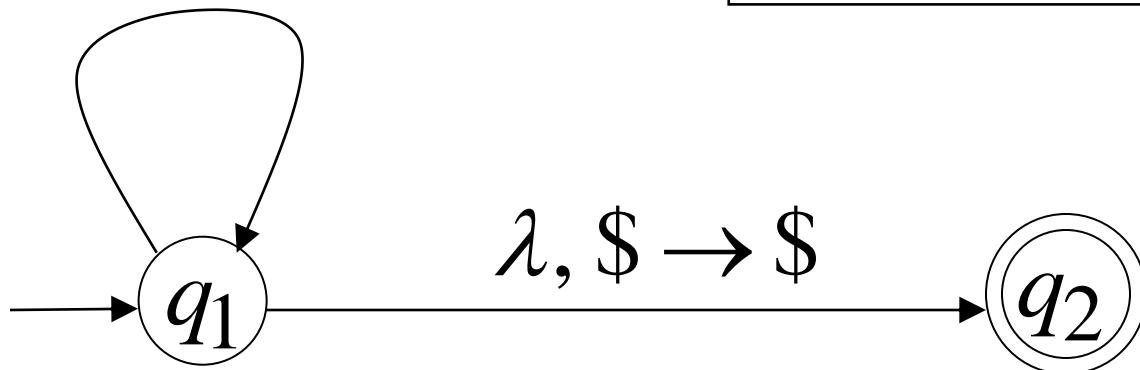
$b, b \rightarrow \lambda$



# Another NPDA example

$L(M) = \{w : n_a(w) = n_b(w), w \in \{a,b\}^*\}$   
 $= \{e, abba, aabb, abbbbaa, \}$

$a, \$ \rightarrow 0\$$      $b, \$ \rightarrow 1\$$   
 $a, 0 \rightarrow 00$      $b, 1 \rightarrow 11$   
 $a, 1 \rightarrow \lambda$      $b, 0 \rightarrow \lambda$



If a and stack(\$, 0)  $\rightarrow$  push 0 and state q1  
If a and stack(1)  $\rightarrow$  pop 1 and state q1

If b and stack(\$, 1)  $\rightarrow$  push 1 and state q1  
If b and stack(0)  $\rightarrow$  pop 0 and state q1

If the input symbol is  $\lambda$  and stack (\$)  $\rightarrow$  change state to q2

# Execution Example: Time 0

Input

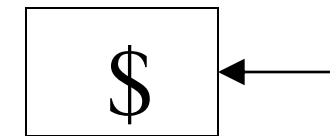
$a$	$b$	$b$	$a$	$a$	$b$
-----	-----	-----	-----	-----	-----



$a, \$ \rightarrow 0\$$        $b, \$ \rightarrow 1\$$

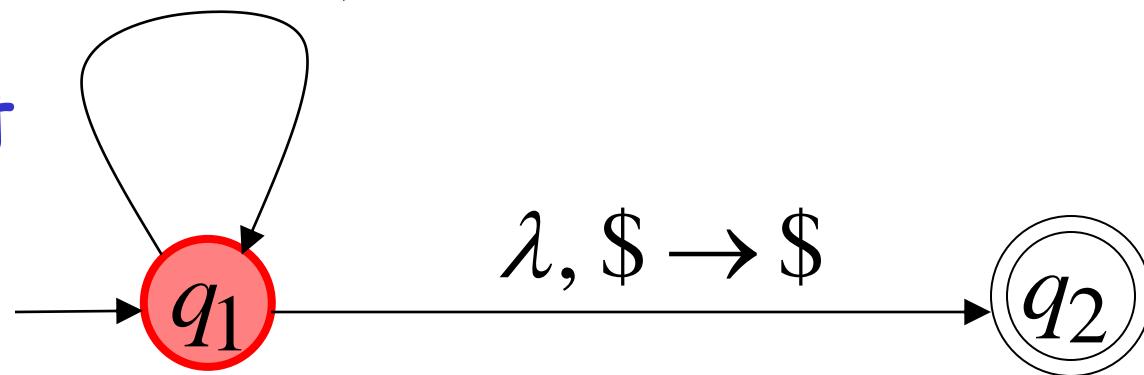
$a, 0 \rightarrow 00$        $b, 1 \rightarrow 11$

$a, 1 \rightarrow \lambda$        $b, 0 \rightarrow \lambda$



Stack

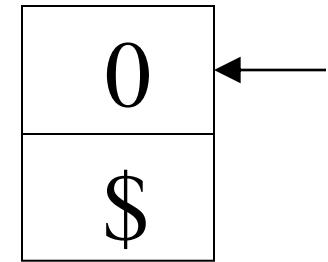
current  
state



Time 1

Input

$a$	$b$	$b$	$a$	$a$	$b$
-----	-----	-----	-----	-----	-----

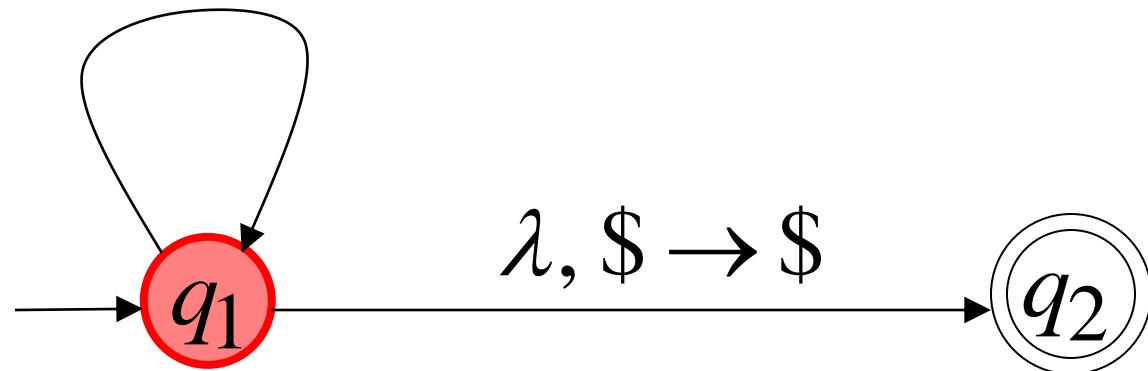


Stack

$a, \$ \rightarrow 0\$$        $b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$        $b, 1 \rightarrow 11$

$a, 1 \rightarrow \lambda$        $b, 0 \rightarrow \lambda$



Time 3

Input

$a$	$b$	$b$	$b$	$a$	$a$
-----	-----	-----	-----	-----	-----



$a, \$ \rightarrow 0\$$

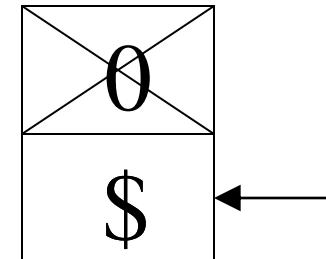
$b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$

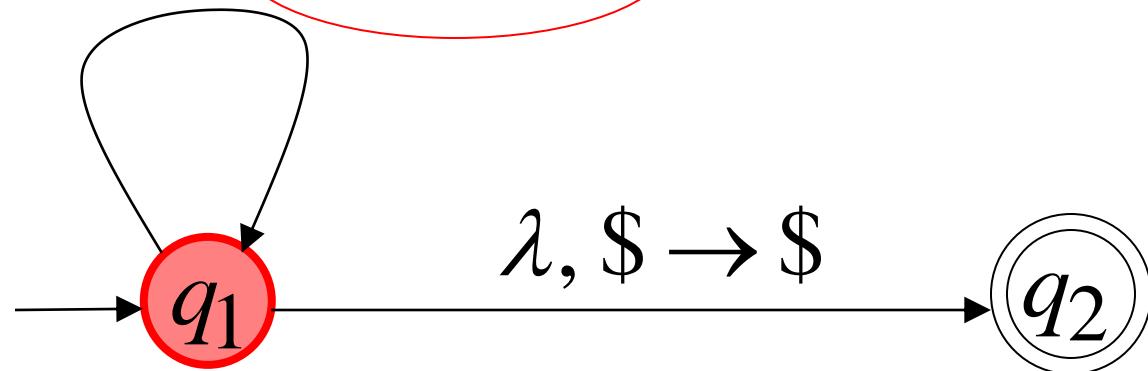
$b, 1 \rightarrow 11$

$a, 1 \rightarrow \lambda$

$b, 0 \rightarrow \lambda$



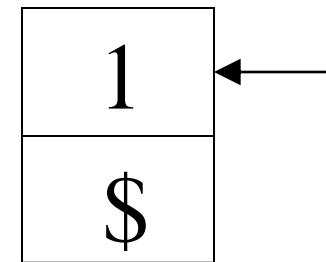
Stack



Time 4

Input

$a$	$b$	$b$	$b$	$a$	$a$
-----	-----	-----	-----	-----	-----



Stack

$a, \$ \rightarrow 0\$$

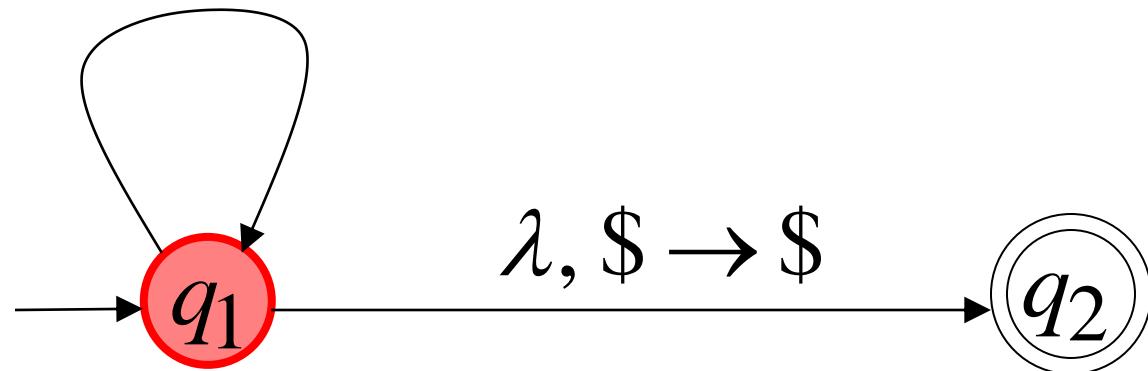
$b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$

$b, 1 \rightarrow 11$

$a, 1 \rightarrow \lambda$

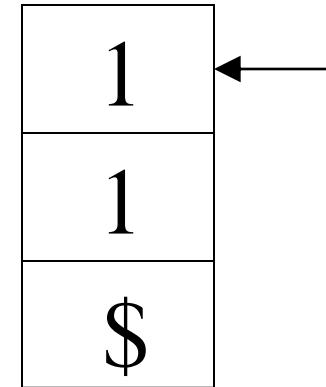
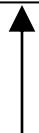
$b, 0 \rightarrow \lambda$



Time 5

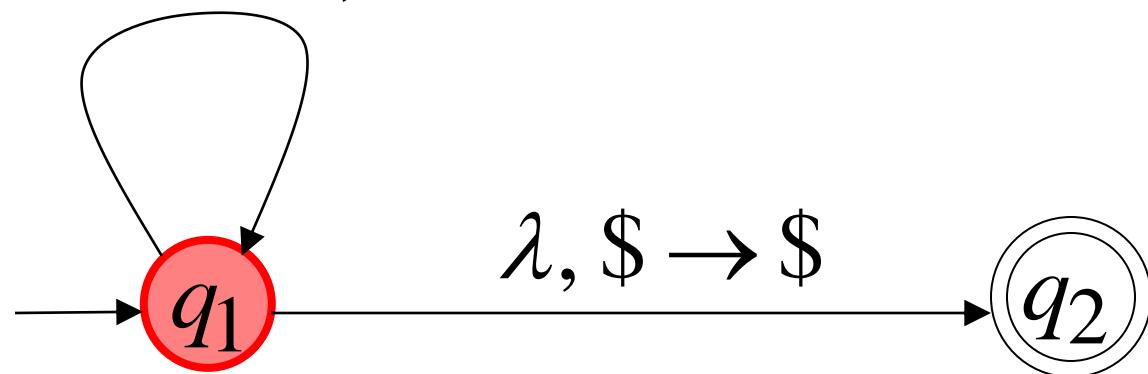
Input

$a$	$b$	$b$	$b$	$a$	$a$
-----	-----	-----	-----	-----	-----



Stack

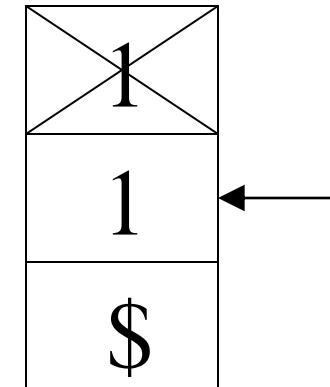
$$\begin{array}{ll} a, \$ \rightarrow 0\$ & b, \$ \rightarrow 1\$ \\ a, 0 \rightarrow 00 & \text{b, } 1 \rightarrow 11 \\ a, 1 \rightarrow \lambda & b, 0 \rightarrow \lambda \end{array}$$



Time 6

Input

$a$	$b$	$b$	$b$	$a$	$a$
-----	-----	-----	-----	-----	-----

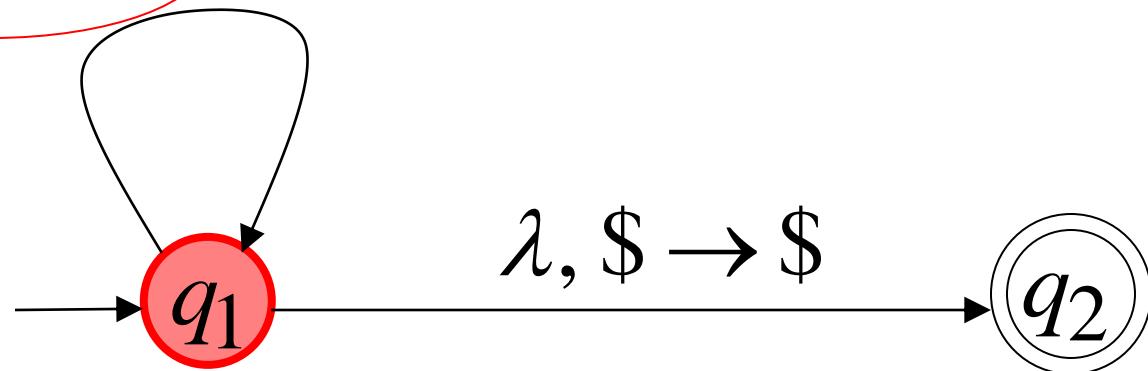


Stack

$$a, \$ \rightarrow 0\$ \quad b, \$ \rightarrow 1\$$$

$$a, 0 \rightarrow 00 \quad b, 1 \rightarrow 11$$

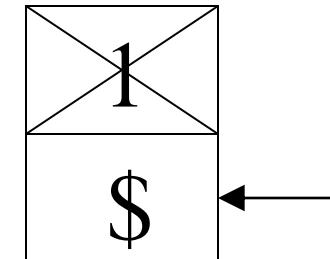
$$a, 1 \rightarrow \lambda \quad b, 0 \rightarrow \lambda$$



Time 7

Input

$a$	$b$	$b$	$b$	$a$	$a$
-----	-----	-----	-----	-----	-----

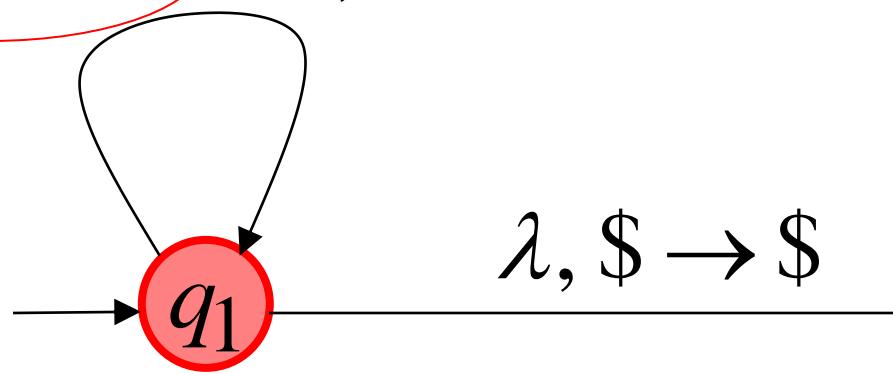


Stack

$a, \$ \rightarrow 0\$$        $b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$        $b, 1 \rightarrow 11$

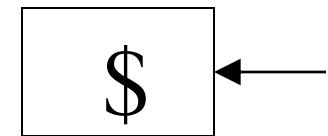
$a, 1 \rightarrow \lambda$        $b, 0 \rightarrow \lambda$



Time 8

Input

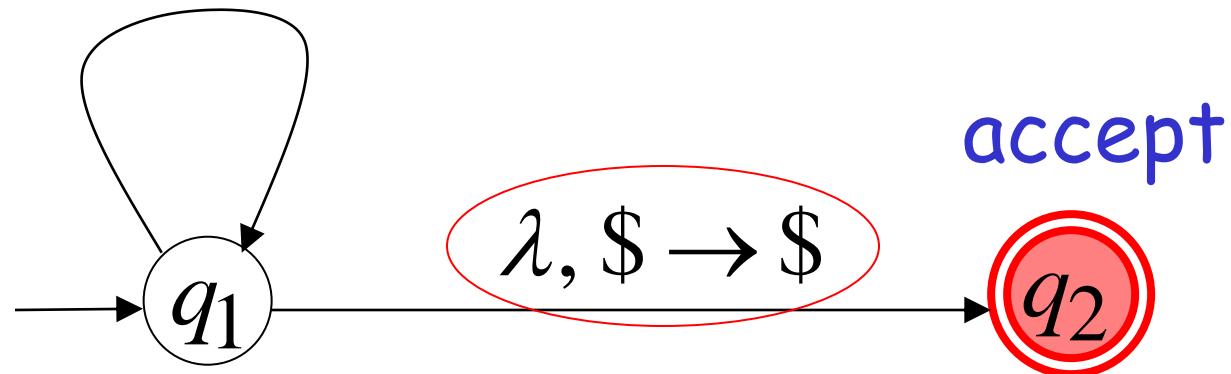
$a$	$b$	$b$	$b$	$a$	$a$
-----	-----	-----	-----	-----	-----



$a, \$ \rightarrow 0\$$        $b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$        $b, 1 \rightarrow 11$

$a, 1 \rightarrow \lambda$        $b, 0 \rightarrow \lambda$



# Construct NPDAs

$$L_1 = L(aaa^*b)$$

$$L_2 = L(aab^*aba^*)$$

$$L_3 = L_1 \cup L_2$$

$$L_4 = \{a^n b^{2n} : n \geq 0\}$$

$$L_5 = \{a^n b^m c^{n+m} : m, n \geq 0\}$$

$$L_6 = \{a^n b^{m+n} c^m : m, n \geq 1\}$$

$$L_7 = \{a^3 b^m c^m : m \geq 0\}$$

$$L_8 = \{a^n b^m : 2n \leq m \leq 3n, \}$$

$$L_9 = \{w : n_a(w) = n_b(w) + 1, w \in \{a,b\}^*\}$$

$$L_{10} = \{w : n_a(w) = 2n_b(w), w \in \{a,b\}^*\}$$

$$L_{11} = \{w : n_a(w) + n_b(w) = n_c(w), w \in \{a, b, c\}^*\}$$

$$L_{12} = \{ab(ab)^n b(ba)^n : n \geq 0\}$$

# Deterministic Pushdown Automata (DPDA)

$$\delta : Q \times (\Sigma \cup \{e\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$$

A PDA is **deterministic** if its transition function satisfies **both** of the following properties

For all  $q \in Q$ ,  $a \in (S \cup \{\varepsilon\})$ , and  $X \in G$ ,

the set  $\delta(q, a, X)$  has **at most** one element

i.e. there is only one move for any input and stack combination

For all  $q \in Q$  and  $X \in G$ ,

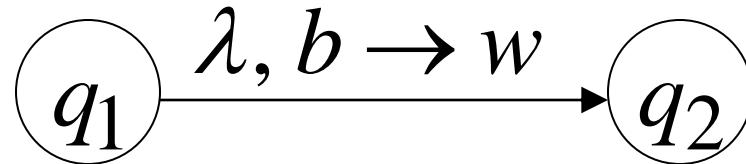
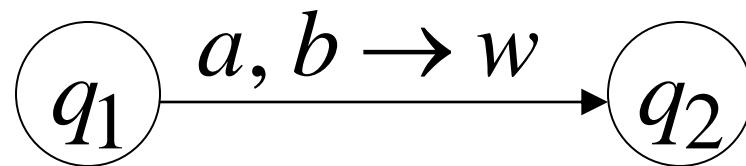
if  $\delta(q, e, X) \neq \{\}$ , then  $\delta(q, a, X) = \{\} \forall a \in S$

i.e. if there is an e-transition from state  $q$  with  $X$  as stack symbol, **then**

there cannot exist another move with stack symbol  $X$  from the *same state  $q$  for any other input symbol.*

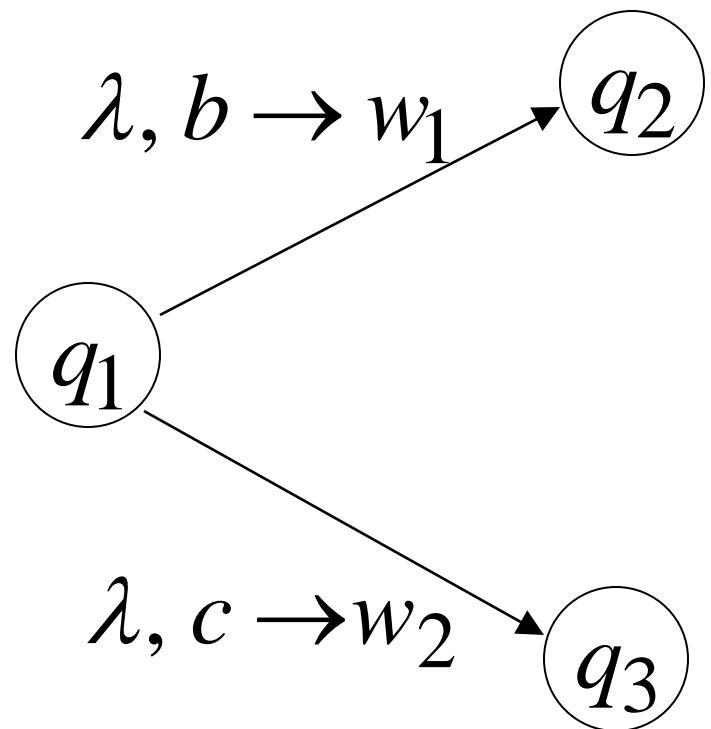
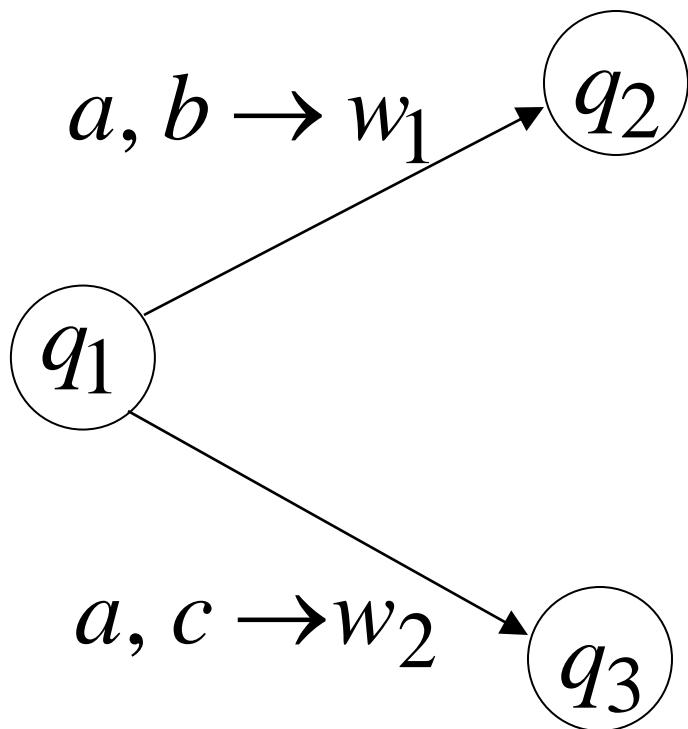
# Deterministic PDA: DPDA

Allowed transitions



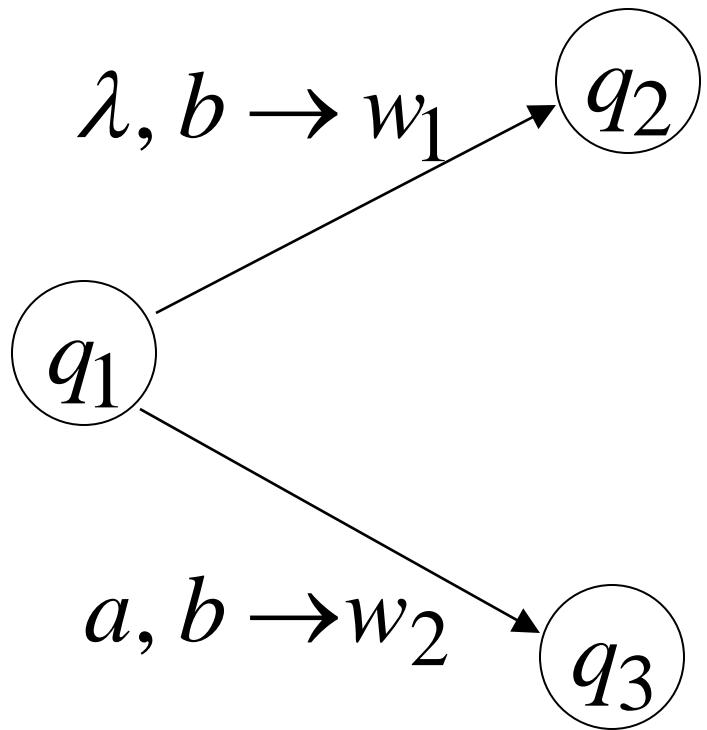
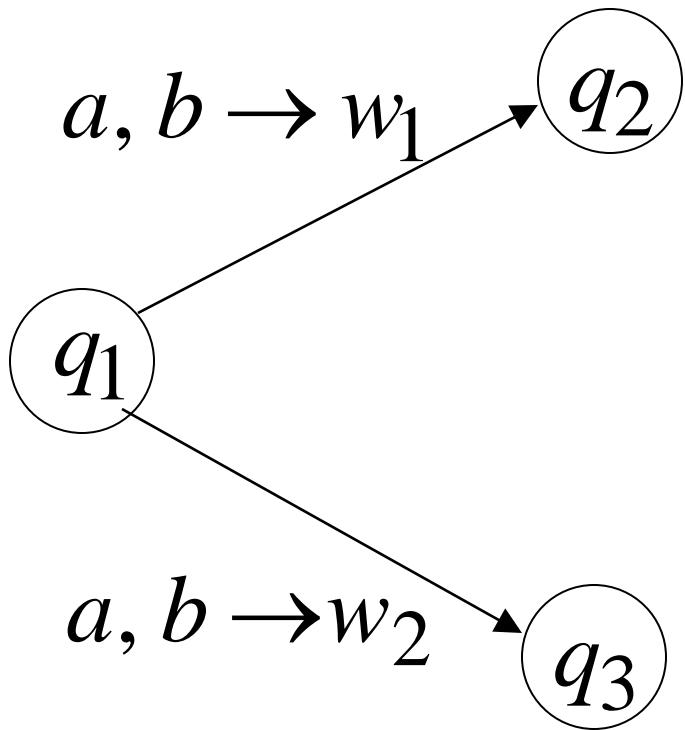
(deterministic choices)

Allowed transitions:



(deterministic choices)

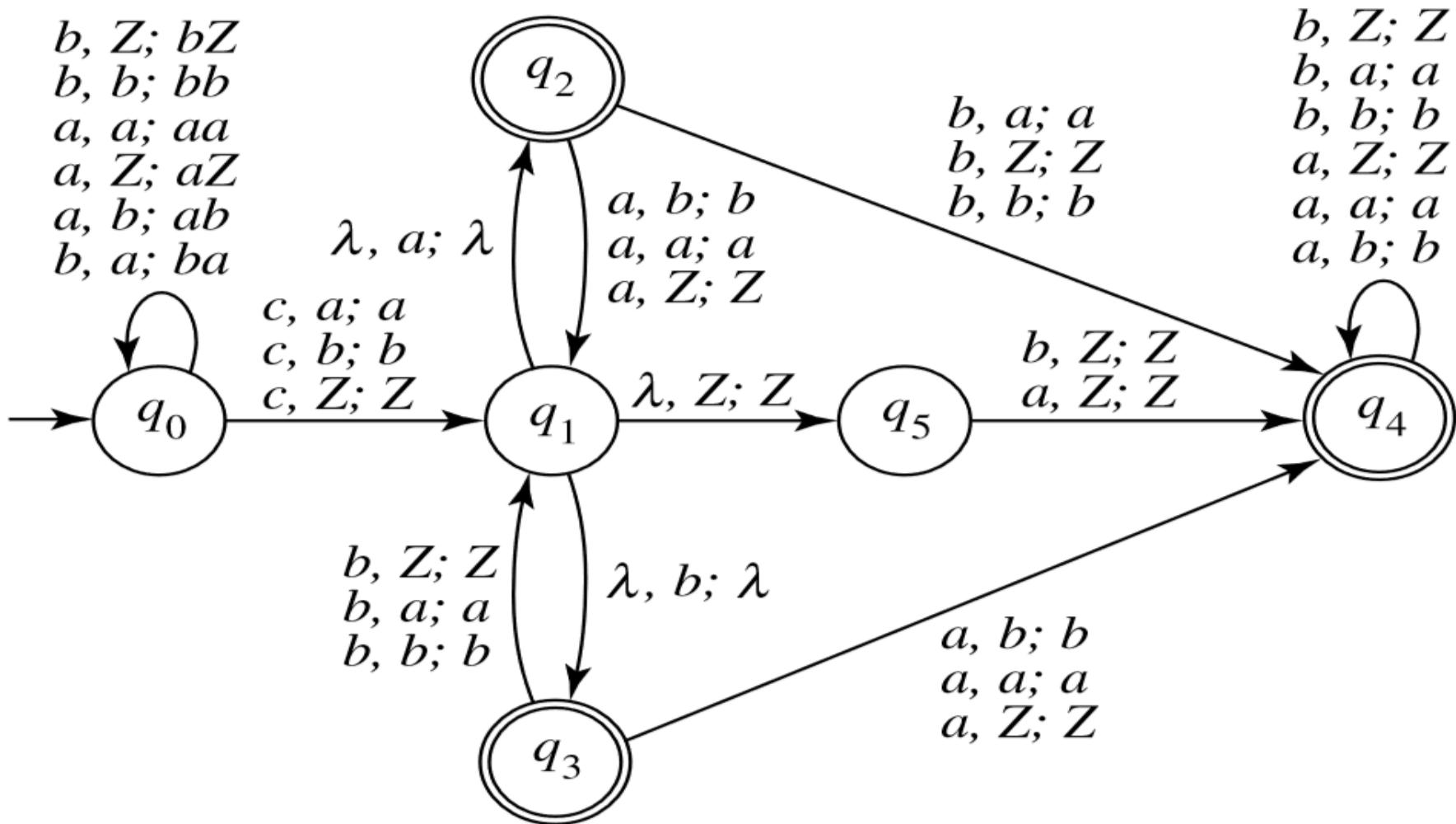
Not allowed in DPDA:



(non deterministic choices)

# DPDA ?

Language:  $L = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^*, w_1 \neq w_2\}$



# Non-deterministic PDA

## Not DPDA

- 7.3.5 show example 7.4 is a npda but that  $L = \{w \in \{a, b\}^*: n_a(w) = n_b(w)\}$  is a deterministic context-free language.
- Machine is npda

$$\delta(q_0, a, 0) = \{(q_0, 00)\}$$

$$\delta(q_0, b, 1) = \{(q_0, 11)\}$$

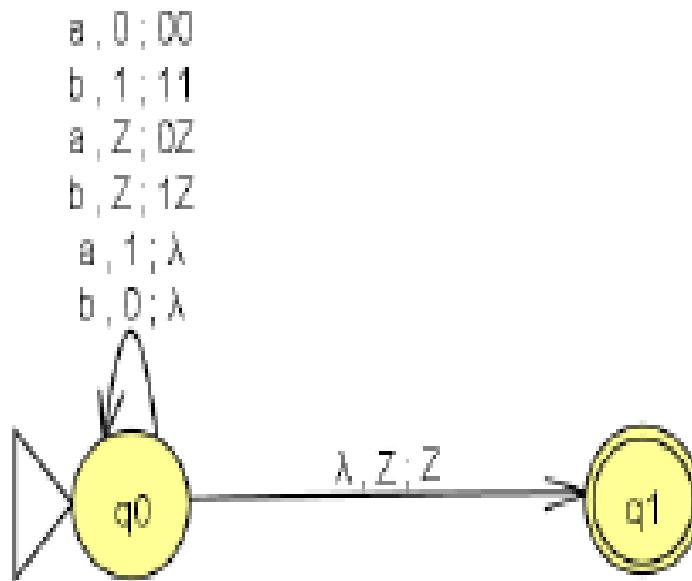
$$\delta(q_0, a, Z) = \{(q_0, 0Z)\}$$

$$\delta(q_0, b, Z) = \{(q_0, 1Z)\}$$

$$\delta(q_0, a, 1) = \{(q_0, \lambda)\}$$

$$\delta(q_0, b, 0) = \{(q_0, \lambda)\}$$

$$\delta(q_0, \lambda, Z) = \{(q_1, Z)\}$$



This is a npda because these transitions violate the 2nd rule associated with dpda,  
 $\delta(q_0, \lambda, Z) \neq \emptyset \Rightarrow \forall c \in \Sigma \quad \delta(q_0, c, Z) = \emptyset$

# Deterministic PDA

Some context-free languages which are initially described in a nondeterministic way using NPDA **can also be described** in a deterministic way using DPDA.

Some context-free languages are inherently nondeterministic, e.g.,  $L = \{w \in (a|b)^*: w = w^R\}$  **cannot be accepted by any dpda**.

Deterministic PDA (DPDA) can only represent a subset of CFL, e.g.,  $L = \{ww^R \mid w \in (a|b)^*\}$  **cannot be represented by DPDA**

A key point in all this is that the **deterministic pushdown automata is not equivalent to nondeterministic pushdown automata**.

Unless otherwise stated, we assume that a PDA is nondeterministic

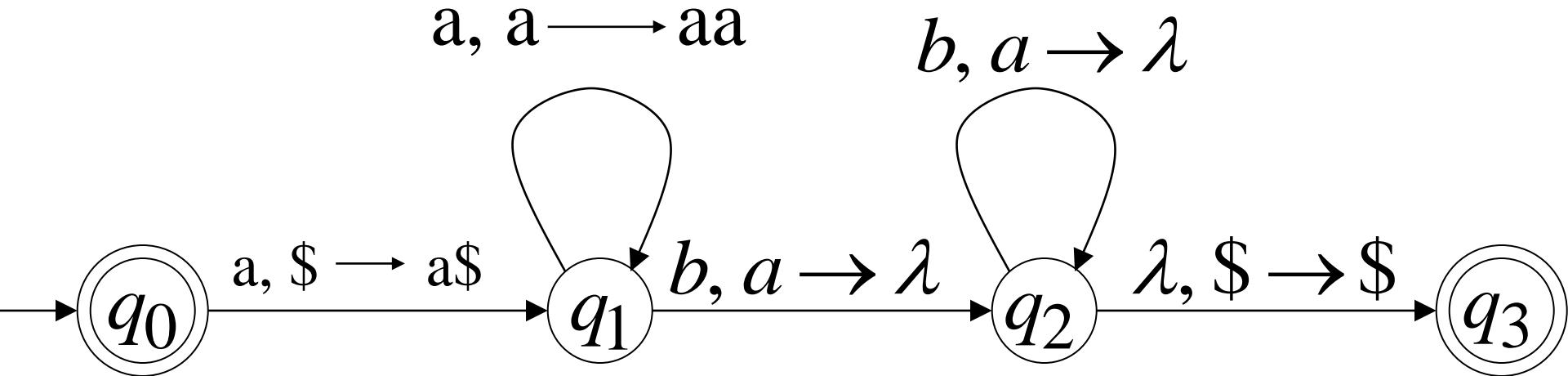
# DPDA example

$$L(M) = \{a^n b^n : n \geq 0\}$$

The language  $L(M) = \{a^n b^n : n \geq 0\}$  is deterministic context-free

If st=q0; sym=a; sk=\$ → push a and change state to q1  
If st=q1; sym=a; sk=a → push a and stay in the state q1  
If st=q1; sym=b; sk=a → pop a and change state to q2  
If st=q2; sym=b; sk=a → pop a and stay in state q2  
If st=q2; sym=λ; sk=\$ → Change state to q3

bbaaaaabbb



## Definition:

A language  $L$  is **deterministic context-free** if there exists some DPDA that accepts it

# Turing Machines

# The Language Hierarchy

$a^n b^n c^n$  ?

$ww$  ?

Some languages  
are not  
Context-Free.

Context-Free Languages

$a^n b^n$

$ww^R$

Regular Languages

$a^*$

$a^* b^*$

# Languages accepted by Turing Machines

$a^n b^n c^n$

$ww$

## Context-Free Languages

$a^n b^n$

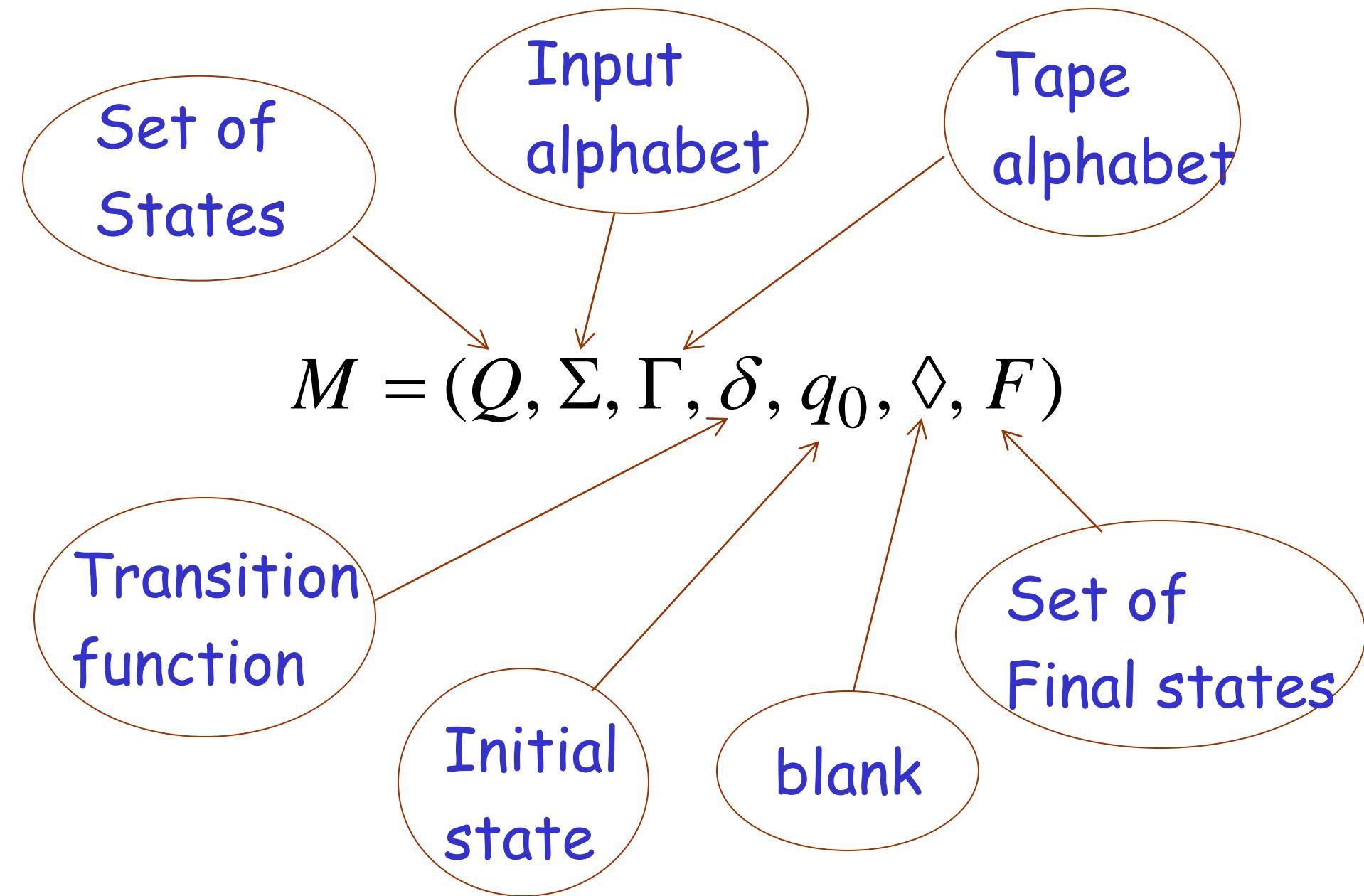
$ww^R$

## Regular Languages

$a^*$

$a^* b^*$

# Turing Machine:



# Turing Machine

A Turing Machine is represented by 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$$

$Q$  is a finite set of states

$$\Sigma \subseteq \Gamma - \{\diamond\}$$

$\Sigma$  is the input alphabet, where  $\diamond \notin \Sigma$

$\Gamma$  is the tape alphabet, a superset of  $\Sigma$ ;  $\diamond \in \Gamma$

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$  is the transition function

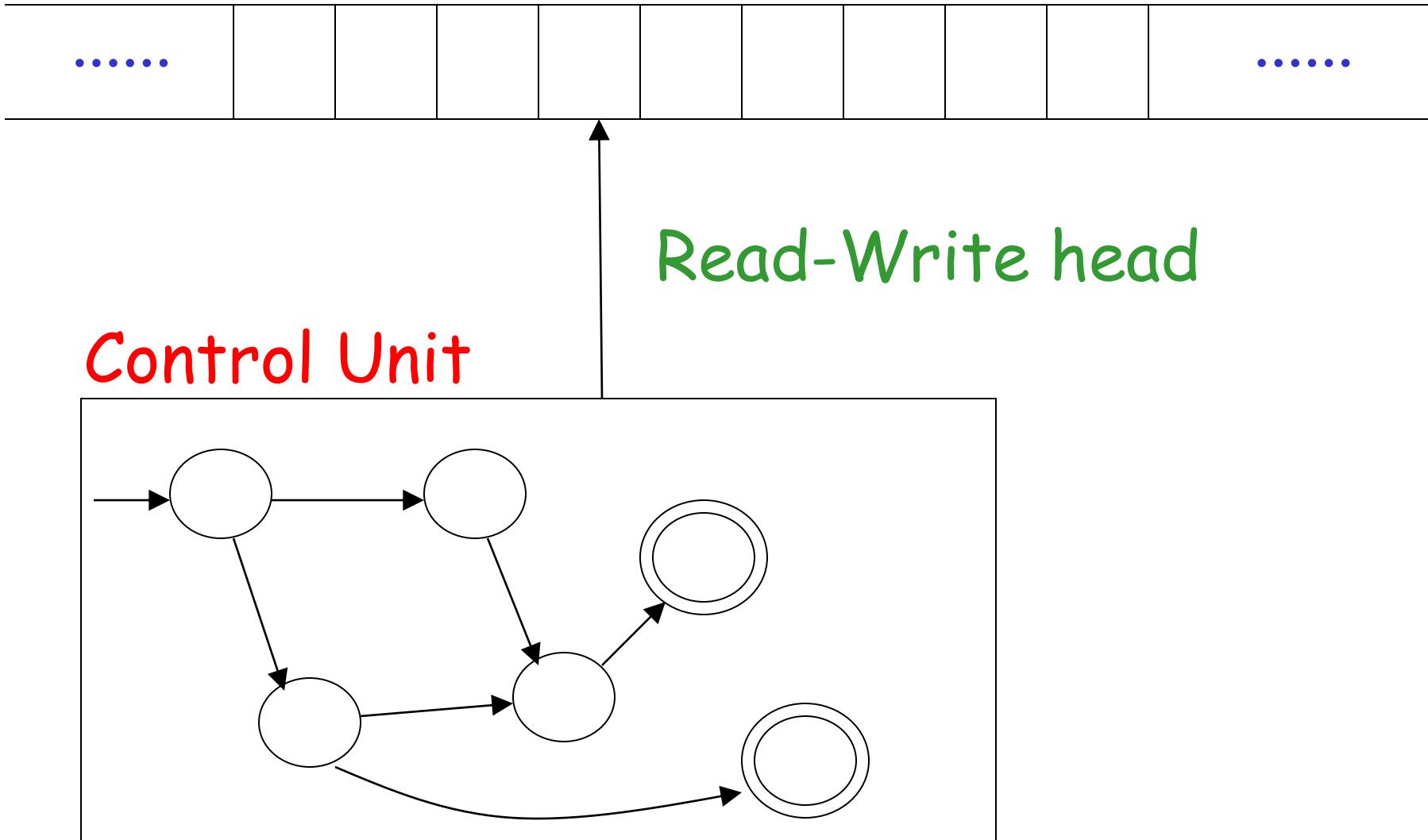
$q_0 \in Q$  is the start state

$\diamond \in \Gamma$  is the blank symbol

$F \subseteq Q$  is the set of final states

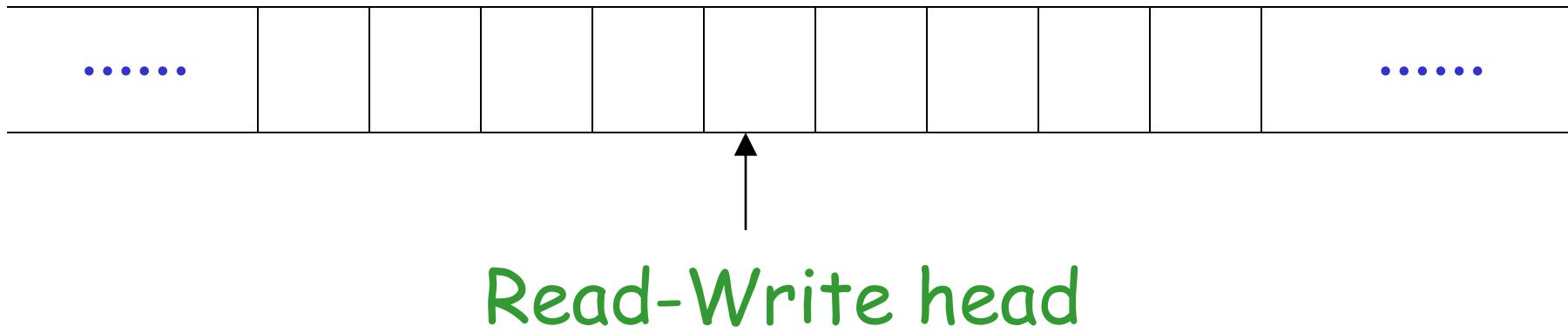
# A Turing Machine

Tape: It is divided into cells.



# The Tape

No boundaries -- infinite length



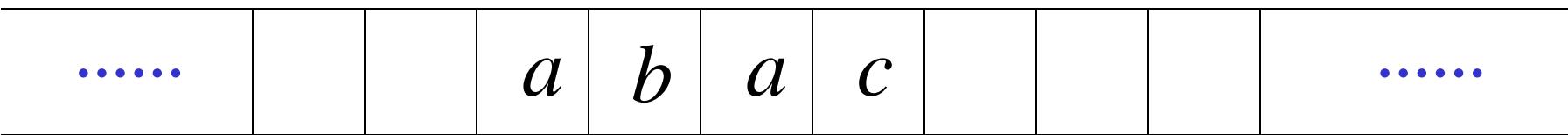
The head moves Left or Right

The head at each time step:

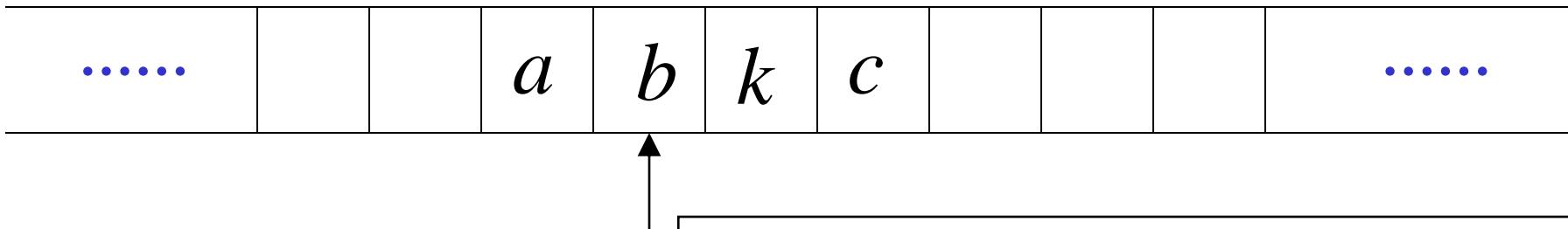
1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

# Example:

Time 0



Time 1



1. Reads *a*

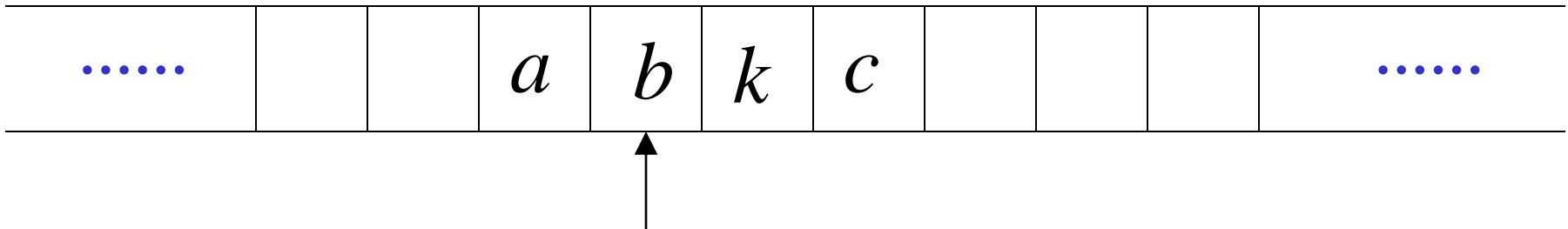
2. Writes *k*

3. Moves Left

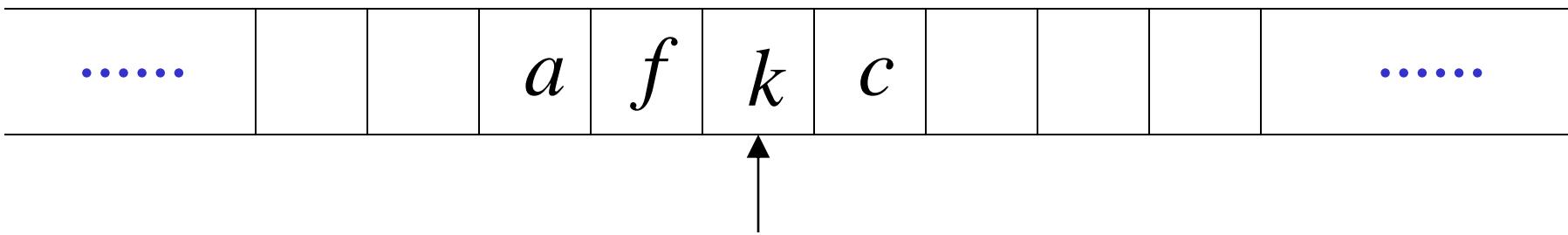
The head at each time step:

- Reads a symbol
- Writes (replaces) the symbol
- Moves Left or Right

Time 1

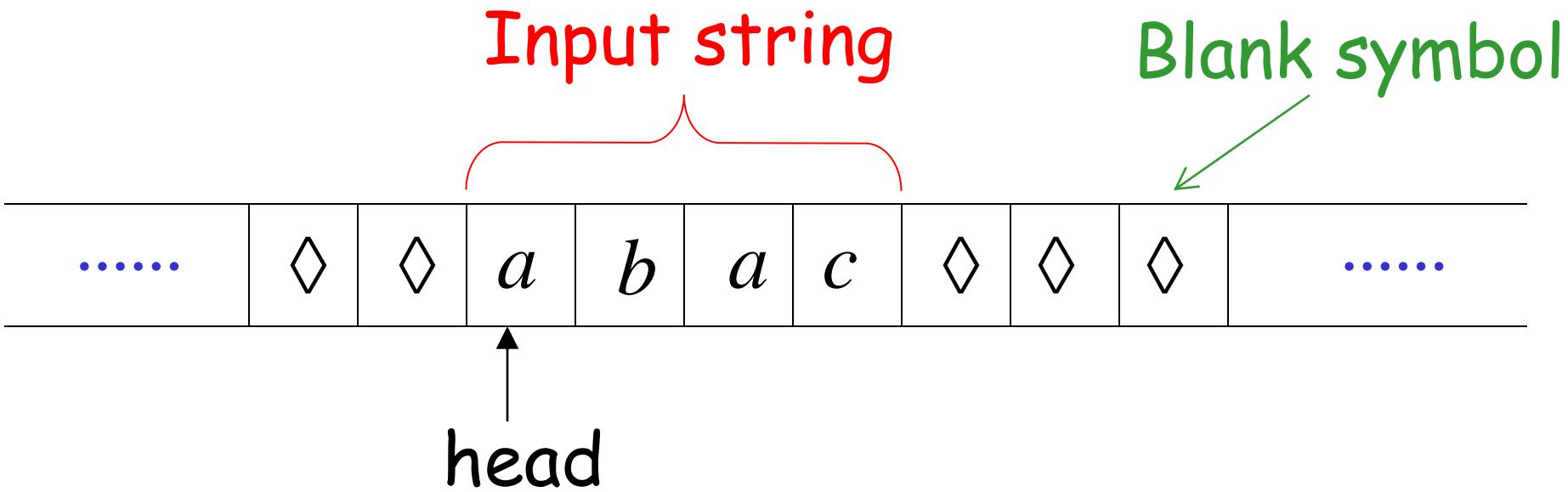


Time 2

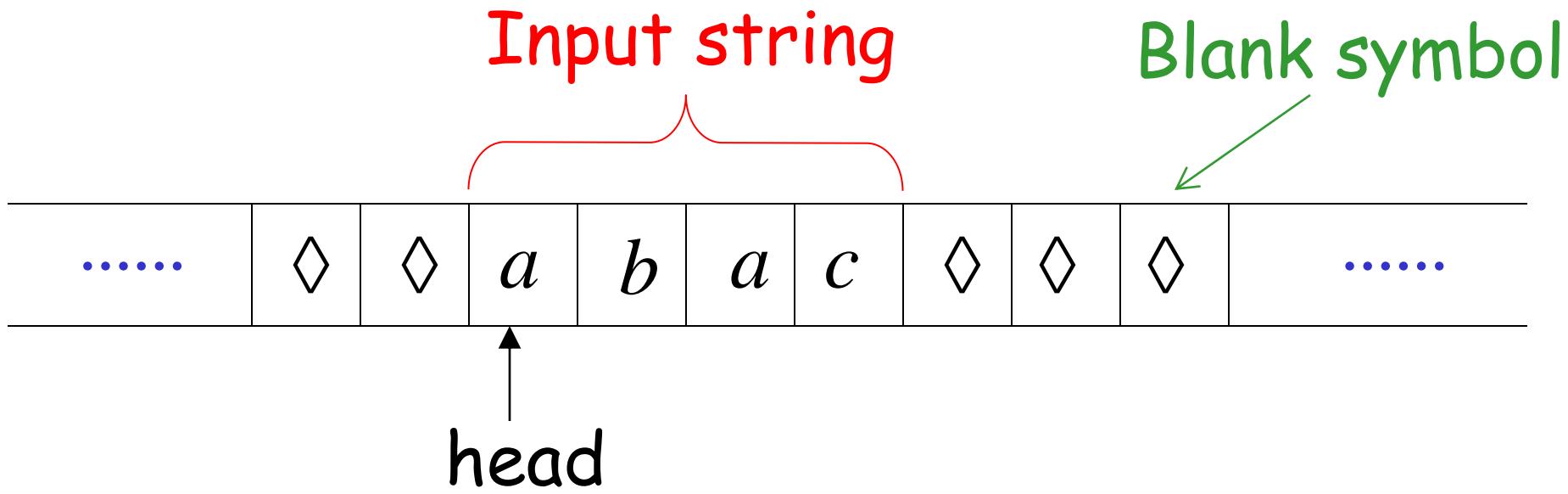


1. Reads  $b$
2. Writes  $f$
3. Moves Right

# The Input String

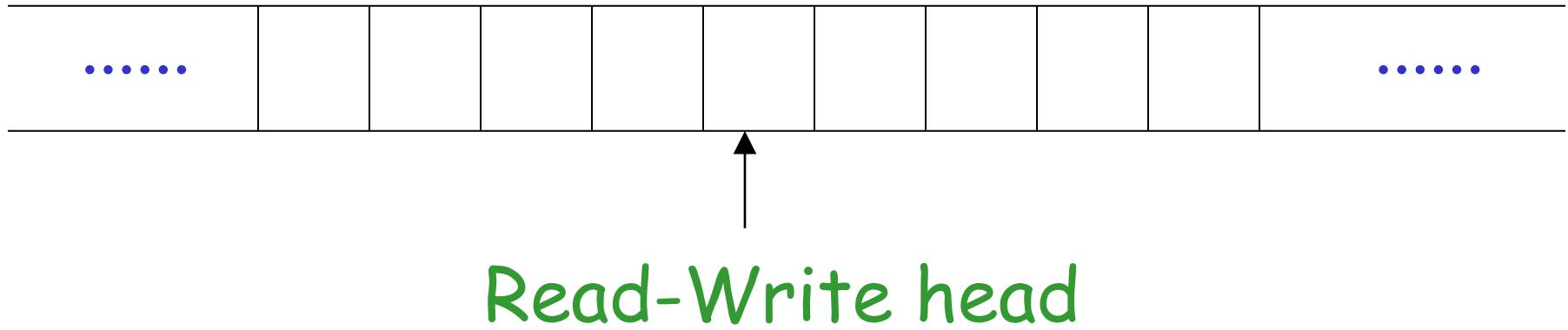


Head starts at the leftmost position of the input string.



Remark: The input string is never empty

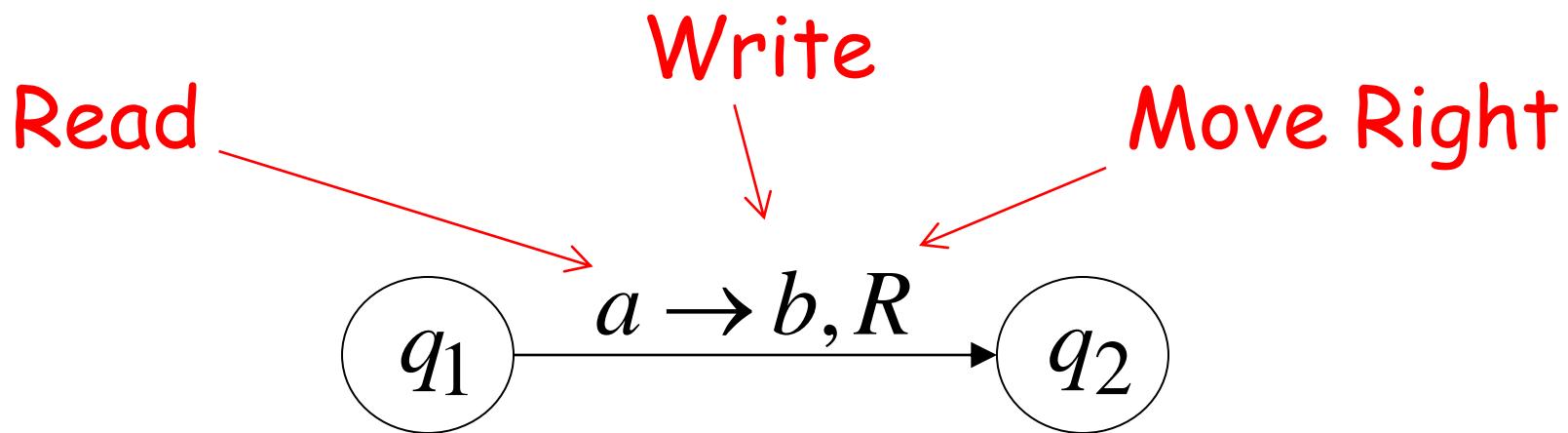
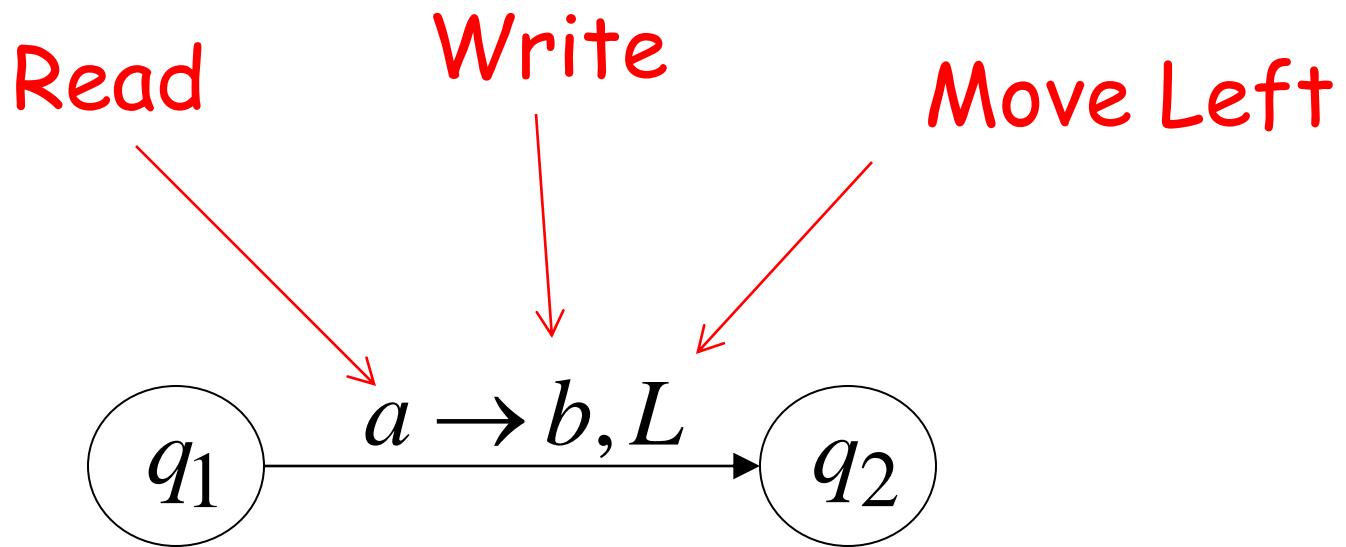
# CONTROL UNIT



At each time step:

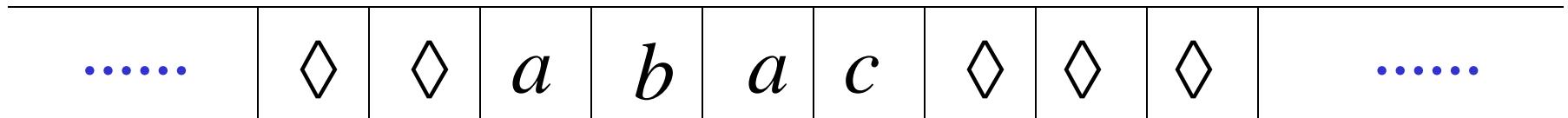
1. Checks the current state
2. Read the current input symbol
3. Decides “what to do next”

# States & Transitions



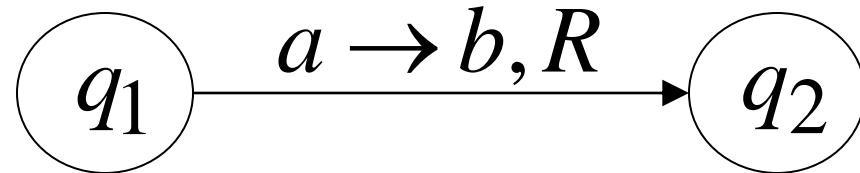
Example:

Time 1

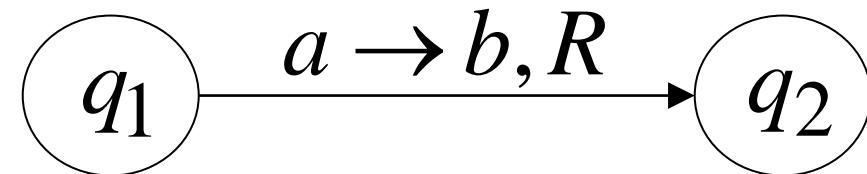
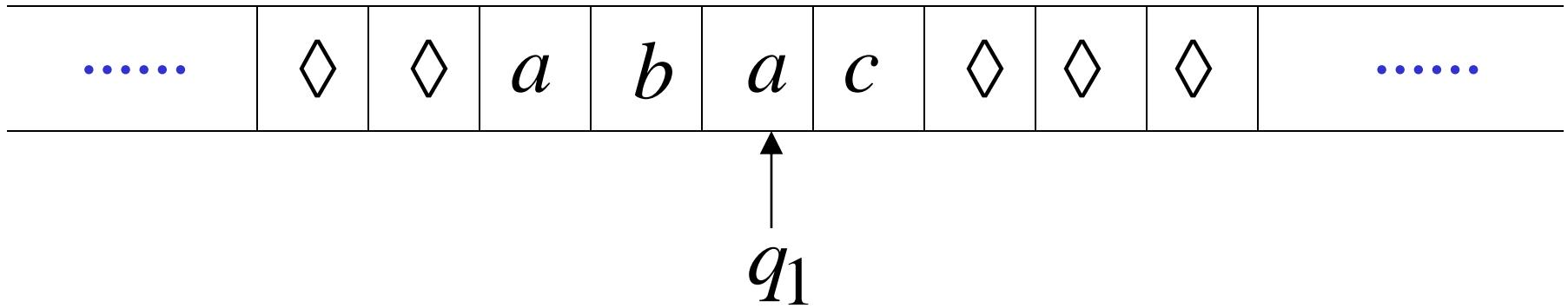


$\uparrow$   
 $q_1$

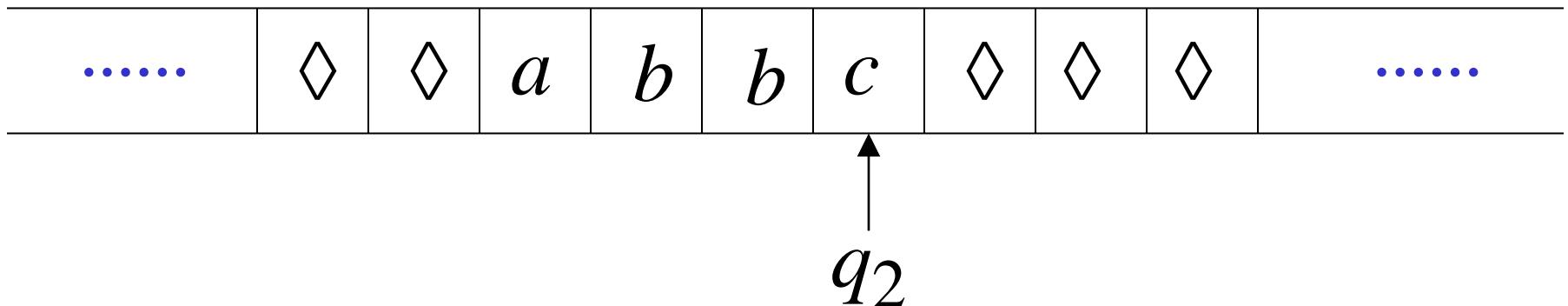
current state



Time 1

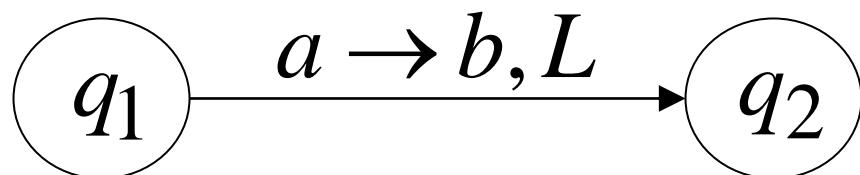
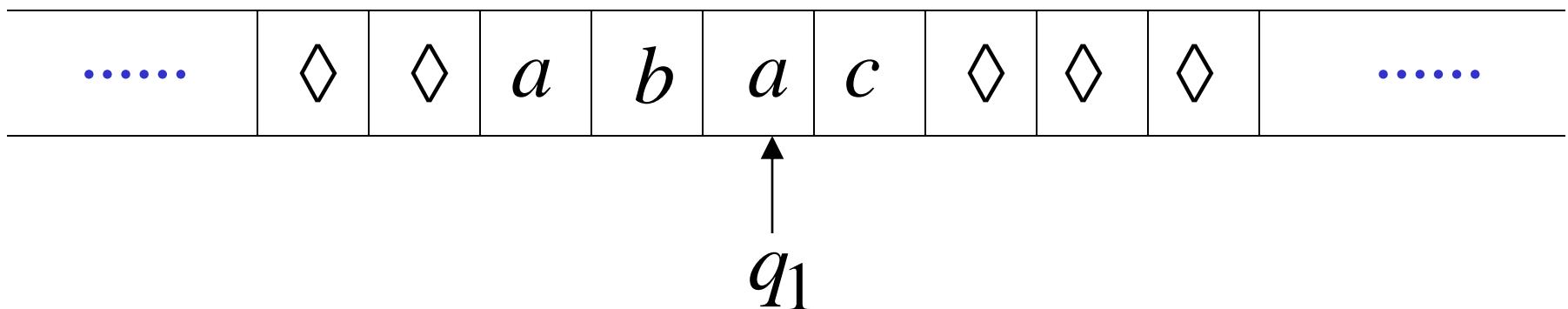


Time 2

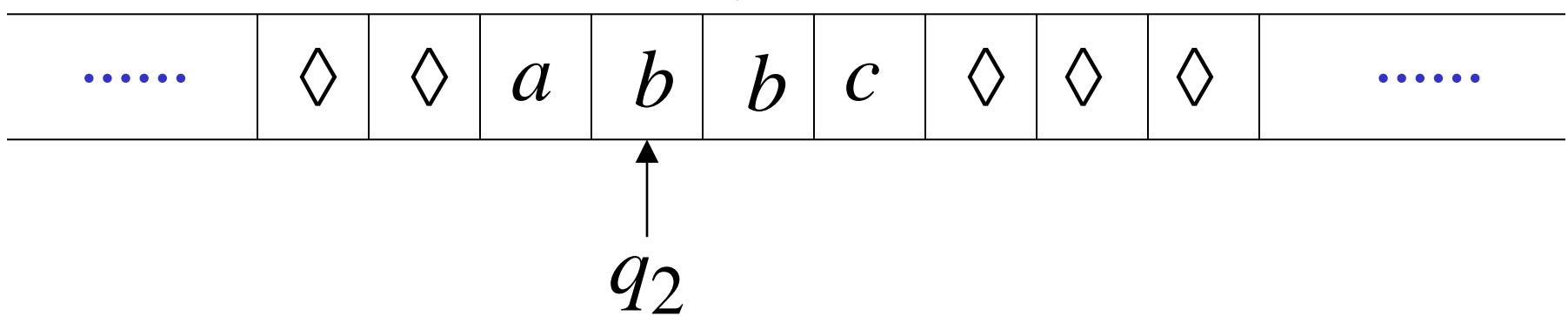


# Example:

Time 1



Time 2



Example:

Time 1

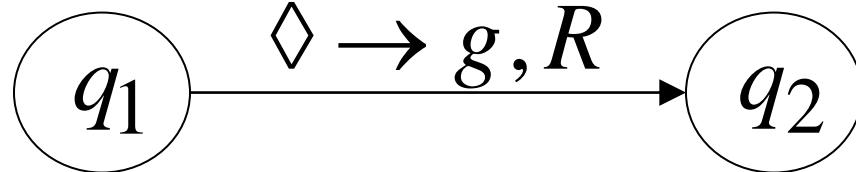
.....	◊	◊	a	b	a	c	◊	◊	◊	.....
-------	---	---	---	---	---	---	---	---	---	-------

↑  
 $q_1$

Time 2

.....	◊	◊	a	b	b	c	g	◊	◊	.....
-------	---	---	---	---	---	---	---	---	---	-------

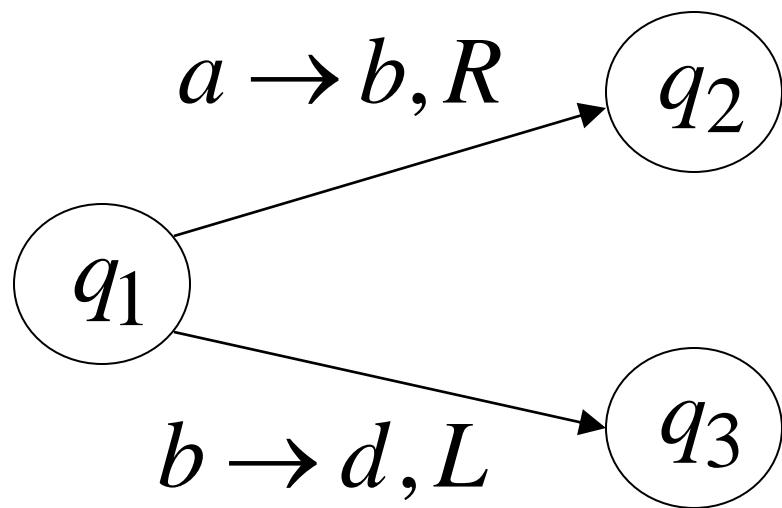
↑  
 $q_2$



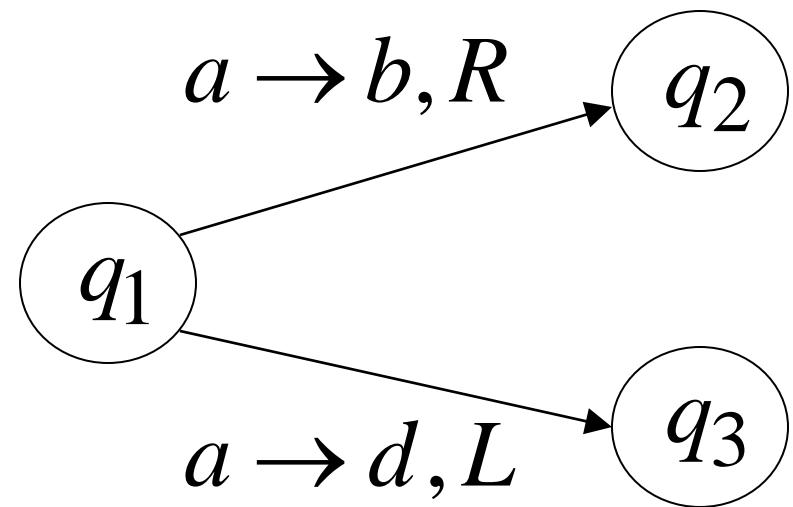
# Determinism

Turing Machines are deterministic

Allowed



Not Allowed

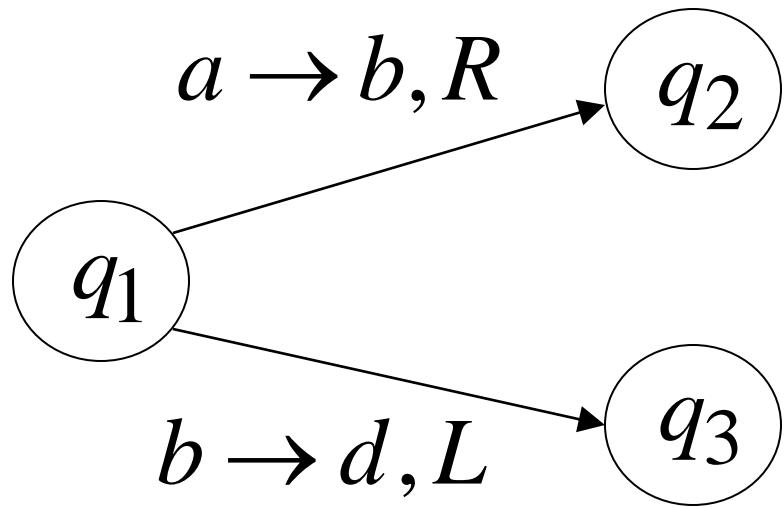
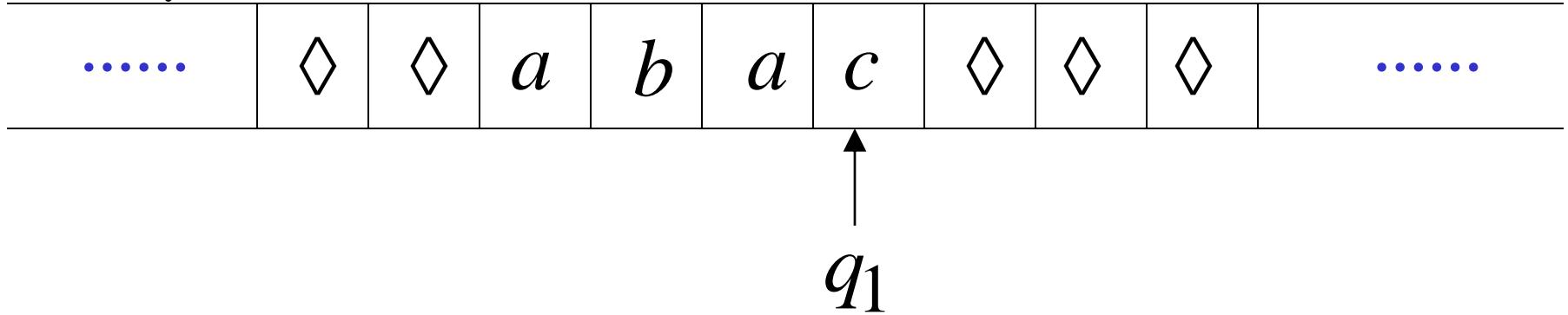


$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

# Partial Transition Function

## Not-defined for some configurations

Example:



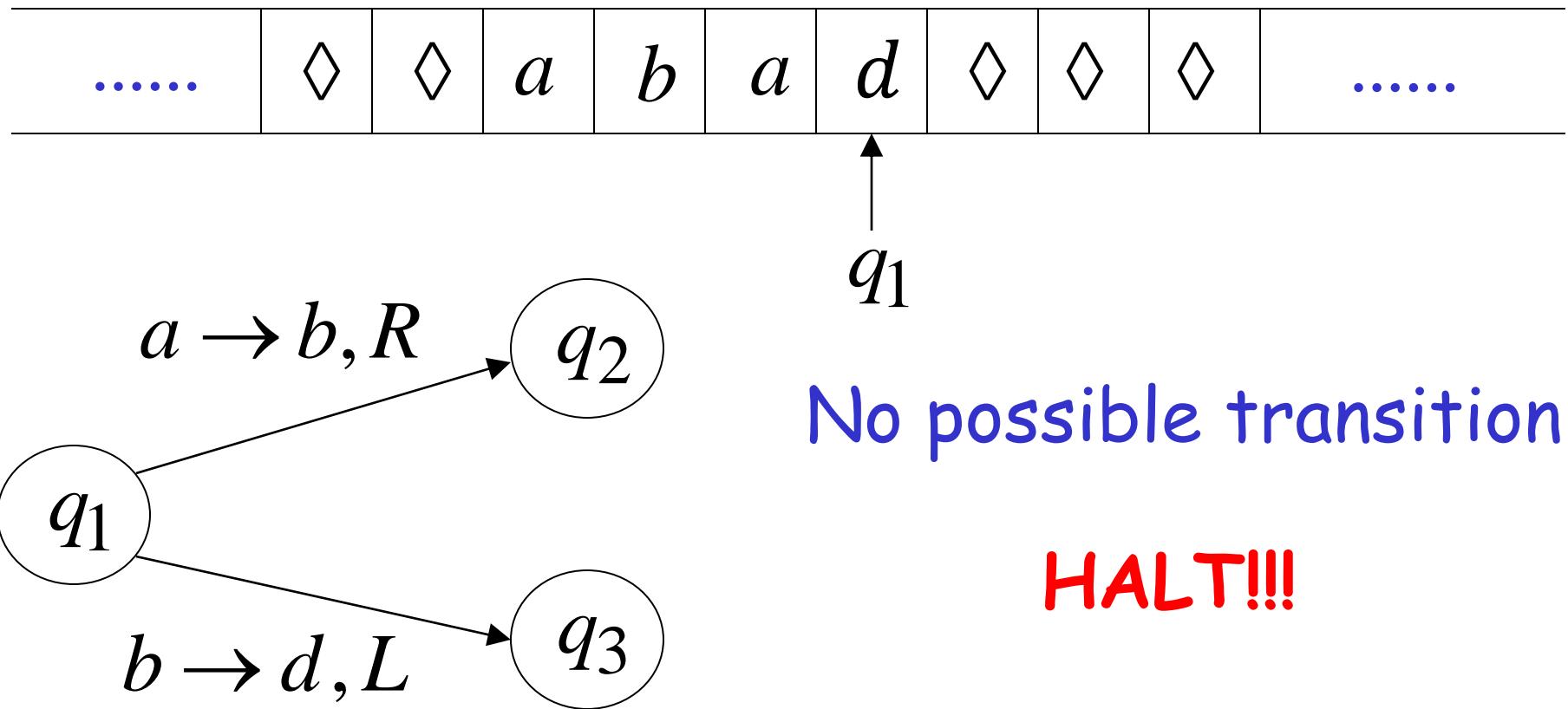
Allowed:

No transition  
for input symbol  $c$

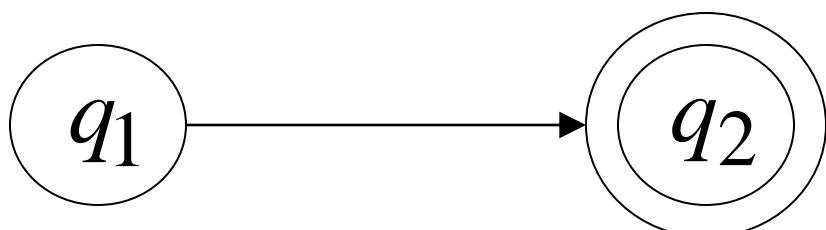
# Halting

A TM is said to **halt** whenever it reaches a configuration for which  $\delta$  is not defined.

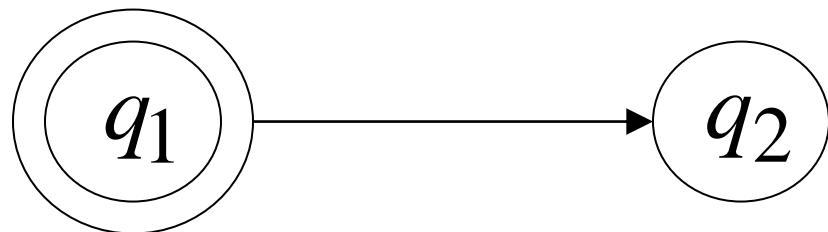
Example:



# Final States



Allowed



Not Allowed

- Final states have no outgoing transitions.
- In a final state the machine halts.

# Acceptance

Accept Input



If machine halts  
in a final state

Reject Input



If machine halts  
in a non-final state

or

If machine enters  
an infinite loop

# Assumptions

- Tape is **unbounded** in both directions (L/R).
- Tape is divided into **cells**.
- Each cell contains **only one input symbol**.
- The head can move in the **Left or Right** directions.
- Head starts at the **leftmost position** of the input string.
- The input string is **never empty**.
- Turing Machines are **deterministic**.
- The transition function is **partial**.
- Final states have no **outgoing transitions**.

# Turing Machine Example

A Turing machine that accepts the language:  $L = L(aa^*)$

$$L = \{ a, aa, aaa, aaaa, \dots \}$$

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$$

## IDEA

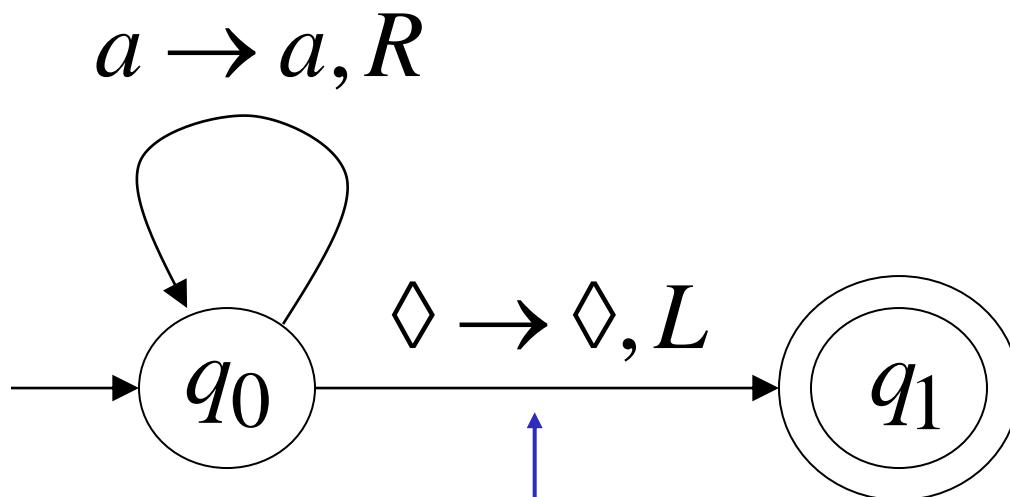
Repeat the following

- Replace leftmost a by a
- Move in the right direction to find the next a

If there are no more a's, then **accept** the string.

# Turing Machine Example

A Turing machine that accepts the language:  $L(aa^*)$

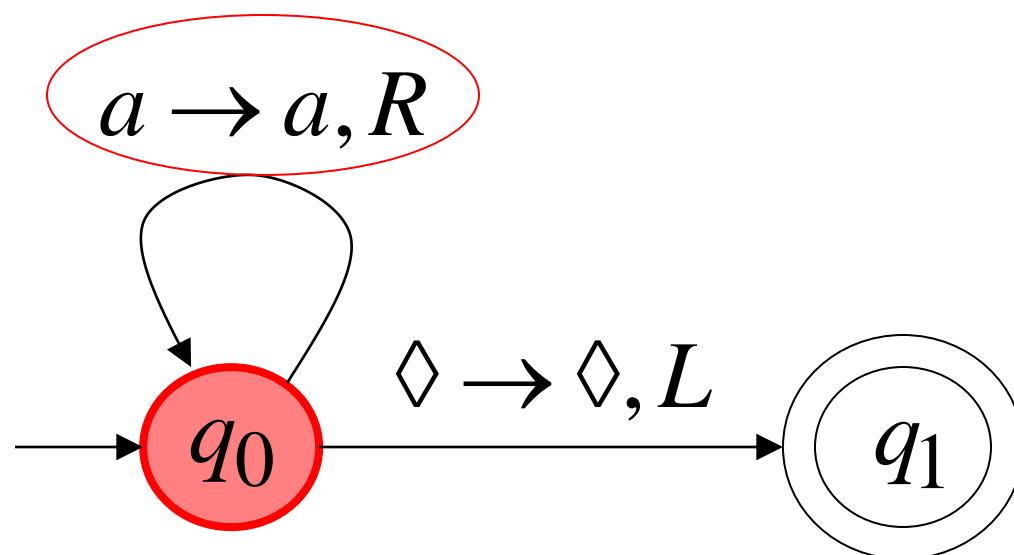
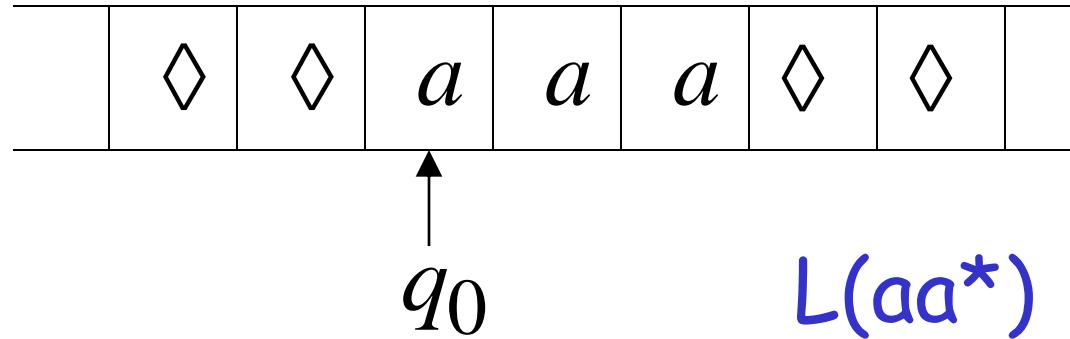


$$M = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$$

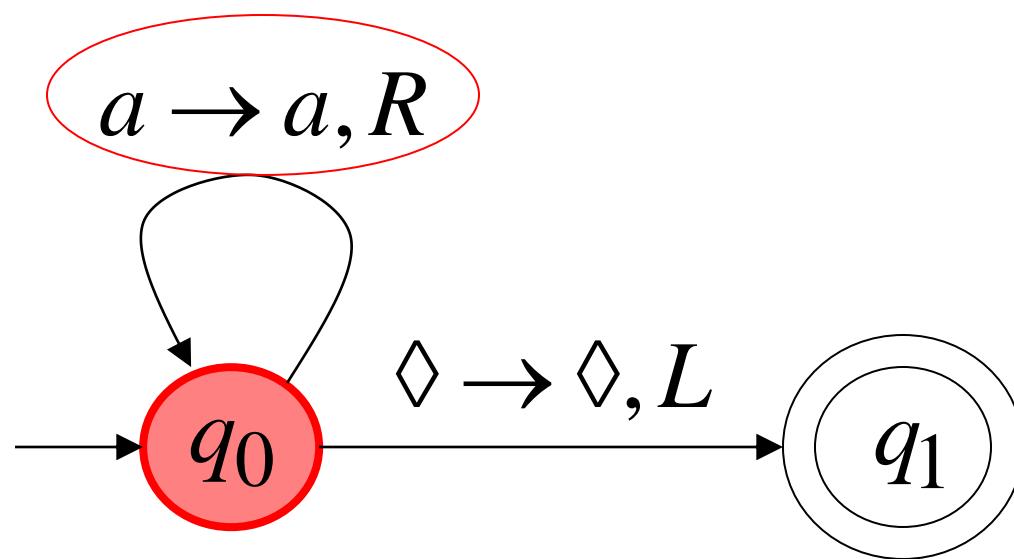
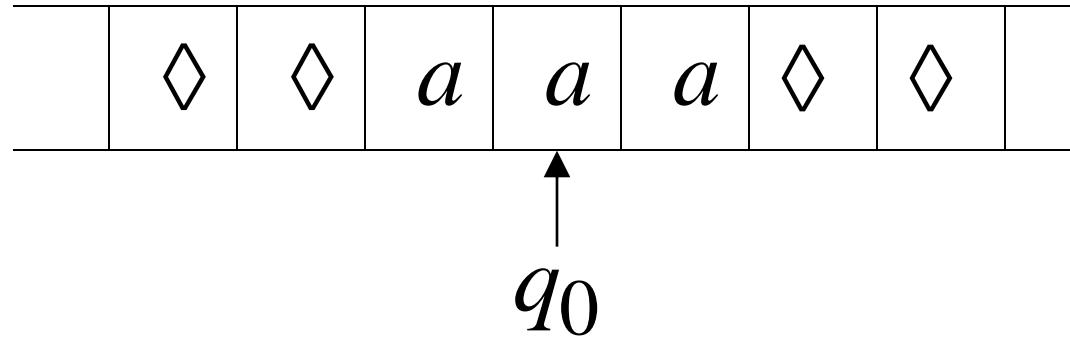
$$M = (\{q_0, q_1\}, \{a\}, \{a, \diamond\}, \delta, q_0, \diamond, \{q_1\})$$

The input string is **never empty**

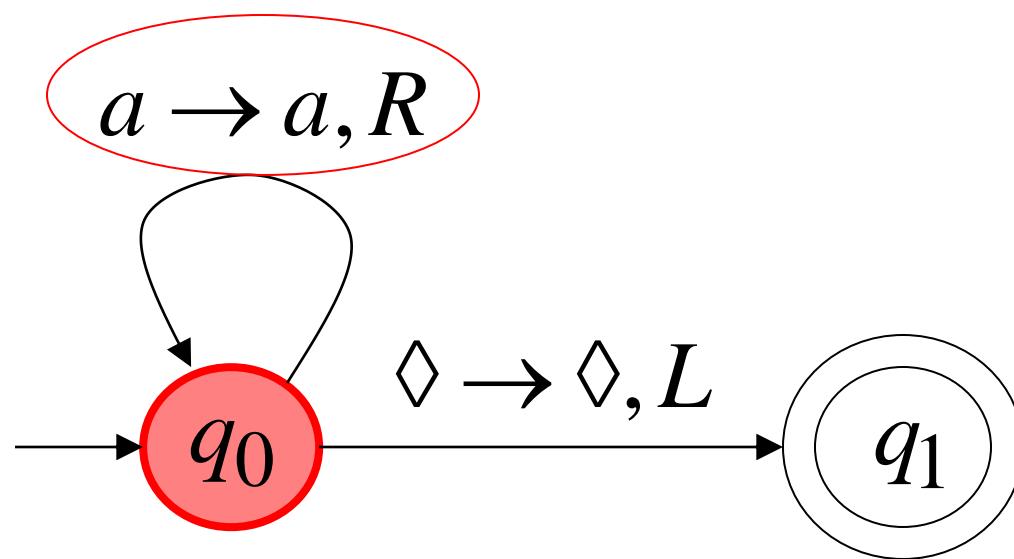
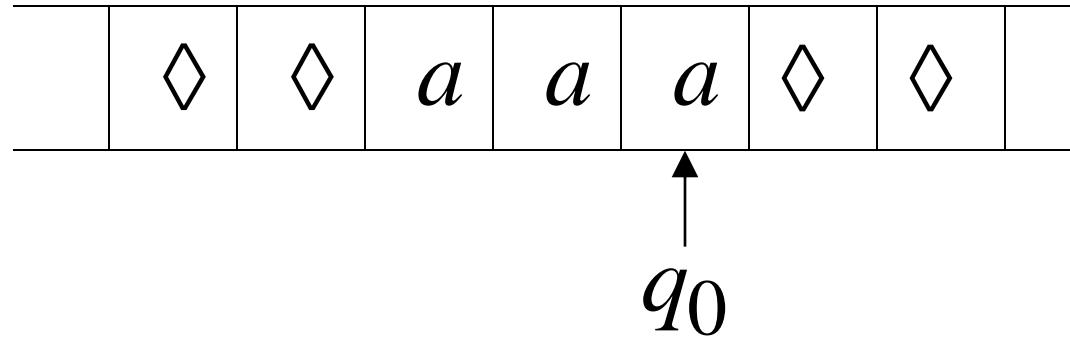
Time 0



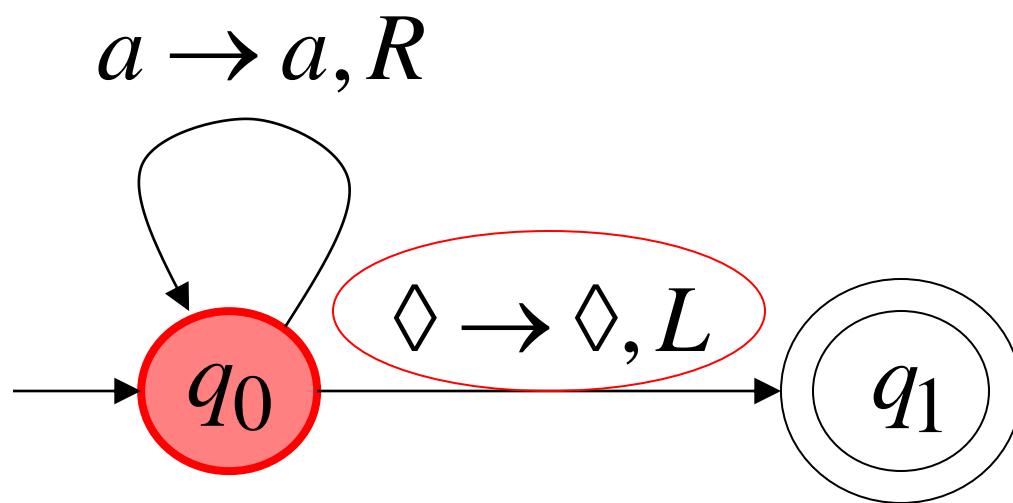
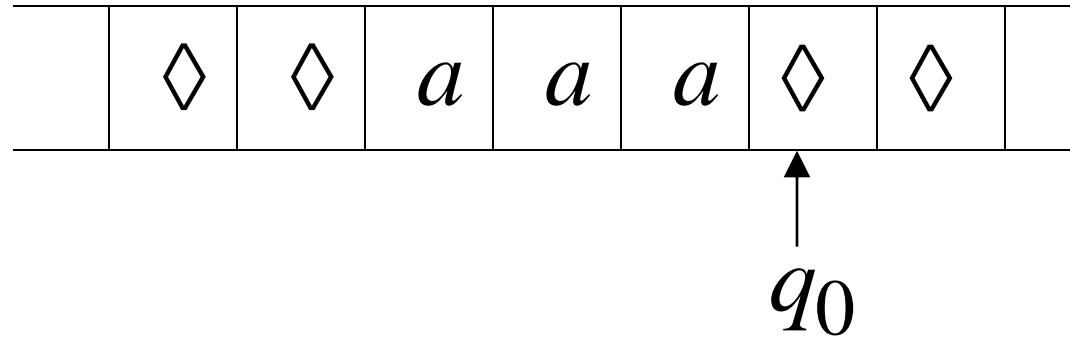
Time 1



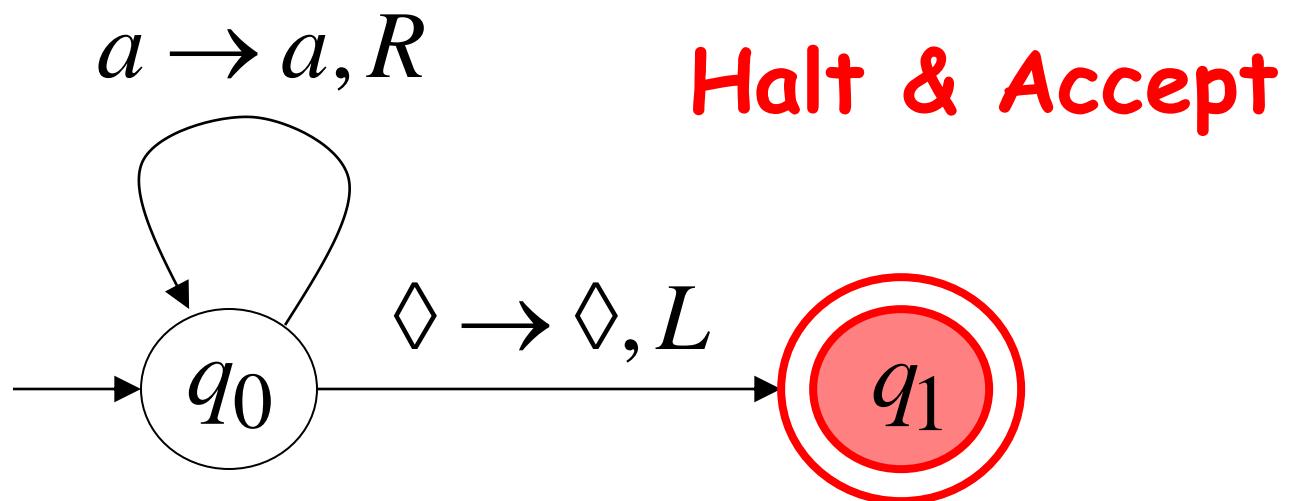
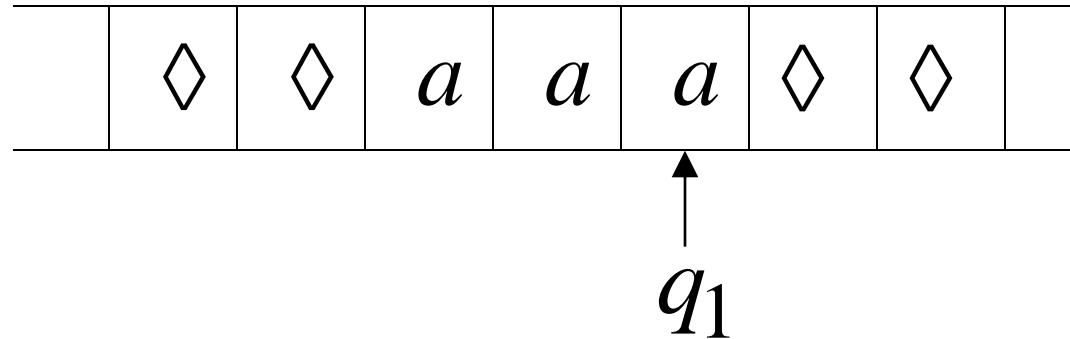
Time 2



Time 3

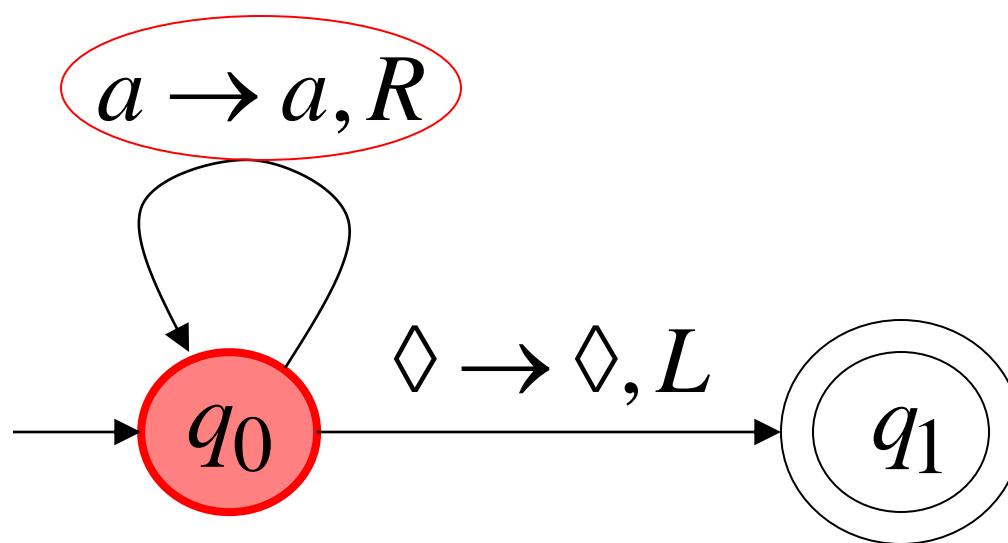
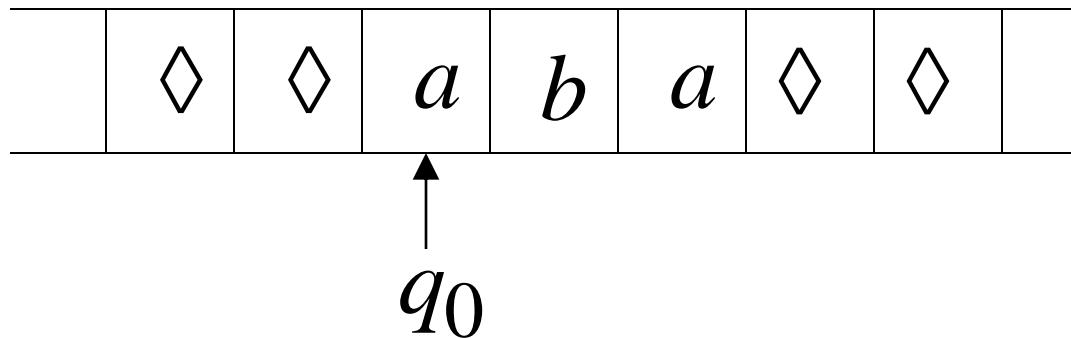


Time 4

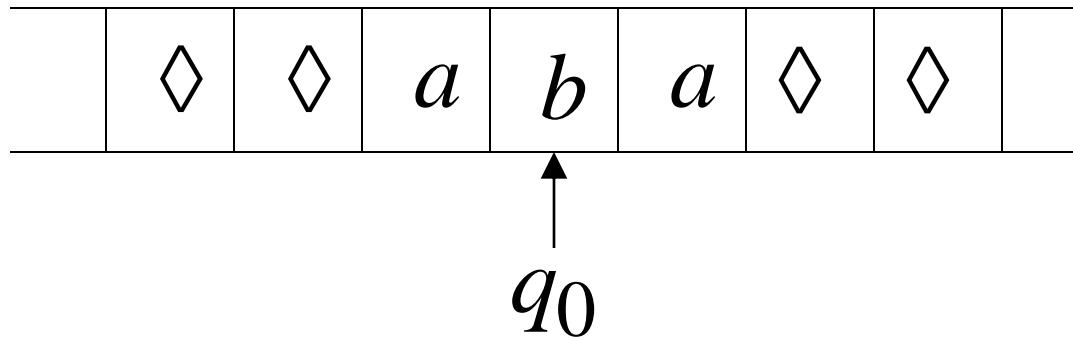


# Rejection Example

Time 0



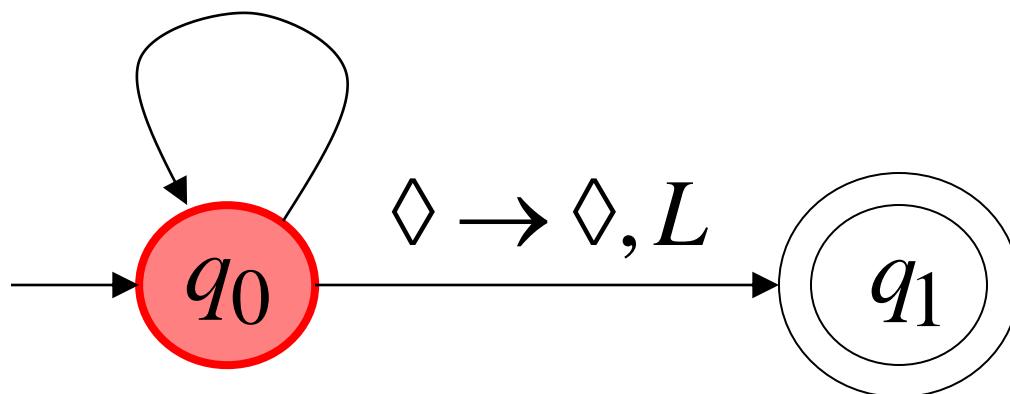
Time 1



No possible Transition

$a \rightarrow a, R$

Halt & Reject

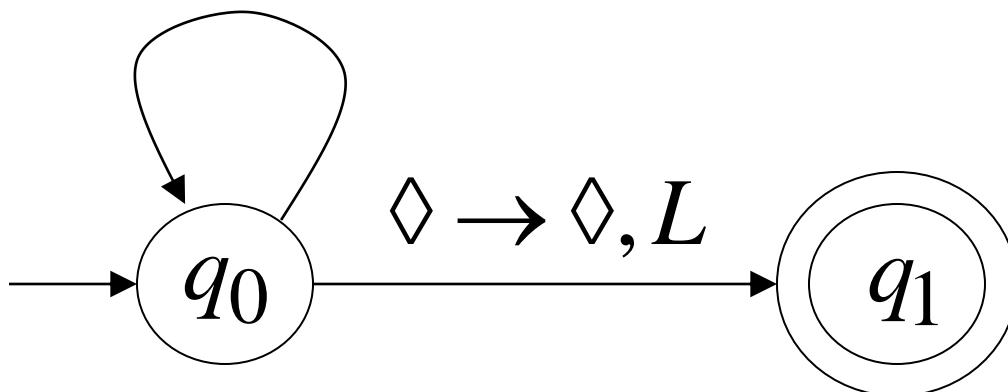


# Infinite Loop Example

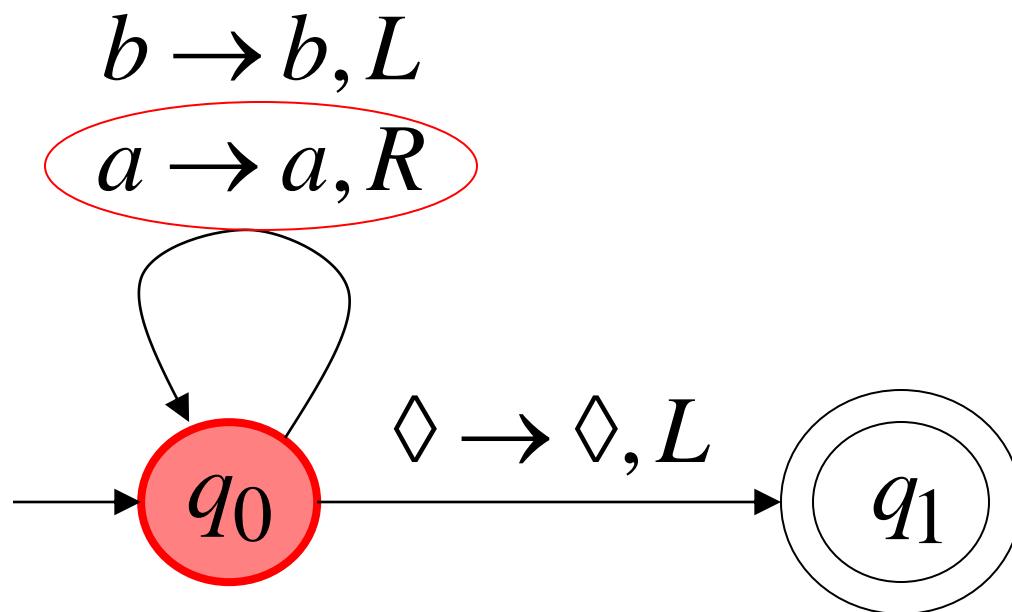
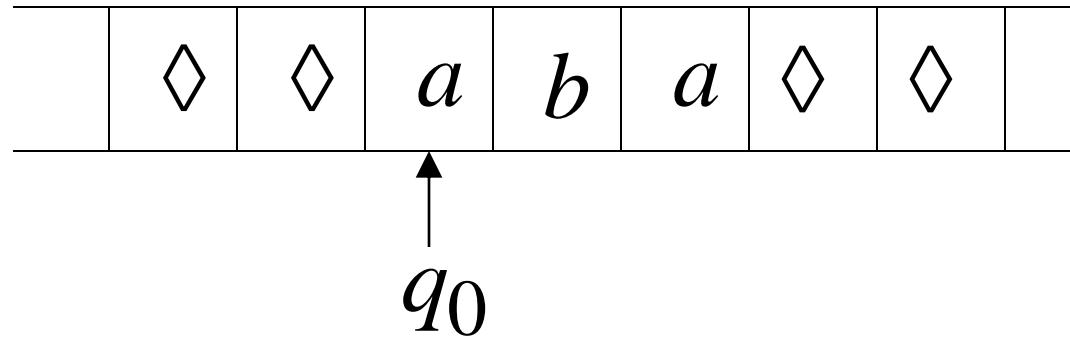
A Turing machine  $aa^* + b(a+b)^*$  for language.

$$b \rightarrow b, L$$

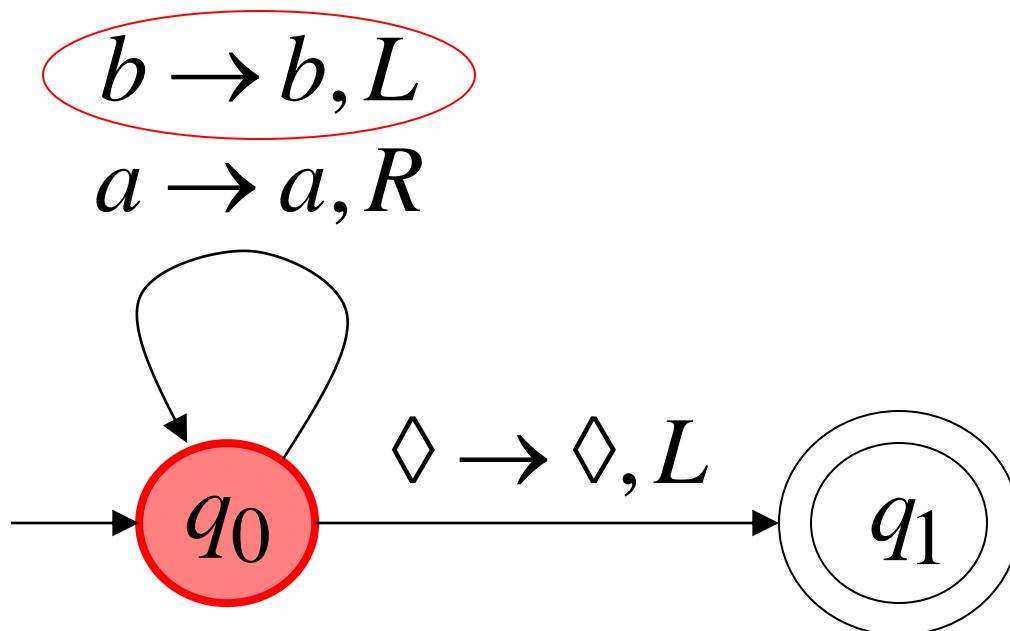
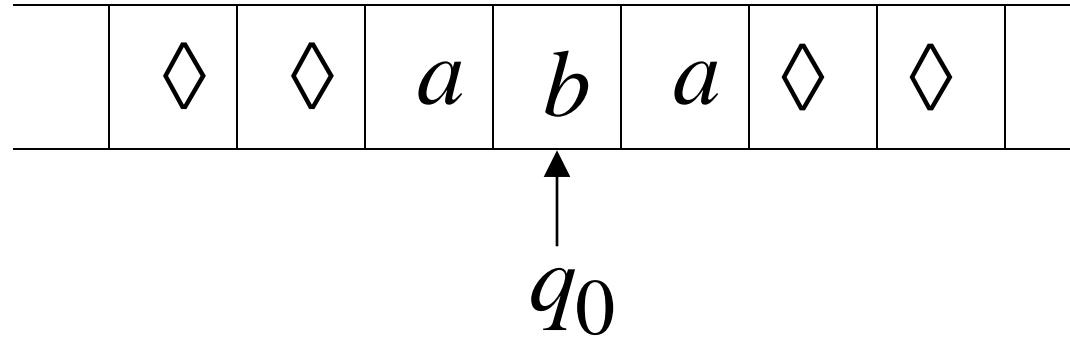
$$a \rightarrow a, R$$



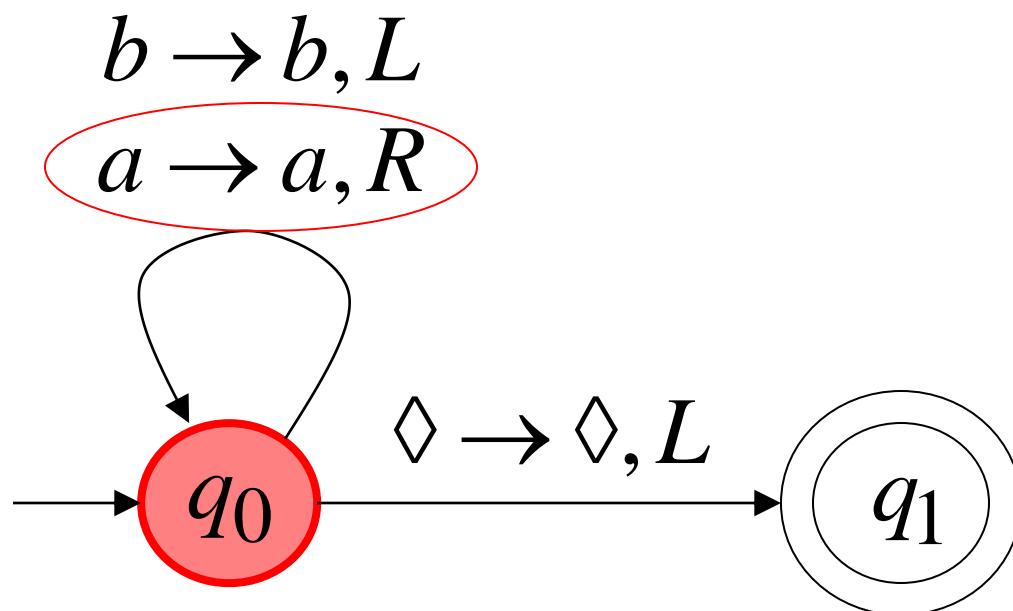
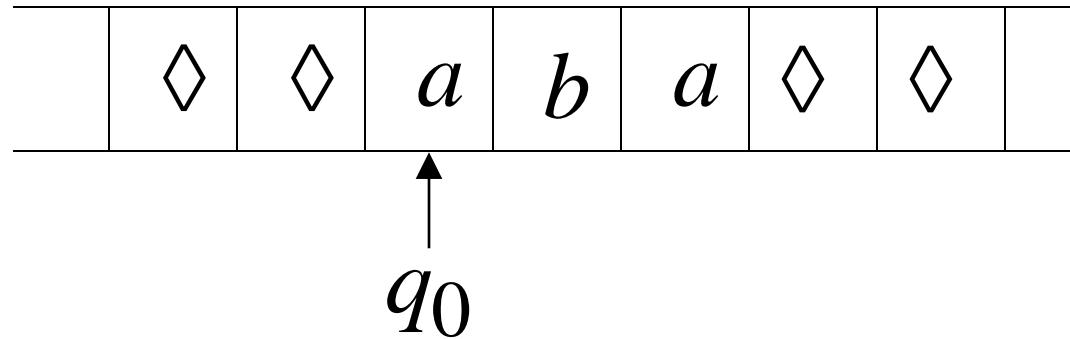
Time 0



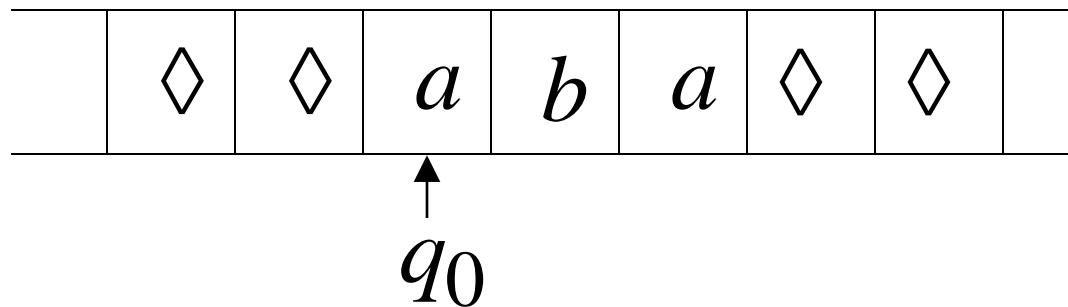
Time 1



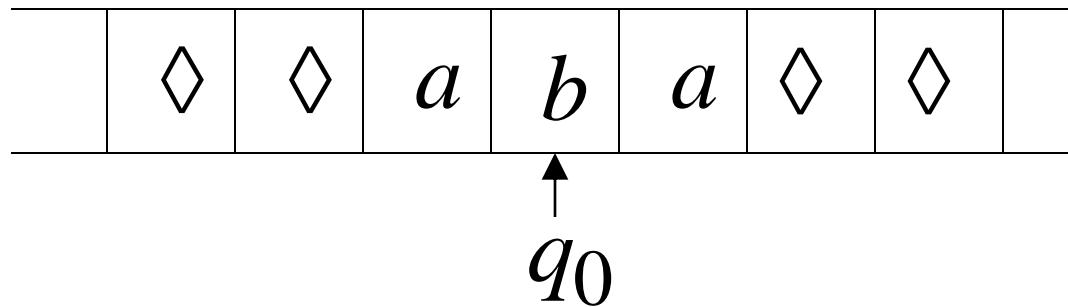
Time 2



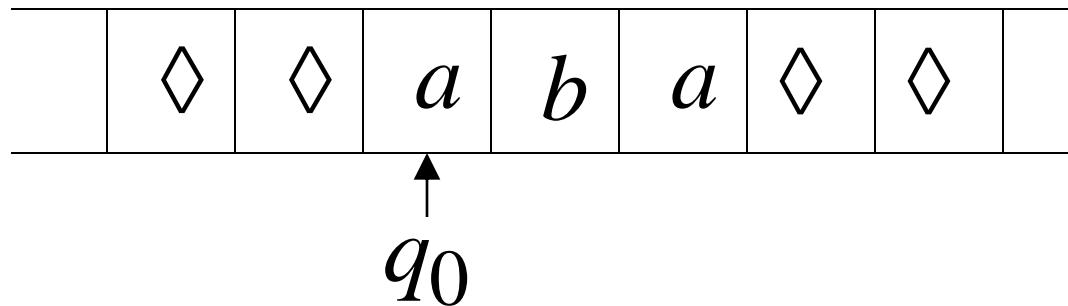
Time 2



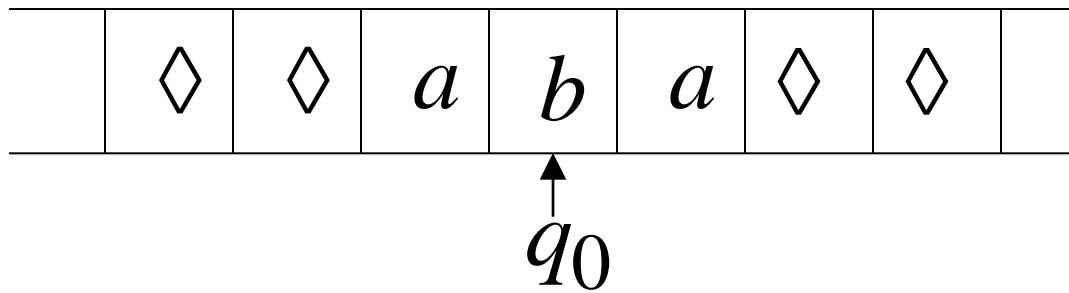
Time 3



Time 4



Time 5



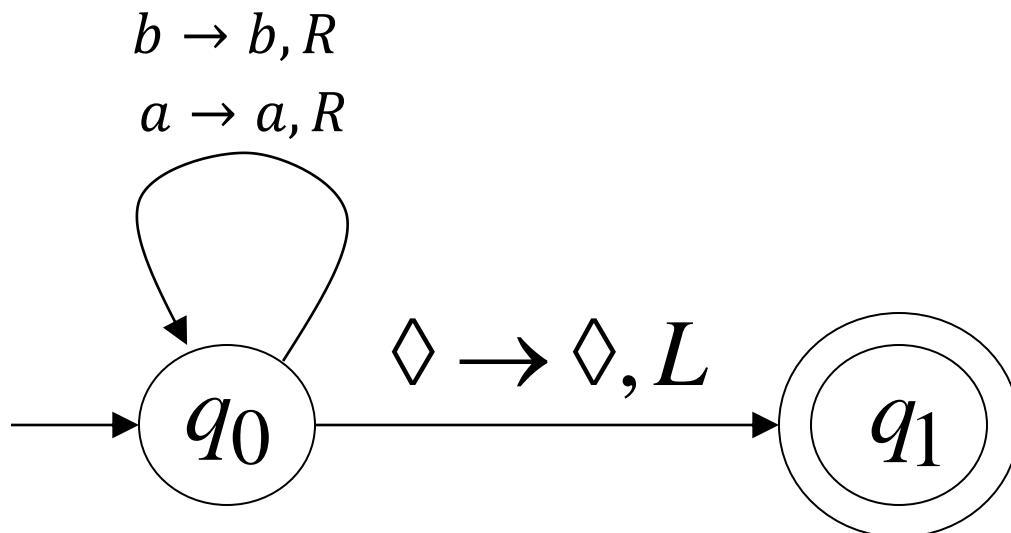
Infinite loop

Because of the infinite loop:

- The final state cannot be reached
- The machine never halts
- The input is never accepted

# Correct TM without Infinite Loop

A Turing machine for language:  $aa^* + b(a+b)^*$



# Turing Machine

Design Turing machine for the language  $L = \{a^n b^n : n \geq 1\}$

**aaabbbb**

## IDEA

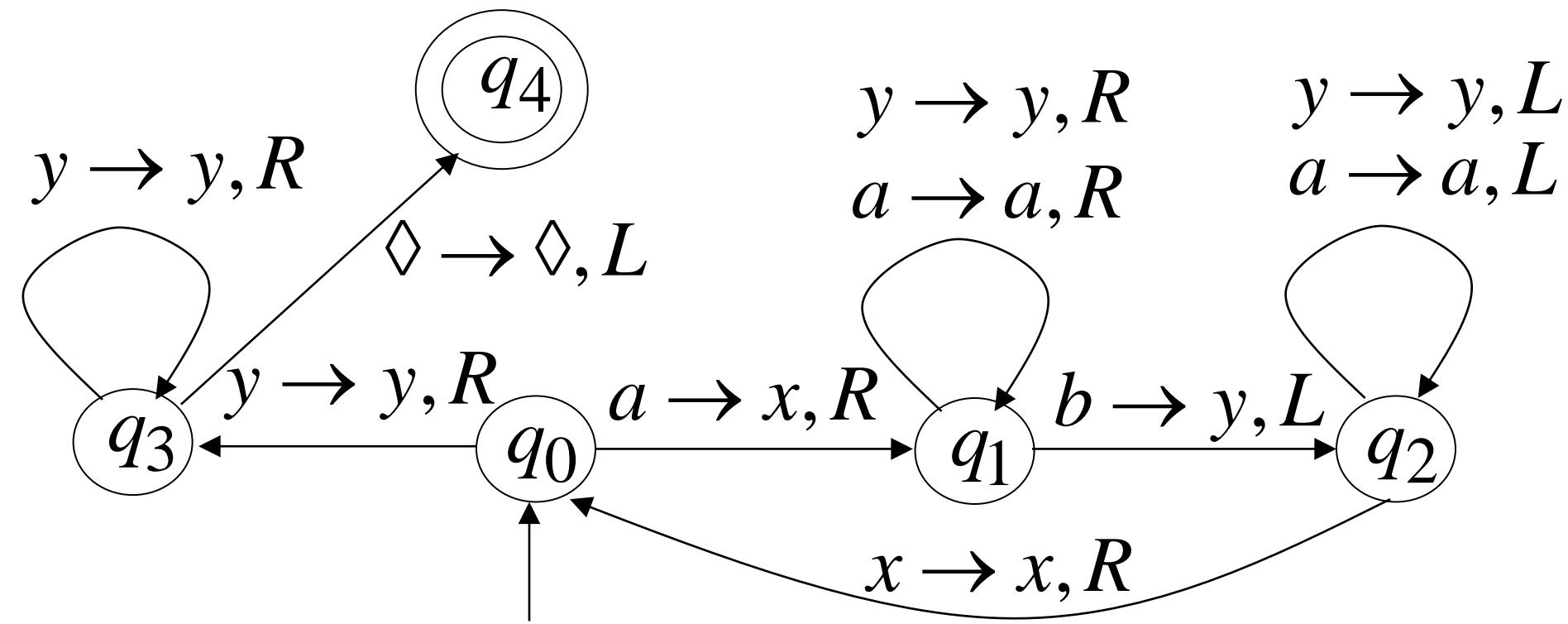
Repeat the following

- Replace leftmost a by x
- Move in the right direction to find the leftmost b;
- Replace b with y, Move in the left direction.

If there are no more a's,

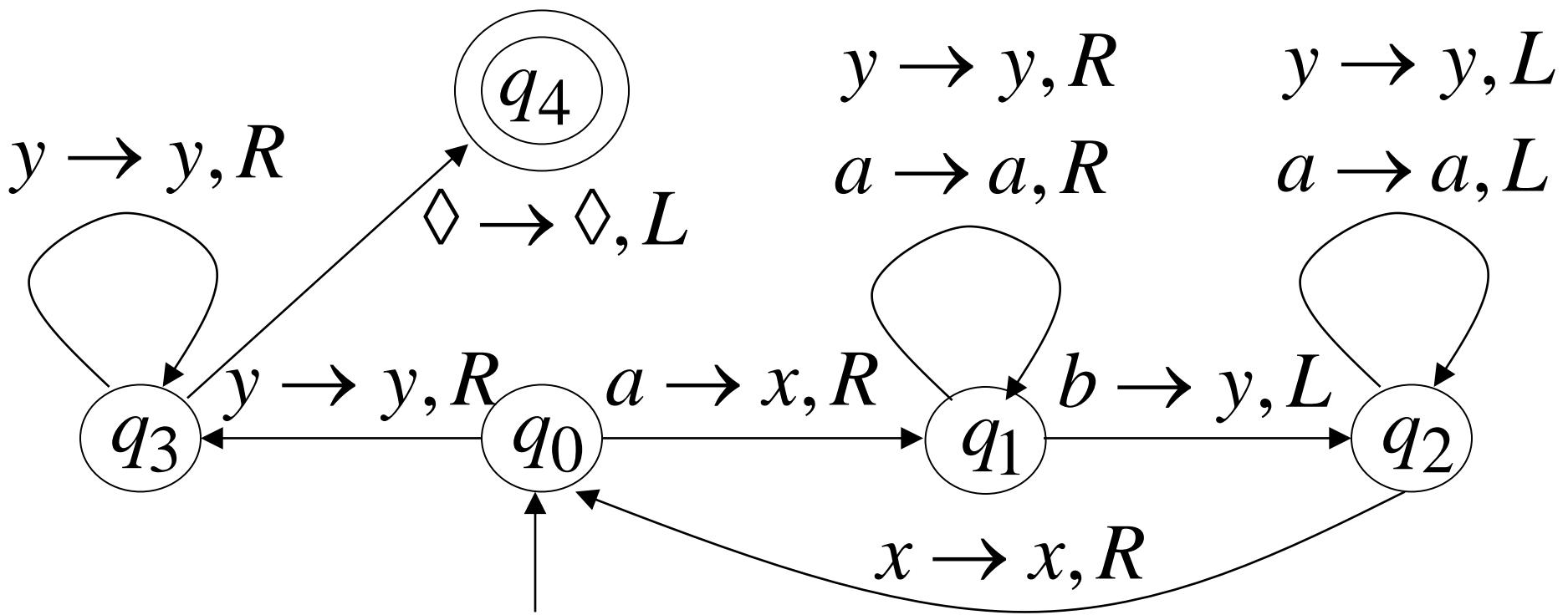
- Search tape for more b's
- If there are no more b's, then **accept** the string.

\$\$\$\$aaabbb\$\$\$\$  
xaabbb  
xaaybb → xxayyb → xxxyyb  
xxxxyy                           xxxxyy

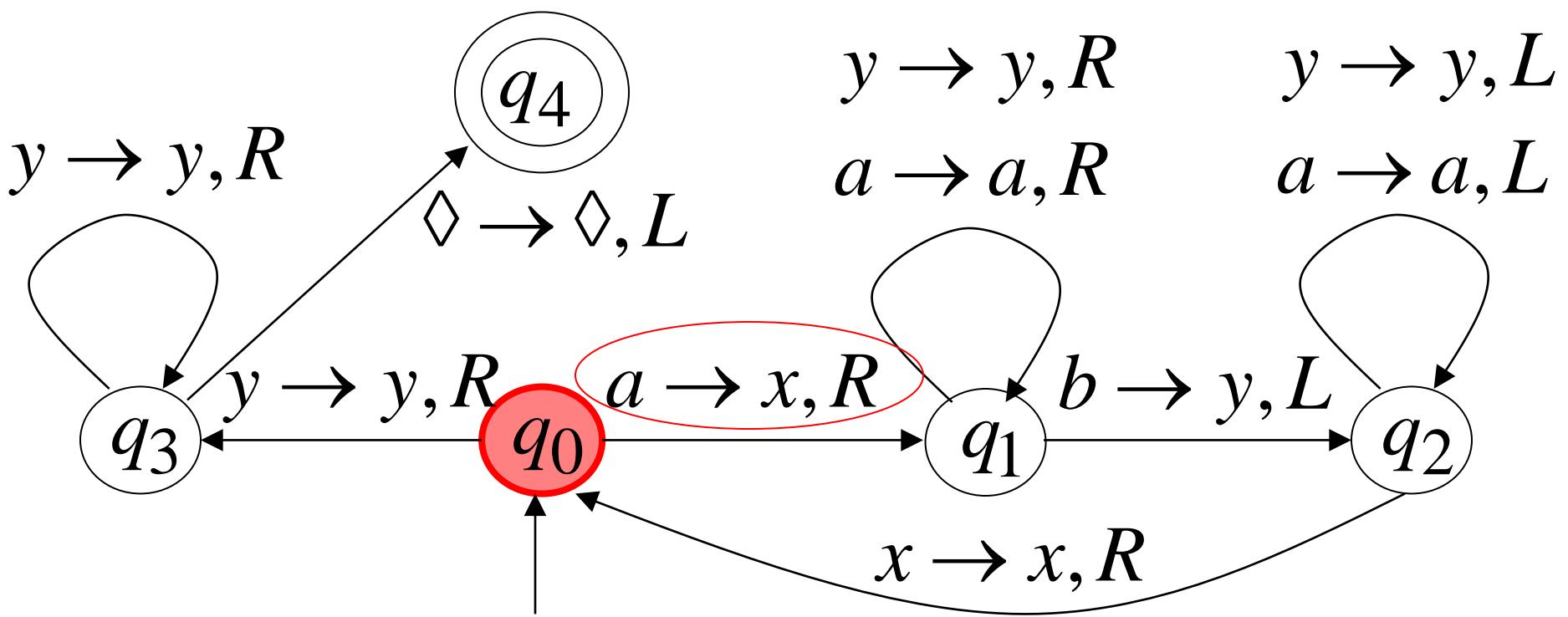
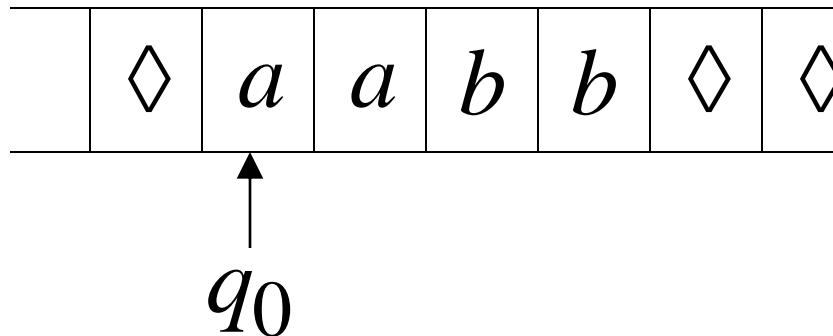


# Turing Machine

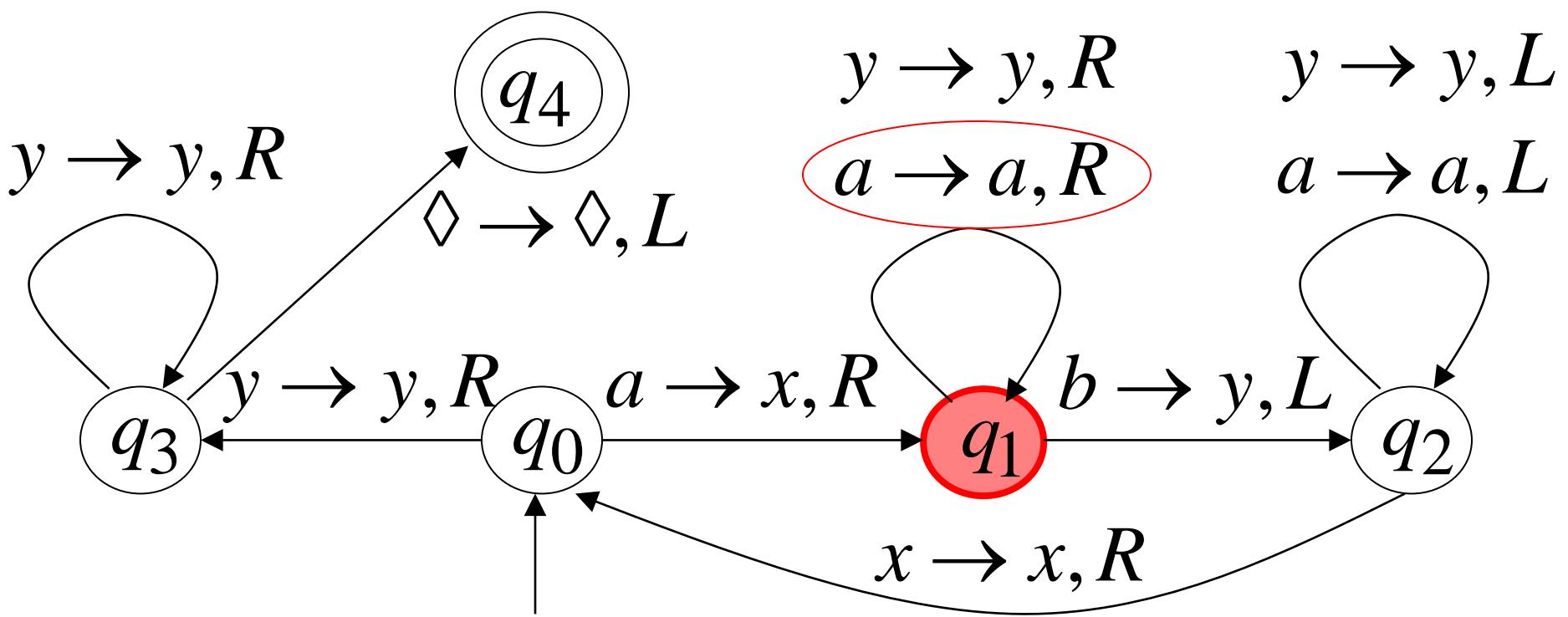
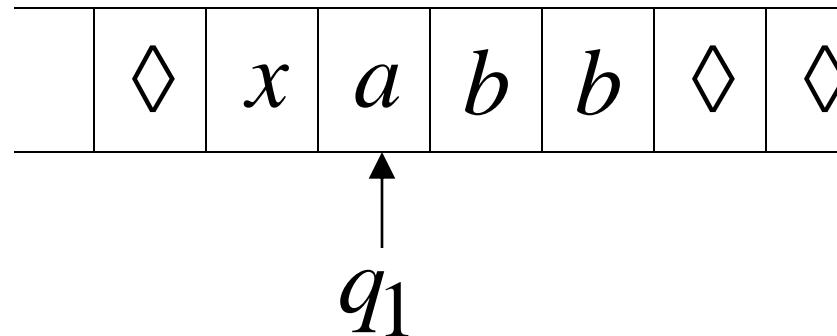
Turing machine for the language  
 $L = \{a^n b^n : n \geq 1\}$



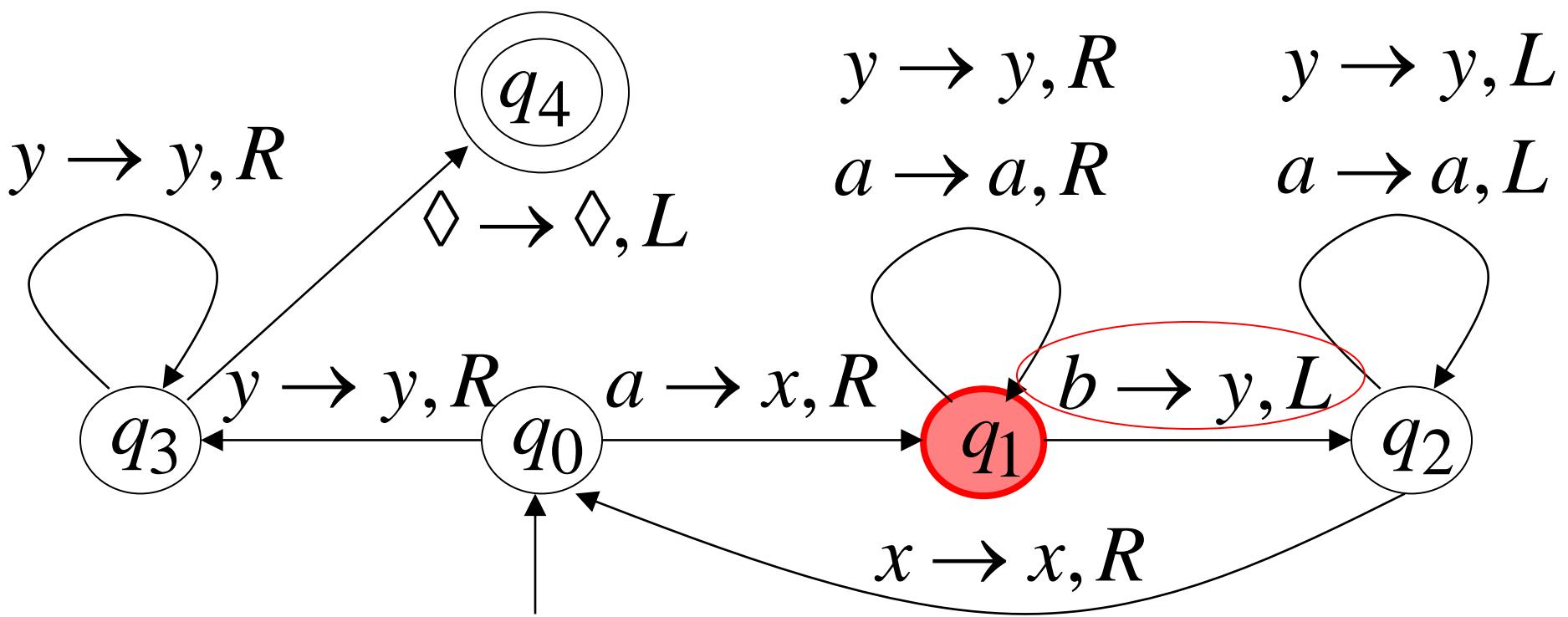
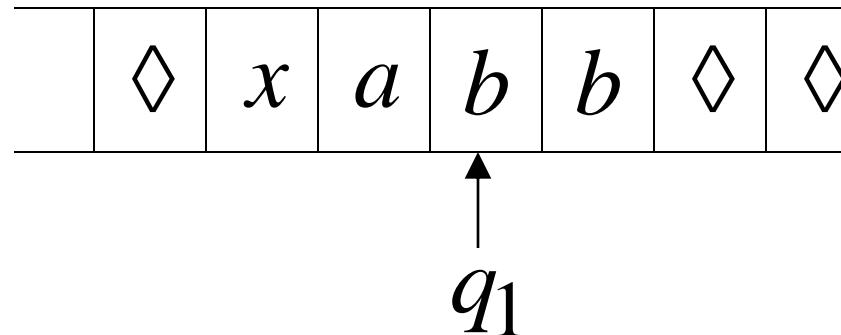
Time 0



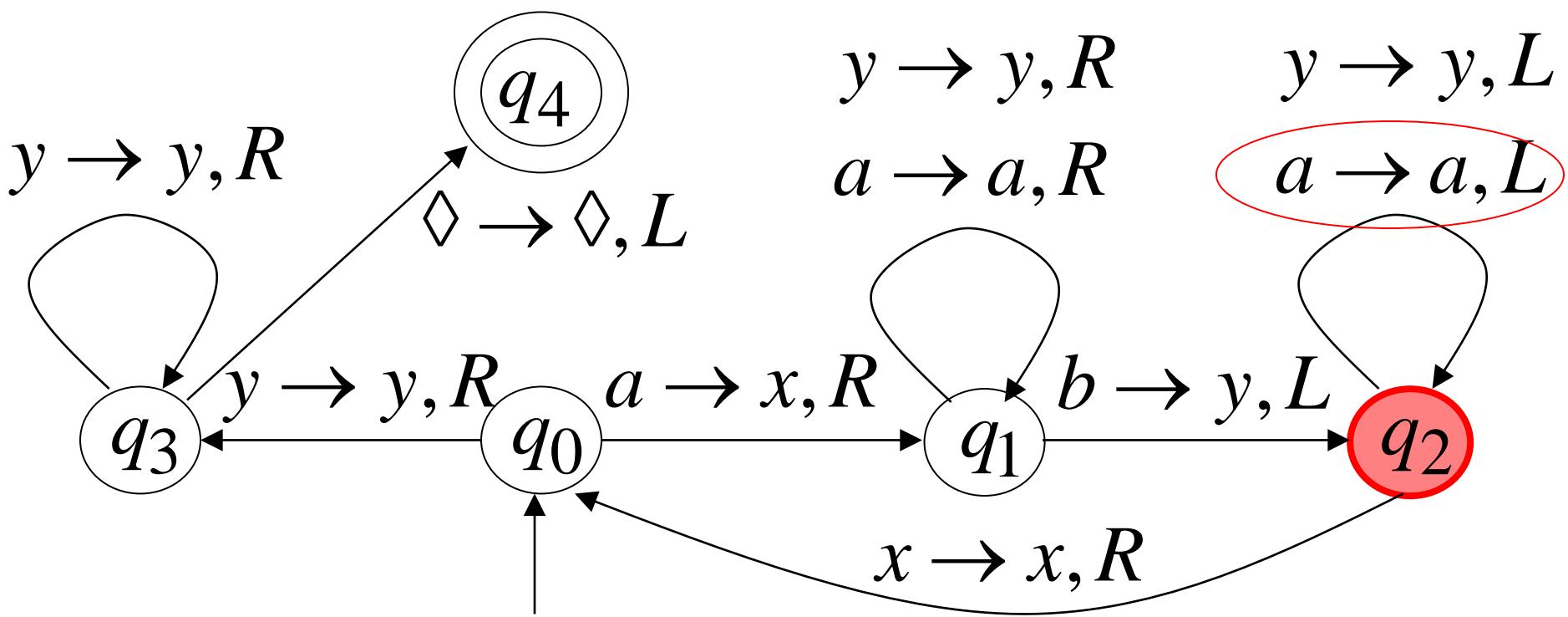
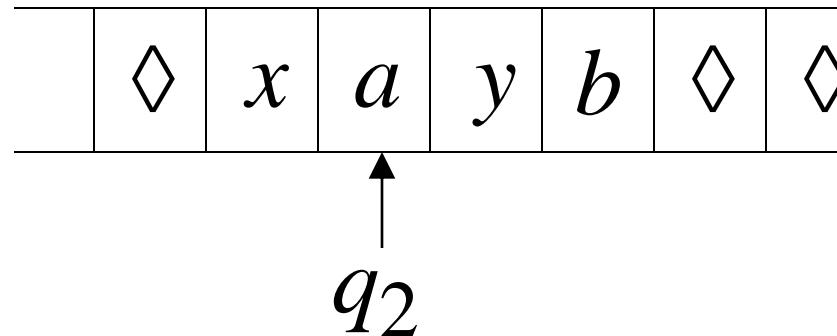
Time 1



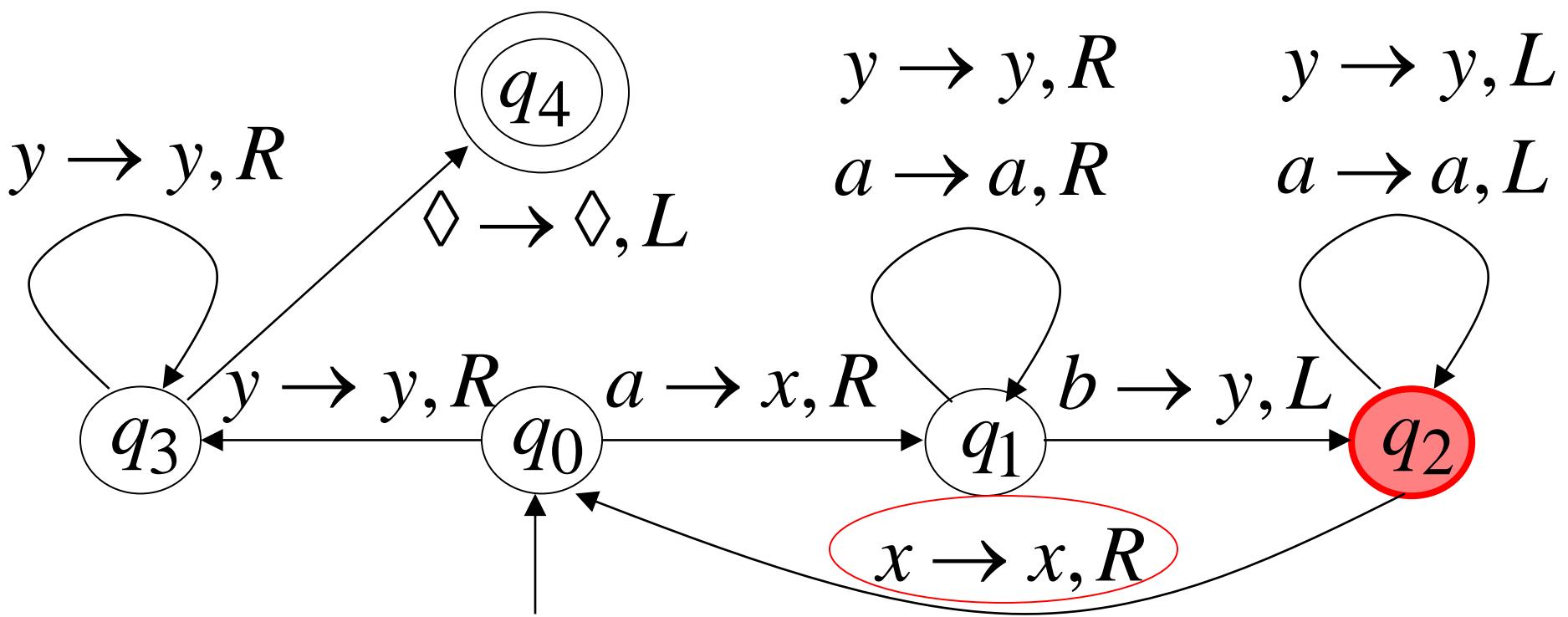
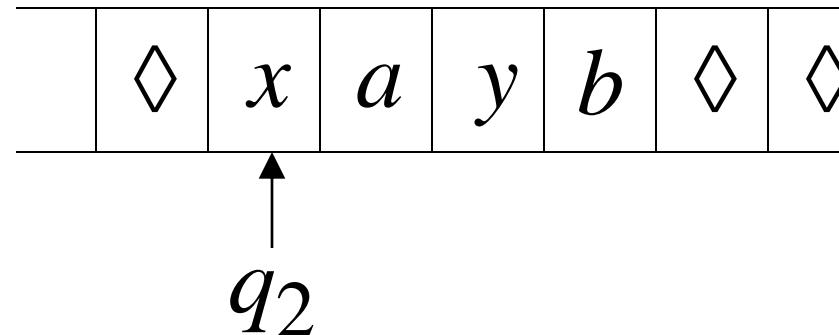
Time 2



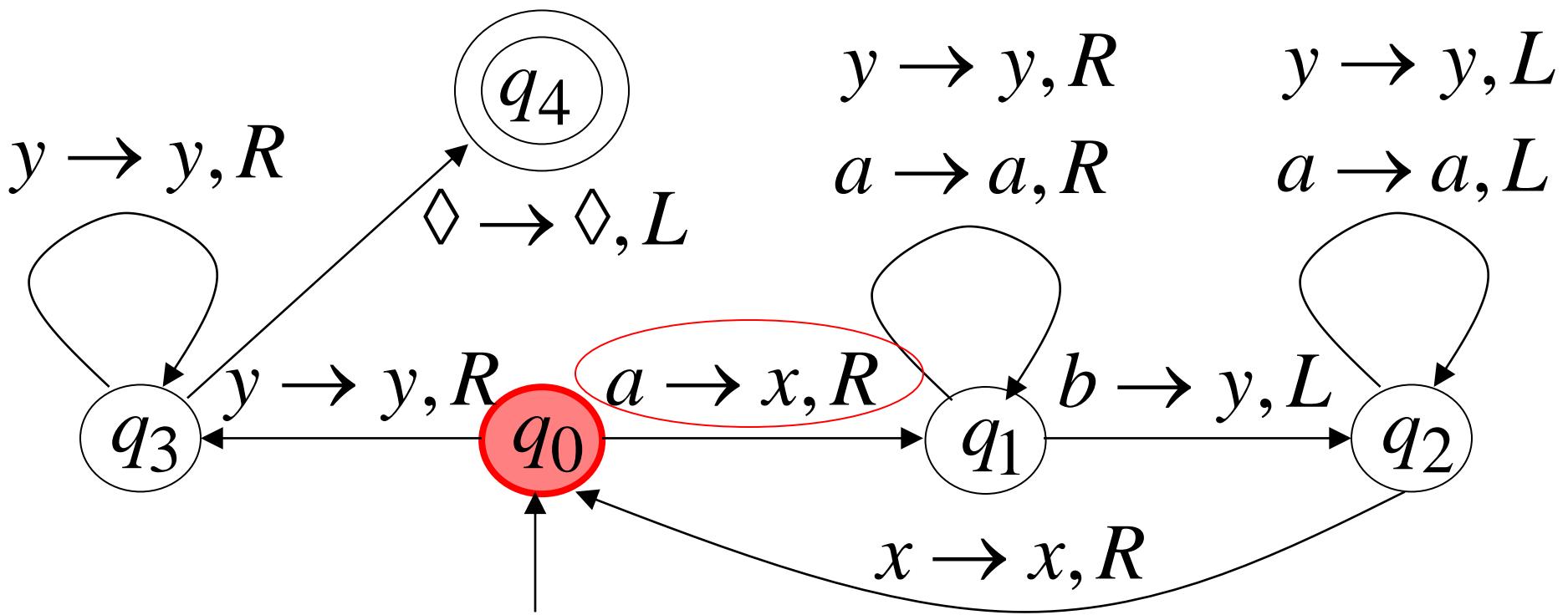
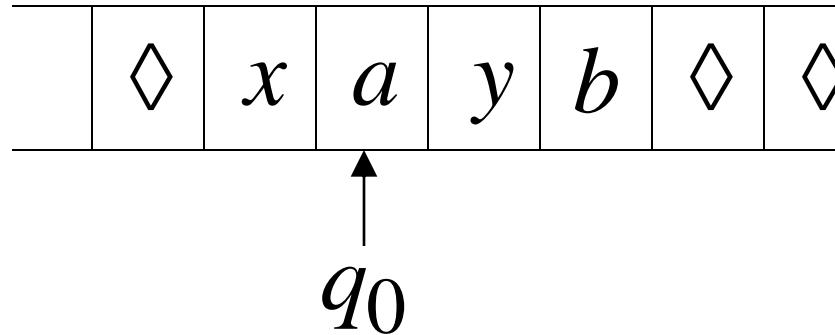
Time 3



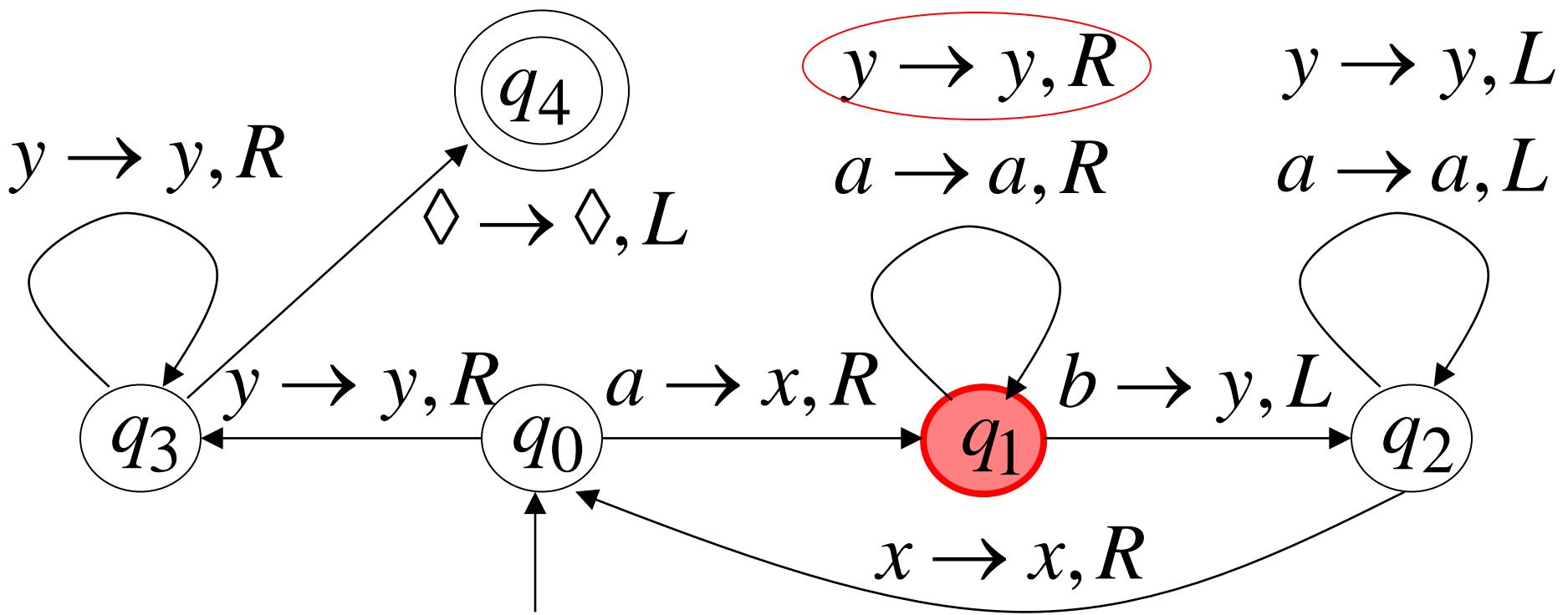
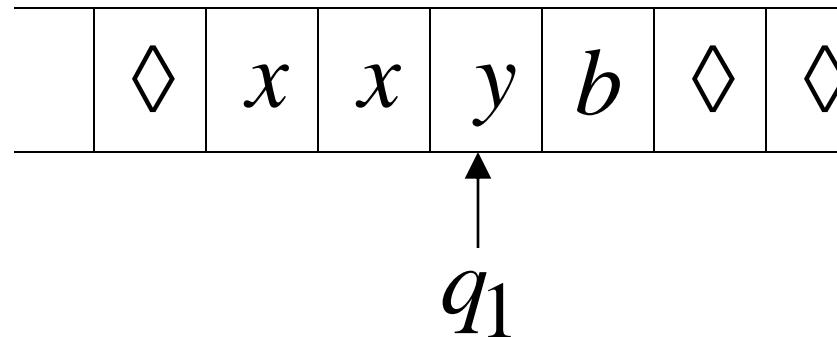
Time 4



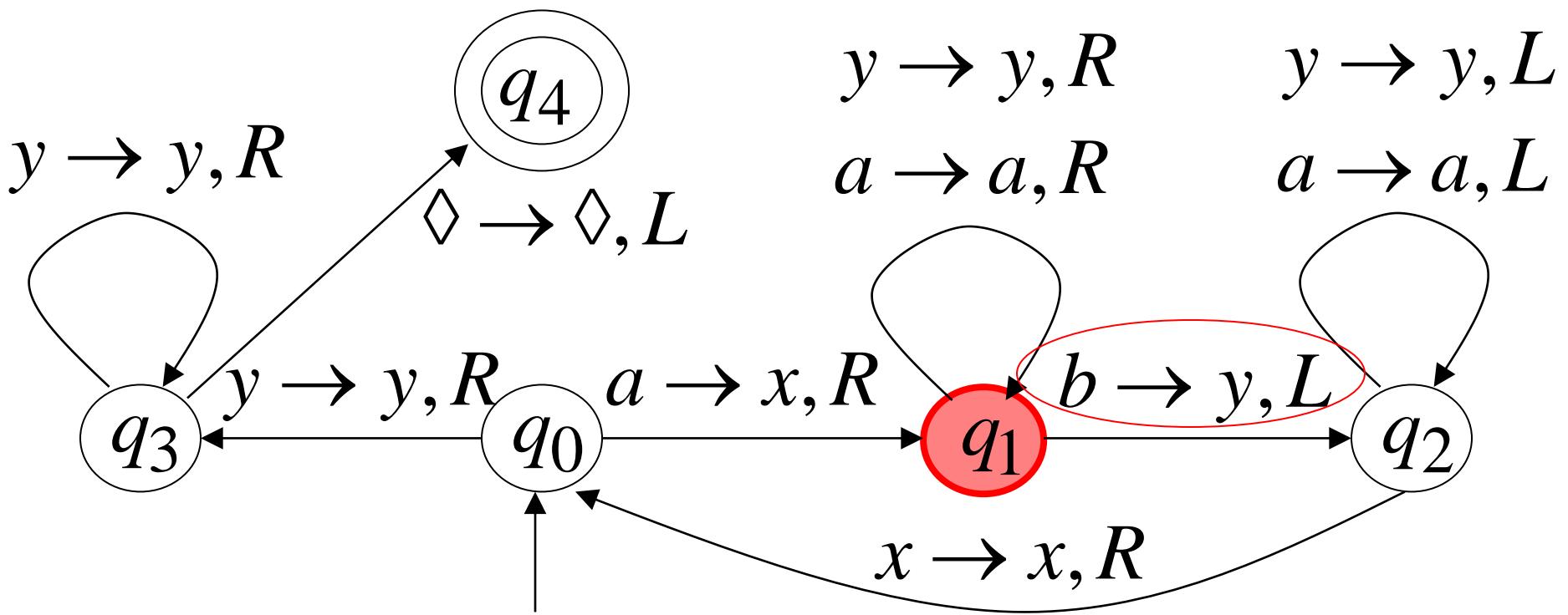
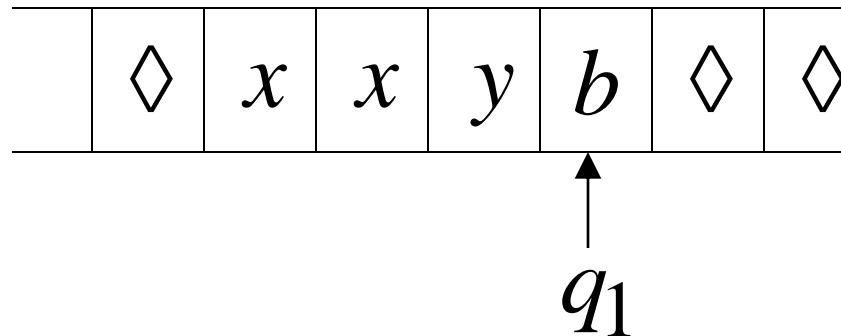
Time 5



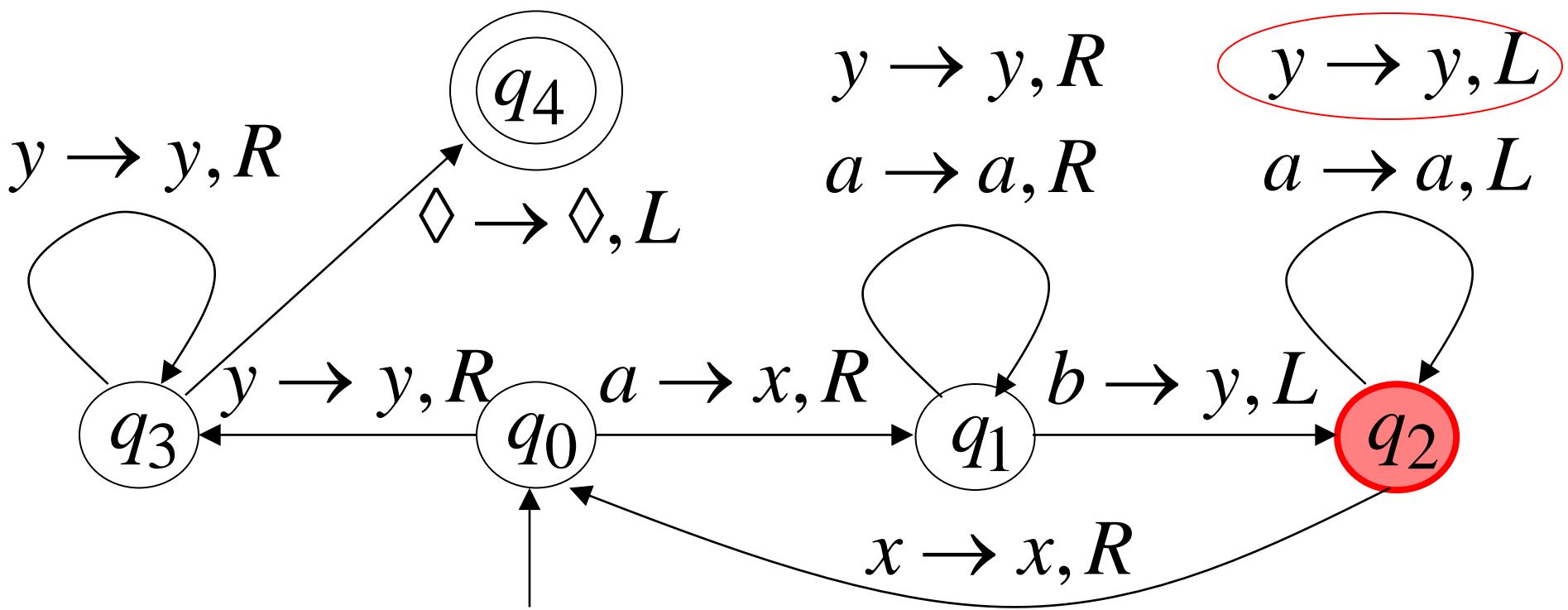
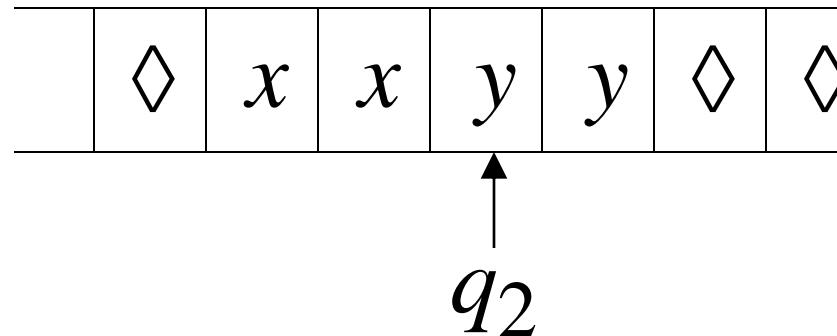
Time 6



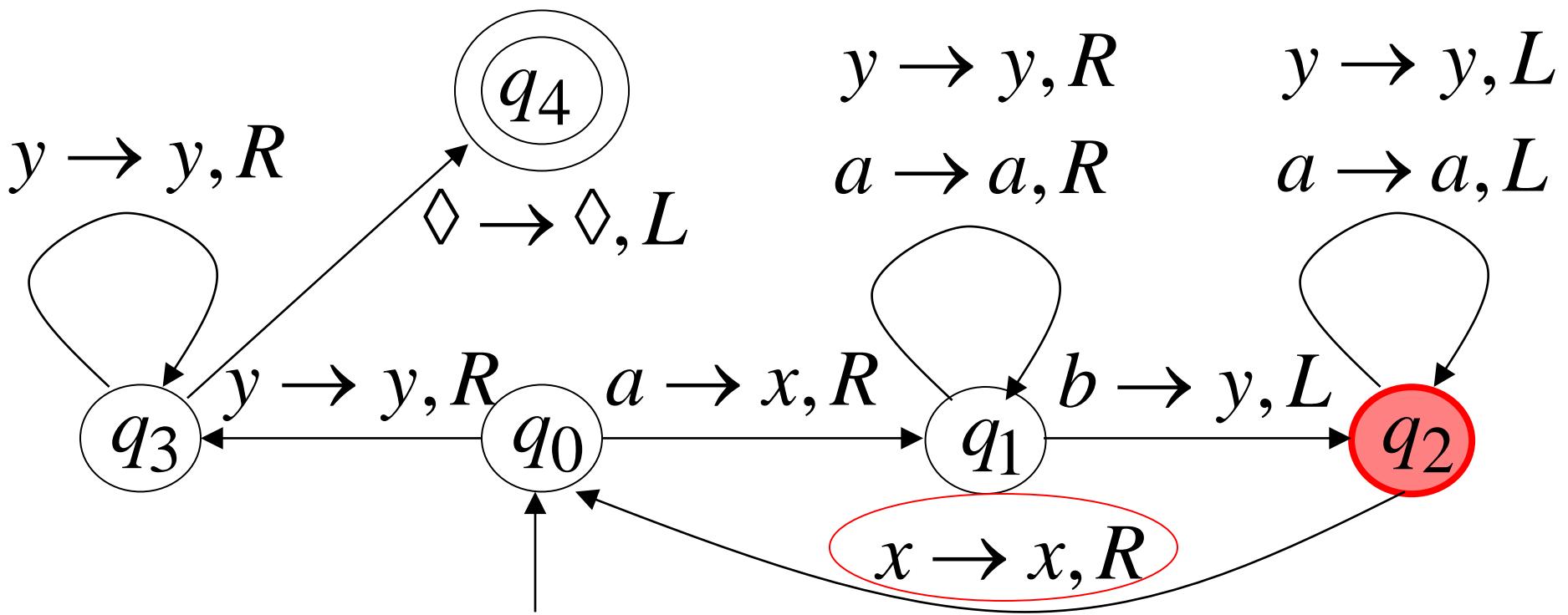
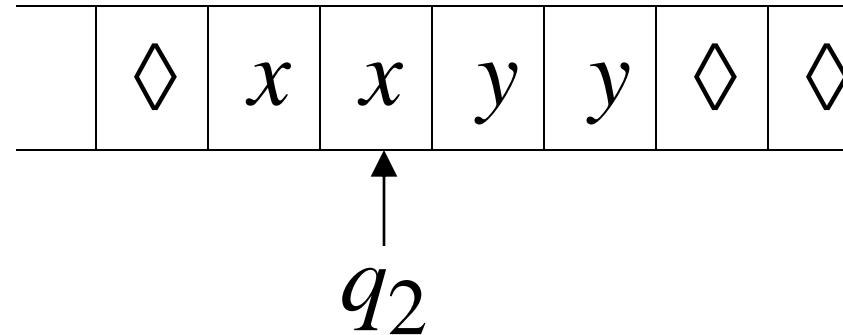
Time 7



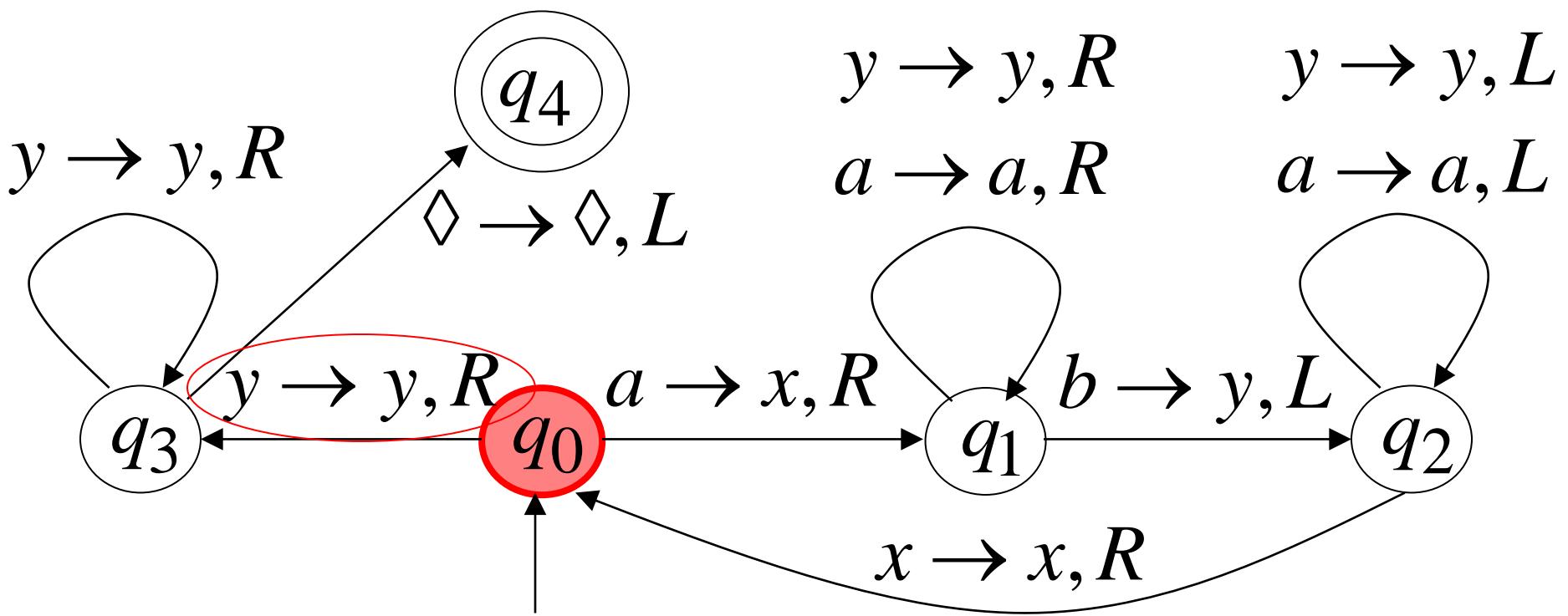
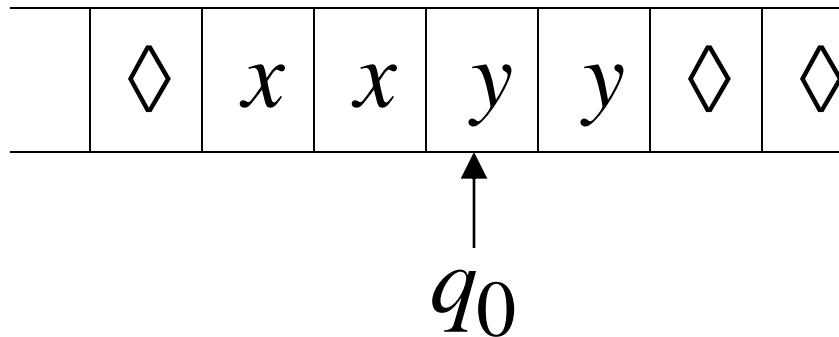
Time 8



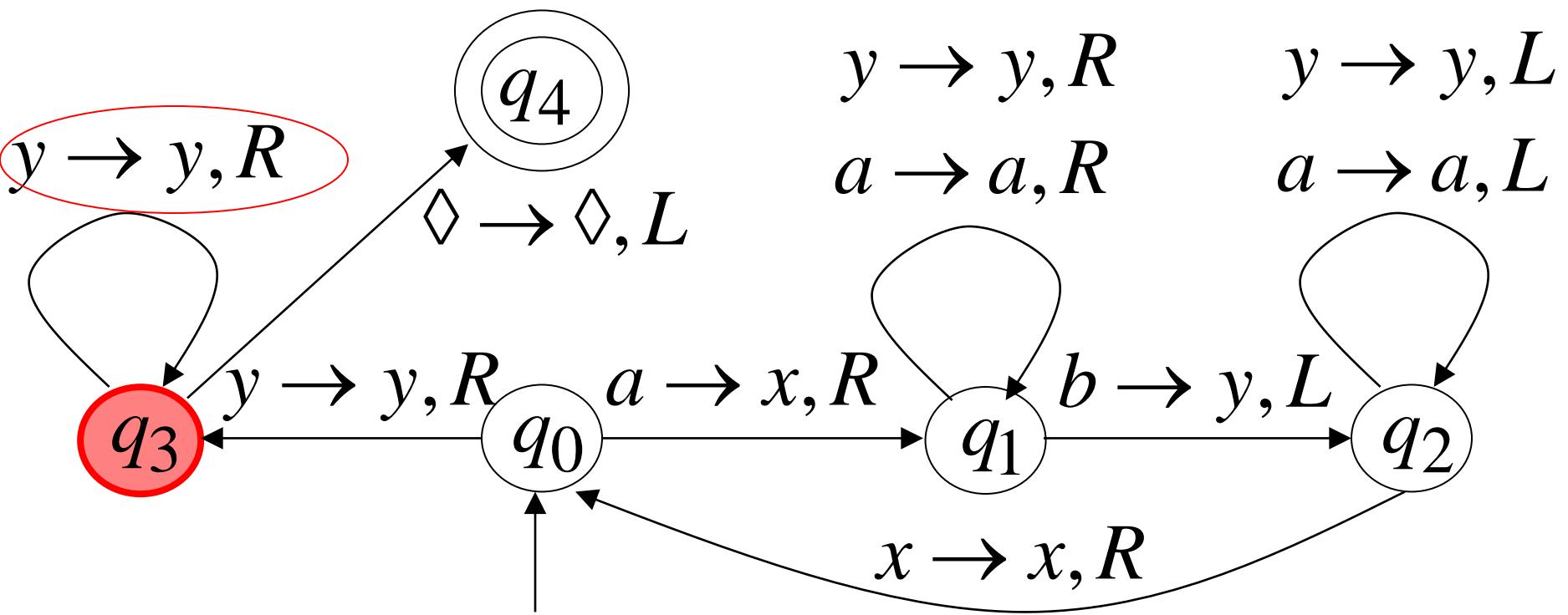
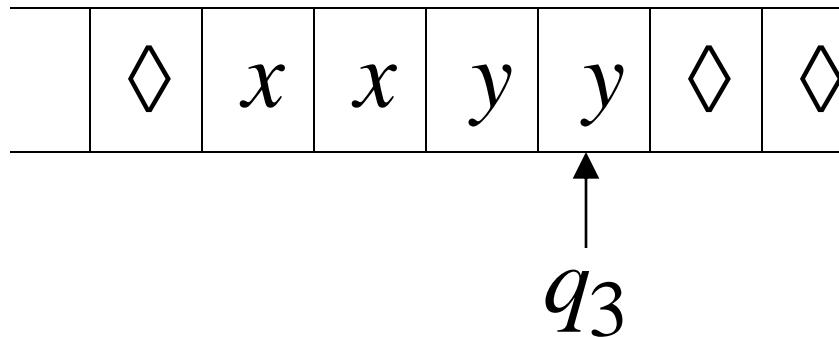
9



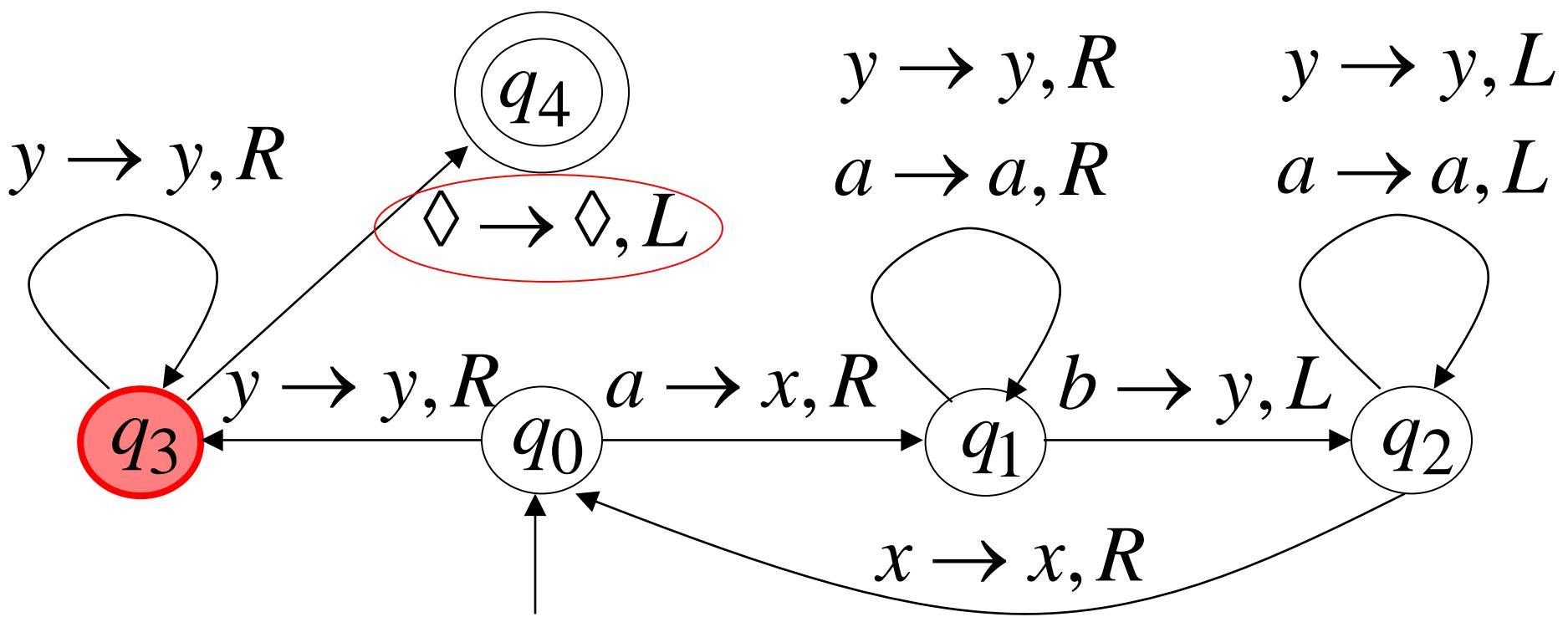
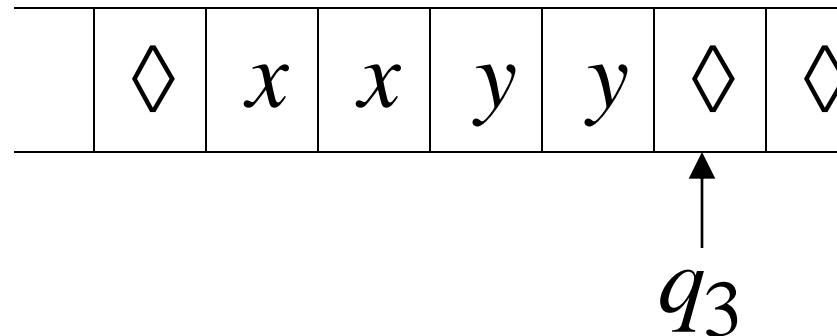
Time 10



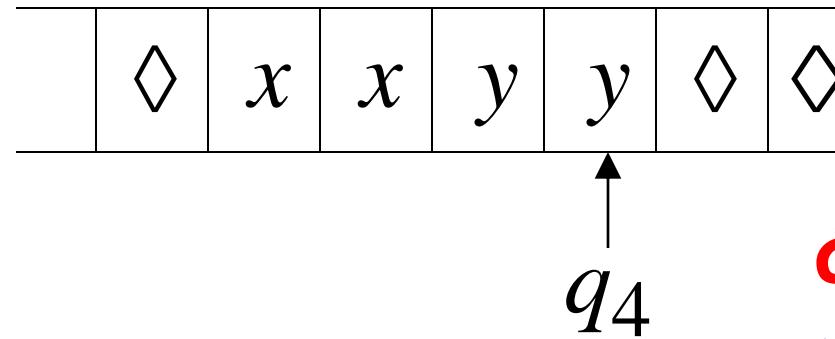
Time 11



Time 12



Time 13

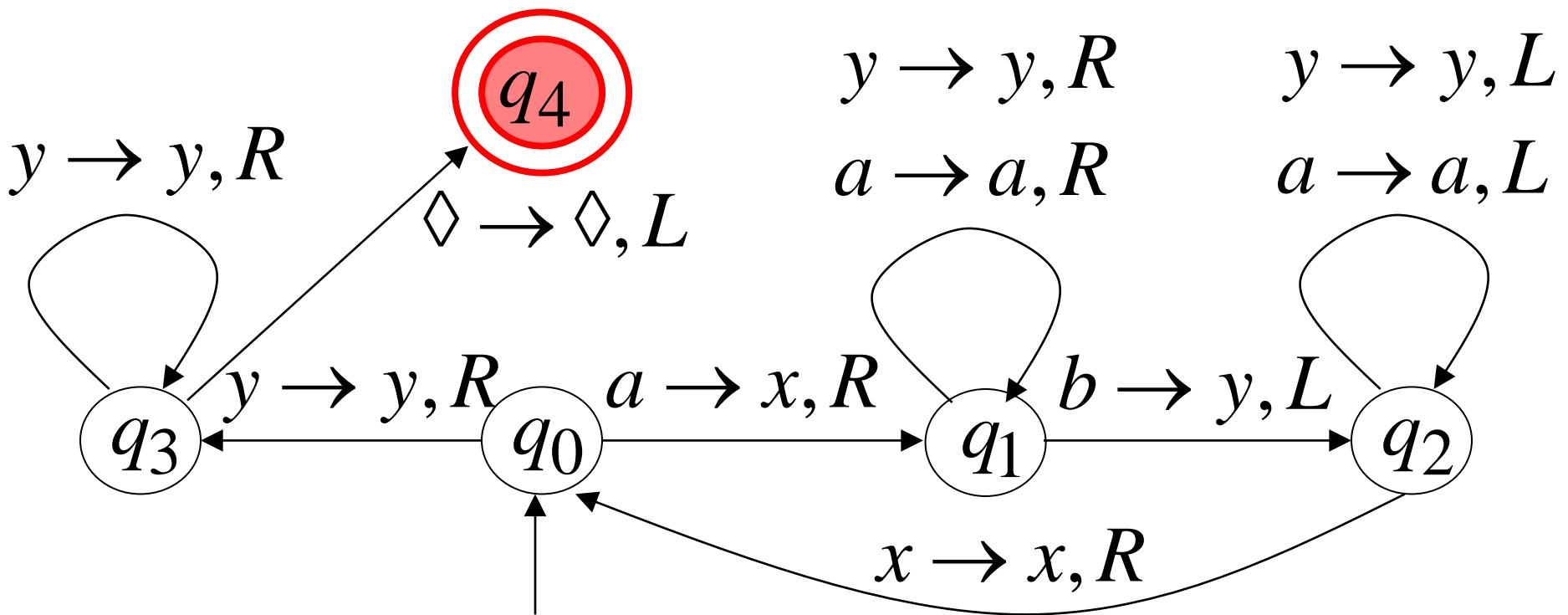


aabbba

xaybaa  $\rightarrow$  xxyba

xxyyaa

Halt & Accept



## Observation:

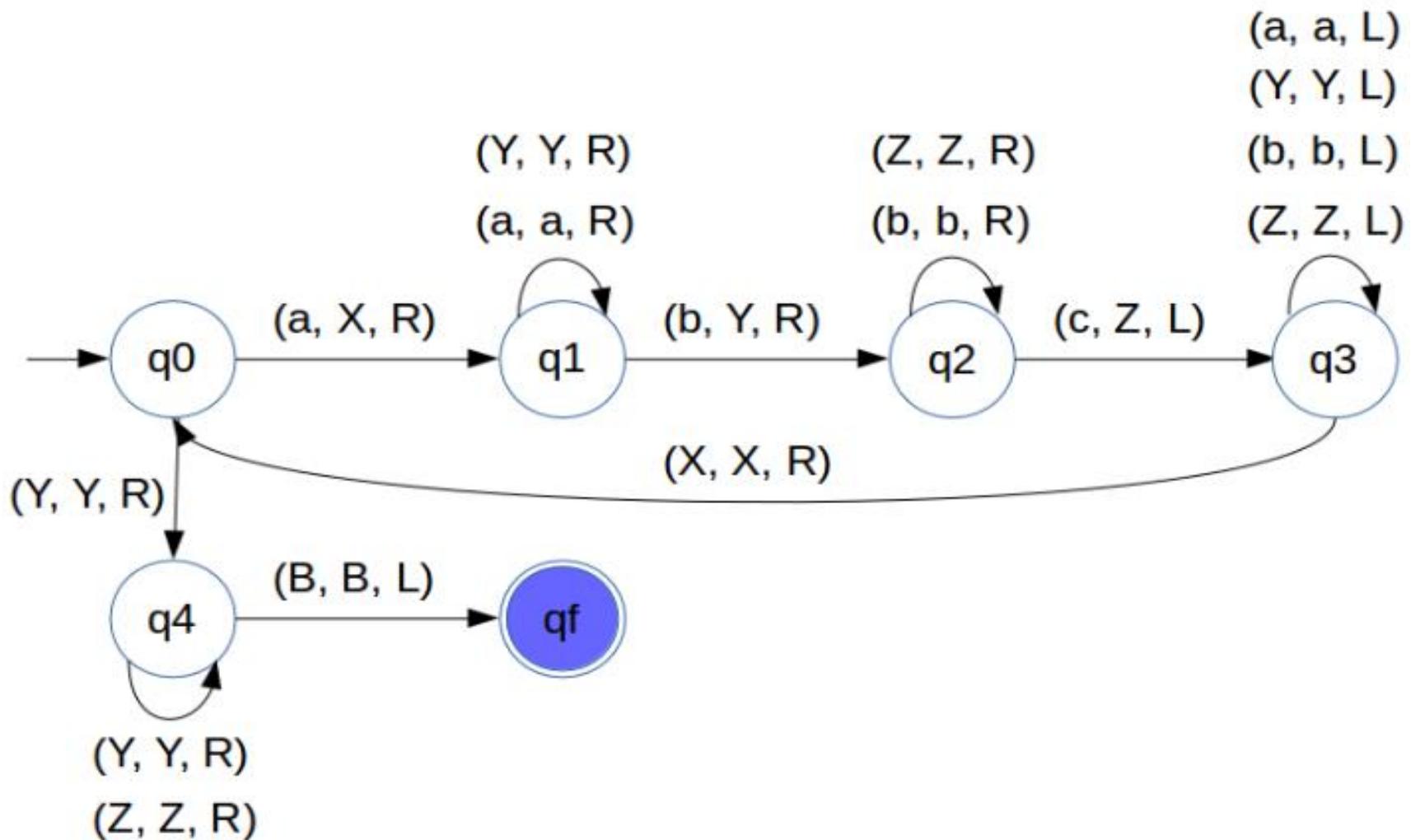
If we modify the Turing machine for the language

$$\{a^n b^n : n \geq 1\}$$

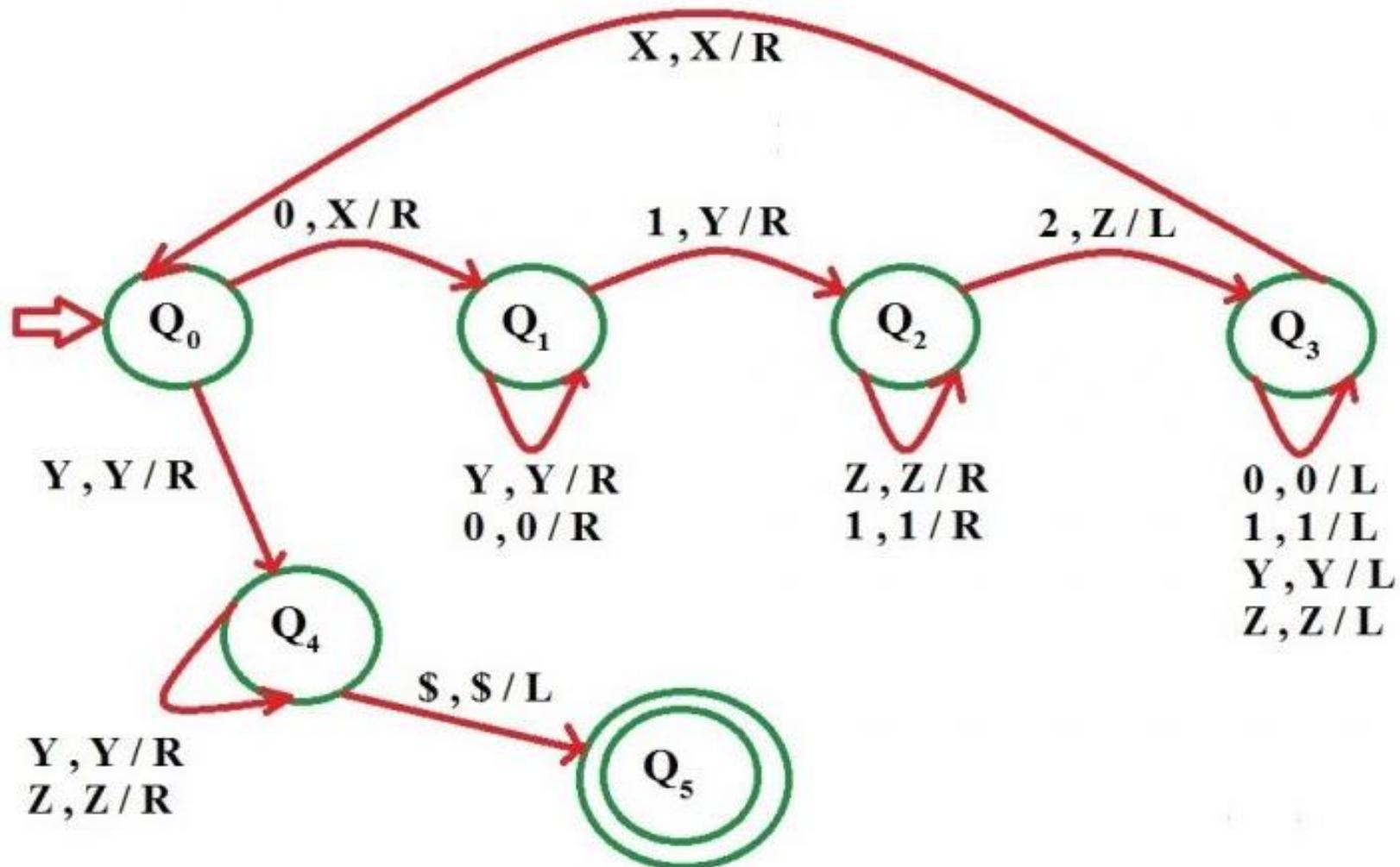
we can easily construct  
a TM for the language

$$\{a^n b^n c^n : n \geq 1\}$$

# Turing Machine for $\{a^n b^n c^n : n \geq 1\}$

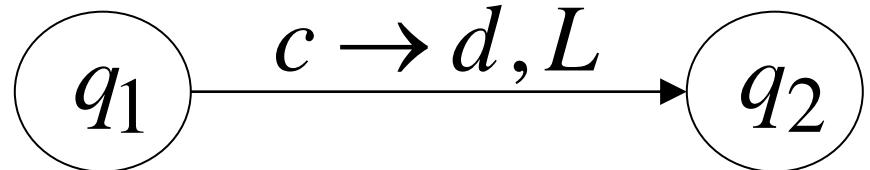


# Turing Machine for $\{0^n 1^n 2^n\}$

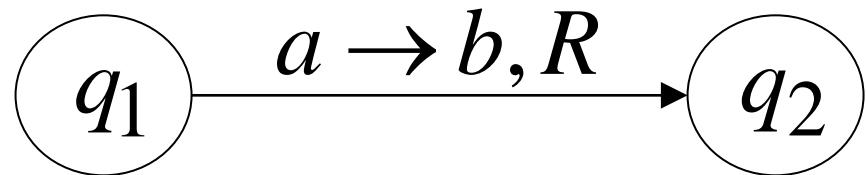


# Instantons Description for Turing Machines

# Transition Function

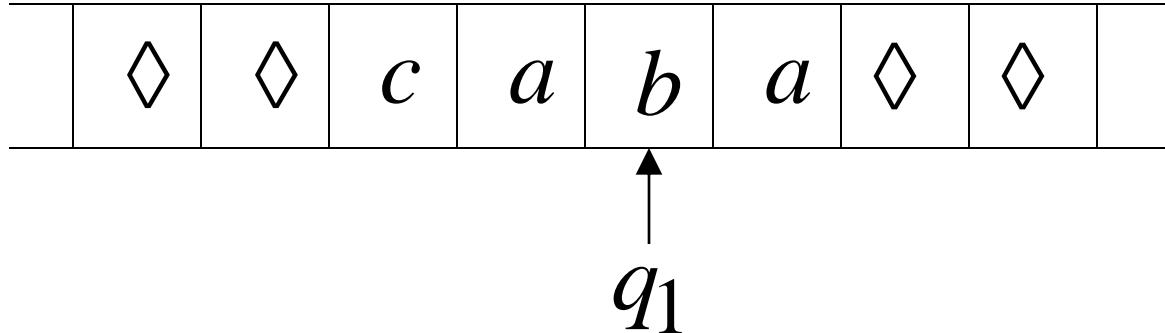


$$\delta(q_1, c) = (q_2, d, L)$$



$$\delta(q_1, a) = (q_2, b, R)$$

# Configuration



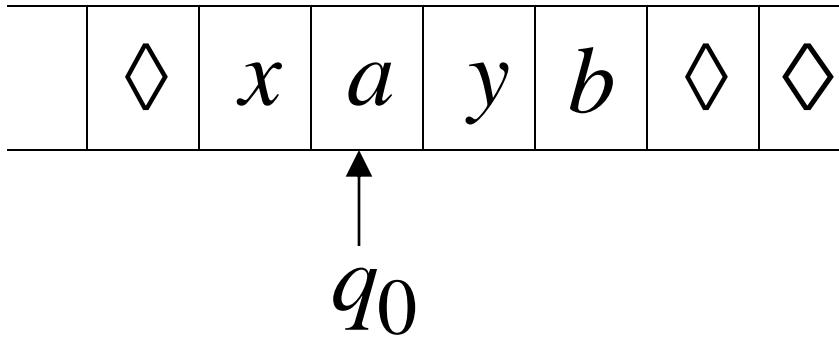
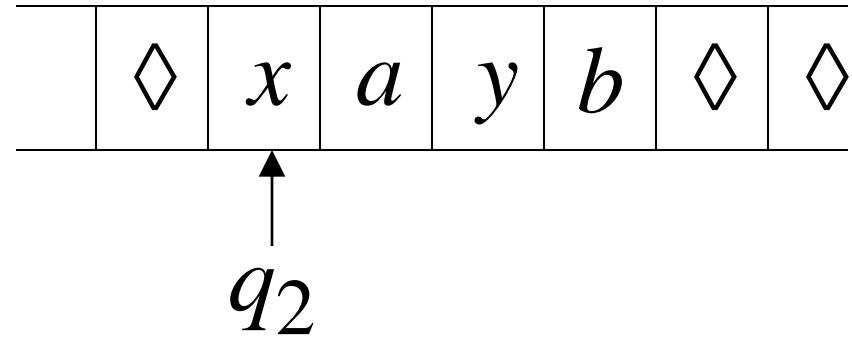
Instantaneous description:  $ca\ q_1\ ba$

Current State + contents of the tape  
+ position of read-write head

The unspecified part of the tape is assumed to contain all blanks.

Time 1

Time 2



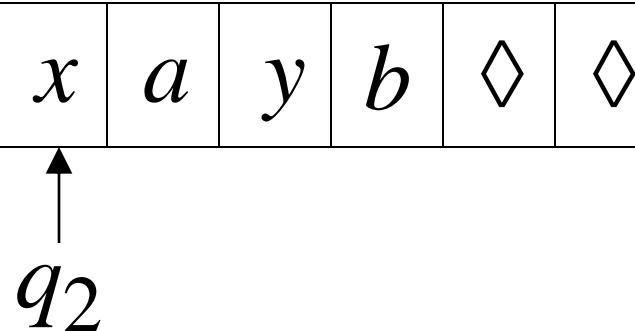
If  $\delta(q_2, x) = (q_0, x, R)$

Then,

$q_2 x a y b \vdash x q_0 a y b$

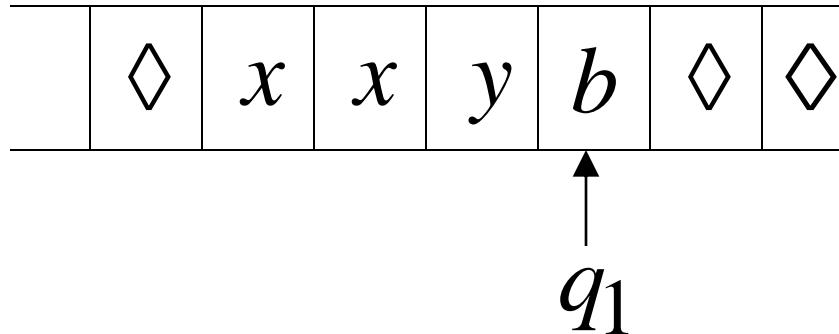
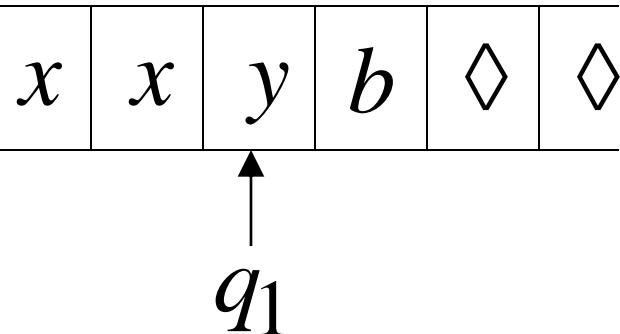
Time 1

Time 2



Time 3

Time 4



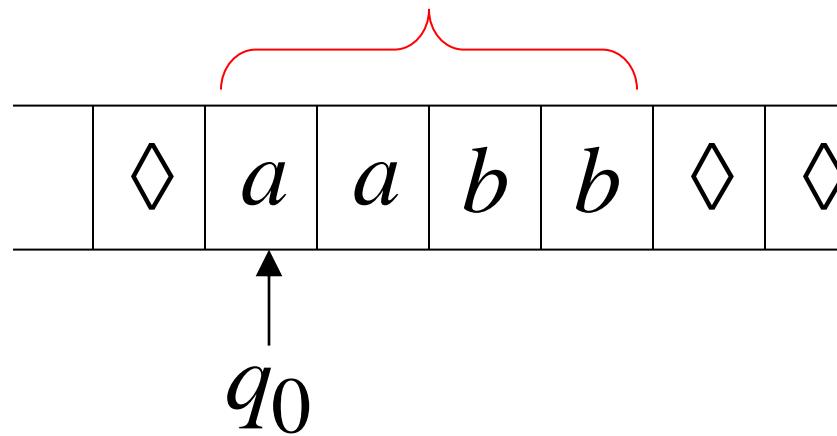
$q_2 x a y b \vdash x q_0 a y b \vdash x x q_1 y b \vdash x x y q_1 b$

Equivalent notation:

$q_2 x a y b \vdash^* x x y q_1 b$

Initial configuration:  $q_0 \ w$

Input string  $w$



The sequence of configurations leading to a halt state is known as computation.

$$q_0 aabb \xrightarrow{*} xxyq_4y$$

There is a possibility that a TM will never halt, proceeding in an endless loop from which it cannot escape

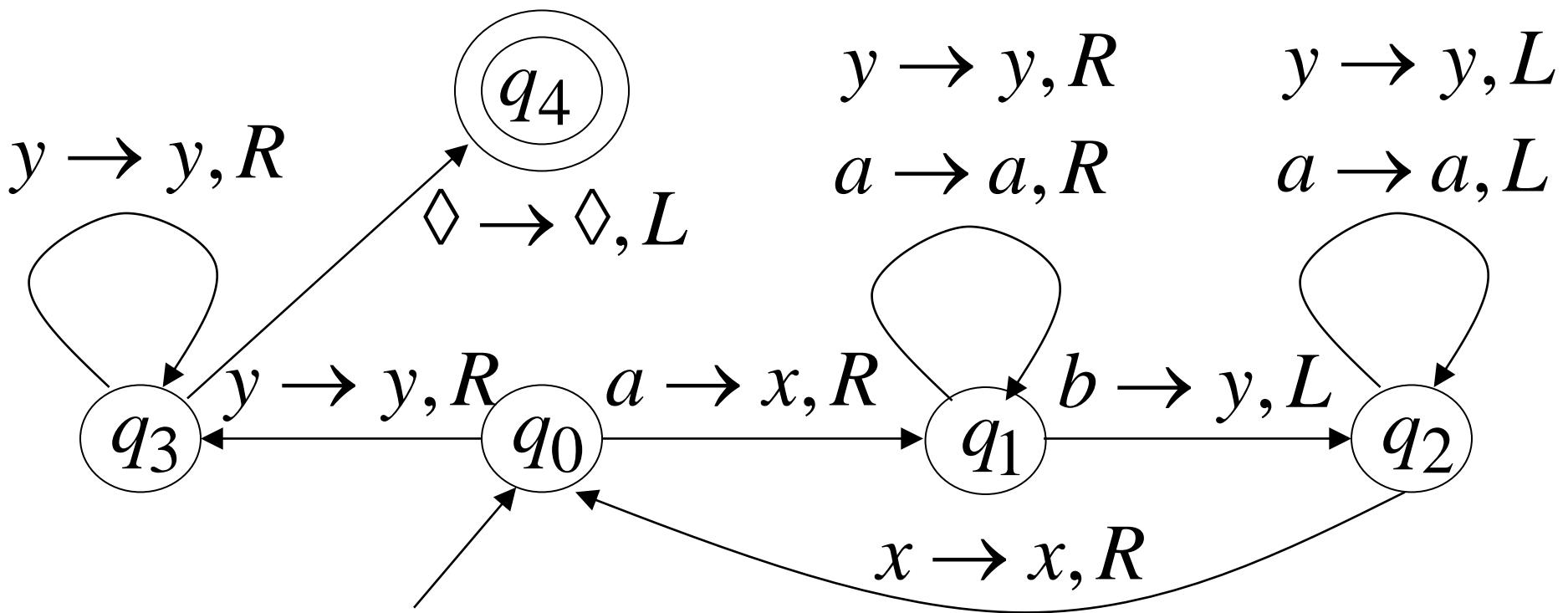
$$x_1 q_0 x_2 \xrightarrow{*} \infty$$

# Find computation

$q_0aaabb$  ----- ?

$q_0aabbb$  ----- ?

$q_0aaaabb$  ----- ?



# TM Language

Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$  be a Turing machine. Then, the language accepted by  $M$  is

$$L(M) = \{w \in \Sigma^*: q_0 w \xrightarrow{*} x_1 q_f x_2, q_f \in F, x_1, x_2 \in \Gamma^*\}$$

Initial state

Final state

$L(M)$  is the set of strings for which TM starts in initial state and after finite sequence of moves enters a final state and halts.

# Standard Turing Machine

The machine we described is the standard:

- Deterministic
- Infinite tape in both directions
- Tape is the input/output file

# Design TM

$L_1 = L(aba^*b)$

$L_2 = \{w : |w| \text{ is even}\}$

$L_3 = \{w : |w| \text{ is a multiple of 3}\}$

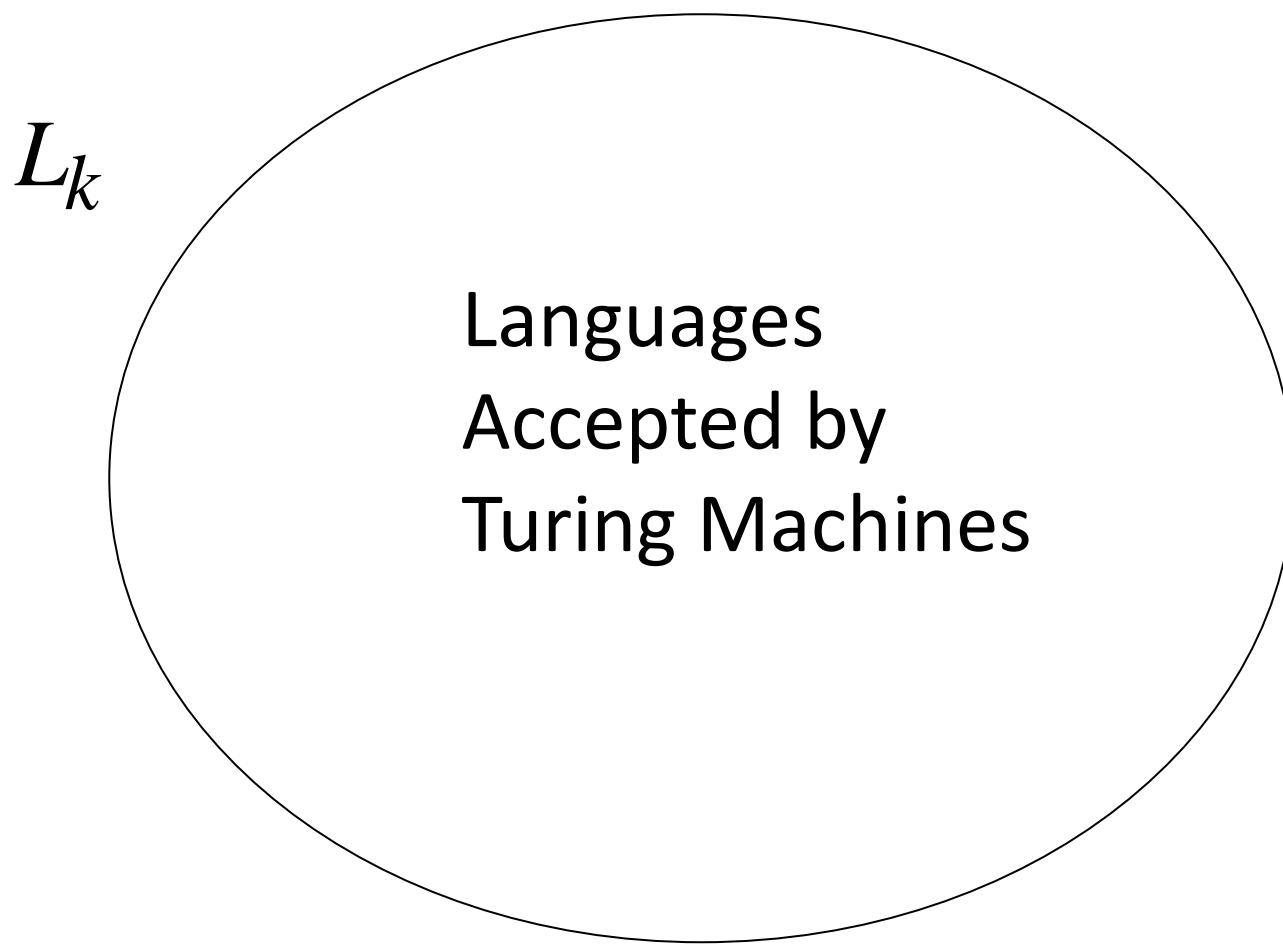
$L_4 = \{a^n b^{2n} : n \geq 1\}$

$L_5 = \{a^n b^m a^{n+m} : n \geq 1, m \geq 0\}$

$L_6 = \{a^n b^n a^n b^n : n \geq 0\}$

$L_7 = \{w : n_a(w) = n_b(w)\}$

# Languages not accepted by Turing Machines



# Restricted TM Models

## Pushdown Automata

A pushdown automaton is a Nondeterministic TM with a tape which we used like a stack.

Deterministic Finite Automata: A DFA is an Standard TM with a restrictions

It can only read the current tape symbol.

It is allowed to move in right direction only

It can't rewrite the current symbol.

# Restricted TM Models

Linear Bounded Automata (LBA):

A LBA is a standard Turing machine with restrictions.

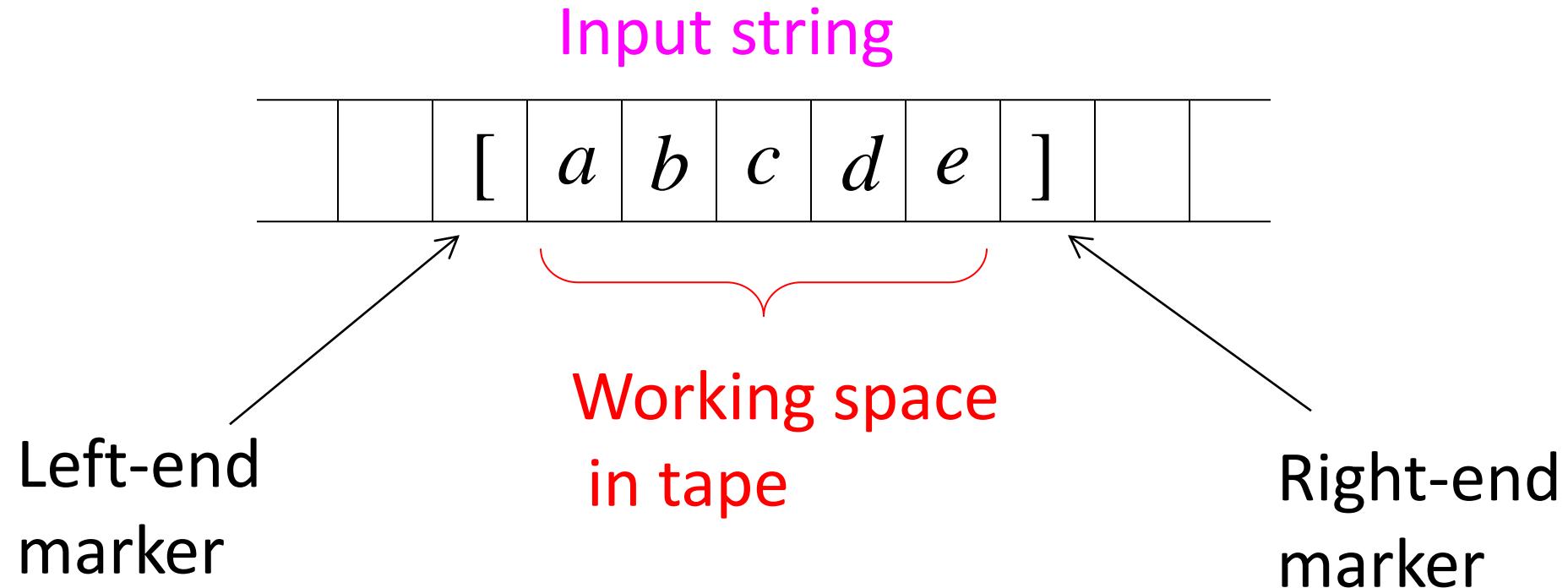
The allowed tape in LBA is restricted by the space (cells) required for all symbols in input string.

To restrict the tape, the input is bracketed by two special symbols, the left-end marker [ and the right-end marker ].

The end markers cannot be rewritten, and the read-write head cannot move to the left of [ or to the right of ].

For an input  $w$ , the initial configuration of LBA is given by  $q_0[w]$ .

# Linear Bounded Automaton (LBA)



All computation is done between end-markers

# Linear Bounded Automata

An LBA can be defined as  $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

$Q$ ,  $q_0$ ,  $B$ , and  $F$  are similar to TM.

$\Sigma = \Sigma \cup \{ [, ] \}$ ,  $\Gamma = \Gamma \cup \{ [, ] \}$

$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$  is a function called the transition function.

$\delta(q_1, [) = (q_2, [, R)$  allowed

$\delta(q_1, [) = (q_2, [, L)$  not- allowed

$\delta(q_1, ]) = (q_2, ], L)$  allowed

$\delta(q_1, ]) = (q_2, ], R)$  not- allowed

# The Accepted Language

Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$  be an LBA. Then the language accepted by  $M$  is

$$L(M) = \{w \in \Sigma^+ : q_0[w] \vdash^* [x_1 q_f x_2], \{q_f\} \cap F \neq \emptyset,$$

$\uparrow \quad x_1, x_2 \in \Gamma^* \}$

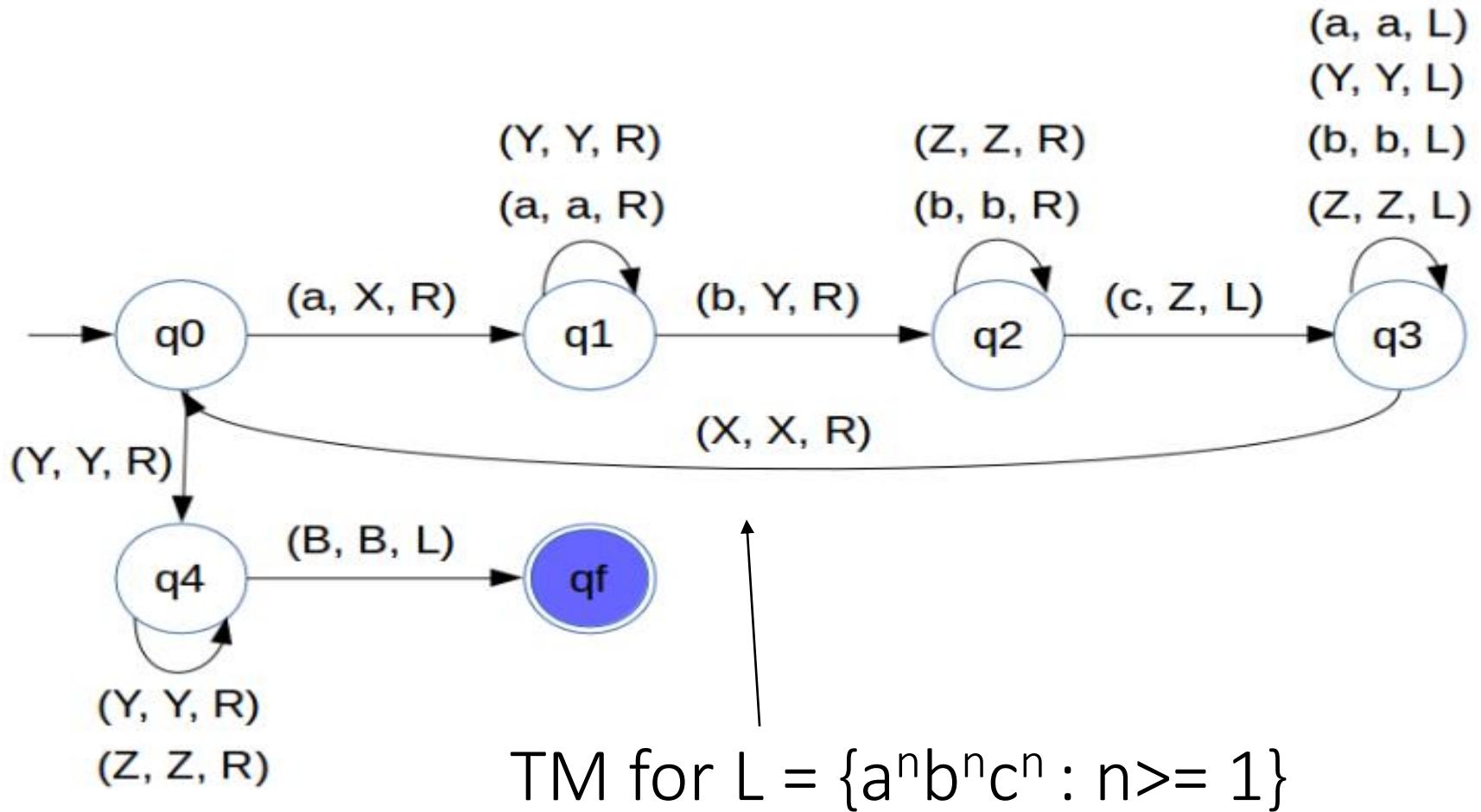
**Initial state**



Final state

$x_1, x_2 \in \Gamma^*$

# Design LBA for $L = \{a^n b^n c^n : n \geq 1\}$



# LBA

Following are standard example of LBA to remember

$$1. L = \{a^n b^n c^n \mid n \geq 1\}$$

$$2. L = \{a^{n!} \mid n \geq 0\}$$

$$3. L = \{a^n \mid n = m^2, m \geq 1\}, \text{ means } n \text{ is perfect square}$$

$$4. L = \{a^n \mid n \text{ is prime}\}$$

$$5. L = \{a^n \mid n \text{ is not a prime}\}$$

$$6. L = \{ww \mid w \in \{a, b\}^+\}$$

$$7. L = \{w^n \mid w \in \{a, b\}^+, n \geq 1\}$$

$$8. L = \{www^R \mid w \in \{a, b\}^+\}$$

Example languages accepted by LBAs:

$$L = \{a^n b^n c^n\}$$

$$L = \{a^{n!}\}$$

LBA's have more power than NPDA's

LBA's have also less power  
than Turing Machines

We define LBA's as Non-Deterministic

## **Open Problem:**

NonDeterministic LBA's  
have same power with  
Deterministic LBA's ?

# Recursively Enumerable (RE) Language

A language  $L$  is said to be recursively enumerable if there exists a Turing machine  $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$  that accepts every string  $w \in L$ .

$$L = \{a^n b^n c^n : n \geq 1\}$$

Accepts string means: It reaches to final state and halts.

But what happens to the string  $w \notin L$ .

It may halt in a non-final state.

It may move in an infinite loop.

## Recursive (REC) Languages

A language  $L$  on  $\Sigma$  is said to be recursive if there exists a Turing machine  $M$  that accepts  $L$  and that halts on every  $w$  in  $\Sigma^+$ .

For  $w \in L$ , It reaches to final state and halts.

For  $w \notin L$ , It reaches to non-final state and halts.

Every Recursive language is always Recursively Enumerable.

# Unrestricted Grammar

**Unrestricted Grammar:** The grammar in which restrictions are not imposed on production rules.

A grammar  $G = (V, T, S, P)$  is called unrestricted if all production rules are of the form

$$x \rightarrow y,$$

where  $x \in (V \cup T)^* V (V \cup T)^*$  and  $y \in (V \cup T)^*$

No restrictions except

$\lambda$  is not allowed on the left side of a production

The left side consists of at least one variable  
(Implicit)

# Unrestricted Grammar

$$L = \{a^{n+1}b^{n+k} : n \geq 1, k = -1, 1, 3, \dots\}$$

1.  $S \rightarrow S_1B,$
2.  $S_1 \rightarrow aS_1b$
3.  $bB \rightarrow bbbB,$
4.  $aS_1b \rightarrow aa$
5.  $B \rightarrow \lambda$

$$S \rightarrow S_1B \rightarrow aS_1bB \rightarrow aaS_1bbB \rightarrow aaabB \rightarrow aaabbB \rightarrow aaabbb$$

The unrestricted grammar corresponds to the largest family of languages that are **recognizable** by mechanical means.

The language generated by unrestricted grammar is known as recursively enumerable (RE) Language.

# Restricted Grammar

**Restricted Grammar:** The grammar in which restrictions are imposed on production rules.

Example:

Regular Grammar

Linear Grammar

Simple grammar

Context-Free Grammar

Context-Sensitive Grammar etc.

# Context-Sensitive Grammar

A grammar  $G = (V, T, S, P)$  is said to be context-sensitive if all productions are of the form

$$x \rightarrow y,$$

The production rule has one restriction

$$|x| \leq |y|$$

where  $x, y \in (V \cup T)^*$ .

The CS Grammar production rule is non-decreasing i.e. the length of the sentential form never decreases.

The language accepted by Context-Sensitive Grammar is known as Context-Sensitive Language.

# Context-Sensitive Grammar

$$L = \{a^n b^n c^n : n \geq 1\}$$

1.  $S \rightarrow abc \mid aAbc,$
2.  $Ab \rightarrow bA$
3.  $Ac \rightarrow Bbcc,$
4.  $bB \rightarrow Bb$
5.  $aB \rightarrow aa \mid aaA$

$S \rightarrow aAbc \rightarrow abAc \rightarrow abBbcc \rightarrow aBbbcc \rightarrow aaAbbcc$   
 $\rightarrow aabAbcc \rightarrow aabbAcc \rightarrow aabbBbccccc \rightarrow aabBbbccccc \rightarrow$   
 $aaBbbbcccc$   
 $\rightarrow aaabbccc$

If there is production  $xAy \rightarrow xuy$ , then  $xAy$  can be replaced by  $xuy$  i.e. it can be applied only in the situation where  $A$  occurs in a context of the string  $x$  on the left and the string  $y$  on the right. Hence, It is known as context-sensitive

# CS Language & LBA

For every context-sensitive language  $L$  not including  $\lambda$ , there exists some linear bounded automaton  $M$  that accepts it i.e.  $L = L(M)$ .

If a language  $L$  is accepted by some linear bounded automaton  $M$ , then there exists a context-sensitive grammar that generates  $L$ .

Every context-sensitive language  $L$  is recursive.

There exists a recursive language that is not context-sensitive.

# Chomsky Hierarchy

Noam Chomsky classifies languages into four language types, type 0 to type 3.

This classification exhibits the relationship between these language families.

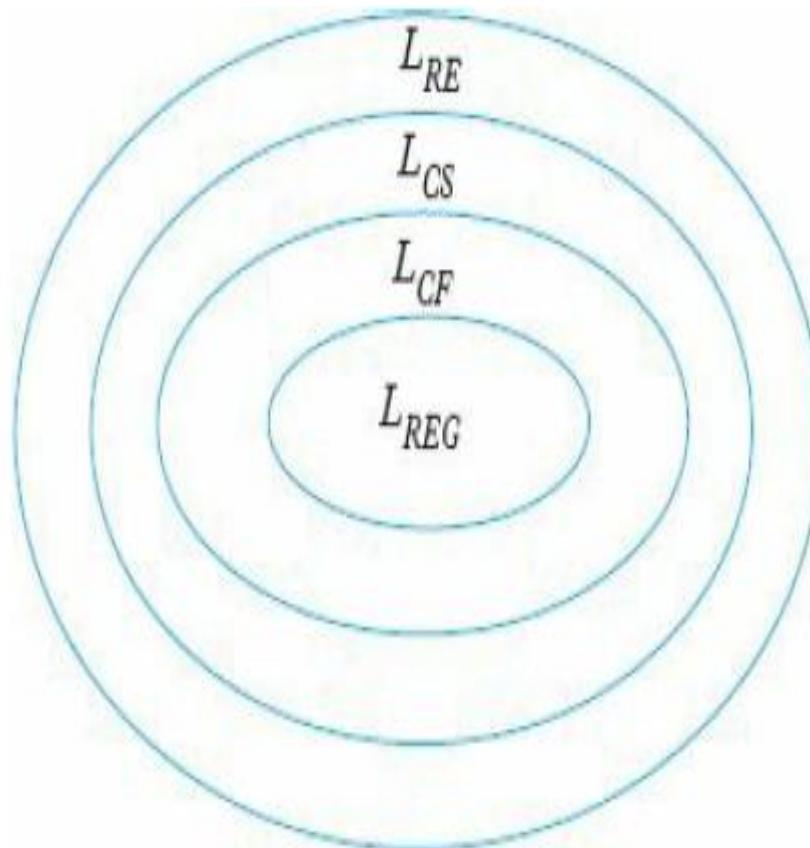
The set of all Regular languages is known as the family of Regular or Type 3 languages.

The set of all Context-Free languages is known as the family of Context-Free or Type 2 languages.

The set of all Context-Sensitive languages is known as the family of Context-Sensitive or Type 1 languages.

The set of all Recursively Enumerable languages is known as the family of Recursively Enumerable or Type 0 languages.

# Chomsky Hierarchy



# The Chomsky Hierarchy

Non-recursively enumerable

Recursively-enumerable

Recursive

Context-sensitive

Context-free

Regular

# Chomsky Hierarchy

$L_1 = \{w : w \in \{a, b\}^*, |w| \text{ is even}\}$  is regular language.

There exists some languages which are not regular.

Example:  $L_2 = \{a^n b^n : n \geq 0\}$  not Regular

- Every regular language is also Context-Free.

There are some languages which are not CFL

Example:  $L = \{a^n b^n c^n : n \geq 1\}$  not CFL

$L = \{ww : w \in \{a, b\}^*\}$  not CFL

- Every context-free language is also context-sensitive.

# Computational Complexity & Decidability

# Computational Complexity

There are some languages which are not Context Sensitive.

The Context-sensitive languages are either **NSPACE** ( $O(n)$ ) or **DSPACE** ( $O(n)$ ).

A language is known as **NSPACE**( $O(n)$ ), if it is accepted using linear space on a Non-deterministic Turing machine.

A language is known as **DSPACE**( $O(n)$ ), if it is accepted using linear space on a Deterministic Turing machine.

Clearly, **DSPACE** is a subset of **NSPACE**, but it is not known whether **DSPACE**=**NSPACE**.

# Computational Complexity

In complexity theory, 'EXPSPACE' is the set of all decision problems solvable by a deterministic Turing machine in  $O(2^{p(n)})$  space, where  $p(n)$  is a polynomial function of  $n$ .

Clearly, NSPACE is a subset of EXASPACE.

Every context-sensitive language is also unrestricted language

Therefore, each language family of Type  $i$  is a proper subset of the family of Type  $i-1$ .

# Decidability/Undecidability

**Decidable:** A problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

**Un-decidable:** A problem is undecidable if there is no Turing Machine that solves all instances of the problem

# Decidable Problem

$A_{DFA} = \{ (D, w) : D \text{ is DFA that accepts input string } w \}$

$A_{NFA} = \{ (N, w) : N \text{ is NFA that accepts input string } w \}$

$E_{DFA} = \{ (A) : A \text{ is DFA and } L(A) = \emptyset \}$

$EQ_{DFA} = \{ (A, B) : A \text{ and } B \text{ are DFAs and } L(A) = L(B) \}$

$A_{CFG} = \{ (G, w) : G \text{ is CFG that generates string } w \}$

$E_{CFG} = \{ (G) : G \text{ is CFG and } L(G) = \emptyset \}$

# Un-decidable Problem (Unsolvable Problems)

The problem is to determine for an arbitrary TM  $M$  and an arbitrary input string  $w$  whether  $M$  with input  $w$  halts or not.

$$A_{TM} = \{ (M, w) : M \text{ is TM and } M \text{ halts on string } w \}$$

Given a Turing machine  $M$ , we ask if we can tell whether it accepts the empty string or not.

$$E_{TM} = \{ (M) : M \text{ is TM and } L(M) = \emptyset \}$$

$$A_{CFG} = \{ (G) : G \text{ is CFG and } G \text{ is ambiguous } \}.$$

# Unsolvable Problems: TM Halting Problem

Given the description of a Turing machine  $M$  and an input  $w$ , does  $M$ , when started in the initial configuration  $q_0w$ , perform a computation that eventually halts?

Can we make a TM that can determine if another TM will accept a string  $w$ ?

The problem is the TM might get stuck in a loop and go forever; we could simulate the original TM, but if it loops, we are stuck.

➤ Is the halting problem solvable?

$\text{HALT}_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on input } w \}$

So the question we need to answer is

Is  $\text{HALT}_{\text{TM}}$  decidable? NO

# Some Unsolvable Problems

## OProgram Halting Problem

OProblem: Given a program  $A$ , and input to program  $I$ , does  $A$  halt if given input  $I$ ?

OIn other words, we want an algorithm  $H$  such that, given any program  $A$  and input  $I$ :

- $H(A, I) = \text{true}$  if  $A(I)$  halts (finishes), and
- $H(A, I) = \text{false}$  if  $A(I)$  will never halt (loops forever)

# Some Unsolvable Problems

## ○ Software Verification Problem

You are given a computer program and a precise specification of what that program is supposed to do (e.g., sort a list of numbers).

You need to verify that the program performs as specified (i.e., that it is correct).

The general problem of software verification is not solvable by computer.

# The Halting Problem

Input:

- Turing Machine M

- String w

Question:

Does M halt on input w?

The problem is to determine for an arbitrary TM M and an arbitrary input string w whether M with input x halts or not.

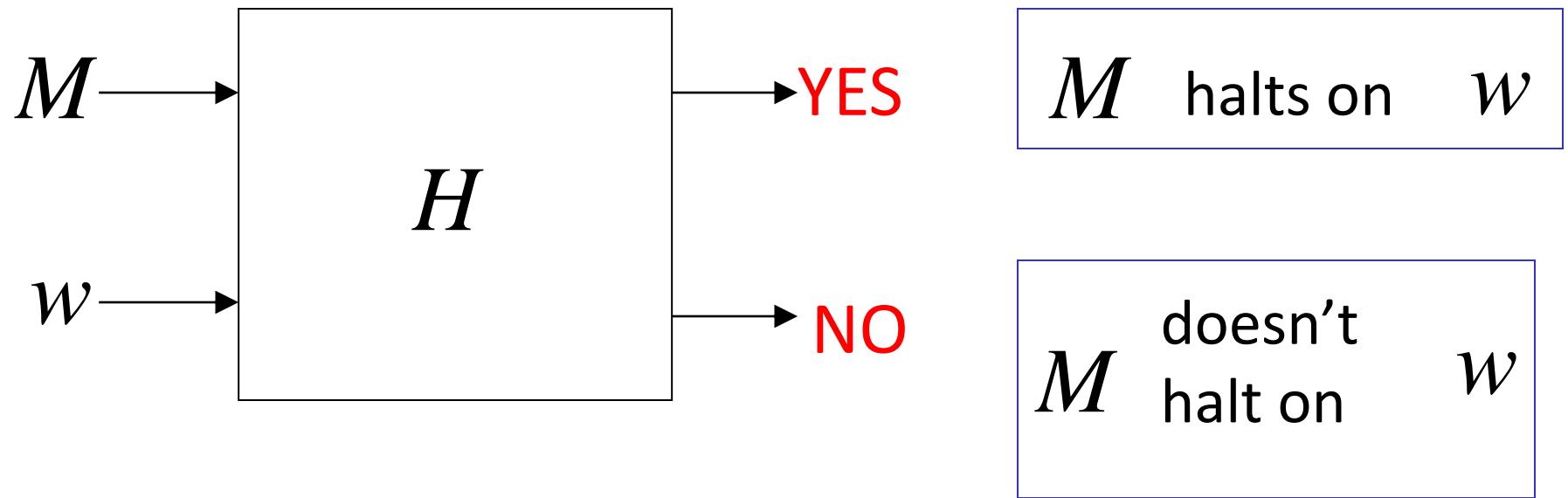
$$HALT_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM that halts on input } w \}$$

**Theorem:**

The halting problem is undecidable

**Proof:** Assume for contradiction that the halting problem is decidable

There exists Turing Machine  $H$   
that solves the halting problem



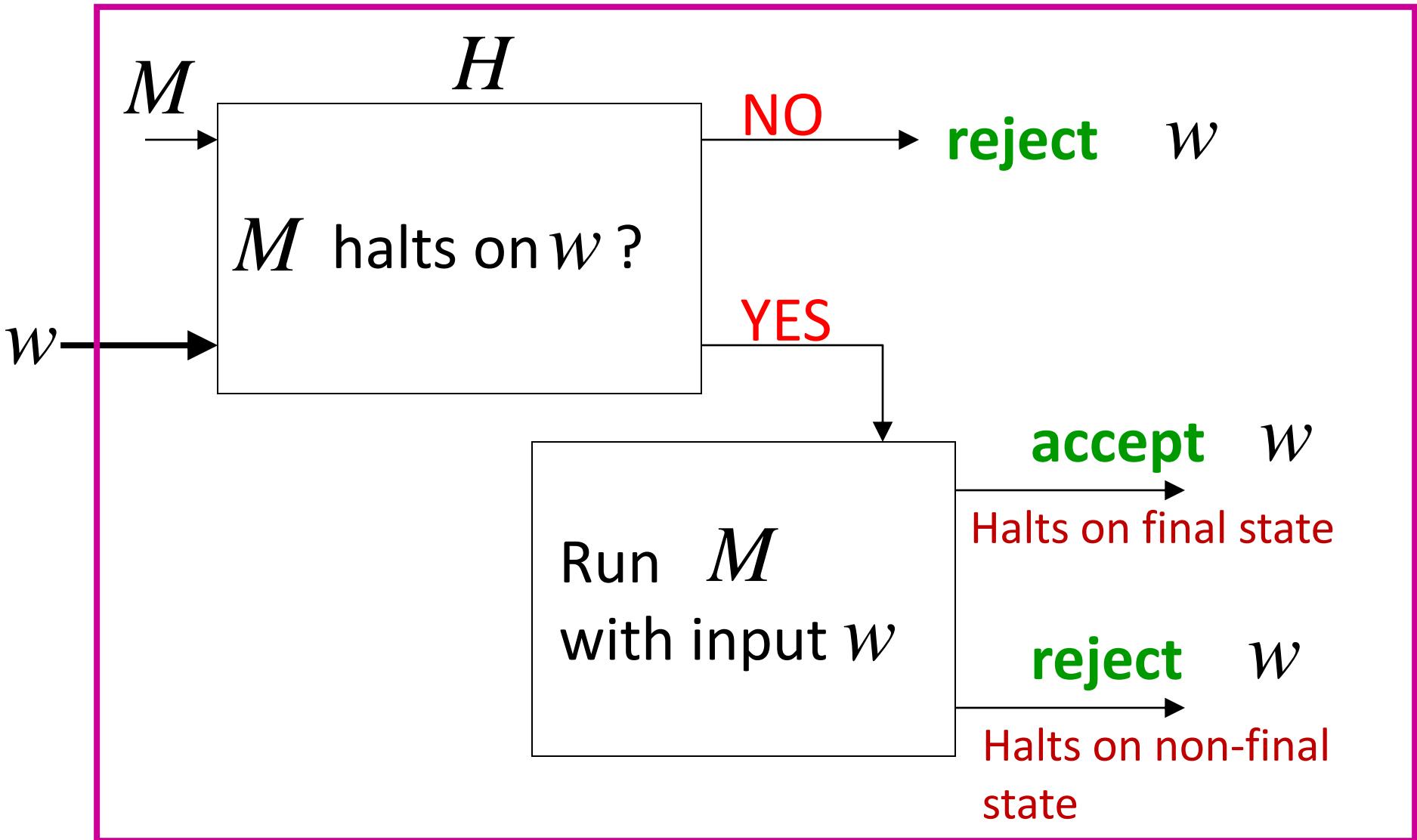
$M$  is arbitrary Turing Machine.

Let  $L$  be a recursively enumerable language accepted by  $M$

We will prove that  $L$  is also recursive:

we will describe a Turing machine that accepts  $L$  and halts on any input

# Turing Machine that accepts L and halts on any input



Therefore,  $L$  is recursive

Since  $L$  is chosen arbitrarily, every recursively enumerable language is also recursive

But there are recursively enumerable languages that are not recursive

**Contradiction!!!!**

Therefore, the halting problem is undecidable

# Turing's Thesis Conjecture/hypothesis

(1930)

Any computation carried out by mechanical means can also be performed by a Turing Machine.

“Any computation that can be carried out by mechanical means/**present-day computers** can also be performed by some Turing Machine”

# Arguments in favor of Turing's Thesis

- Anything that can be done on any existing digital computer can also be done by some Turing machine.
- **No one has yet been able to suggest a problem**, solvable by what we intuitively consider an algorithm, for which a **TM program cannot be written**.
- Alternative models have been proposed for mechanical computation, but none of them is more powerful than TM model.

# Definition of Algorithm:

An algorithm for function  $f(w)$   
is a

Turing Machine which computes  $f(w)$

**Recall** A function  $f$  is **computable** if there exists some Turing Machine  $M$  such that:

$$q_0 w \vdash^* q_f f(w), \quad q_f \in F \quad \text{For all } w \in D$$

# Algorithms are Turing Machines

When we say:

There exists an algorithm

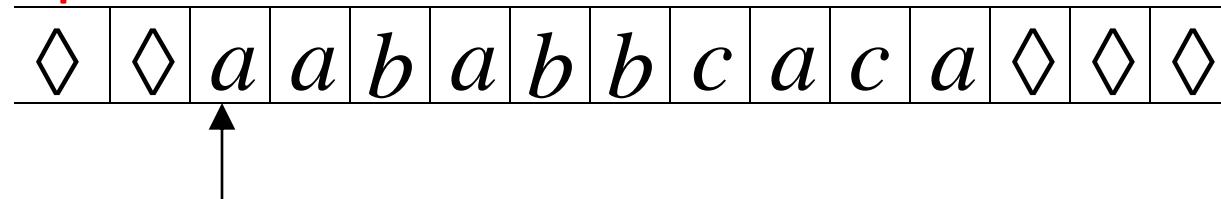
We mean:

There exists a Turing Machine  
that executes the algorithm

# Turing Machine Variants

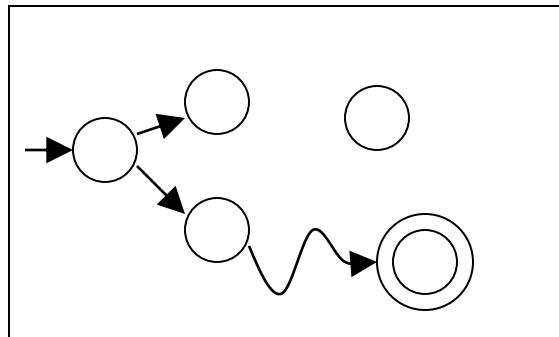
# The Standard Model

Infinite Tape



Read-Write Head (Left or Right)

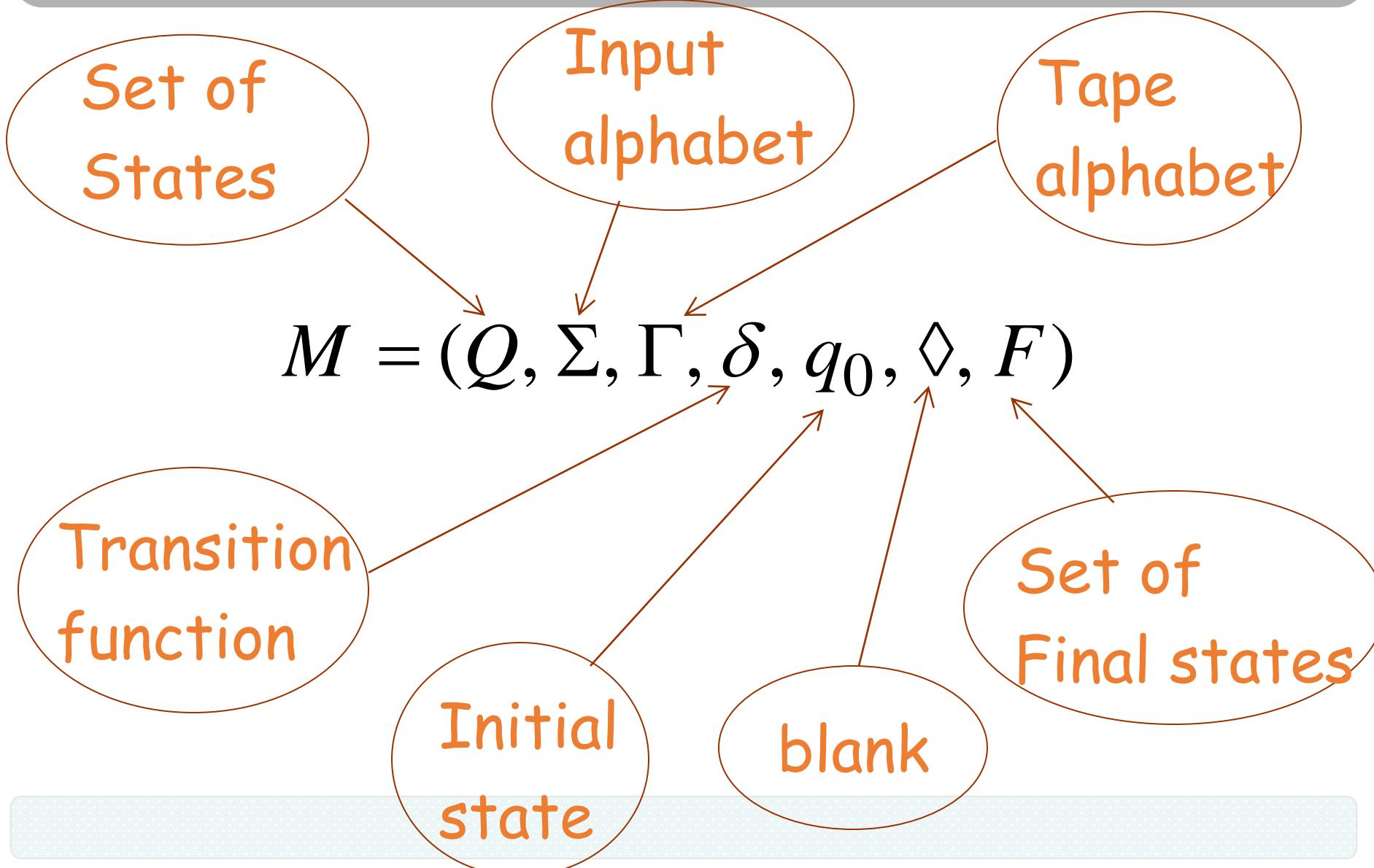
Control Unit



Deterministic

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$$

# Standard Turing Machine



# TM Variants

- Standard Turing Machines (TM) may not be necessarily efficient, as it may require large number of states for simulating even simple behavior.
- Modifications for improvements are feasible by
  - Increasing move options;
  - Increasing R/W Heads;
  - Increasing number of tapes;
  - Enhancing tape dimensions;
  - Adding special purpose memory such as queue, stack etc.
- These enhancements may speed up and ease the computation.
- These extensions do not add anything extra to the power of standard TM.

# Variations of the Standard Model

Turing machines with:

- Stay-Option
- Semi-Infinite Tape
- Off-Line
- Multitape
- Multidimensional
- Nondeterministic

# Equivalence

We want to prove:

Each **Class** has the same power with the **Standard Model**

Same Power of two classes means:

- Both classes of Turing machines accept the same languages.

# Same Power of two classes

Consider two classes of automata  $C_1$  and  $C_2$ .

If for every automation  $M_1$  of first class  $C_1$

There is an automation  $M_2$  of second class  $C_2$

such that:  $L(M_1) = L(M_2)$

And vice-versa

## TM with Stay Option

- A TM with Stay Option is like an ordinary TM but the read-write head can stay in place after rewriting the cell content.

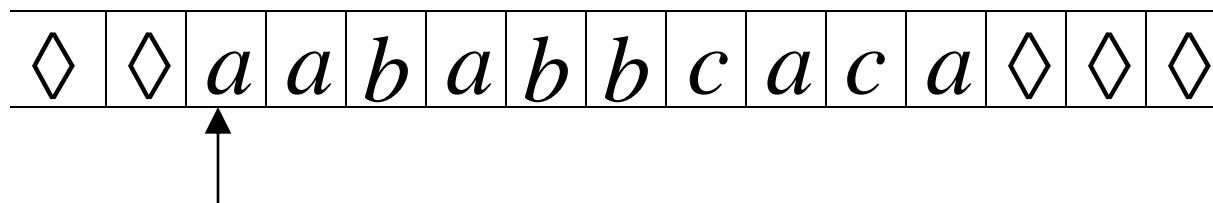
It can be defined as  $M = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$   
 $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$

The possible transitions are:

- $\delta(q_i, a) = (q_j, b, L)$  or
- $\delta(q_i, a) = (q_j, b, R)$  or
- $\delta(q_i, a) = (q_j, b, S)$

# Turing Machines with Stay-Option

The head can stay in the same position

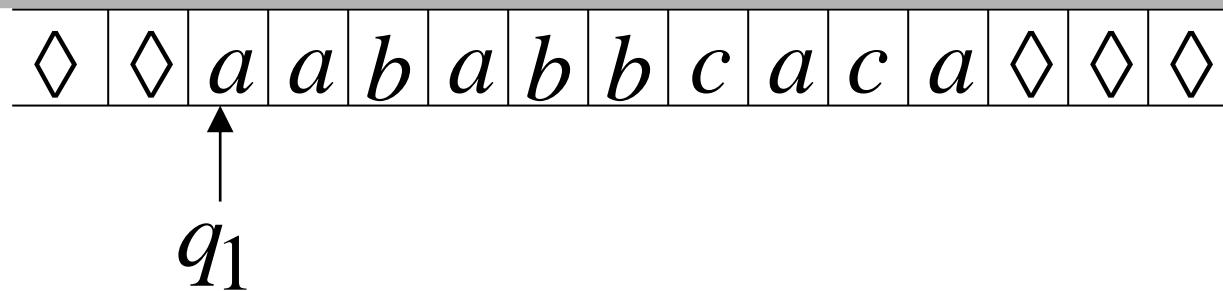


Left, Right, Stay

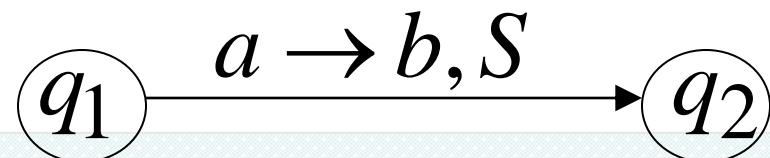
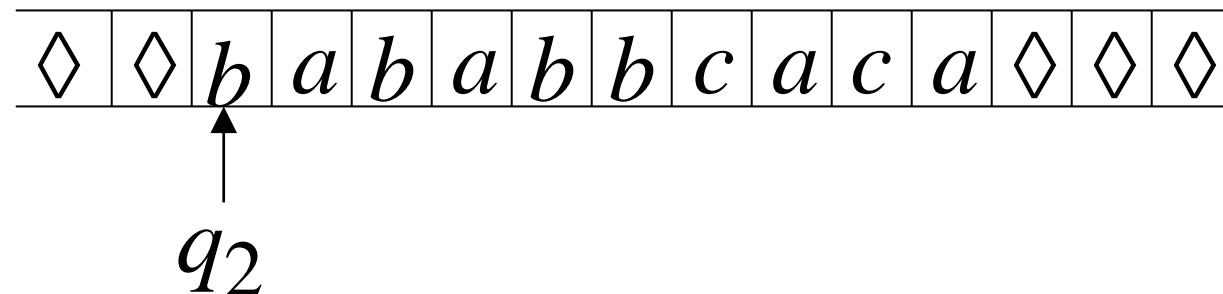
L,R,S: moves

Example:

Time 1



Time 2



# Stay-option TM is equivalent to Standard TM

## Theorem:

Stay-Option Turing Machines  
have the same power with  
Standard Turing machines

## Proof:

**Part 1:** Standard Machines  
are at least as powerful as  
Stay-Option machines

**Proof:** A standard TM can simulate a  
Stay-Option TM

# Stay-option TM is equivalent to Standard TM

## Theorem:

Stay-Option Turing Machines have the same power with Standard Turing machines

## Proof:

Part 2: Stay-Option Machines are at least as powerful as Standard machines

Proof: A stay-option TM can simulate Standard TM.

(A stay-option TM; that never uses the S move; can simulate Standard TM.)

## Part 1: A standard TM can simulate a Stay-Option TM

o A Stay-Option Turing machine can be simulated by standard Turing Machine.

o Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$  be a stay-option Turing machine.  
The transitions of may be any of the followings

- $\delta(q_i, a) = (q_j, b, L)$  or
- $\delta(q_i, a) = (q_j, b, R)$  or
- $\delta(q_i, a) = (q_j, b, S)$

o Let  $M' = (Q', \Sigma, \Gamma, \delta', q'_0, B, F')$  be a Standard Turing machine.

- For all  $q_i \in Q$ ,  $q'_i \in Q'$
- $q'_0 = q_0$
- For all  $q_f \in F$ ,  $q'_f \in F'$

## Part 1: A standard TM can simulate a Stay-Option TM

- For each move of M, the simulating machine M' does the following.
- If the move of M does not involve the stay-option, the simulating machine M' performs one move, essentially identical to the move of M.
  - For each transition of M,  $\delta(q_i, a) = (q_j, b, L)$ .
  - **The transition of M' will be  $\delta'(q'_i, a) = (q'_j, b, L)$**
- For each transition of M,  $\delta(q_i, a) = (q_j, b, R)$ .
- **The transition of M' will be  $\delta'(q'_i, a) = (q'_j, b, R)$**

## Part 1: A standard TM can simulate a Stay-Option TM

If the stay option S is involved in the move of M, then M' will make **two moves**:

- The first move rewrites the symbol and moves the read-write head in right direction;
- The second (move) moves the read-write head left, leaving the tape contents unaltered.
  
- For each transition,  $\delta(q_i, a) = (q_j, b, S)$ . The standard Turing machine will have two moves

$$\delta'(q'_i, a) = (q'_k, b, R)$$

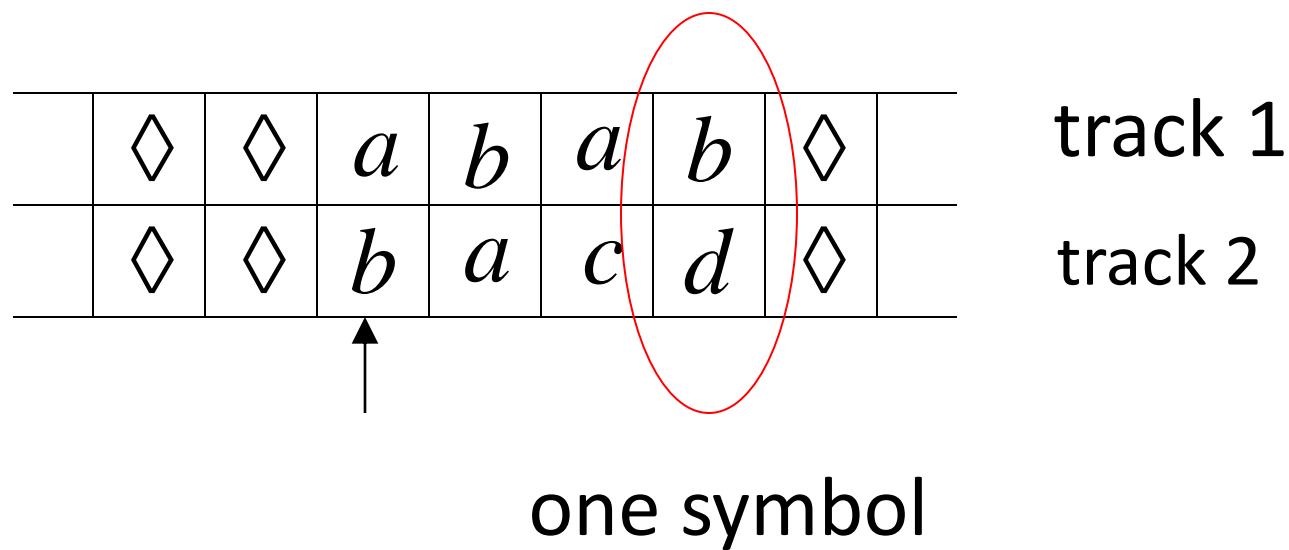
$$\delta'(q'_k, c) = (q'_j, c, L)$$

For all  $c \in \Gamma$  and  $Q' = Q \cup \{q'_k\}$ .

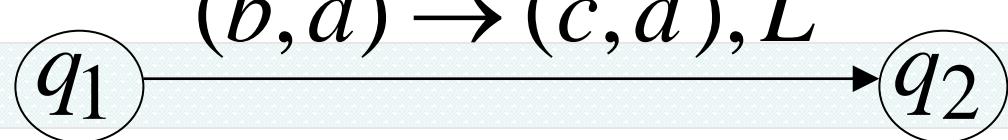
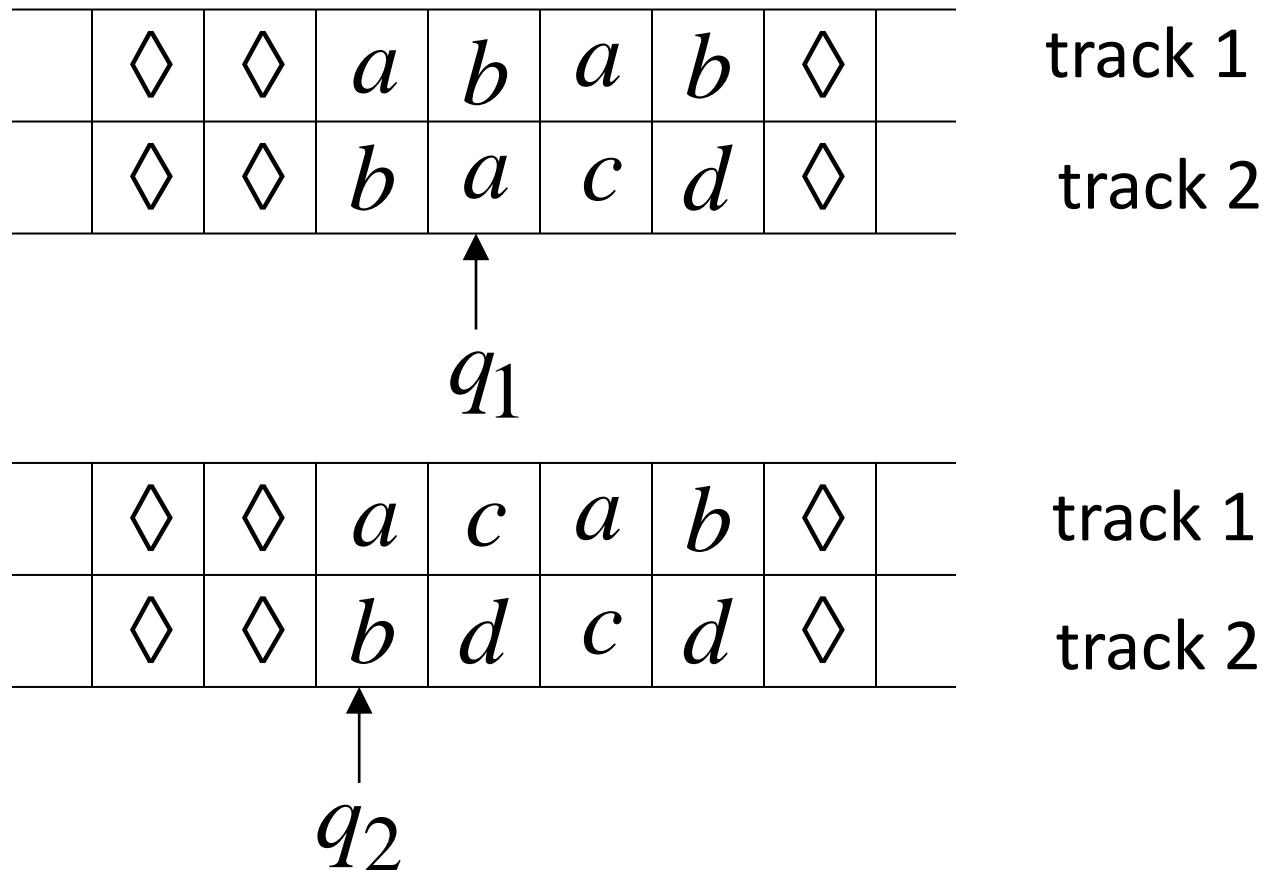
In this way, every transition of M can be simulated by the M'.

Hence, It is obvious that every computation of M has a corresponding computation of M'.

# Standard Machine--Multiple Track Tape



# Standard Machine--Multiple Track Tape



# Multi-tape TM

- A multi-tape TM is like a standard TM **but it has several tapes instead of one tape each with its own independently controlled read-write head.**

◦ An n-Tape Turing machine can be defined as  $\mathbf{M} = (Q, \Sigma, \Gamma, \delta, q_0, \diamond, F)$

$$\delta: Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{L, R\}^n$$

- The transition function is changed to allow for reading, writing, and moving **the heads on all the tapes simultaneously**.
- It can read on multiple tapes and move in different directions on each tape as well as write a different symbol on each tape, all in one move.

# 2-Tape TM

- In 2-tape Turing machine,  $n = 2$ .

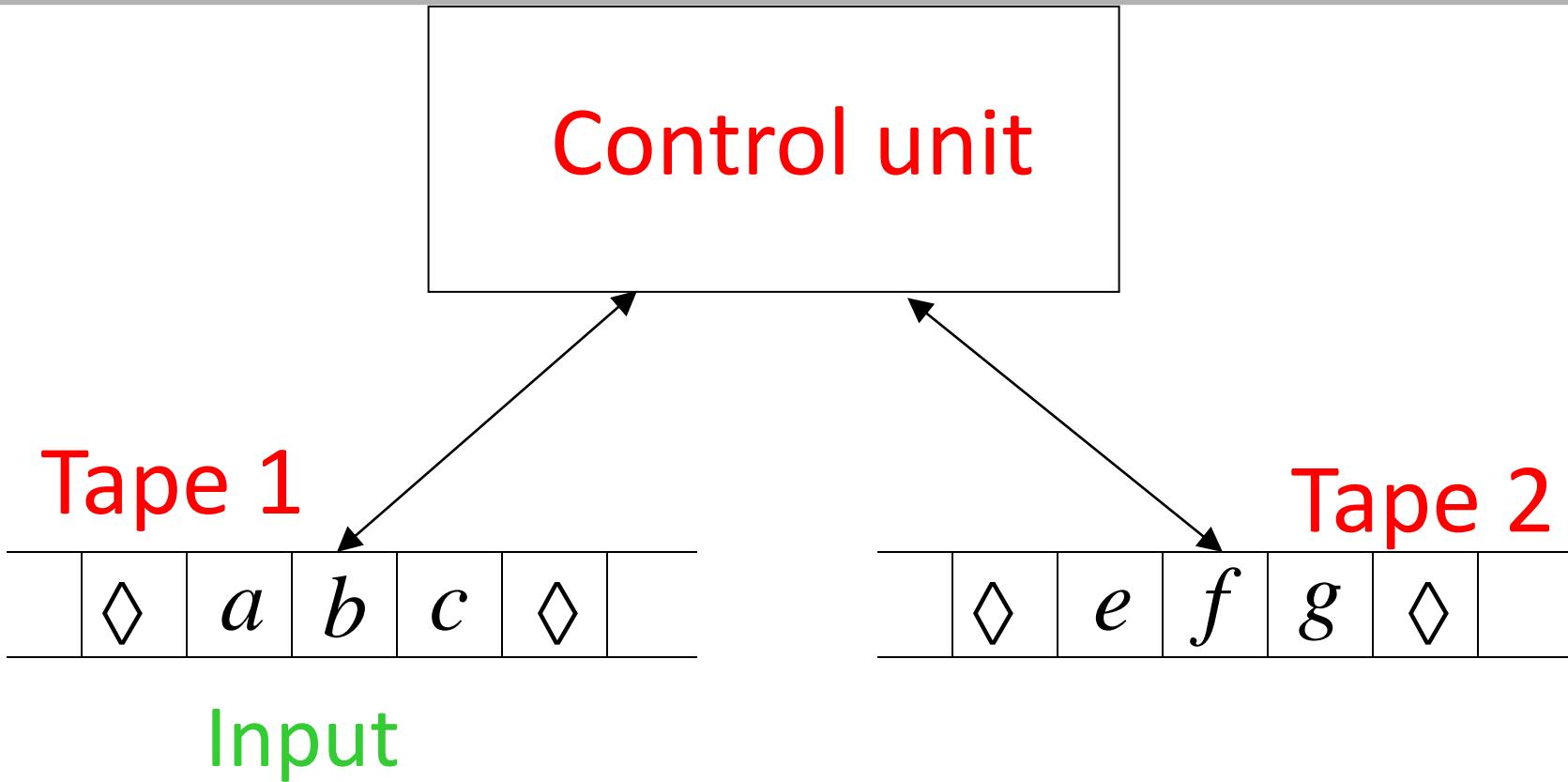
- The transition is given as follows

$$\delta: Q \times \Gamma^2 \rightarrow Q \times \Gamma^2 \times \{L, R\}^2$$

- One of the transition rule will be

$$\delta(q_0, a, e) = (q_1, x, y, L, R)$$

# 2-Tape Turing Machines



Tape 1

Time 1

Tape 2

	◊	a	b	c	◊	
--	---	---	---	---	---	--

$q_1$

	◊	e	f	g	◊	
--	---	---	---	---	---	--

$q_1$

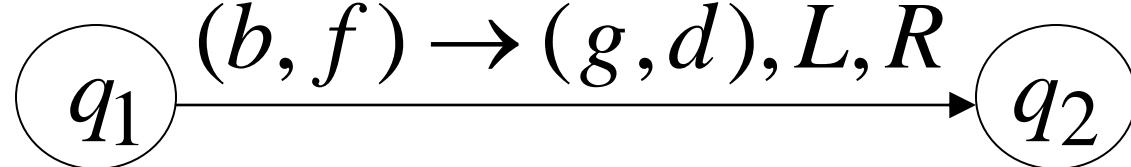
Time 2

	◊	a	g	c	◊	
--	---	---	---	---	---	--

$q_2$

	◊	e	d	g	◊	
--	---	---	---	---	---	--

$q_2$



# Multi-tape TM is equivalent to Standard TM

Multi-tape TMs simulate  
Standard TMs.

Use just one tape

Standard TMs  
simulate  
Multi-tape TMs:

- Use a TM with multi-track tape
- A tape of the Multiple tape machine corresponds to a pair of tracks

# Simulation of 2-tape TM by Standard TM

- A 2-Tape Turing machine can be simulated by standard Turing Machine.

- Let  $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$  be 2-tape TM.

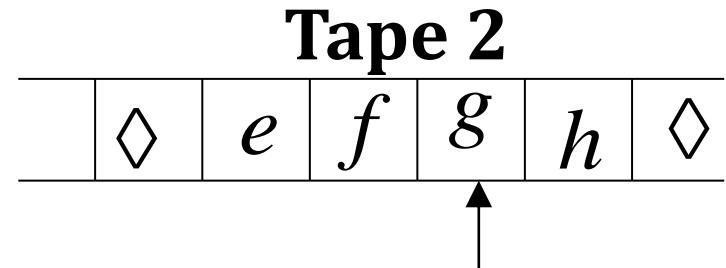
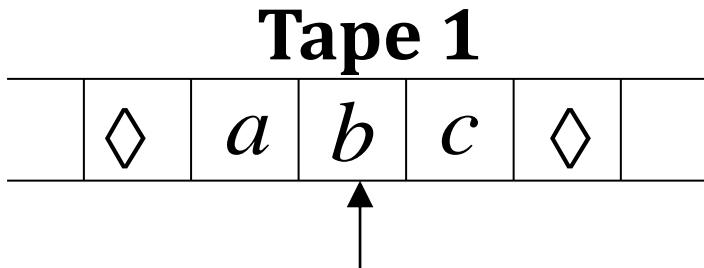
One of the transition of this machine is given by

$$\delta(q_0, a, e) = (q_1, x, y, L, R)$$

- Let  $M' = (Q', \Sigma, \Gamma, \delta', q'_0, B, F')$  be a Standard Turing machine. The single tape is divided **into four tracks**.

- The first track represents the contents of tape 1 of M.
- The nonblank part of the second track has all zeros, except for a single 1 marking the position of M's read-write head.
- The third track represents the contents of tape 2 of M.
- The nonblank part of the fourth track has all zeros, except for a single 1 marking the position of M's read-write head.

# Simulation of 2-tape TM by Standard TM



**Standard machine with four tracks tape**

		a	b	c			
		0	1	0			
		e	f	g	h		
		0	0	1	0		

**Tape 1  
head position**

**Tape 2  
head position**

# Simulation of 2-tape TM by Standard TM

- For each transition of M,  $\delta(q_0, a, e) = (q_1, x, y, L, R)$ . The standard TM has the identical move

$$\delta'(q'_0, a, e) = (q'_1, x, y, L, R).$$

- The symbol a represents the current tape symbol on first track of Standard TM while e represents the current tape symbol on third track.

- The simulation of each move of M requires a number of moves of M'.**

- The M' simulates the behaviour of M by considering the tracks rather than tapes. All four tracks of M's tape are modified to reflect the move of M.

# Simulation of 2-tape TM by Standard TM

## Reference point

#	a	b	c			
#	0	1	0			
#	e	f	g	h		
#	0	0	1	0		

↑

Tape 1  
head position  
  
Tape 2  
head position

Repeat for each state transition:

1. Return to reference point
2. Find current symbol in Tape 1
3. Find current symbol in Tape 2
4. Make transition

# Simulation of 2-tape TM by Standard TM

- Starting from some standard position( Say the left end marker),  $M'$  searches track 2 to locate the 1. The symbol found in the corresponding cell on track 1 is remembered by putting the control unit of  $M'$  into a state chosen for this purpose.
- Similarly, track 4 and 3 are searched for the position of the read-write head of  $M'$ .  
 $\delta'(q'_0, a, e) = (q'_1, x, y, L, R)$ .
- The read-head in  $q'_0$  reads  $a$  on first track. It changes it to  $x$  and moves left i.e. it writes 0 on current cell in track two and writes 1 on the left cell of the current cell.
- The read-head in  $q'_0$  reads  $e$  on third track. It changes it to  $y$  and moves right i.e. it writes 0 on current cell in fourth track and writes 1 on the right cell of the current cell.
- Finally, the read-write head of  $M'$  returns to the standard position for the simulation of the next move.

# Same power doesn't imply same speed

$$L = \{a^n b^n\}$$

**Standard Turing machine:  $O(n^2)$  Time**

**Standard TM Goes back and forth, to match the a's with the b's.**

**A 2-tape machine:  $O(n)$  Time**

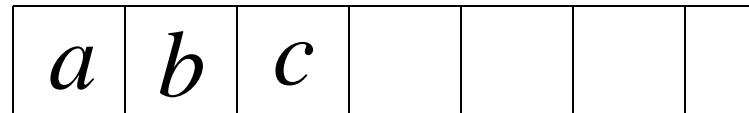
- It copies  $b^n$  to Tape-2 in  $O(n)$  time
- Compares  $a^n$  on tape-1 and  $b^n$  on tape-2 in  $O(n)$  time.

# Off-line TM

- In Standard TM, there is only one tape which is input as well as output file.
  - The initial tape content is known as input.
  - Whenever the control reaches to final state and halts, the tape content is the output.
- The TM with different input and output tapes is known as Off-line Turing Machine.
  - The input tape is **read-only**.
  - The output tape is readable and writable (**Read/Write Allowed**).
  - Initially, the content of input tape may be similar to the content of output tape.
- Each move in offline TM is governed by
  - The internal state
  - Current symbol on input tape
  - Current symbol on the output tape
- As the off-line TM moves, the content of output tape changes according to the transition function. However, the content of input tape does not change.

# The Off-Line Machine

## Input File



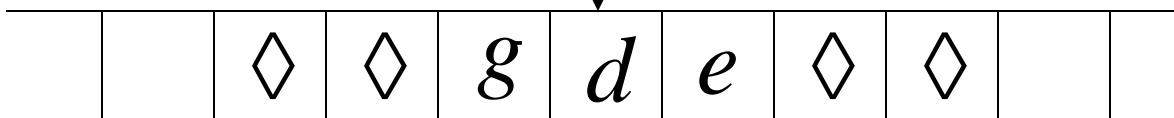
read-only



Control Unit

Tape

read-write



# Off-Line TM

- An Offline TM is like a standard TM **but it has different input and output files with its own head.**

- An Offline TM can be defined as  $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

$$\delta: Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

- For each transition of  $M$ ,  $\delta(q_0, a, e) = (q_1, y, L)$

# Off-line TM is equivalent to Standard TM

## Part-1:

Off-line TM simulate  
Standard TM

- Copy input file to tape
- Continue computation as in Standard Turing machine

## Part-2:

Standard machines  
simulate  
Off-line TM.

- Use a TM with multi-track tape
- Input tape corresponds to first two tracks
- output tape corresponds to last two tracks
- Continue computation similar to Multi-tape Turing machine

# Non-Deterministic TM (NTM)

- A important variant, with several possible next configurations at each step.
  - A Non-Deterministic Turing machine (NTM) can be define as  $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

$$\delta: Q \times \Gamma \rightarrow 2^{(Q \times \Gamma \times \{L, R\})}$$

# Non-deterministic TM

- The NTM may have transitions specified by

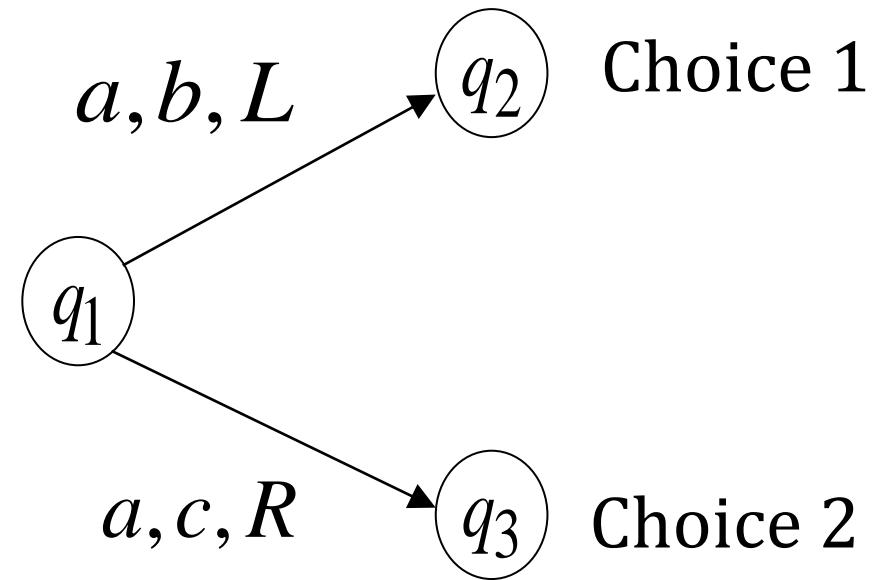
$$\delta(q_1, a) = \{(q_2, b, L), (q_3, c, R)\},$$

*The following moves are possible*

$$bbq_0aaa \xrightarrow{} bq_2bbaa$$

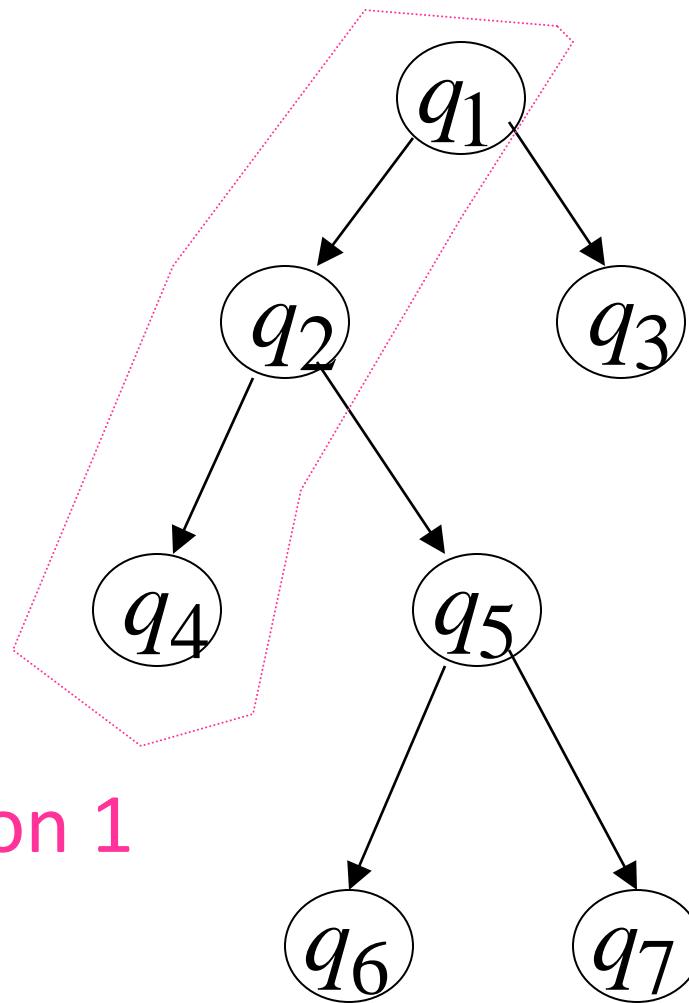
Or

$$bbq_0aaa \xrightarrow{} bbcq_3aa$$



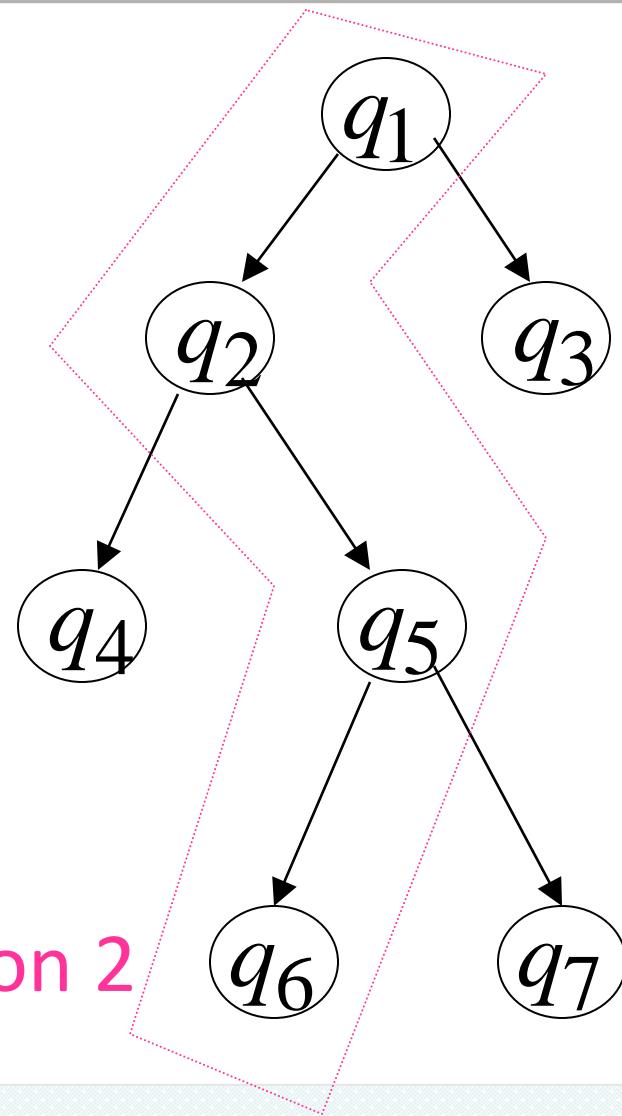
Non Deterministic Choice

# Non-Deterministic Choices



Computation 1

# Non-Deterministic Choices



Computation 2

# NTM

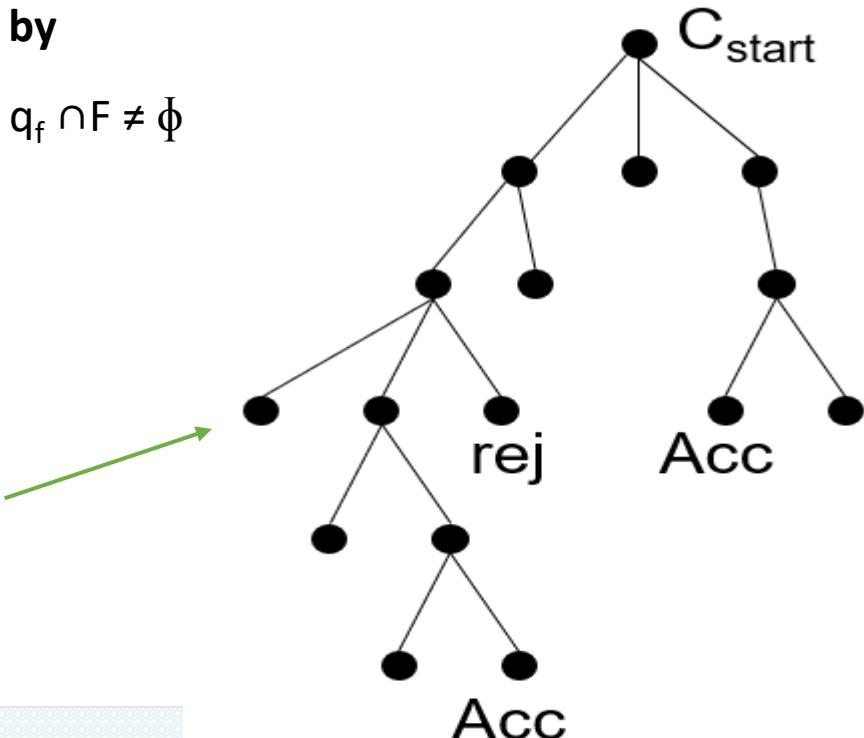
The NTM M accepts string w if there exist at least one configuration  $C_1, C_2, \dots, C_k$ .

- $C_1$  is start configuration of M on input w.
- $C_i \Rightarrow C_{i+1}$  for  $i = 1, 2, 3, \dots, k-1$ .
- $C_k$  is an accepting configuration.

- The language accepted by the NTM is given by

$$L = \{w : q_0 w \vdash x_1 q_f x_2, x_1, x_2 \in \Gamma^*, q_f \cap F \neq \emptyset\}$$

Computation Tree



# Simulation

NonDeterministic Machines simulate  
Standard (deterministic) Machines:

Every deterministic machine  
is also a nondeterministic machine

Deterministic machines simulate  
NonDeterministic machines:

Deterministic machine:

Keeps track of all possible computations

# Turing's Thesis

Any computation that can be carried out by mechanical means/present-day computers can also be performed by some TM.

- However,
  - The Turing Machine (discussed) is restricted to perform only one type of computation.
  - Digital computers, on the other hand, are general-purpose machines that can be programmed to do different jobs at different times.
  - **Standard Turing machines cannot be considered equivalent to general-purpose digital computers.**

# A limitation of Turing Machines

Standard Turing Machines are “hardwired”



they execute  
only one program

Real Computers are re-programmable

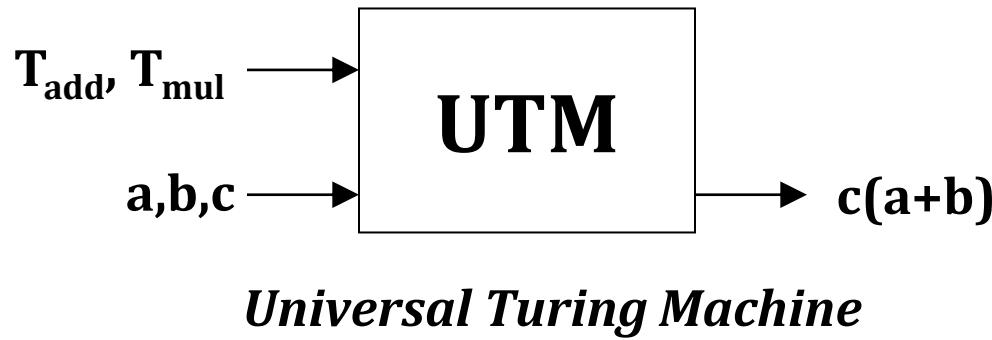
Solution: Universal Turing Machine

Attributes:

- Reprogrammable machine
- Simulates any other Turing Machine

# Universal Turing Machine

- A universal Turing machine  $M_u$  is an automaton that, given as input the description of any TM  $M$  and a string  $w$ , can simulate the computation of  $M$  on  $w$ .
- It can simulate any other Turing machine.
- It accepts both inputs i.e. data and instructions to execute the program.

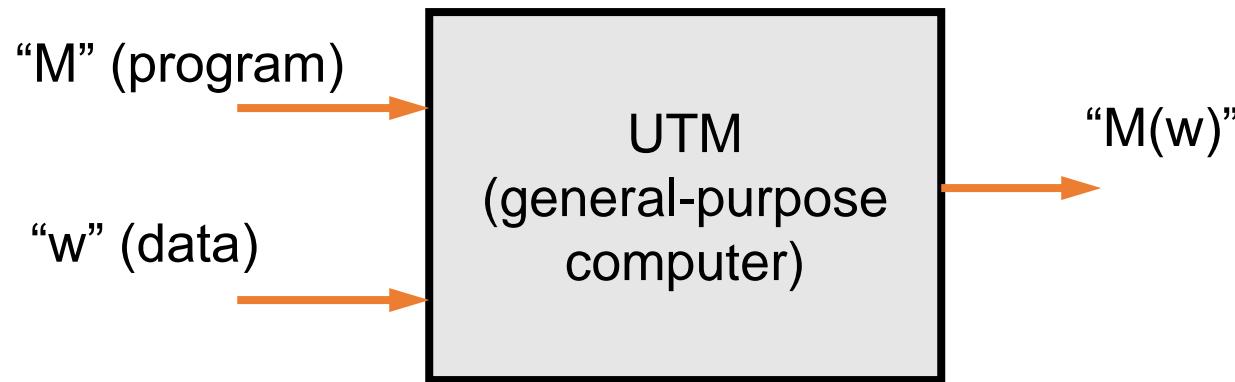


*A computer is a universal Turing Machine*

# UTM

**UTM is a general-purpose TM, or a TM simulator.**

**Given a specifications of TM  $M$  and a description of an input  $w$ , it can simulate the execution of  $M$  on  $w$ .**



# UTM

- For each TM  $M_i = (Q_i, \Sigma_i, \Gamma_i, \delta_i, q_0, B, F_i)$

Assumptions,

$Q_i = \{q_1, q_2, \dots, q_n\}$ , Where  $q_1$  the initial state,  $q_2$  the single final state

$\Gamma_i = \{a_1, a_2, \dots, a_m\}$ , Where  $a_1$  represents the blank symbol

## Encoding

$q_1$  is represented by 1,  $q_2$  is represented by 11,  $q_3$  is represented by 111, and so on.

$a_1$  is encoded as 1,  $a_2$  is encoded as 11,  $a_3$  is encoded by 111 and so on  
 $L$  and  $R$  moves are represented by 1 and 11 respectively.

The transition  $\delta(q_1, a_2) = (q_2, a_3, L)$  is encoded as

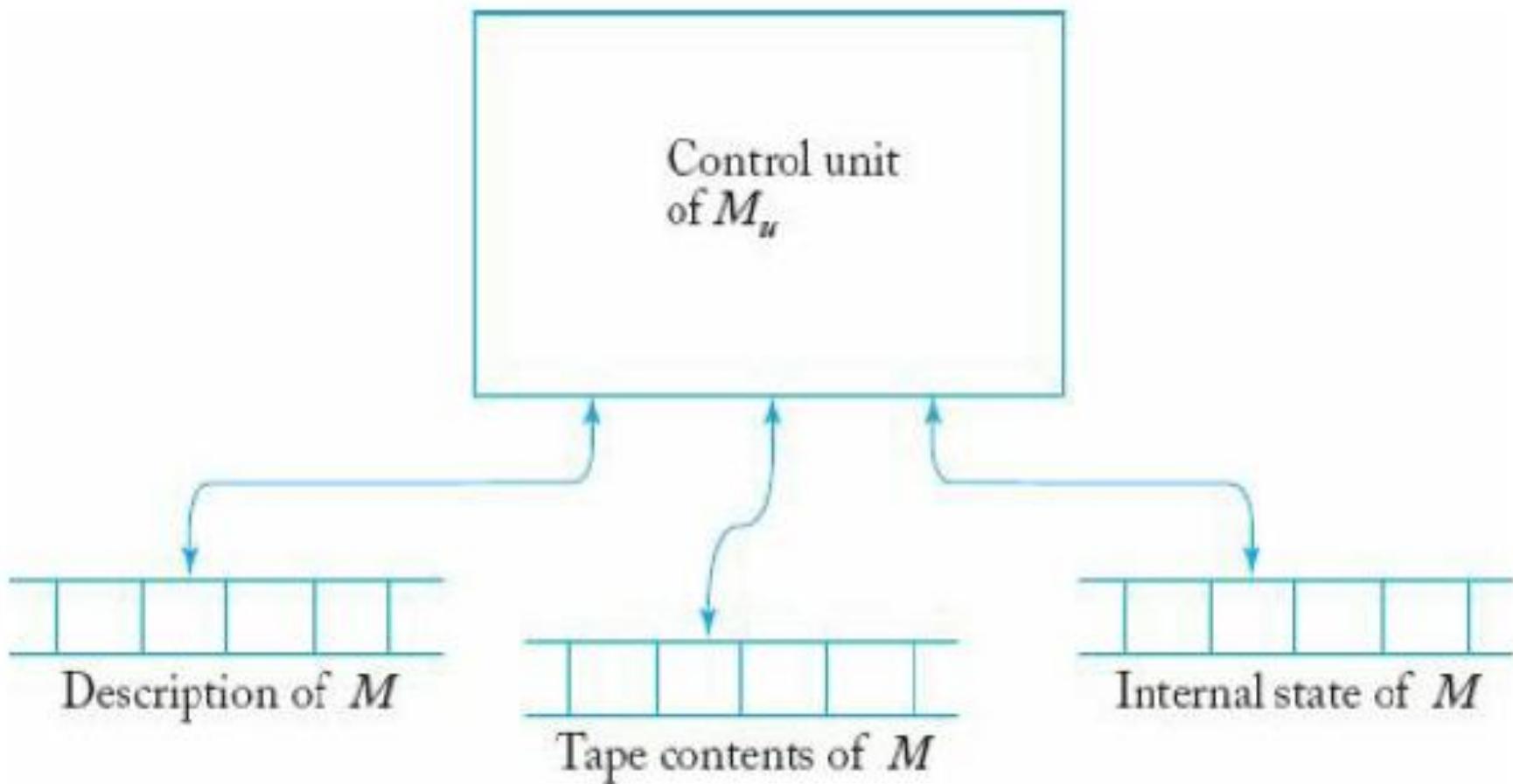
#10110110111010#

The symbol 0 is used as a separator between the 1's.

The symbol 00 is used as a separator **between two transitions**.

The symbol 000 is used as a separator **between the TMs**.

# Universal Turing Machine



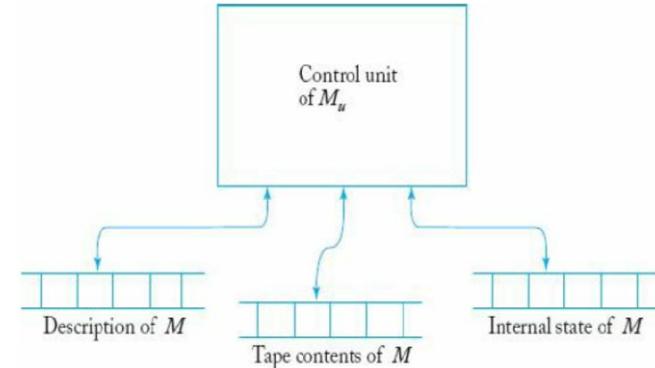
# UTM

A universal Turing machine  $M_u$  is 3-Tape TM.

**Tape-1 consists of description of each TM M in encoded form as a binary string of 0's and 1's**

Tape-2 consists of input string in encoded form

Tape-3 consists of states of TM in encoded form



For any input  $M$  and  $w$ ,

- UTM looks first at the contents of tapes 2 and 3 to determine the configuration of  $M$ .
- UTM then searches tape 1 to see what  $M$  would do in this configuration.
- Finally, tapes 2 and 3 will be modified to reflect the result of the move.

# A Turing Machine is described as a string of 0's and 1's

Therefore:

The set of Turing machines forms a language:

each string of the language is the binary encoding of a Turing Machine

$L = \{ 010100101,$

(Turing Machine 1)

$00100100101111,$

(Turing Machine 2)

$111010011110010101,$

.....

..... }

# Countable Sets

Infinite sets are either:

Countable

or

Uncountable

# Countable Sets

**Countable set:** Any finite set or Any *Countably infinite* set.

There is a one to one correspondence  
between  
elements of the set  
and  
Natural numbers

A set  $S$  is *countable* if there exists an injective function  $f$  from  $S$  to the natural numbers  $\mathbf{N} = \{0, 1, 2, 3, \dots\}$ .

If such an  $f$  can be found that is also surjective (and therefore bijective), then  $S$  is called *countably infinite*.

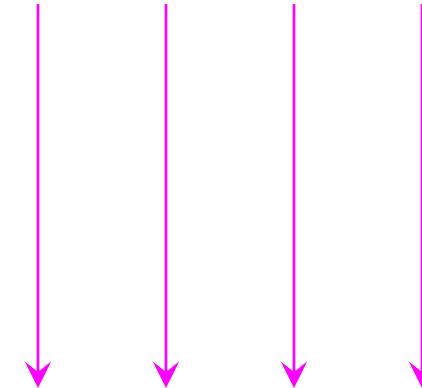
Example:

The set of even integers  
is countable.

Even Integers:

0, 2, 4, 6, K

Correspondence:



Positive Integers:

1, 2, 3, 4, K

$2n$  corresponds to  $n + 1$

Example:

The set of rational numbers  
is countable

Rational Numbers:

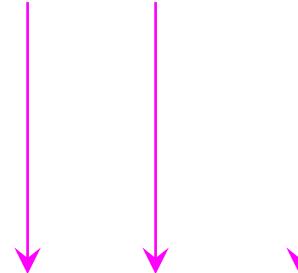
$$\frac{1}{2}, \frac{3}{4}, \frac{7}{8}, K$$

# Naïve Proof

Rational Numbers:

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, K$$

Correspondence:



Positive Integers:

$$1, 2, 3, K$$

Doesn't work:

We will never count  
numbers with denominator 2, 3.....:

$$\frac{2}{1}, \frac{2}{2}, \frac{2}{3}, K$$

# Better Approach

$$\frac{1}{1}$$

$$\frac{1}{2}$$

$$\frac{1}{3}$$

$$\frac{1}{4}$$

..

$$\frac{2}{1}$$

$$\frac{2}{2}$$

$$\frac{2}{3}$$

..

$$\frac{3}{1}$$

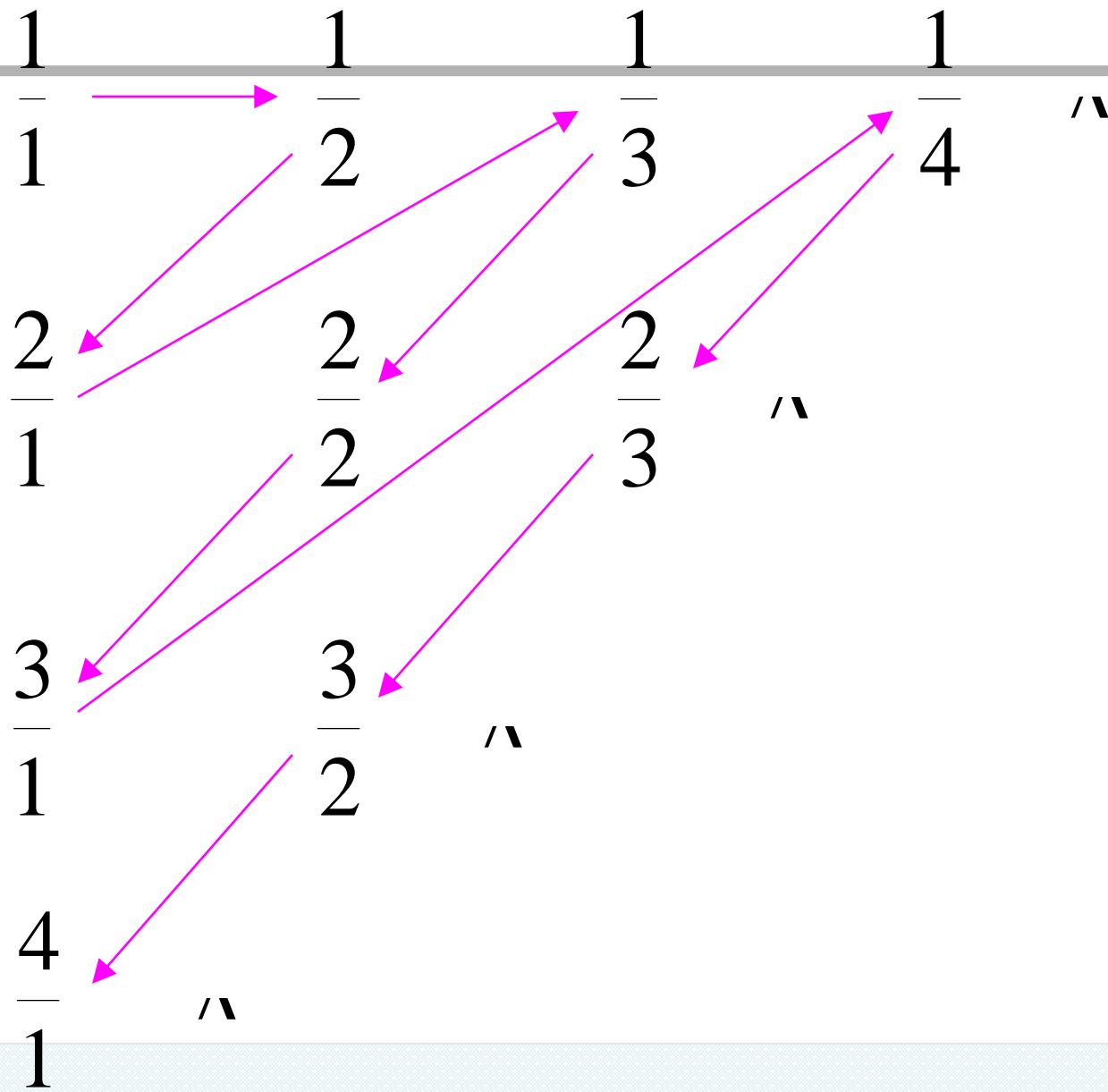
$$\frac{3}{2}$$

..

$$\frac{4}{1}$$

..

# Better Approach



Example:

The set of rational numbers  
is countable

Rational Numbers:

$$\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{2}{2}, K$$

Correspondence:

Positive Integers:

$$1, 2, 3, 4, 5, K$$

We proved: the set of rational numbers is countable by  
describing an enumeration procedure

# Enumeration Procedure

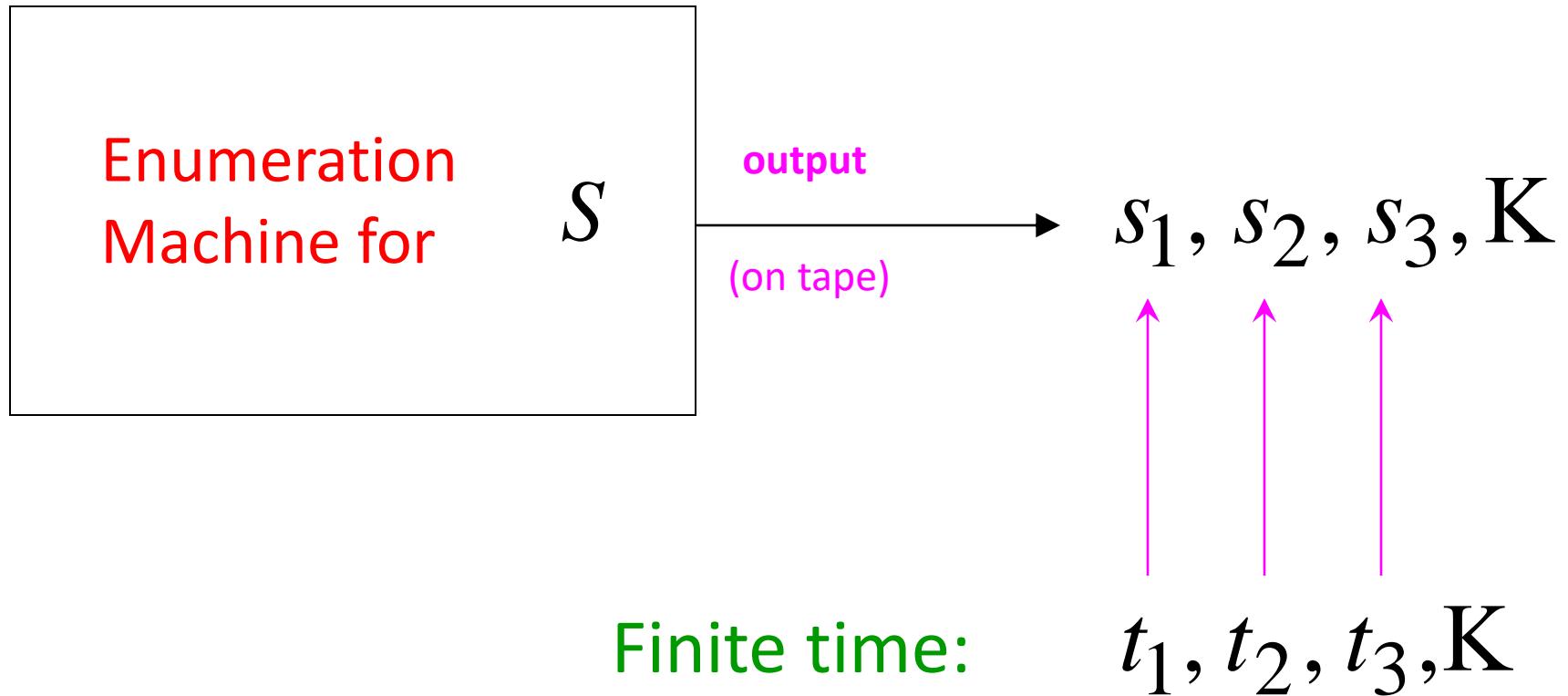
Let  $S$  be a set of strings on some alphabet  $\Sigma$

An **enumeration procedure** for  $S$  is a Turing Machine  
that generates all strings of  $S$  one by one

and

Each string is generated in **finite time**

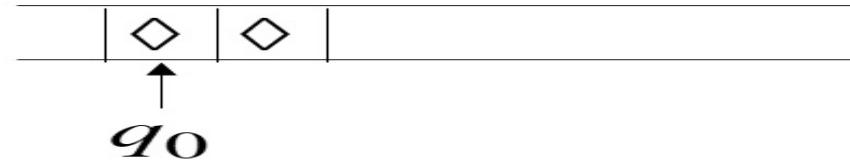
Strings     $s_1, s_2, s_3, \dots \in S$



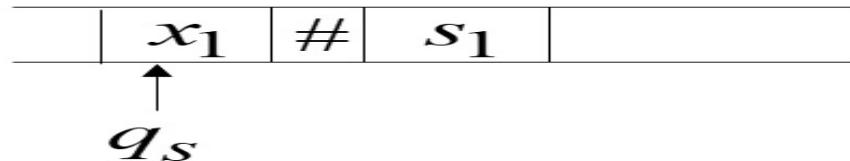
## Enumeration Machine

## Configuration

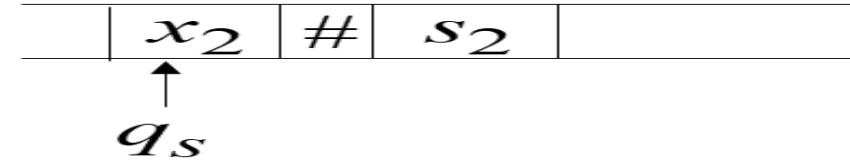
Time  $t_0$



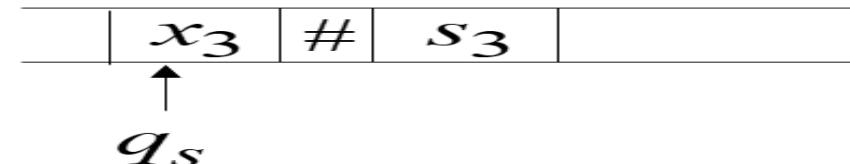
Time  $t_1$



Time  $t_2$



Time  $t_3$



## Observation:

If for a set, there is an enumeration procedure,  
then the set is countable

# Countable Sets

Example: The set of all strings  $\{a,b,c\}^+$  is countable

Proof: An enumeration procedure that produces its elements in some order

Naive procedure:

Produce the strings in lexicographic order:

Doesn't work:

strings starting with  $b$   
will never be produced

$a$   
 $aa$   
 $aaa$   
 $aaaa$   
.....

# Better Approach

**Length of string**

**+ alphabetic ordering of all equal length strings**

1. Produce all strings of length 1
2. Produce all strings of length 2
3. Produce all strings of length 3
4. Produce all strings of length 4
- .....

$a$   
 $b$   
 $c$

length 1

$aa$   
 $ab$   
 $ac$   
 $ba$   
 $bb$   
 $bc$   
 $ca$   
 $cb$   
 $cc$

length 2

$aaa$   
 $aab$   
 $aac$   
.....

length 3

Produce strings in  
**Proper Order:**

Using this order; there exist  
One-to-one correspondence  
between strings in S  
and natural set.

**Theorem:**

The set of all Turing Machines, although infinite, is countable

**Proof:** Any Turing Machine can be encoded with a binary string of 0's and 1's

Find an enumeration procedure for the set of Turing Machine strings

# Enumeration Procedure:

Repeat

1. Generate the next binary string of 0's and 1's in proper order
2. Check if the string describes a Turing Machine
  - if YES: print string on output tape
  - if NO: ignore string

Since every TM has a finite description, any specific TM will eventually be generated by this process.

# Uncountable Sets

Definition:

A set is uncountable if it is not countable

Let  $S$  be an infinite countable set.

The powerset  $2^S$  of  $S$  is **uncountable**

## Proof:

Since  $S$  is countable, we can write

$$S = \{s_1, s_2, s_3, \dots\}$$

↑  
Elements of  $S$

Elements of the powerset have the form:

$$\{s_1, s_3\}$$

$$\{s_5, s_7, s_9, s_{10}\}$$

.....

We encode each element of the power set with a binary string of 0's and 1's

Powerset element	Encoding				
	$s_1$	$s_2$	$s_3$	$s_4$	$\wedge$
$\{s_1\}$	1	0	0	0	$\wedge$
$\{s_2, s_3\}$	0	1	1	0	$\wedge$
$\{s_1, s_3, s_4\}$	1	0	1	1	$\wedge$

Any element  $t$  of  $2^S$  is a sequence of 0's and 1's with a 1 in position  $i$  iff  $s_i$  is in  $t$ .

Let's assume (for contradiction)  
that the powerset is countable.

Then: we can enumerate  
the elements of the powerset  
i.e. Its elements can be written in  
some order i.e.  $t_1, t_2, t_3 \dots$

# Powerset element

## Encoding

$t_1$	1	0	0	0	0	$\wedge$
$t_2$	1	1	0	0	0	$\wedge$
$t_3$	1	1	0	1	0	$\wedge$
$t_4$	1	1	0	0	1	$\wedge$

$\wedge$

Take the powerset element  
whose bits are the complements  
in the diagonal

$t_1$       1      0      0      0      0       $\wedge$

$t_2$       1      1      0      0      0       $\wedge$

$t_3$       1      1      0      1      0       $\wedge$

$t_4$       1      1      0      0      1       $\wedge$

New element: 0011K

(binary complement of diagonal)

The new element must be some  $t_i$   
of the powerset

However, it's impossible:

from definition of  $t_i$

the  $i$ -th bit of  $t_i$  must be

the complement of itself

And it cannot be  $t_i$

Contradiction!!!

Since, we have a contradiction:

The powerset  $2^S$  of  $S$  is uncountable

# An Application: Languages

Example Alphabet :  $\{a,b\}$

The set of all Strings:

$$S = \{a,b\}^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

infinite and countable

A language is a subset of :  $S$

$$L = \{aa, ab, aab\}$$

# An Application: Languages

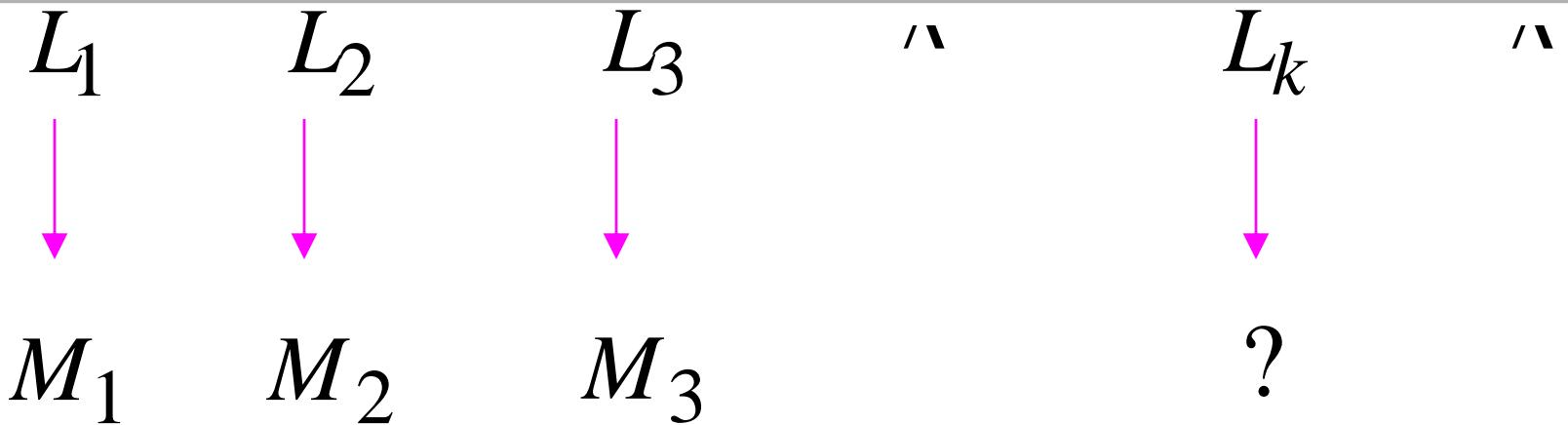
The powerset of  $S$  contains all languages:

$$2^S = \{ \{\lambda\}, \{a\}, \{a,b\}, \{aa, ab, aab\}, K \}$$

$L_1$     $L_2$     $L_3$     $L_4$     $\dots$

**uncountable**

# Languages: uncountable



Set of Turing machines: countable

There are more languages  
than Turing Machines

# Conclusion:

There are some languages not accepted by Turing Machines

(These languages cannot be described by algorithms)

# Thank You