Introduction to Automata Theory

MODULE - 1

Introduction

- The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An **automaton** (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

- An automaton with a finite number of states is called a **Finite Automaton (FA)** or **Finite State Machine (FSM).**

Why to study Theory of Computation?

- Theory of computation is mainly concerned with the study of how problems can be solved using algorithms.

- It is the study of mathematical properties both of problems and of algorithms for solving problems that depend on neither the details of today's technology nor the programming language.

Why to study Theory of Computation?

- It is still useful in two key ways:
  - It provides a set of ***abstract structures*** that are useful for solving certain classes of problems. These abstract structures can be implemented on whatever hardware/software platform is available
  - It defines provable limits to ***what can be computed*** regardless of processor speed or memory size. An understanding of these limits helps us to focus our design effort in areas in which it can pay off, rather than on the computing equivalent of the search for a perpetual motion machine.

The goal is to discover fundamental properties of the problems like:

- Is there any computational ***solution*** to the problem? ıf not. is there a restricted but useful variation of the problem for which a solution does exist?
- If a solution exists, can it be implemented using some ***fixed amount of memory***?
- If a solution exists. how ***efficient*** is it? More specifically. how do its time and space requirements grow as the size of the problem grows?
- Are there groups of problems that are ***equivalent*** in the sense that if there is an efficient solution to one member of the group there is an efficient solution to all the others?

Applications of theory of computation:

- ***Development of Machine Languages***: Enables both machine-machine and person-machine communication. Without them, none of today's applications of computing could exist. Example: Network communication protocols, HTML etc.

- ***Development of modern programming languages:*** Both the design and the implementation of modern programming languages rely heavily on the theory of context-free languages. Context- free grammars are used to document the languages syntax and they form the basis for the parsing techniques that all compilers use.

- ***Natural language processing***: It is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages.

- ***Automated hardware systems:*** Systems as diverse as parity checkers, vending machines, communication protocols, and building security devices can be straightforwardly described as finite state machines, which is a part of theory of computation.

Applications of theory of computation:

- ***Video Games:*** Many interactive video games are use large nondeterministic finite state machines.
- ***Security*** is perhaps the most important property of many computer systems. The undecidability results of computation show that there cannot exist a general-purpose method for automatically verifying arbitrary security properties of programs.
- ***Artificial intelligence:*** Artificial intelligence programs solve problems in task domains ranging from medical diagnosis to factory scheduling. Various logical frameworks have been proposed for representing and reasoning with the knowledge that such programs exploit.
- ***Graph Algorithms:*** Many natural structures, including ones as different as organic molecules and computer networks can be modeled as graphs. The theory of complexity tells us that, is there exist efficient algorithms for answering some important questions about graphs. Some questions are "hard", in the sense that no efficient algorithm for them is known nor is one likely to be developed.

Alphabets & Strings

- An alphabet Σ is a finite set of symbols
  - $\Sigma_1 = \{a, b, \ldots, z\}$
  - $\Sigma_2 = \{0, 1\}$
- A string is a finite sequence of symbols from an alphabet
  fire, truck are both strings over $\{a, \ldots, z\}$
- length of a string is the number of symbols in the string
  - $|fire| = 4$, $|truck| = 5$
- The empty string ε is a string of 0 characters $| \varepsilon | = 0$
  - $\circ$ is the concatenation operator $w_1 = fire$, $w_2 = truck$
    $w_1 \circ w_2 = firetruck$
    $w_2 \circ w_1 = truckfire$
    $w_2 \circ w_2 = trucktruck$
- Often drop the $\circ$: $w_1w_2 = firetruck$
- For any string w, $w\varepsilon = w$

Concatenation & Reversal

- We can concatenate a string with itself:
  - $w^1 = w$
  - $w^2 = ww$
  - $w^3 = www$
- By definition, $w^0 = \varepsilon$
- Can reverse a string: w^R
  - $truck^R = kcurt$

Relations on strings

- **Substring:** A string ***s*** is a substring of a string of **t** iff ***s*** occurs contiguously as part of **t**.
  - For example: aaa is a substring of aaabbbaaa, aaaaaa is not a substring of aaabbbaaa

- **Proper Substring:** A string **s** is a proper substring of a string **t**, iff **s** is a substring of t and s ≠t. Every string is a substring (although not a proper substring) of itself. The empty string **ε** is a substring of every string

- **Prefix**: A string s is a prefix of t, iff $\exists x \in \sum*(t = sx)$.
  - A string s is a proper prefix of a string t iff s is a prefix of t and s≠t.
  - Every string is a prefix (although not a proper prefix) of itself.
  - The empty string ε, is a prefix of every string.
  - For example. the prefixes of abba are: ε, a, ab, abb, abba.

Functions on Strings

- The length of a string s, which we will write as |s|, is the number of symbols in s.
  For example: |ε| = 0
  - |1001101| = 7
  - |fire| = 4, |truck| = 5
- For any symbol c and string s, we define the function #c(s) to be the number of times that the symbol c occurs in s. So, for example, #a(abbaaa) = 4.
- The concatenation of two strings s and t, written s || t or simply st, is the string formed by appending t to s.
  For example, if x = good and y = bye, then xy = goodbye.
  - So |xy| = |x| + |y|.
- The empty string, ε, is the identity for concatenation of strings.
  - So ∀x (x ε = ε x = x).
- Concatenation, as a function defined on strings, is associative.
  - So ∀s, t, w ((st)w = s(tw)).

String Replication and Reversal

- Next we define string replication. For each string w and each natural number i, the string w^i is defined as:
  - $w^0 = \varepsilon$
  - $w^{i+1} = w^i w$

For example: $a^3 = aaa$

$(bye)^2 = byebye$

$a^0b^3 = bbb$

String reversal: For each string w, the reverse of w, which we will write w^R , is defined as:

- if $|w| = 0$ then $w^R = w = \varepsilon$
- if $|w| \geq 1$ then $\exists a \in \Sigma\ (\exists u \in \Sigma^*\ (w = ua))$. (i.e., the last character of w is a.) Then define $w^R = a\ u^R$.

Relations on strings

- **<u>Suffix</u>**: A string s is a suffix of t, iff $\exists x \in \sum*(t = xs)$.
  - A string s is a proper suffix of a string t iff s is a suffix of t and $s \neq t$.
  - Every string is a suffix (although not a proper suffix) of itself.
  - The empty string ε, is a suffix of every string.
  - For example. the suffix of abba are: ε, a, ba, bba, abba.

Languages

- A language is a (finite or infinite) set of strings over a finite alphabet Σ. When we are talking about more than one language, we will use the notation Σ_L to mean the alphabet from which the strings in the language L are formed.

Defining Languages Given an Alphabet

- Let Σ = {a, b}. Σ * = {ε, a, b, aa, ab, ba, bb, aaa, aab, ...}.
  - Some examples of languages over Σ are: ϕ, {ε}, {a, b}, {ε, a, aa, aaa, aaaa, aaaaa}, {ε, b, bb, bbb, bbbb, bbbbb, ...}

Techniques for Defining Languages

- We will use a variety of techniques for defining the languages that we wish to consider. Since languages are sets, we can define them using any of the set-defining techniques.

Languages

Example: All a's Precede All b's

- Let L = {x ∈ {a, b}* : all a's precede all b's in w}. The strings ε, a, aa, aabbb, and bb are in L.

Example: Strings That End in a

- Let L = {x : ∃ y ∈ {a, b}* (x = ya)}.

  The strings a, aa, aaa, bbaa, and ba are in L.

  The strings ε, bab, and bca are not in L.

  L consists of all strings that can be formed by taking some string in {a, b}* and concatenating a single a onto the end of it.

The Empty Language

- Let L = {} = ϕ. L is the language that contains no strings.

Languages

- The Empty Language is Different From the Empty String
- Let L = {ε}, the language that contains a single string, ε.
- Note that L is different from ϕ.

Functions on Languages

- Since languages are sets, all of the standard set operations are well-defined on languages. In particular, we will find union, intersection, difference, and complement to be useful.

Set Functions Applied to Languages

- Let: $\Sigma = \{a, b\}$.
- L_1 = {strings with an even number of a's}.
- L_2 = {strings with no b's} = {ε, a, aa, aaa, aaaa, aaaaa, aaaaaa, …}.
  - $L_1 \cup L_2$ = {all strings of just a's plus strings that contain b's and an even number of a's}.
  - $L_1 \cap L_2$ = {ε, aa, aaaa, aaaaaa, aaaaaaaa…}.
  - $L_2 - L_1$ = {a, aaa, aaaaa, aaaaaaa, …}.
  - $\neg(L_2 - L_1)$ = {strings with at least one b} ∪ {strings with an even number of a's}

Functions on Languages

- Because languages are sets of strings, it makes sense to define operations on them in terms of the operations that we have already defined on strings. Three useful ones to consider are **concatenation, Kleene star**, and **reverse**.

Let L_1 and L_2 be two languages defined over some alphabet Σ. Then their **concatenation**, written L_1L_2 is:

$L_1L_2 = \{w \in \Sigma^* : \exists s \in L_1 (\exists t \in L_2 (w = st))\}$.

Concatenation of Languages

Let L_1 = {cat, dog, mouse, bird}.

L_2 = {bone, food}.

L_1L_2 = {catbone, catfood, dogbone, dogfood, mousebone, mousefood, birdbone, birdfood}

Functions on Languages

- The language $\{\varepsilon\}$ is the identity for concatenation of languages. So, for all languages L,
  $L\{\varepsilon\} = \{\varepsilon\}L = L$.
- The language ϕ is a zero for concatenation of languages. So, for all languages L,
  $L\,\phi = \phi\,L = \phi$.

- Concatenation, as a function defined on languages, is associative. So, for all languages L_1, L_2, , and L_3 :
  $((L_1L_2)\,L_3 = L_1(L_2L_3))$.

Functions on Languages

- Let L be a language defined over some alphabet Σ. Then the Kleene star of L, written L^* is:

  $L^* = \{\varepsilon\} \cup \{w \in \Sigma^* : \exists k \geq 1 \, (\exists w_1, w_2, \ldots w_k \in L \, (w = w_1 w_2 \ldots w_k))\}$.

- In other words, L^* is the set of strings that can be formed by concatenating together zero or more strings from L.

Example Kleene Star

- Let L = {dog, cat, fish}. Then:
- L^* = {ε, dog, cat, fish, dogdog, dogcat, …, fishdog, …, fishcatfish, fishdogfishcat, …}.

Functions on Languages

- Since languages are sets, all of the standard set operations are well-defined on languages. In particular, we will find union, intersection, difference, and complement to be useful. Complement will be defined with Σ^* as the universe unless we explicitly state otherwise.

- Set Functions Applied to Languages
- Let: $\Sigma = \{a, b\}$.
- L_1 = {strings with an even number of a's}.
- L_2 = {strings with no b's} = {ε, a, aa, aaa, aaaa, aaaaa, aaaaaa, …}.
- $L_1 \cup L_2$ = {all strings of just a's plus strings that contain b's and an even number of a's}.
- $L_1 \cap L_2$ = {ε, aa, aaaa, aaaaaa, aaaaaaaa…}.
- $L_2 - L_1$ = {a, aaa, aaaaa, aaaaaaa, …}.
- $\neg(L_2 - L_1)$ = {strings with at least one b} ∪ {strings with an even number of a's}

Functions on Languages

- Because languages are sets of strings, it makes sense to define operations on them in terms of the operations that we have already defined on strings. Three useful ones to consider are concatenation, Kleene star, and reverse.

- Let L_1 and L_2 be two languages defined over some alphabet Σ. Then their **concatenation**, written L_1L_2 is:

$$L_1L_2 = \{w \in \Sigma^* : \exists s \in L_1 \, (\exists t \in L_2 \, (w = st))\}.$$

Concatenation of Languages

Let L_1 = {cat, dog, mouse, bird}.

L_2 = {bone, food}.

- L_1L_2 = {catbone, catfood, dogbone, dogfood, mousebone, mousefood, birdbone, birdfood}

Functions on Languages

- The language $\{\varepsilon\}$ is the identity for concatenation of languages.

  So, for all languages L, $L\{\varepsilon\} = \{\varepsilon\}L = L$.

- The language ϕ is a zero for concatenation of languages. So, for all languages L,

  $L\,\phi = \phi\,L = \phi$.

That ϕ is a zero follows from the definition of the concatenation of two languages as the set consisting of all strings that can be formed by selecting some string s from the first language and some string t from the second language and then concatenating them together. There are no ways to select a string from the empty set.

- Concatenation, as a function defined on languages, is associative. So, for all languages L_1, L_2, and L_3:

  $((L_1L_2)L_3 = L_1(L_2L_3))$.

Functions on Languages

- Let L be a language defined over some alphabet Σ. Then the **Kleene star** of L, written L* is:

$$L^* = \{\varepsilon\} \cup \{w \in \Sigma^* : \exists k \geq 1 \, (\exists w_1, w_2, \ldots w_k \in L \, (w = w_1 w_2 \ldots w_k))\}.$$

- In other words, L* is the set of strings that can be formed by concatenating together zero or more strings from L.

Example: **Kleene Star**

- Let L = {dog, cat, fish}. Then:
- L* = {ε, dog, cat, fish, dogdog, dogcat, …, fishdog, …, fishcatfish, fishdogfishcat, …}.

Concatenation & Reversal

- We define string reversal. For each string w, the reverse of w, which we will write w^R , is defined as:

if $|w| = 0$ then $w^R = w = \varepsilon$

if $|w| \geq 1$ then $\exists a \in \Sigma\ (\exists u \in \Sigma^*\ (w = ua))$.
(i.e., the last character of w is a.)

Then define $w^R = a\ u^R$.

- **Theorem 1. Concatenation and Reverse of Strings**

Theorem 2.1: If w and x are strings, then $(w\ x)^R = x^R\ w^R$.

For example, $(\text{name tag})^R = (\text{tag})^R\ (\text{name})^R = \text{gateman}$.

Proof: The proof is by induction on $|x|$:

Base case: $|x| = 0$. Then $x = \varepsilon$, and $(wx)^R = (w\ \varepsilon)^R = (w)^R = \varepsilon w^R = \varepsilon^R\ w^R = x^R\ w^R$.

Prove: $\forall n \geq 0\ (((|x| = n) \rightarrow ((w\ x)^R = x^R\ w^R)) \rightarrow ((|x| = n + 1) \rightarrow ((w\ x)^R = x^R\ w^R)))$.

Consider any string x, where $|x| = n + 1$. Then $x = u\ a$ for some character a and $|u| = n$. So:

$(w\ x)^R = (w\ (u\ a))^R$	rewrite x as ua
$= ((w\ u)\ a)^R$	associativity of concatenation
$= a\ (w\ u)^R$	definition of reversal
$= a\ (u^R\ w^R)$	induction hypothesis
$= (a\ u^R)\ w^R$	associativity of concatenation
$= (ua)^R\ w^R$	definition of reversal
$= x^R\ w^R$	rewrite ua as x

Functions on Languages

- Let L be a language defined over some alphabet Σ.

Then the reverse of L, written L^R is: $L^R = \{w \in \Sigma^* : w = x^R \text{ for some } x \in L\}$.

In other words, L^R is the set of strings that can be formed by taking some string in L and reversing it.

Theorem 2.4 Concatenation and Reverse of Languages

Theorem: If L_1 and L_2 are languages, then $(L_1 L_2)^R = L_2^R L_1^R$

Proof: If x and y are strings,

then $\forall x (\forall y ((xy)^R = y^R x^R))$	Theorem 2.1
$(L_1 L_2)^R = \{(xy)^R : x \in L_1 \text{ and } y \in L_2\}$	Definition of concatenation of languages
$= \{y^R x^R : x \in L_1 \text{ and } y \in L_2\}$	Lines 1 and 2
$= L_2^R L_1^R$	Definition of concatenation of languages

A Language Hierarchy

- Defining the Task: Language Recognition

- Assume that we are given:
  - the definition of a language L.
  - a string w.

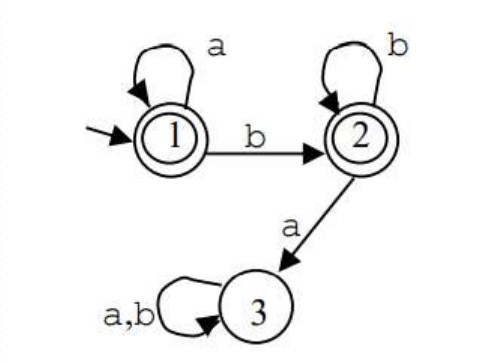
- Then we must answer the question: “Is w in L?” This question is an instance of a more general class that we will call **decision problems**.

- A decision problem is simply a problem that requires a yes or no answer.

A Machine-Based Hierarchy of Language Classes

1. **The Regular Languages**

- The first model we will consider is the finite state machine or FSM. Figure shows a simple FSM that accepts strings of a's and b's, where all a's come before all b's.



- We will call the class of languages that can be accepted by some FSM regular.

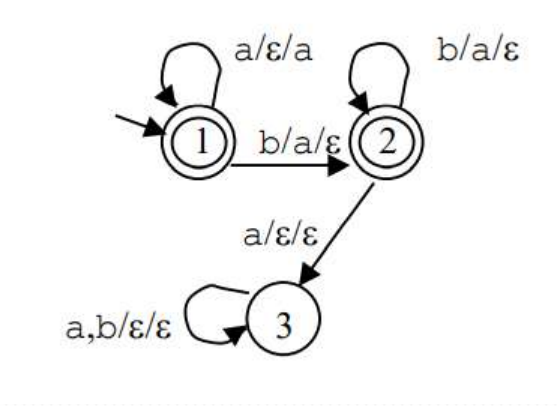
A Machine-Based Hierarchy of Language Classes

2. The Context-Free Languages

- There are useful simple languages that are not regular.
- Consider, for example, Bal, the language of balanced parentheses. Bal contains strings like (()) and ()(); it does not contain strings like ()))(.
- Because it's hard to read strings of parentheses, let's consider instead the related language $A^nB^n = \{a^nb^n : n \geq o\}$.
- But languages like Bal and A^nB^n are important. For example, almost every programming language and query language allows parentheses, so any front end for such a language must be able to check to see that the parentheses are balanced. Can we augment the FSM in a simple way and thus be able to solve this problem? The answer is yes.
- Suppose that we add one thing, a single stack. We will call any machine that consists of an FSM, plus a single stack, a pushdown automaton or PDA

A Machine-Based Hierarchy of Language Classes

- We can easily build a PDA M to accept A^nB^n. The idea is that, each time it sees an a, M will push it onto the stack. Then, each time it sees a b, it will pop an a from the stack.



A Machine-Based Hierarchy of Language Classes

3. The Decidable and Semi-decidable Languages

- But there are useful straightforward languages that are not context-free. Consider, for example, the language of English sentences in which some word occurs more than once.

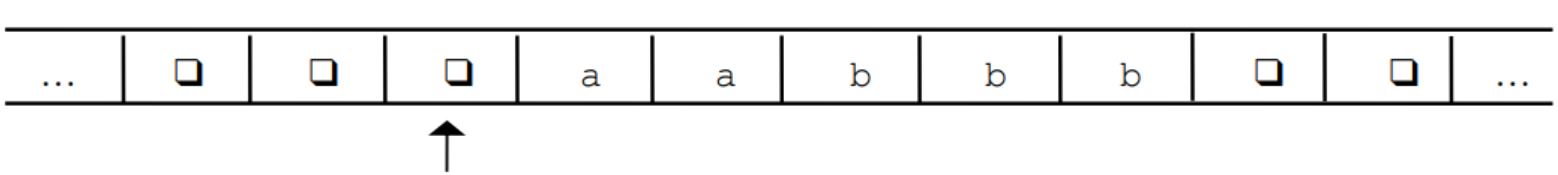
- Let $A^nB^nC^n = \{a^nb^nc^n: n \geq 0\}$, i.e., the language composed of all strings of a's, b's, and c's such that all the a's come first, followed by all the b's, then all the c's, and the number of a's equals the number of b's equals the number of c's.

- But it is easy to write a program to accept $A^nB^nC^n$. So, if we want a class of machines that can capture everything we can write programs to compute, we need a model that is stronger than the PDA.

- To meet this need, we will introduce a third kind of machine. We will get rid of the stack and replace it with an infinite tape. The tape will have a single read/write head. Only the tape square under the read/write head can be accessed (for reading or for writing). The read/write head can be moved one square in either direction on each move. The resulting machine is called a Turing machine.

A Machine-Based Hierarchy of Language Classes

- We will also change the way that input is given to the machine. Instead of streaming it, one character at a time, the way we did for FSMs and PDAs, we will simply write the input string onto the tape and then start the machine with the read/write head just to the left of the first input character.

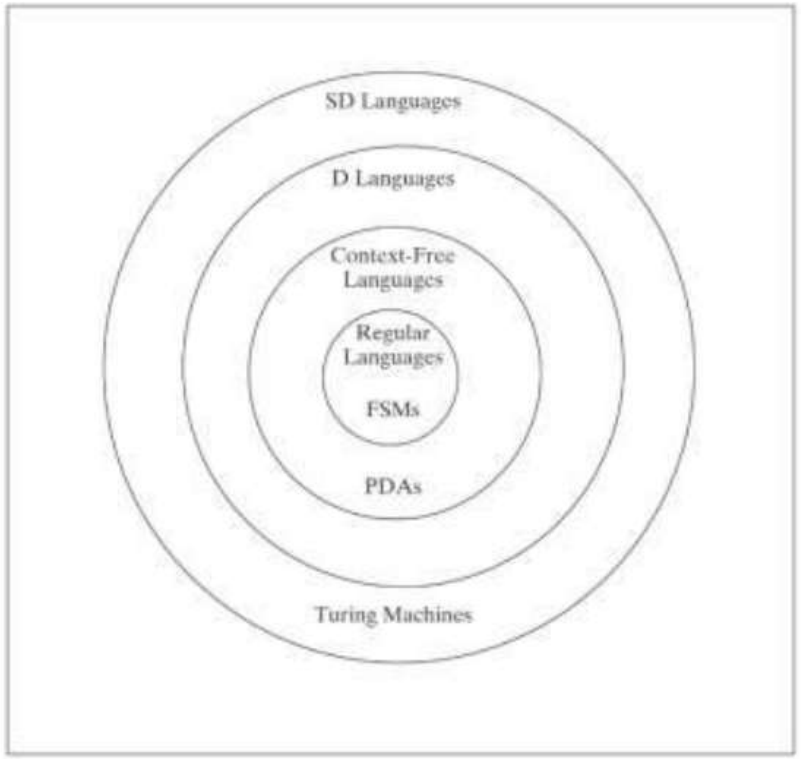


- The arrow under the tape indicates the location of the read/write head.

The Computational Hierarchy and Why It Is Important

We have now defined four language classes:

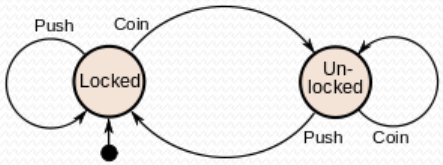
- regular languages, which can be accepted by some finite state machine,
- context-free languages, which can be accepted by some pushdown automaton,
- decidable (or simply D) languages, which can decided by some Turing machine that always halts, and
- semidecidable (or SD) languages, which can be *semi*-decided by some Turing machine that halts on all strings in the language.



Finite State Machine

- A **finite state machine** (sometimes called a finite state automaton) is a computation model that can be implemented with hardware or software and can be used to simulate sequential logic and some computer programs. Finite state automata generate regular languages. Finite state machines can be used to model problems in many fields including mathematics, artificial intelligence, games, and linguistics.

- Example: turnstile

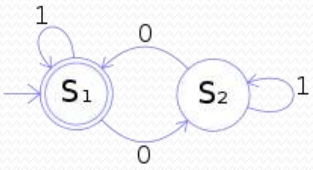


Finite State Machine

- There are two types of finite state machines (FSMs):
  - Deterministic finite state machines, often called deterministic finite automata, and
  - Non-deterministic finite state machines, often called non-deterministic finite automata.

Deterministic Finite State Machine

- A deterministic finite automaton (DFA) is described by a five-element tuple: (Q,Σ,δ,qo,F)
  - Q = a finite set of states
  - Σ = a finite, nonempty input alphabet
  - δ = a series of transition functions
  - qo = the starting state
  - F = the set of accepting states
- There must be exactly one transition function for every input symbol in Σ from each state.
- DFAs can be represented by diagrams of this form:



Deterministic Finite State Machine

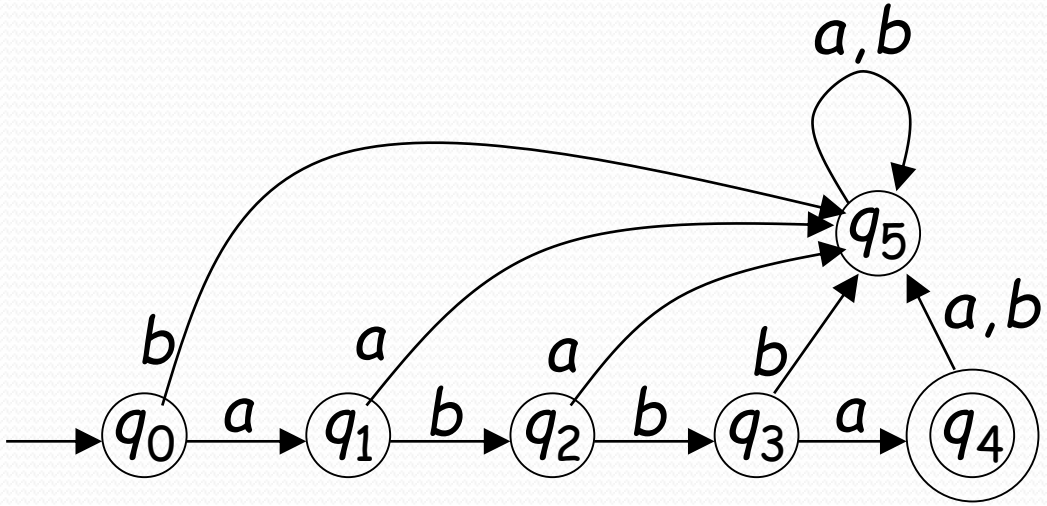
- **Example : A Simple Language of a's and b's**
- Let $L = \{w \in \{a, b\}^* : \text{every } a \text{ is immediately followed by a } b\}$. L can be accepted by the DFSM $M = (\{q0, q1, q2\},$
- $\{a, b\}, \delta, q0, \{q0\})$, where:

$\delta = \{ ((q0, a), q1),$
$((q0, b), q0),$
$((q1, a), q2),$
$((q1, b), q0),$
$((q2, a), q2),$
$((q2, b), q2)) \}.$

- The tuple notation that we have just used for δ is quite hard to read. We will generally find it useful to draw δ as a transition diagram instead. When we do that, we will use two conventions:
  - The start state will be indicated with an unlabeled arrow pointing into it.
  - The accepting states will be indicated with double circles.

Set of States Q

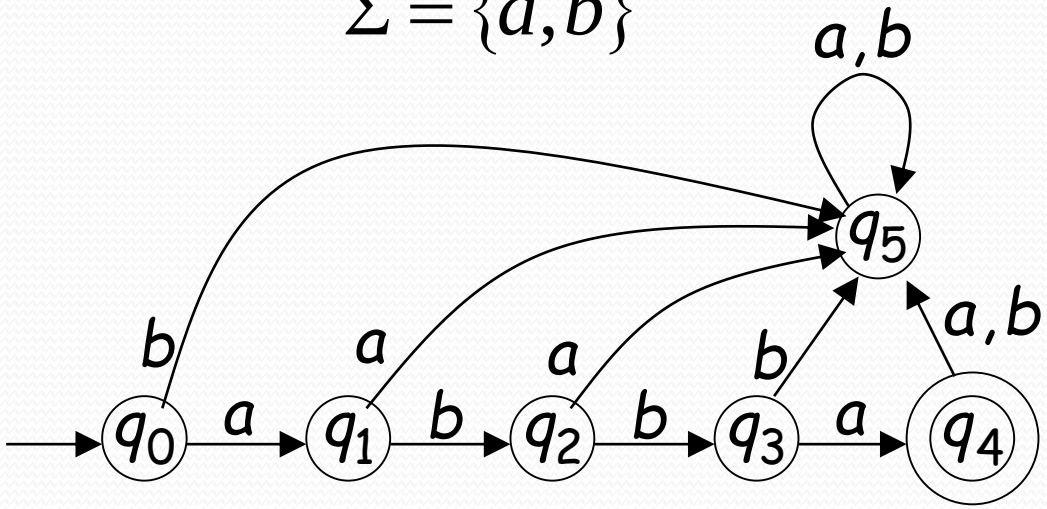
Example $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$



Input Alphabet Σ

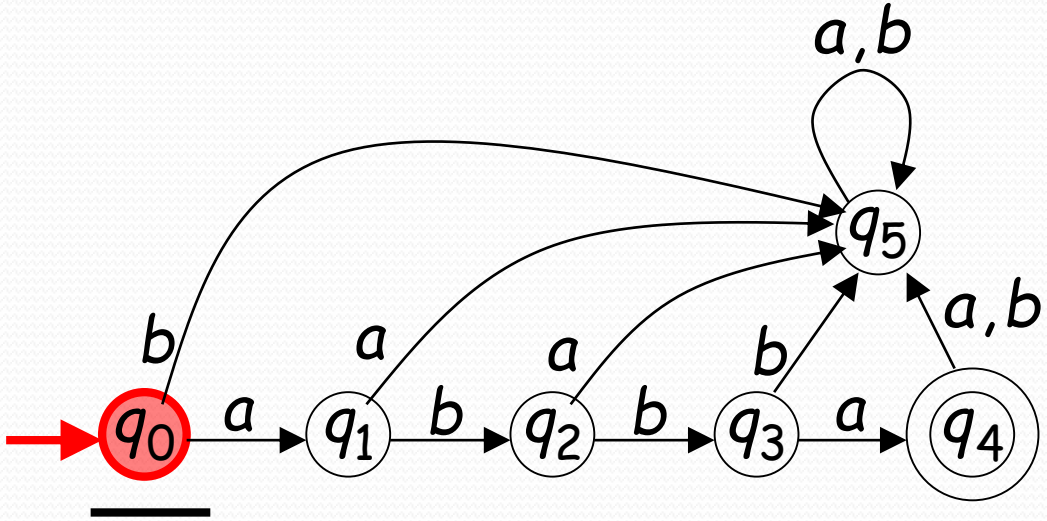
Example

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



Initial State q_0

Example

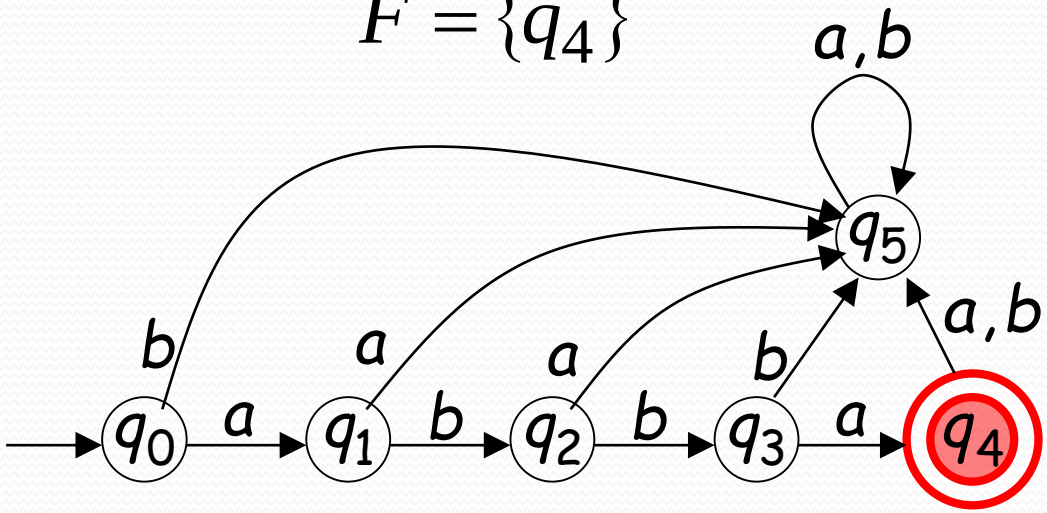


Set of Accepting States

$F \subseteq Q$

Example

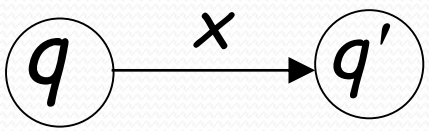
$F = \{q_4\}$



Transition Function

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

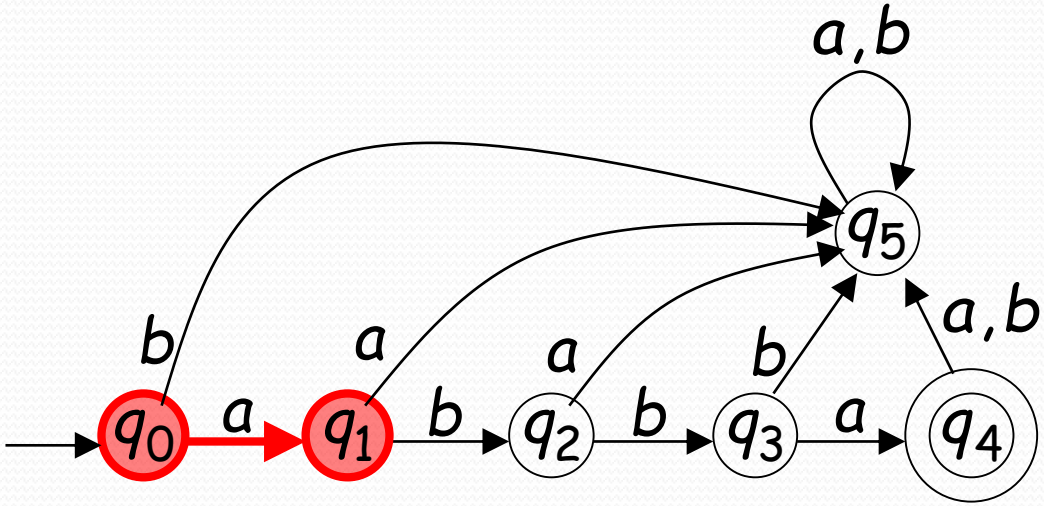
$$\delta(q, x) = q'$$



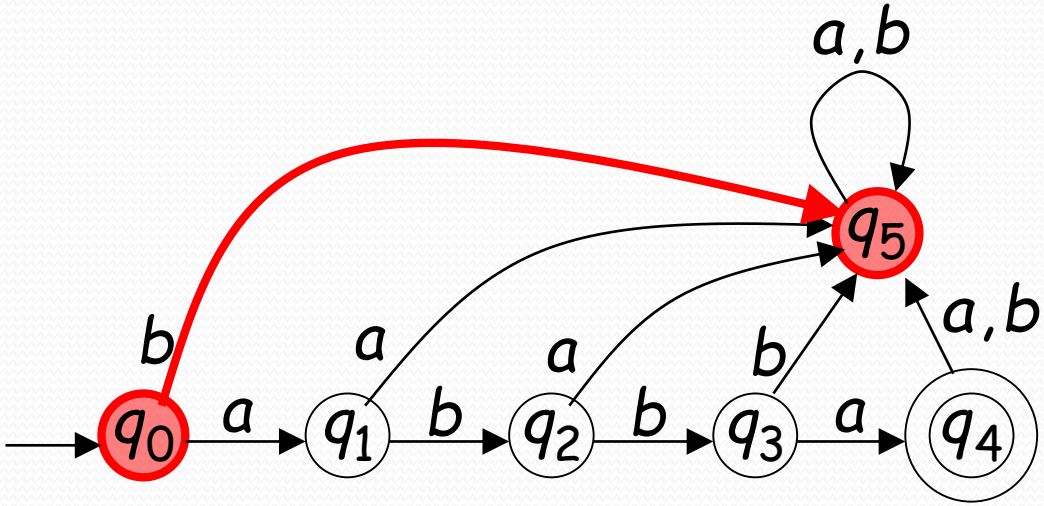
Describes the result of a transition from state q with symbol x

Example:

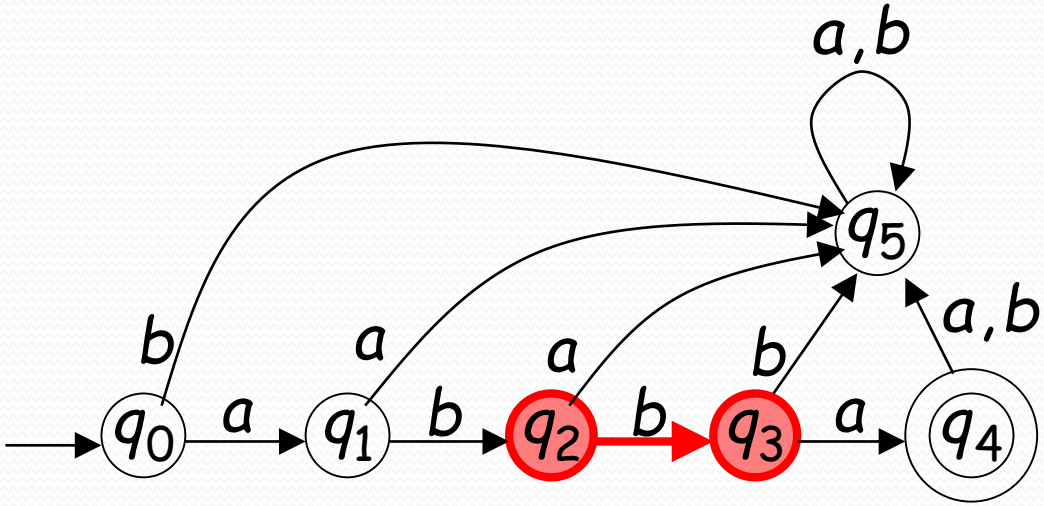
$$\delta(q_0, a) = q_1$$



$$\delta(q_0, b) = q_5$$



$$\delta(q_2, b) = q_3$$

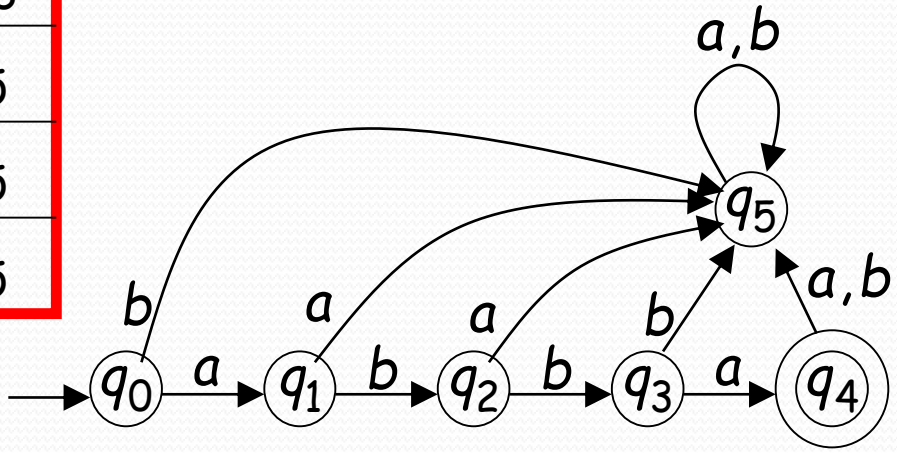


Transition Table for δ

symbols

states

δ	a	b
q_0	q_1	q_5
q_1	q_5	q_2
q_2	q_5	q_3
q_3	q_4	q_5
q_4	q_5	q_5
q_5	q_5	q_5



Extended Transition Function

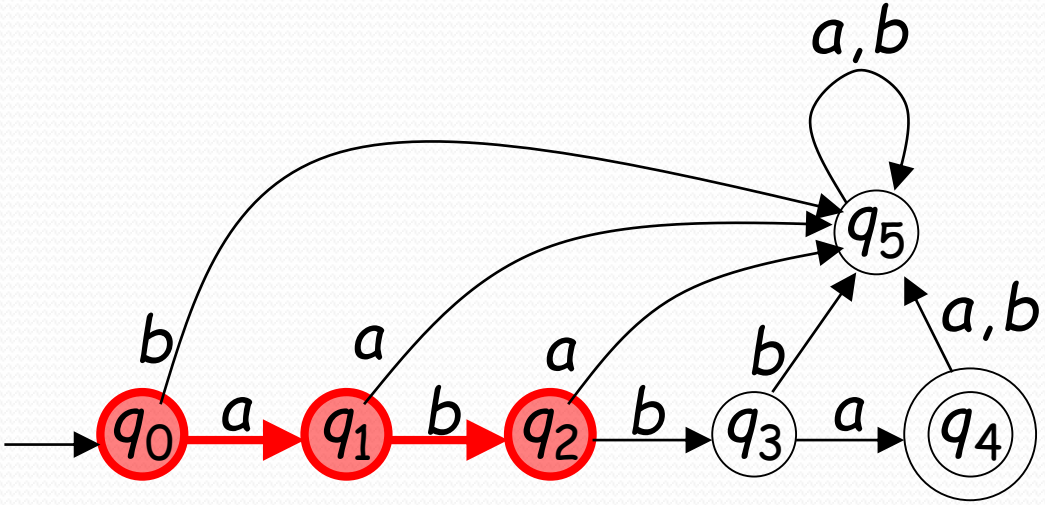
$$\delta^* : Q \times \Sigma^* \rightarrow Q$$

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

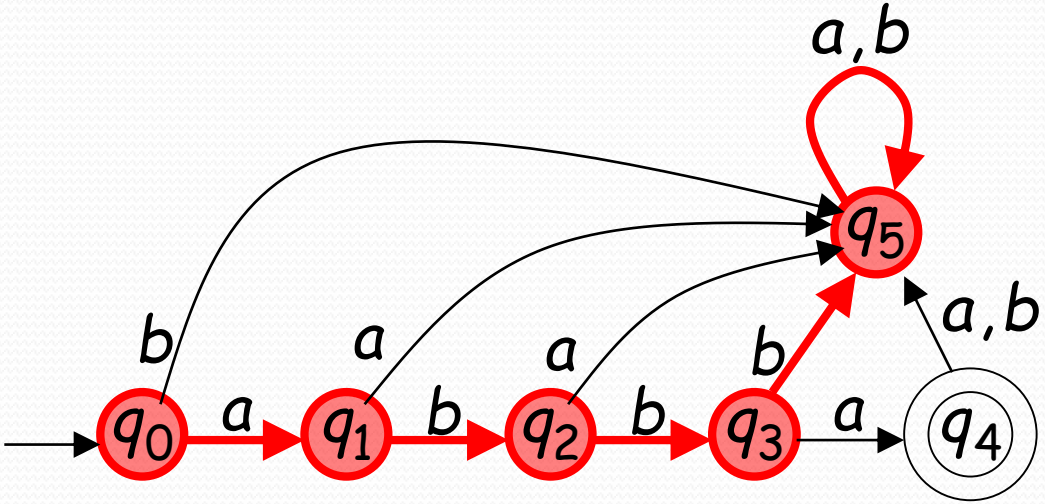
Describes the resulting state after scanning string w from state q

Example:

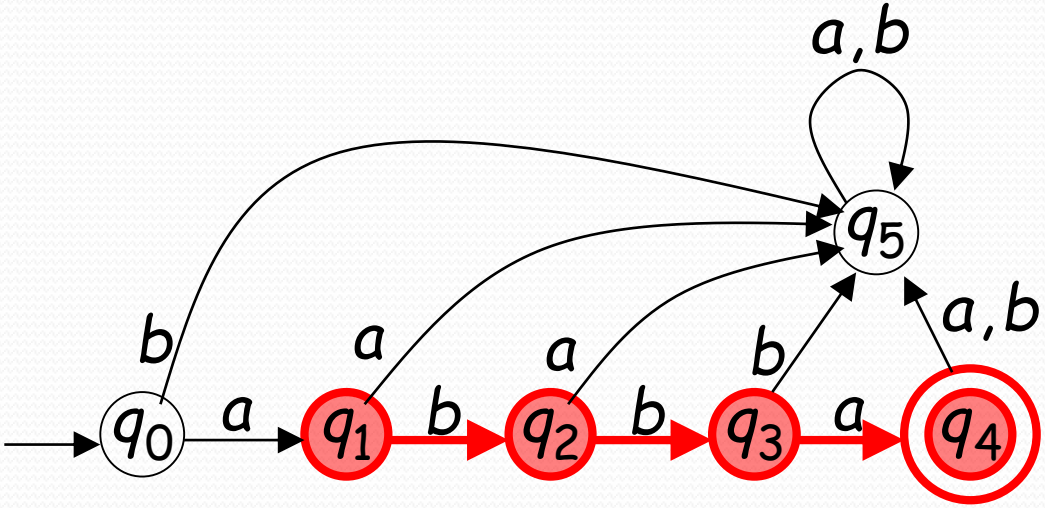
$$\delta^*(q_0, ab) = q_2$$



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



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



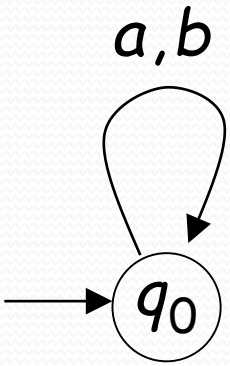
Special case:

for any state q

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

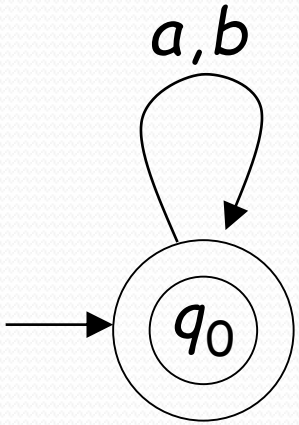
More DFA Examples

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



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

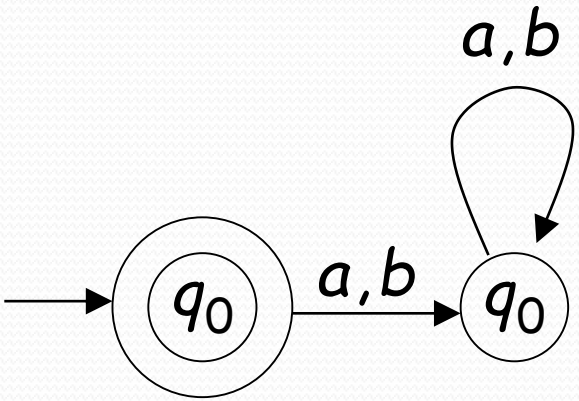
Empty language



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

All strings

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

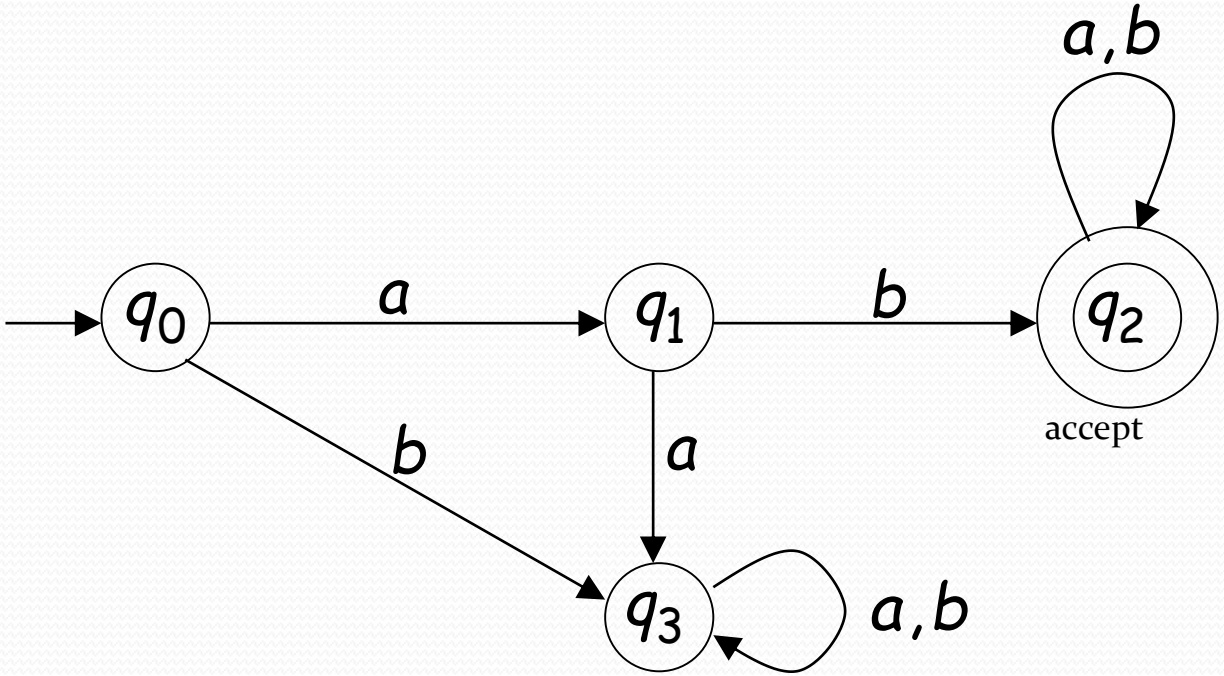


$$L(M) = \{\varepsilon\}$$

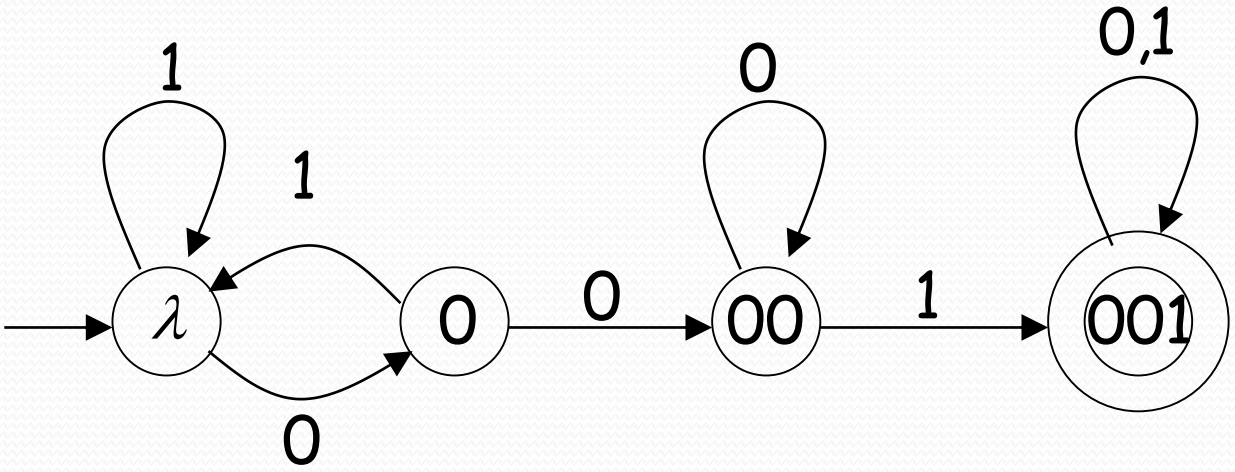
Language of the empty string

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

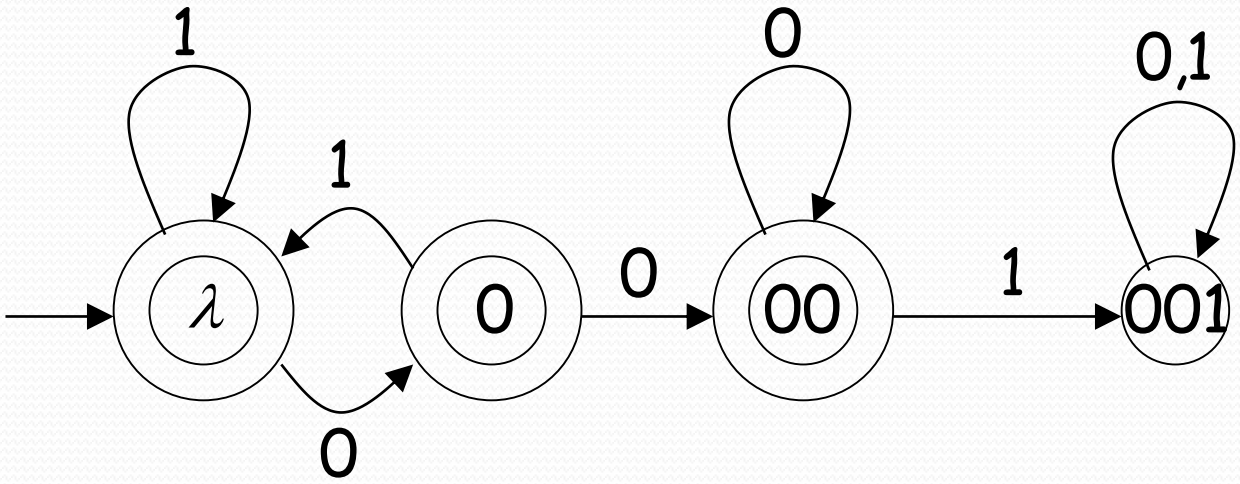
$$L(M) = \{ \text{ all strings with prefix } ab \}$$



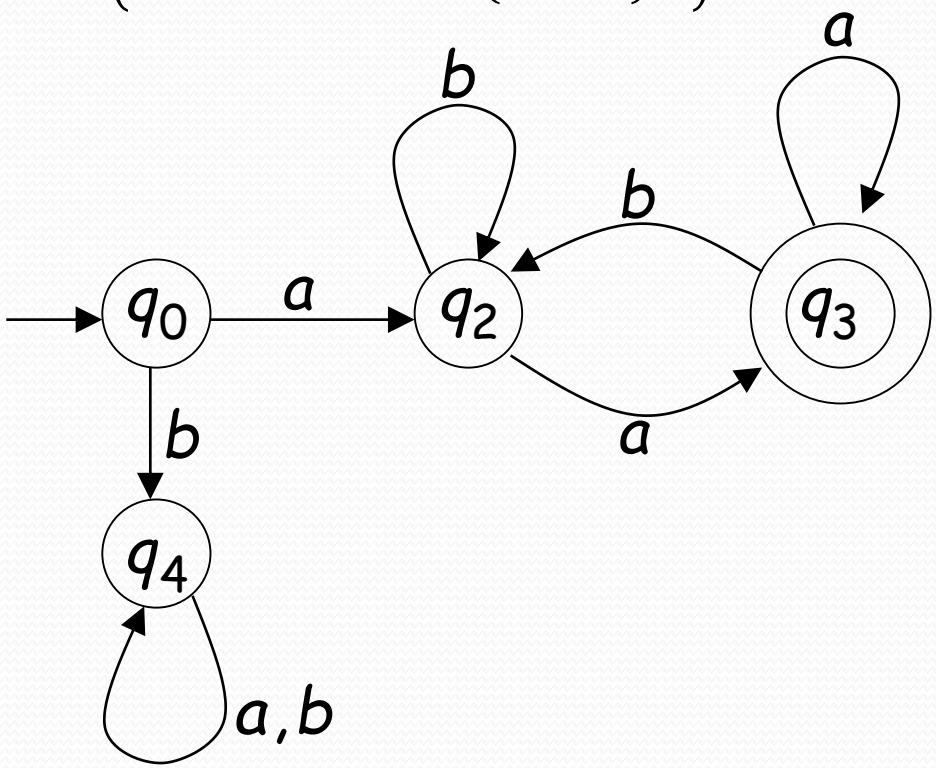
$L(M)$ = { all binary strings containing substring 001}



$L(M)$ = { all binary strings without substring 001 }

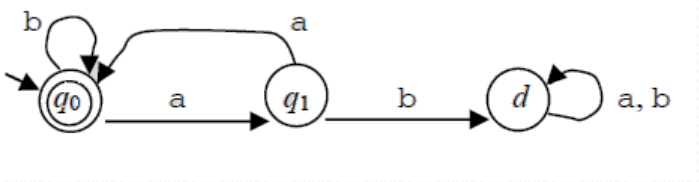


$$L(M) = \{awa : w \in \{a,b\}^*\}$$



Deterministic Finite State Machine

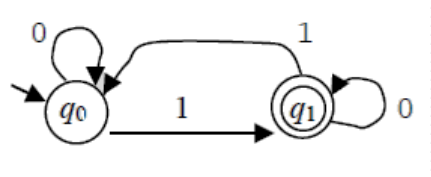
- **Example : Even Length Regions of a's**
- Let *L = {w ∈ {a, b}* : every a region in w is of even length}. L can be accepted by the DFSM M:*



- If *M sees a b in state q1, then there has been an a region whose length is odd. So, no matter what happens next, M* must reject. So it goes to the dead state *d*.

Deterministic Finite State Machine

- **Example : Checking for Odd Parity**
- Let *L = {w ∈{0, 1}* : w has odd parity}. A binary string has odd parity iff the number of 1's in it is odd. So L can* be accepted by the DFSM *M:*



Deterministic Finite State Machine

- **Theorem 5.1 DFSMs Halt**

- ***Theorem: Every DFSM M, on input w, halts after |w| steps.***

- ***Proof:*** *On input w, M executes some computation* $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_n$, *where*
  C_0 *is an initial configuration* and
  C_n *is of the form* (q, ε), *for some state* $q \in K_M$.
  C_n *is either an accepting or a rejecting configuration, so M will* halt when it reaches C_n.
  Each step in the computation consumes one character of w. So $n = |w|$.
  Thus M will halt after $|w|$ *steps.*