

PDAs Accept Context-Free Languages

Theorem:

$$\left\{ \begin{array}{l} \text{Context-Free} \\ \text{Languages} \\ (\text{Grammars}) \end{array} \right\} = \left\{ \begin{array}{l} \text{Languages} \\ \text{Accepted by} \\ \text{PDAs} \end{array} \right\}$$

Proof - Step 1:

$$\left\{ \begin{array}{l} \text{Context-Free} \\ \text{Languages} \\ (\text{Grammars}) \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Languages} \\ \text{Accepted by} \\ \text{PDAs} \end{array} \right\}$$

Convert any context-free grammar G
to a PDA M with: $L(G) = L(M)$

Proof - Step 2:

$$\left\{ \begin{array}{l} \text{Context-Free} \\ \text{Languages} \\ (\text{Grammars}) \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Languages} \\ \text{Accepted by} \\ \text{PDAs} \end{array} \right\}$$

Convert any PDA M to a context-free grammar G with: $L(G) = L(M)$

Proof - step 1

Convert

Context-Free Grammars
to
PDAs

Take an arbitrary context-free grammar G

We will convert G to a PDA M such that:

$$L(G) = L(M)$$

Conversion Procedure:

For each production in G

$$A \rightarrow w$$

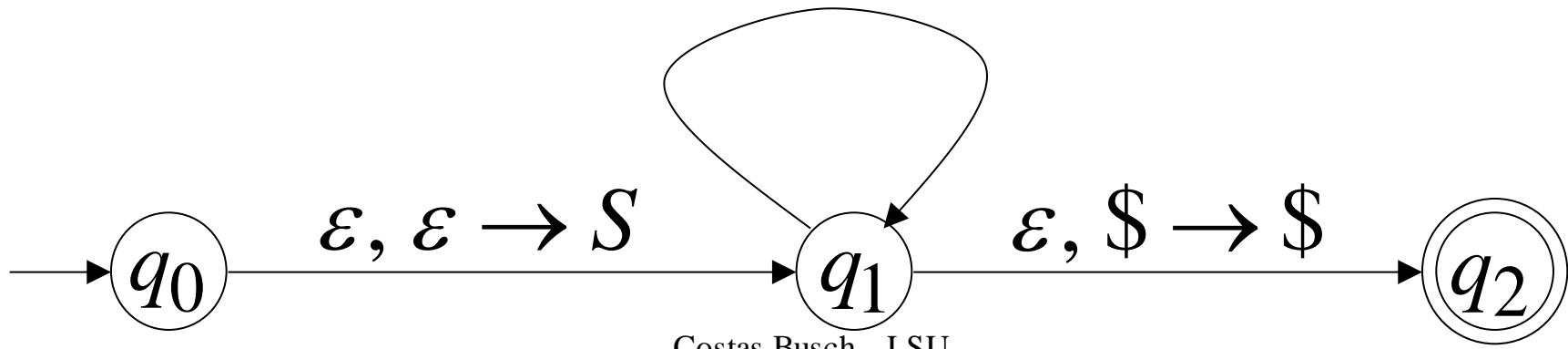
$$\epsilon, A \rightarrow w$$

For each terminal in G

$$a$$

$$a, a \rightarrow \epsilon$$

Add transitions



Grammar

$$S \rightarrow aSTb$$

$$S \rightarrow b$$

$$T \rightarrow Ta$$

$$T \rightarrow \varepsilon$$

Example

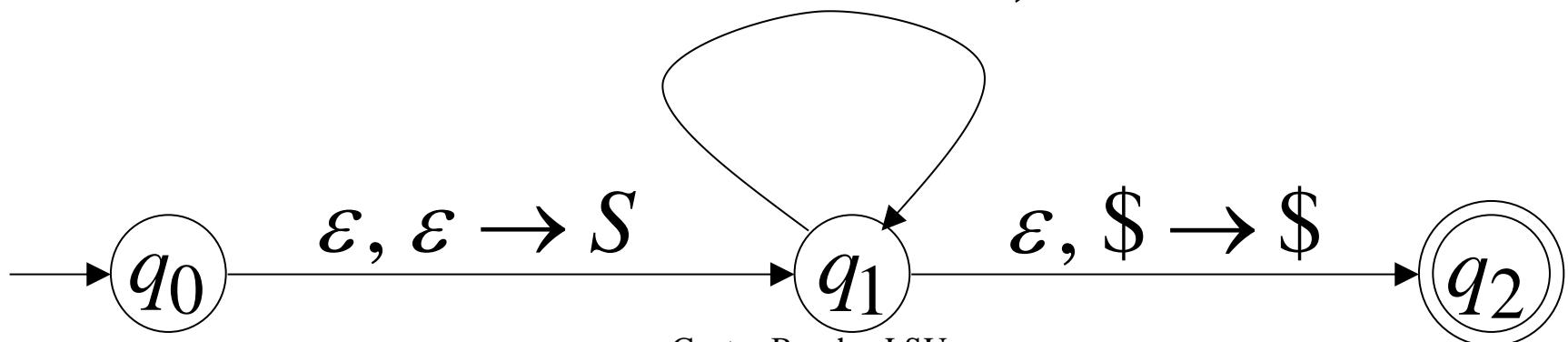
PDA

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

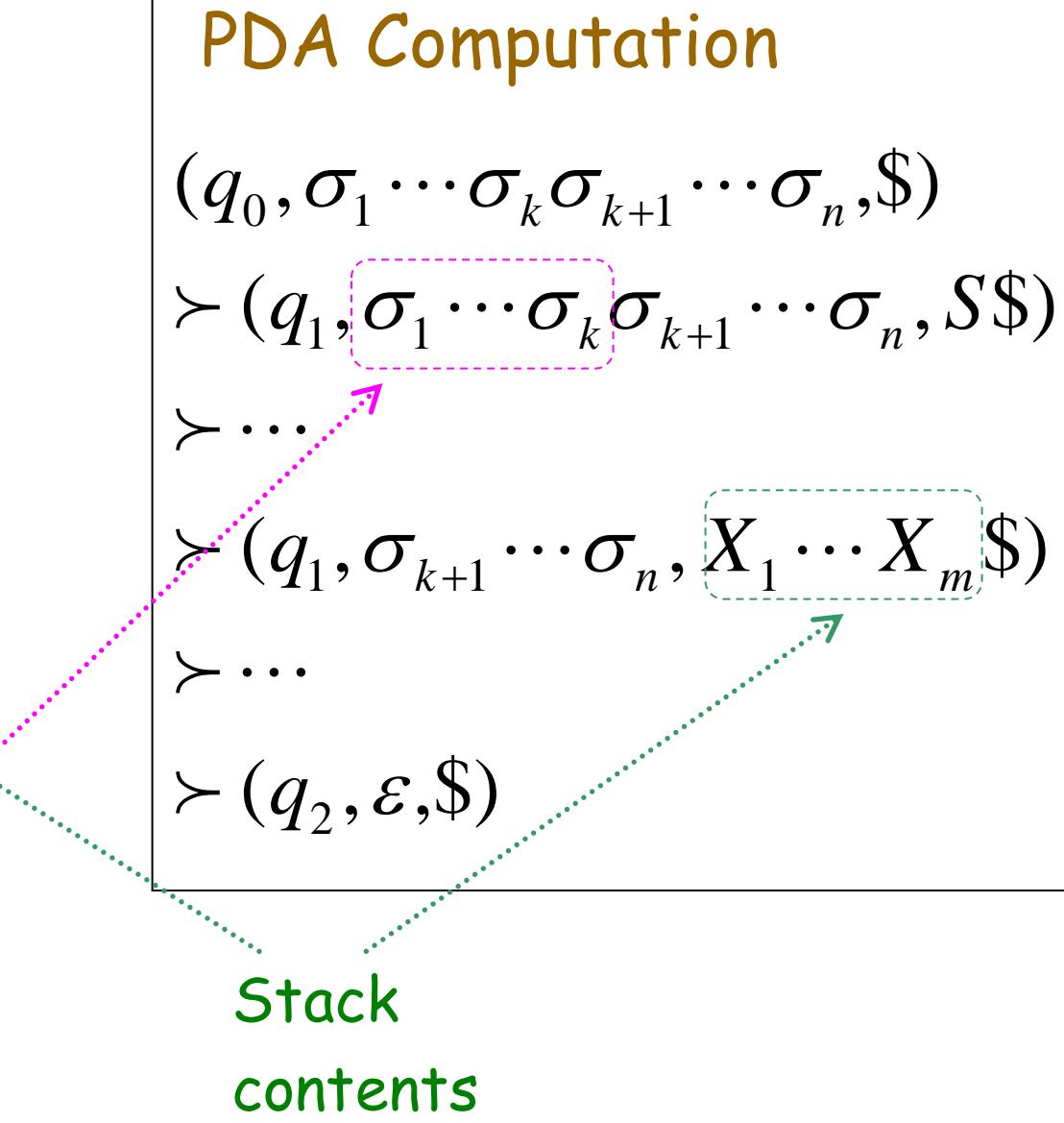
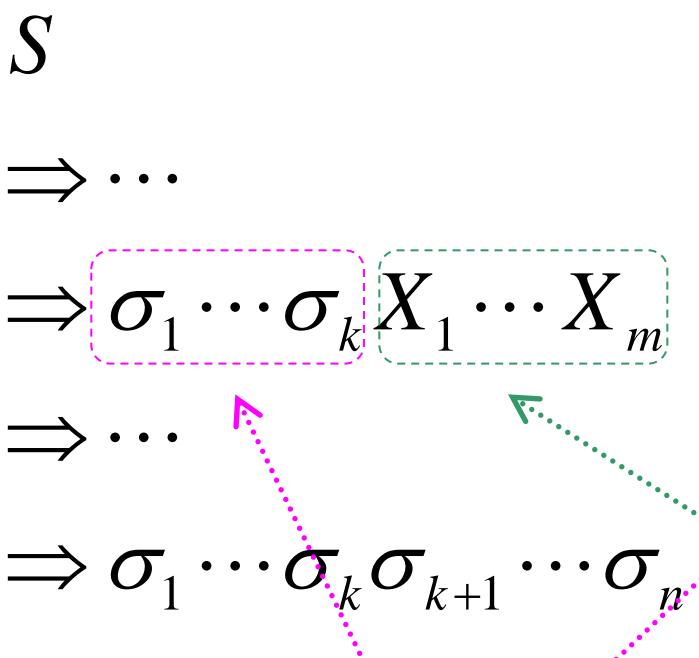
$$\varepsilon, T \rightarrow Ta \quad a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon \quad b, b \rightarrow \varepsilon$$



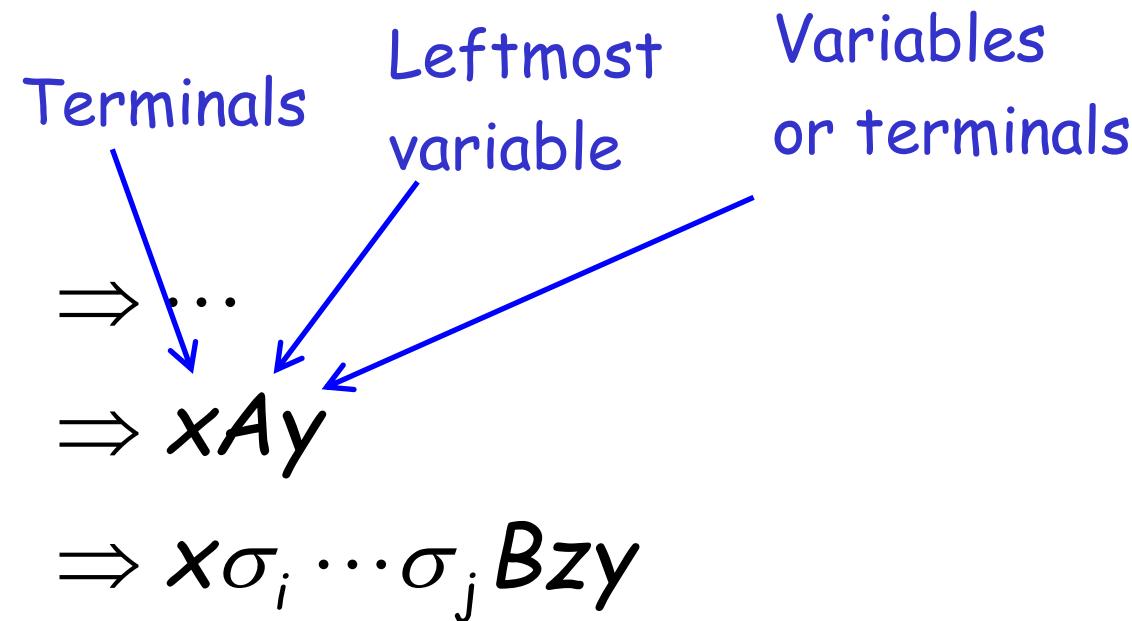
PDA simulates leftmost derivations

Grammar Leftmost Derivation

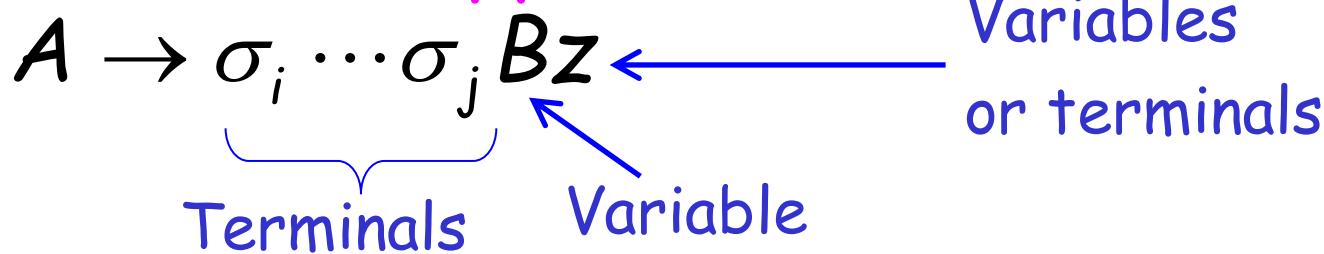


Grammar

Leftmost Derivation



Production applied



Grammar

Leftmost Derivation

$\Rightarrow \dots$

$\Rightarrow xAy$

$\Rightarrow x\sigma_i \cdots \sigma_j Bzy$

Production applied

$A \rightarrow \sigma_i \cdots \sigma_j Bz$

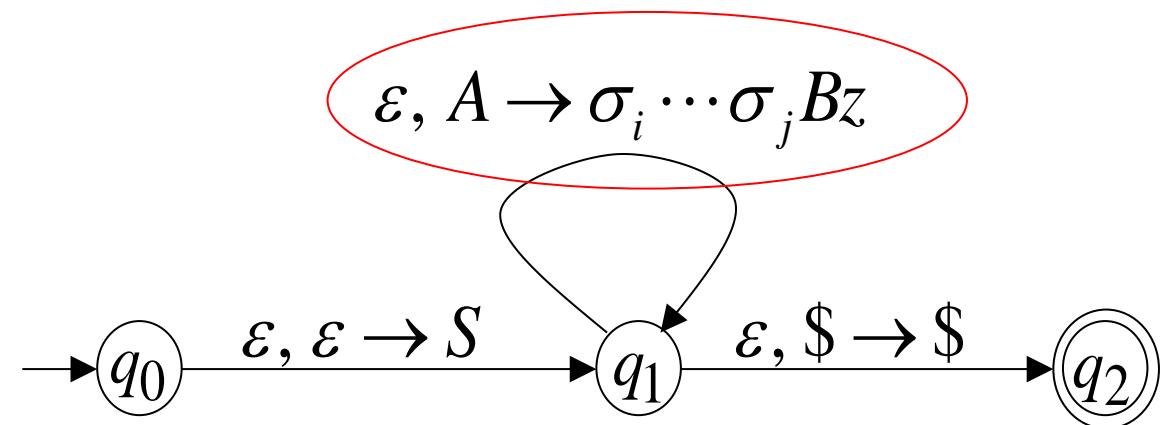
PDA Computation

$\gamma \dots$

$\succ (q_1, \sigma_i \cdots \sigma_n, Ay\$)$

$\succ (q_1, \sigma_i \cdots \sigma_n, \sigma_i \cdots \sigma_j Bzy\$)$

Transition applied



Grammar

Leftmost Derivation

$\Rightarrow \dots$

$\Rightarrow xAy$

$\Rightarrow x\sigma_i \dots \sigma_j Bzy$

$\succ \dots$

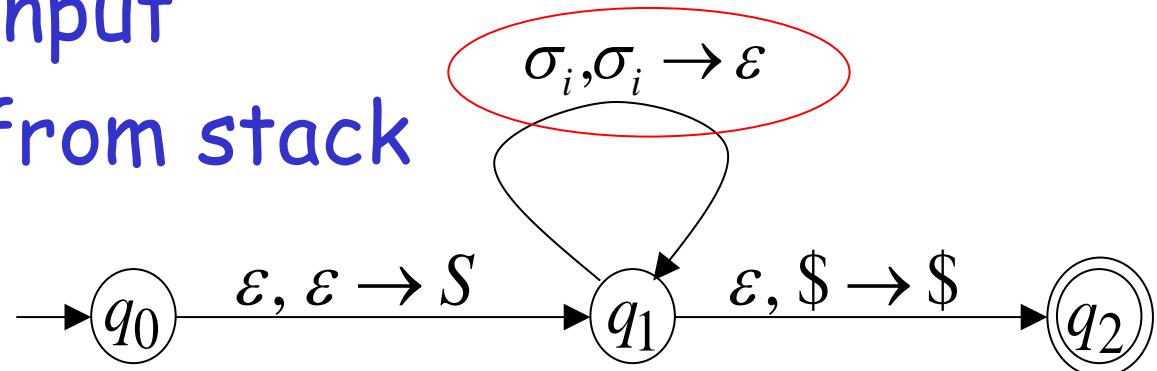
$\succ (q_1, \sigma_i \dots \sigma_n, Ay \$)$

$\succ (q_1, \sigma_i \dots \sigma_n, \sigma_i \dots \sigma_j Bzy \$)$

$\succ (q_1, \sigma_{i+1} \dots \sigma_n, \sigma_{i+1} \dots \sigma_j Bzy \$)$

Read σ_i from input
and remove it from stack

Transition applied



Grammar

Leftmost Derivation

$\Rightarrow \dots$

$\Rightarrow xAy$

$\Rightarrow x\sigma_i \dots \sigma_j Bzy$

All symbols $\sigma_i \dots \sigma_j$
have been removed
from top of stack

PDA Computation

$\vdash \dots$

$\vdash (q_1, \sigma_i \dots \sigma_n, Ay \$)$

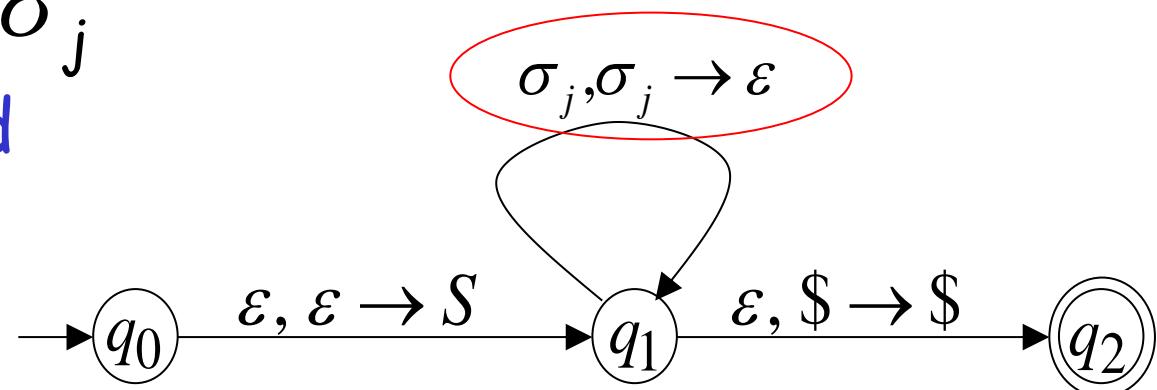
$\vdash (q_1, \sigma_i \dots \sigma_n, \sigma_i \dots \sigma_j Bzy \$)$

$\vdash (q_1, \sigma_{i+1} \dots \sigma_n, \sigma_{i+1} \dots \sigma_j Bzy \$)$

$\vdash \dots$

$\vdash (q_1, \sigma_{j+1} \dots \sigma_n, Bzy \$)$

Last Transition applied



The process repeats with the next leftmost variable

$\Rightarrow \dots$
 $\Rightarrow xAy$
 $\Rightarrow x\sigma_i \dots \sigma_j Bzy$
 $\Rightarrow x\sigma_i \dots \sigma_j \sigma_{j+1} \dots \sigma_k Cpzy$



$\succ \dots$
 $\succ (q_1, \sigma_{j+1} \dots \sigma_n, Bzy \$)$
 $\succ (q_1, \sigma_{j+1} \dots \sigma_n, \sigma_{j+1} \dots \sigma_k Cpzy \$)$
 $\succ \dots$
 $\succ (q_1, \sigma_{k+1} \dots \sigma_n, Cpzy \$)$

Production applied

$B \rightarrow \sigma_{j+1} \dots \sigma_k Cp$

And so on.....

Example:

Input

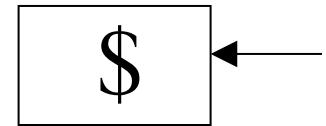
a	b	a	b
-----	-----	-----	-----



Time 0

$$\varepsilon, S \rightarrow aSTb$$

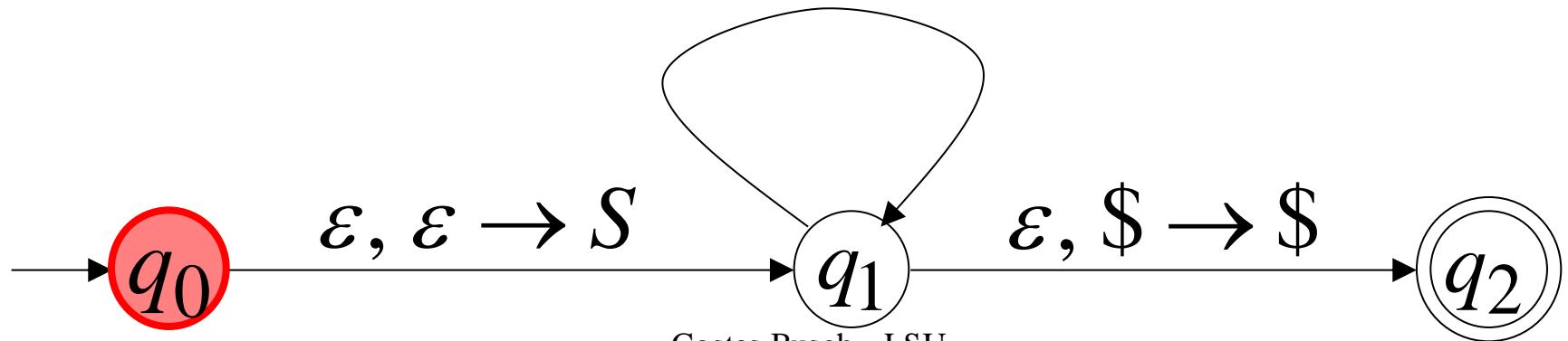
$$\varepsilon, S \rightarrow b$$



Stack

$$\varepsilon, T \rightarrow Ta \quad a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon \quad b, b \rightarrow \varepsilon$$



Derivation: S

Input

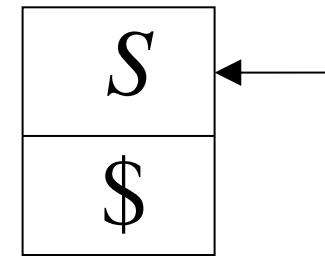
a	b	a	b
-----	-----	-----	-----



Time 1

$$\varepsilon, S \rightarrow aSTb$$

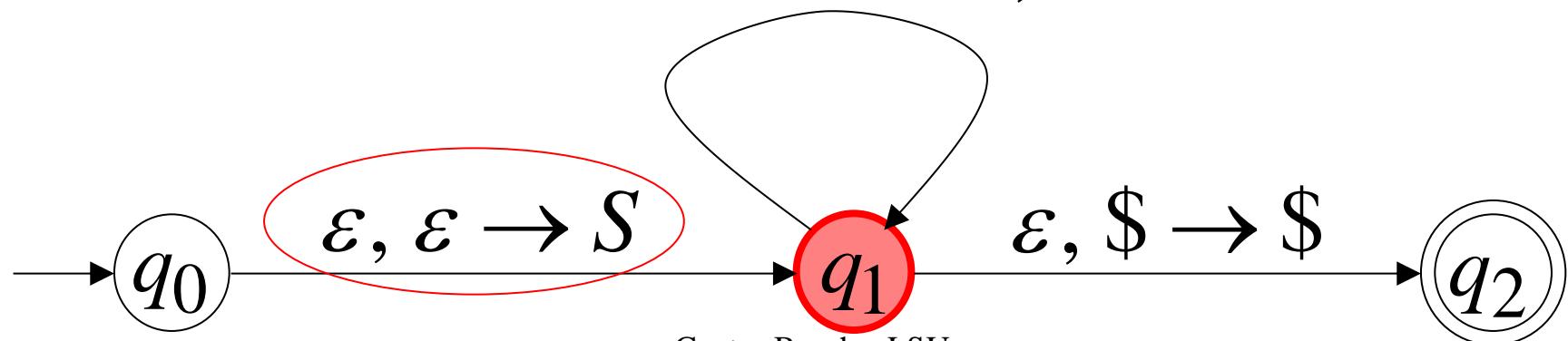
$$\varepsilon, S \rightarrow b$$



Stack

$$\varepsilon, T \rightarrow Ta \quad a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon \quad b, b \rightarrow \varepsilon$$



Derivation: $S \Rightarrow aSTb$

Input

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

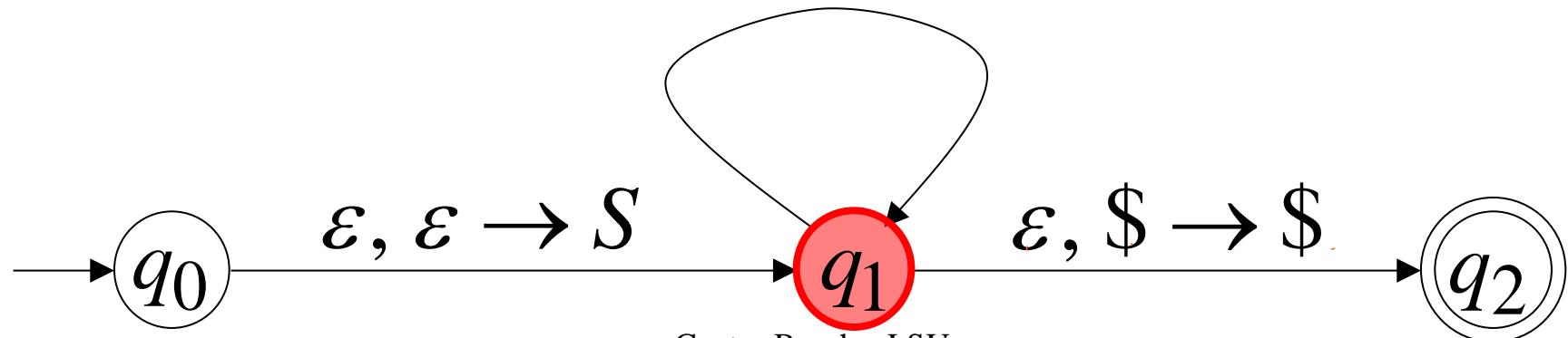
Time 2

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

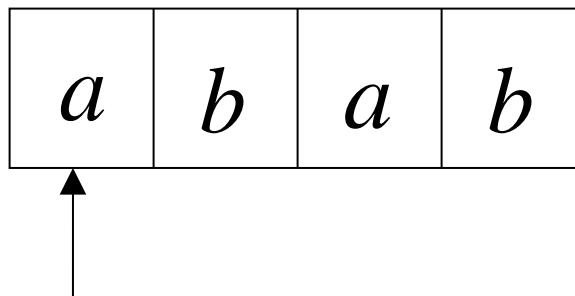
$$\varepsilon, T \rightarrow Ta \quad a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon \quad b, b \rightarrow \varepsilon$$



Derivation: $S \Rightarrow aSTb$

Input



Time 3

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

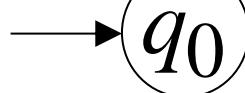
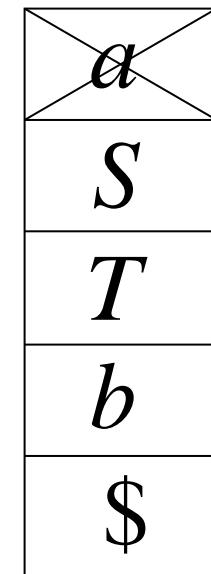
$$\varepsilon, T \rightarrow Ta$$

$a, a \rightarrow \varepsilon$

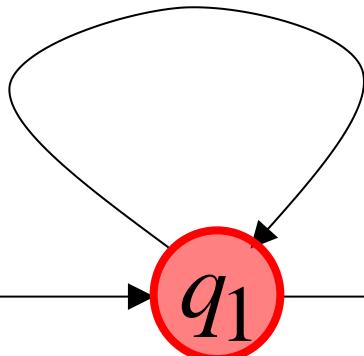
$$\varepsilon, T \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$

Stack



$$\varepsilon, \varepsilon \rightarrow S$$

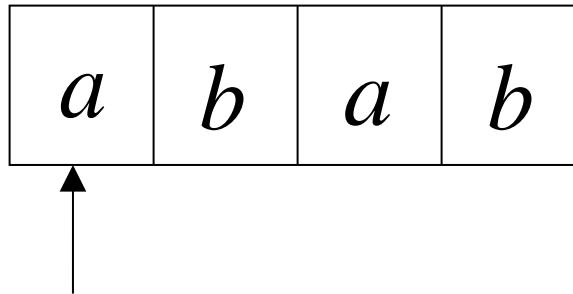


$$\varepsilon, \$ \rightarrow \$$$



Derivation: $S \Rightarrow aSTb \Rightarrow abTb$

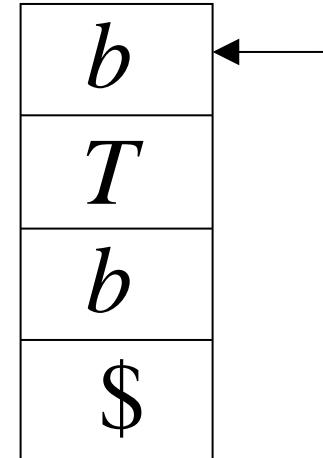
Input



Time 4

$$\varepsilon, S \rightarrow aSTb$$

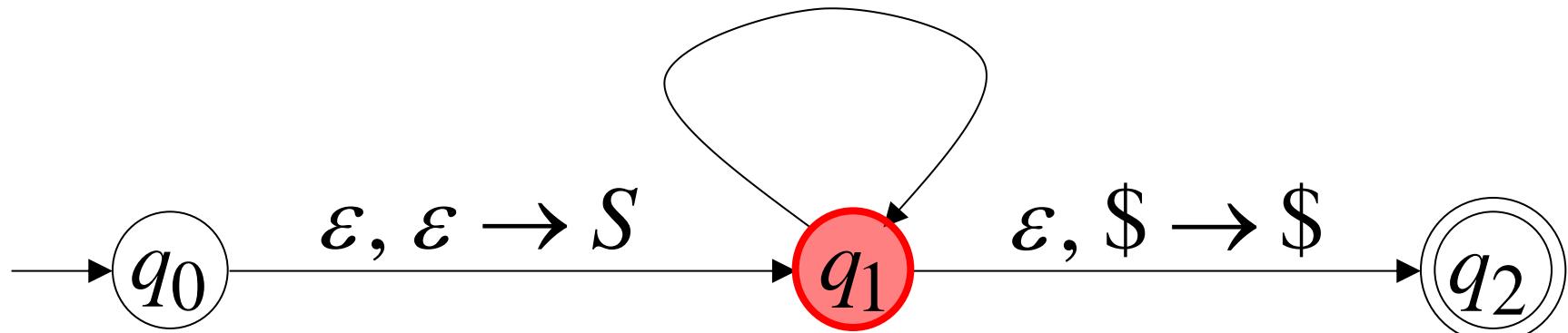
$$\varepsilon, S \rightarrow b$$



Stack

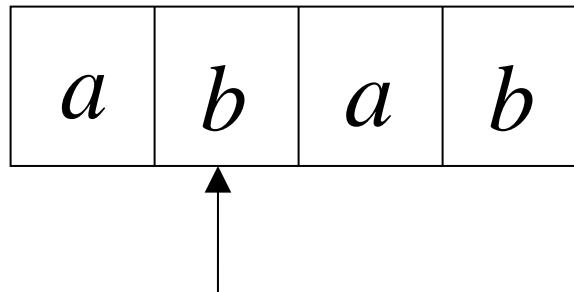
$$\varepsilon, T \rightarrow Ta \quad a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon \quad b, b \rightarrow \varepsilon$$



Derivation: $S \Rightarrow aSTb \Rightarrow abTb$

Input



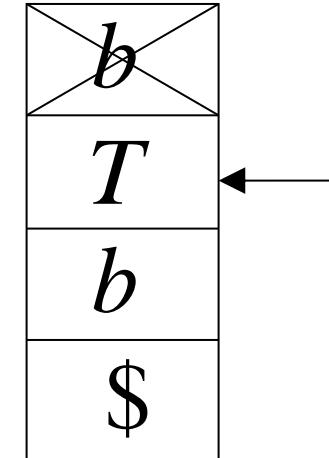
Time 5

$$\varepsilon, S \rightarrow aSTb$$

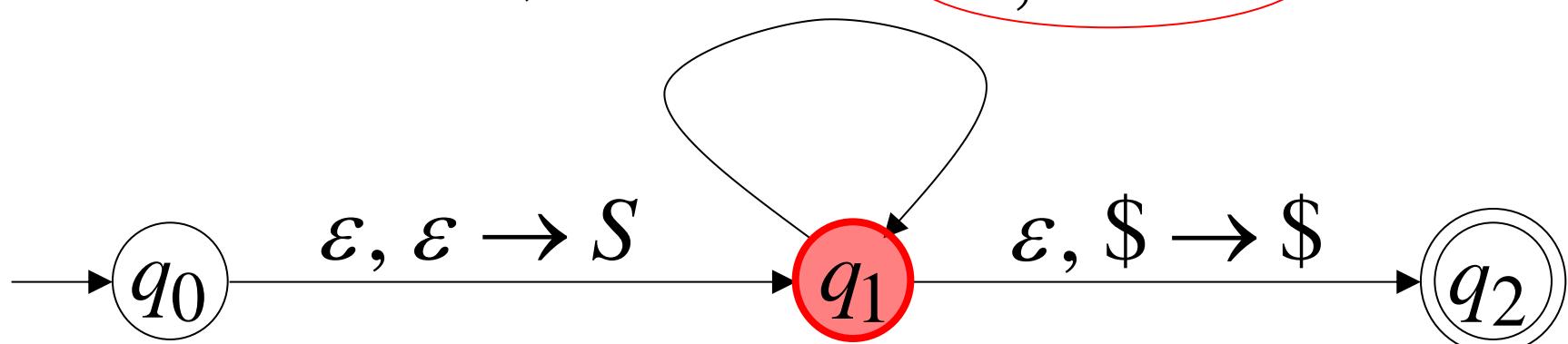
$$\varepsilon, S \rightarrow b$$

$$\varepsilon, T \rightarrow Ta \quad a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon \quad b, b \rightarrow \varepsilon$$

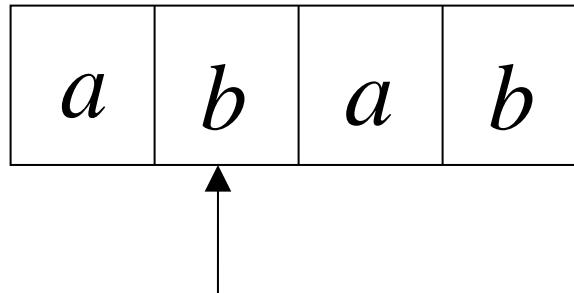


Stack



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab$

Input



Time 6

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

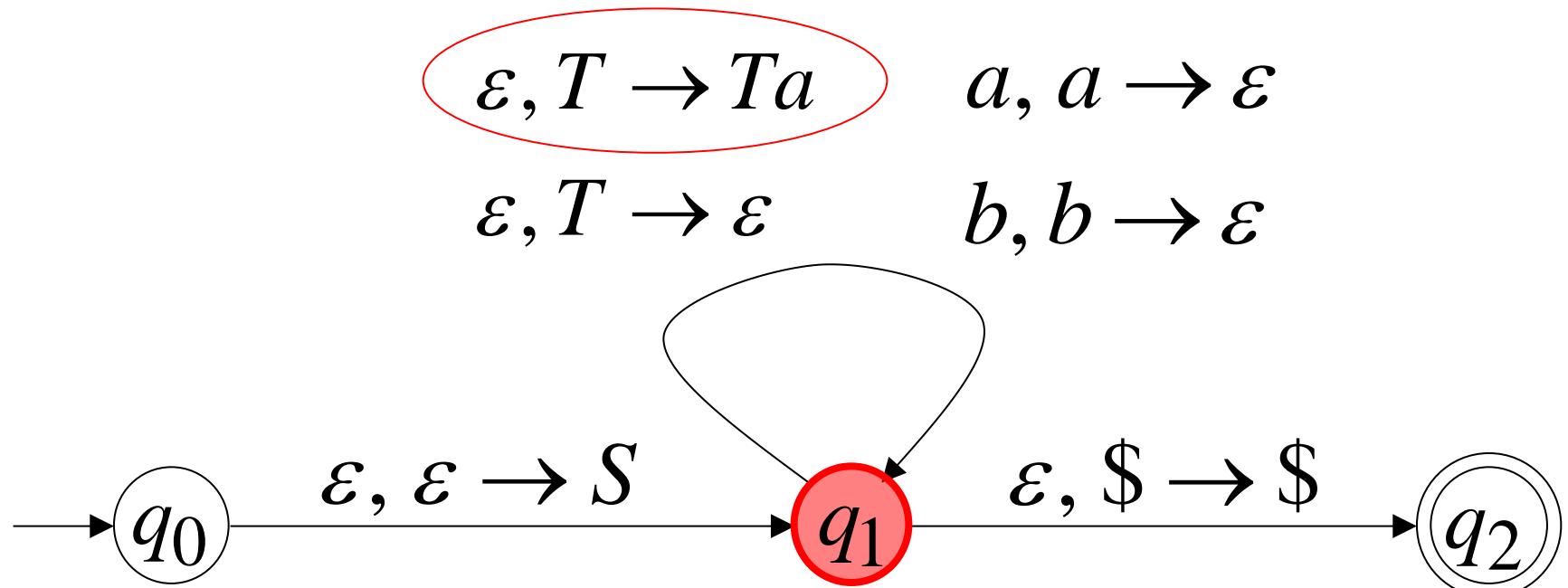
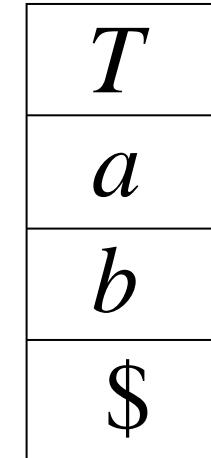
$$\varepsilon, T \rightarrow Ta$$

$$a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon$$

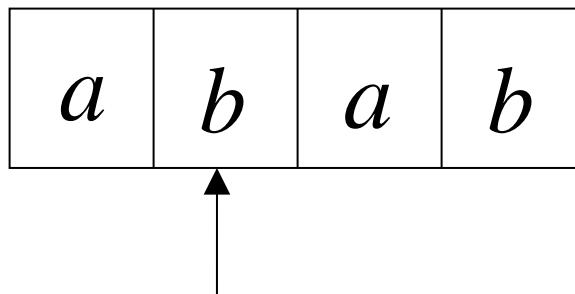
$$b, b \rightarrow \varepsilon$$

Stack



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab \Rightarrow abab$

Input



Time 7

$$\varepsilon, S \rightarrow aSTb$$

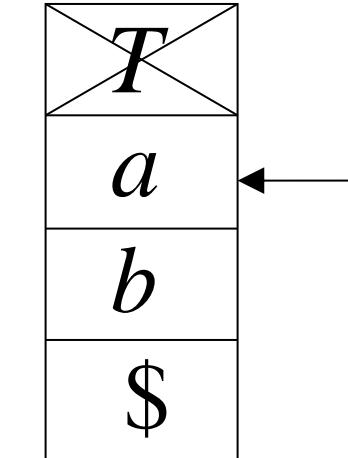
$$\varepsilon, S \rightarrow b$$

$$\varepsilon, T \rightarrow Ta$$

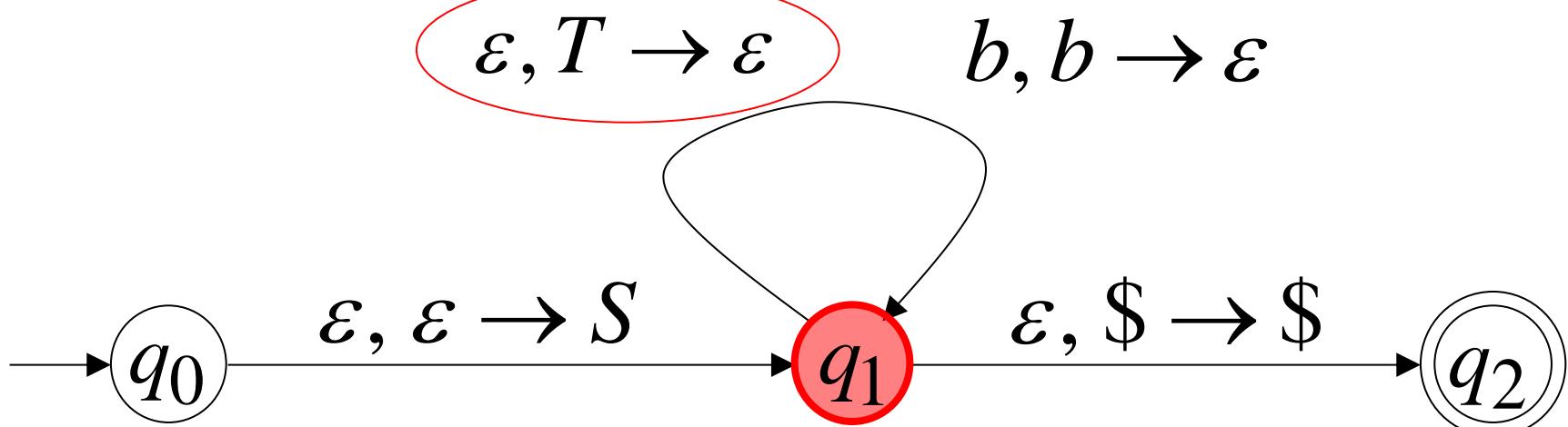
$$a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



Stack



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab \Rightarrow abab$

Input

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



Time 8

$$\varepsilon, S \rightarrow aSTb$$

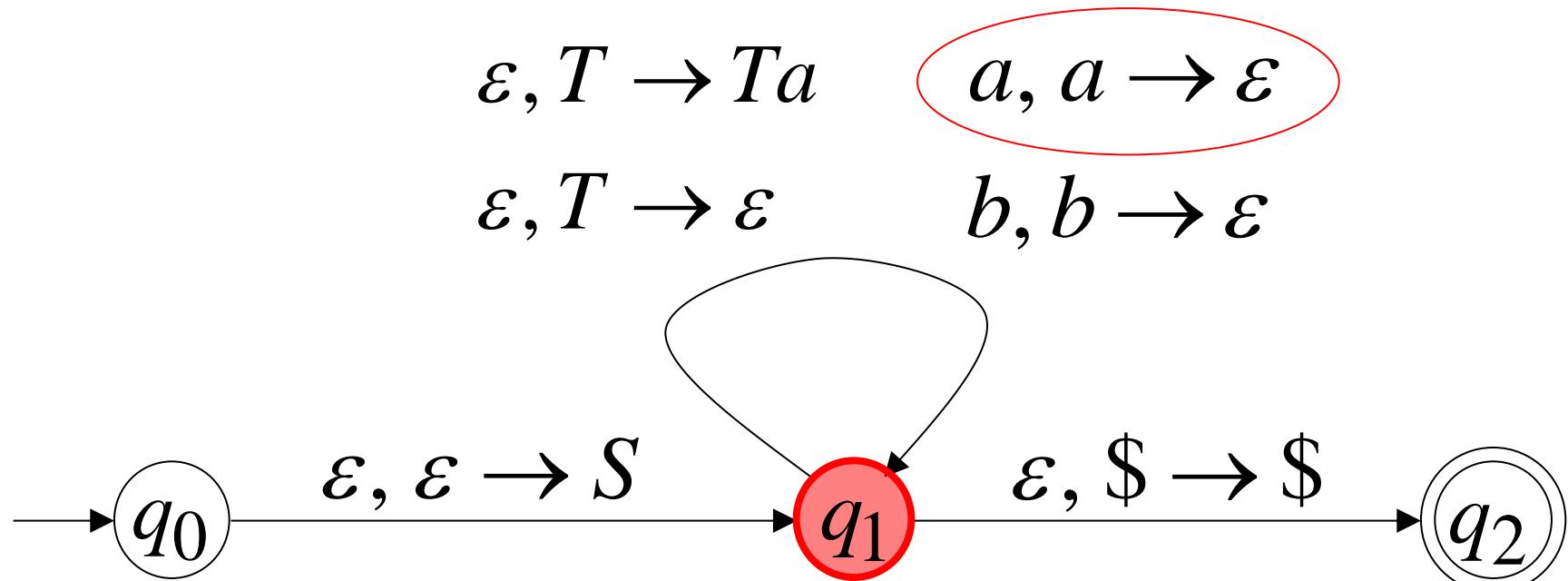
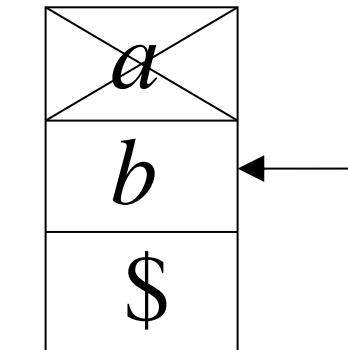
$$\varepsilon, S \rightarrow b$$

$$\varepsilon, T \rightarrow Ta$$

$$a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon$$

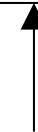
$$b, b \rightarrow \varepsilon$$



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab \Rightarrow abab$

Input

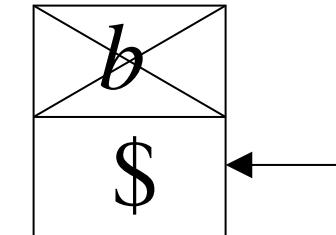
a	b	a	b
-----	-----	-----	-----



Time 9

$$\varepsilon, S \rightarrow aSTb$$

$$\varepsilon, S \rightarrow b$$

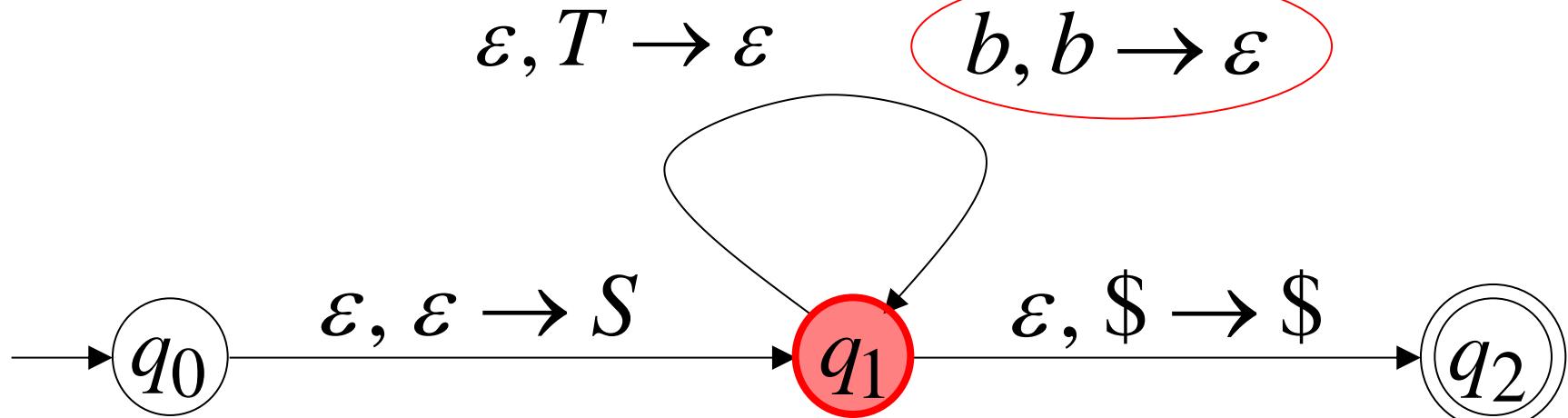


Stack

$$\varepsilon, T \rightarrow Ta \quad a, a \rightarrow \varepsilon$$

$$\varepsilon, T \rightarrow \varepsilon$$

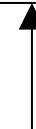
$$b, b \rightarrow \varepsilon$$



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow abTab \Rightarrow abab$

Input

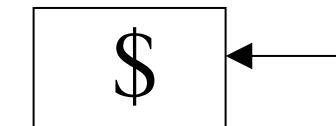
a	b	a	b
-----	-----	-----	-----



Time 10

$$\varepsilon, S \rightarrow aSTb$$

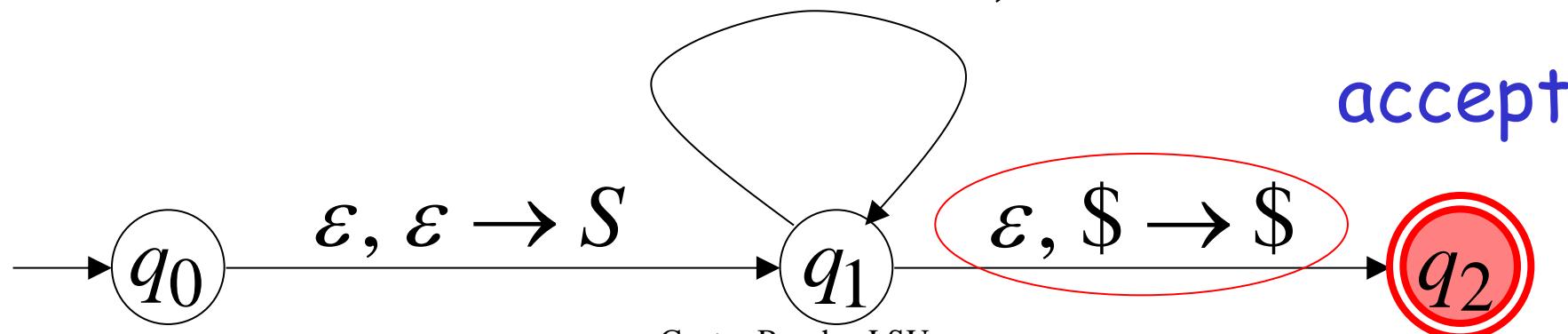
$$\varepsilon, S \rightarrow b$$



Stack

$$\varepsilon, T \rightarrow Ta \quad a, a \rightarrow \varepsilon$$

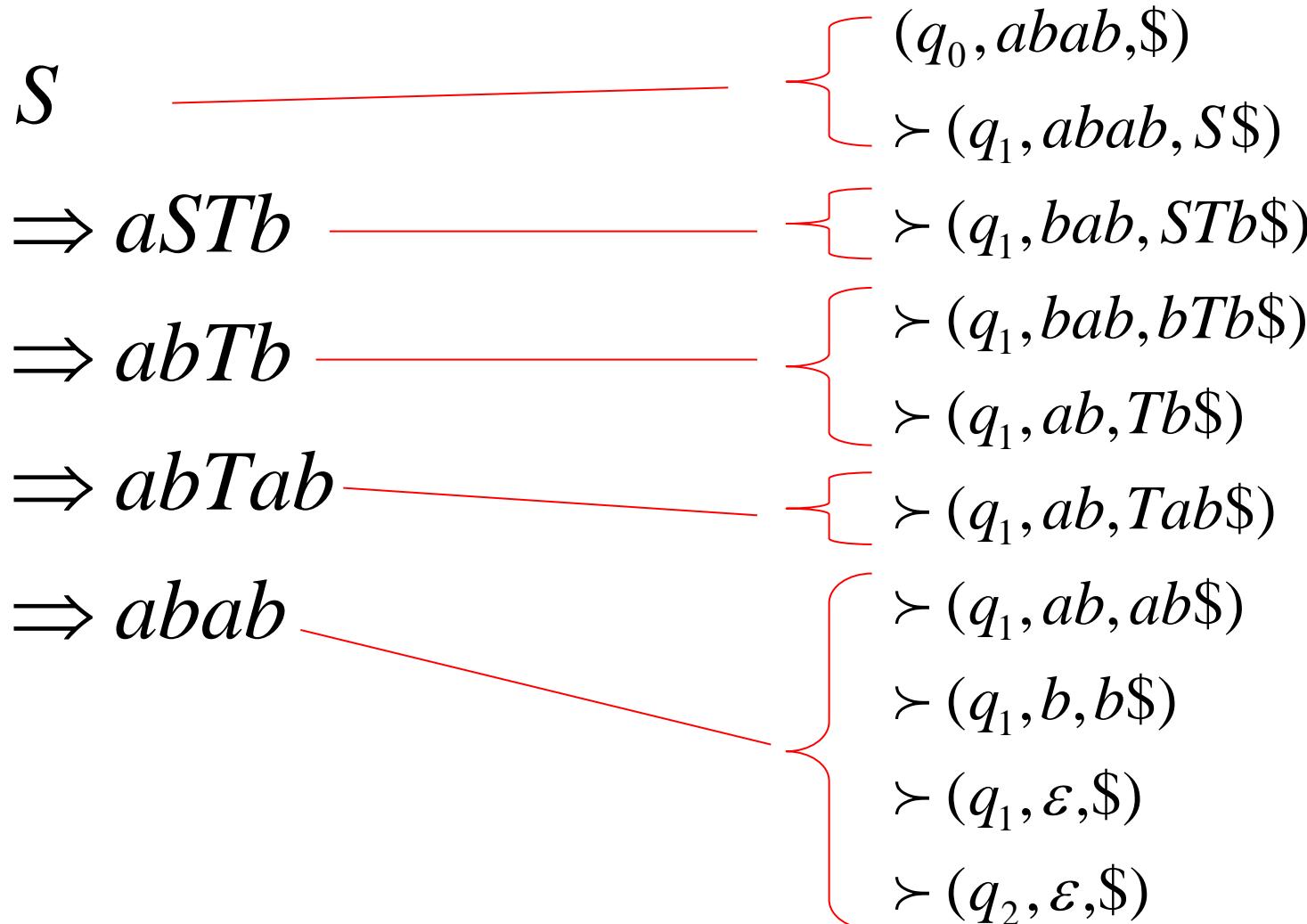
$$\varepsilon, T \rightarrow \varepsilon \quad b, b \rightarrow \varepsilon$$



Grammar

Leftmost Derivation

PDA Computation

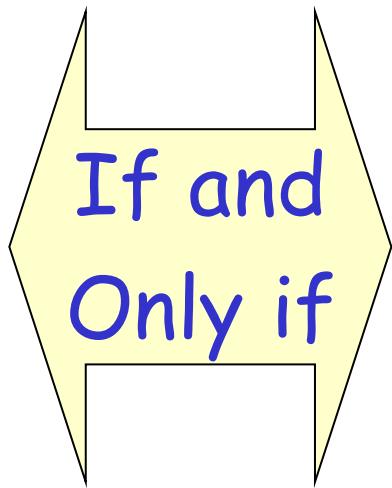


In general, it can be shown that:

Grammar G

generates
string w

$$S \xrightarrow{*} w$$



PDA M
accepts w

$$(q_0, w, \$) \xrightarrow{*} (q_2, \epsilon, \$)$$

Therefore $L(G) = L(M)$

Proof - step 2

Convert
PDAs
to
Context-Free Grammars

Take an arbitrary PDA M

We will convert M

to a context-free grammar G such that:

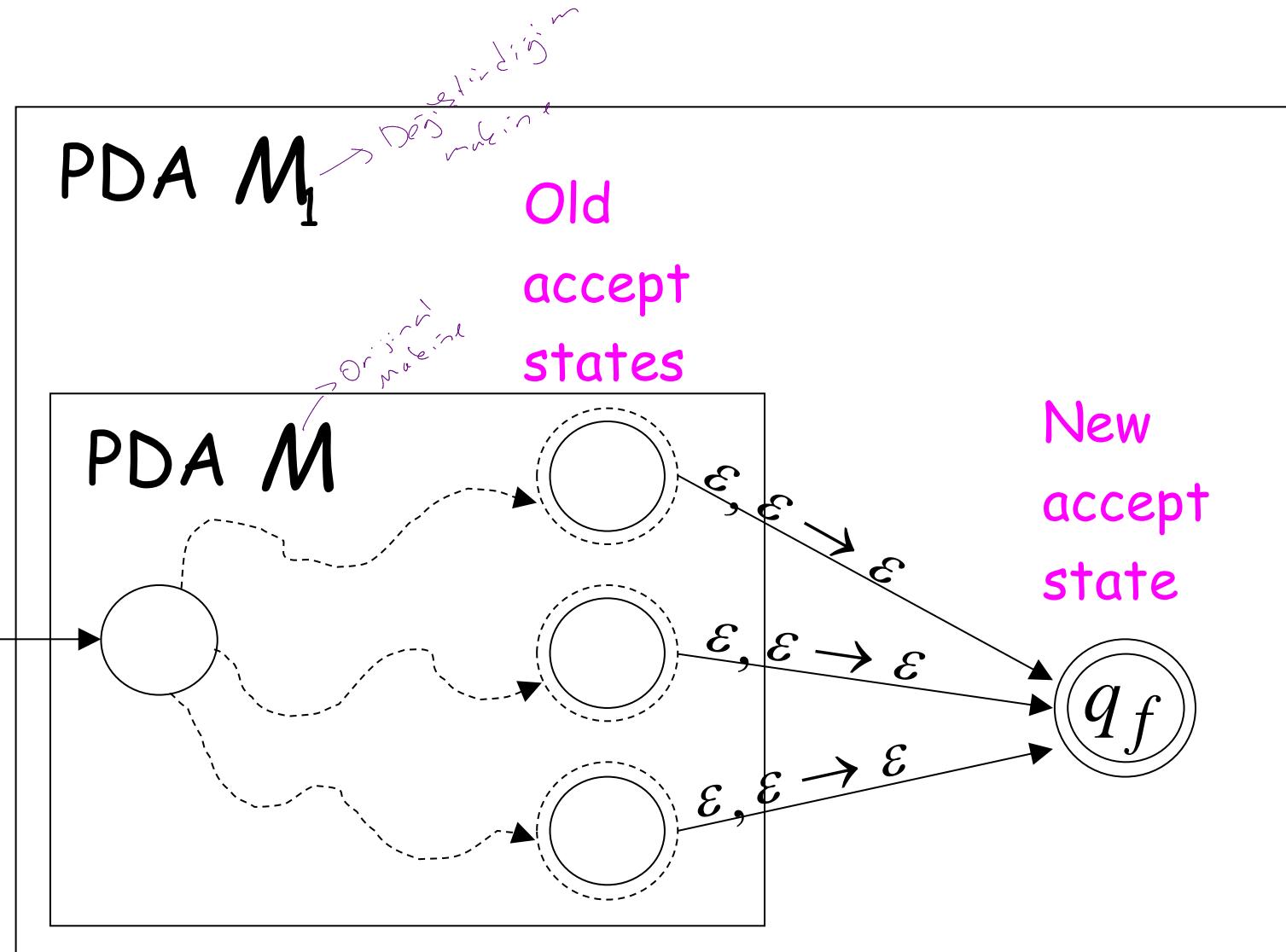
$$L(M) = L(G)$$

First modify PDA M so that:

*(,)
onem
LHS)*

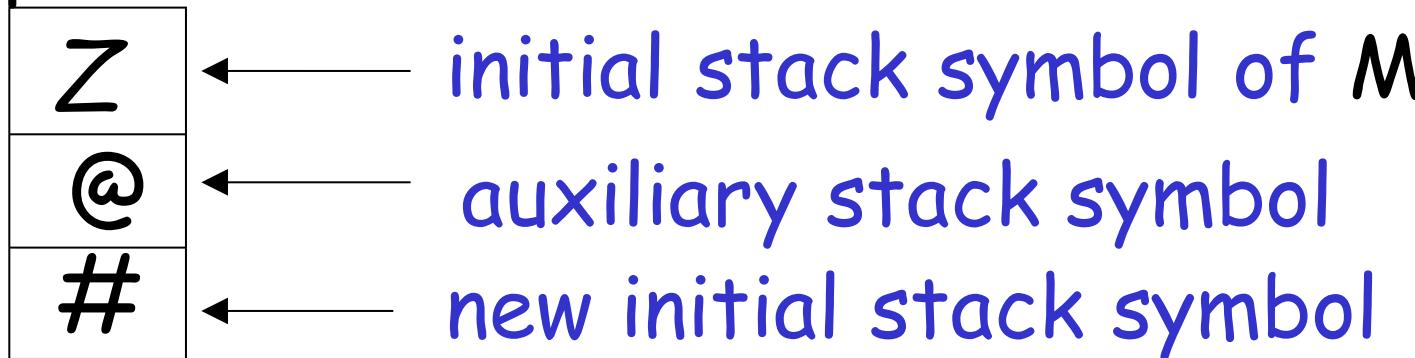
1. The PDA has a single accept state
2. Use new initial stack symbol $\#$
3. On acceptance the stack contains only stack symbol $\#$ (this symbol is not used in any transition)
4. Each transition either pushes a symbol or pops a symbol but not both together

1. The PDA has a single accept state

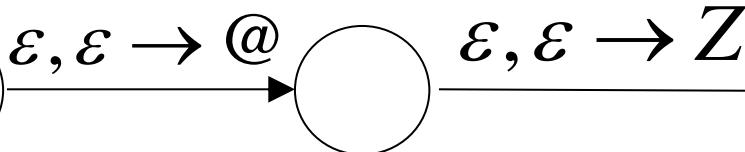


2. Use new initial stack symbol

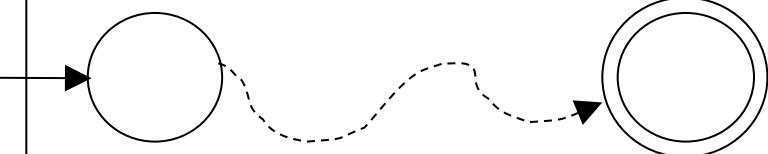
Top of stack



PDA M_2

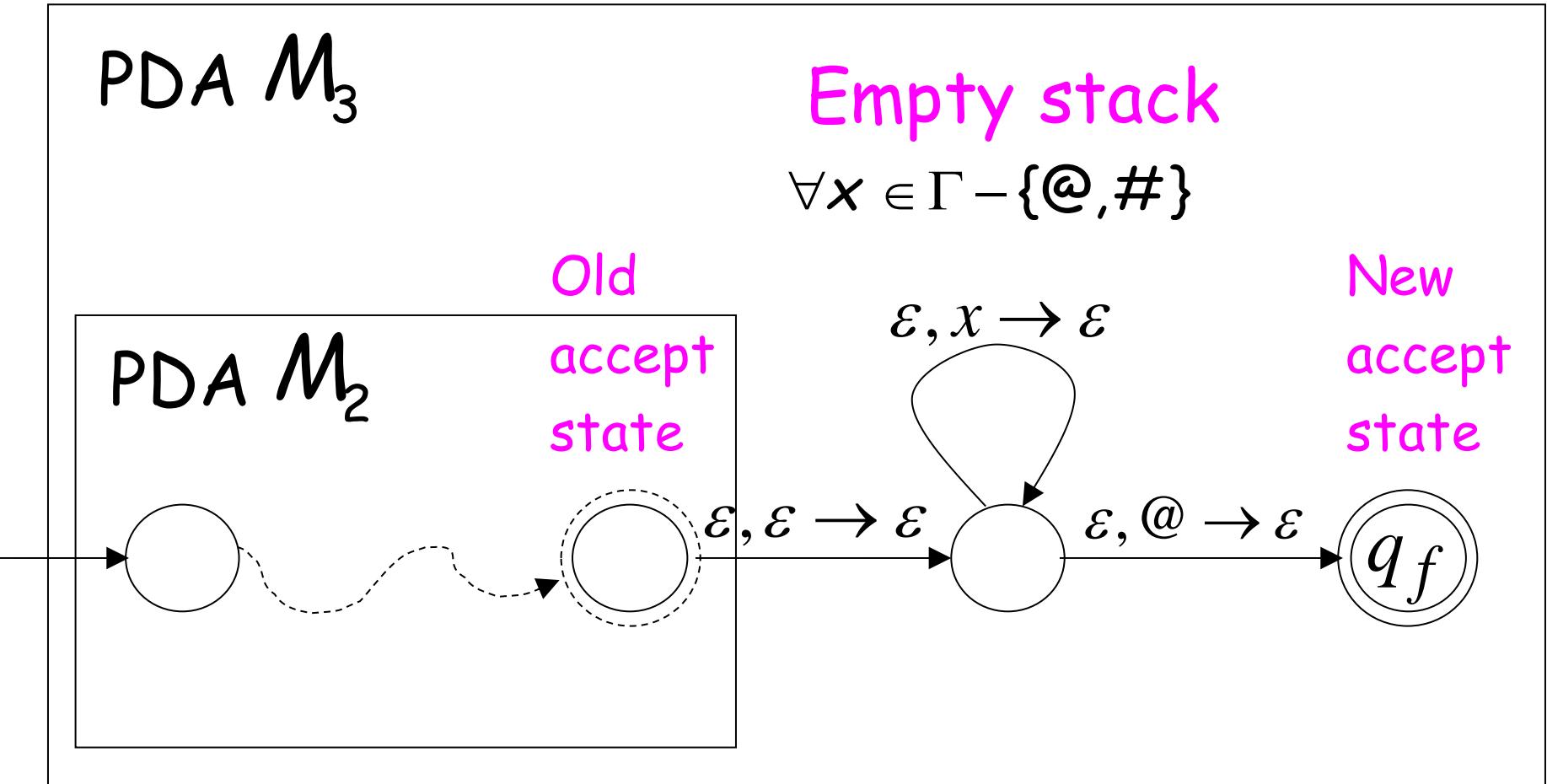


PDA M_1



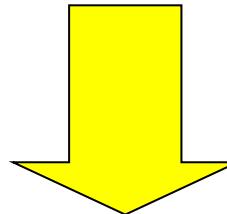
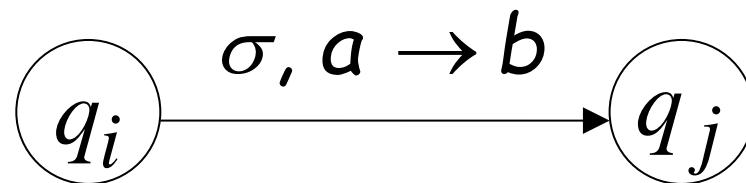
M_1 still thinks that Z is the initial stack

3. On acceptance the stack contains only stack symbol #
(this symbol is not used in any transition)

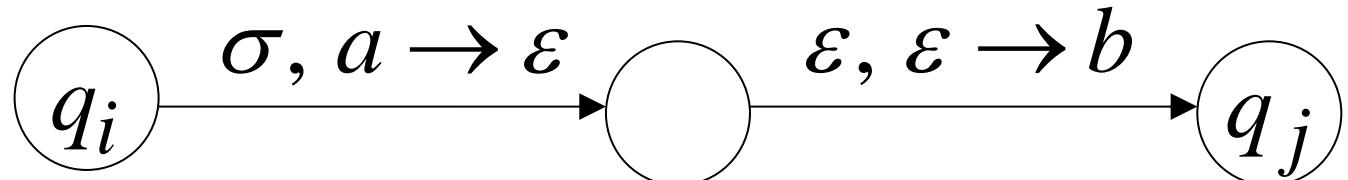


4. Each transition either pushes a symbol
or pops a symbol but not both together

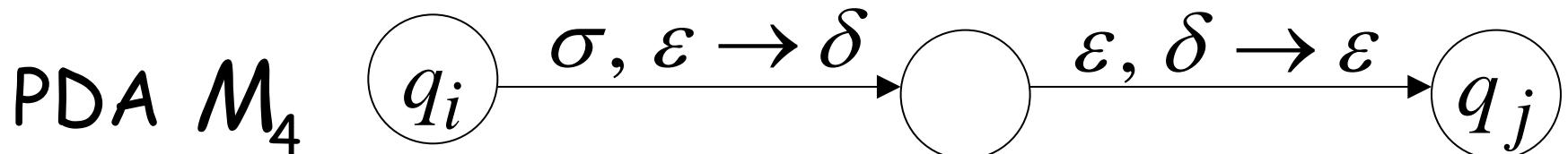
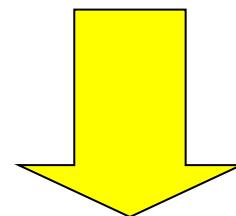
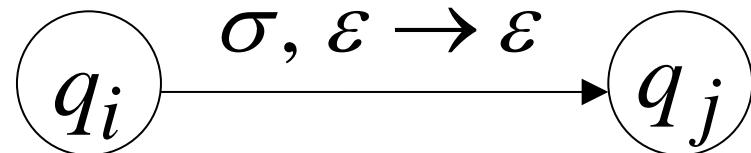
PDA M_3



PDA M_4



PDA M_3

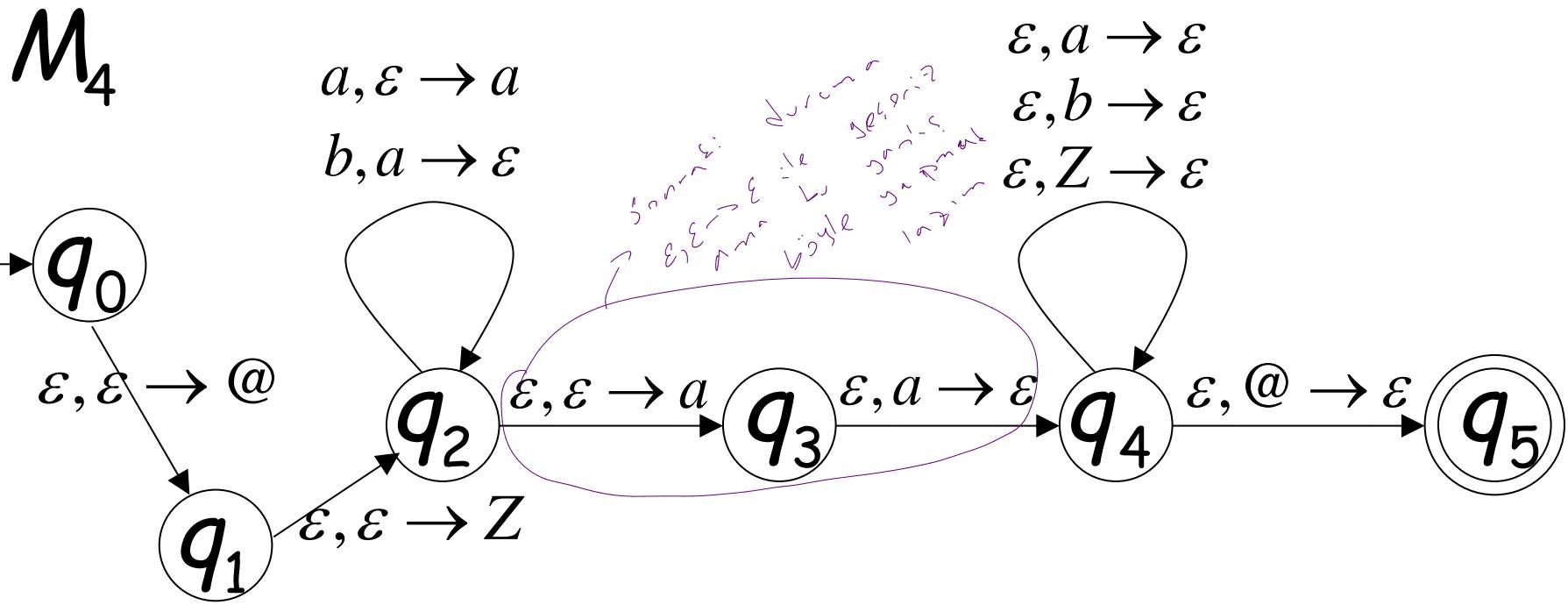
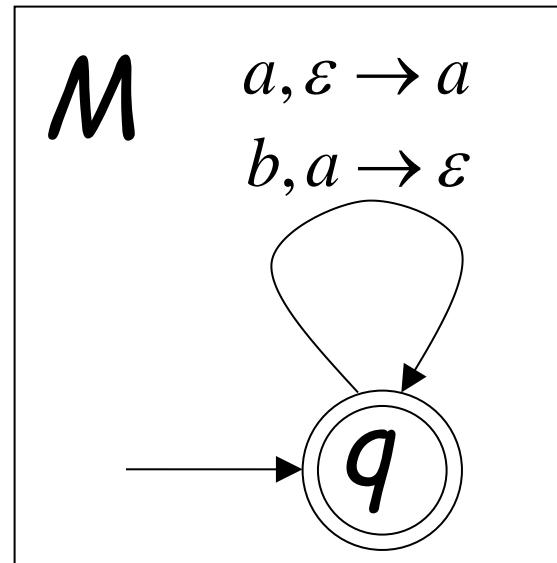


Where δ is a symbol of the stack alphabet

PDA M_4 is the final modified PDA

Note that the new initial stack symbol # is never used in any transition

Example:

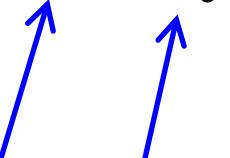


Grammar Construction

Variables:

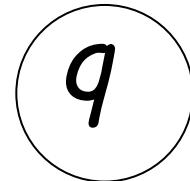
$$A_{q_i, q_j}$$

States of PDA



PDA

Kind 1: for each state

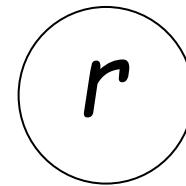
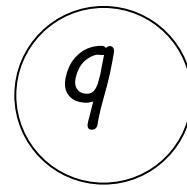
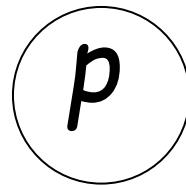


Grammar

$$A_{qq} \rightarrow \epsilon$$

PDA

Kind 2: for every three states

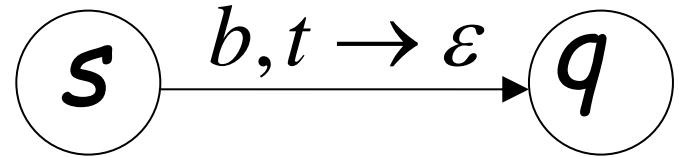
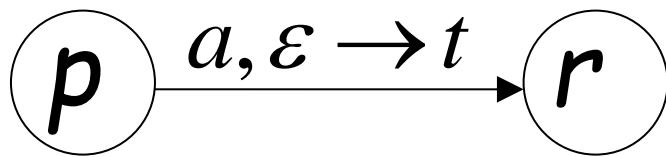


Grammar

$$A_{pq} \rightarrow A_{pr} A_{rq}$$

PDA

Kind 3: for every pair of such transitions

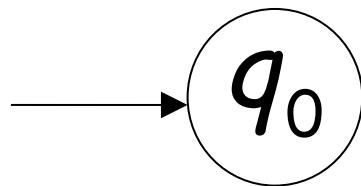


Grammar

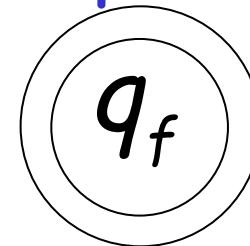
$$A_{pq} \rightarrow a A_{rs} b$$

PDA

Initial state



Accept state



Grammar

Start variable

$A_{q_0 q_f}$

Example:

PDA

M_4

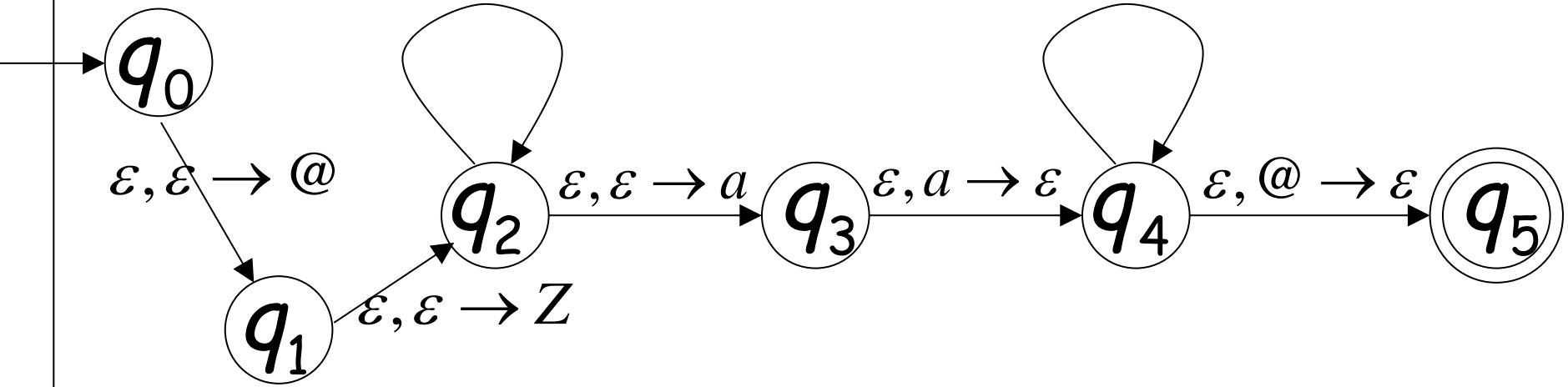
$a, \varepsilon \rightarrow a$

$b, a \rightarrow \varepsilon$

$\varepsilon, a \rightarrow \varepsilon$

$\varepsilon, b \rightarrow \varepsilon$

$\varepsilon, Z \rightarrow \varepsilon$



Grammar

Kind 1: from single states

$$A_{q_0q_0} \rightarrow \epsilon$$

$$A_{q_1q_1} \rightarrow \epsilon$$

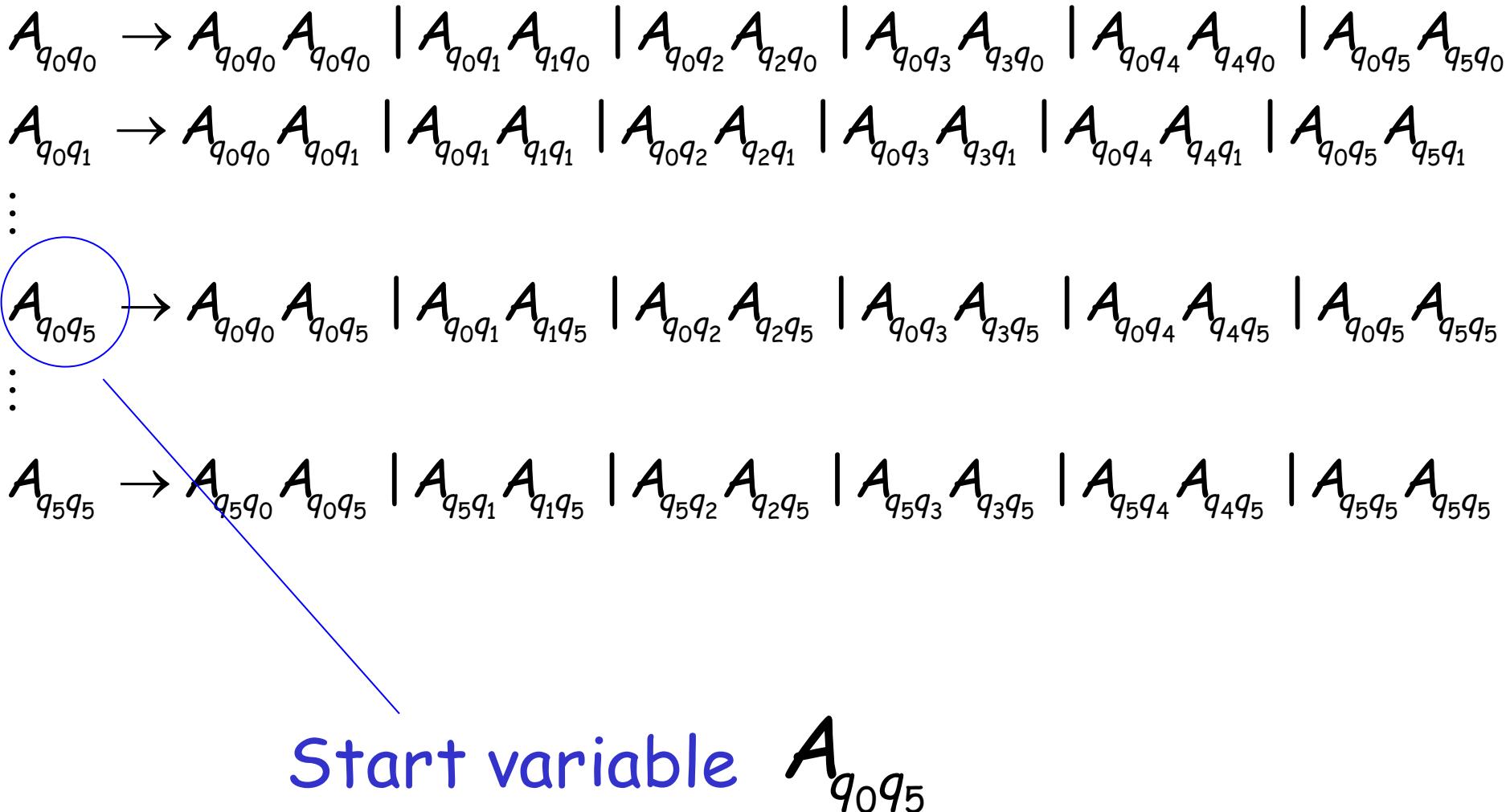
$$A_{q_2q_2} \rightarrow \epsilon$$

$$A_{q_3q_3} \rightarrow \epsilon$$

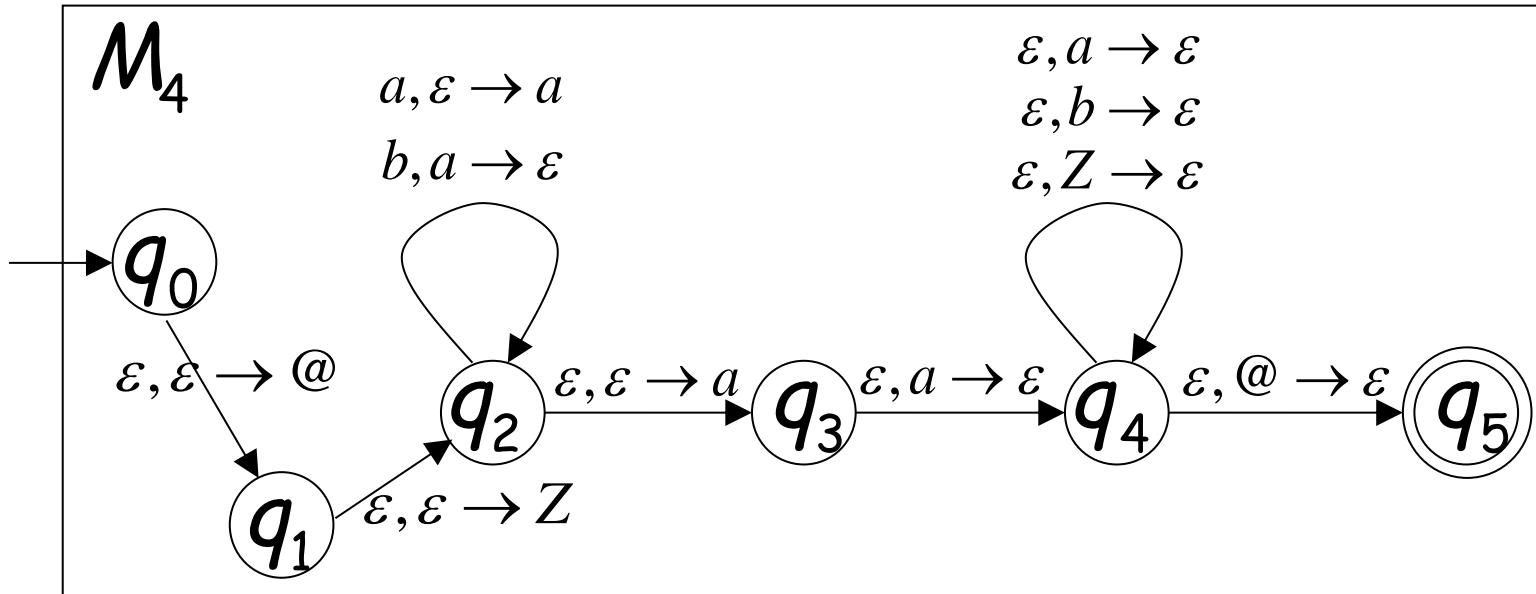
$$A_{q_4q_4} \rightarrow \epsilon$$

$$A_{q_5q_5} \rightarrow \epsilon$$

Kind 2: from triplets of states



Kind 3: from pairs of transitions



$$\begin{array}{lll}
 A_{q_0q_5} \rightarrow A_{q_1q_4} & A_{q_2q_4} \rightarrow aA_{q_2q_4} & A_{q_2q_2} \rightarrow A_{q_2q_2} b \\
 A_{q_1q_4} \rightarrow A_{q_2q_4} & A_{q_2q_2} \rightarrow aA_{q_2q_2} b & A_{q_2q_4} \rightarrow A_{q_3q_3} \\
 & A_{q_2q_4} \rightarrow aA_{q_2q_3} & A_{q_2q_4} \rightarrow A_{q_3q_4}
 \end{array}$$

Suppose that a PDA M is converted
to a context-free grammar G

We need to prove that $L(G) = L(M)$

or equivalently

$$L(G) \subseteq L(M)$$

$$L(G) \supseteq L(M)$$

$$L(G) \subseteq L(M)$$

We need to show that if G has derivation:

$$A_{q_0 q_f} \xrightarrow{*} w \quad (\text{string of terminals})$$

Then there is an accepting computation in M :

$$(q_0, w, \#) \xrightarrow{*} (q_f, \varepsilon, \#)$$

with input string w

We will actually show that if G has derivation:

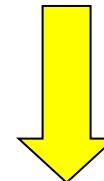
$$A_{pq} \xrightarrow{*} W$$

Then there is a computation in M :

$$(p, w, \varepsilon) \xsucc{*} (q, \varepsilon, \varepsilon)$$

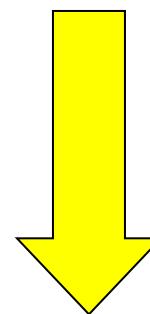
Therefore:

$$A_{q_0 q_f} \xrightarrow{*} w$$



$$(q_0, w, \varepsilon) \succ (q_f, \varepsilon, \varepsilon)$$

Since there is no transition
with the # symbol



$$(q_0, w, \#) \succ (q_f, \varepsilon, \#)$$

Lemma:

If $A_{pq} \xrightarrow{*} w$ (string of terminals)

then there is a computation
from state p to state q on string w
which leaves the stack empty:

$$(p, w, \epsilon) \xrightarrow{*} (q, \epsilon, \epsilon)$$

Proof Intuition:

$$A_{pq} \Rightarrow \cdots \Rightarrow W$$

Type 2

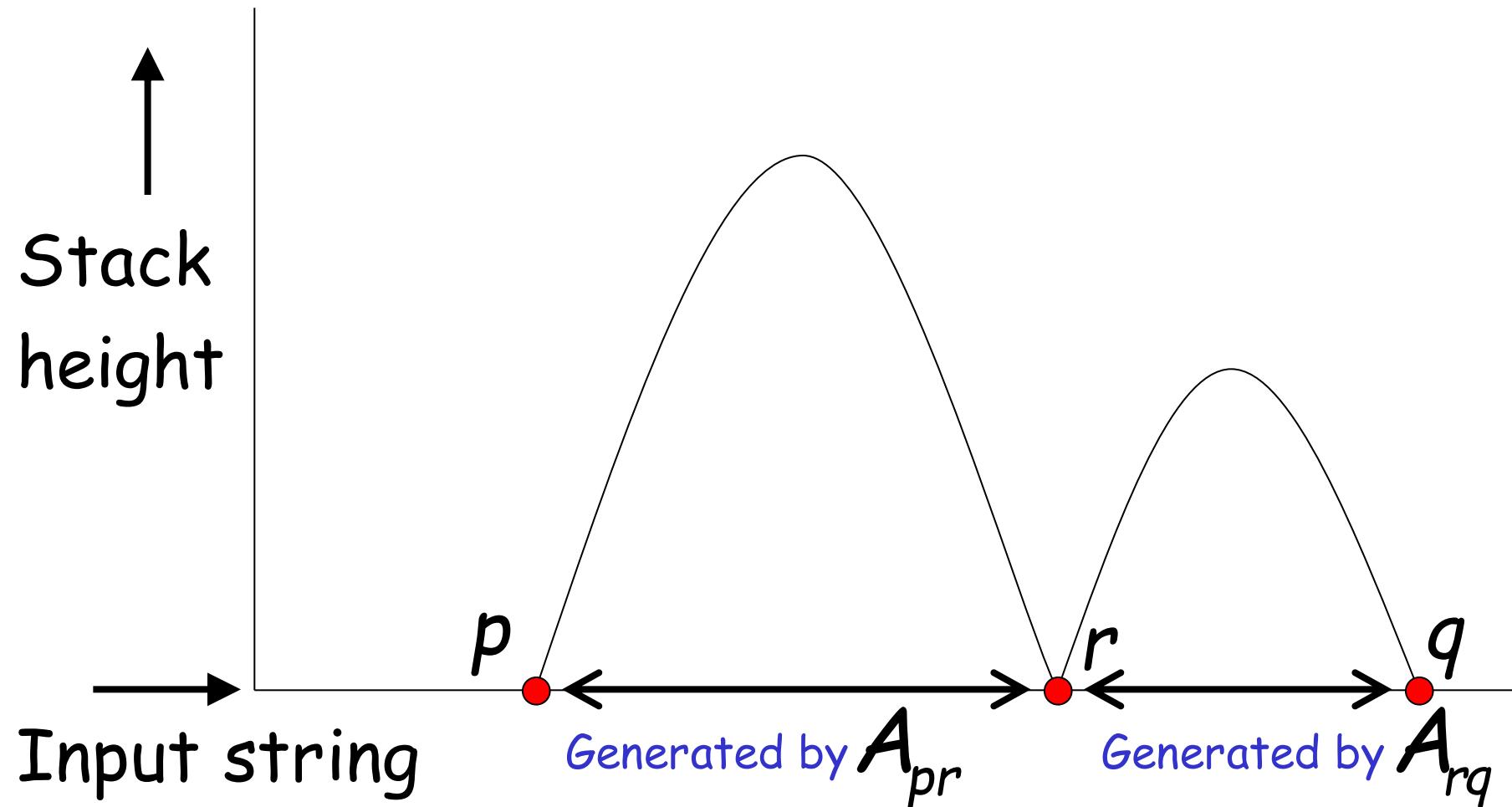
Case 1: $A_{pq} \Rightarrow A_{pr} A_{rq} \Rightarrow \cdots \Rightarrow W$

Type 3

Case 2: $A_{pq} \Rightarrow a A_{rs} b \Rightarrow \cdots \Rightarrow W$

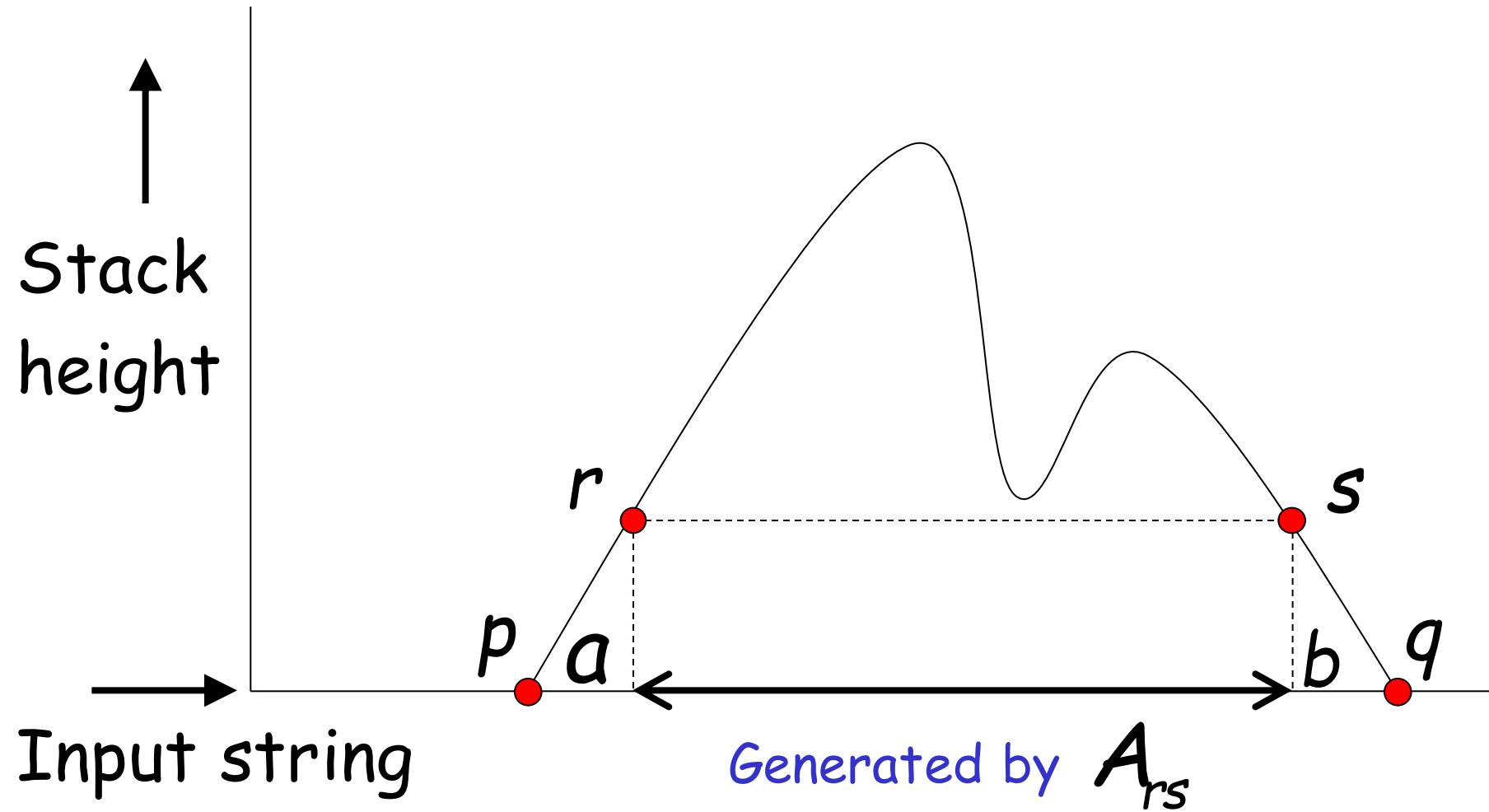
Type 2

Case 1: $A_{pq} \Rightarrow A_{pr} A_{rq} \Rightarrow \dots \Rightarrow w$



Type 3

Case 2: $A_{pq} \Rightarrow aA_{rs}b \Rightarrow \dots \Rightarrow W$



Formal Proof:

We formally prove this claim
by induction on the number
of steps in derivation:

$$A_{pq} \Rightarrow \cdots \Rightarrow W$$

number of steps

Induction Basis: $A_{pq} \Rightarrow W$
(one derivation step)

A Kind 1 production must have been used:

$$A_{pp} \rightarrow \epsilon$$

Therefore, $p = q$ and $w = \epsilon$

This computation of PDA trivially exists:

$$(p, \epsilon, \epsilon) \xrightarrow{*} (p, \epsilon, \epsilon)$$

Induction Hypothesis:

$$A_{pq} \Rightarrow \cdots \Rightarrow W$$

k derivation steps

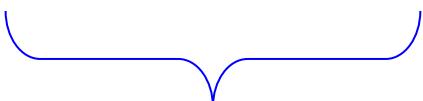
suppose it holds:

$$(p, w, \varepsilon) \overset{*}{\succ} (q, \varepsilon, \varepsilon)$$

Induction Step:

$$A_{pq} \Rightarrow \cdots \Rightarrow W$$

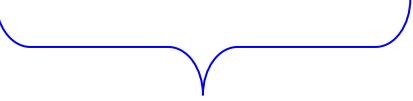
$k + 1$ derivation steps



We have to show:

$$(p, w, \varepsilon) \xrightarrow{*} (q, \varepsilon, \varepsilon)$$

$$A_{pq} \Rightarrow \cdots \Rightarrow W$$



$k + 1$ derivation steps

Type 2

Case 1: $A_{pq} \Rightarrow A_{pr} A_{rq} \Rightarrow \cdots \Rightarrow W$

Type 3

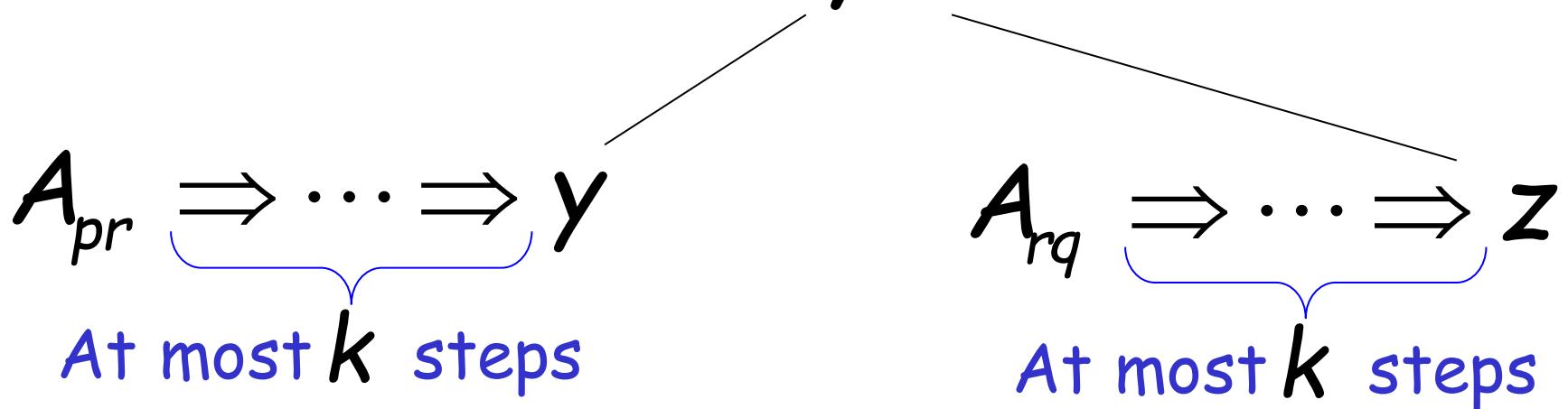
Case 2: $A_{pq} \Rightarrow a A_{rs} b \Rightarrow \cdots \Rightarrow W$

Type 2

Case 1: $A_{pq} \Rightarrow A_{pr} A_{rq} \Rightarrow \dots \Rightarrow w$

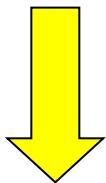
$k + 1$ steps

We can write $w = yz$



$$A_{pr} \Rightarrow \cdots \Rightarrow y$$

At most k steps

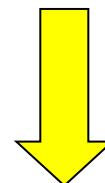


From induction
hypothesis, in PDA:

$$(p, y, \varepsilon) \xrightarrow{*} (r, \varepsilon, \varepsilon)$$

$$A_{rq} \Rightarrow \cdots \Rightarrow z$$

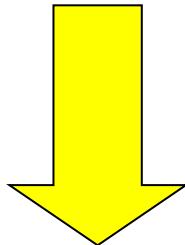
At most k steps



From induction
hypothesis, in PDA:

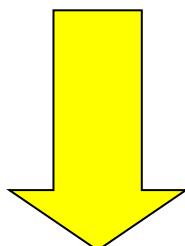
$$(r, z, \varepsilon) \xrightarrow{*} (q, \varepsilon, \varepsilon)$$

$$(p, y, \varepsilon) \succ^* (r, \varepsilon, \varepsilon) \qquad \qquad (r, z, \varepsilon) \succ^* (q, \varepsilon, \varepsilon)$$



$$(p, yz, \varepsilon) \succ^* (r, z, \varepsilon) \succ^* (q, \varepsilon, \varepsilon)$$

since $w = yz$



$$(p, w, \varepsilon) \succ^* (q, \varepsilon, \varepsilon)$$

Type 3

Case 2: $A_{pq} \Rightarrow aA_{rs}b \Rightarrow \dots \Rightarrow w$

$k + 1$ steps

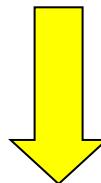
We can write $w = ayb$

$A_{rs} \Rightarrow \dots \Rightarrow y$

At most k steps

$A_{rs} \Rightarrow \dots \Rightarrow y$

At most k steps

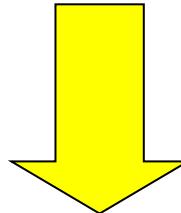


From induction hypothesis,
the PDA has computation:

$$(r, y, \varepsilon) \xrightarrow{*} (s, \varepsilon, \varepsilon)$$

Type 3

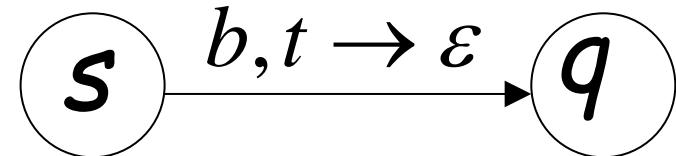
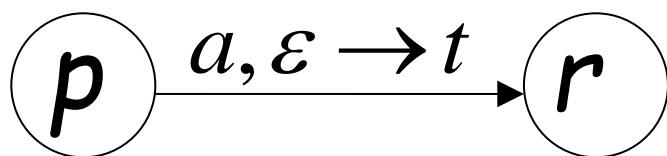
$$A_{pq} \Rightarrow a A_{rs} b \Rightarrow \dots \Rightarrow w$$

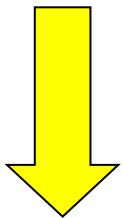
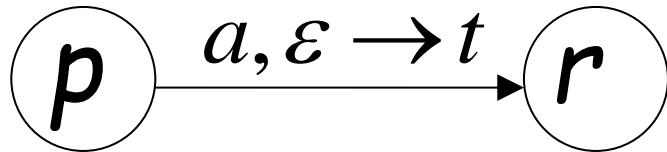


Grammar contains production

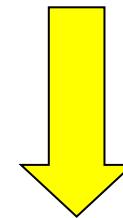
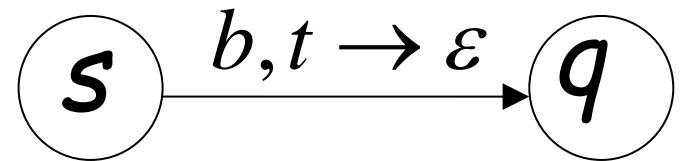
$$A_{pq} \rightarrow a A_{rs} b$$

And PDA Contains transitions





$$(p, ayb, \varepsilon) \succ (r, yb, t)$$



$$(s, b, t) \succ (q, \varepsilon, \varepsilon)$$

We know

$$(r, y, \varepsilon) \xsucc{*} (s, \varepsilon, \varepsilon) \quad \longrightarrow \quad (r, yb, t) \xsucc{*} (s, b, t)$$

$$(p, ayb, \varepsilon) \succ (r, yb, t)$$

We also know

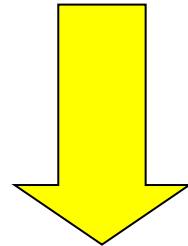
$$(s, b, t) \succ (q, \varepsilon, \varepsilon)$$

Therefore:

$$(p, ayb, \varepsilon) \succ (r, yb, t) \xsucc{*} (s, b, t) \succ (q, \varepsilon, \varepsilon)$$

$$(p, ayb, \varepsilon) \succ (r, yb, t) \xsucc{*} (s, b, t) \succ (q, \varepsilon, \varepsilon)$$

since $w = ayb$



$$(p, w, \varepsilon) \xsucc{*} (q, \varepsilon, \varepsilon)$$

END OF PROOF

So far we have shown:

$$L(G) \subseteq L(M)$$

With a similar proof we can show

$$L(G) \supseteq L(M)$$

Therefore: $L(G) = L(M)$



EDUCO

The art of cooperation

Spring 2016

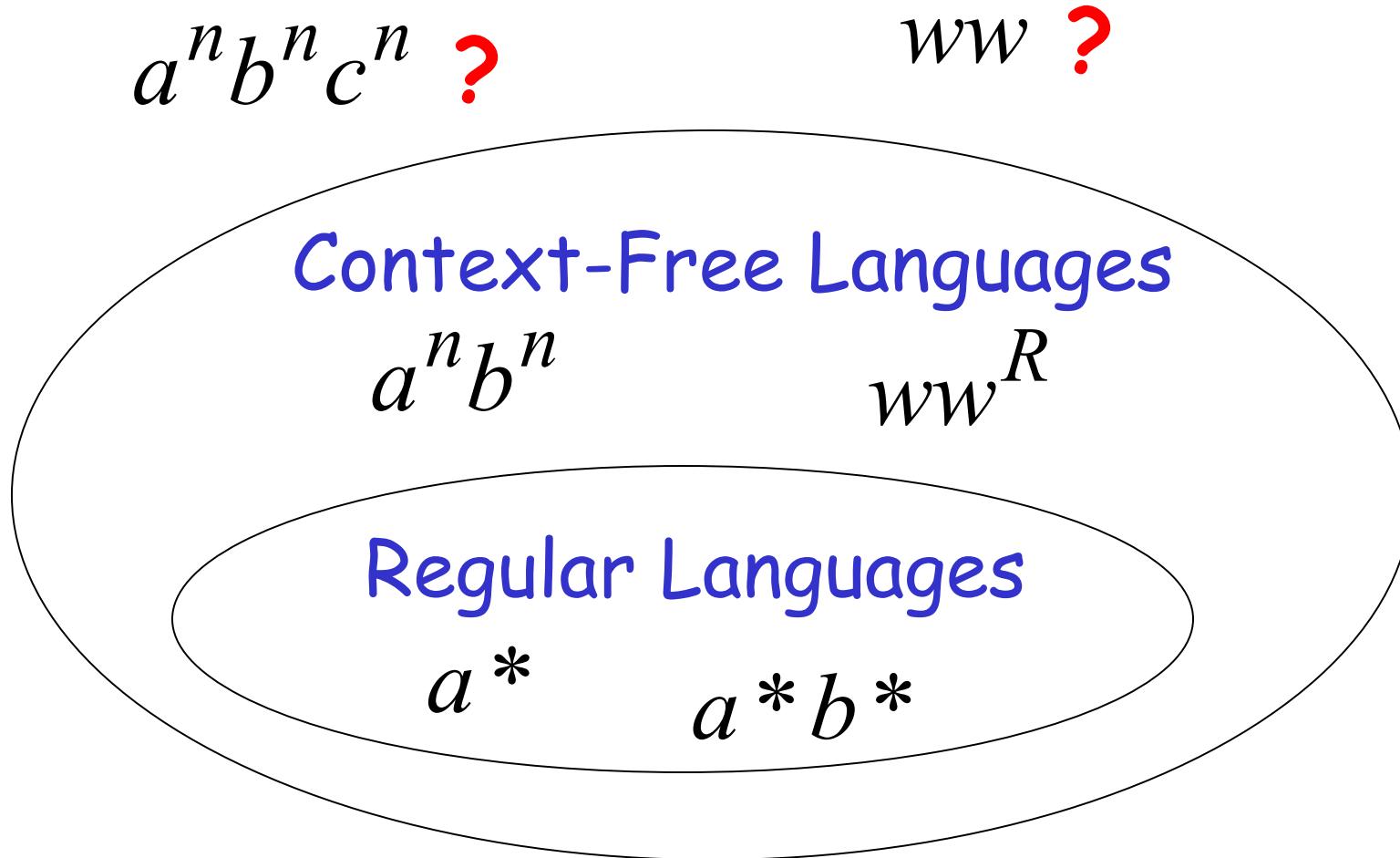
BLM2502 Theory of Computation

- » Course Outline
- » Week Content
- » 1 Introduction to Course
- » 2 Computability Theory, Complexity Theory, Automata Theory, Set Theory, Relations, Proofs, Pigeonhole Principle
- » 3 Regular Expressions
- » 4 Finite Automata
- » 5 Deterministic and Nondeterministic Finite Automata
- » 6 Epsilon Transition, Equivalence of Automata
- » 7 Pumping Theorem
- » 8 April 10 - 14 week is the first midterm week
- » 9 Context Free Grammars, Parse Tree, Ambiguity
- » 10 Pumping Theorem, Normal Forms
- » 11 Pushdown Automata
- » **12 Turing Machines, Recognition and Computation, Church-Turing Hypothesis**
- » 13 Turing Machines, Recognition and Computation, Church-Turing Hypothesis
- » 14 May 22 – 27 week is the second midterm week
- » 15 Review
- » 16 Final Exam date will be announced



Course Materials

The Language Hierarchy



Languages accepted by Turing Machines

$a^n b^n c^n$

ww

Context-Free Languages

$a^n b^n$

ww^R

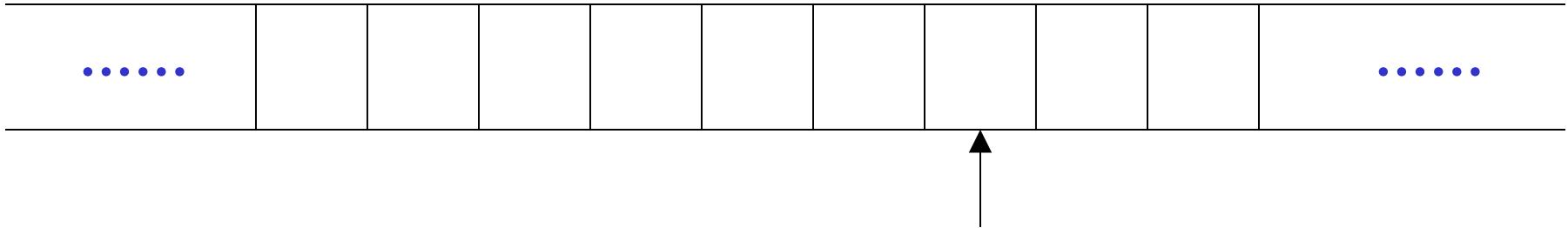
Regular Languages

a^*

$a^* b^*$

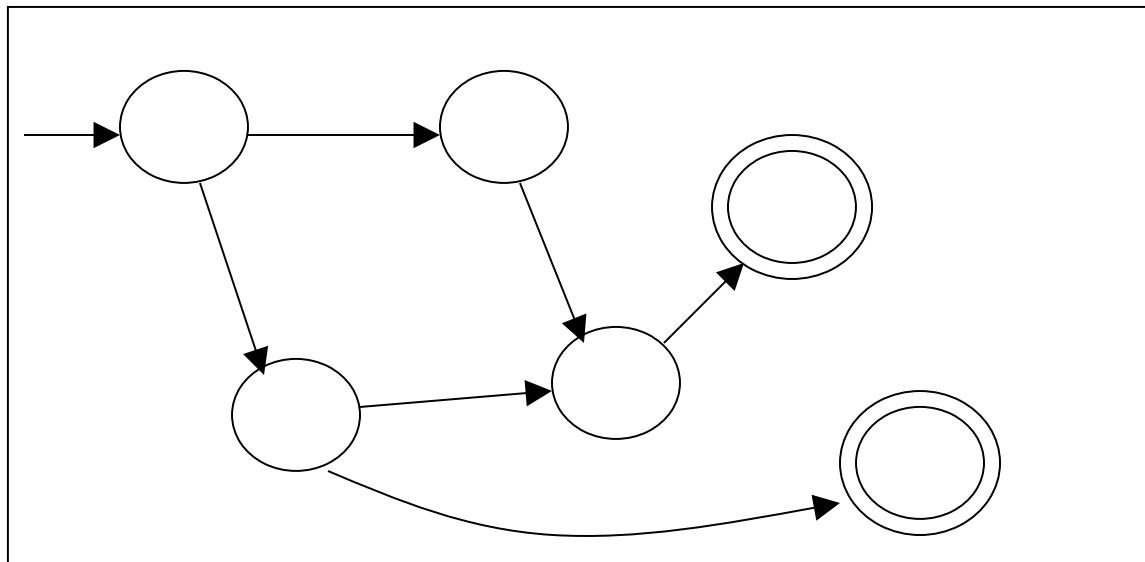
A Turing Machine

Tape



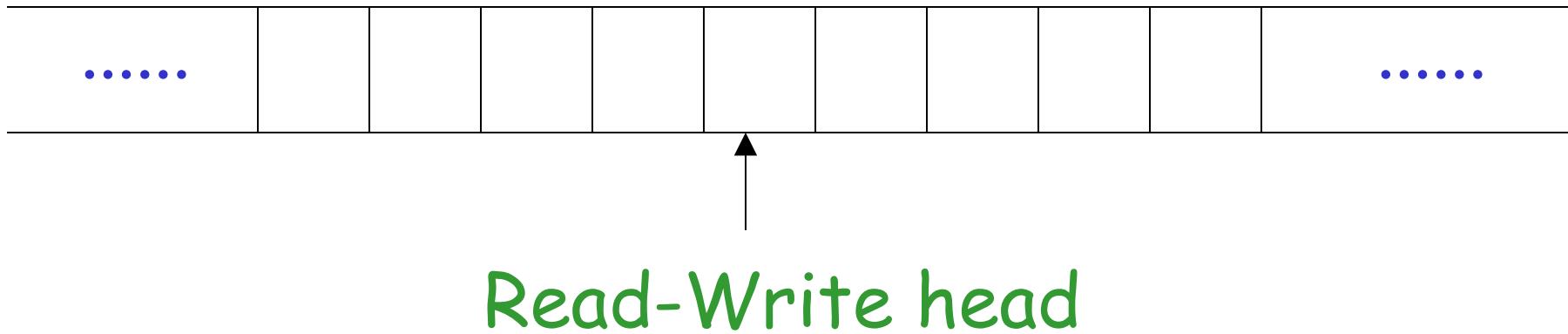
Read-Write head

Control Unit

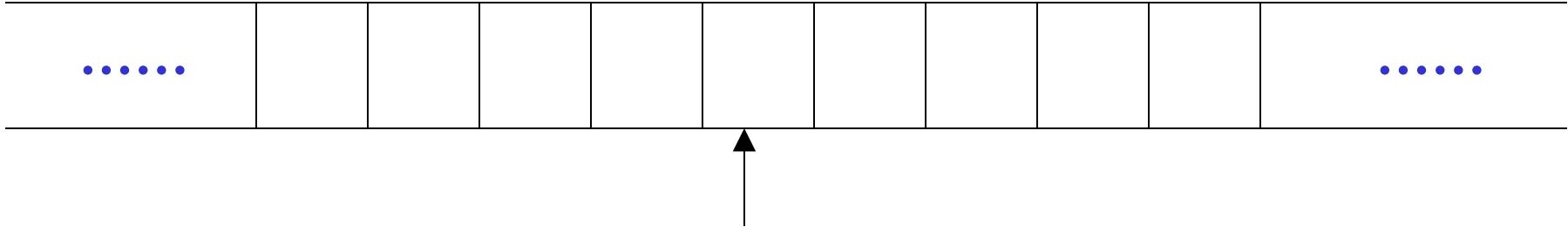


The Tape

No boundaries -- infinite length



The head moves Left or Right



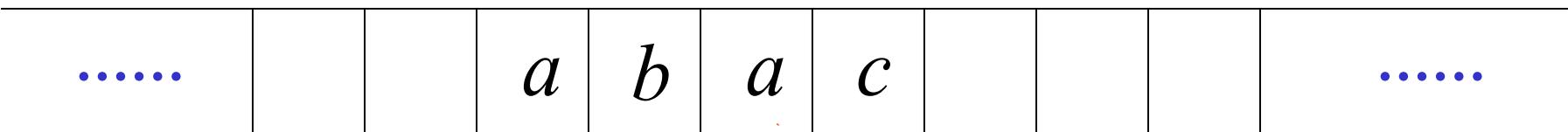
Read-Write head

The head at each transition (time step):

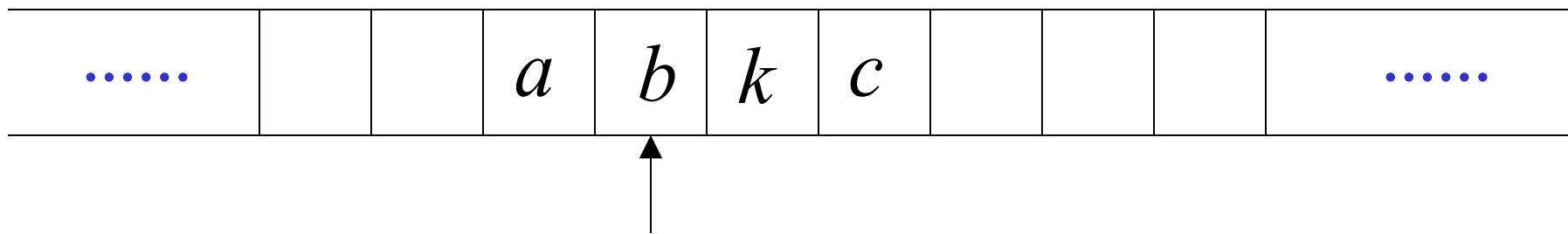
1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

Example:

Time 0



Time 1

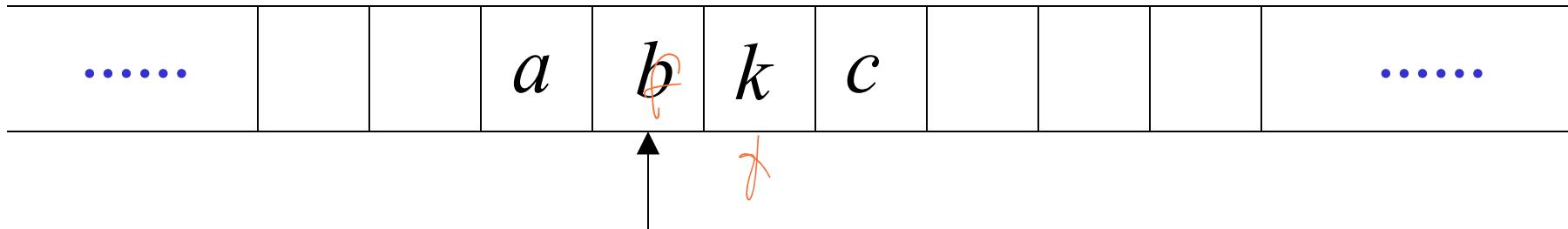


1. Reads a

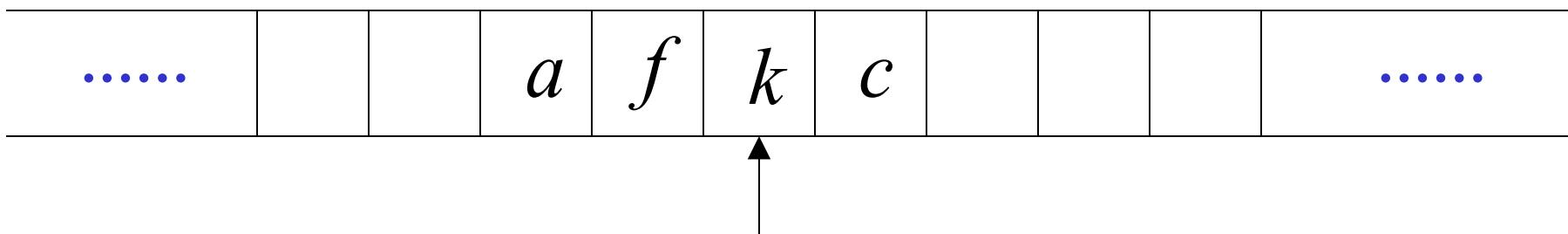
2. Writes k

3. Moves Left

Time 1

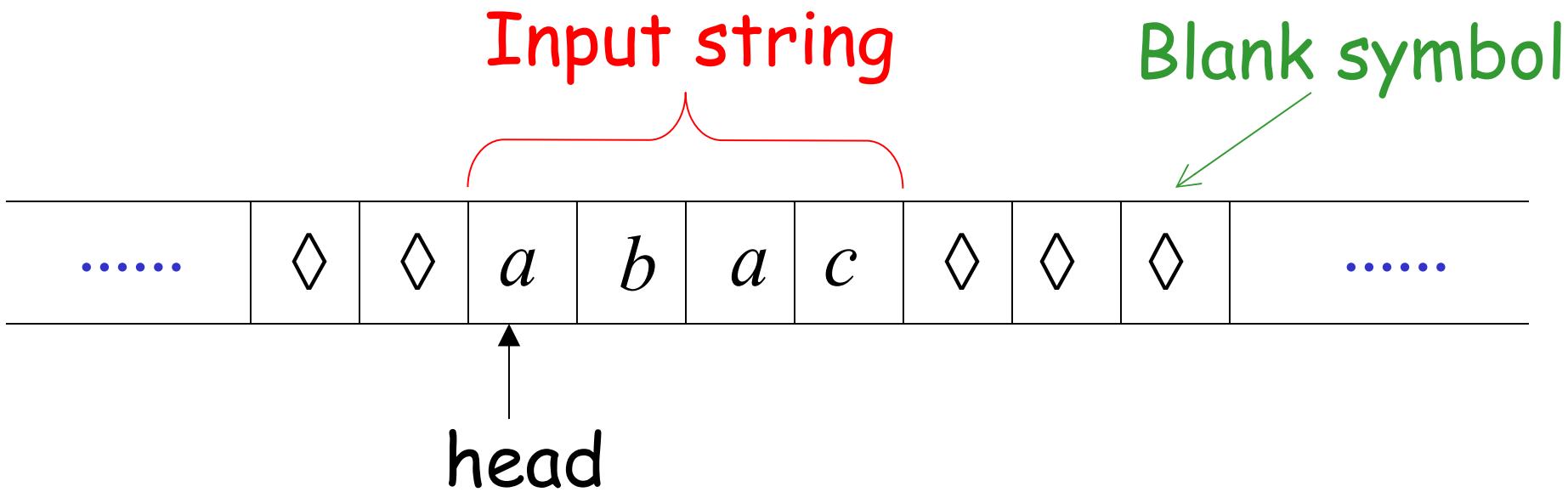


Time 2



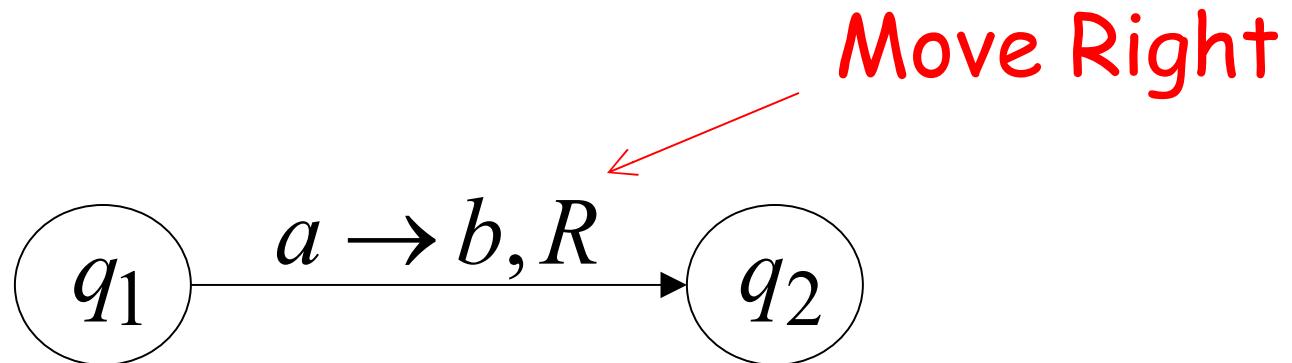
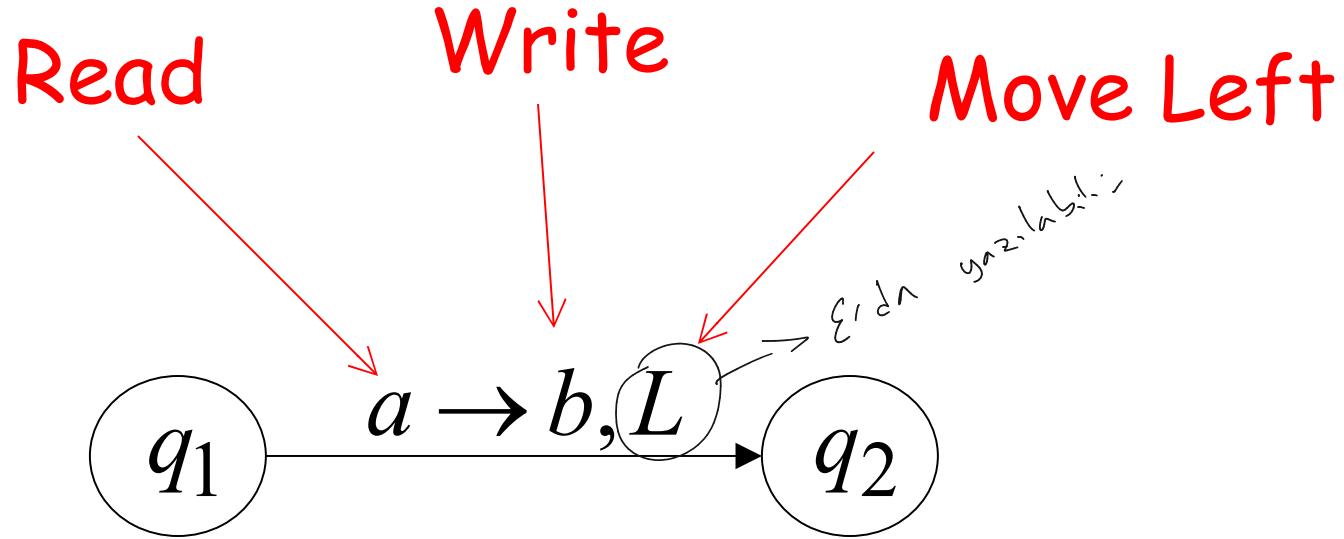
1. Reads b
2. Writes f
3. Moves Right

The Input String



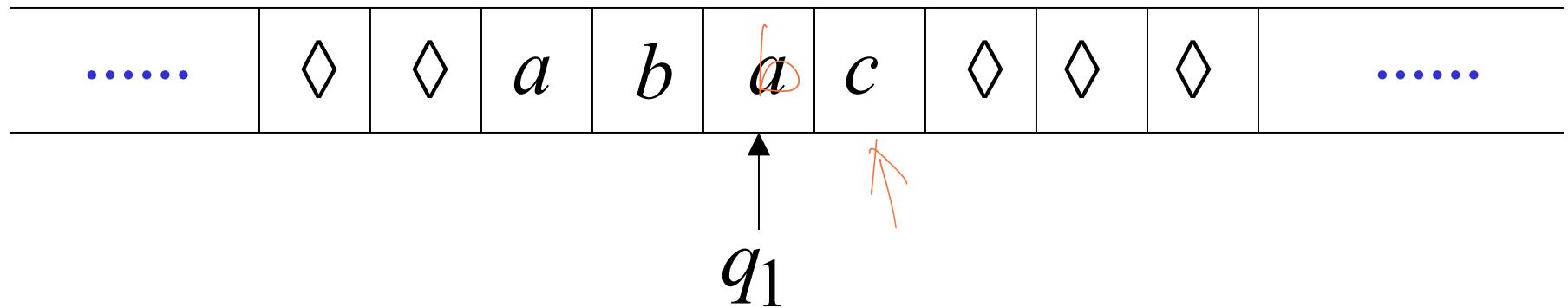
Head starts at the leftmost position
of the input string

States & Transitions

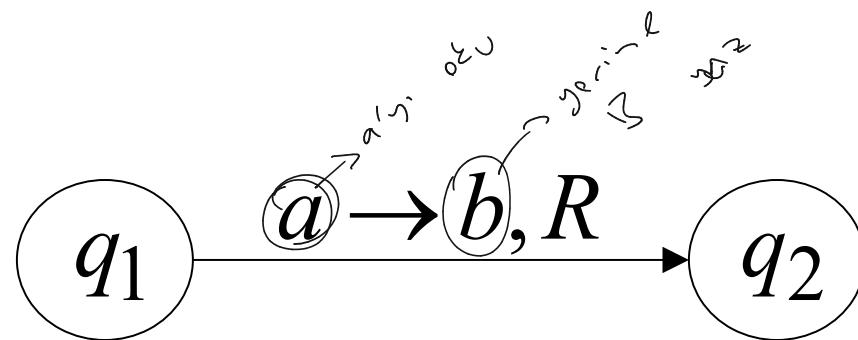


Example:

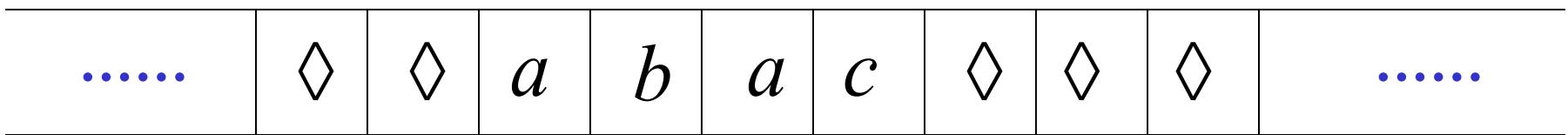
Time 1



current state

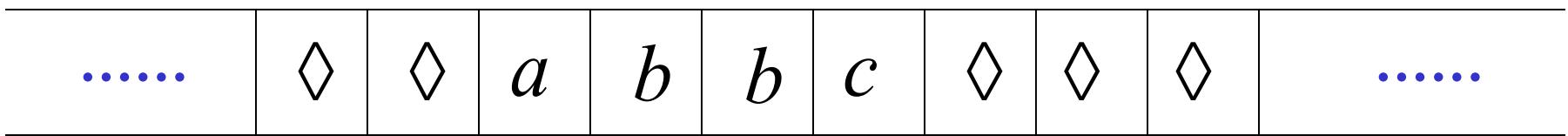


Time 1

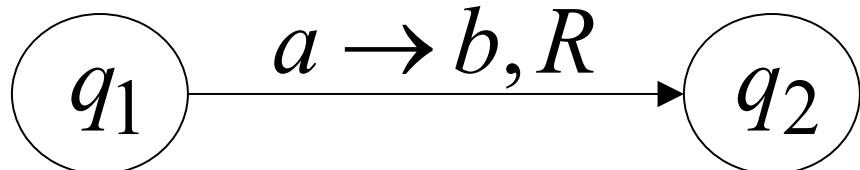


q_1

Time 2

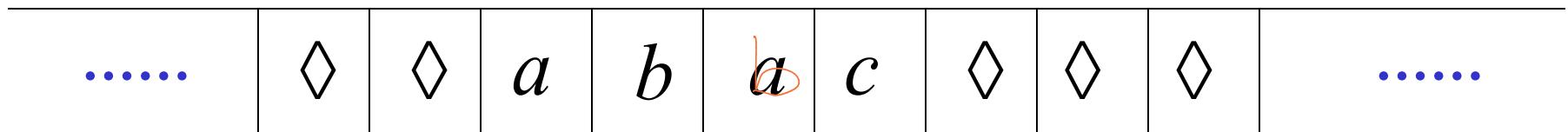


q_2



Example:

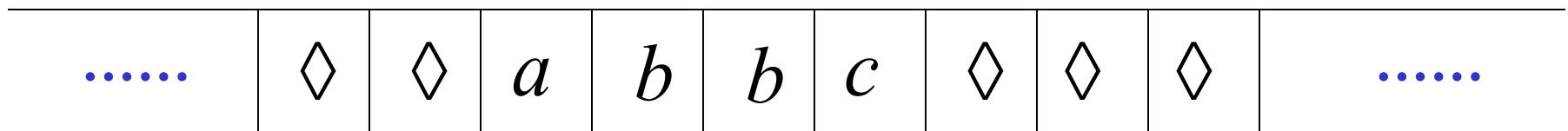
Time 1



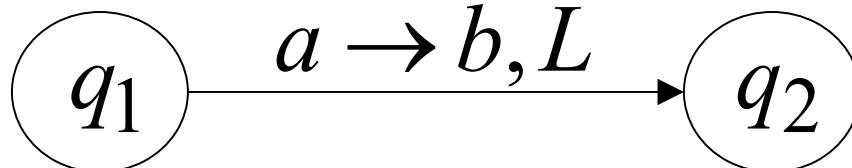
q_1

re) $e^{ck} = \text{HALT}$

Time 2

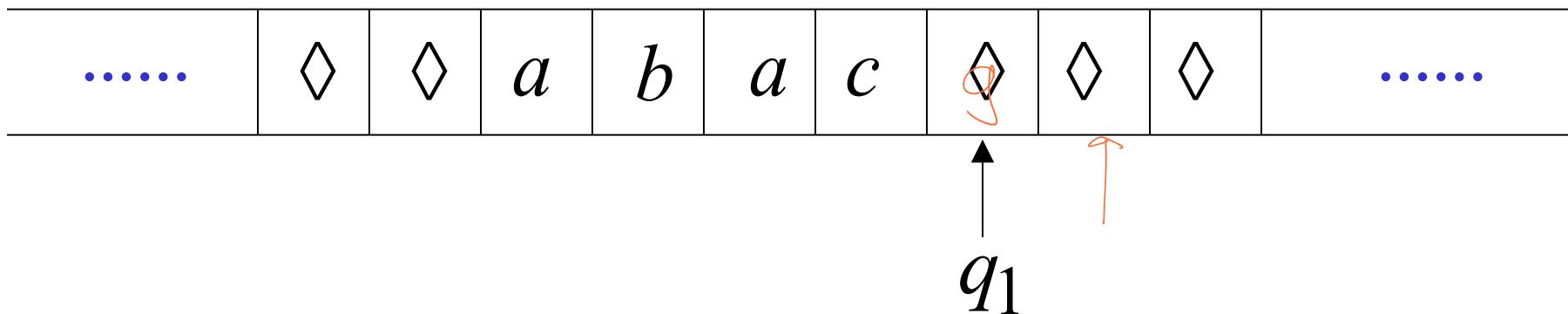


q_2

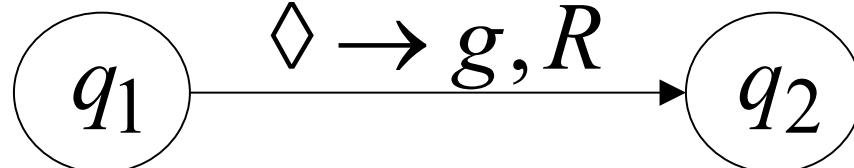
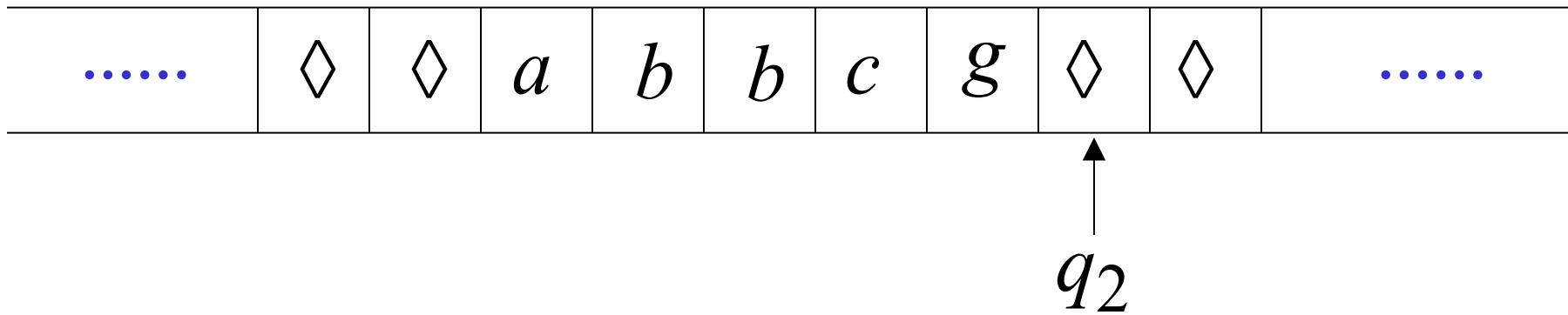


Example:

Time 1



Time 2

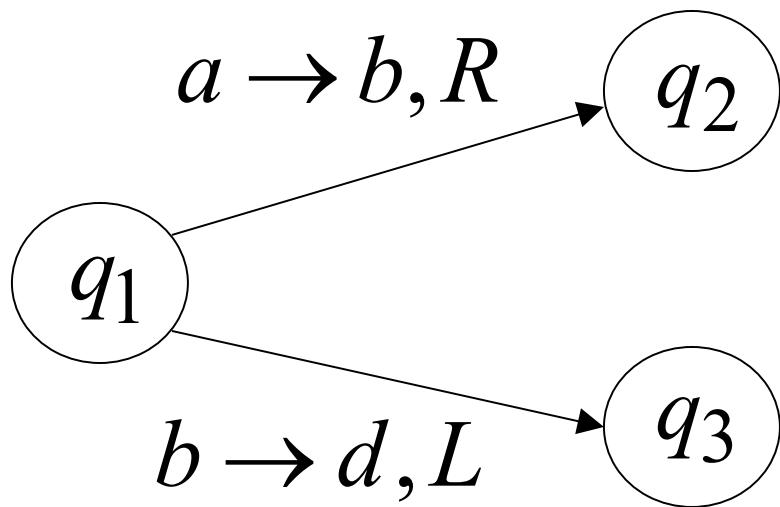


Determinism

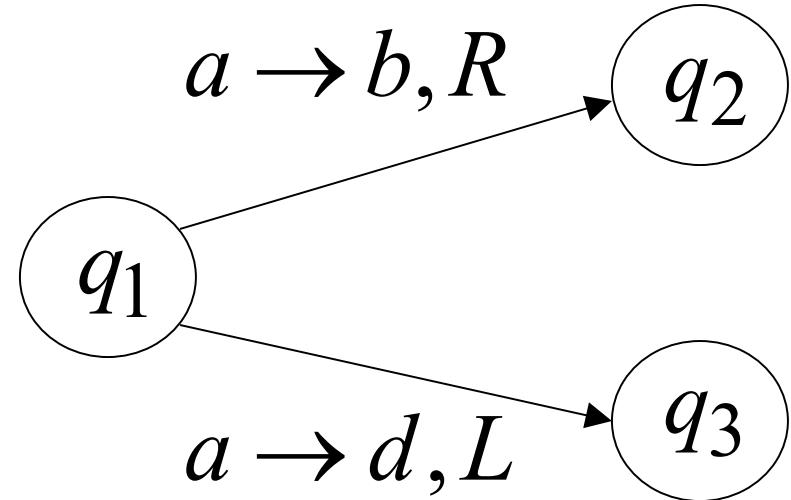
Turing Machines are deterministic

DFA
o.b.i.

Allowed



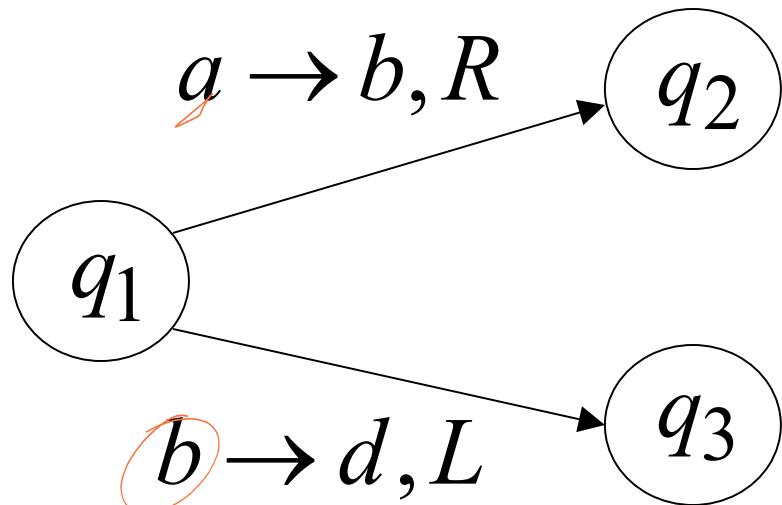
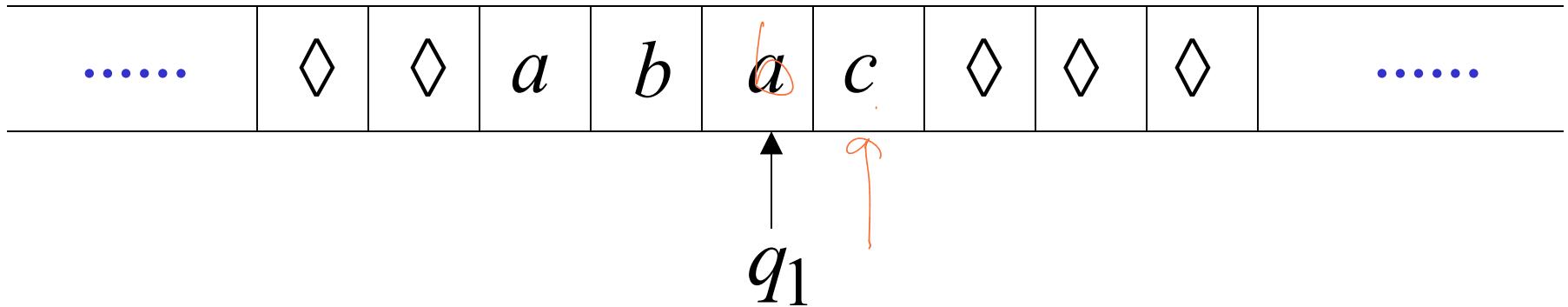
Not Allowed



No epsilon transitions allowed

Partial Transition Function

Example:



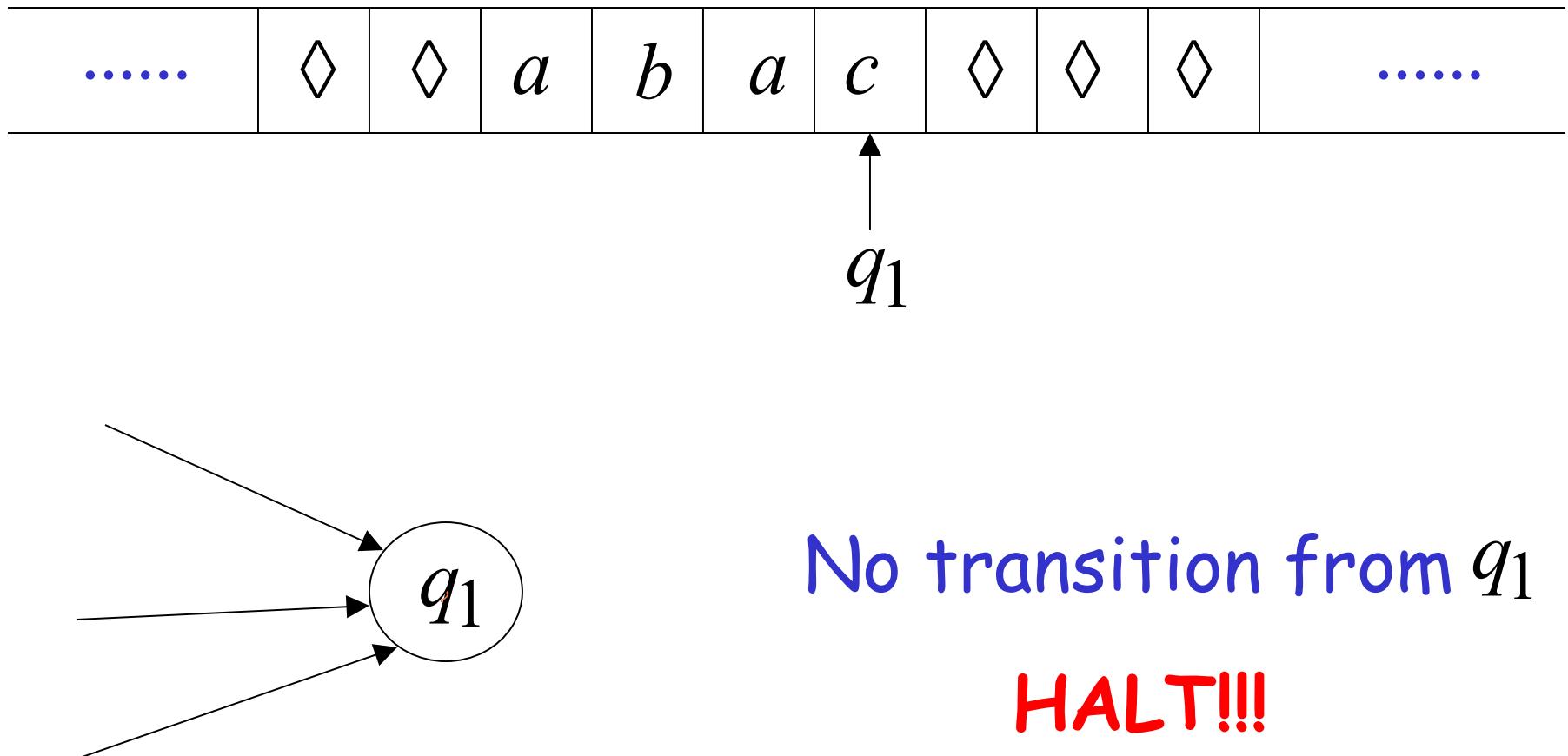
Allowed:

No transition
for input symbol *c*

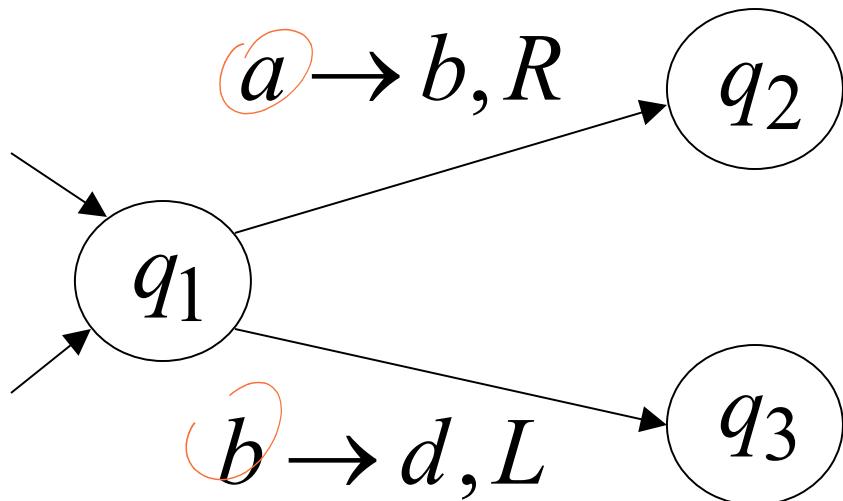
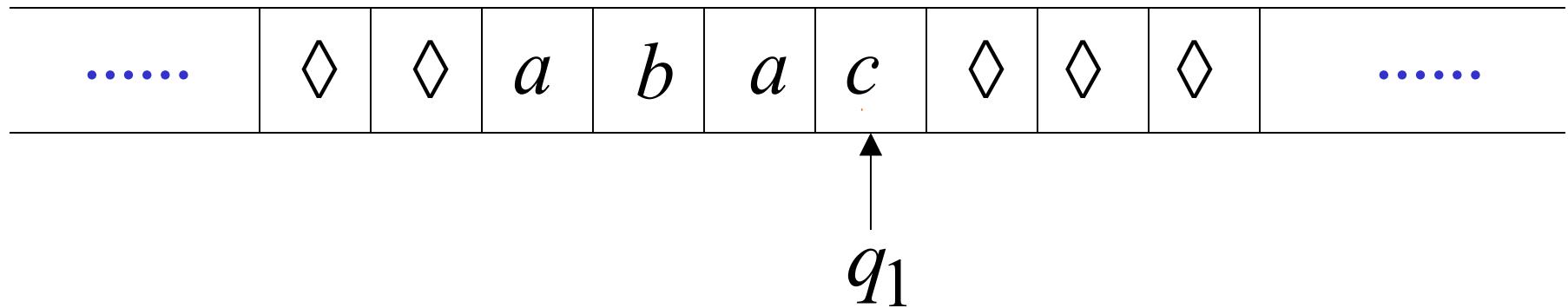
Halting

The machine *halts* in a state if there is no transition to follow

Halting Example 1:



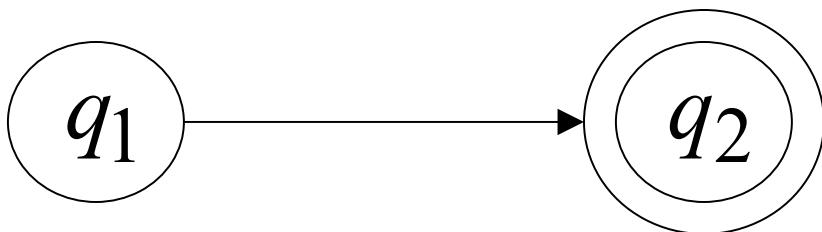
Halting Example 2:



No possible transition
from q_1 and symbol c

HALT!!!

Accepting States



Allowed



Not Allowed

- Accepting states have no outgoing transitions
- The machine halts and accepts

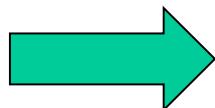
Acceptance

Accept Input string



If machine halts
in an accept state

Reject Input string



If machine halts
in a non-accept state

*→ compiler or
gibberish*

or

If machine enters
an infinite loop

Observation:

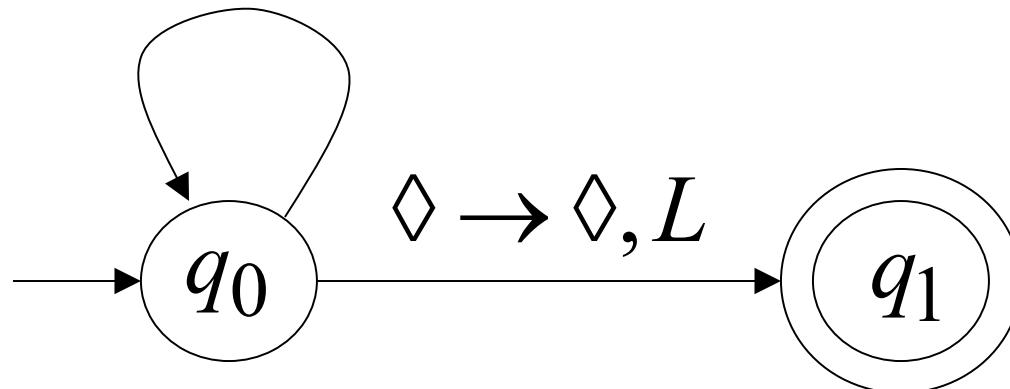
In order to accept an input string,
it is not necessary to scan all the
symbols in the string

Turing Machine Example

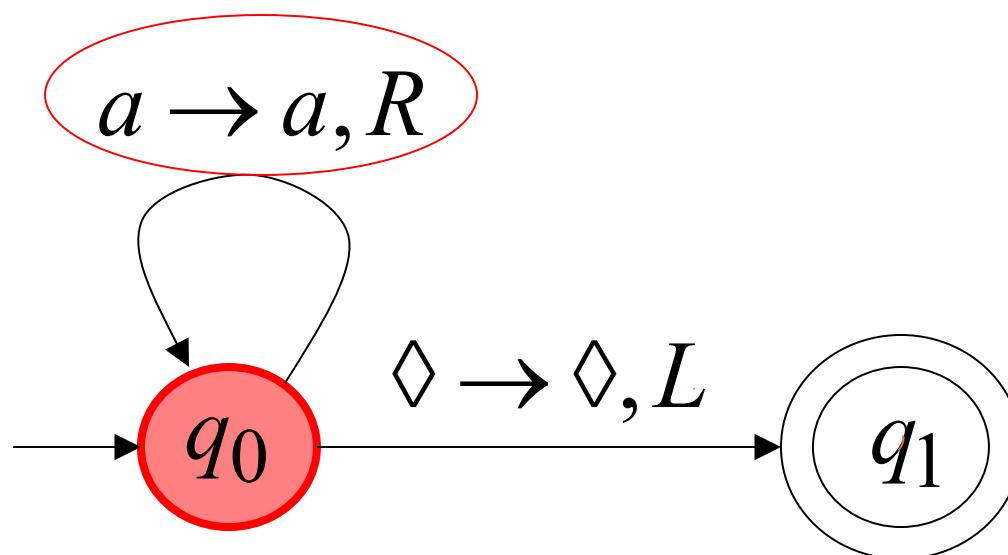
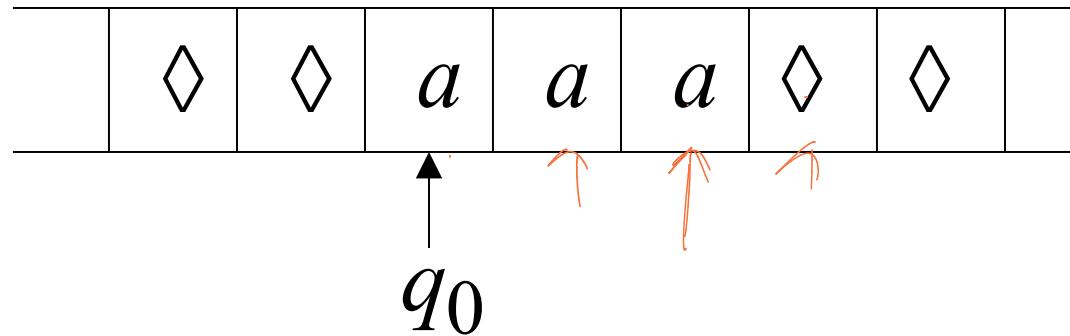
Input alphabet $\Sigma = \{a, b\}$

Accepts the language: a^*

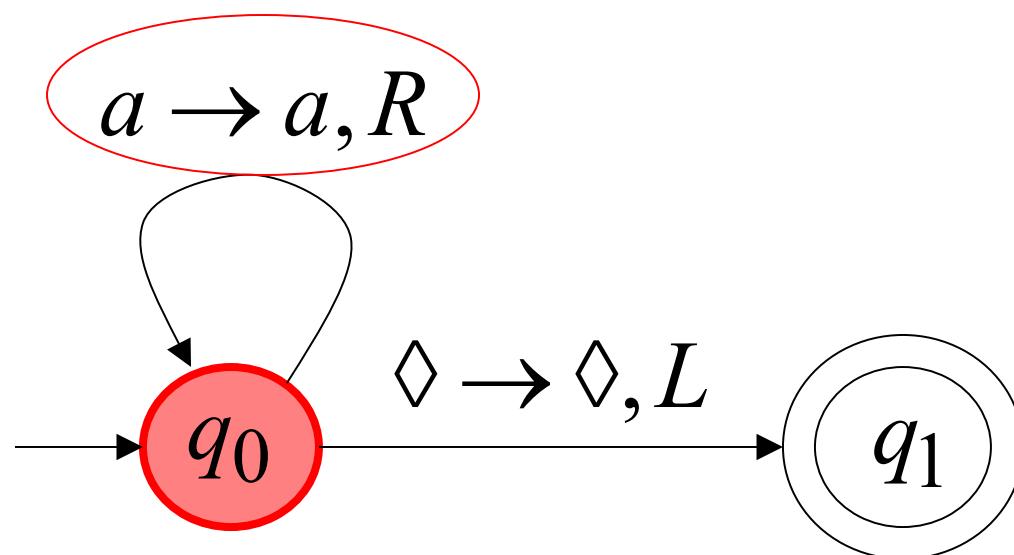
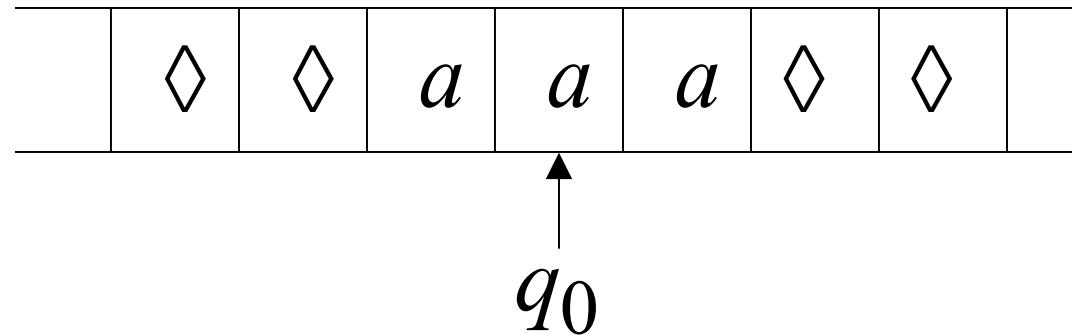
$$a \rightarrow a, R$$



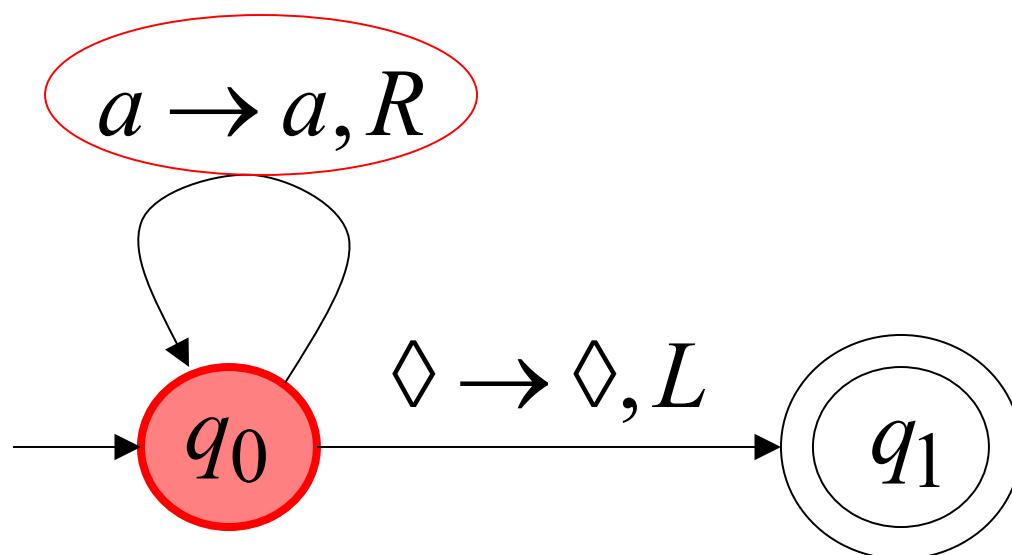
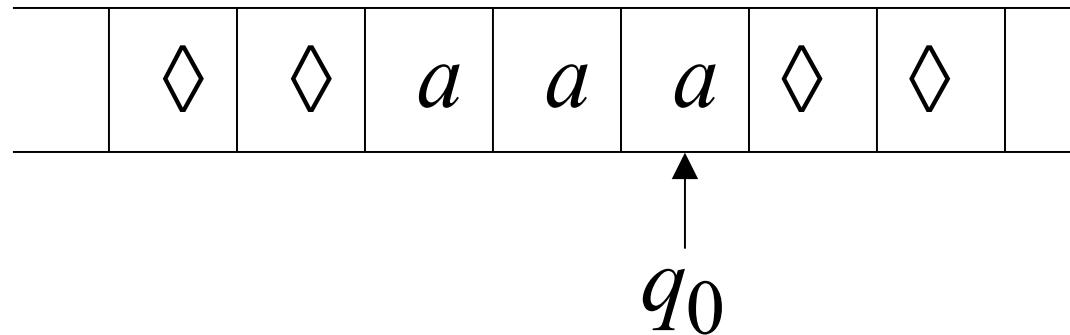
Time 0



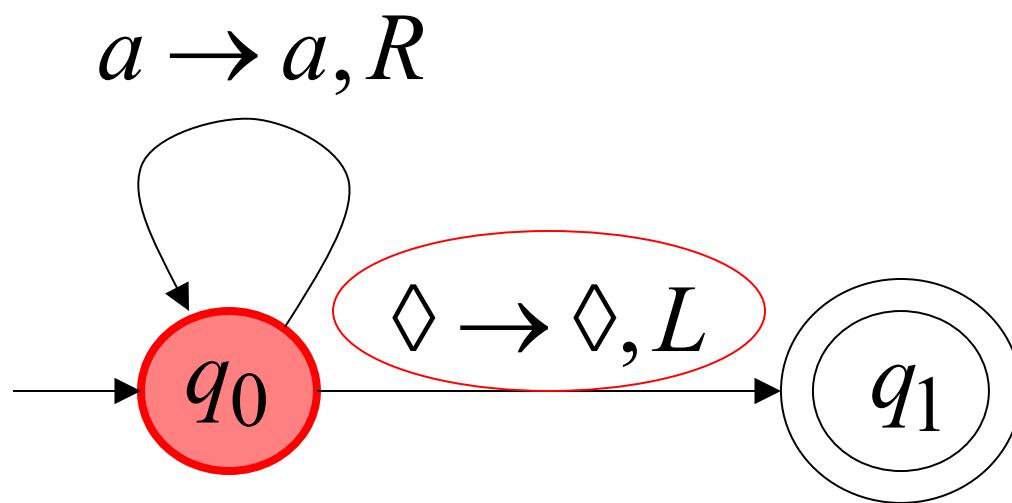
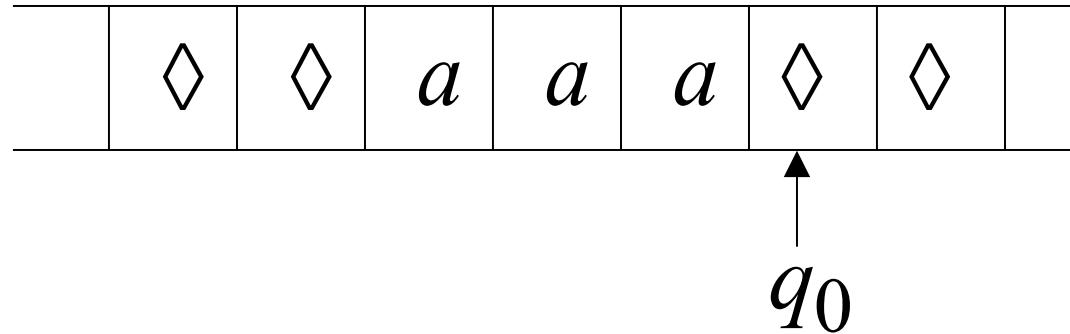
Time 1



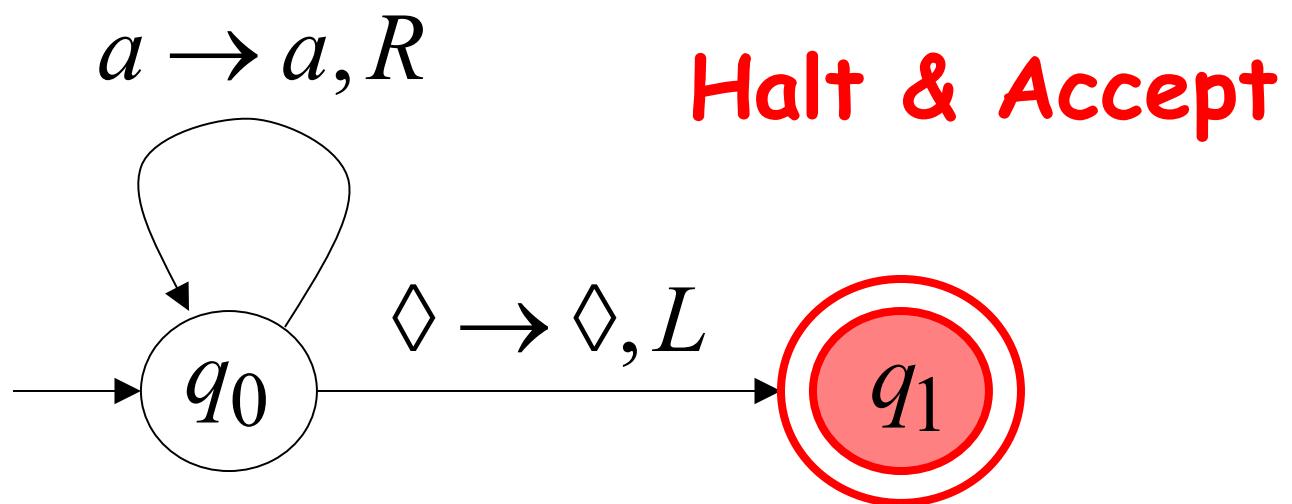
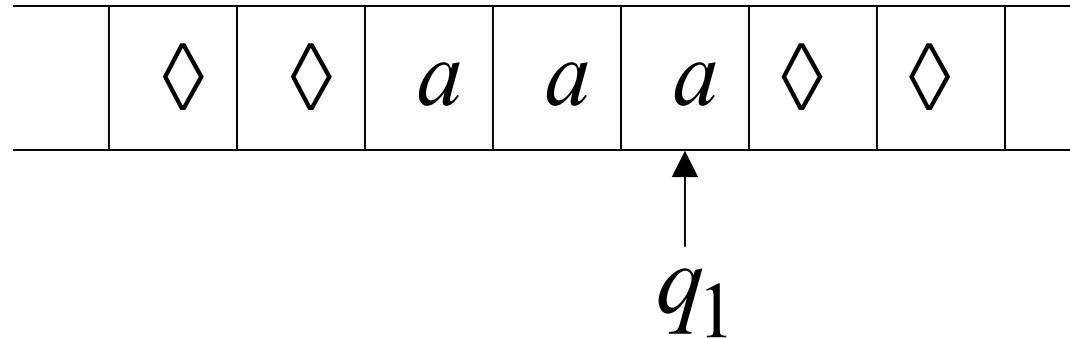
Time 2



Time 3

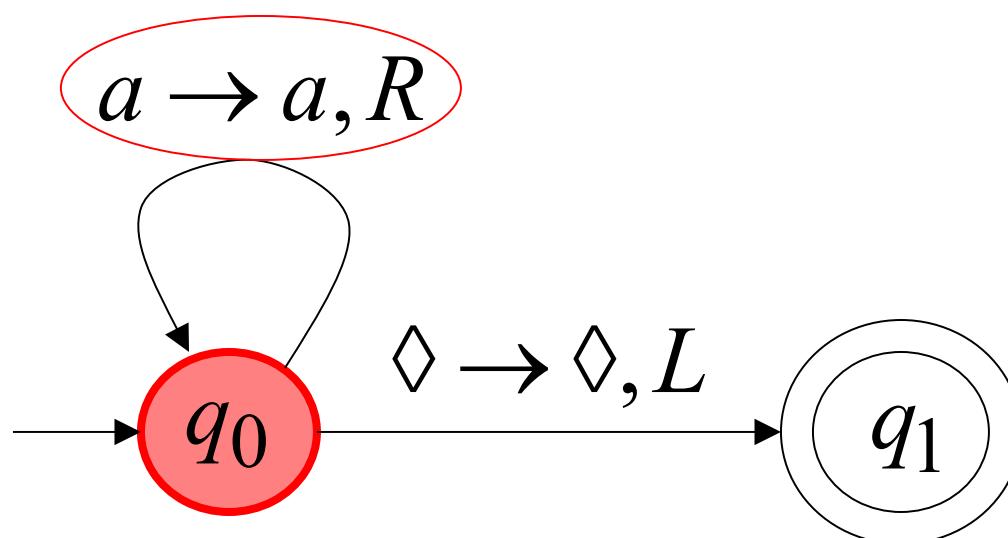
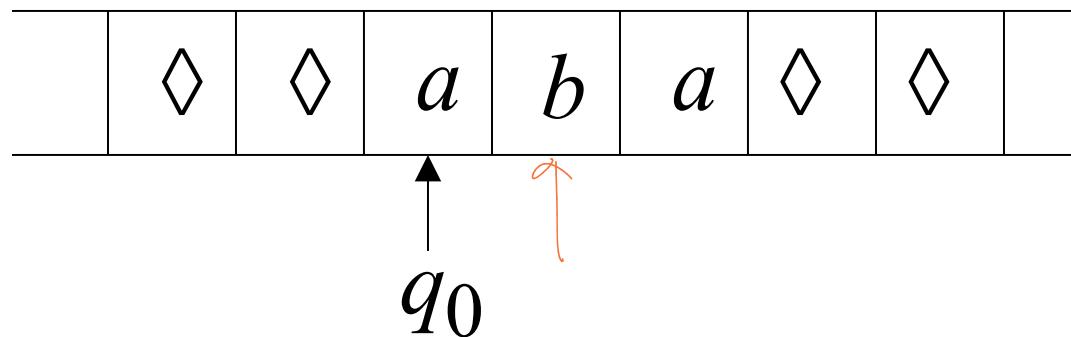


Time 4

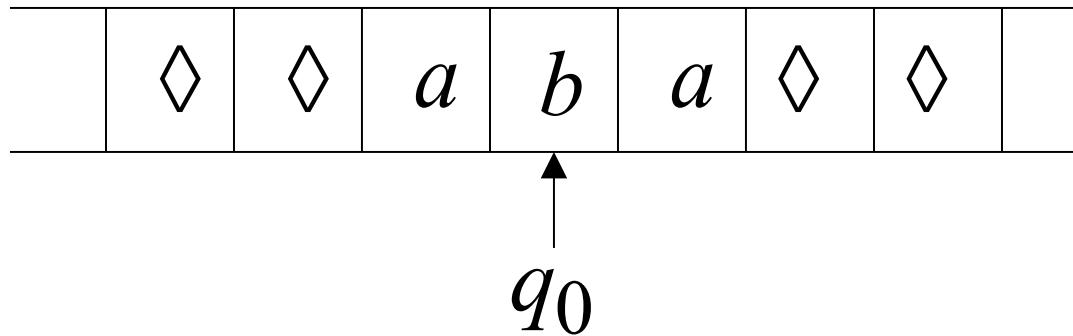


Rejection Example

Time 0



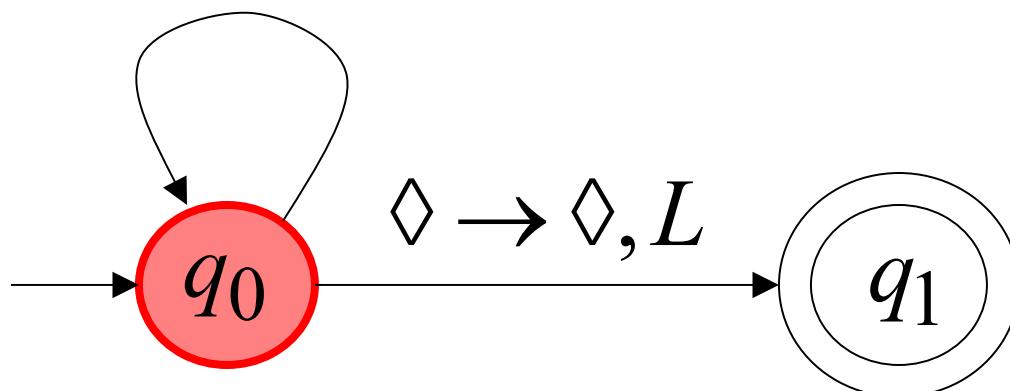
Time 1



No possible Transition

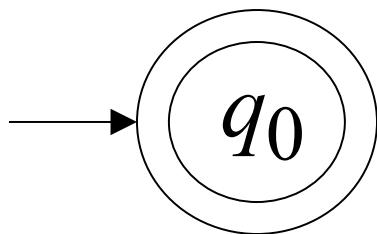
$a \rightarrow a, R$

Halt & Reject

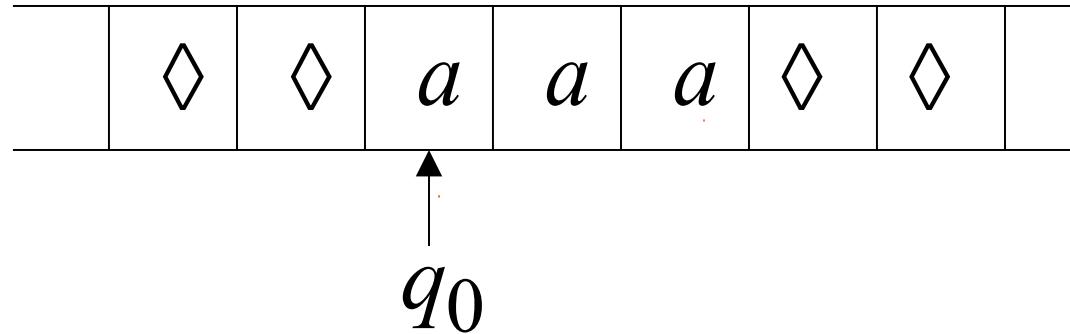


A simpler machine for same language
but for input alphabet $\Sigma = \{a\}$

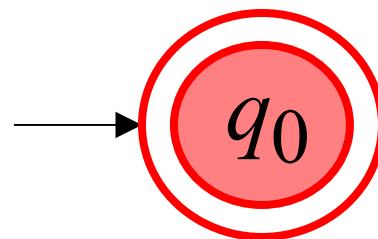
Accepts the language: a^*



Time 0



Halt & Accept



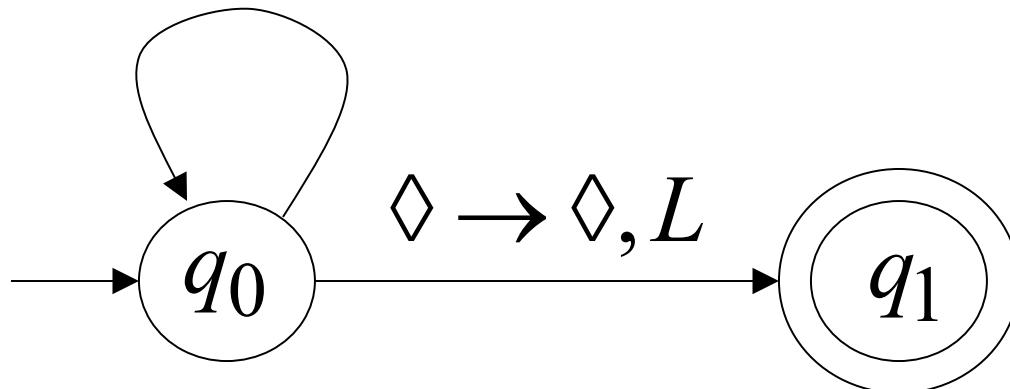
Not necessary to scan input

Infinite Loop Example

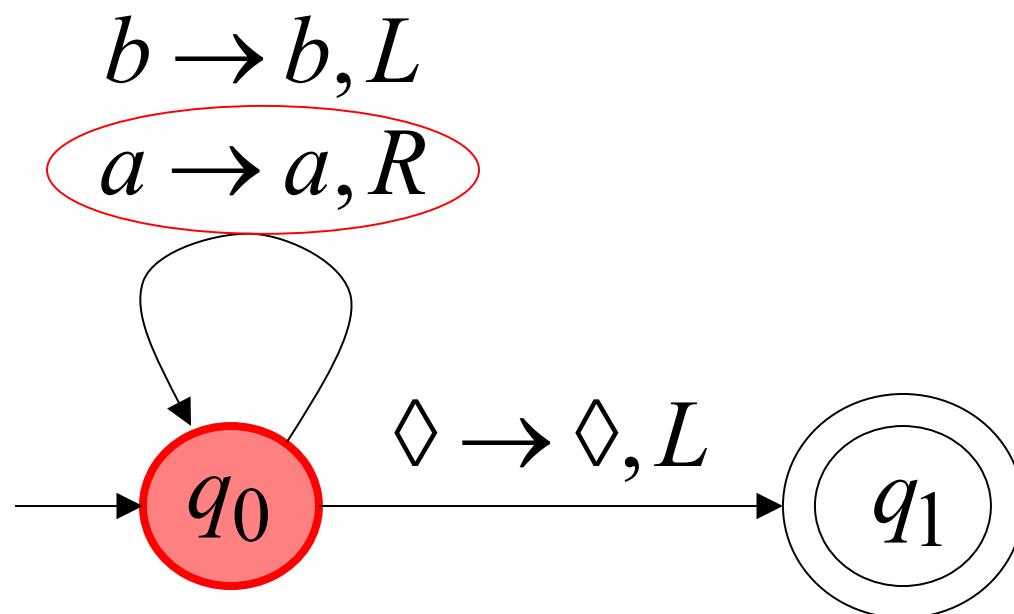
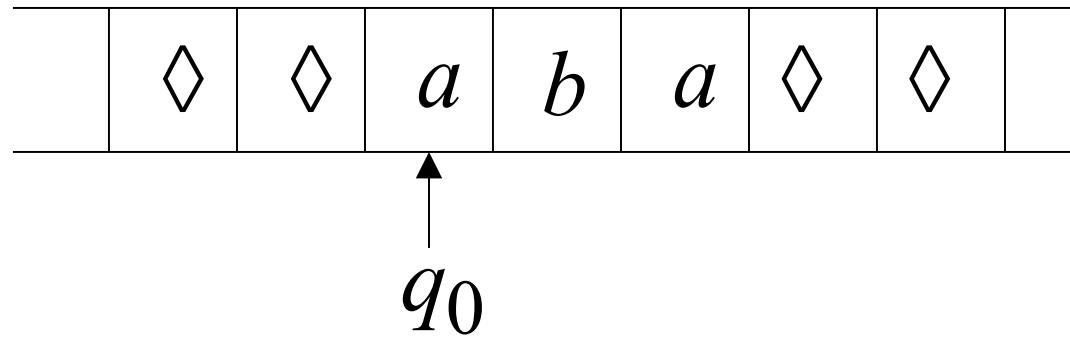
A Turing machine
for language $a^* + b(a+b)^*$

$$b \rightarrow b, L$$

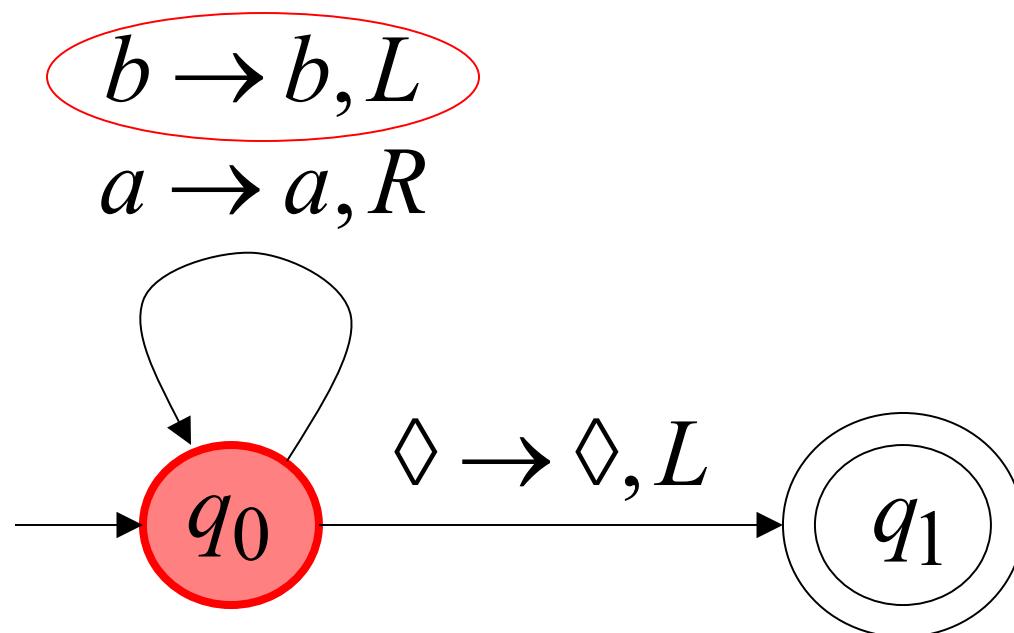
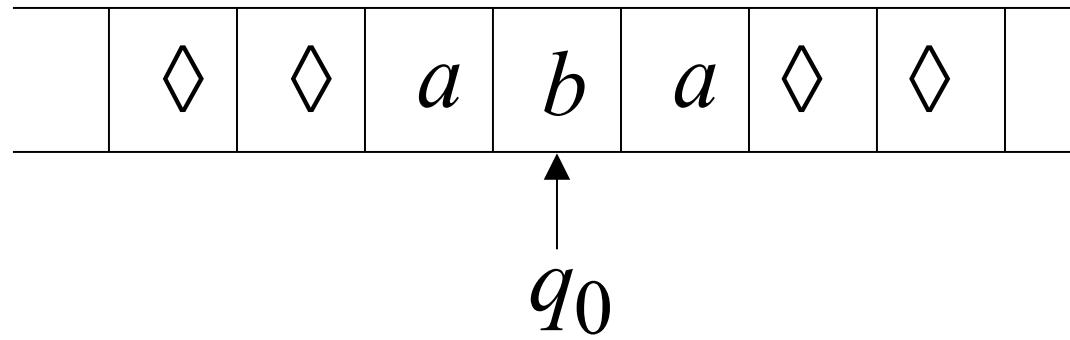
$$a \rightarrow a, R$$



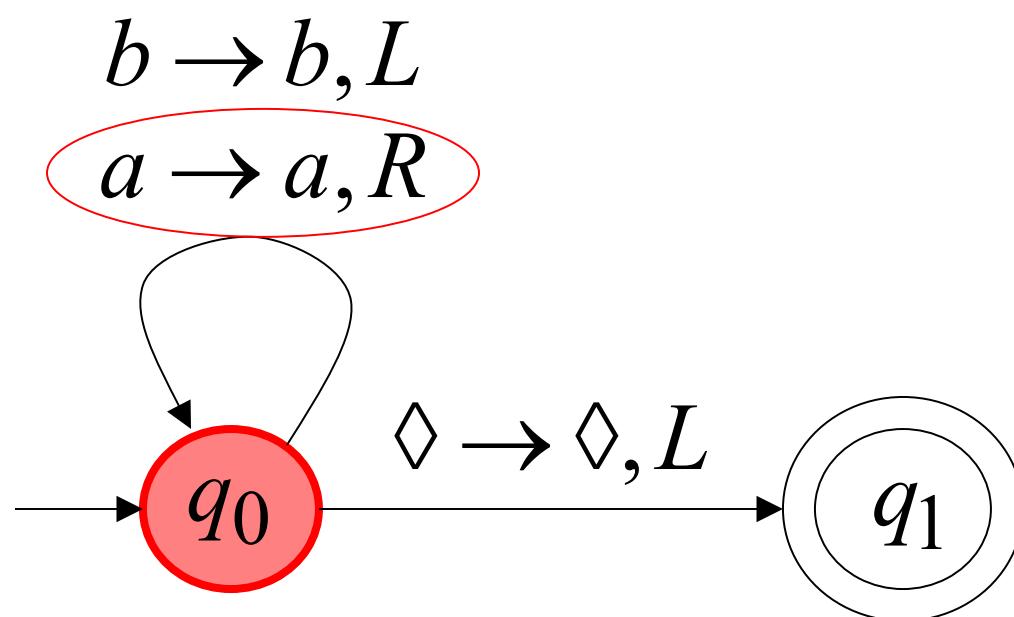
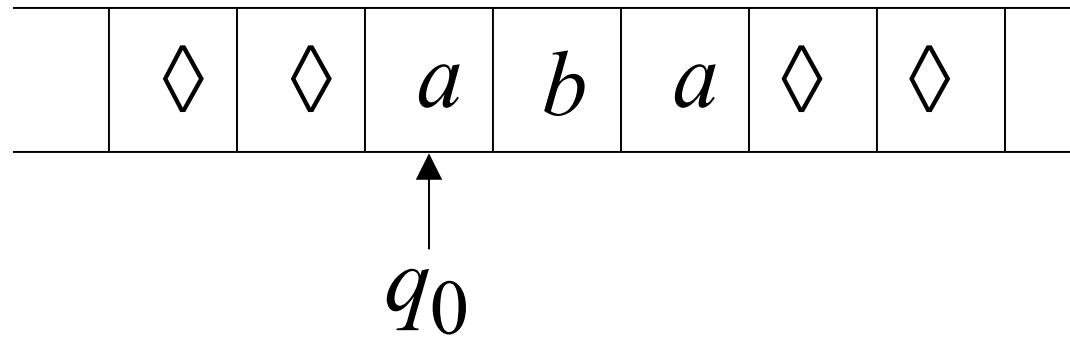
Time 0



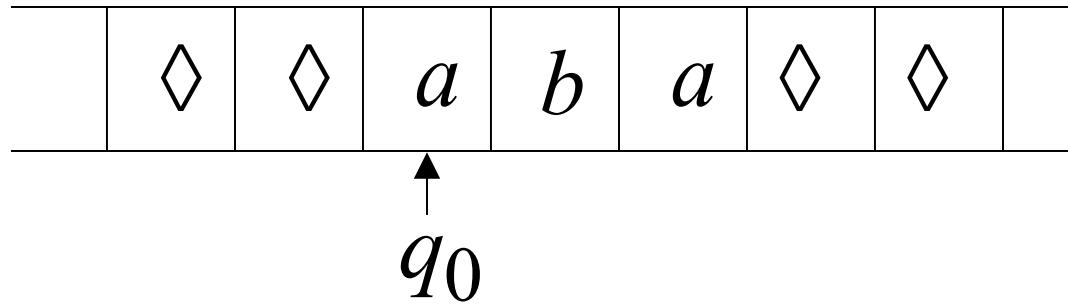
Time 1



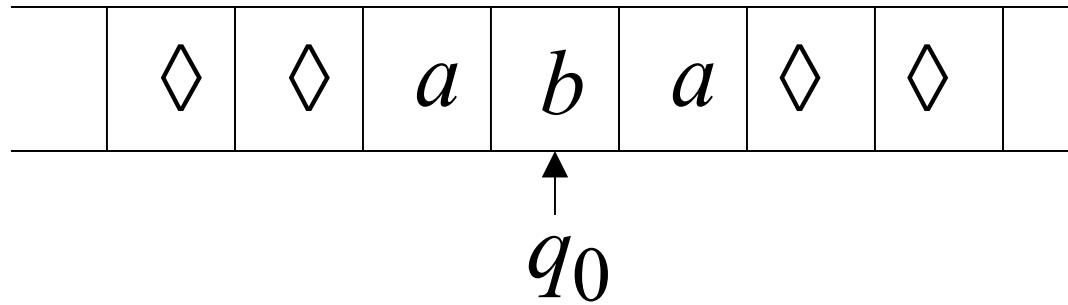
Time 2



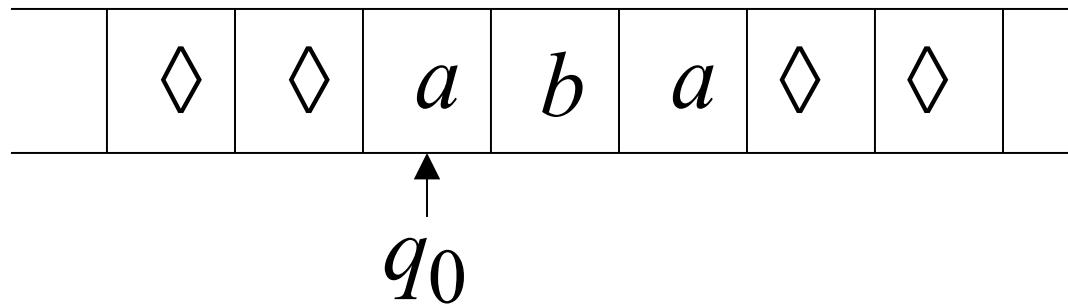
Time 2



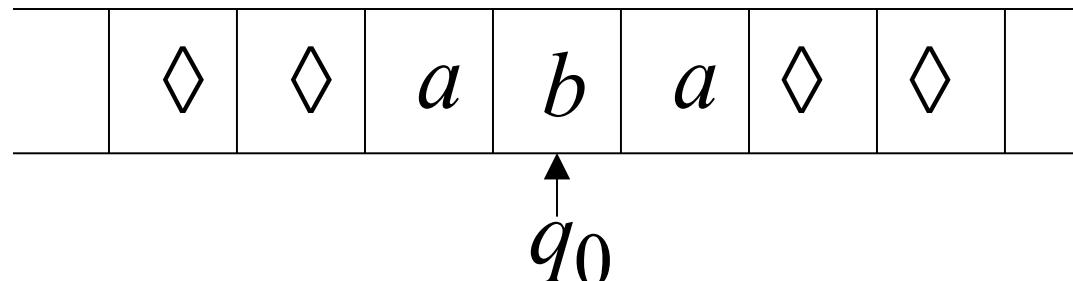
Time 3



Time 4



Time 5



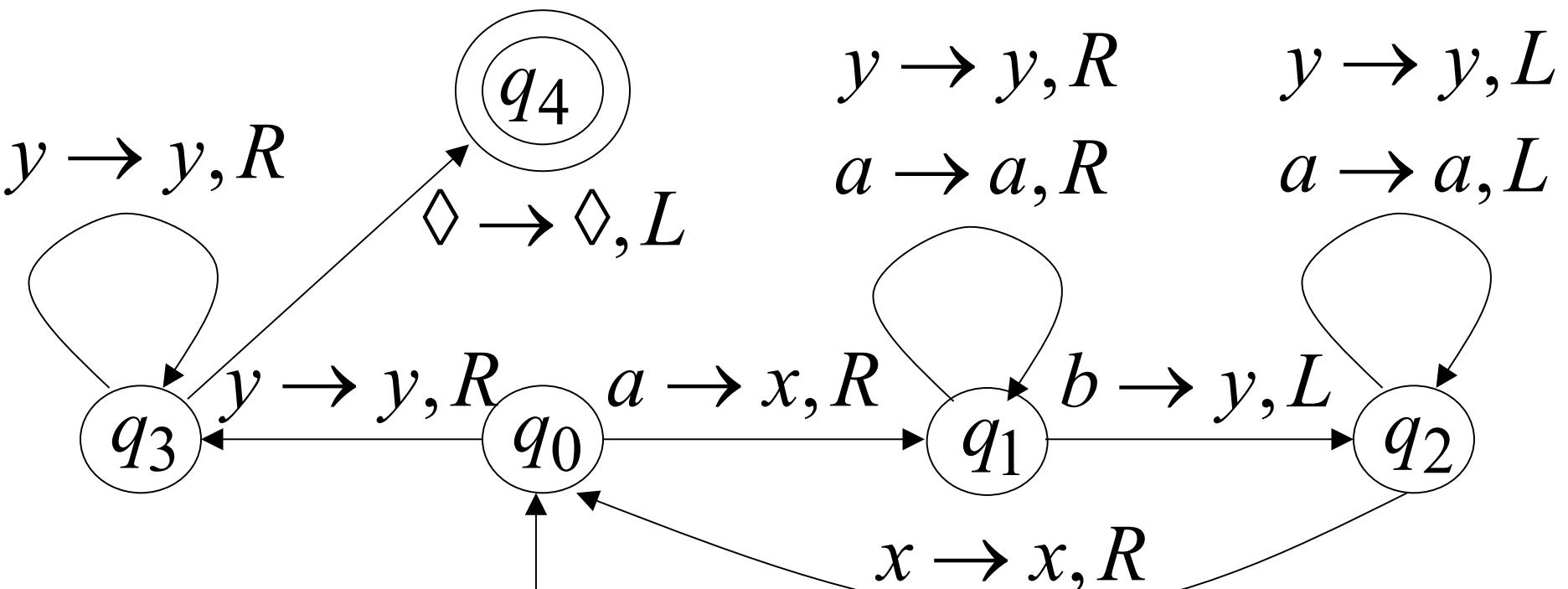
Infinite loop

Because of the infinite loop:

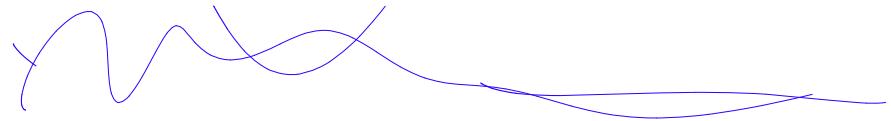
- The accepting state cannot be reached
- The machine never halts
- The input string is rejected

Another Turing Machine Example

Turing machine for the language $\{a^n b^n\}$
 $n \geq 1$



Basic Idea:



Match a's with b's:

Repeat:

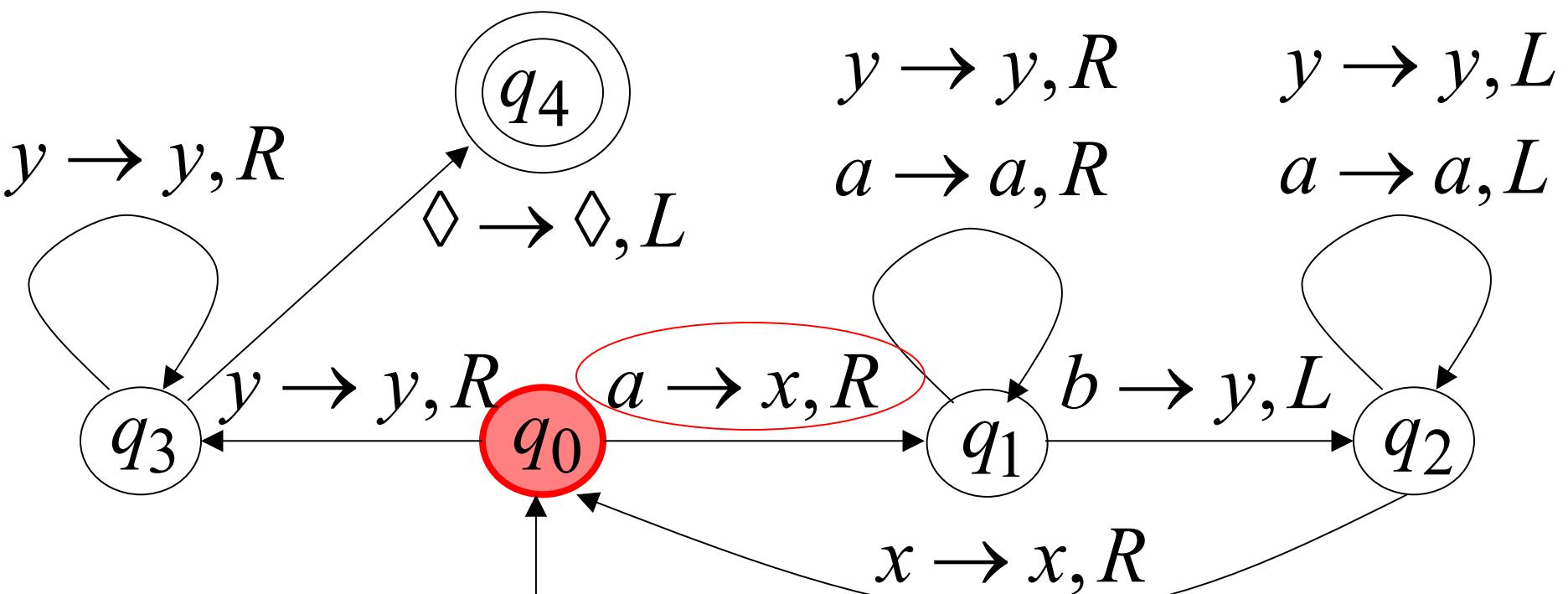
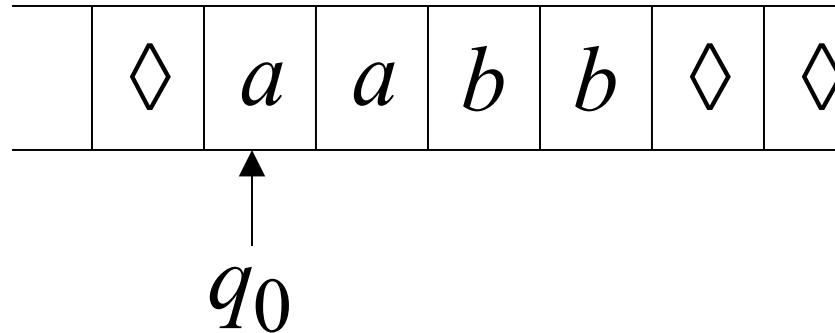
 replace leftmost a with x

 find leftmost b and replace it with y

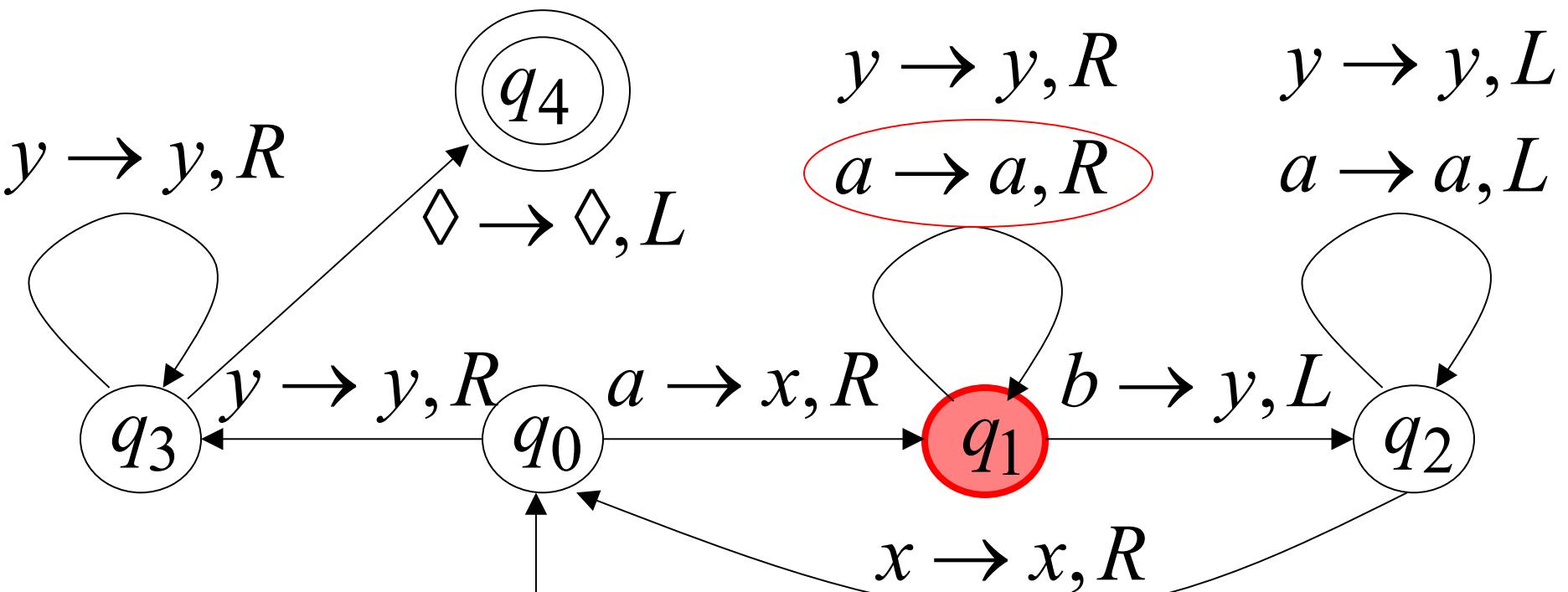
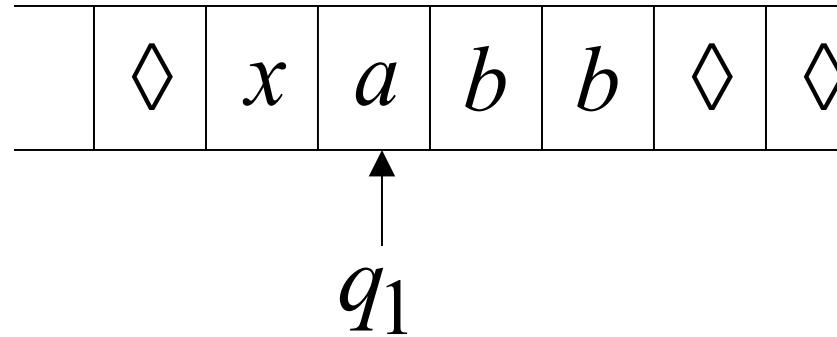
Until there are no more a's or b's

If there is a remaining a or b reject

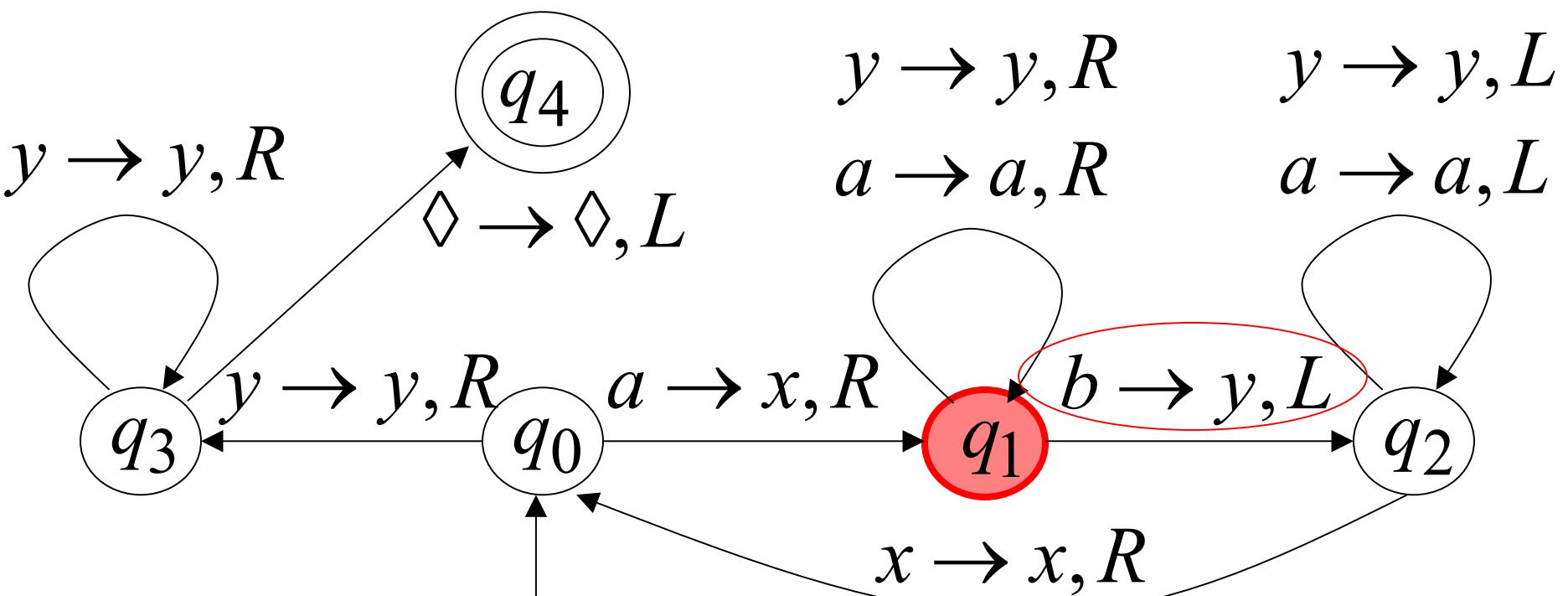
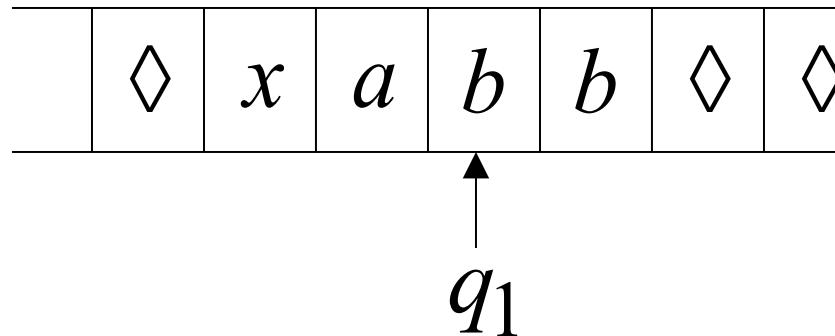
Time 0



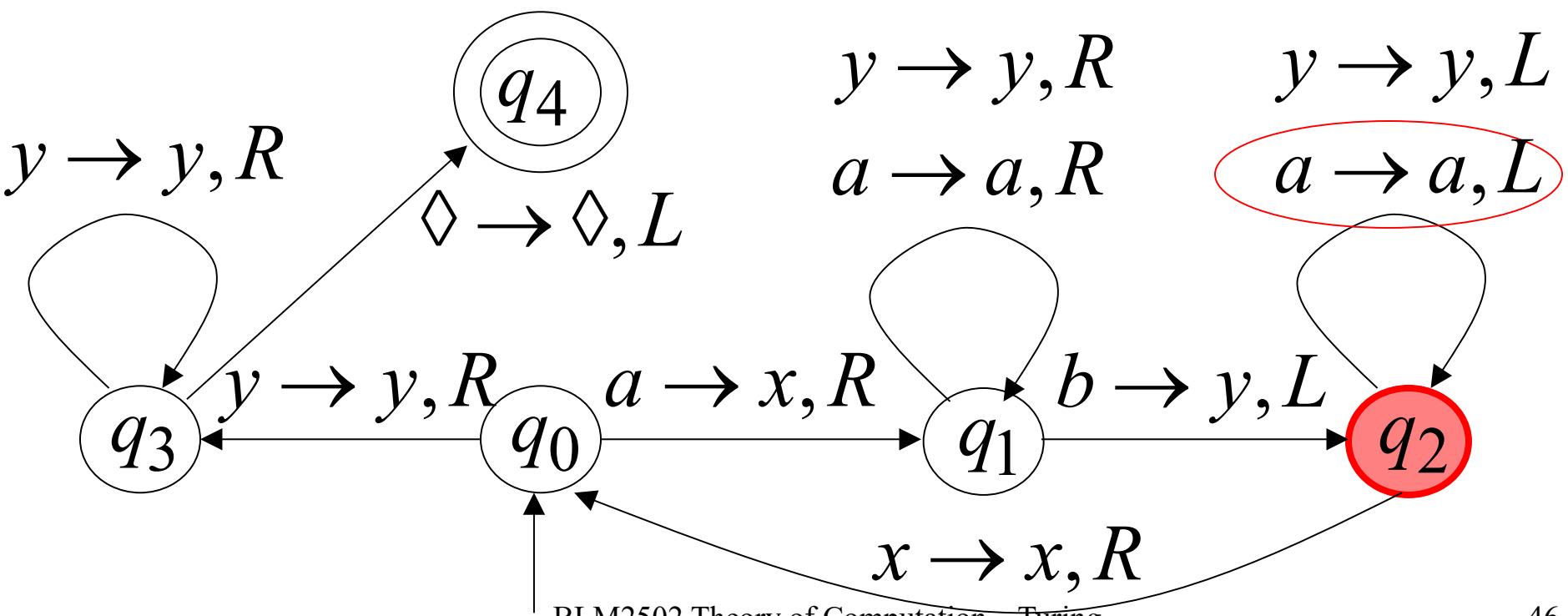
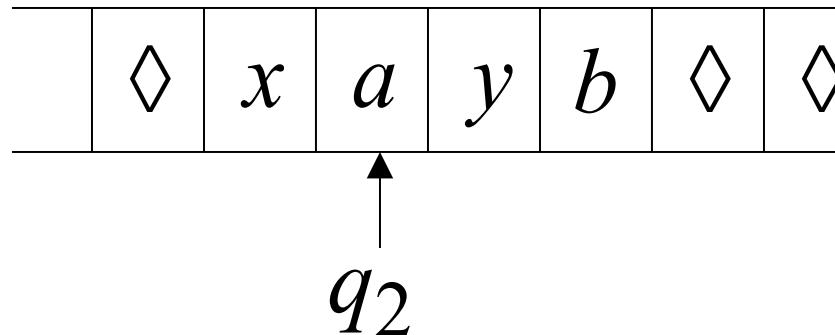
Time 1



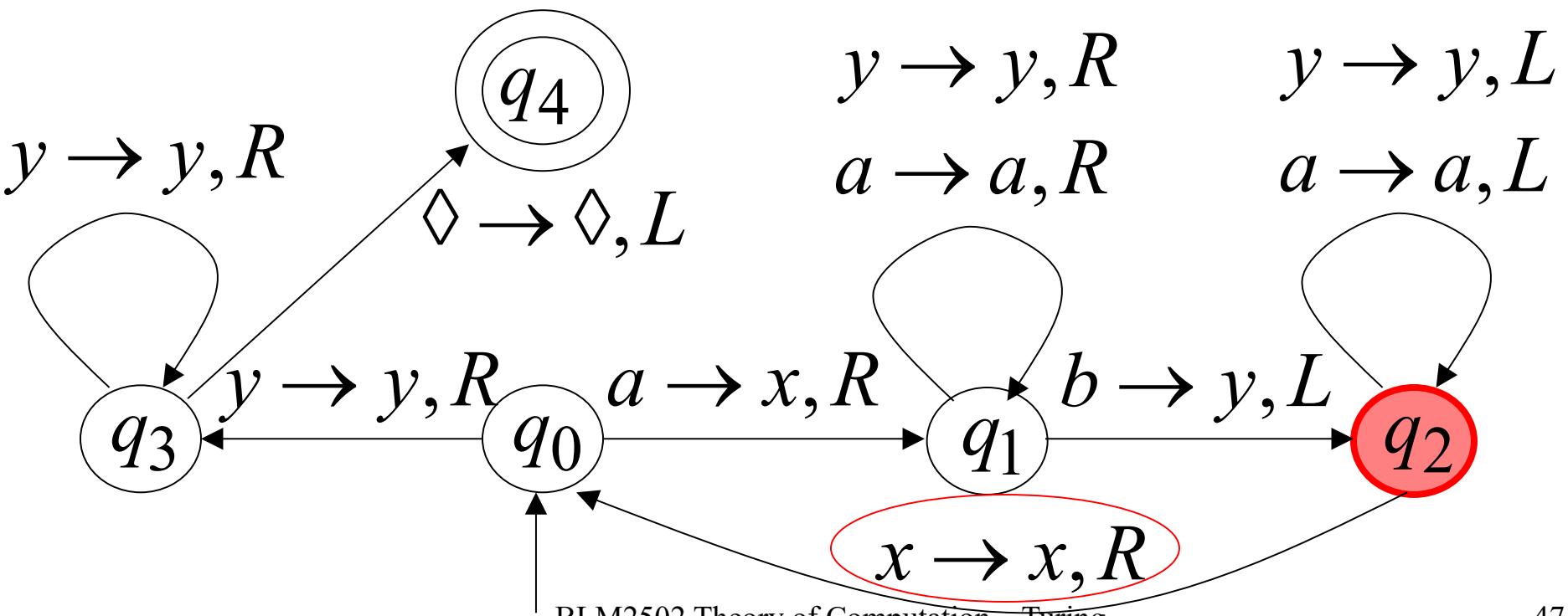
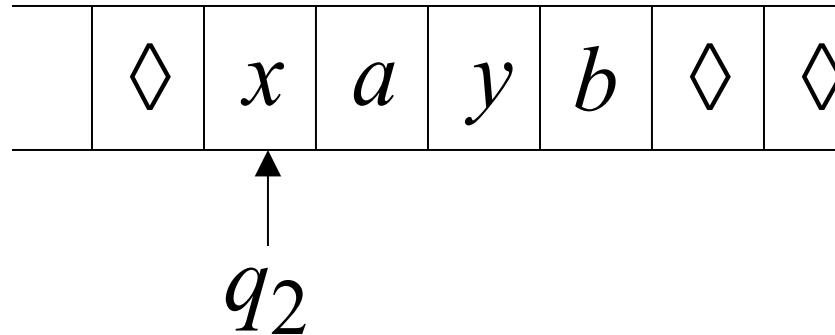
Time 2



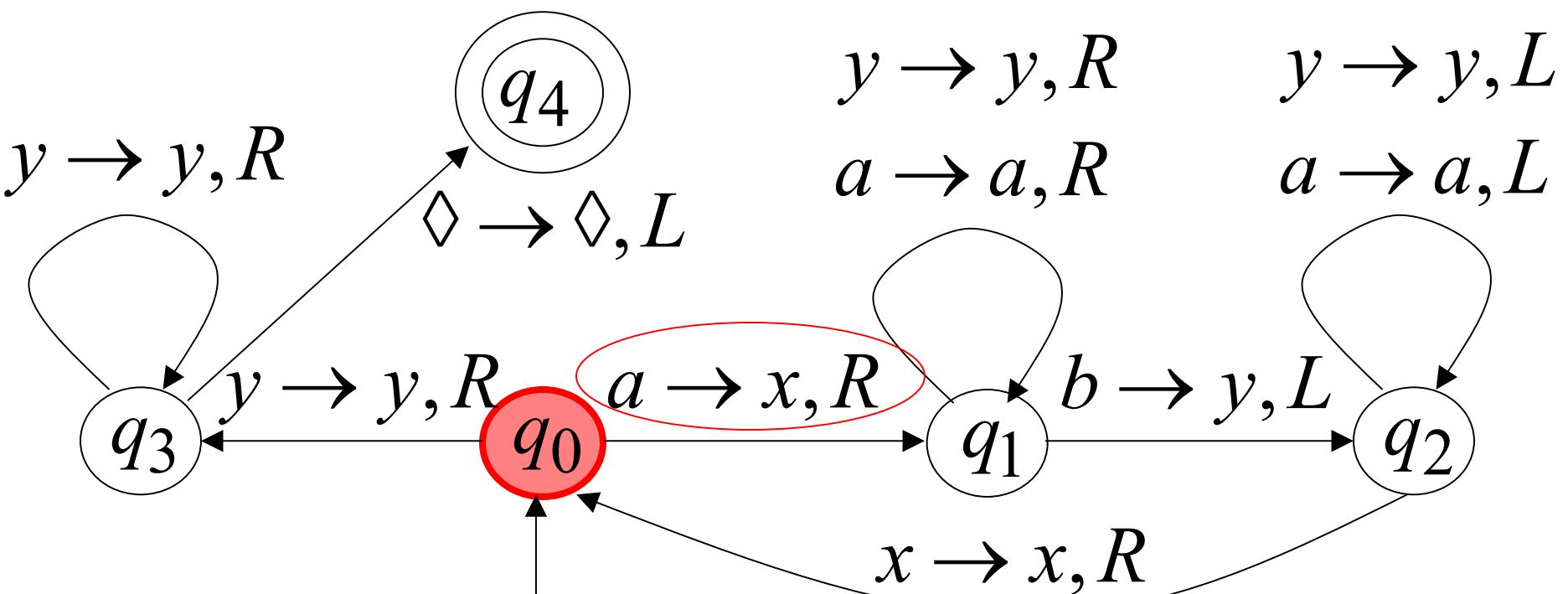
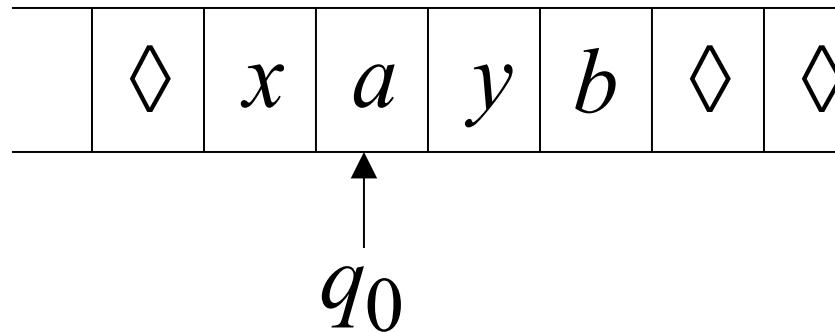
Time 3



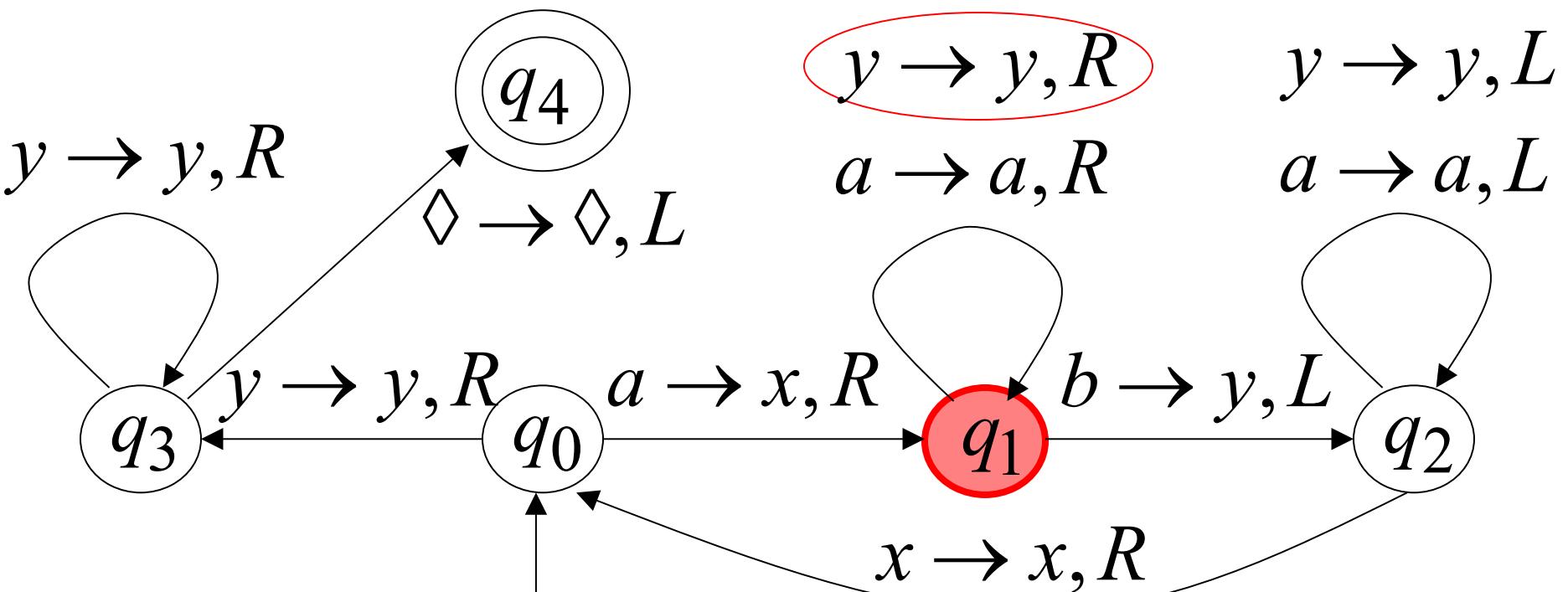
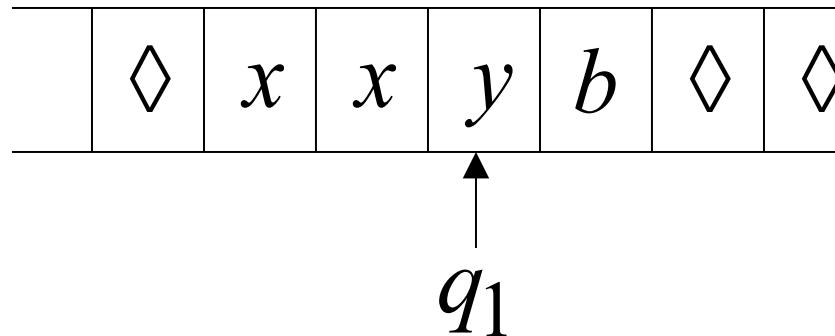
Time 4



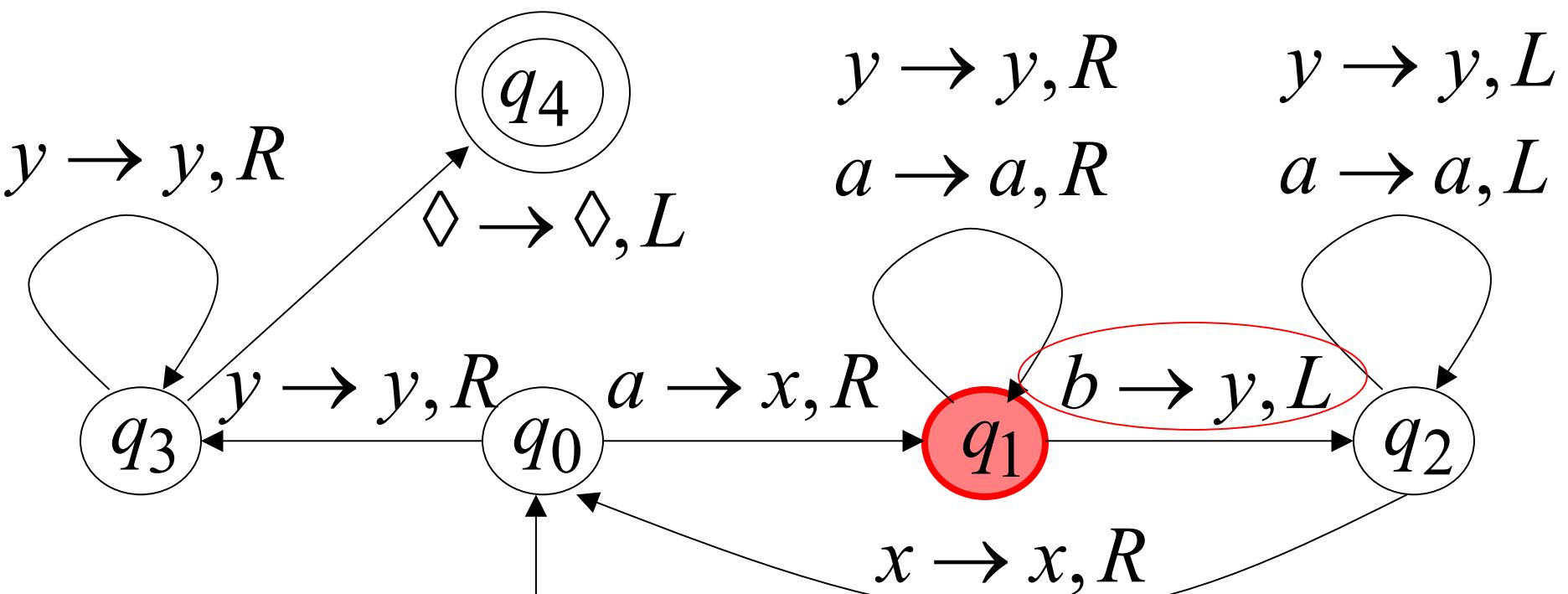
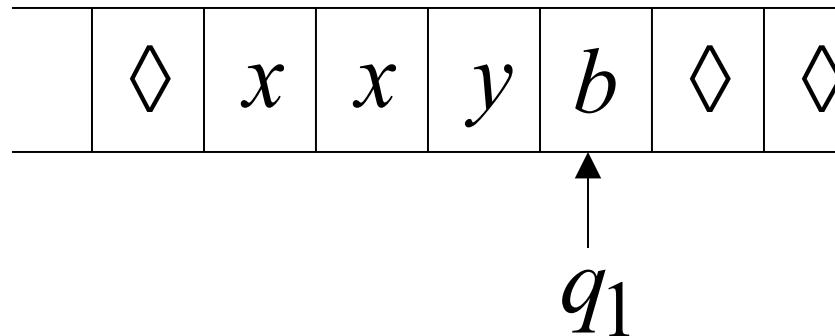
Time 5



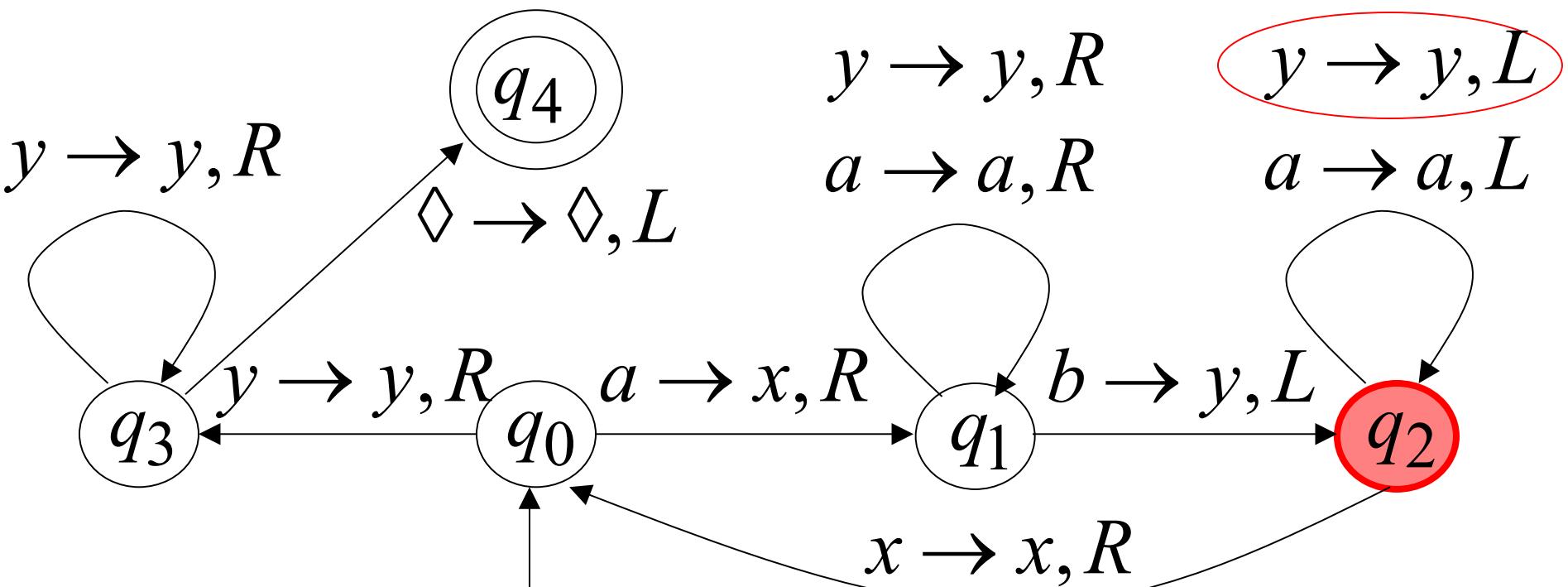
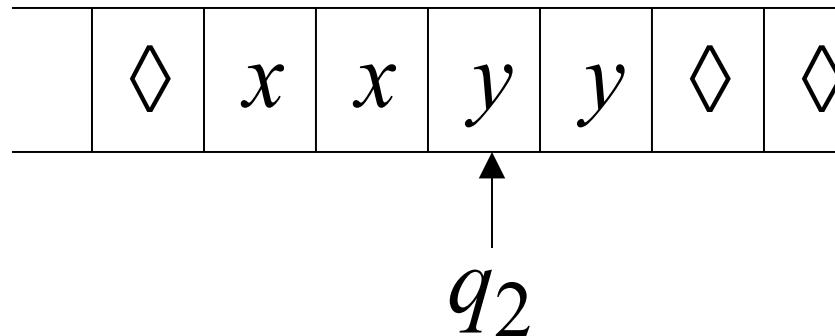
Time 6



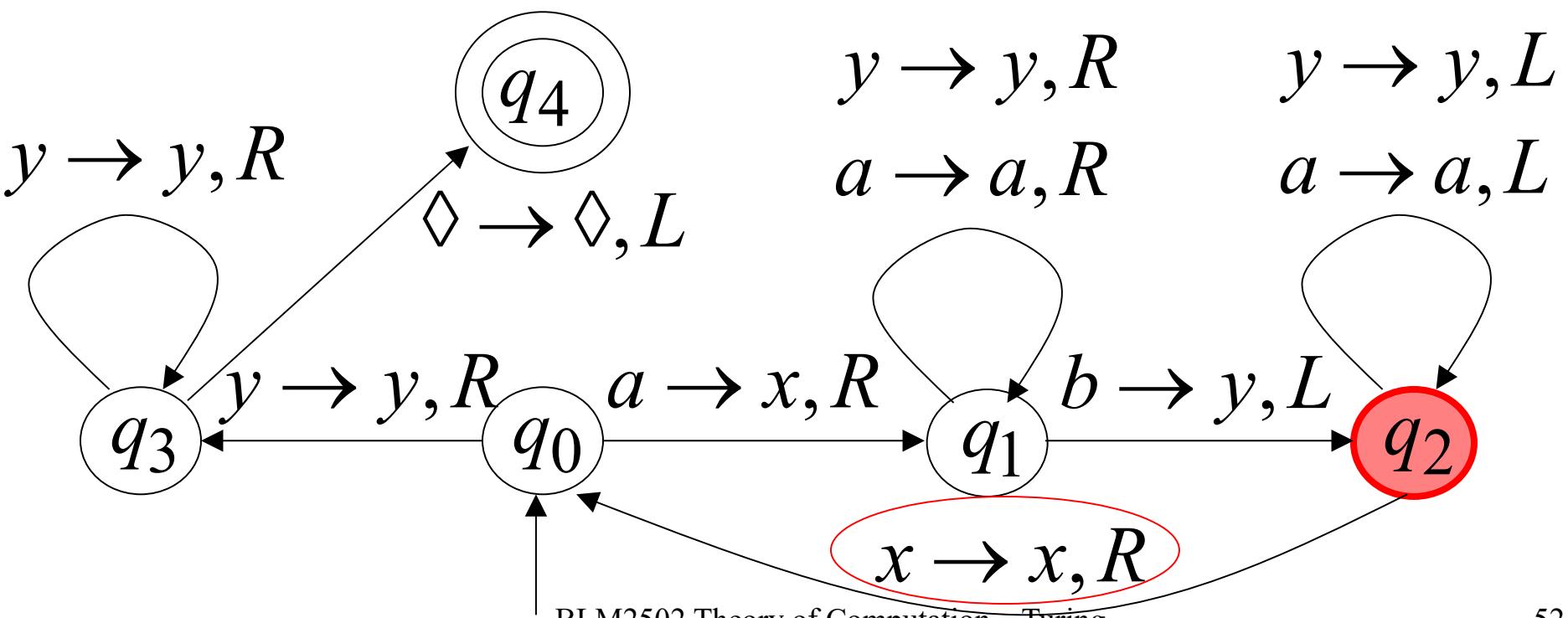
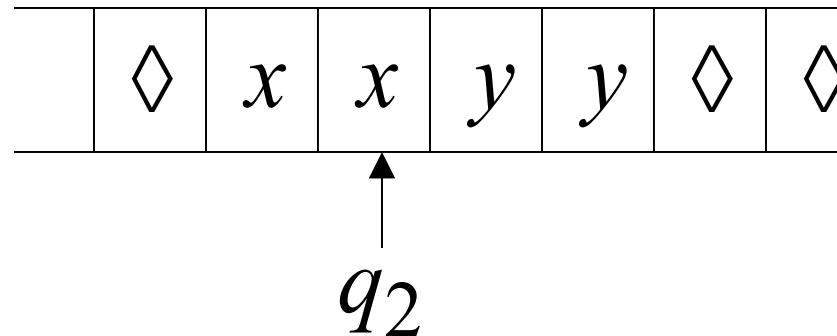
Time 7



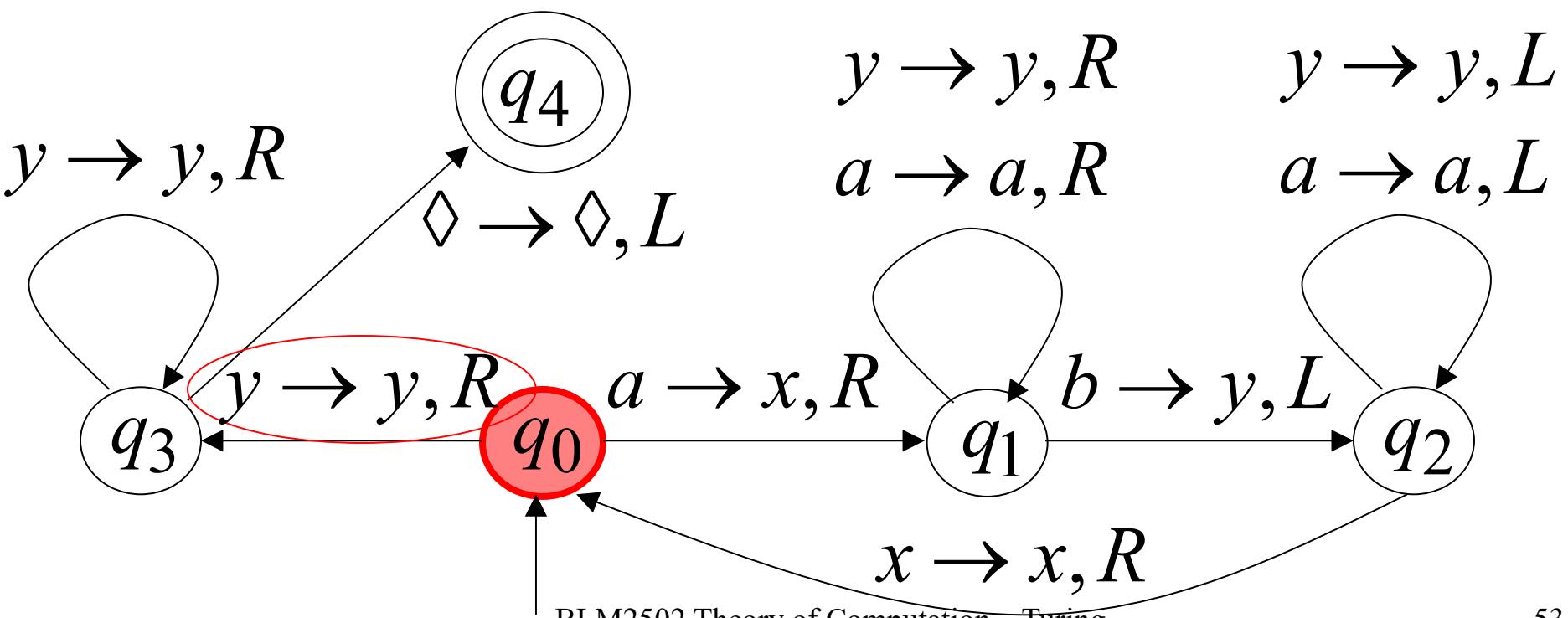
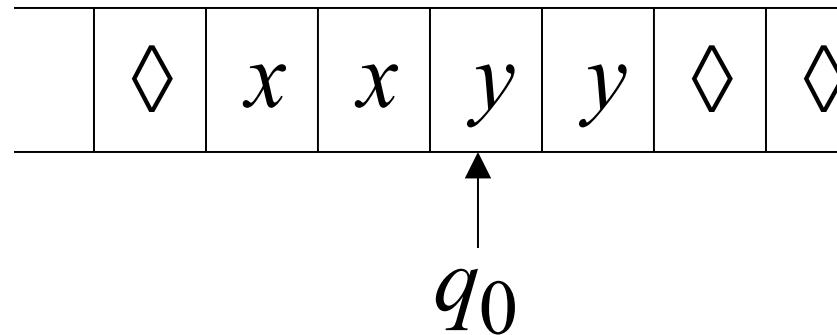
Time 8



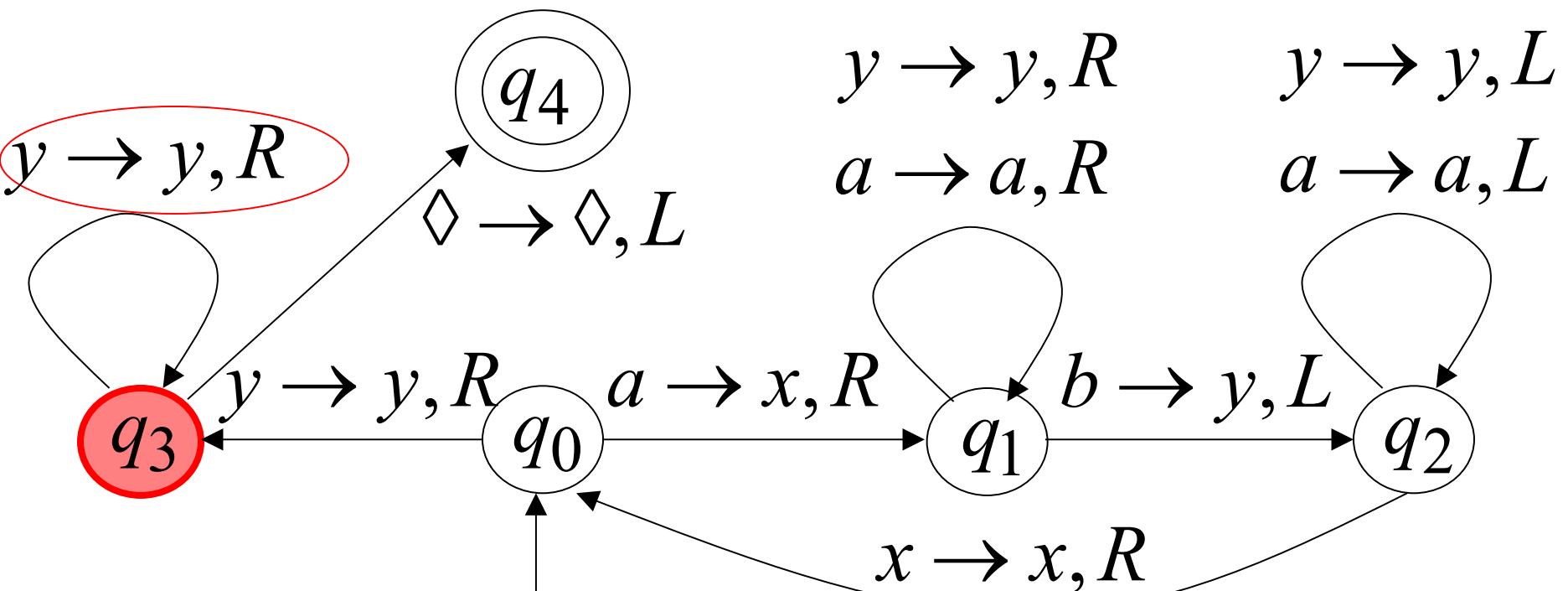
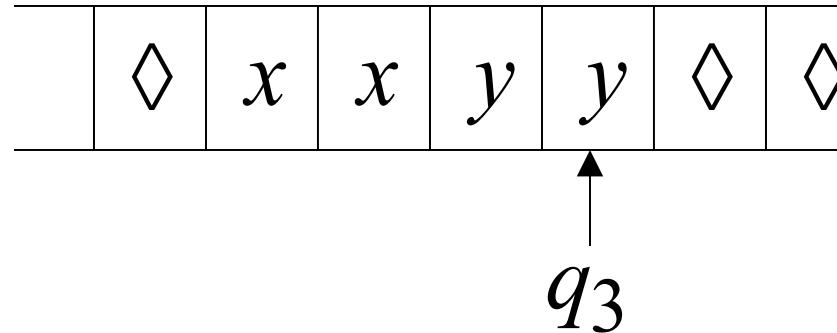
Time 9



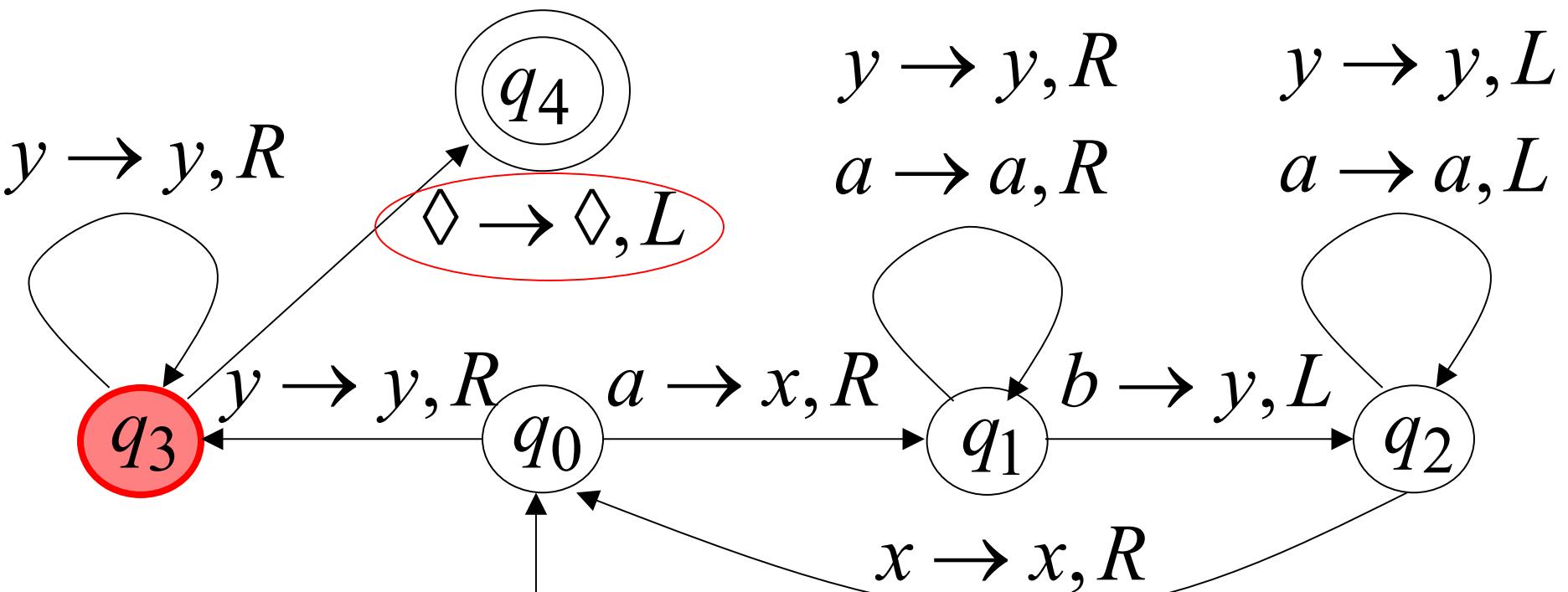
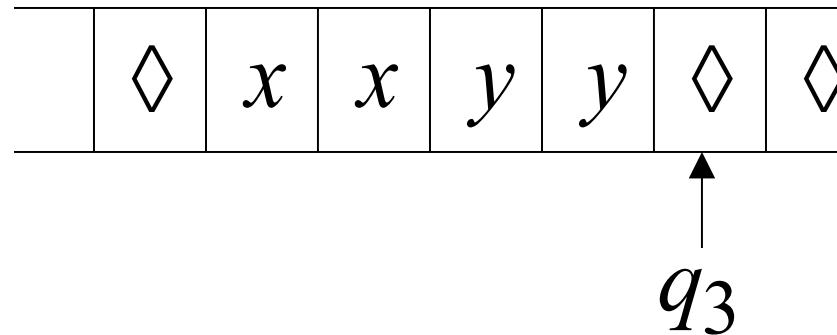
Time 10



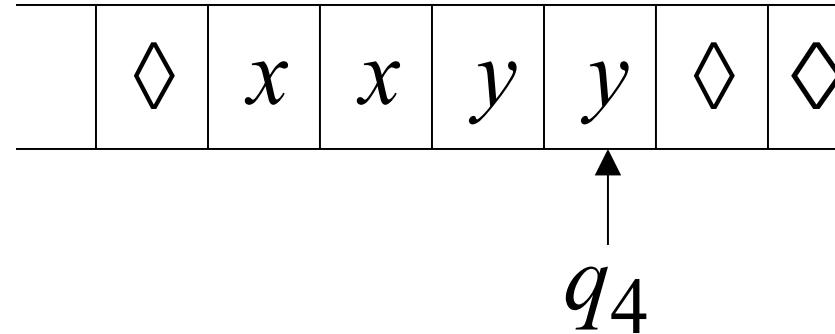
Time 11



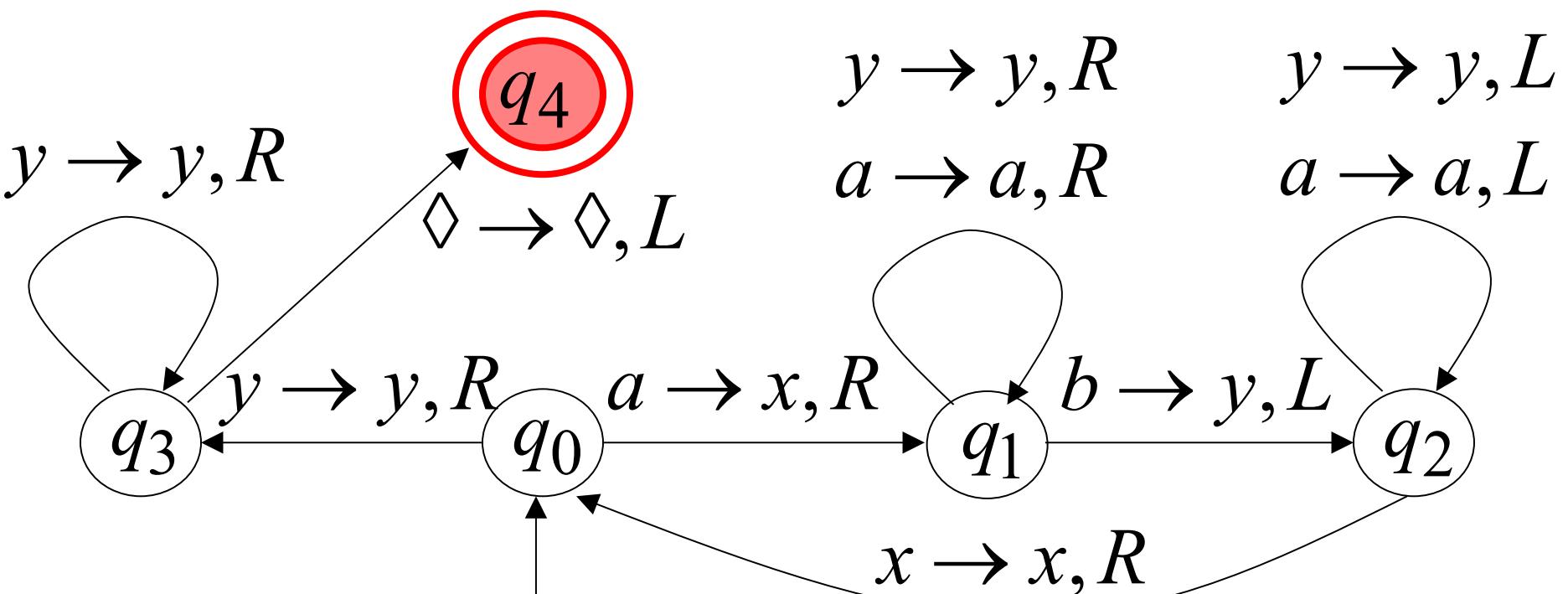
Time 12



Time 13



Halt & Accept



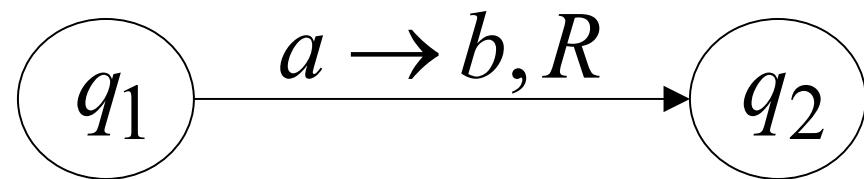
Observation:

If we modify the machine for the language $\{a^n b^n\}$

we can easily construct a machine for the language $\{a^n b^n c^n\}$

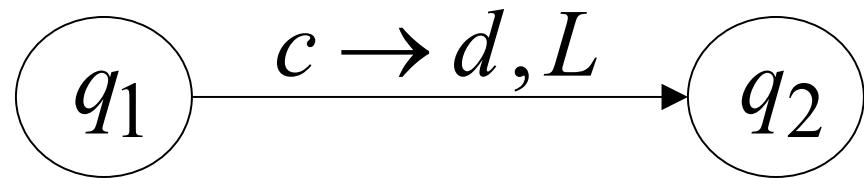
Formal Definitions for Turing Machines

Transition Function



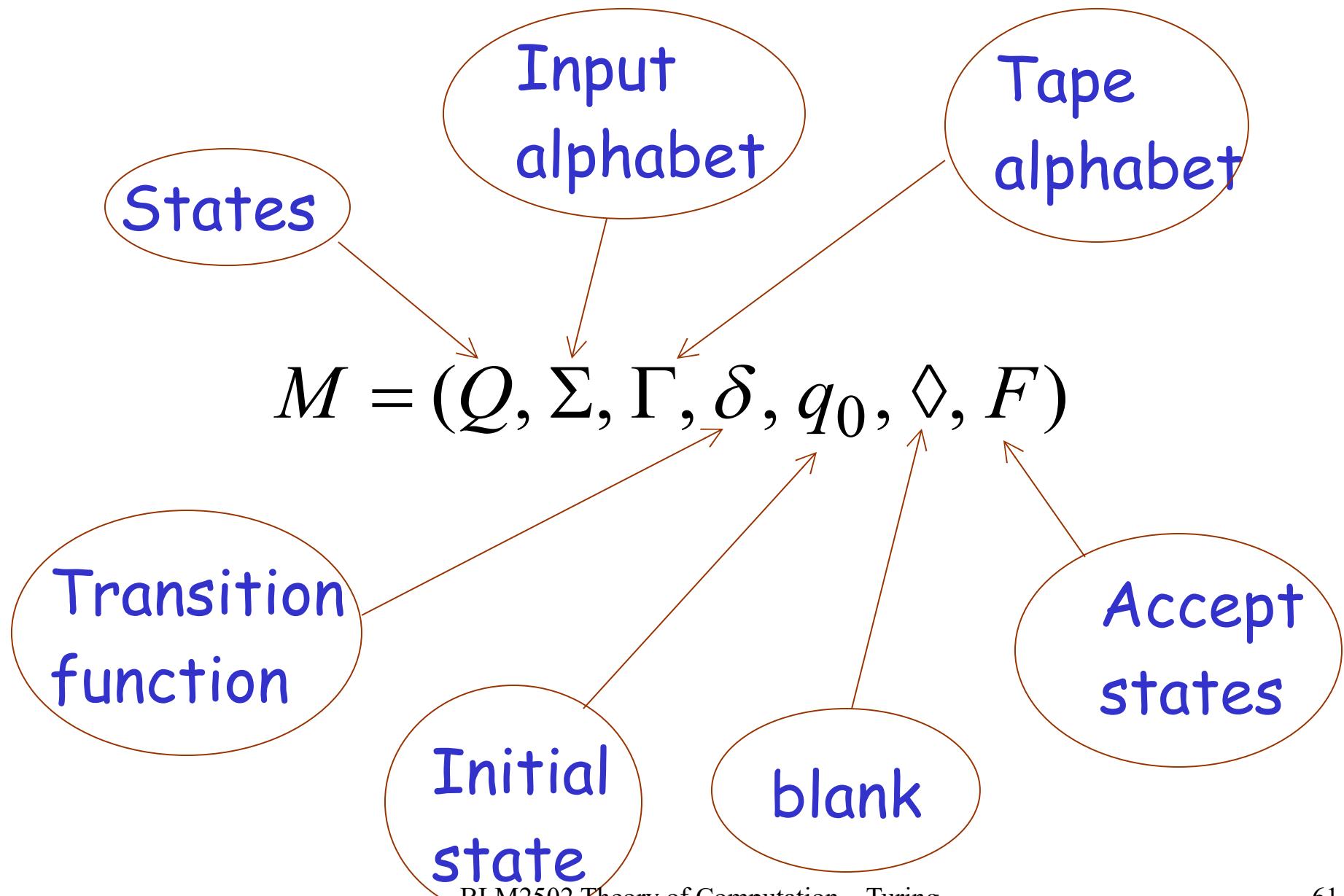
$$\delta(q_1, a) = (q_2, b, R)$$

Transition Function

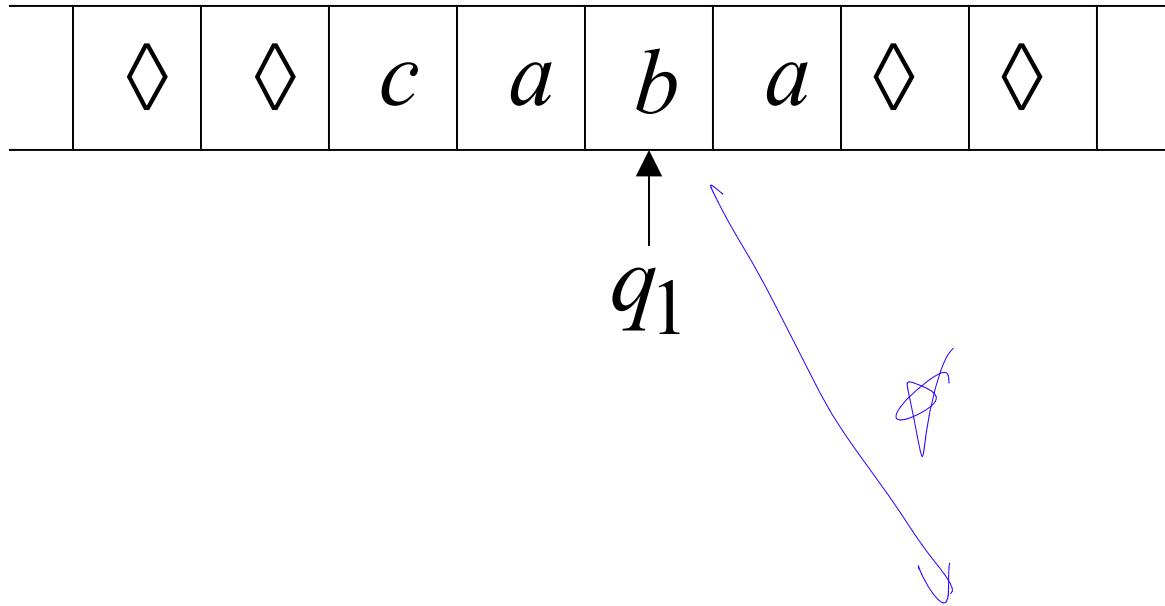


$$\delta(q_1, c) = (q_2, d, L)$$

Turing Machine:



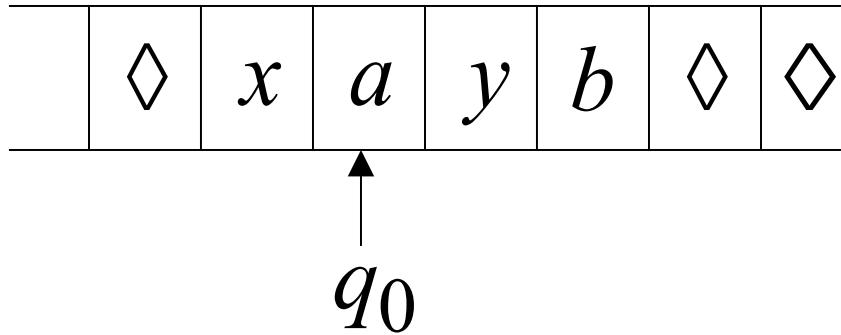
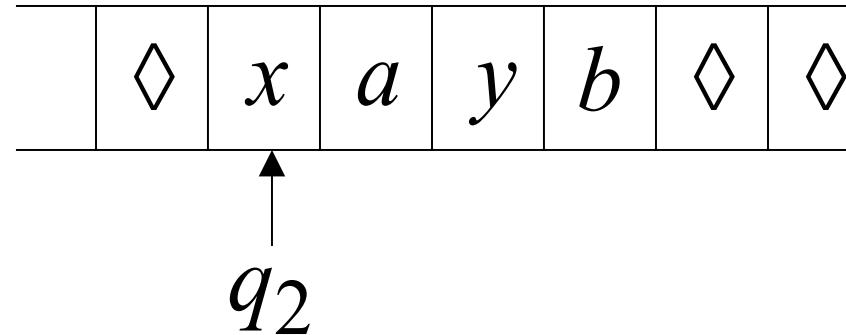
Configuration



Instantaneous description: $ca\ q_1\ ba$

Time 4

Time 5



A Move:

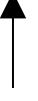
$$q_2 \ xayb \succ x \ q_0 \ ayb$$

(yields in one mode)

Time 4

	◊	x	a	y	b	◊	◊
--	---	---	---	---	---	---	---

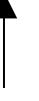
q_2



Time 5

	◊	x	a	y	b	◊	◊
--	---	---	---	---	---	---	---

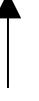
q_0



Time 6

	◊	x	x	y	b	◊	◊
--	---	---	---	---	---	---	---

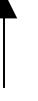
q_1



Time 7

	◊	x	x	y	b	◊	◊
--	---	---	---	---	---	---	---

q_1



A computation

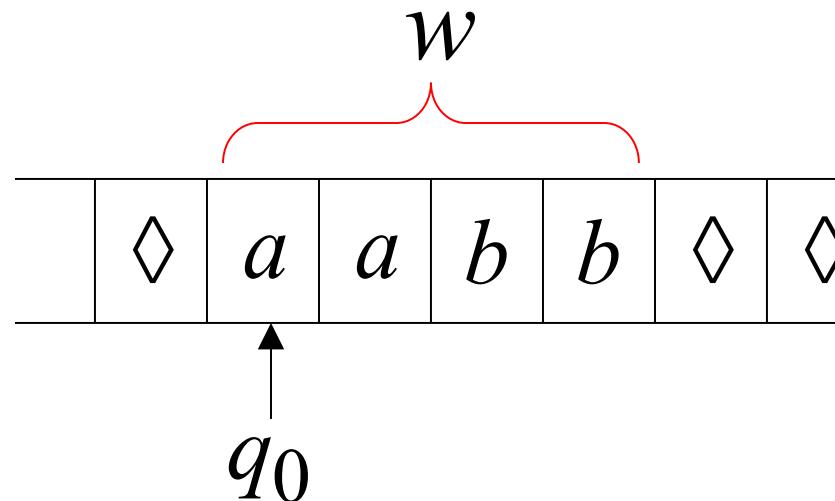
$q_2 \ xayb \succ x \ q_0 \ ayb \succ xx \ q_1 \ yb \succ xxy \ q_1 \ b$

$$q_2 \ xayb \succ x \ q_0 \ ayb \succ xx \ q_1 \ yb \succ xxy \ q_1 \ b$$

Equivalent notation:
$$q_2 \ xayb \stackrel{*}{\succ} xxy \ q_1 \ b$$

Initial configuration: $q_0 \ w$

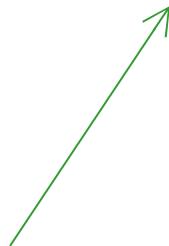
Input string



The Accepted Language

For any Turing Machine M

$$L(M) = \{w : q_0 w \xrightarrow{*} x_1 q_f x_2\}$$



Initial state



Accept state

If a language L is accepted
by a Turing machine M
then we say that L is:

- Turing Recognizable

Other names used:

- Turing Acceptable
- Recursively Enumerable

Computing Functions with Turing Machines

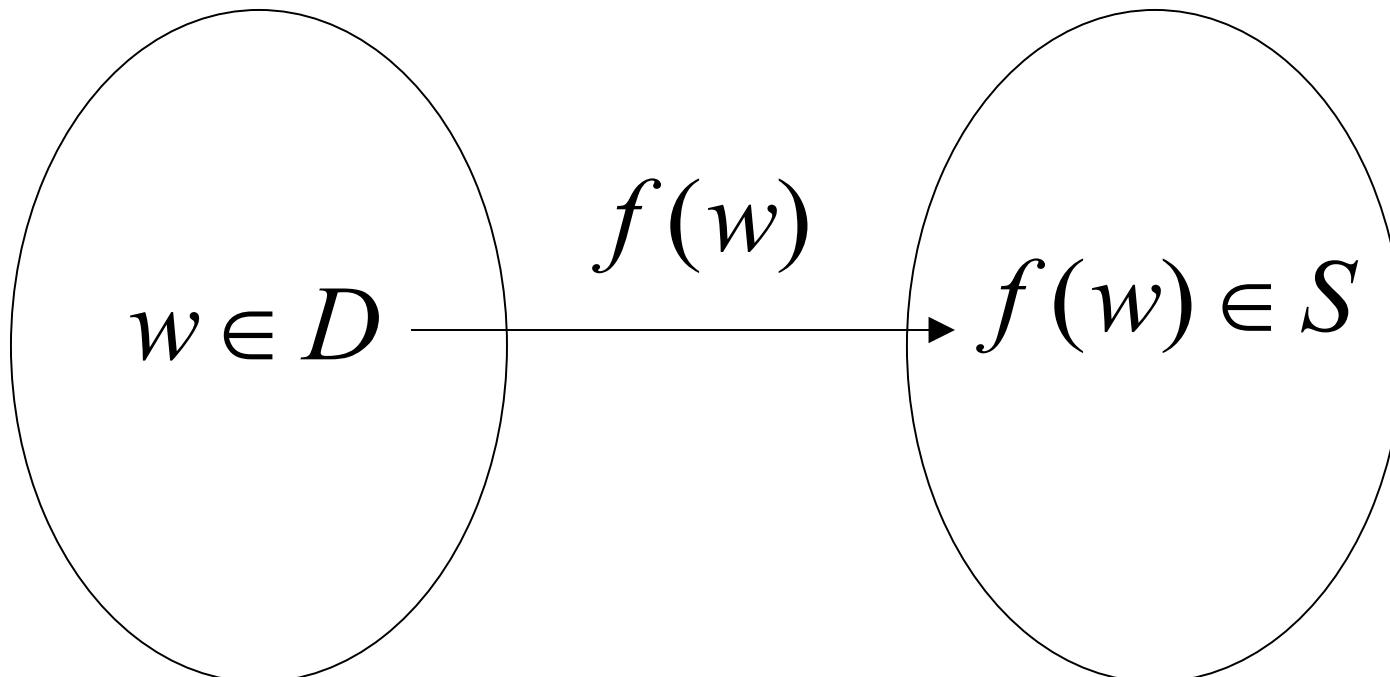
A function

$f(w)$

has:

Domain: D

Result Region: S



A function may have many parameters:

Example: Addition function

$$f(x, y) = x + y$$

Integer Domain

Decimal: 5

Binary: 101

Unary: 11111

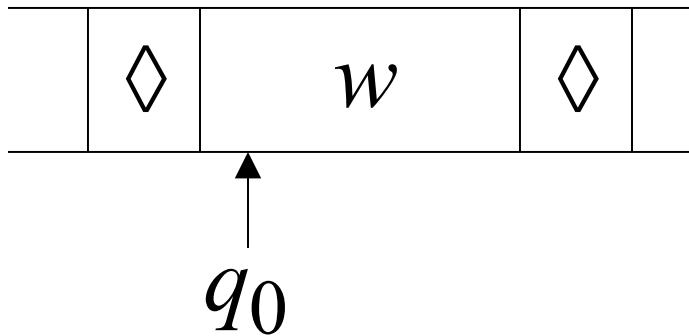
We prefer **unary** representation:

easier to manipulate with Turing machines

Definition:

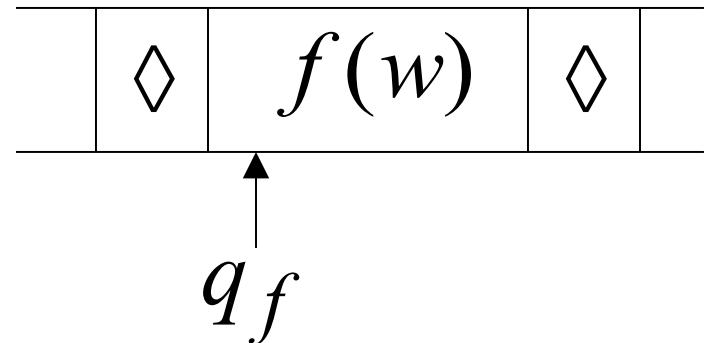
A function f is computable if there is a Turing Machine M such that:

Initial configuration



initial state

Final configuration

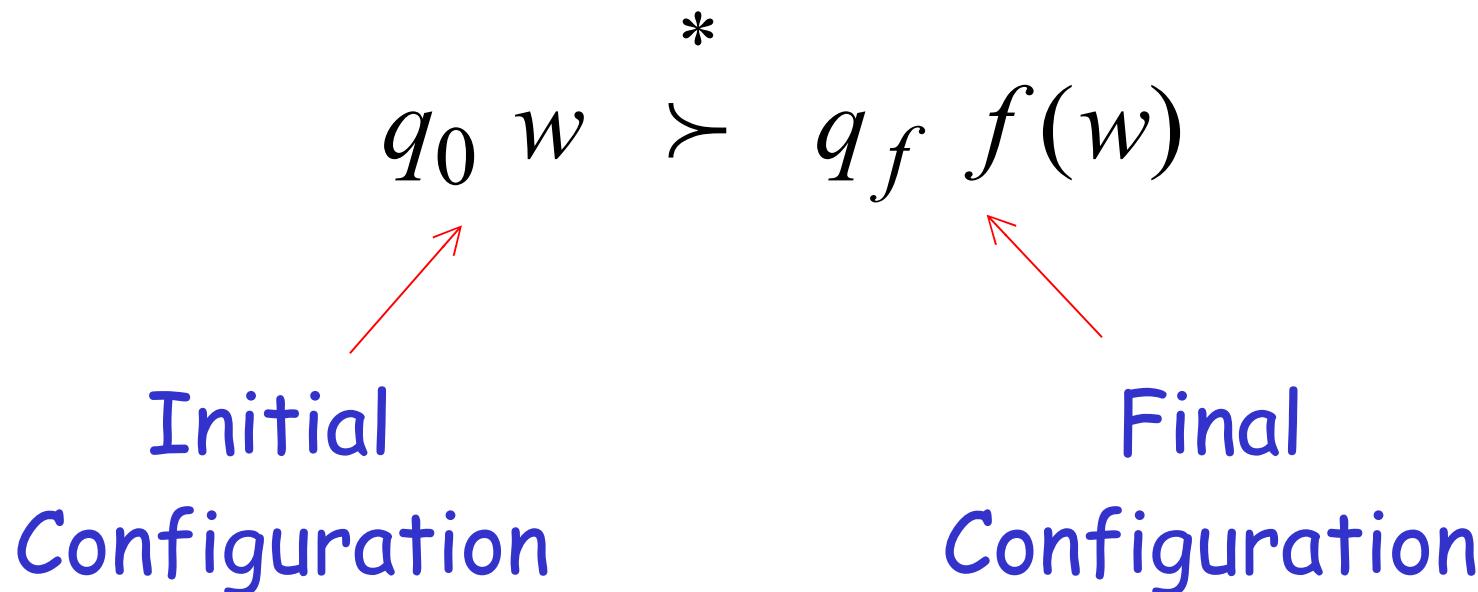


accept state

For all $w \in D$ Domain

In other words:

A function f is computable if there is a Turing Machine M such that:



For all $w \in D$ Domain

Example

The function $f(x, y) = x + y$ is computable

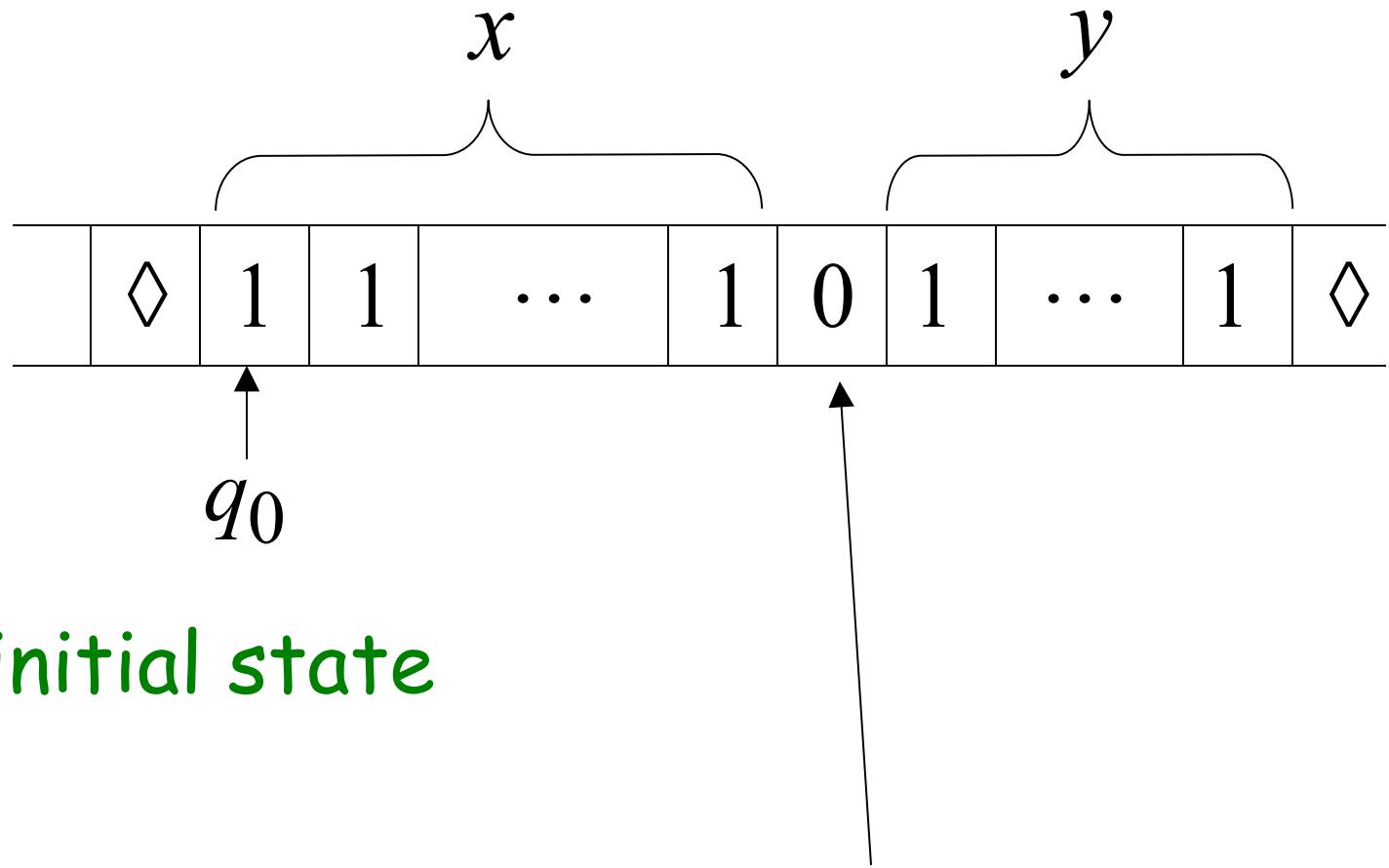
x, y are integers

Turing Machine:

Input string: $x0y$ unary

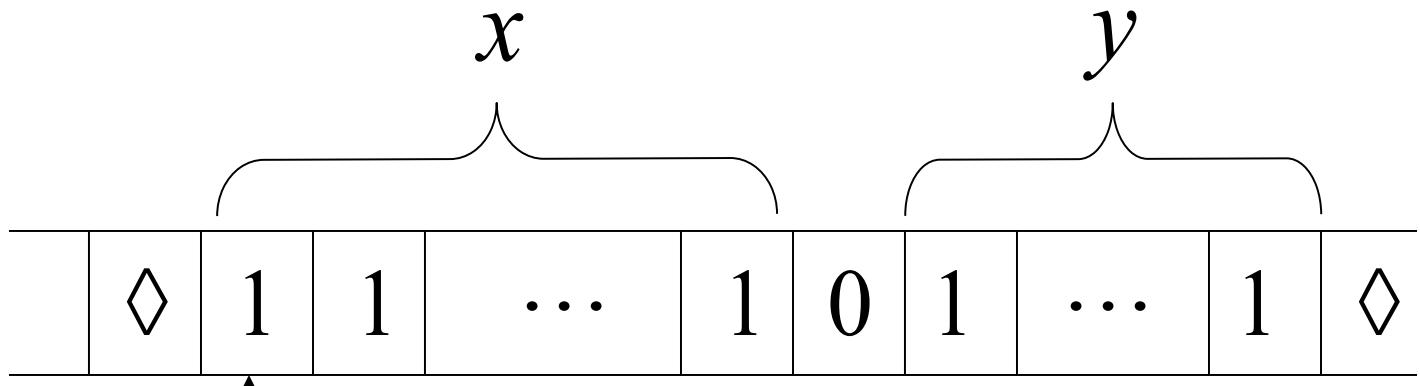
Output string: $xy0$ unary

Start



The 0 is the delimiter that separates the two numbers

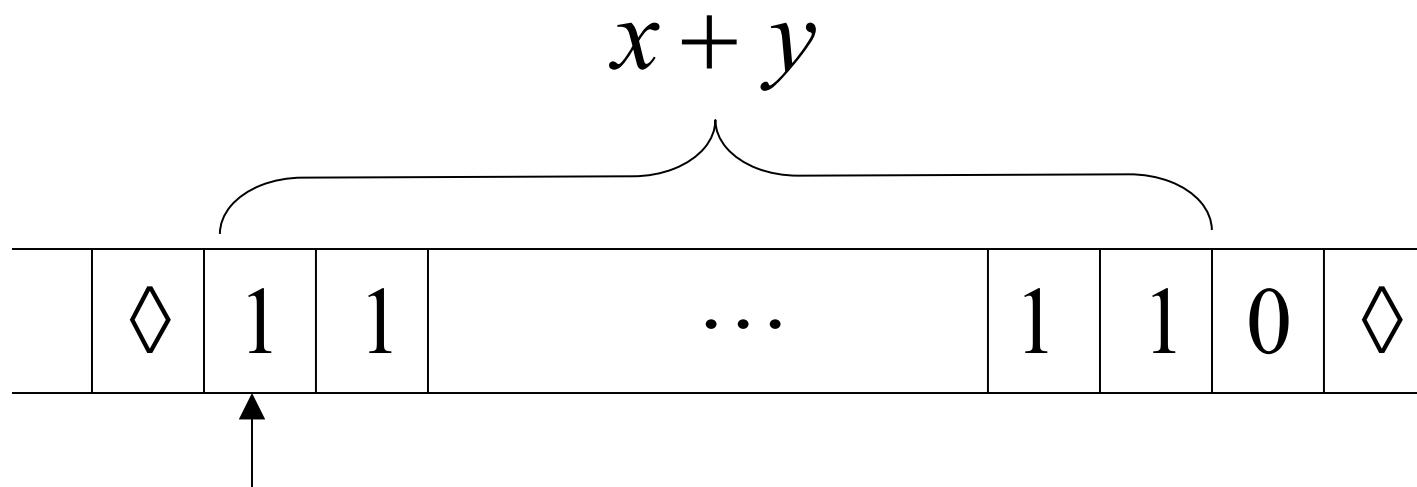
Start



q_0 initial state

Fix \diamond as \perp
gapless

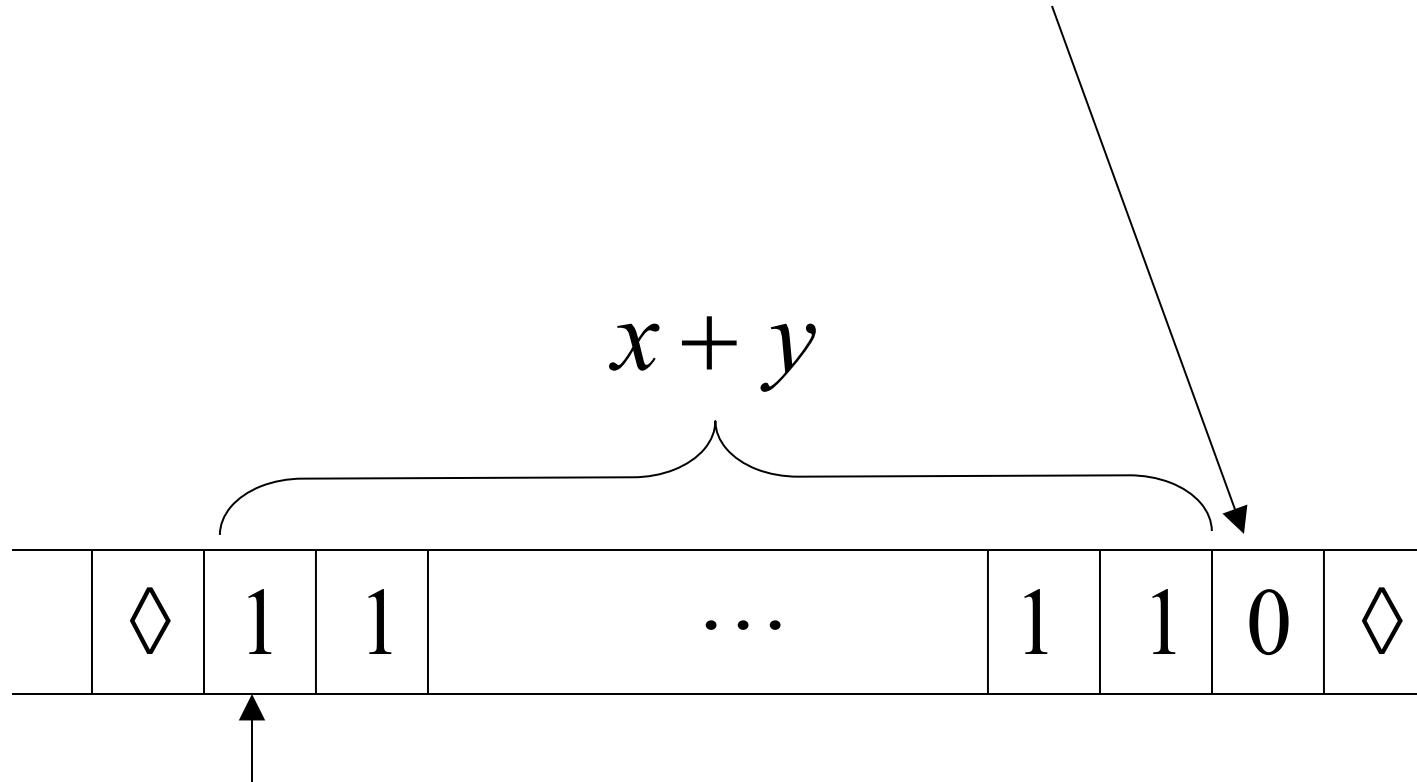
Finish



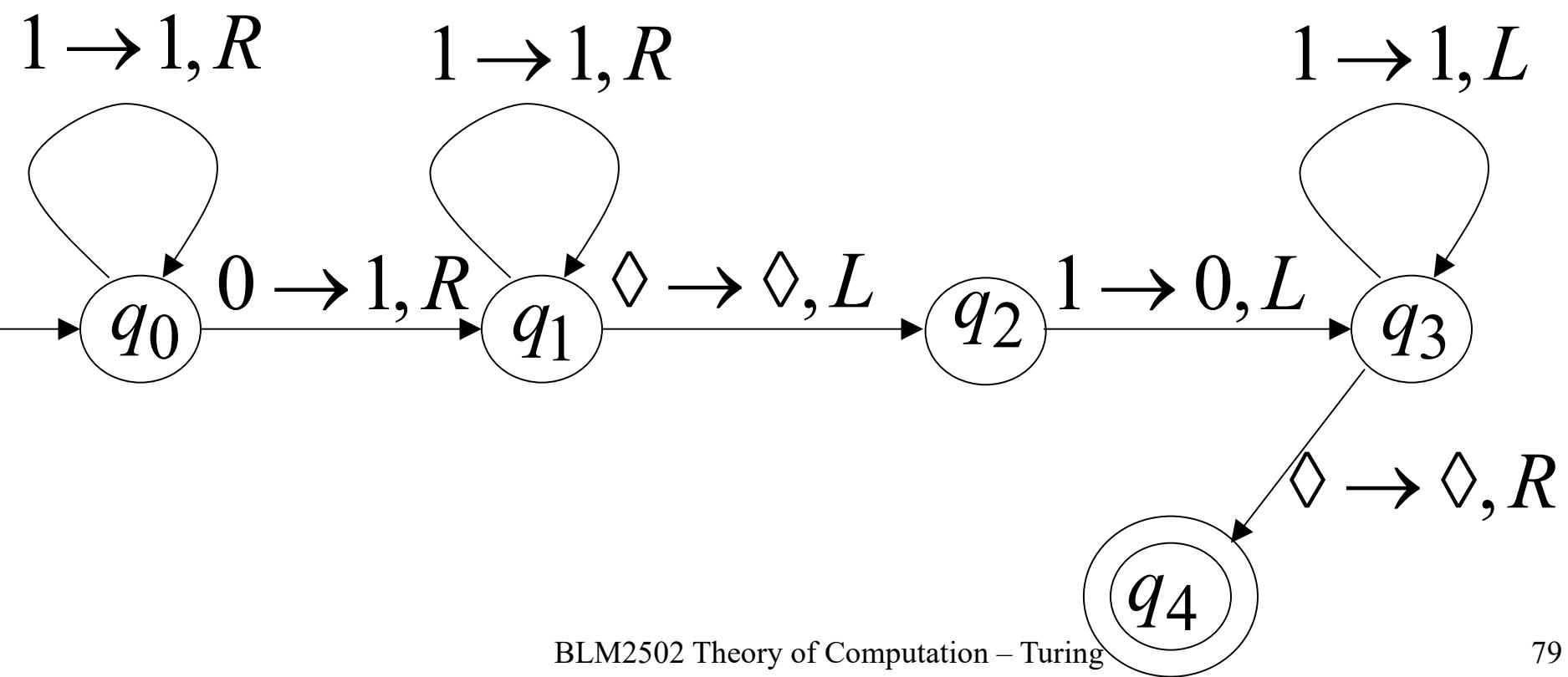
q_f final state

The 0 here helps when we use
the result for other operations

Finish



Turing machine for function $f(x, y) = x + y$

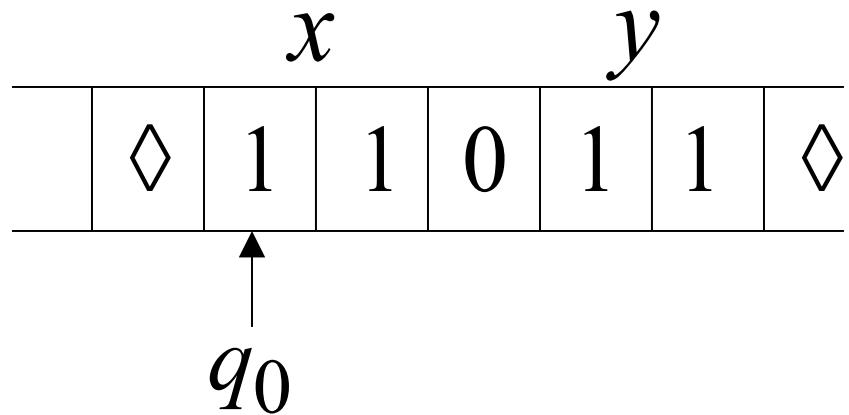


Execution Example:

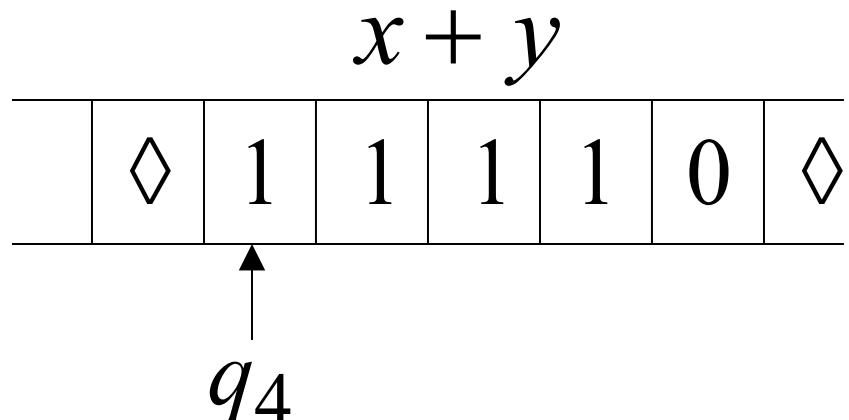
$$x = 11 \quad (=2)$$

$$y = 11 \quad (=2)$$

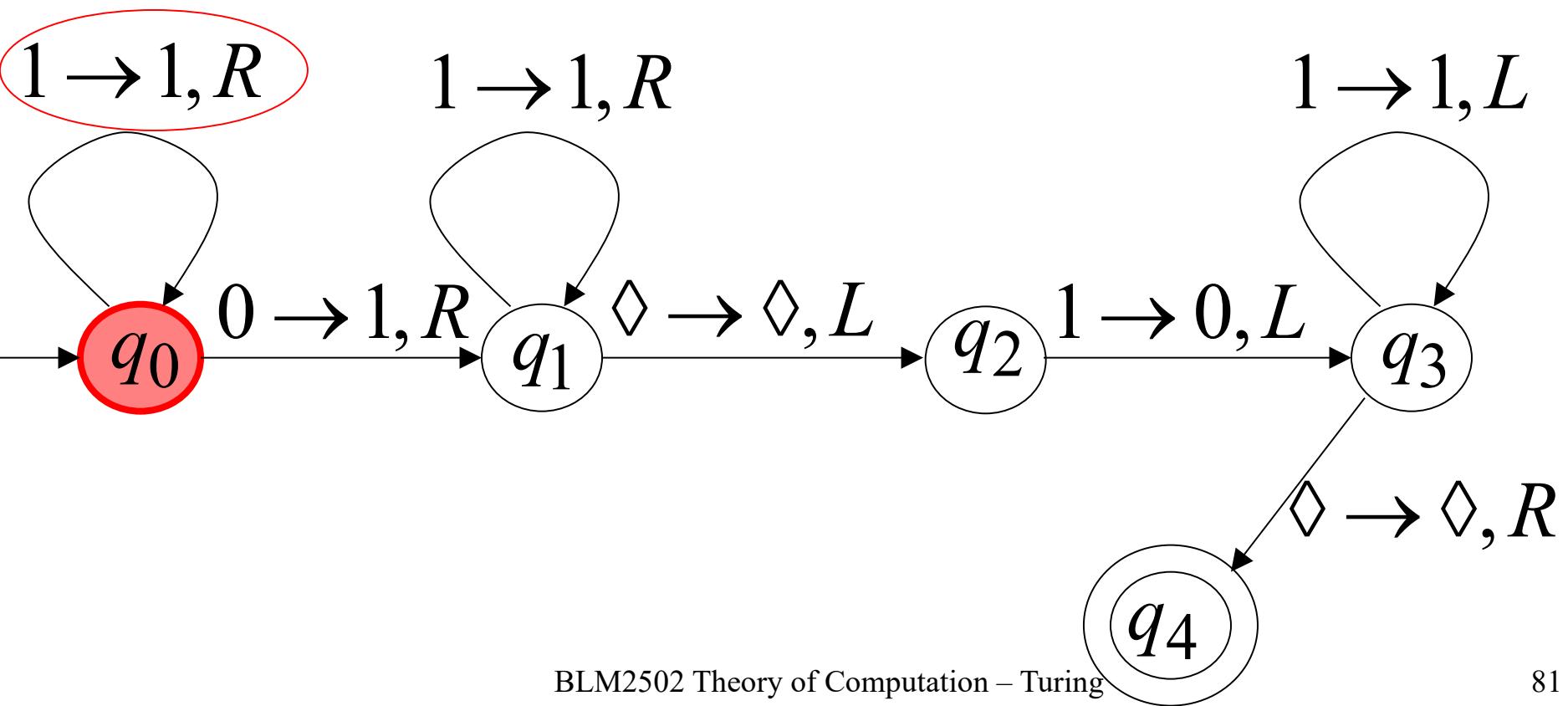
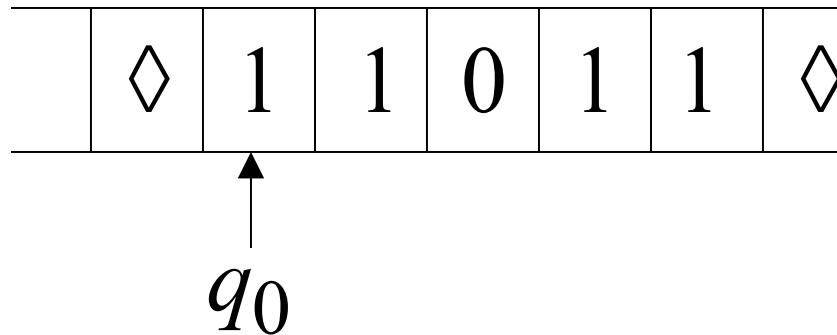
Time 0



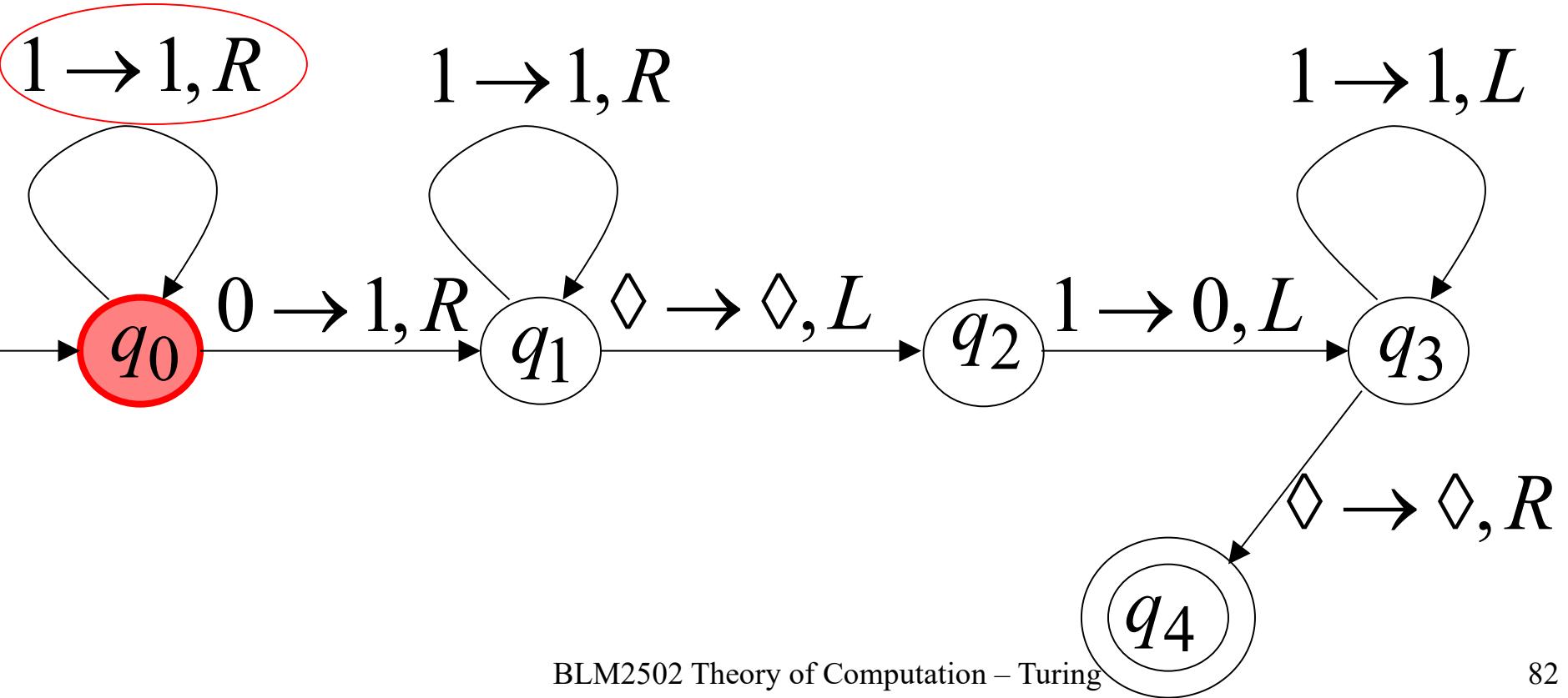
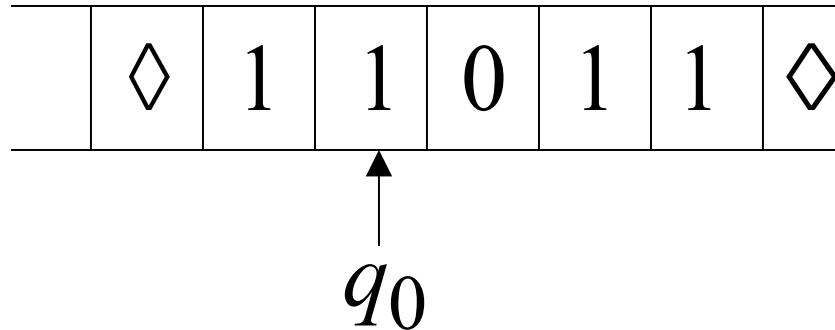
Final Result



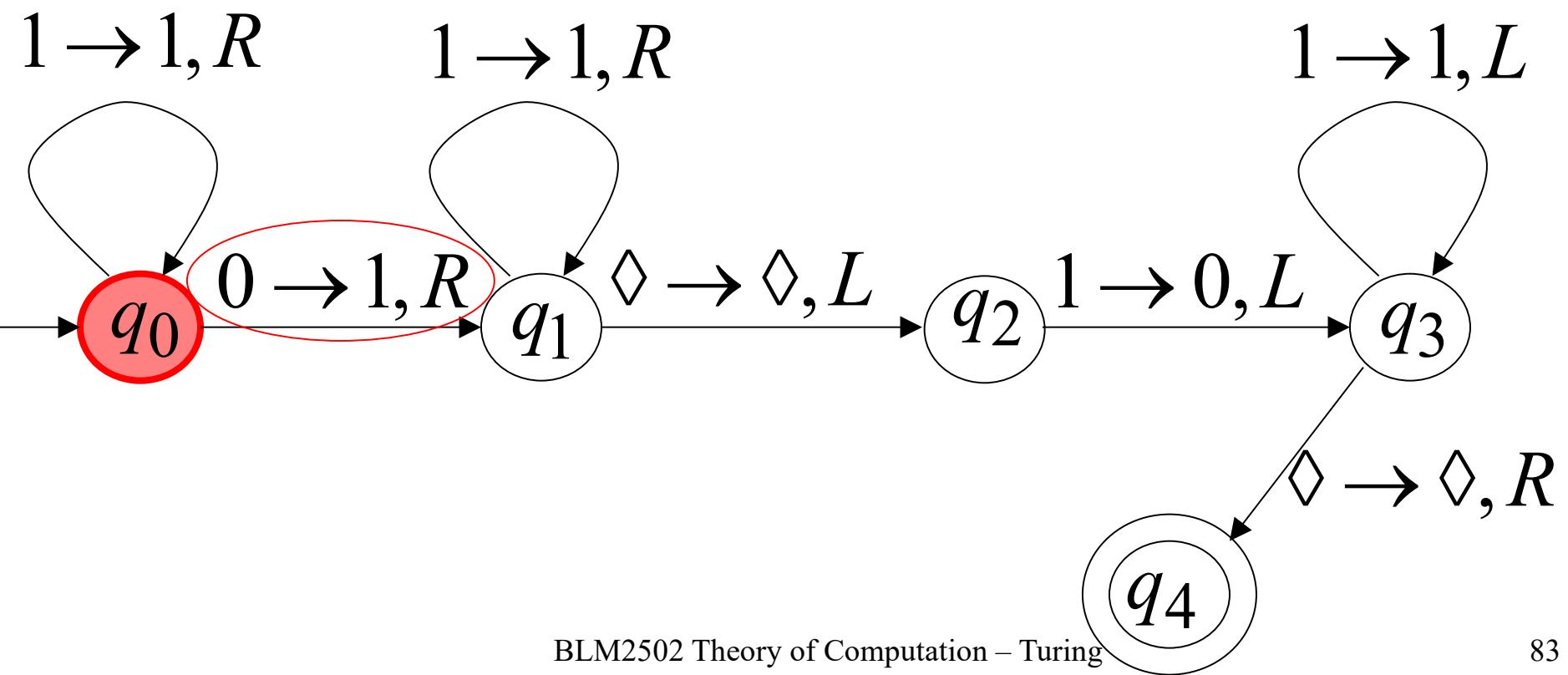
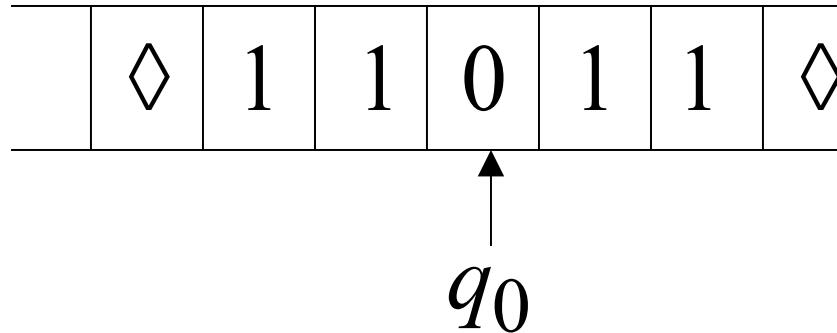
Time 0



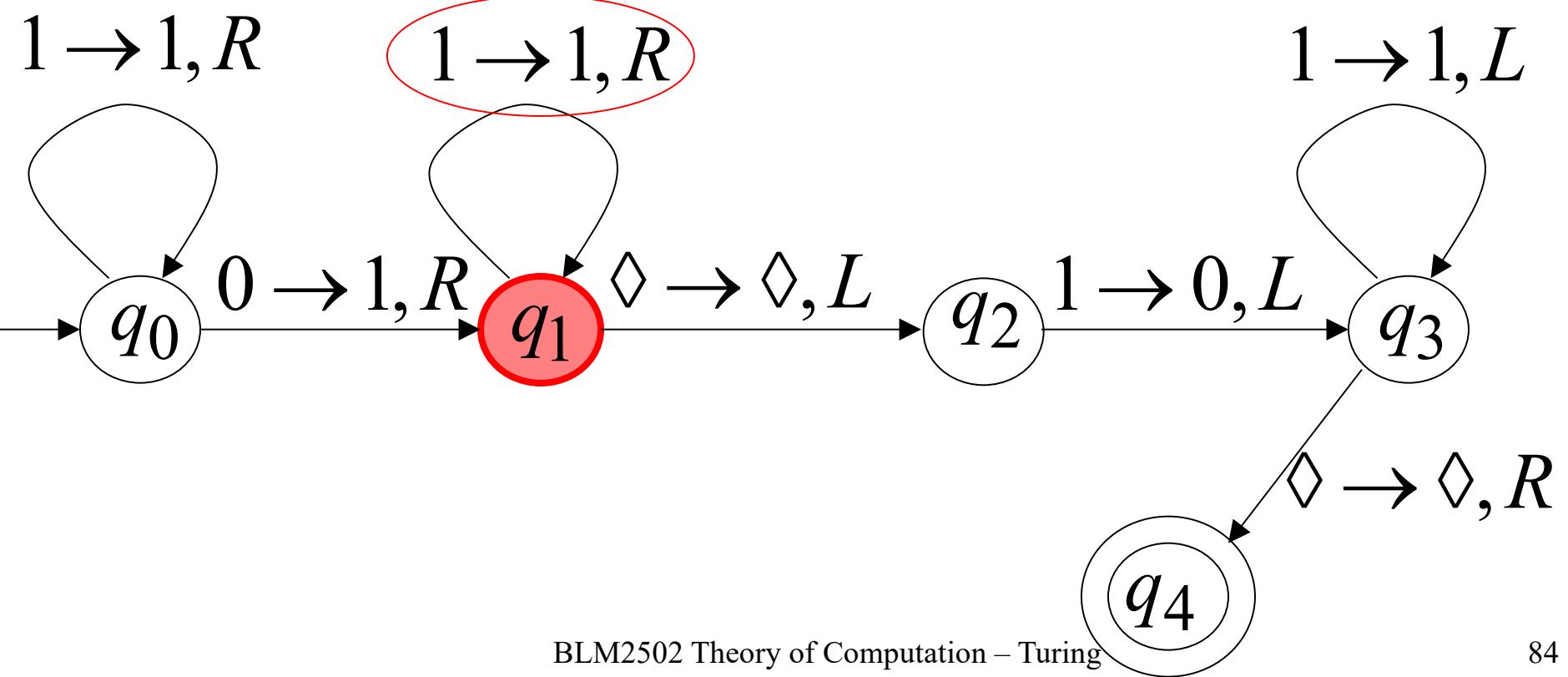
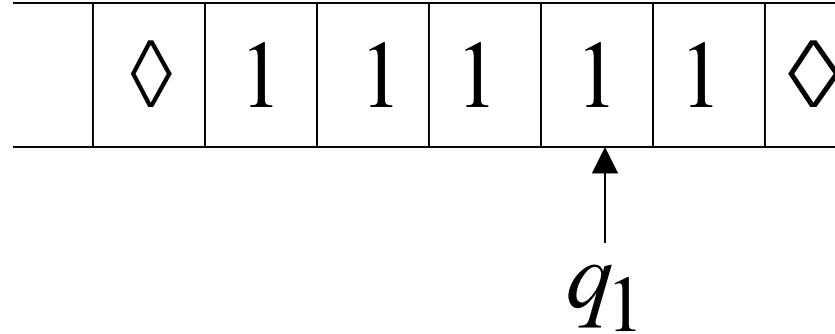
Time 1



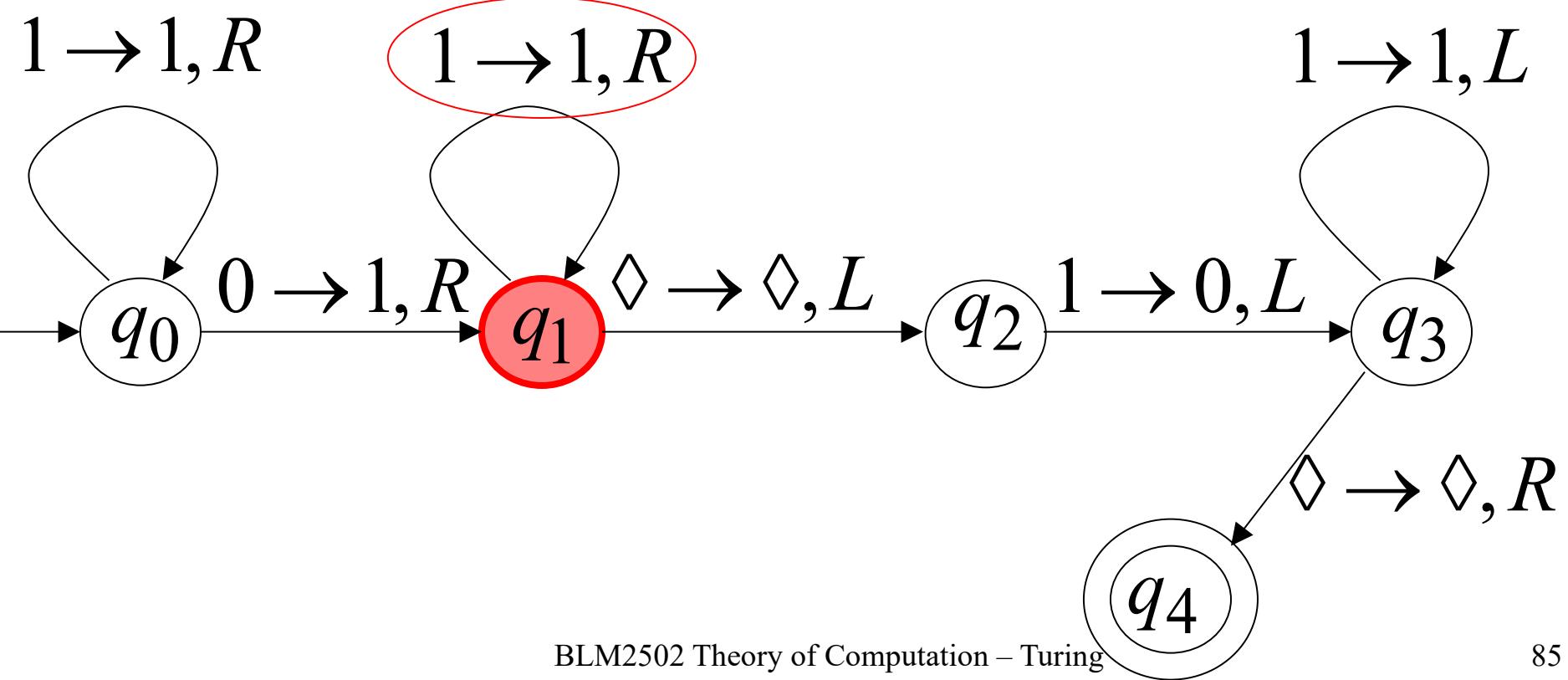
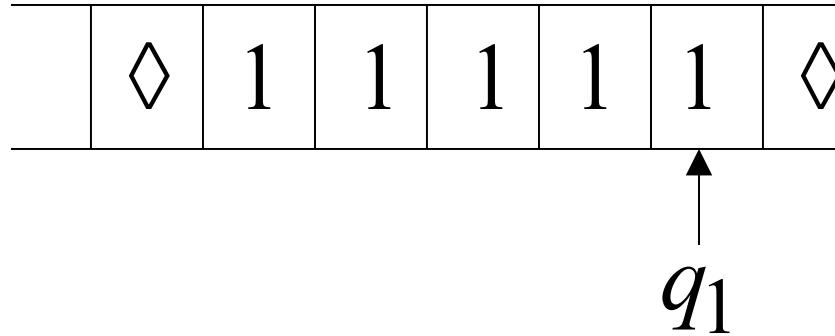
Time 2



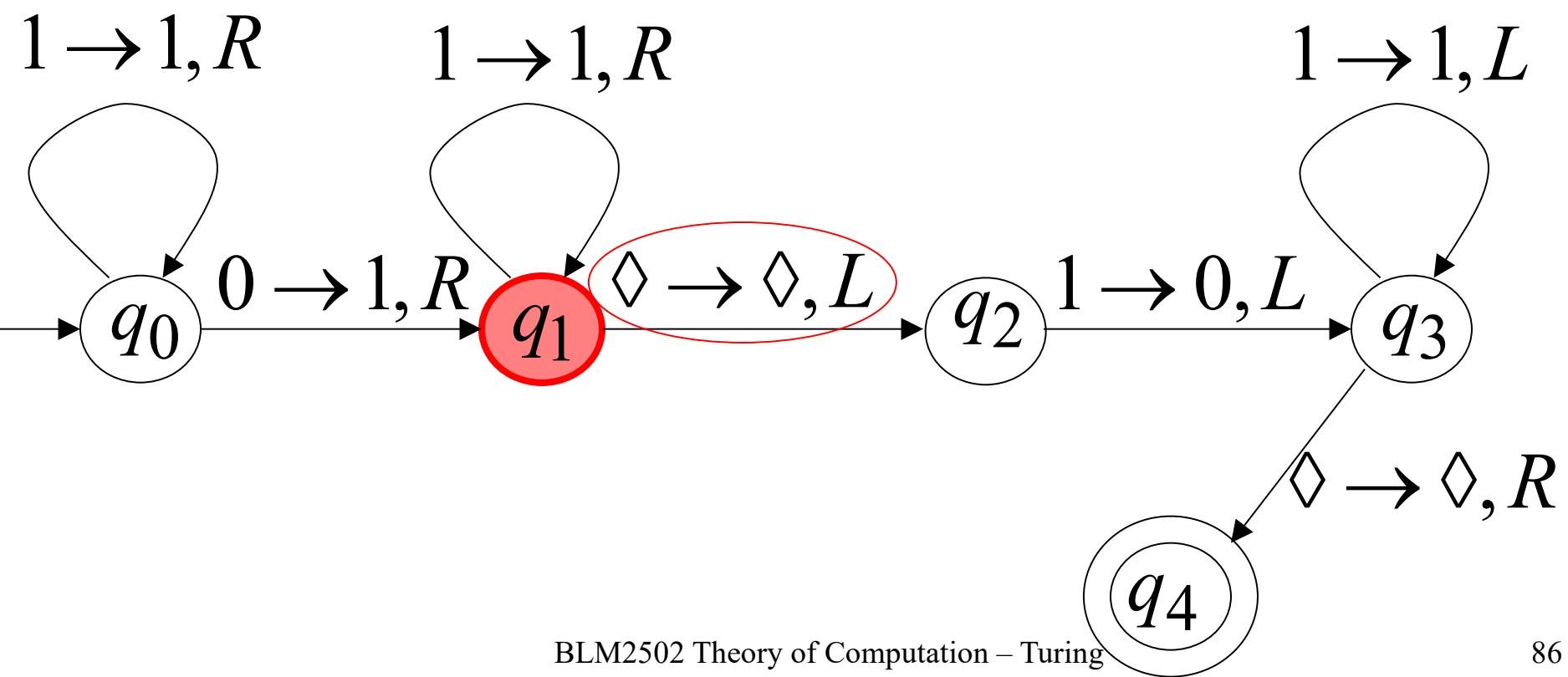
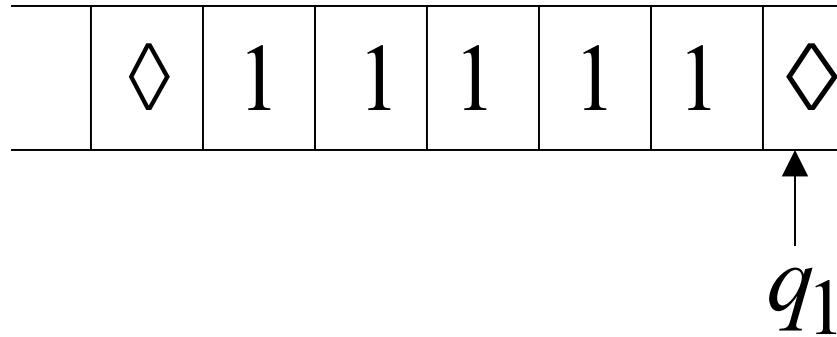
Time 3



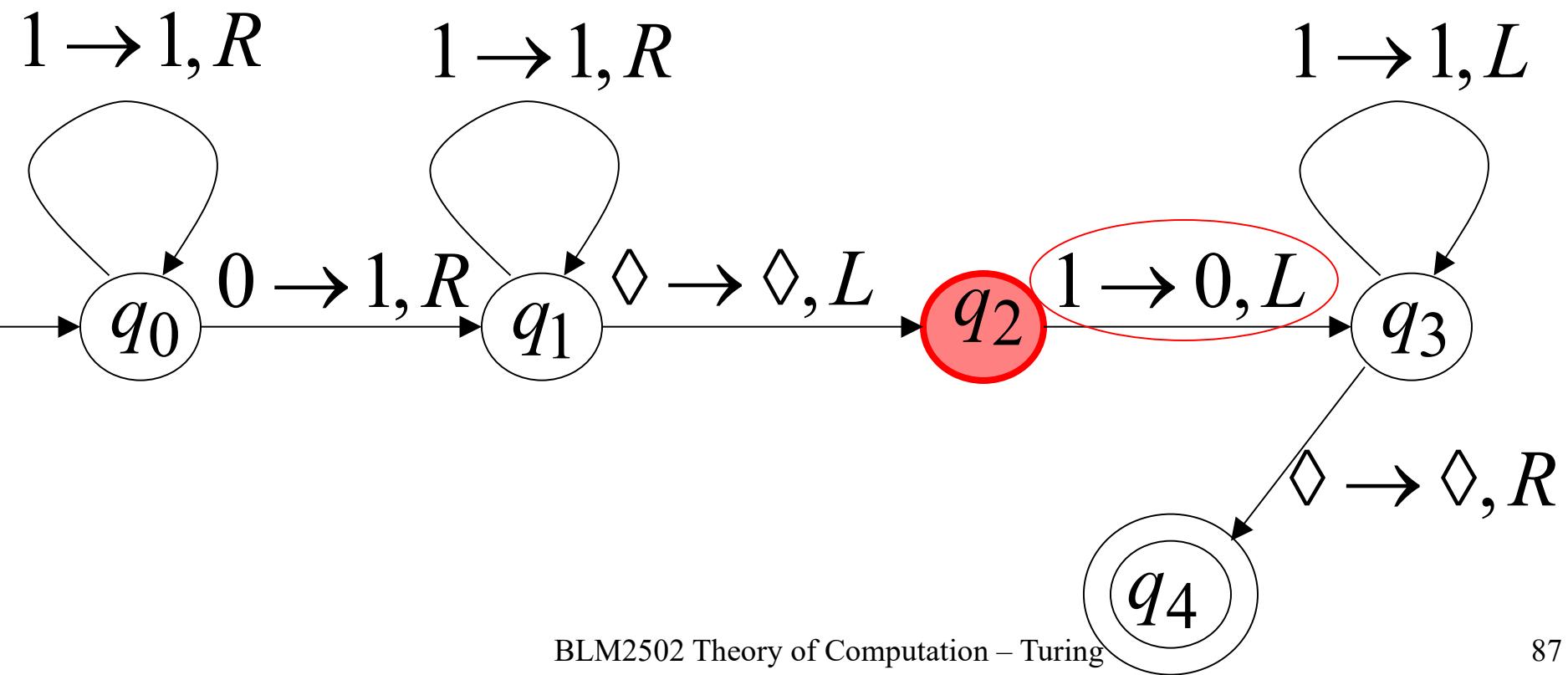
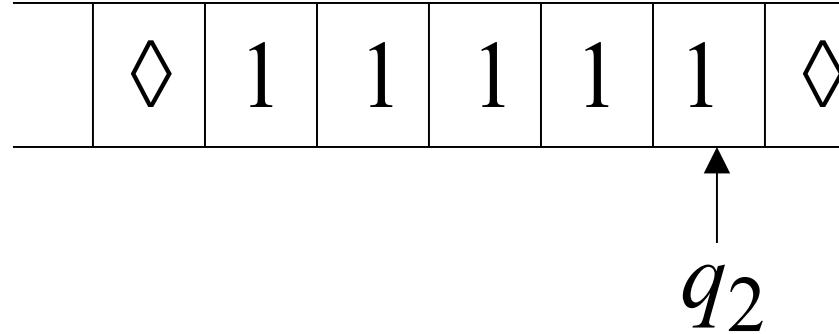
Time 4



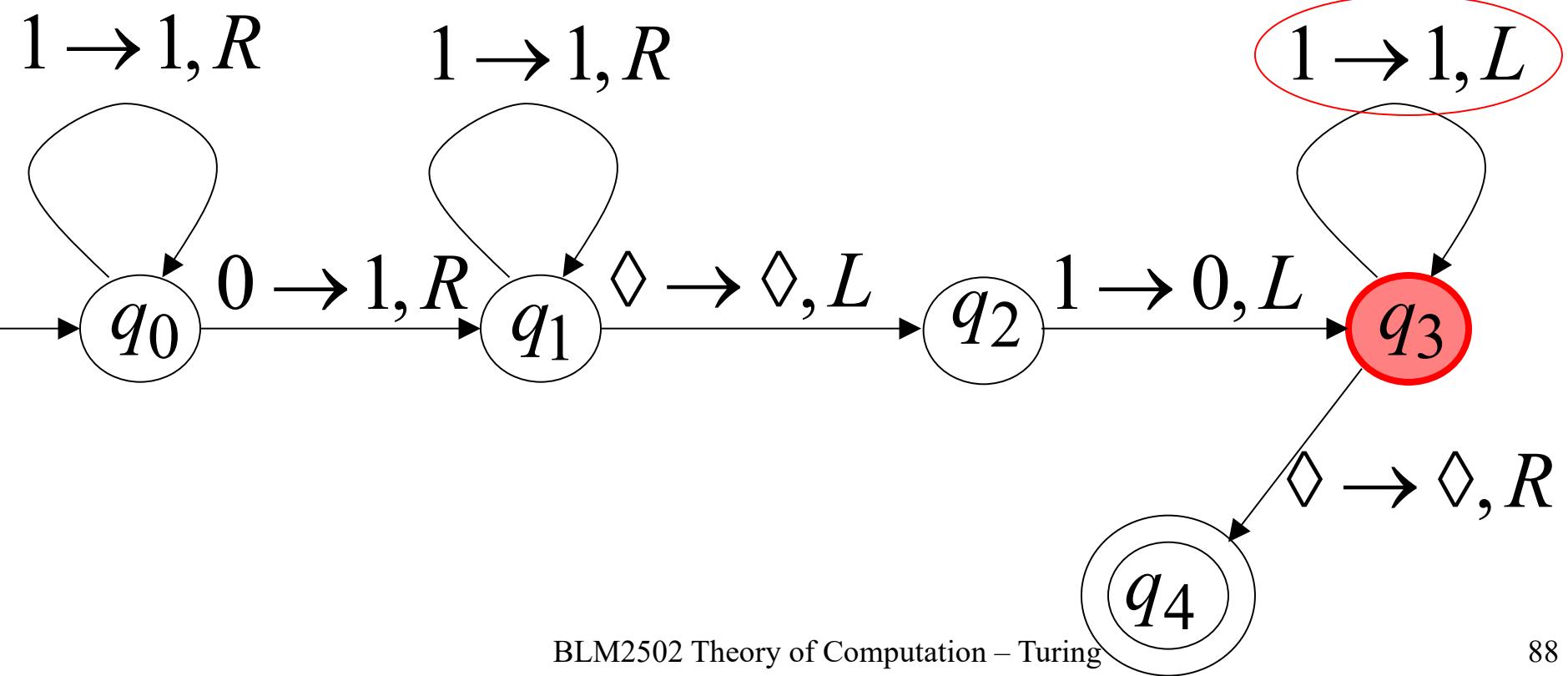
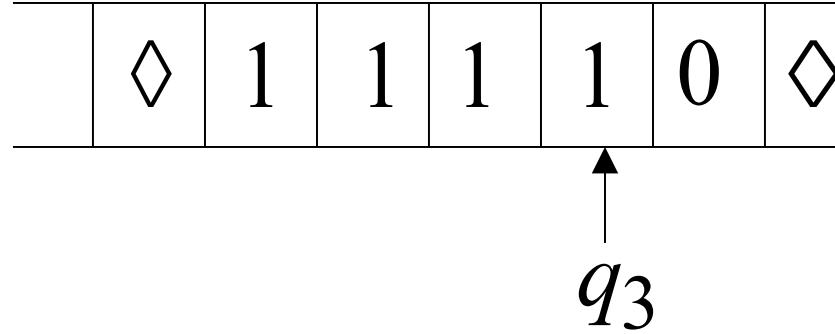
Time 5



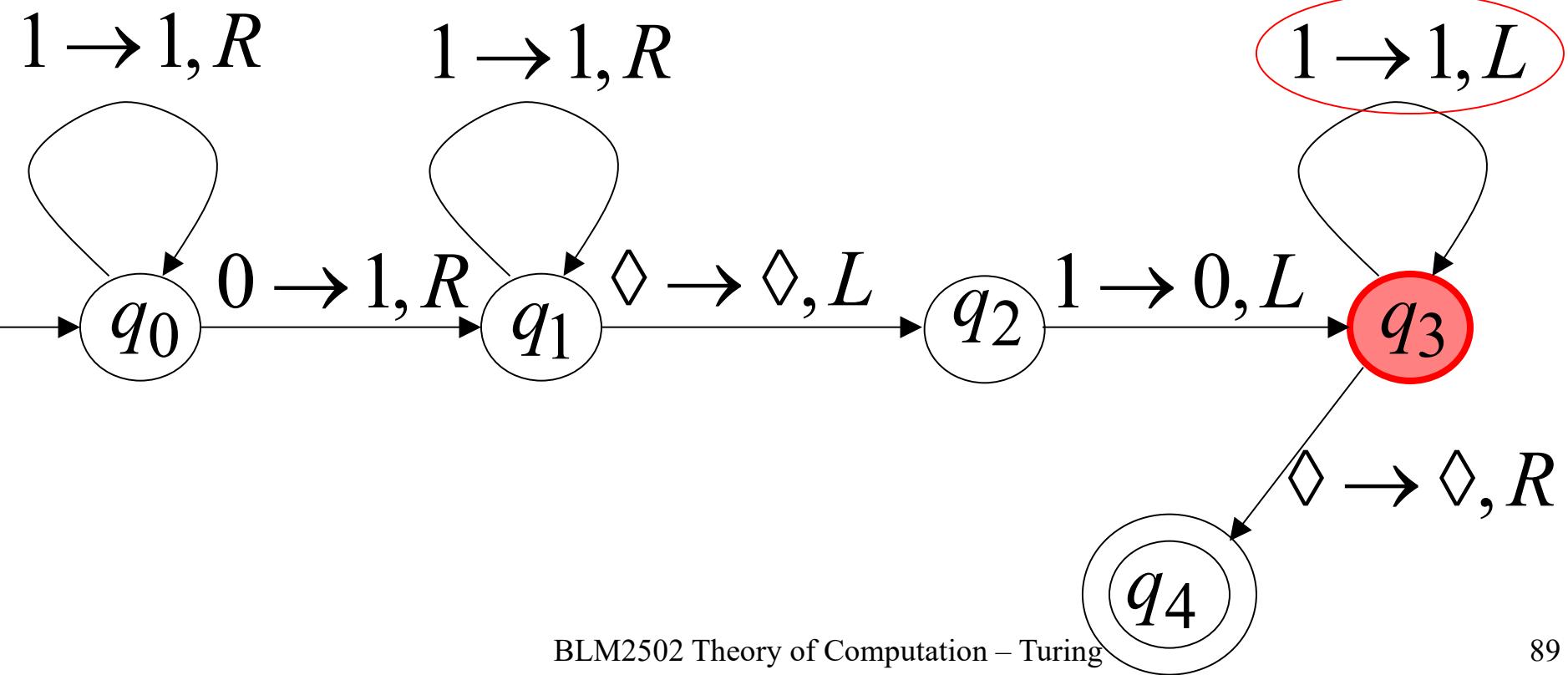
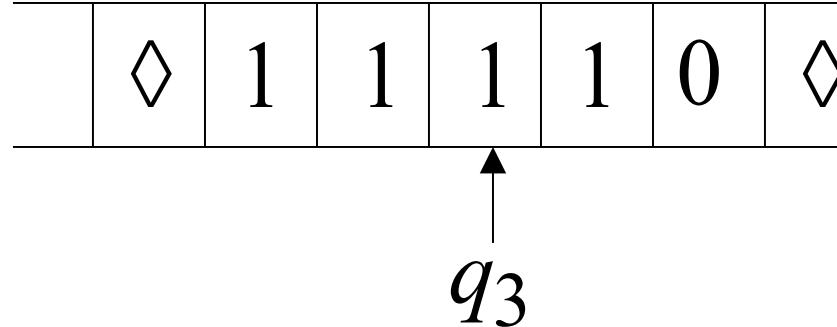
Time 6



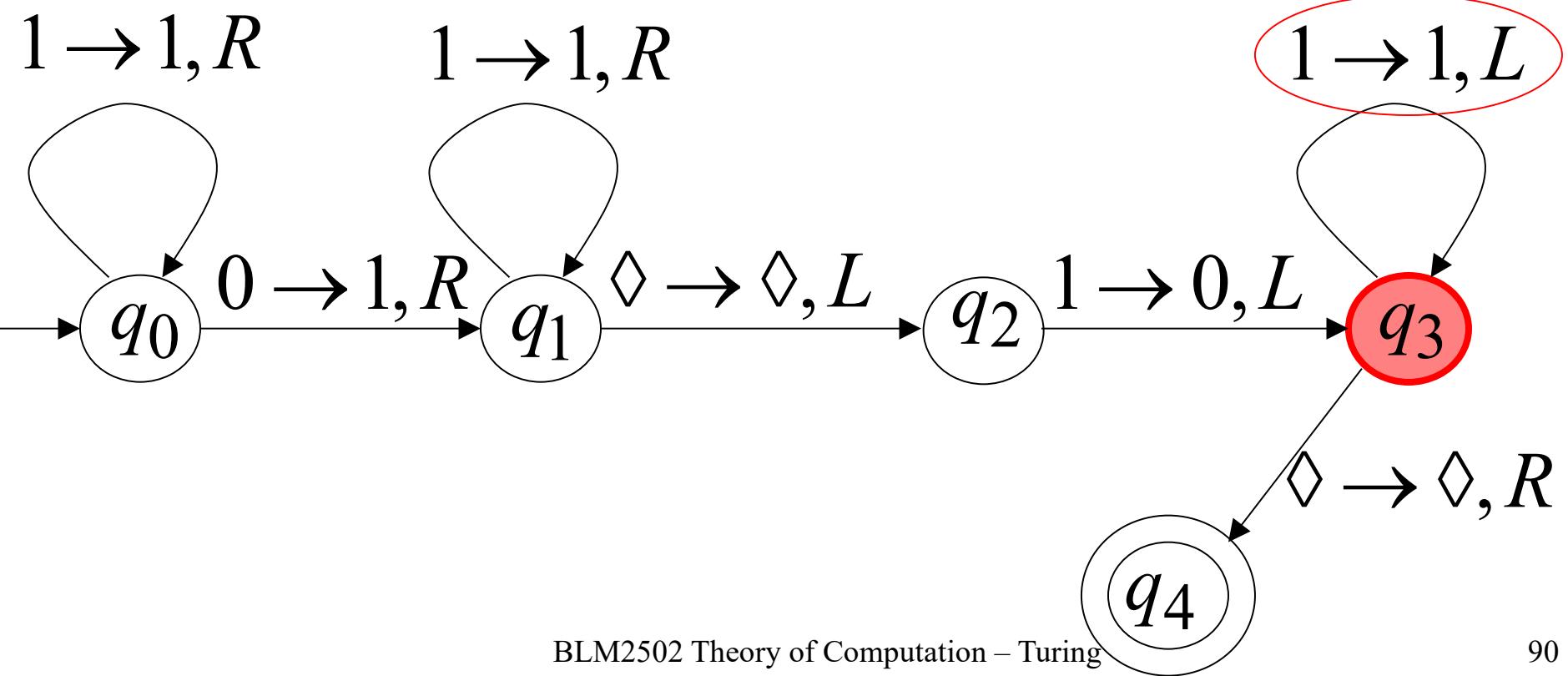
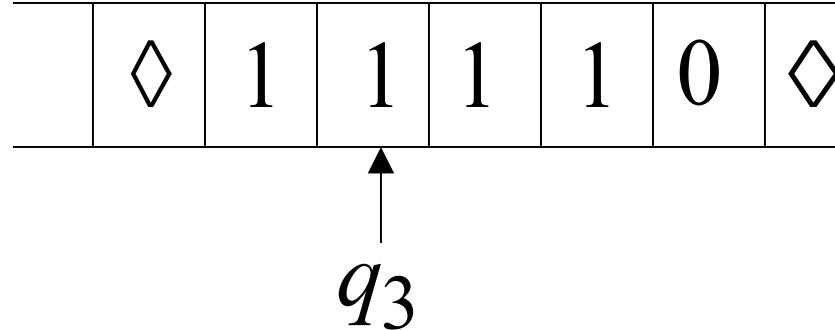
Time 7



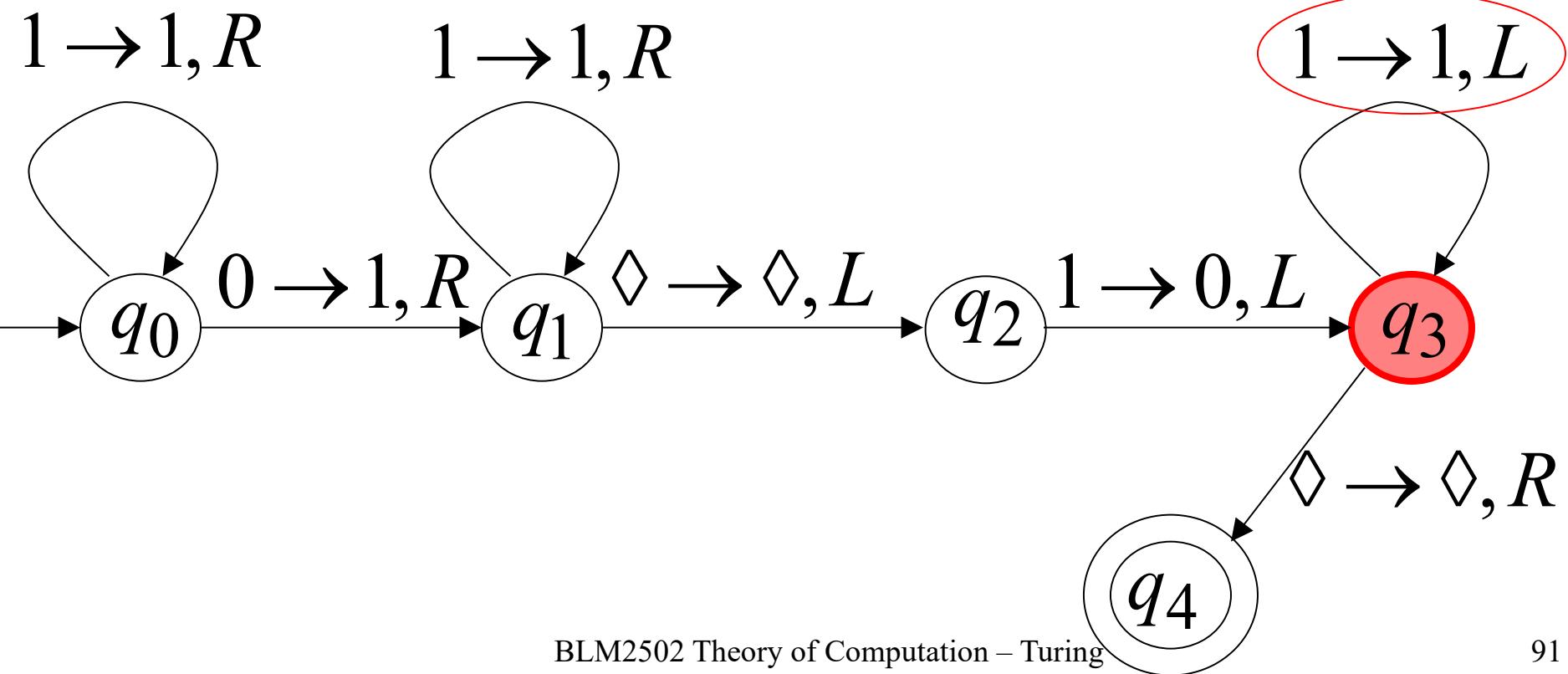
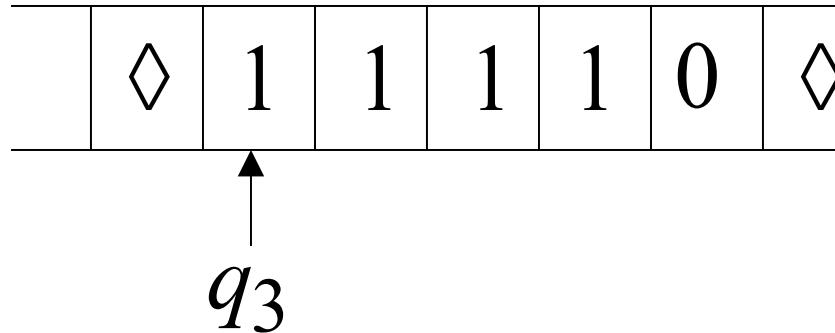
Time 8



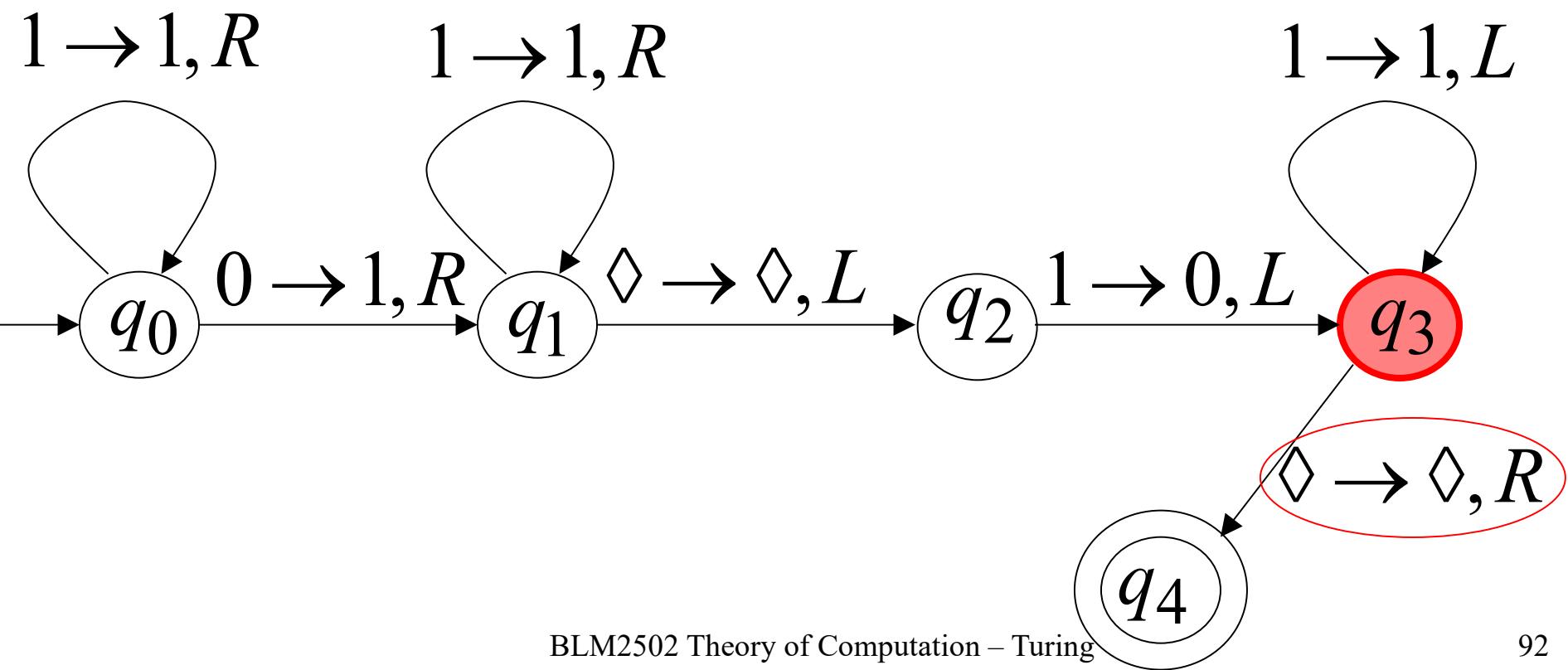
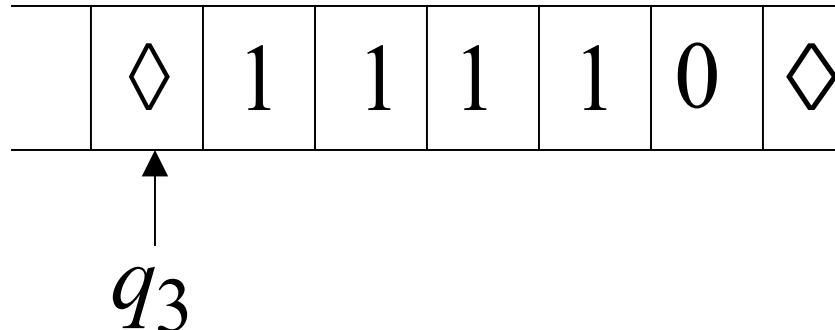
Time 9



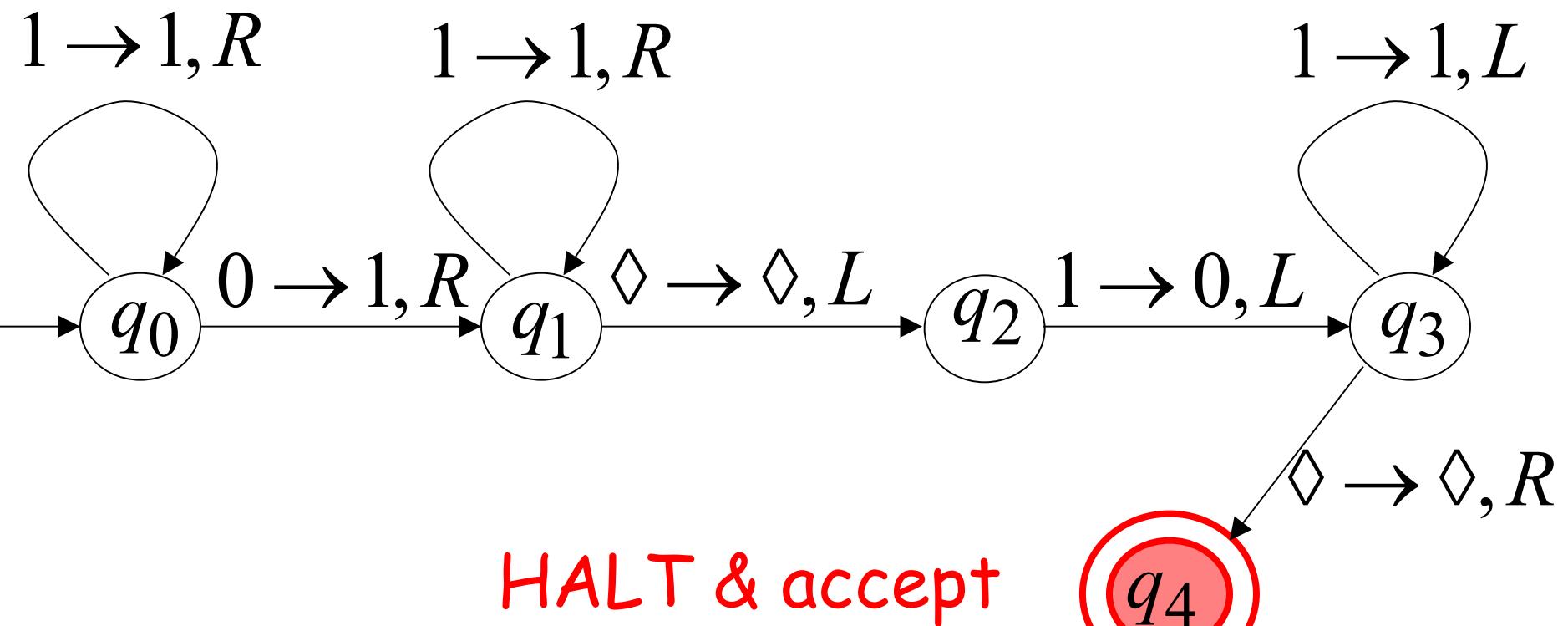
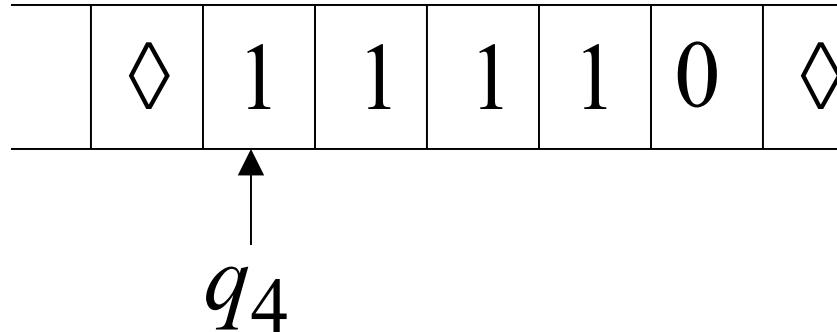
Time 10



Time 11



Time 12



Another Example

The function $f(x) = 2x$ is computable

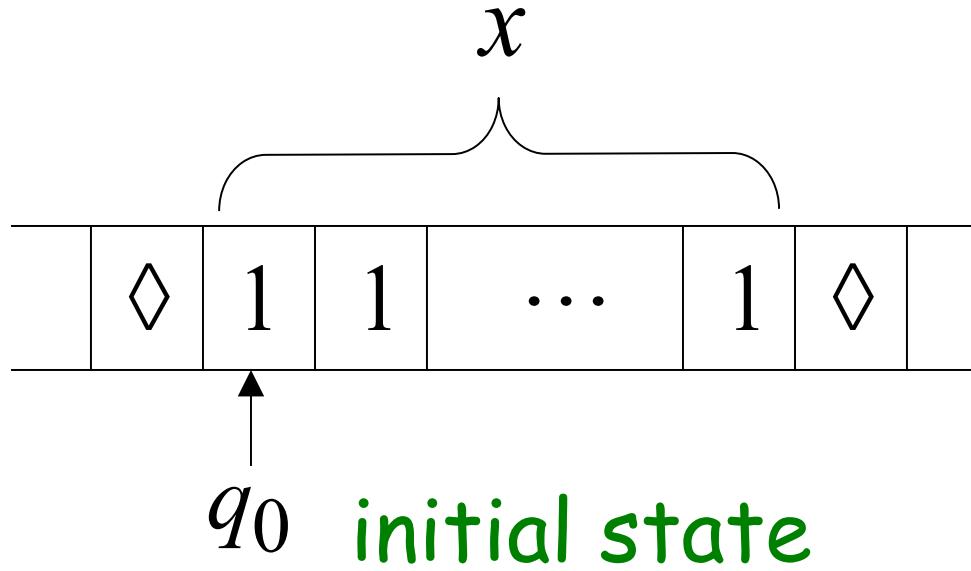
x is integer

Turing Machine:

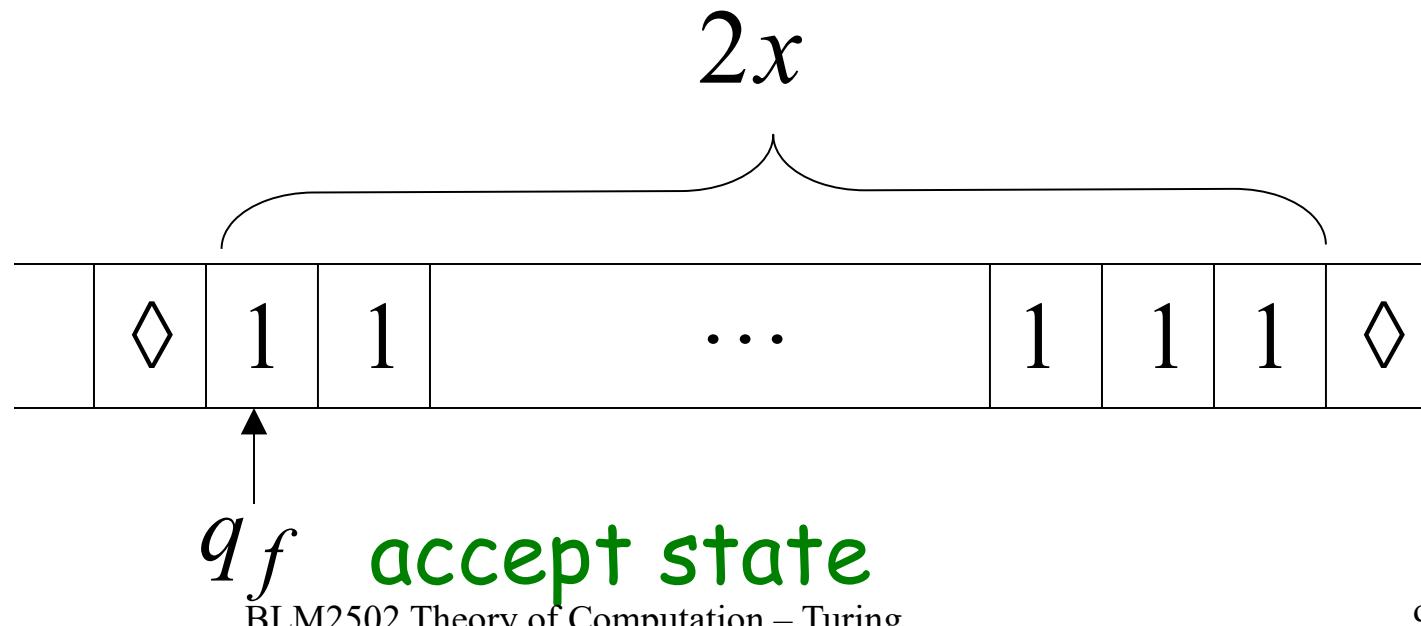
Input string: x unary

Output string: xx unary

Start



Finish

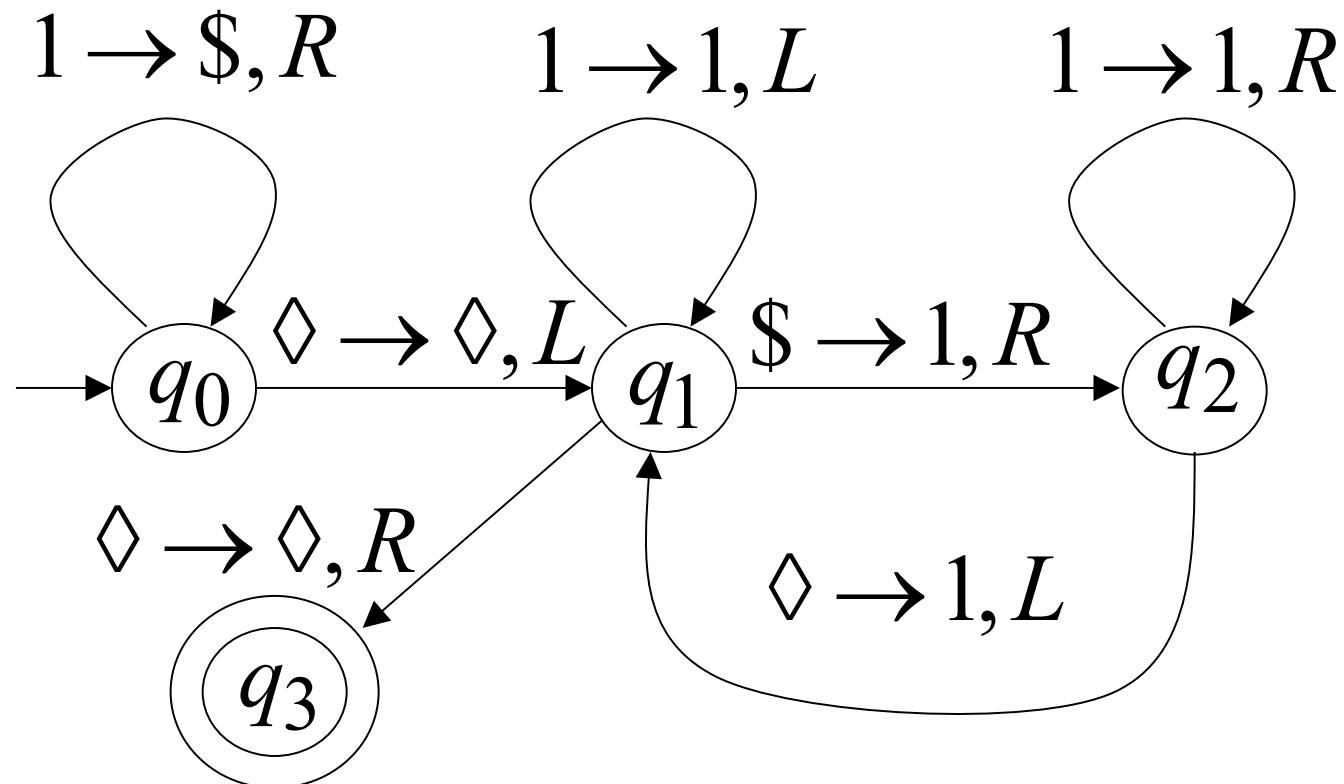


Turing Machine Pseudocode for $f(x) = 2x$

- Replace every 1 with \$
- Repeat:
 - Find rightmost \$, replace it with 1
 - Go to right end, insert 1

Until no more \$ remain

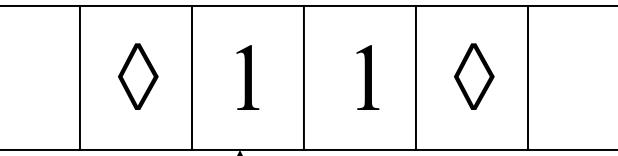
Turing Machine for $f(x) = 2x$



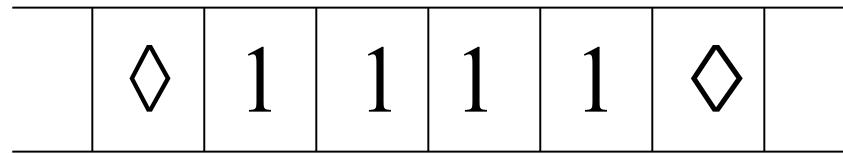
Example

Start

Finish

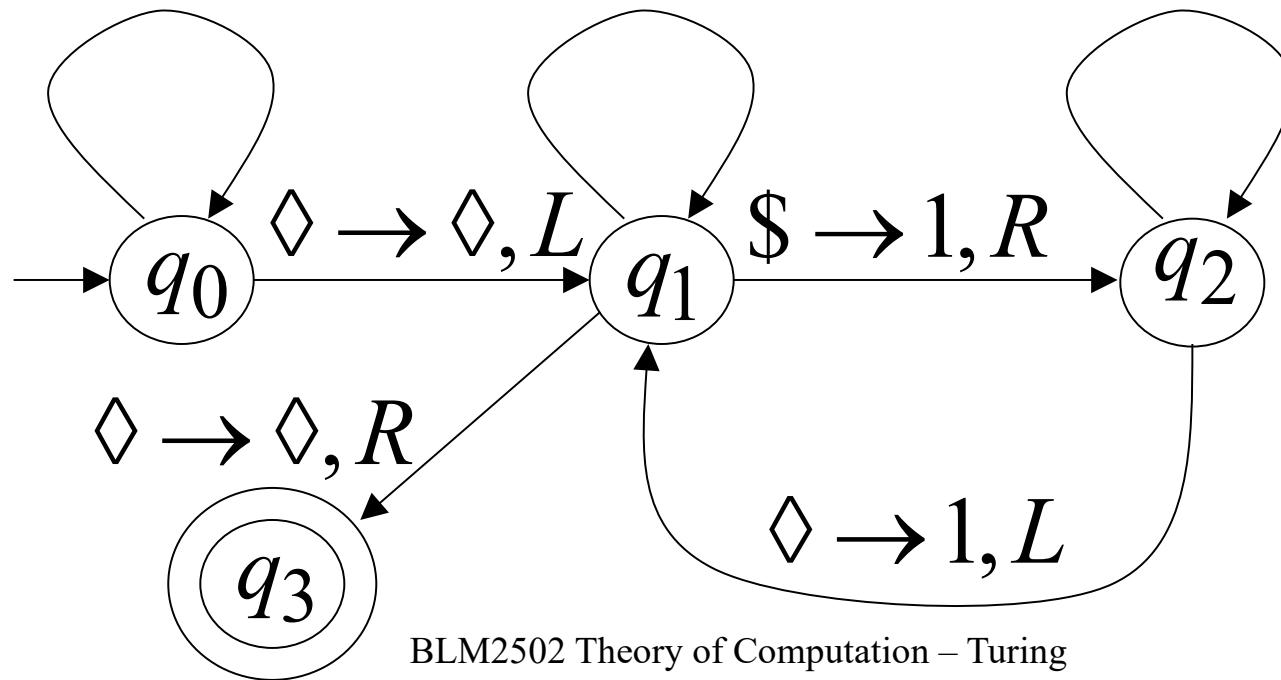


q_0



q_3

$1 \rightarrow \$, R$ $1 \rightarrow 1, L$ $1 \rightarrow 1, R$



Another Example

The function
is computable

$$f(x, y) = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{if } x \leq y \end{cases}$$

Input: $x0y$

Output: 1 or 0

Turing Machine Pseudocode:

- Repeat
 - Match a 1 from x with a 1 from y
 - Until all of x or y is matched
 - If a 1 from x is not matched
 - erase tape, write 1 $(x > y)$
 - else
 - erase tape, write 0 $(x \leq y)$

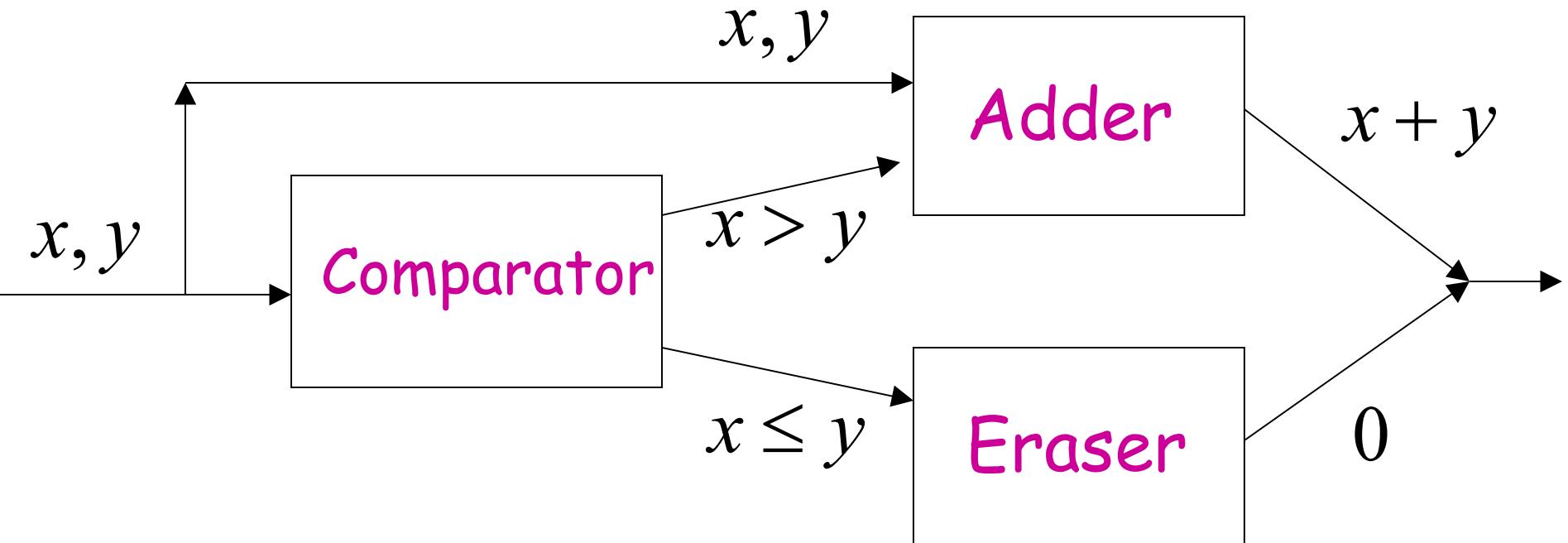
Combining Turing Machines

Block Diagram



Example:

$$f(x, y) = \begin{cases} x + y & \text{if } x > y \\ 0 & \text{if } x \leq y \end{cases}$$



Turing's Thesis

Turing's thesis (1930):

Any computation carried out
by mechanical means
can be performed by a Turing Machine

Algorithm:

An algorithm for a problem is a Turing Machine which solves the problem

The algorithm describes the steps of the mechanical means

This is easily translated to computation steps of a Turing machine

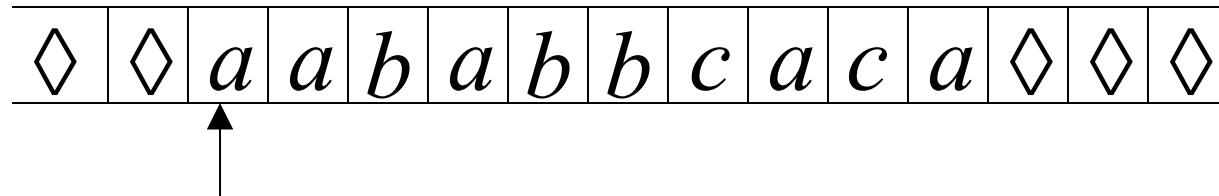
When we say: There exists an algorithm

We mean: There exists a Turing Machine
that executes the algorithm

Variations of the Turing Machine

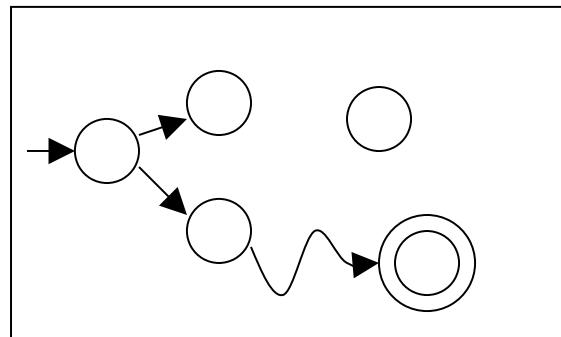
The Standard Model

Infinite Tape



Read-Write Head (Left or Right)

Control Unit



Deterministic

Variations of the Standard Model

Turing machines with:

- Stay-Option
- Semi-Infinite Tape
- Multitape
- Multidimensional
- Nondeterministic

Different Turing Machine **Classes**

Same Power of two machine classes:

both classes accept the
same set of languages

We will prove:

each new class has the same power
with Standard Turing Machine

(accept Turing-Recognizable Languages)

Same Power of two classes means:

for any machine M_1 of first class

there is a machine M_2 of second class

such that: $L(M_1) = L(M_2)$

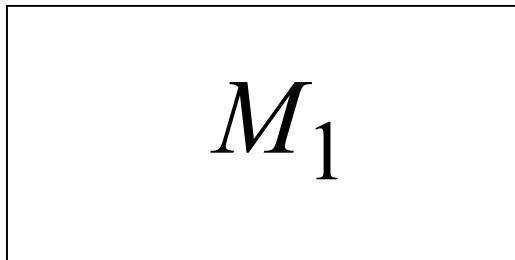
and vice-versa

Simulation: A technique to prove same power.

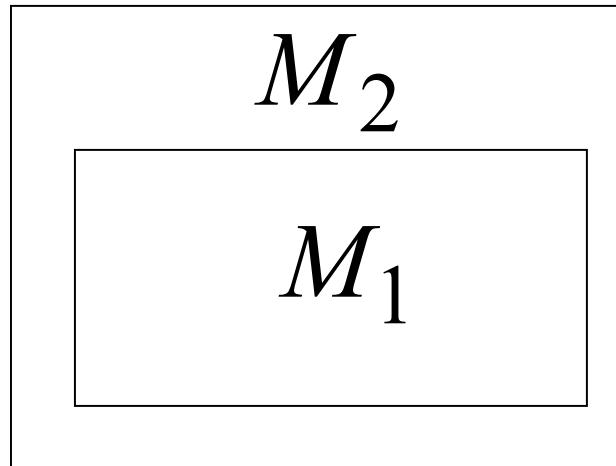
Simulate the machine of one class
with a machine of the other class

First Class

Original Machine

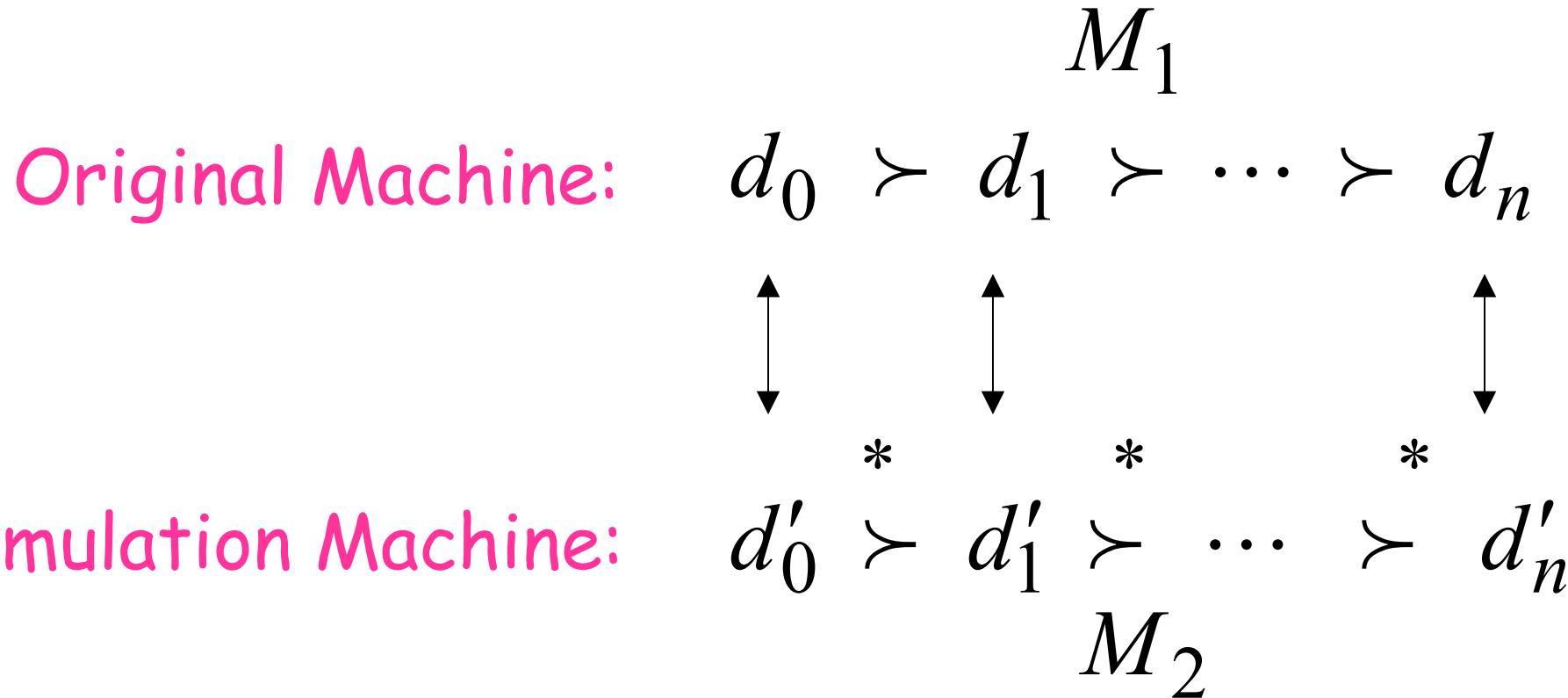


Second Class
Simulation Machine



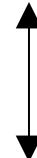
simulates M_1

Configurations in the Original Machine M_1
have corresponding configurations
in the Simulation Machine M_2



Accepting Configuration

Original Machine:

 d_f 

Simulation Machine:

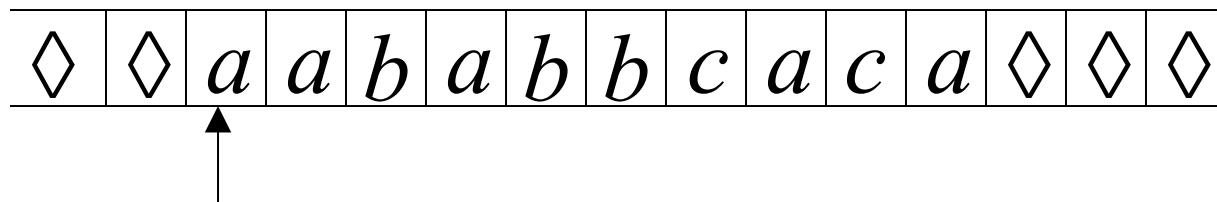
 d'_f

the Simulation Machine
and the Original Machine
accept the same strings

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

Turing Machines with Stay-Option

The head can stay in the same position



Left, Right, Stay

L,R,S: possible head moves

Example:

Time 1

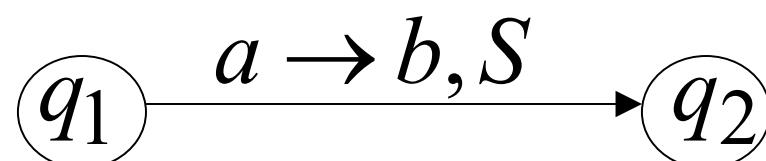
◊	◊	a	a	b	a	b	b	c	a	c	a	◊	◊	◊
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

q_1

Time 2

◊	◊	b	a	b	a	b	b	c	a	c	a	◊	◊	◊
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

q_2



Theorem: Stay-Option machines

have the same power with

Standard Turing machines

Proof: 1. Stay-Option Machines

simulate Standard Turing machines

2. Standard Turing machines

simulate Stay-Option machines

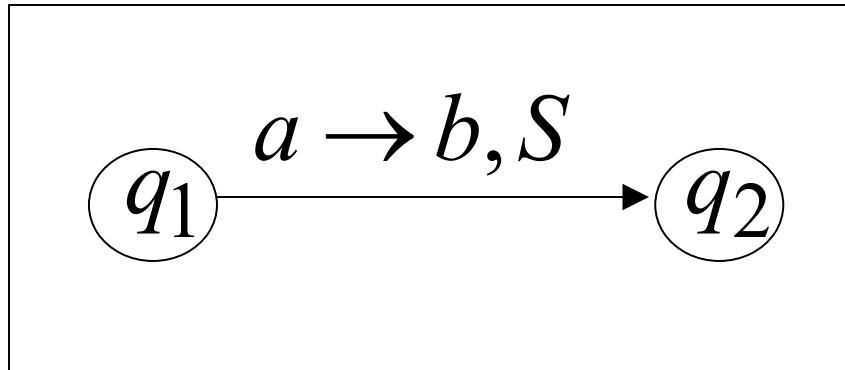
1. Stay-Option Machines simulate Standard Turing machines

Trivial: any standard Turing machine
is also a Stay-Option machine

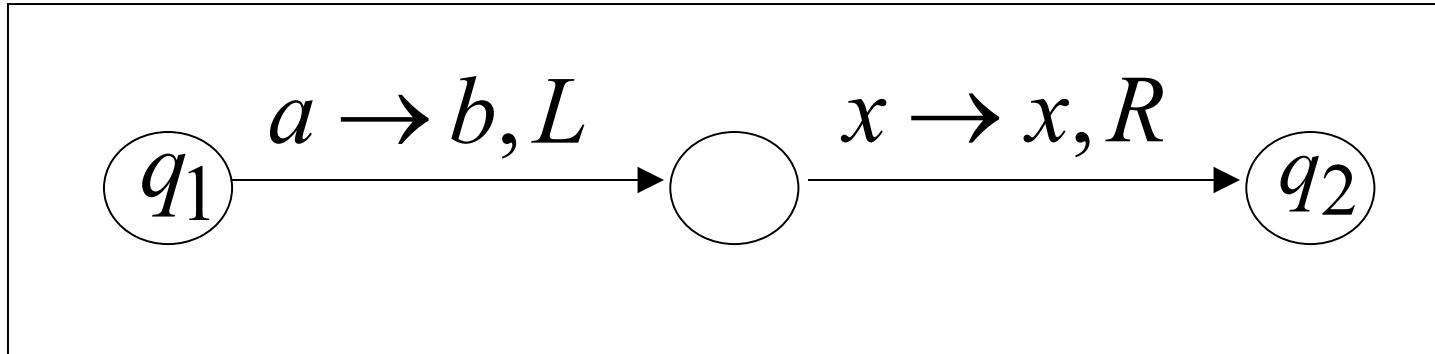
2. Standard Turing machines simulate Stay-Option machines

We need to simulate the **stay** head option
with two head moves, one **left** and one **right**

Stay-Option Machine



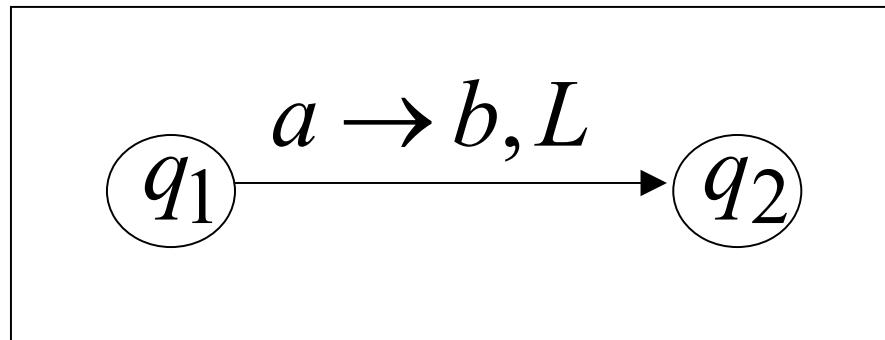
Simulation in Standard Machine



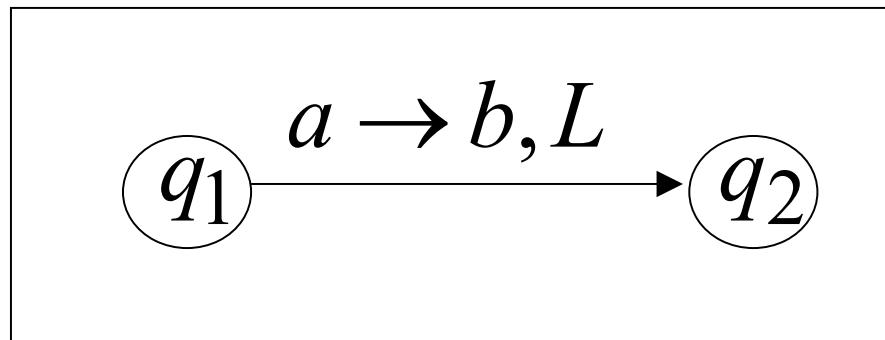
For every possible tape symbol x

For other transitions nothing changes

Stay-Option Machine

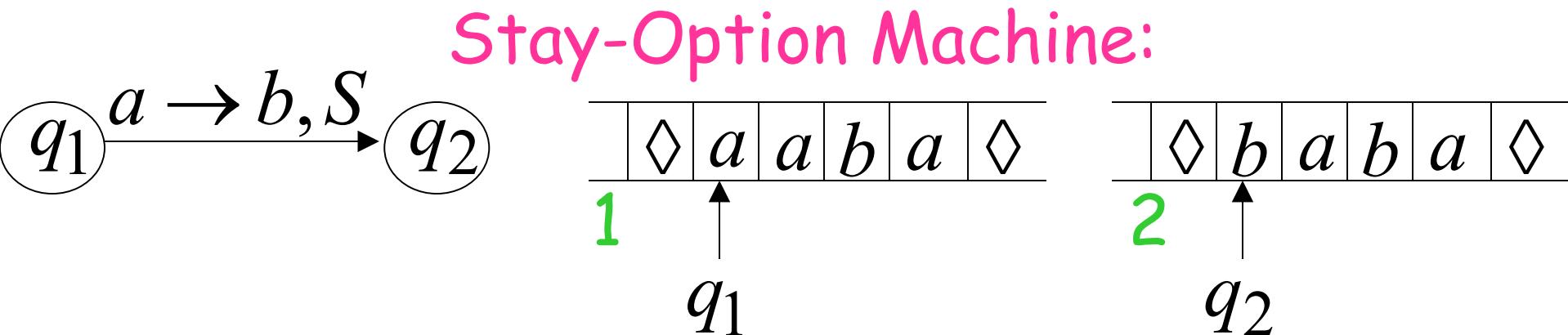


Simulation in Standard Machine

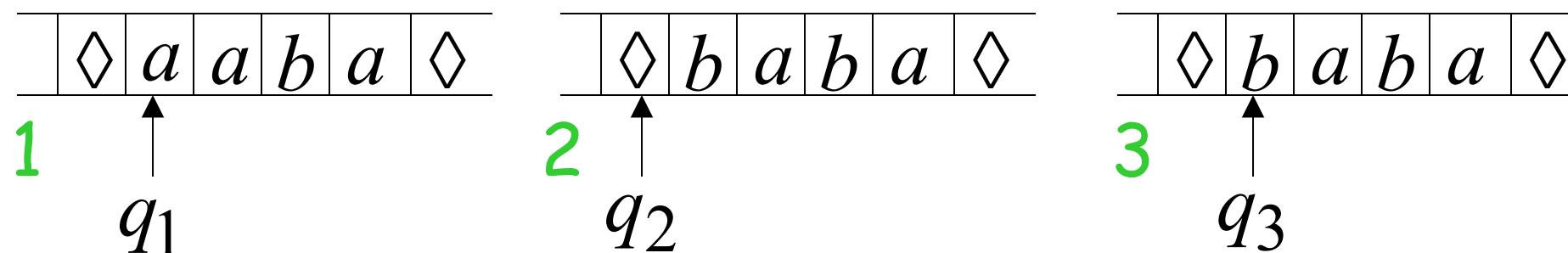


Similar for Right moves

example of simulation



Simulation in Standard Machine:

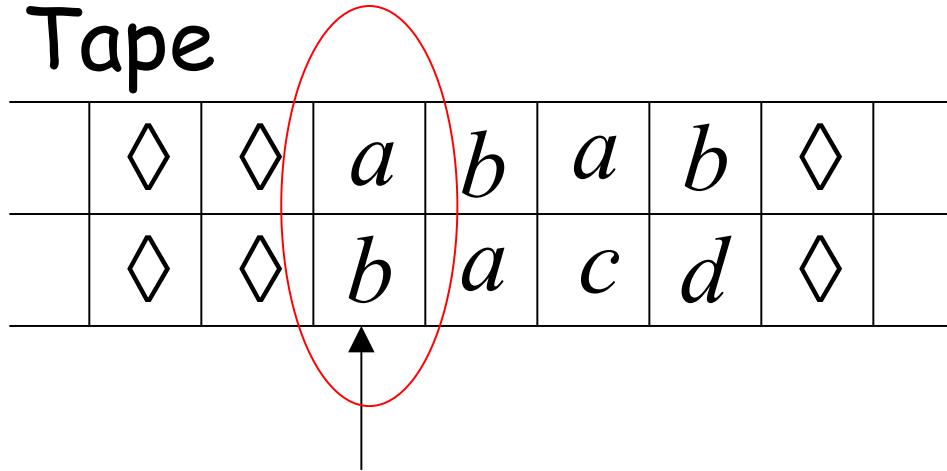


END OF PROOF

A useful trick: Multiple Track Tape

helps for more complicated simulations

One Tape



track 1
track 2

One head

One symbol (a, b)

It is a standard Turing machine, but each tape alphabet symbol describes a pair of actual useful symbols

	◊	◊	a	b	a	b	◊
	◊	◊	b	a	c	d	◊

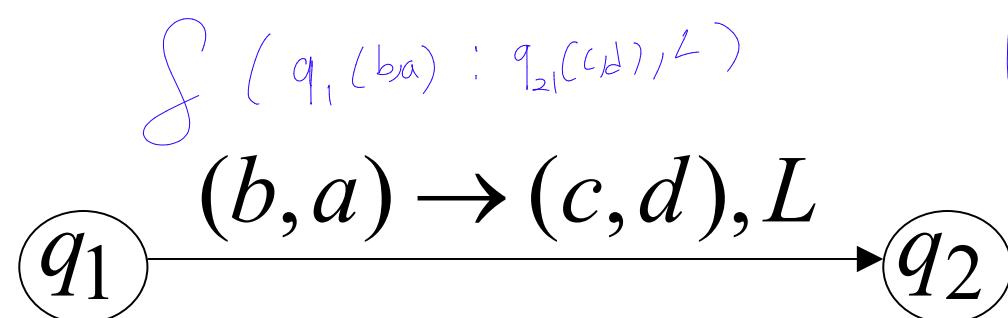
q_1

track 1
track 2

	◊	◊	a	c	a	b	◊
	◊	◊	b	d	c	d	◊

q_2

track 1
track 2

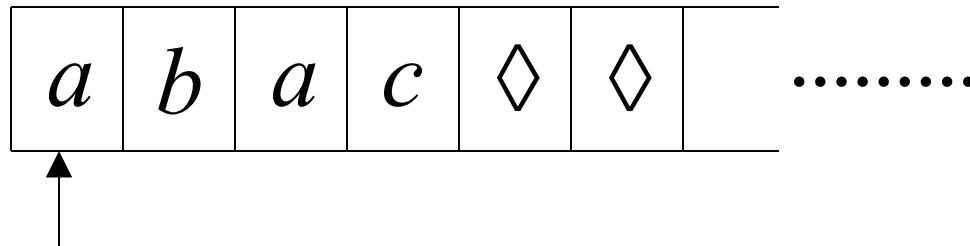


$$\sum = \{ (a, b), (c, d), (e, f), (g, h), \dots \}$$

↳ Alphabet : {a, b, c, d, e, f, g, h, ...}

Semi-Infinite Tape

The head extends infinitely only to the right



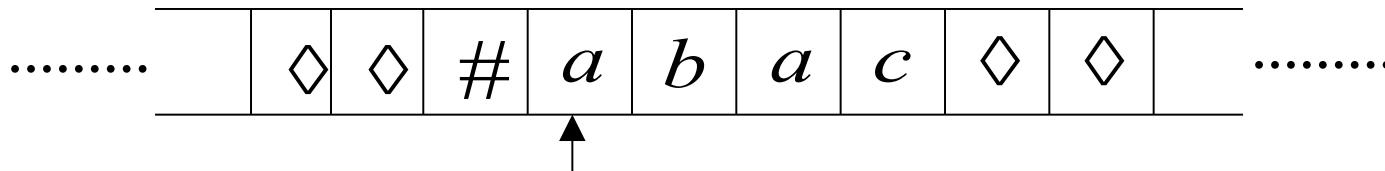
- Initial position is the leftmost cell
- When the head moves left from the border, it returns back to leftmost position

Theorem: Semi-Infinite machines
have the same power with
Standard Turing machines

Proof: 1. Standard Turing machines
simulate Semi-Infinite machines

2. Semi-Infinite Machines
simulate Standard Turing machines

1. Standard Turing machines simulate Semi-Infinite machines:



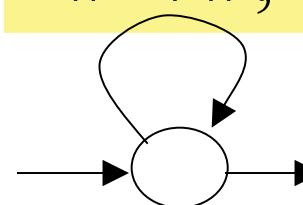
Standard Turing Machine

Semi-Infinite machine modifications

a. insert special symbol #
at left of input string

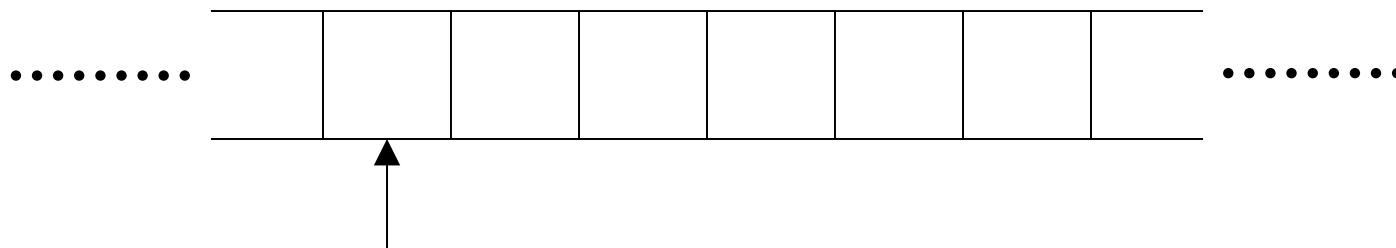
b. Add a self-loop
to every state
(except states with no
outgoing transitions)

→ #, R

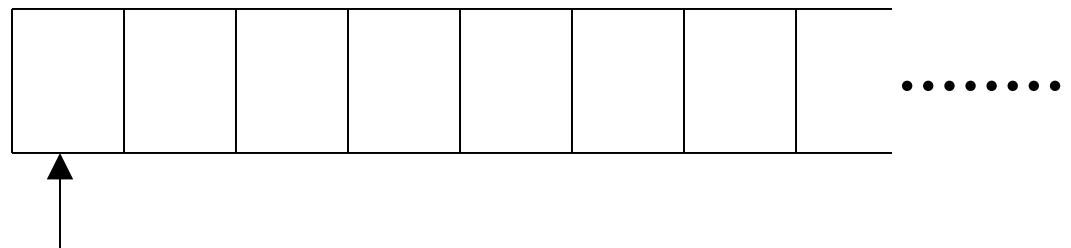


2. Semi-Infinite tape machines simulate Standard Turing machines:

Standard machine

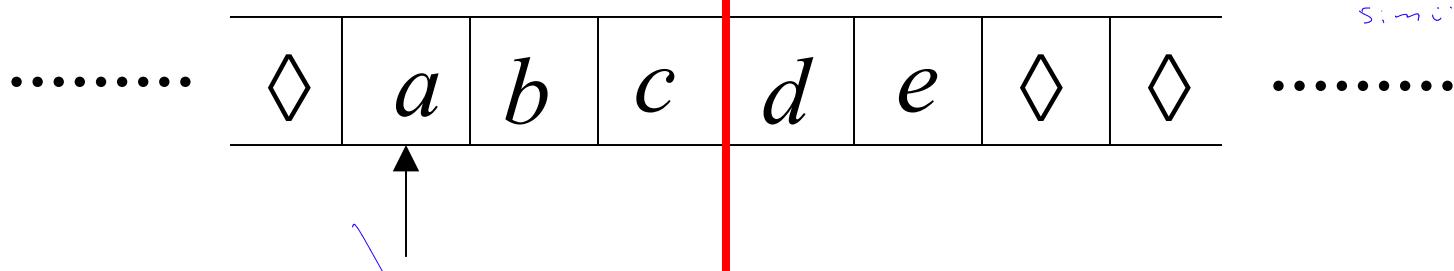


Semi-Infinite tape machine



Squeeze infinity of both directions
to one direction

Standard machine



reference point

Semi-Infinite tape machine with two tracks

Right part

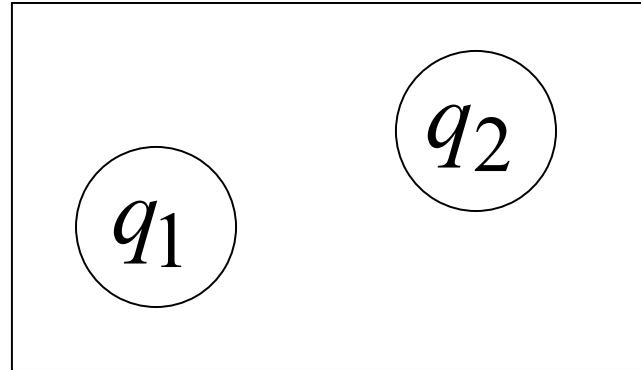
#	<i>d</i>	<i>e</i>	◊	◊	◊	
#	<i>c</i>	<i>b</i>	<i>a</i>	◊	◊	

.....

Left part

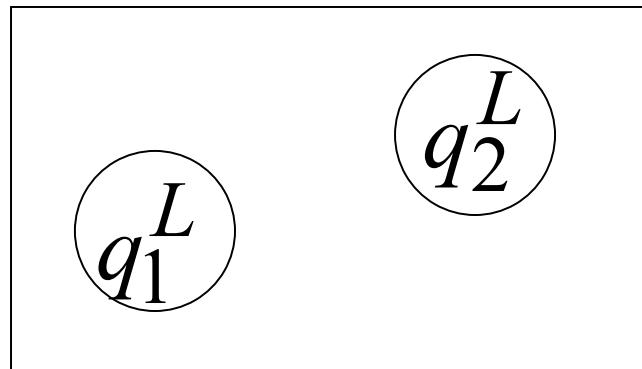


Standard machine

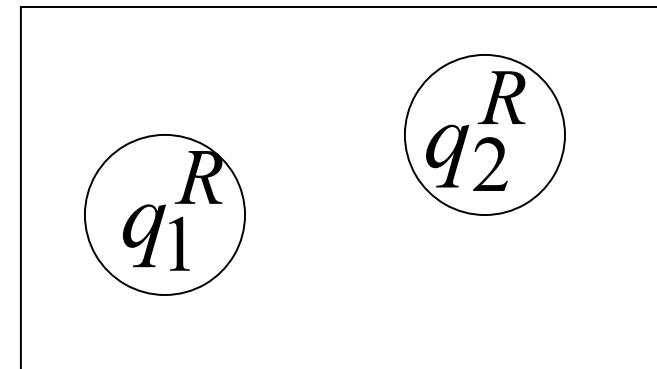


Semi-Infinite tape machine

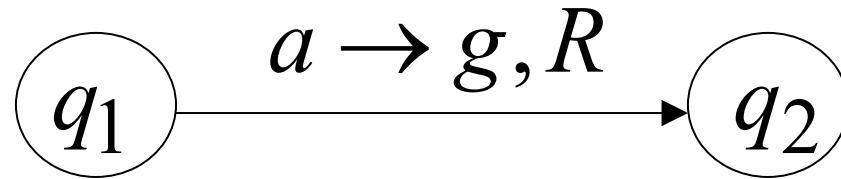
Left part



Right part

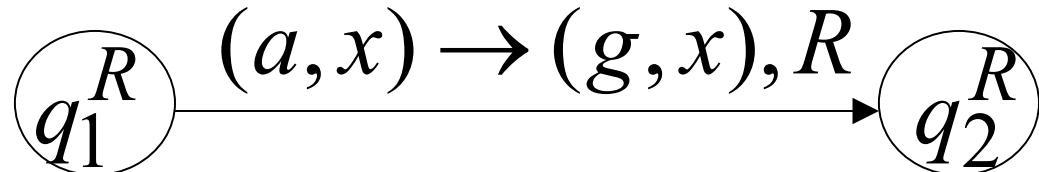


Standard machine

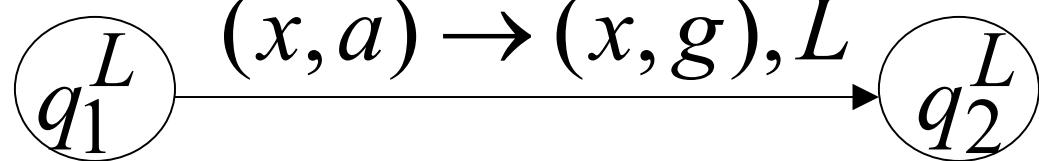


Semi-Infinite tape machine

Right part



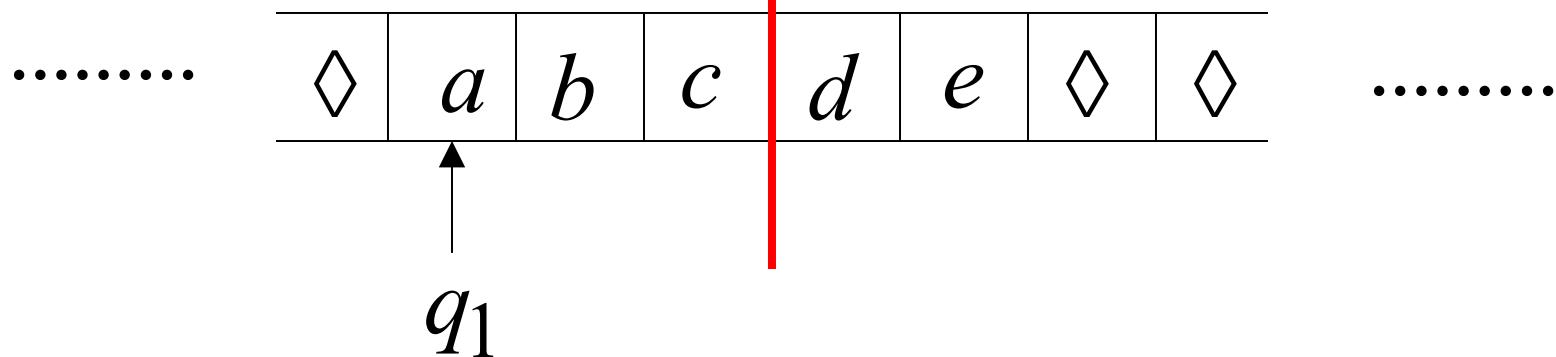
Left part



For all tape symbols x

Time 1

Standard machine



Semi-Infinite tape machine

Right part

#	d	e	diamond	diamond	diamond	
#	c	b	a	diamond	diamond	

.....

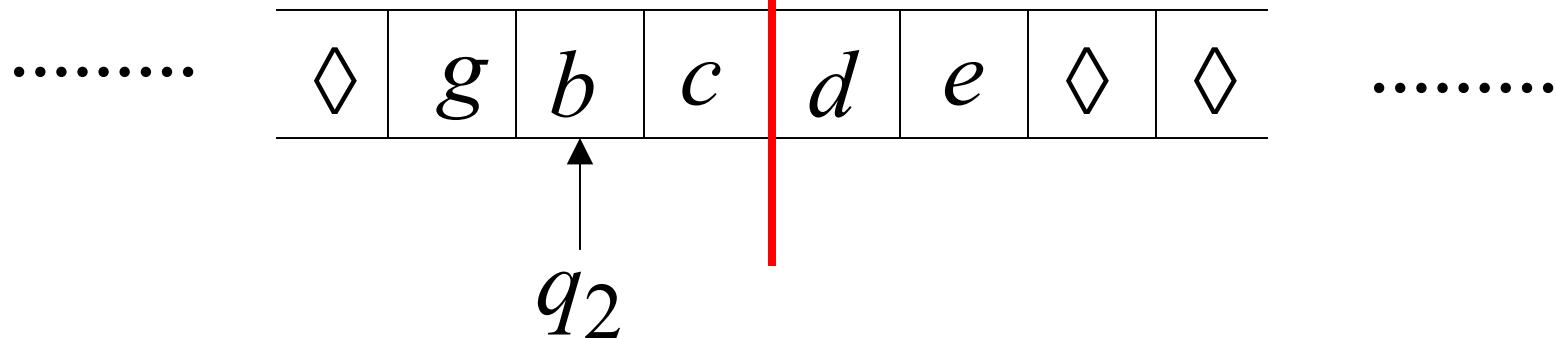
Left part

#	c	b	a	diamond	diamond	
#	c	b	a	diamond	diamond	

q_1^L

Time 2

Standard machine



Semi-Infinite tape machine

Right part

#	d	e	\diamond	\diamond	\diamond	
#	c	b	g	\diamond	\diamond	

.....

Left part

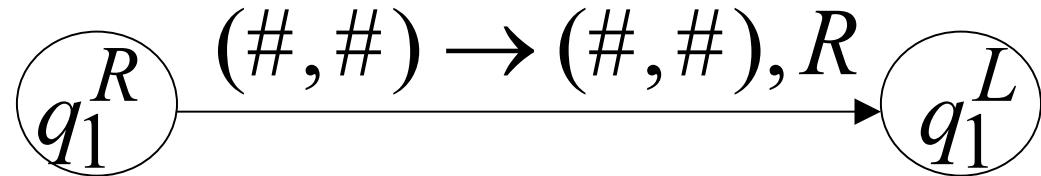
#	c	b	g	\diamond	\diamond	
#	c	b	g	\diamond	\diamond	

q_2^L

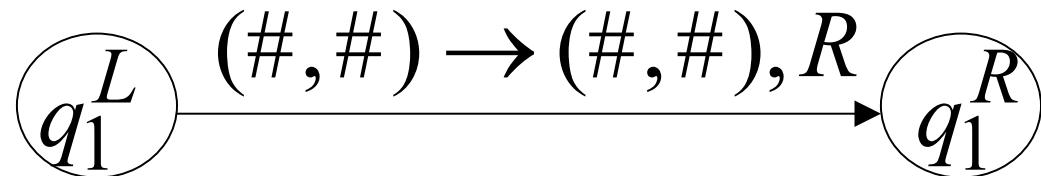
At the border:

Semi-Infinite tape machine

Right part



Left part



Semi-Infinite tape machine

Right part

Left part

Time 1

#	d	e	◊	◊	◊	
#	c	b	g	◊	◊	

q_1^L

.....

Right part

Left part

Time 2

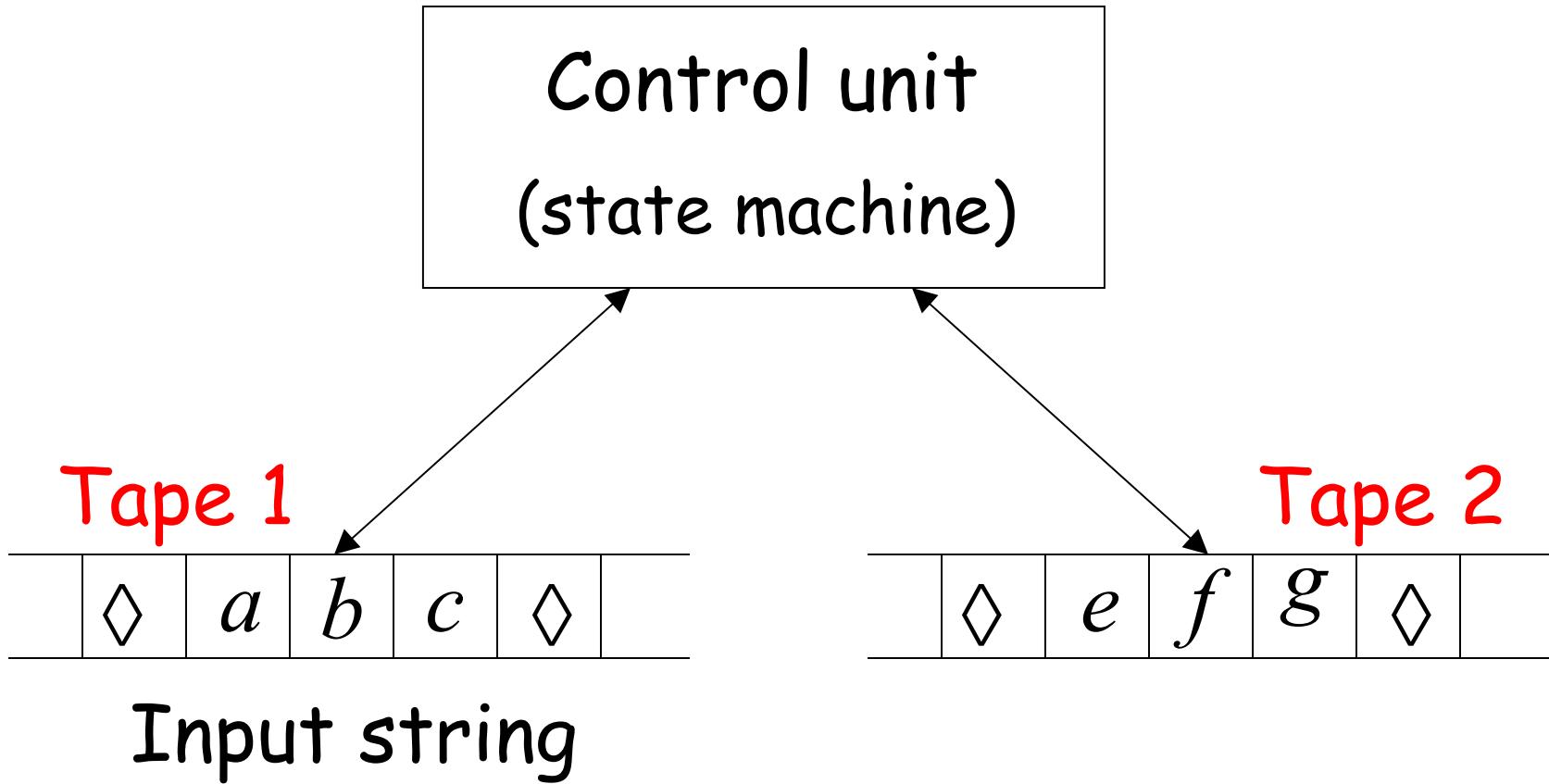
#	d	e	◊	◊	◊	
#	c	b	g	◊	◊	

q_1^R

.....

END OF PROOF

Multi-tape Turing Machines



Input string appears on Tape 1

Tape 1

	◊	a	b	c	◊	
--	---	---	---	---	---	--

q_1

Time 1

	◊	e	f	g	◊	
--	---	---	---	---	---	--

q_1

Tape 1

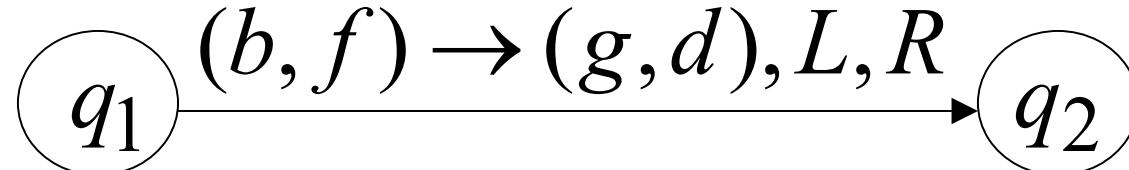
	◊	a	g	c	◊	
--	---	---	---	---	---	--

q_2

Time 2

	◊	e	d	g	◊	
--	---	---	---	---	---	--

q_2



Theorem: Multi-tape machines
have the same power with
Standard Turing machines

Proof: 1. Multi-tape machines
simulate Standard Turing machines

2. Standard Turing machines
simulate Multi-tape machines

1. Multi-tape machines simulate
Standard Turing Machines:

Trivial: Use one tape

2. Standard Turing machines simulate Multi-tape machines:

Standard machine:

- Uses a multi-track tape to simulate the multiple tapes
- A tape of the Multi-tape machine corresponds to a pair of tracks

Multi-tape Machine

Tape 1

	◊	a	b	c	◊	
--	---	---	---	---	---	--

↑

Tape 2

	◊	e	f	g	h	◊
--	---	---	---	---	---	---

↑

Standard machine with four track tape

		a	b	c			
		0	1	0			
		e	f	g	h		
		0	0	1	0		

↑

Tape 1
head position
Tape 2
head position

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 Multi-tape state transition:

1. Return to reference point
2. Find current symbol in Track 1 and update
3. Return to reference point
4. Find current symbol in Tape 2 and update

END OF PROOF

Same power doesn't imply same speed:

$$L = \{a^n b^n\}$$

Kar�amal

Standard Turing machine: $O(n^2)$ time

Go back and forth $O(n^2)$ times
to match the a's with the b's

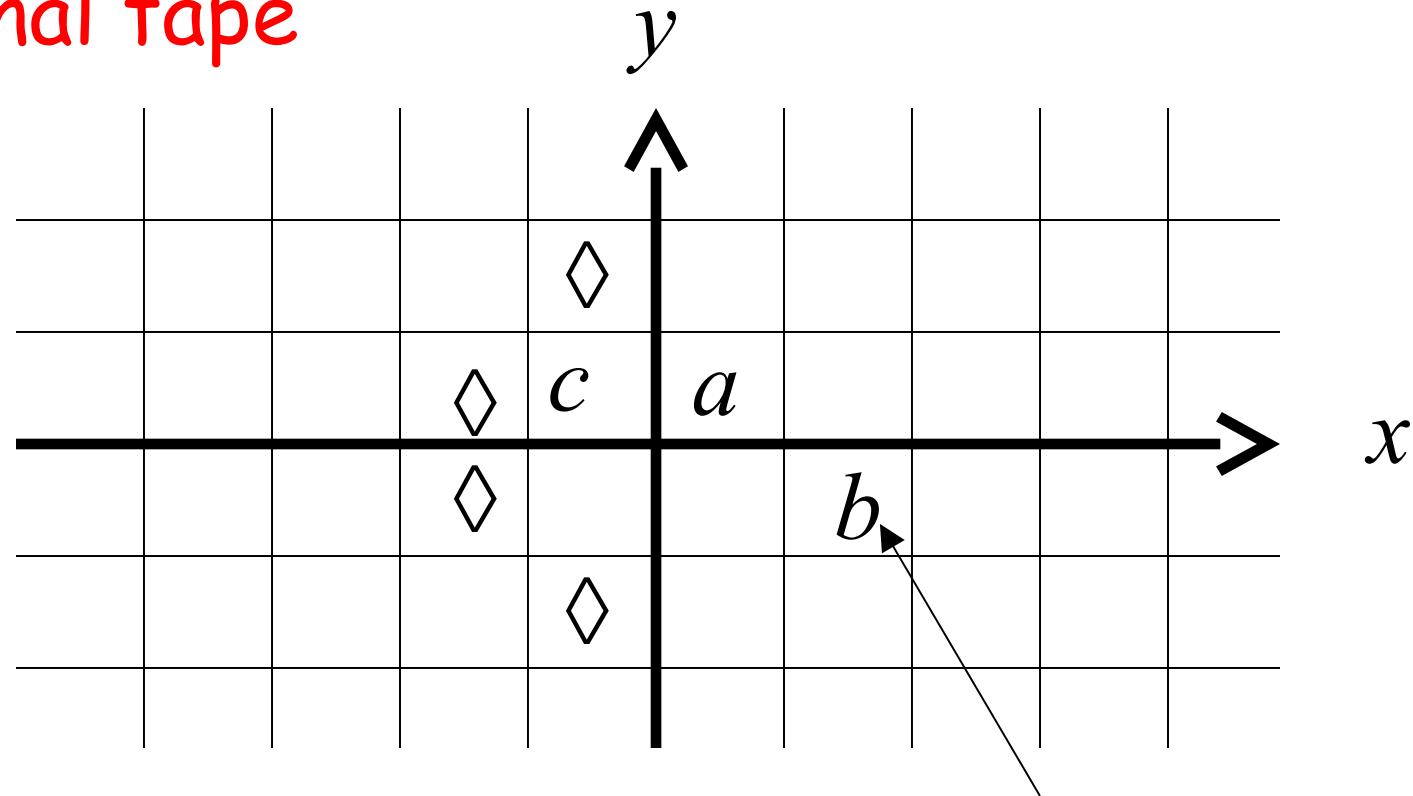
2-tape machine: $O(n)$ time

1. Copy b^n to tape 2 ($O(n)$ steps)

2. Compare a^n on tape 1
and b^n on tape 2 ($O(n)$ steps)

Multidimensional Turing Machines

2-dimensional tape



MOVES: L,R,U,D

U: up D: down

HEAD

Position: +2, -1

Theorem: Multidimensional machines have the same power with Standard Turing machines

Proof: 1. Multidimensional machines simulate Standard Turing machines

2. Standard Turing machines simulate Multi-Dimensional machines

1. Multidimensional machines simulate Standard Turing machines

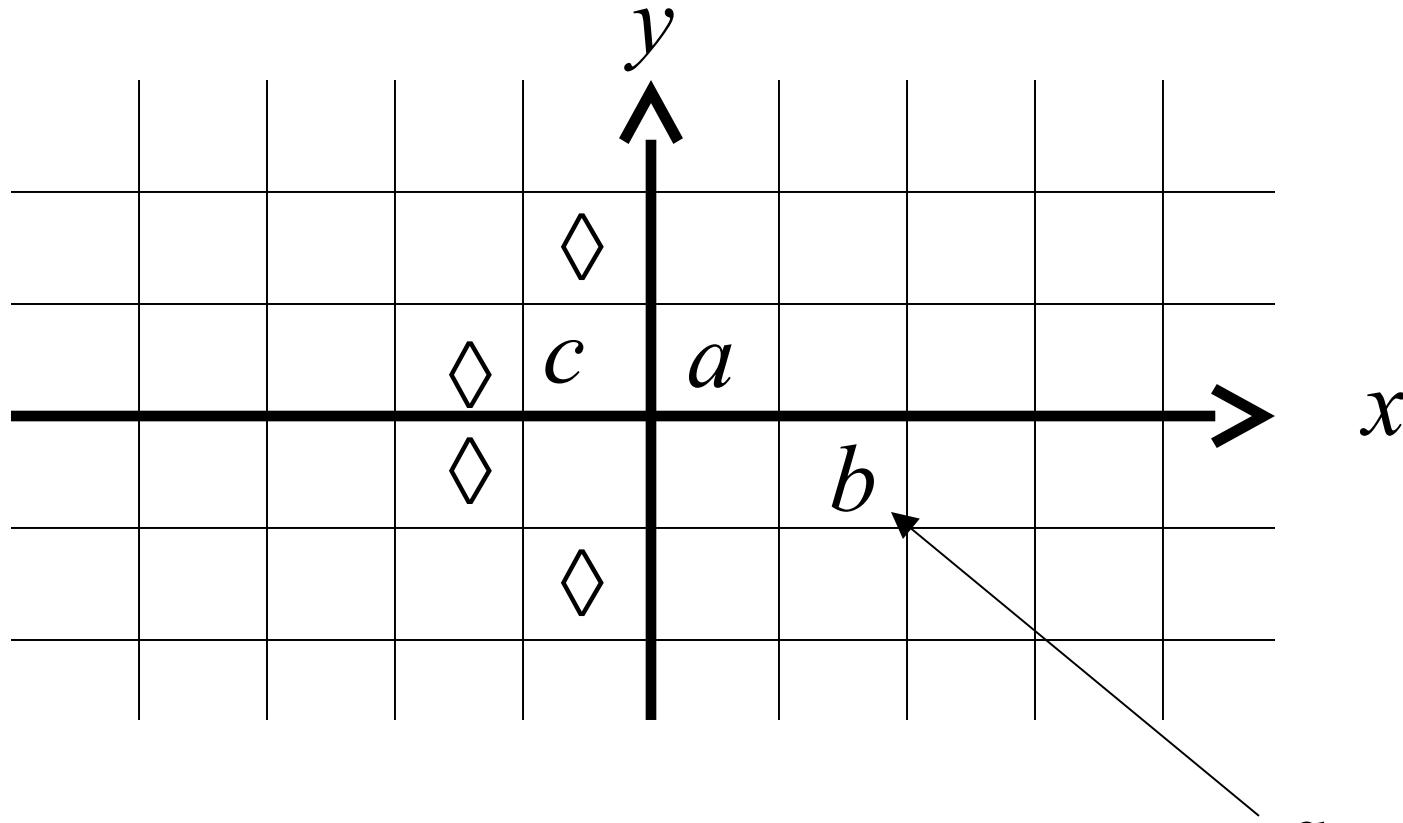
Trivial: Use one dimension

2. Standard Turing machines simulate Multidimensional machines

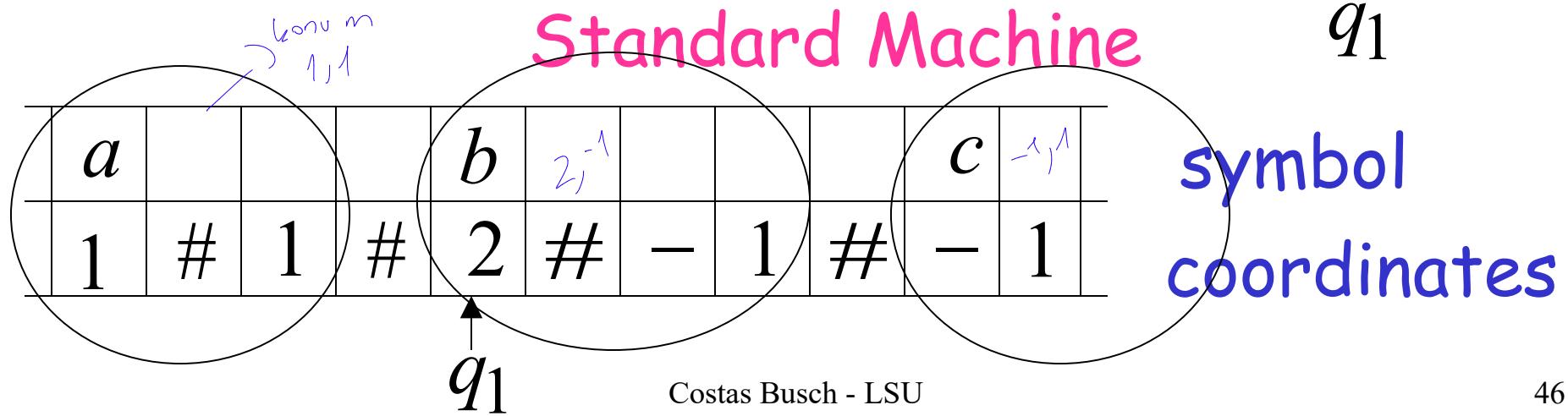
Standard machine:

- Use a two track tape
- Store symbols in track 1
- Store coordinates in track 2

2-dimensional machine



Standard Machine



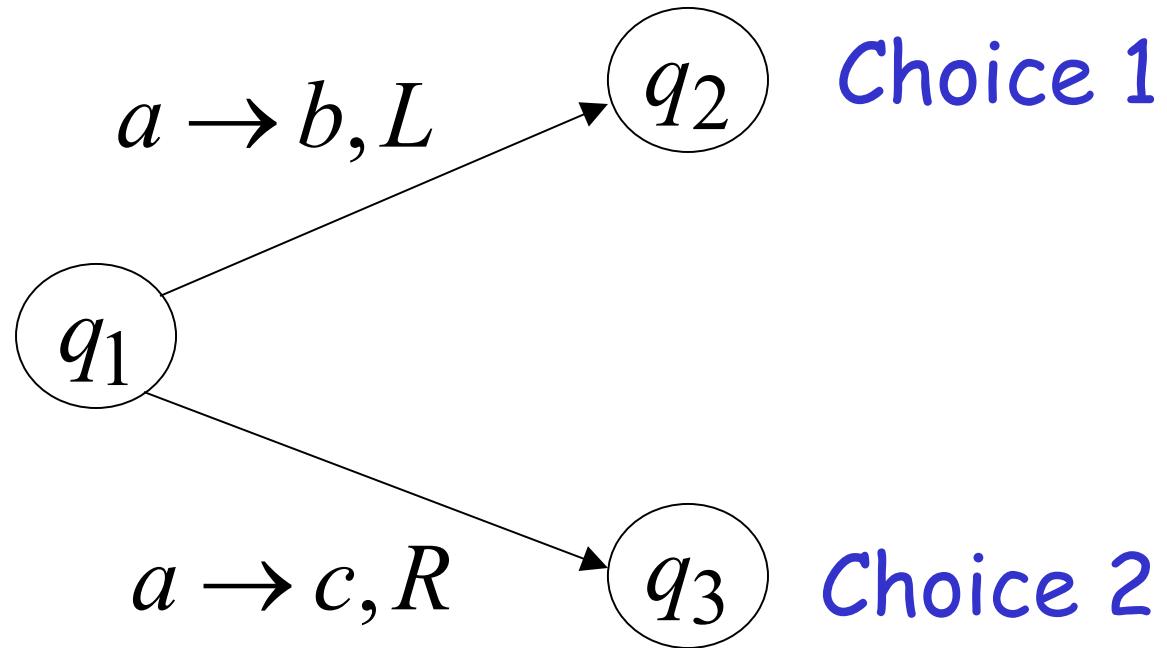
Standard machine:

Repeat for each transition followed
in the 2-dimensional machine:

1. Update current symbol
2. Compute coordinates of next position
3. Find next position on tape

END OF PROOF

Nondeterministic Turing Machines

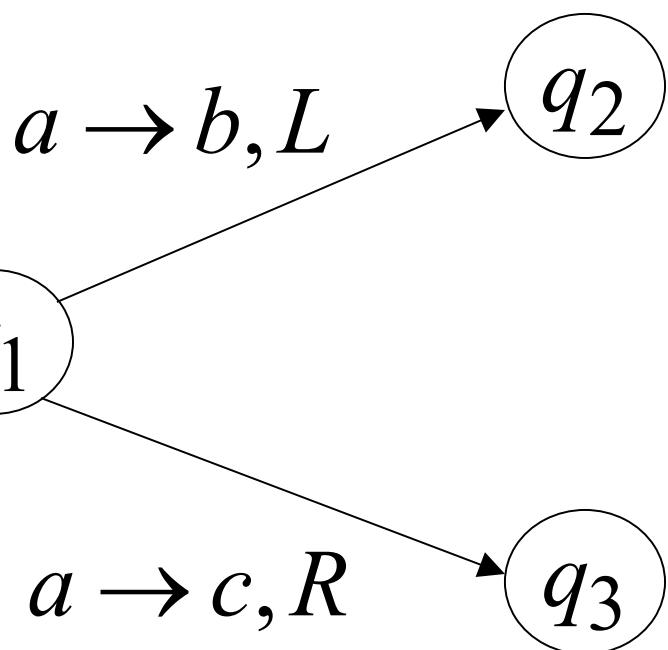


Allows Non Deterministic Choices

Time 0

	◊	a	b	c	◊	
		q_1				

Time 1



Choice 1

	◊	b	b	c	◊	
		q_2				

Choice 2

	◊	c	b	c	◊	
		q_3				

Input string w is accepted if there is a computation:

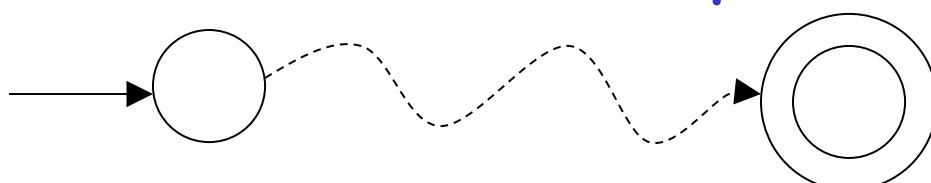
$$q_0 w \prec x q_f y$$

A green upward-pointing arrow indicating the next step in the sequence.

Final Configuration

Any accept state

There is a computation:



Theorem: Nondeterministic machines

have the same power with

Standard Turing machines

Proof: 1. Nondeterministic machines

simulate Standard Turing machines

2. Standard Turing machines

simulate Nondeterministic machines

1. Nondeterministic Machines simulate Standard (deterministic) Turing Machines

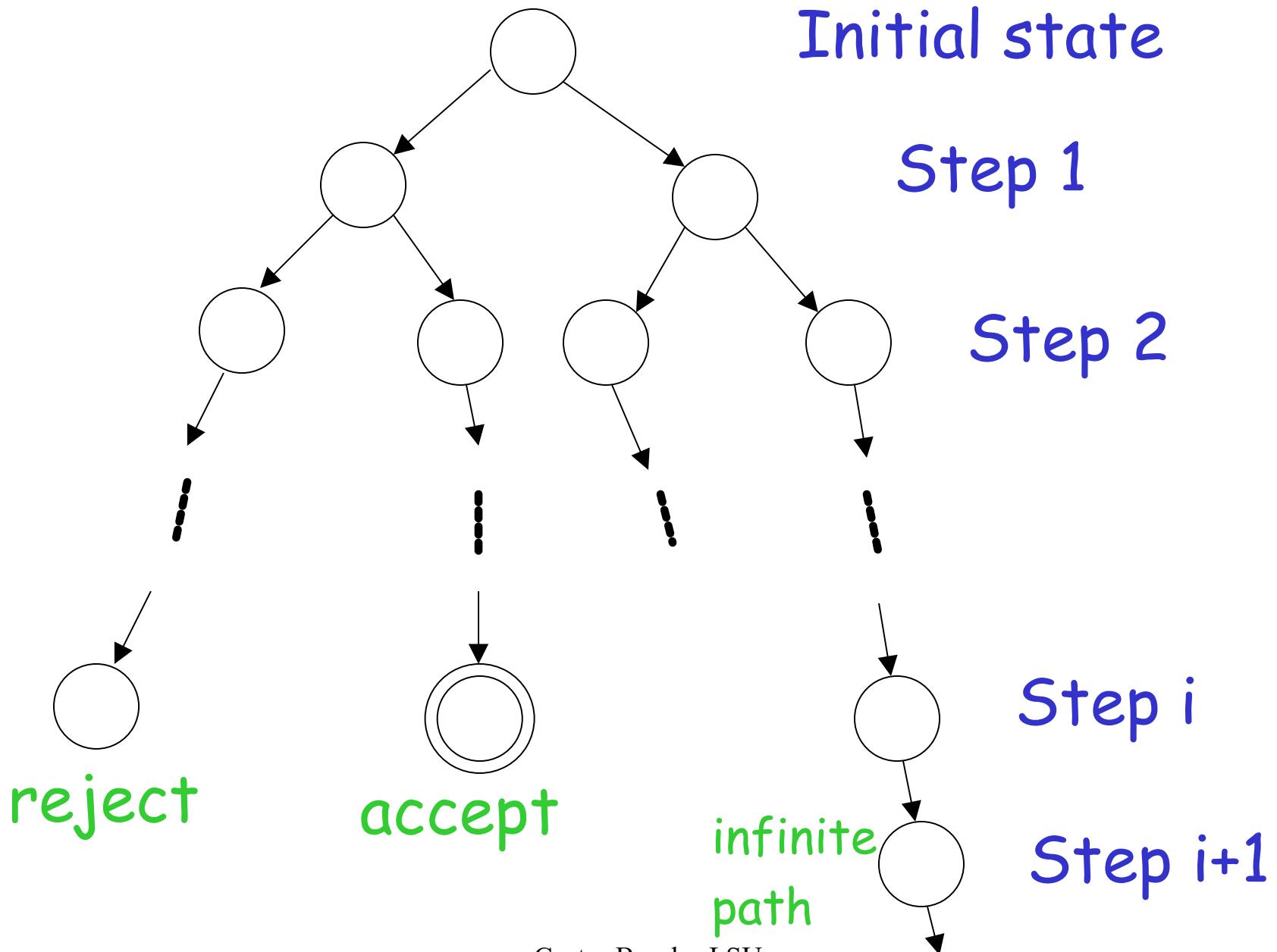
Trivial: every deterministic machine
is also nondeterministic

2. Standard (deterministic) Turing machines simulate Nondeterministic machines:

Deterministic machine:

- Uses a 2-dimensional tape
(equivalent to standard Turing machine with one tape)
- Stores all possible computations
of the non-deterministic machine
on the 2-dimensional tape

All possible computation paths

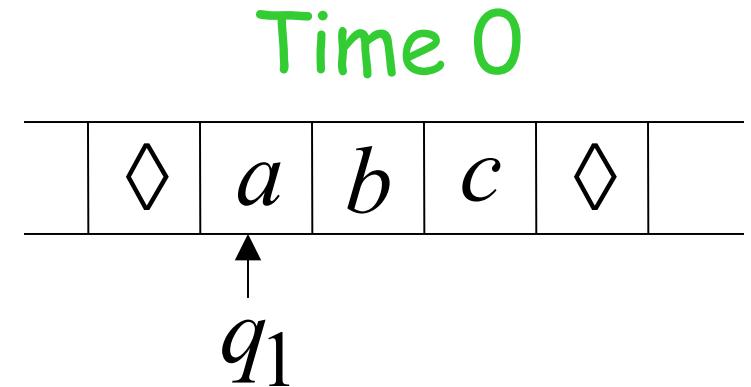
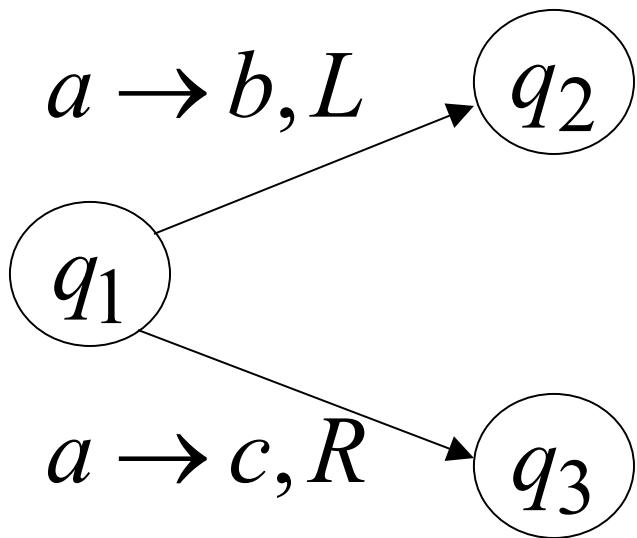


The Deterministic Turing machine
simulates all possible computation paths:

- simultaneously
- step-by-step
- with breadth-first search

depth-first may result getting stuck at exploring
an infinite path before discovering the accepting path

NonDeterministic machine

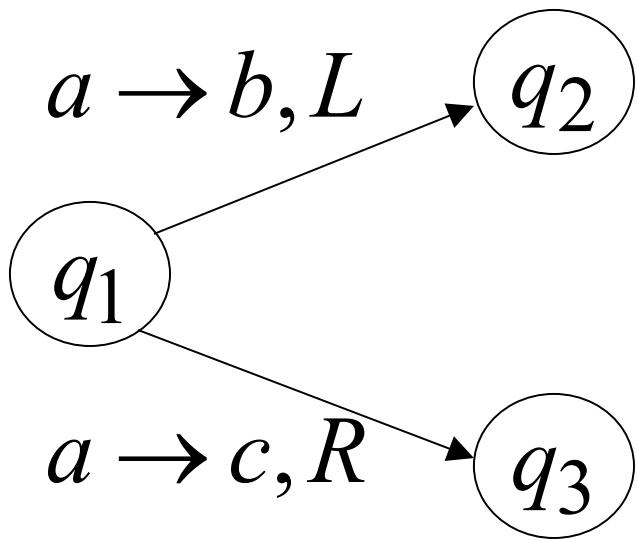


Deterministic machine

	#	#	#	#	#	#	
	#	a	b	c	#		
	#	q_1			#		
	#	#	#	#	#		

current
configuration

NonDeterministic machine



Time 1

	◊	b	b	c	◊	
--	---	-----	-----	-----	---	--

q_2

	◊	c	b	c	◊	
--	---	-----	-----	-----	---	--

q_3

Choice 1

Choice 2

Deterministic machine

	#	#	#	#	#	#	
#		b	b	c	#		
#	q_2				#		
#		c	b	c	#		
#			q_3		#		

Computation 1

Computation 2

Deterministic Turing machine

Repeat

For each configuration in current step
of non-deterministic machine,
if there are two or more choices:

1. Replicate configuration
2. Change the state in the replicas

Until either the input string is accepted
or rejected in all configurations

If the non-deterministic machine accepts the input string:

The deterministic machine accepts and halts too

The simulation takes in the worst case exponential time compared to the shortest length of an accepting path

If the non-deterministic machine does not accept the input string:

1. The simulation halts if all paths reach a halting state

OR

2. The simulation never terminates if there is a never-ending path (infinite loop)

In either case the deterministic machine rejects too (1. by halting or 2. by simulating the infinite loop)

END OF PROOF

A Universal Turing Machine

A limitation of Turing Machines:

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
simulates any Turing Machine M

Input of Universal Turing Machine:

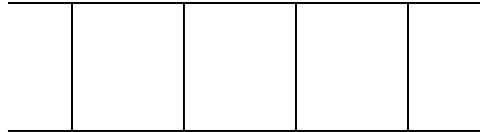
Description of transitions of M

Input string of M

Three tapes



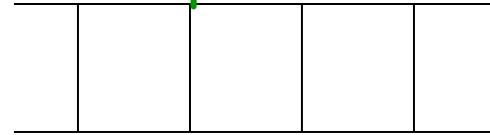
Tape 1



Description of M

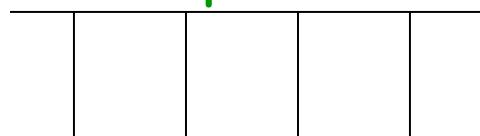
Diger makinenin formal tipini
vermek lazim

Tape 2



Tape Contents of M

Tape 3



State of M

Tape 1

--	--	--	--	--

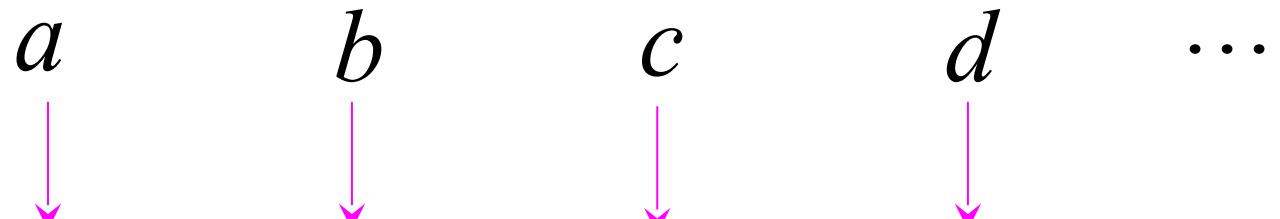
Description of M

We describe Turing machine M
as a string of symbols:

We encode M as a string of symbols

Alphabet Encoding

Symbols:



Encoding:

State Encoding

States:

q_1

q_2

q_3

q_4

...



Encoding:

1

11

111

1111

Head Move Encoding

Move:

L

R



Encoding:

1

11

Transition Encoding

Transition:

$$\delta(q_1, a) = (q_2, b, L)$$

Encoding:

1 0 1 0 1 1 0 1 1 0 1

separator

Turing Machine Encoding

Transitions:

$$\delta(q_1, a) = (q_2, b, L) \quad \delta(q_2, b) = (q_3, c, R)$$

Encoding:

1 0 1 0 1 1 0 1 1 0 1 0 0 1 1 0 1 1 0 1 1 1 0 1 1 1 0 1 1

↑
separator

Tape 1 contents of Universal Turing Machine:

binary encoding
of the simulated machine M

Tape 1

1 0 1 0 11 0 11 0 10011 0 1 10 111 0 111 0 1100...



A Turing Machine is described
with a binary string of 0's and 1's

Therefore:

The set of Turing machines
forms a language:

each string of this language is
the binary encoding of a Turing Machine

Language of Turing Machines

$L = \{ 1010110101, \dots \}$ (Turing Machine 1)

$101011101011, \dots$ (Turing Machine 2)

$1110101111010111, \dots$

Countable Sets

Infinite sets are either:

Countable

or

Uncountable

Countable set:

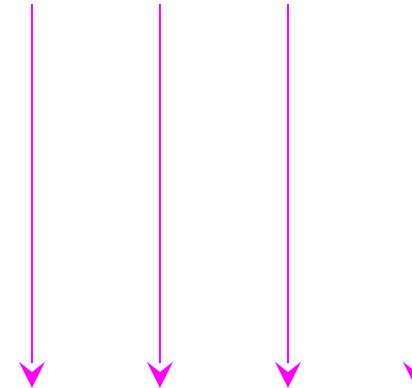
There is a one to one correspondence (injection)
of
elements of the set
to
Positive integers (1,2,3,...)

Every element of the set is mapped to a positive number
such that no two elements are mapped to same number

Example: The set of even integers is countable

Even integers: 0, 2, 4, 6, ...
(positive)

Correspondence:



Positive integers: 1, 2, 3, 4, ...

$2n$ corresponds to $n + 1$

Example: The set of rational numbers
is countable

Rational numbers:

$$\frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \dots$$

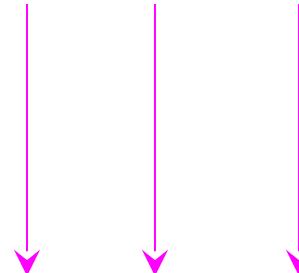
Naïve Approach

Rational numbers:

Nominator 1

$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \dots$

Correspondence:



Positive integers:

1, 2, 3, ...

Doesn't work:

we will never count
numbers with nominator 2:

$\frac{2}{1}, \frac{2}{2}, \frac{2}{3}, \dots$

Better Approach

$$\begin{array}{cccc} \frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ & & & \dots \end{array}$$

$$\begin{array}{ccc} \frac{2}{1} & \frac{2}{2} & \frac{2}{3} \\ & & \dots \end{array}$$

$$\begin{array}{cc} \frac{3}{1} & \frac{3}{2} \\ & \dots \end{array}$$

$$\begin{array}{c} 4 \\ \hline 1 \\ \dots \end{array}$$

$$\frac{1}{1}$$



$$\frac{1}{2}$$

$$\frac{1}{3}$$

$$\frac{1}{4}$$

...

$$\frac{2}{1}$$

$$\frac{2}{2}$$

$$\frac{2}{3}$$

...

$$\frac{3}{1}$$

$$\frac{3}{2}$$

...

$$\frac{4}{1}$$

...

$$\frac{1}{1}$$



$$\frac{1}{2}$$

$$\frac{2}{1}$$



$$\frac{2}{2}$$

$$\frac{1}{3}$$

$$\frac{1}{4}$$

...

$$\frac{2}{3}$$

...

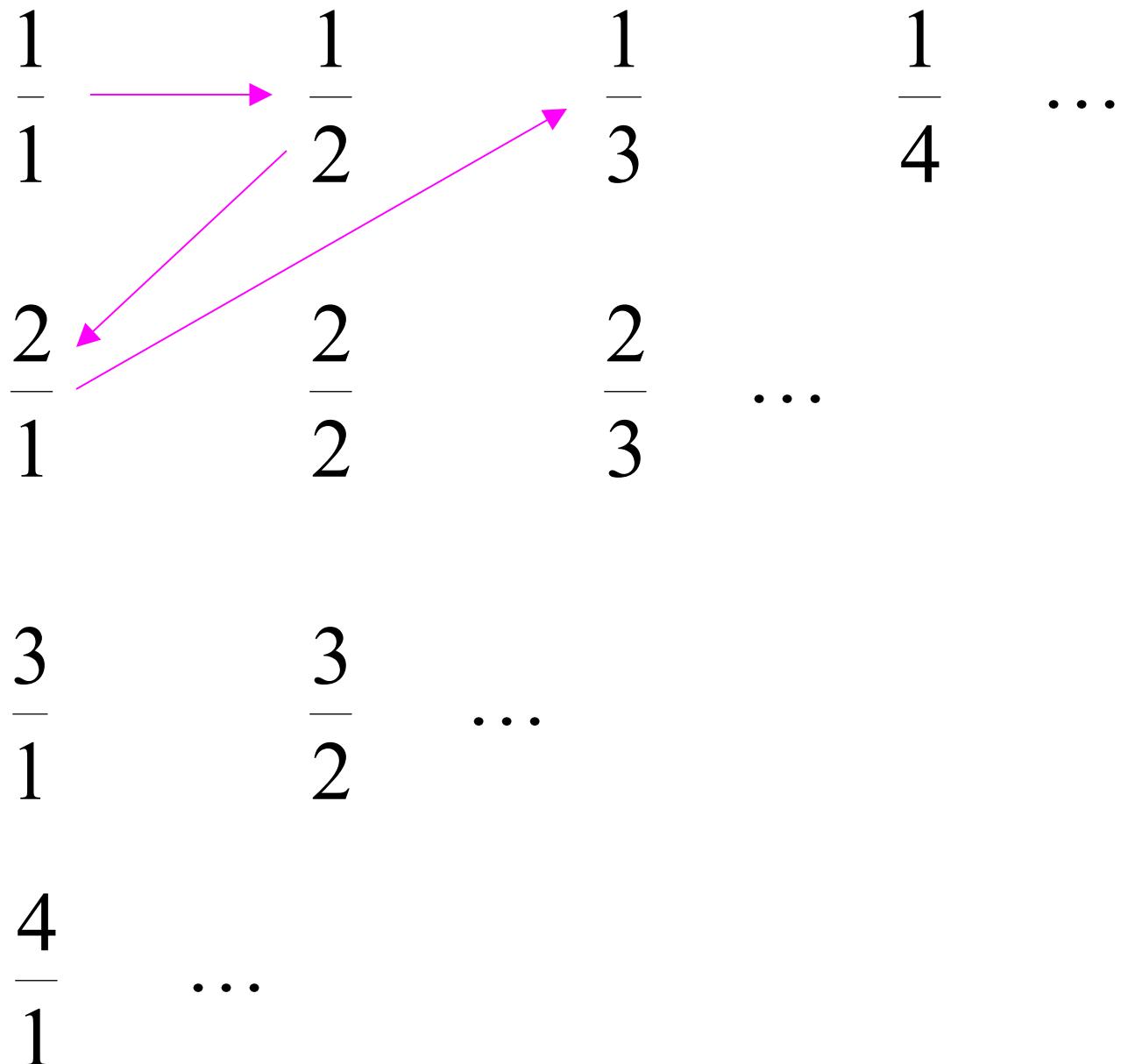
$$\frac{3}{1}$$

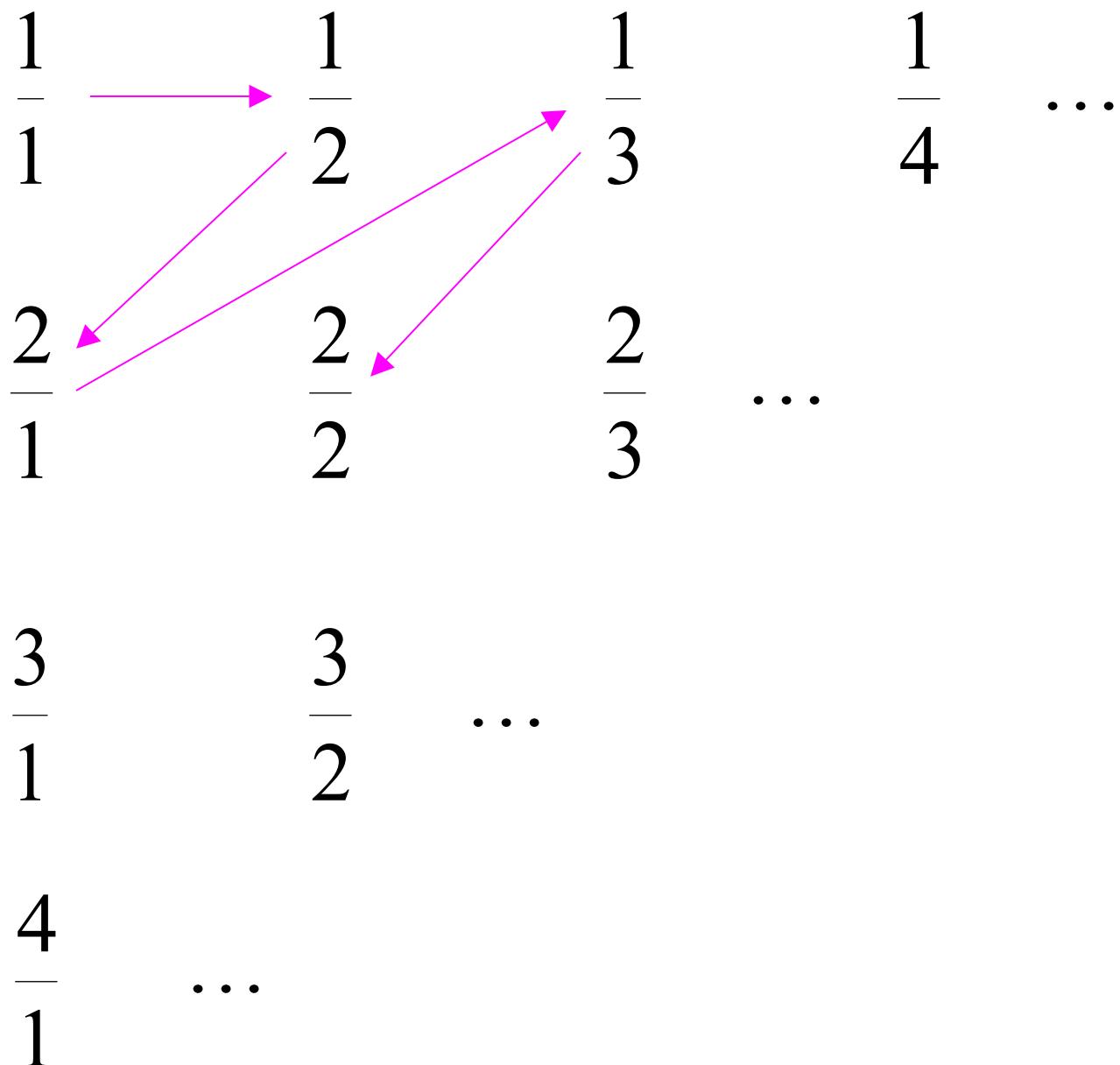
$$\frac{3}{2}$$

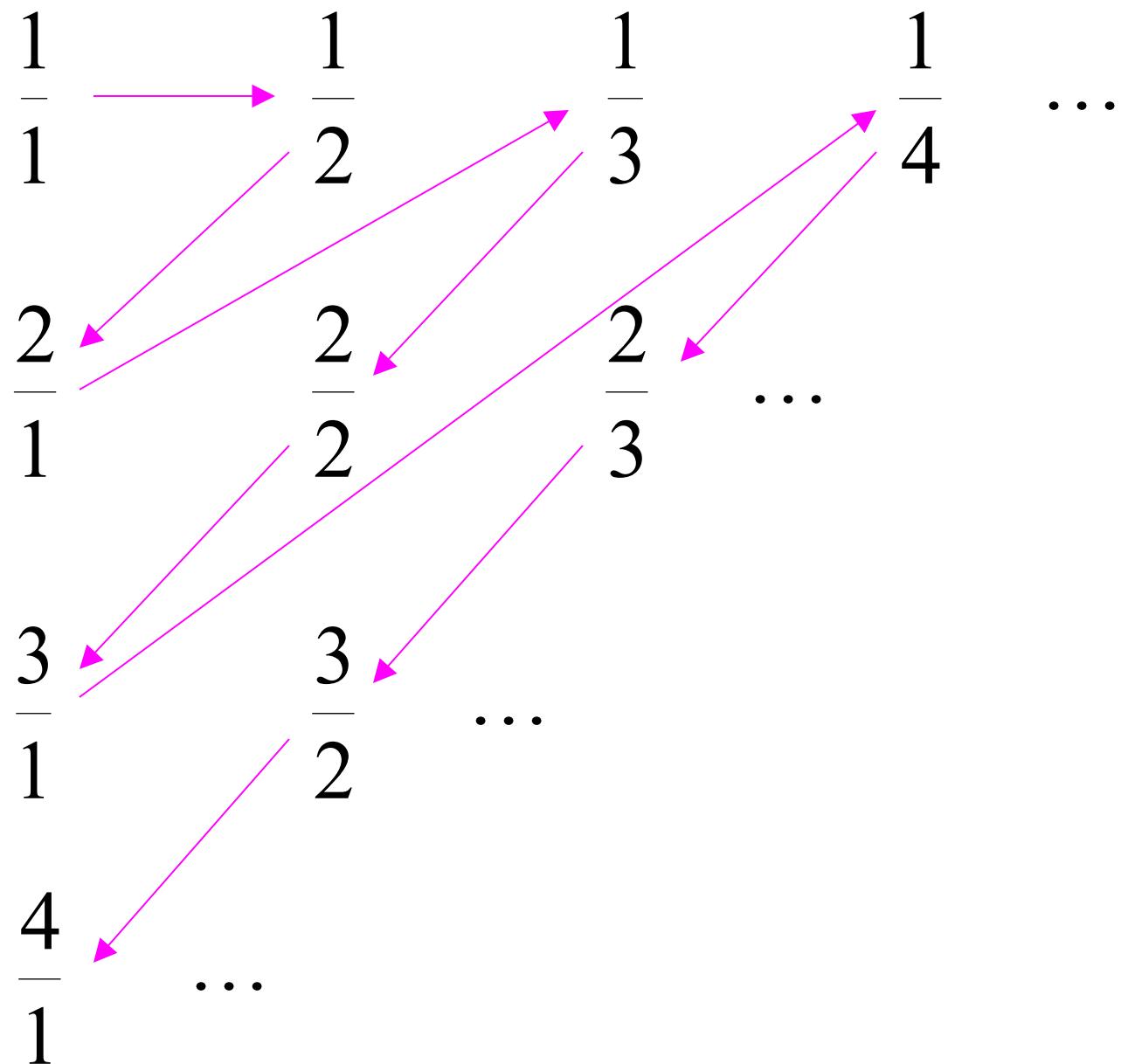
...

$$\frac{4}{1}$$

...







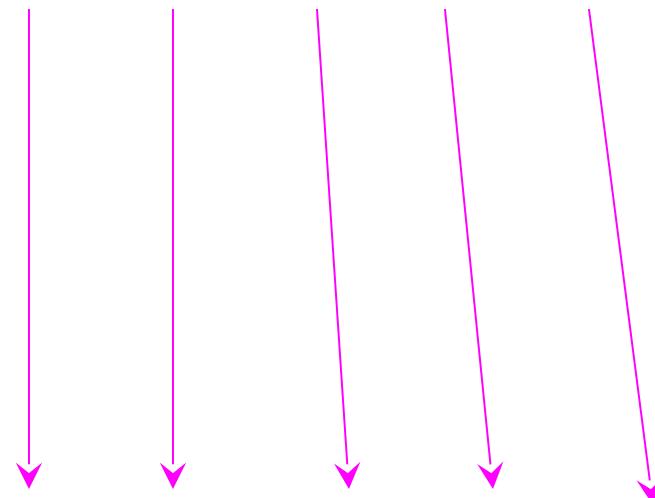
Rational Numbers:

$$\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{2}{2}, \dots$$

Correspondence:

Positive Integers:

$$1, 2, 3, 4, 5, \dots$$



We proved:

the set of rational numbers is countable
by describing an enumeration procedure
(enumerator)
for the correspondence to natural numbers

Definition

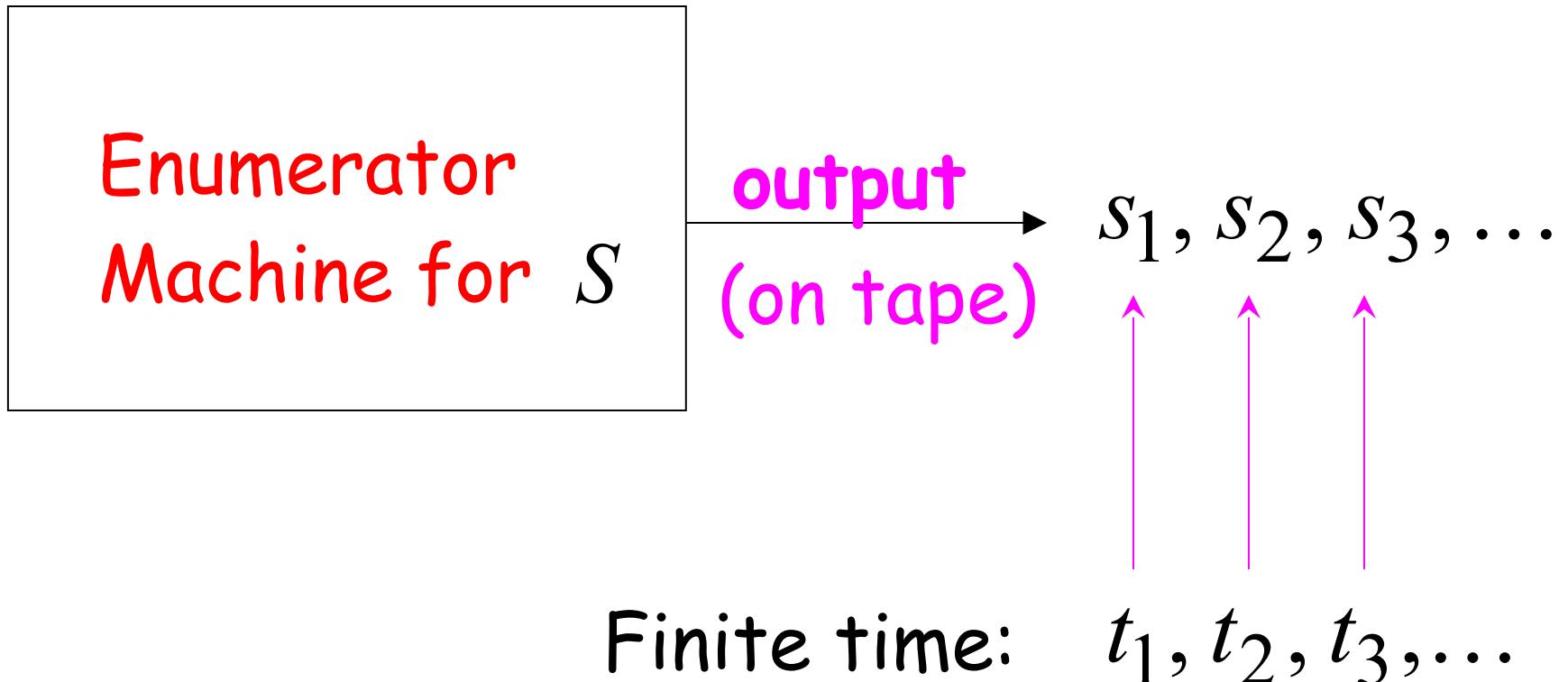
Let S be a set of strings (Language)

An **enumerator** for S is a Turing Machine
that generates (prints on tape)
all the strings of S one by one

and

each string is generated in finite time

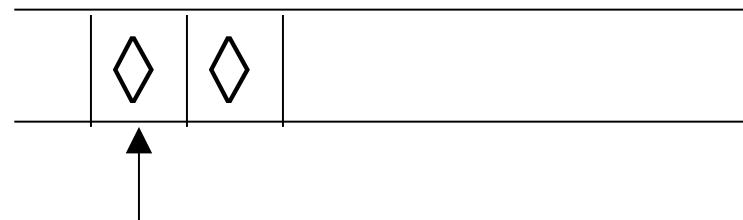
strings $s_1, s_2, s_3, \dots \in S$



Enumerator Machine

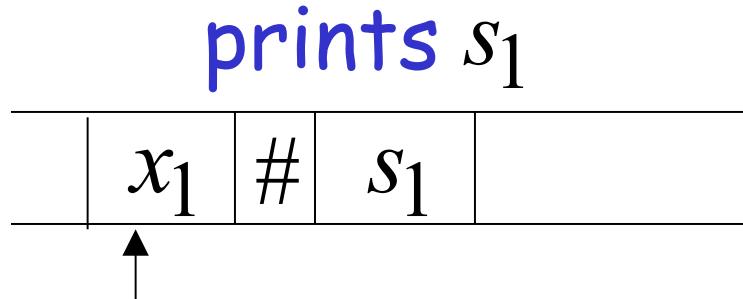
Configuration

Time 0



q_0

Time t_1



q_s

prints s_1

prints s_2

Time t_2

	x_2	#	s_2	
--	-------	---	-------	--



q_s

prints s_3

Time t_3

	x_3	#	s_3	
--	-------	---	-------	--



q_s

Observation:

If for a set S there is an enumerator,
then the set is countable

↳ B.: d.h. eine Enumerator var/sa
count able d.h.

The enumerator describes the correspondence of S to natural numbers

Example: The set of strings $S = \{a,b,c\}^+$
is countable

Approach:

We will describe an enumerator for S

Naive enumerator:

Produce the strings in lexicographic order:

$$s_1 = a$$

$$s_2 = aa$$

$$\vdots \quad aaa$$

$$aaaa$$

.....

Doesn't work:

strings starting with b
will never be produced

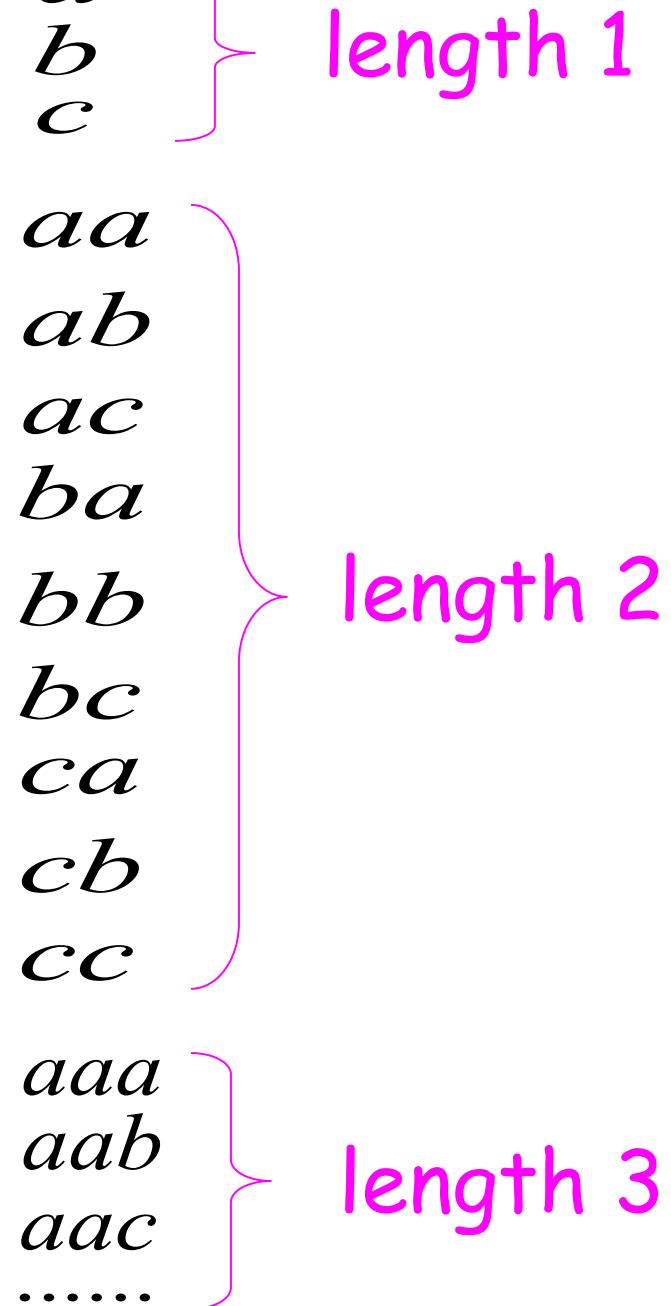
Better procedure: **Proper Order**
(Canonical Order)

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

⋮

$$\begin{aligned}s_1 &= a \\ s_2 &= b \\ \vdots &= c\end{aligned}$$

Produce strings in
Proper Order:



Theorem: The set of all Turing Machines
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

Enumerator:

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

Binary strings

0

ignore

1

ignore

00

ignore

01

⋮
⋮
⋮

1 0 1 0 1 1 0 1 1 0 0

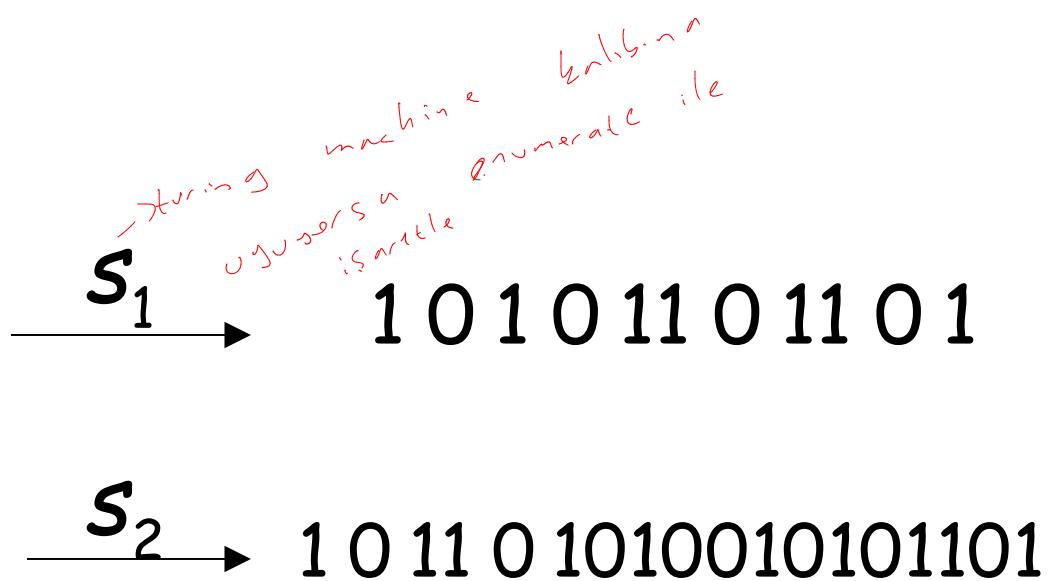
1 0 1 0 1 1 0 1 1 0 1

⋮
⋮
⋮

1 0 1 1 0 1 0 1 0 0 1 0 1 0 1 1 0 1

⋮
⋮
⋮

Turing Machines



End of Proof

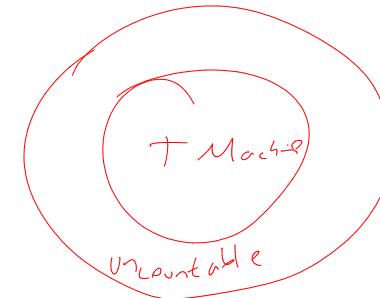
Simpler Proof:

Each Turing machine binary string is mapped to the number representing its value

Uncountable Sets

We will prove that there is a language L
which is not accepted by any Turing machine

Technique:



Turing machines are countable

Languages are uncountable

(there are more languages than Turing Machines)

Theorem:

If S is an infinite countable set, then

the powerset 2^S of S is uncountable.

*infinite
re countable
set in powerset:
uncountable d.r.*

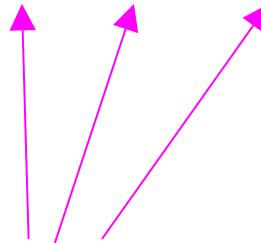
The powerset 2^S contains all possible subsets of S

Example: $S = \{a, b\}$ $2^S = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

Proof:

Since S is countable, we can list its elements in some order

$$S = \{s_1, s_2, s_3, \dots\}$$



Elements of S

Elements of the powerset 2^S have the form:

\emptyset

$\{s_1, s_3\}$

$\{s_5, s_7, s_9, s_{10}\}$

⋮

They are subsets of S

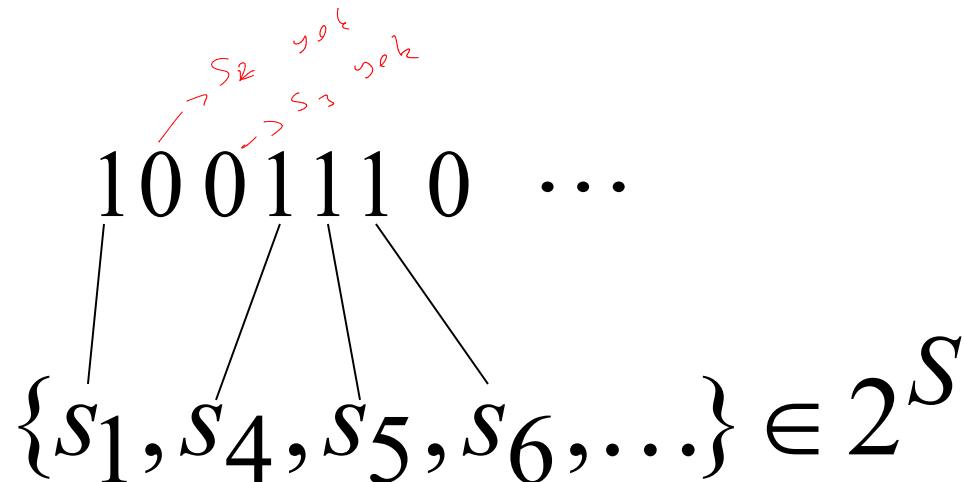
We encode each subset of S
with a binary string of 0's and 1's

Subset of S	Binary encoding	s_1	s_2	s_3	s_4	\dots
$\{s_1\}$	Binary encoding	1	0	0	0	\dots
$\{s_2, s_3\}$		0	1	1	0	\dots
$\{s_1, s_3, s_4\}$		1	0	1	1	\dots

Every infinite binary string corresponds to a subset of S :

Example:

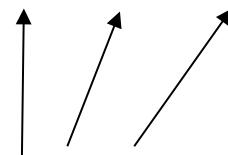
Corresponds to:



Let's assume (for contradiction)
that the powerset 2^S is countable

Then: we can list the elements of the
powerset in some order

$$2^S = \{t_1, t_2, t_3, \dots\}$$



Subsets of S

Powerset element

Binary encoding example

t_1	1	0	0	0	0	\dots
-------	---	---	---	---	---	---------

t_2	1	1	0	0	0	\dots
-------	---	---	---	---	---	---------

t_3	1	1	0	1	0	\dots
-------	---	---	---	---	---	---------

t_4	1	1	0	0	1	\dots
-------	---	---	---	---	---	---------

\dots

\dots

t = the binary string whose bits
are the complement of the diagonal

t_1	1	0	0	0	0	\dots
t_2	1	1	0	0	0	\dots
t_3	1	1	0	1	0	\dots
t_4	1	1	0	0	1	\dots

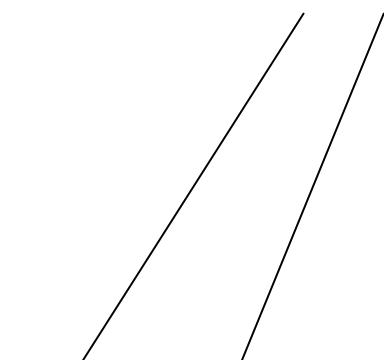
Binary string: $t = 0011\dots$

(binary complement of diagonal)

The binary string

$$t = 0011\dots$$

corresponds
to a subset of S :

$$t = \{s_3, s_4, \dots\} \in 2^S$$


t = the binary string whose bits
are the complement of the diagonal

t_1	1	0	0	0	0	\dots
t_2	1	1	0	0	0	\dots
t_3	1	1	0	1	0	\dots
t_4	1	1	0	0	1	\dots
$t = \underline{0}011\dots$						

Question: $t = t_1$? NO: differ in 1st bit

t = the binary string whose bits
are the complement of the diagonal

t_1	1	0	0	0	0	\dots
t_2	1	1	0	0	0	\dots
t_3	1	1	0	1	0	\dots
t_4	1	1	0	0	1	\dots
$t = 0011\dots$						

Question: $t = t_2$? NO: differ in 2nd bit

t = the binary string whose bits
are the complement of the diagonal

t_1	1	0	0	0	0	\dots
t_2	1	1	0	0	0	\dots
t_3	1	1	0	1	0	\dots
t_4	1	1	0	0	1	\dots
$t = 00\textcircled{1}1\dots$						

Question: $t = t_3$? NO: differ in 3rd bit

Thus: $t \neq t_i$ for every i

since they differ in the i th bit

However, $t \in 2^S \Rightarrow t = t_i$ for some i

Contradiction!!!

Therefore the powerset 2^S is uncountable

End of proof

An Application: Languages

Consider Alphabet : $A = \{a, b\}$

The set of all strings:

$$S = \{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

infinite and countable

because we can enumerate
the strings in proper order

Consider Alphabet : $A = \{a, b\}$

The set of all strings:

$$S = \{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

infinite and countable

Any language is a subset of S :

$$L = \{aa, ab, aab\}$$

Consider Alphabet : $A = \{a, b\}$

The set of all Strings:

$S = A^* = \{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$

infinite and countable

The powerset of S contains all languages:

$2^S = \{\emptyset, \{\varepsilon\}, \{a\}, \{a, b\}, \{aa, b\}, \dots, \{aa, ab, aab\}, \dots\}$

uncountable

Consider Alphabet : $A = \{a, b\}$

Turing machines:

M_1 M_2 M_3 ...

accepts

Languages accepted

L_1 L_2 L_3 ...

countable

Denote: $X = \{L_1, L_2, L_3, \dots\}$

Note: $X \subseteq 2^S$
 $(S = \{a, b\}^*)$

Languages accepted
by Turing machines:

X countable

All possible languages: 2^S uncountable

Therefore: $X \neq 2^S$

(since $X \subseteq 2^S$, we get $X \subset 2^S$)

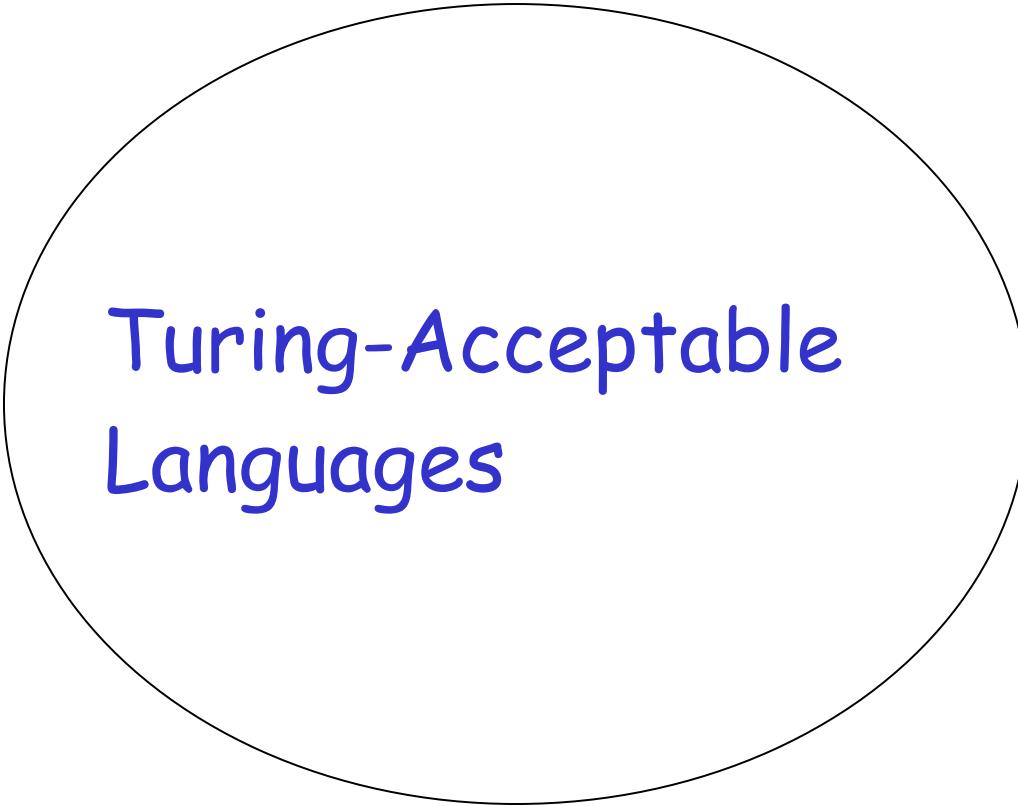
Conclusion:

There is a language L not accepted by any Turing Machine:

$$X \subset 2^S \longrightarrow \exists L \in 2^S \text{ and } L \notin X$$

Non Turing-Acceptable Languages

L



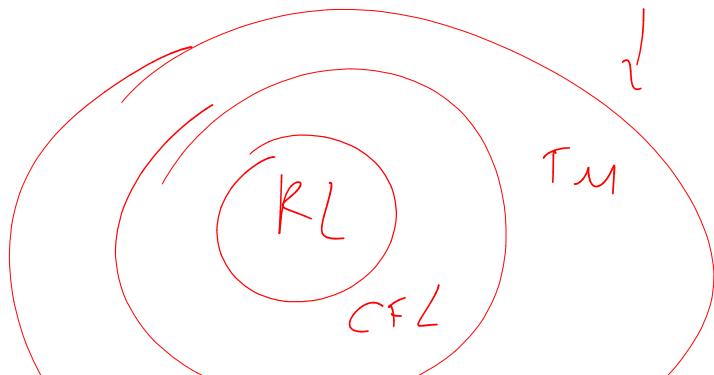
Turing-Acceptable
Languages

Note that: $X = \{L_1, L_2, L_3, \dots\}$

is a multi-set (elements may repeat)
since a language may be accepted
by more than one Turing machine

However, if we remove the repeated elements,
the resulting set is again countable since every element
still corresponds to a positive integer

20 Puan TF \rightarrow Decidability



TM Jigsaw
ister \otimes

$\rightarrow PL?$

Decidable Languages

Push down automato

Recall that:

TF sorusu
false gel:

A language L is **Turing-Acceptable**
if there is a Turing machine M
that accepts L

Also known as: **Turing-Recognizable**

or

Recursively-enumerable
languages

Turing-Acceptable \rightarrow Bir Language': turing-
turing machine valid

For any input string w :

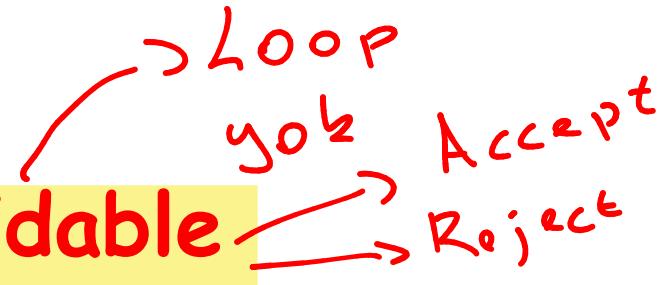
$w \in L \rightarrow M$ halts in an accept state

$w \notin L \rightarrow M$ halts in a non-accept state

or loops forever

Definition:

A language L is **decidable**
if there is a Turing machine (decider) M
which accepts L
and halts on every input string



Also known as *recursive languages*

Turing-Decidable

For any input string w :

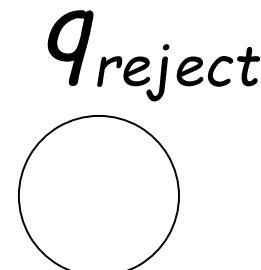
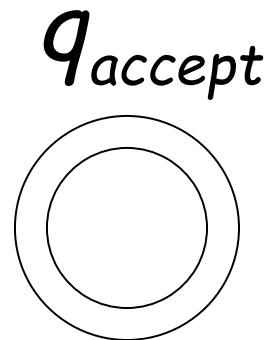
$w \in L \rightarrow M$ halts in an accept state

$w \notin L \rightarrow M$ halts in a non-accept state

Observation:

Every decidable language is Turing-Acceptable

Sometimes, it is convenient to have Turing machines with single accept and reject states

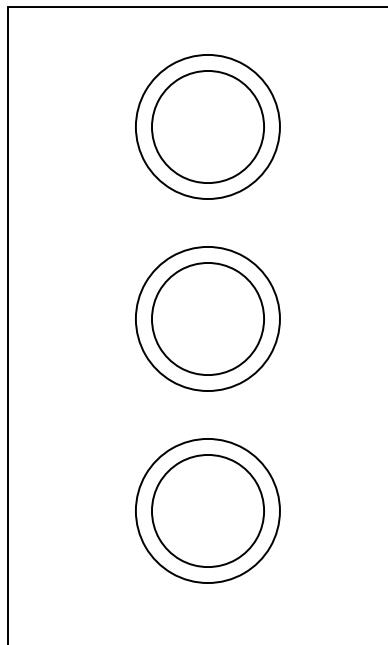


These are the only halting states

That result to possible
halting configurations

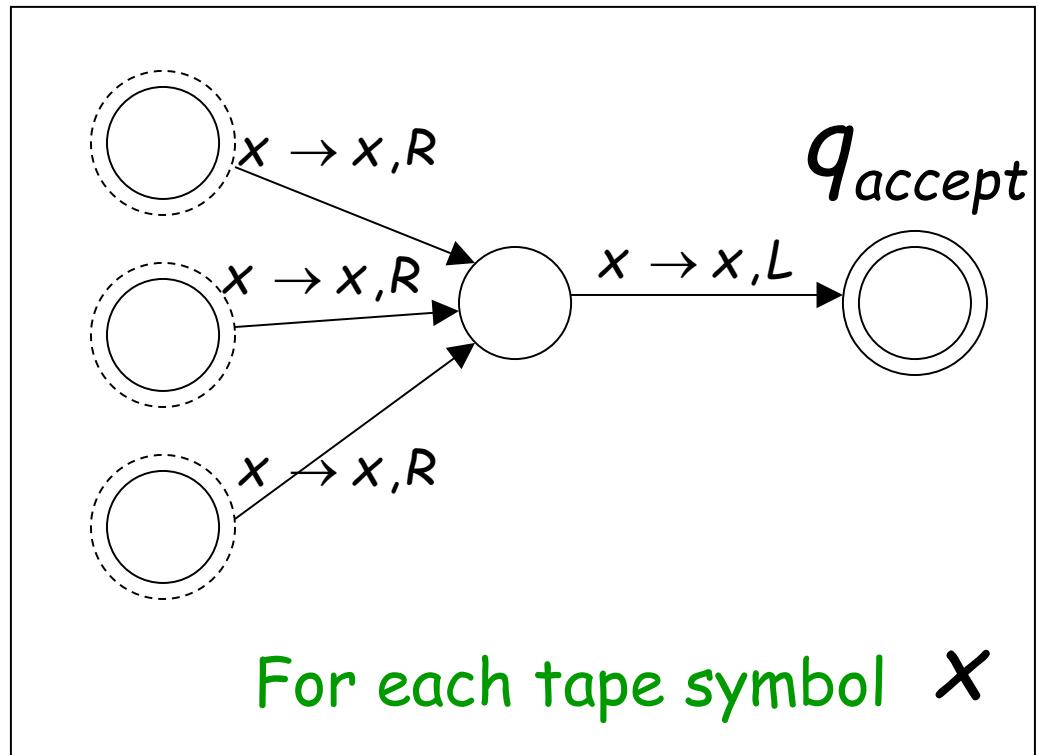
We can convert any Turing machine to have single accept and reject states

Old machine



Multiple
accept states

New machine

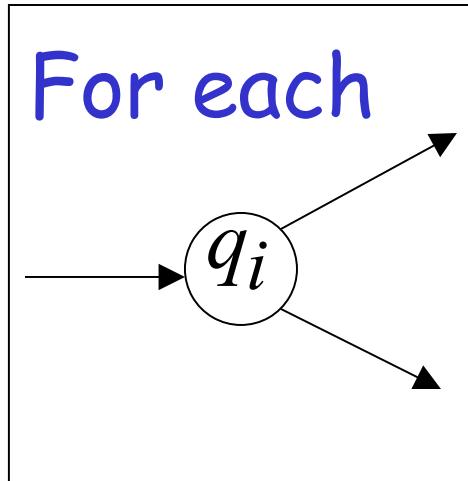


One accept state

Do the following for each possible halting state:

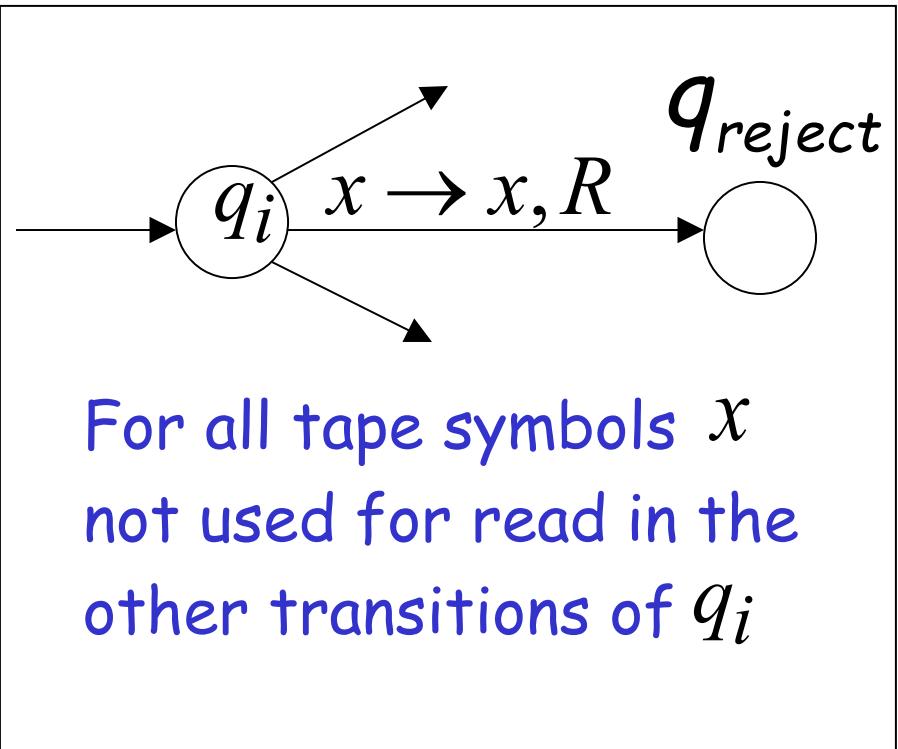
Old machine

For each



Multiple
reject states

New machine



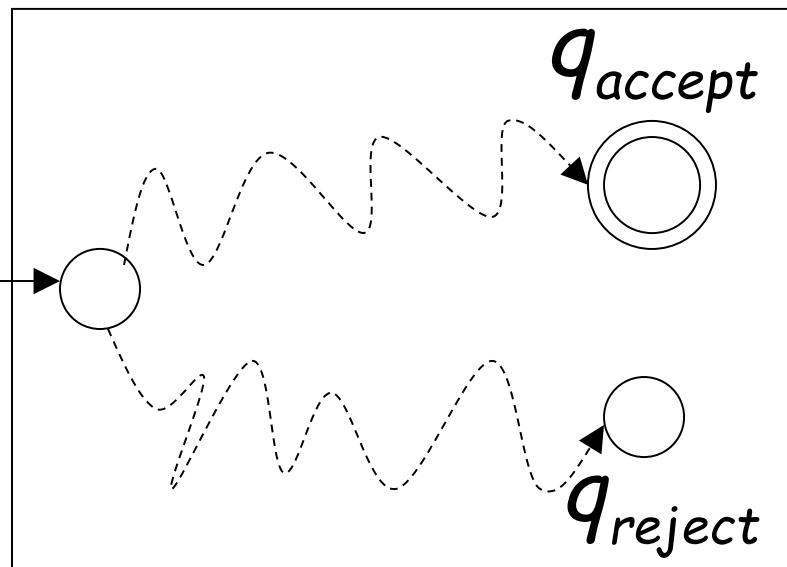
One reject state

For a decidable language L :

→ loop goes to T.M.

Decider for L

Input string



Decision
On Halt:

Accept

Reject

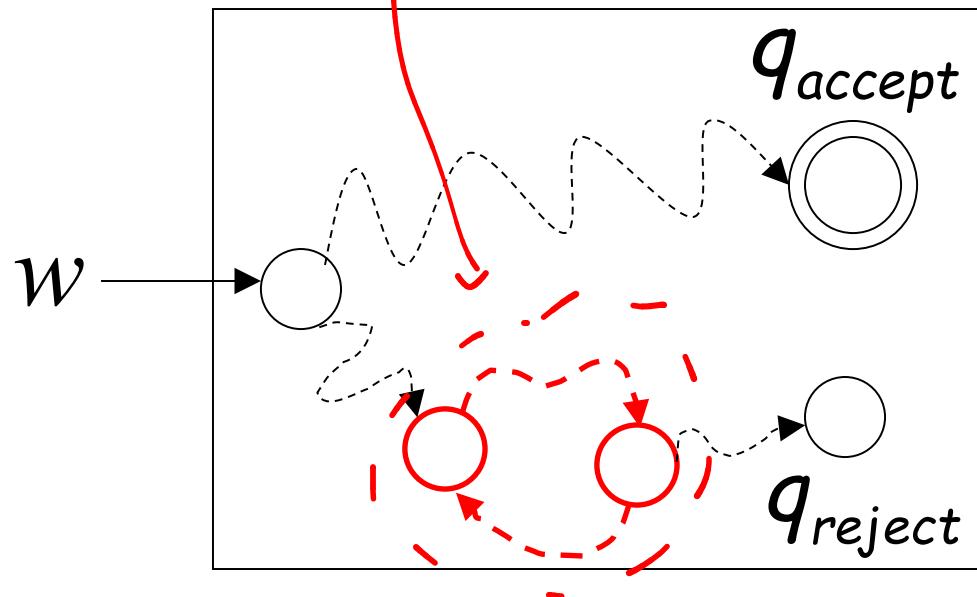
For each input string, the computation halts in the accept or reject state

For a Turing-Acceptable language L :

↓
Loop 0abil.

Turing Machine for L

Input
string



It is possible that for some input string
the machine enters an infinite loop

A computational problem is decidable
if the corresponding language is decidable

We also say that the problem is **solvable**

Problem: Is number x prime?

Corresponding language:

$$PRIMES = \{ 2, 3, 5, 7, \dots \}$$

We will show it is decidable

Decider for PRIMES :

On input number x :

Divide x with all possible numbers
between 2 and \sqrt{x}

If any of them divides x

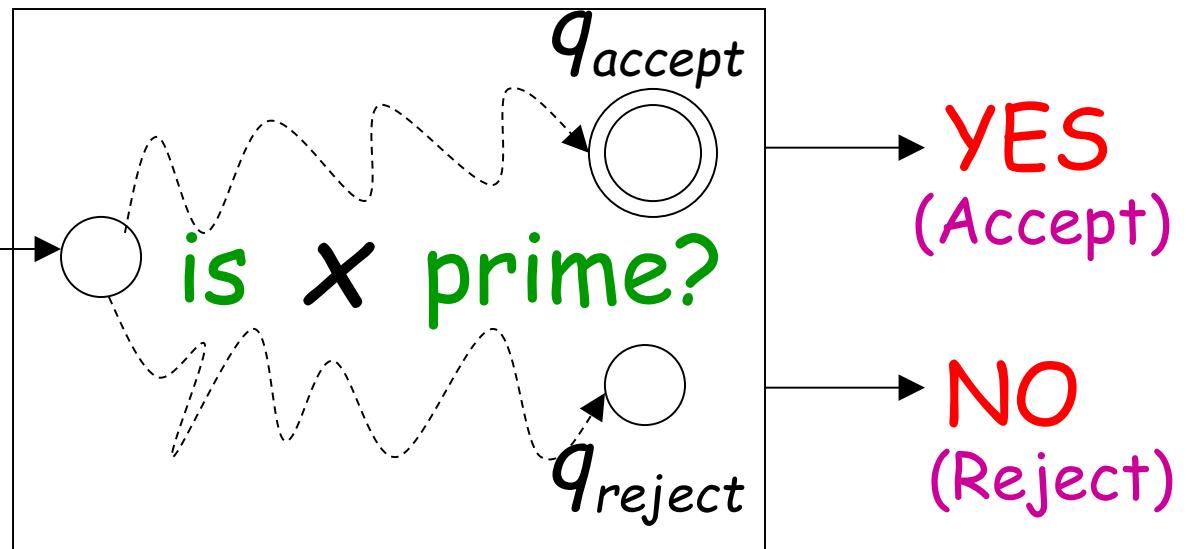
Then reject

Else accept

the decider Turing machine can be designed based on the algorithm

Decider for PRIMES

Input number x
(Input string)



Problem: Does DFA M accept
the empty language $L(M) = \emptyset?$

EMPTY
Problem

Corresponding Language: (Decidable)

$\text{EMPTY}_{\text{DFA}} =$

$\{\langle M \rangle : M \text{ is a DFA that accepts empty language } \emptyset\}$

↑
 $\rightarrow \text{DFA in } \text{turing}$

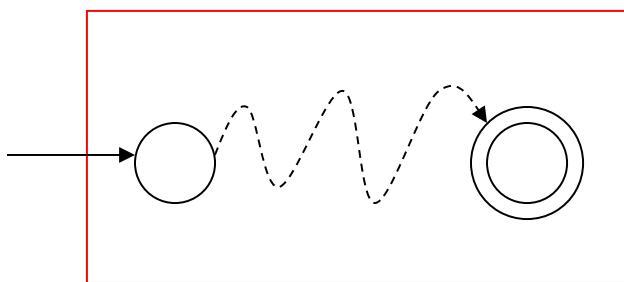
Description of DFA M as a string
(For example, we can represent M as a binary string, as we did for Turing machines)

Decider for $\text{EMPTY}_{\text{DFA}}$:

On input $\langle M \rangle$:

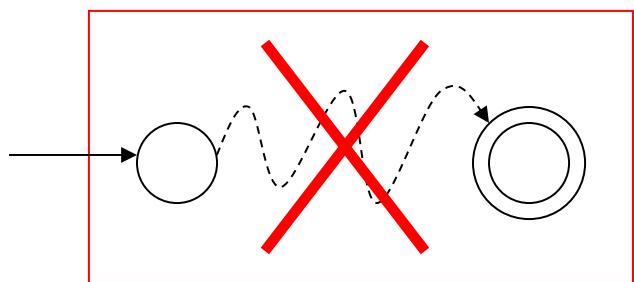
Determine whether there is a path from the initial state to any accepting state

DFA M



$$L(M) \neq \emptyset$$

DFA M



$$L(M) = \emptyset$$

Decision: **Reject $\langle M \rangle$**

Accept $\langle M \rangle$

Problem: Does DFA M accept
a finite language?

Corresponding Language: (Decidable)

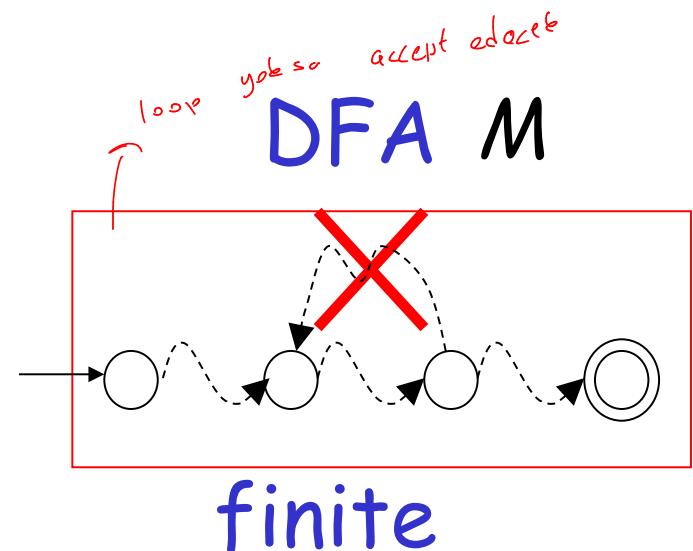
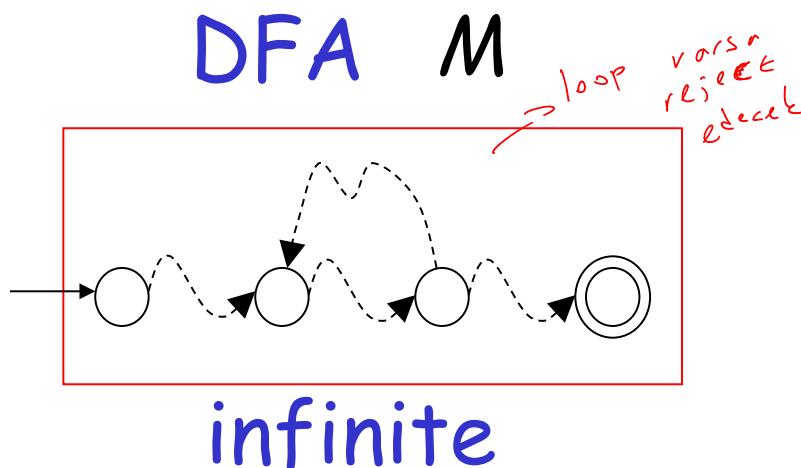
$\text{FINITE}_{\text{DFA}} =$

$\{\langle M \rangle : M \text{ is a DFA that accepts a finite language}\}$

Decider for $\text{FINITE}_{\text{DFA}}$:

On input $\langle M \rangle$:

Check if there is a walk with a cycle
from the initial state to an accepting state



Decision: **Reject $\langle M \rangle$**
(NO)

Accept $\langle M \rangle$
(YES)

Problem: Does DFA M accept string w ?

Corresponding Language: (Decidable)

$A_{DFA} = \{ \langle M, w \rangle : M \text{ is a DFA that accepts string } w \}$

Decider for A_{DFA} :

On input string $\langle M, w \rangle$:

Run DFA M on input string w

If M accepts w

Then accept $\langle M, w \rangle$ (and halt)

Else reject $\langle M, w \rangle$ (and halt)

Problem: Do DFAs M_1 and M_2 accept the same language?

Corresponding Language: (Decidable)

$EQUAL_{DFA} = \{ \langle M_1, M_2 \rangle : M_1 \text{ and } M_2 \text{ are DFAs that accept the same languages} \}$

$\{ \langle M_1, M_2 \rangle : M_1 \text{ and } M_2 \text{ are DFAs that accept the same languages} \}$

Decider for $EQUAL_{DFA}$:

On input $\langle M_1, M_2 \rangle$:

Let L_1 be the language of DFA M_1

Let L_2 be the language of DFA M_2

Construct DFA M such that:

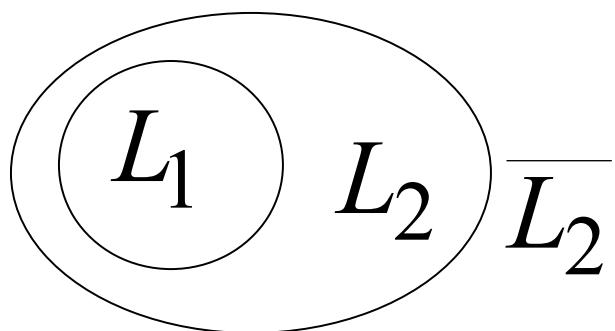
$$L(M) = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$$

(combination of DFAs)

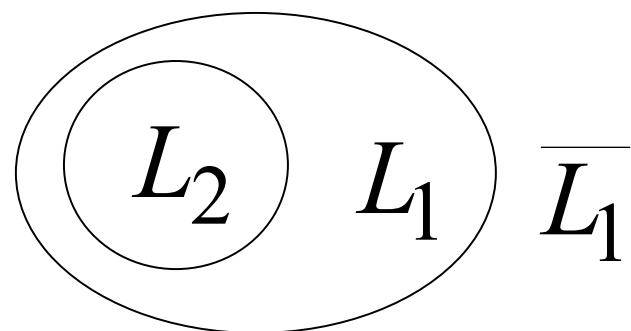
$$(L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2) = \emptyset$$



$$L_1 \cap \overline{L_2} = \emptyset \quad \text{and} \quad \overline{L_1} \cap L_2 = \emptyset$$



$$L_1 \subseteq L_2$$



$$L_2 \subseteq L_1$$



$$L_1 = L_2$$

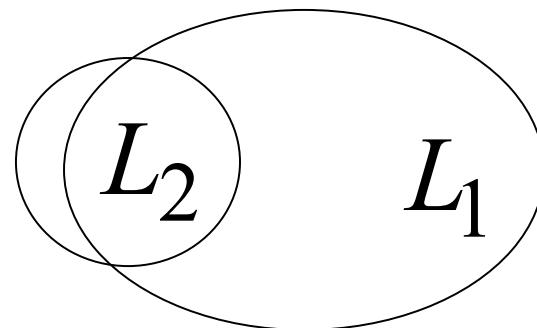
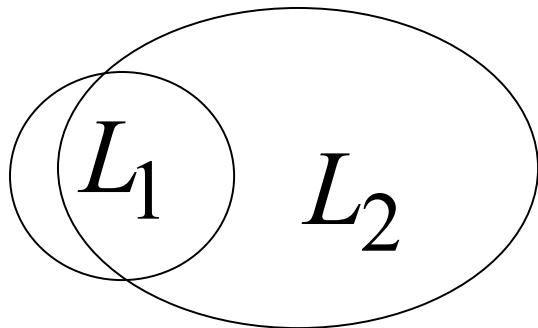
$$(L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2) \neq \emptyset$$



$$L_1 \cap \overline{L_2} \neq \emptyset$$

or

$$\overline{L_1} \cap L_2 \neq \emptyset$$



$$L_1 \not\subset L_2$$

$$L_2 \not\subset L_1$$



$$L_1 \neq L_2$$

Therefore, we only need
to determine whether

$$L(M) = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2) = \emptyset$$

which is a solvable problem for DFAs:

$\text{EMPTY}_{\text{DFA}}$

Theorem:

If a language L is decidable,
then its complement \bar{L} is decidable too



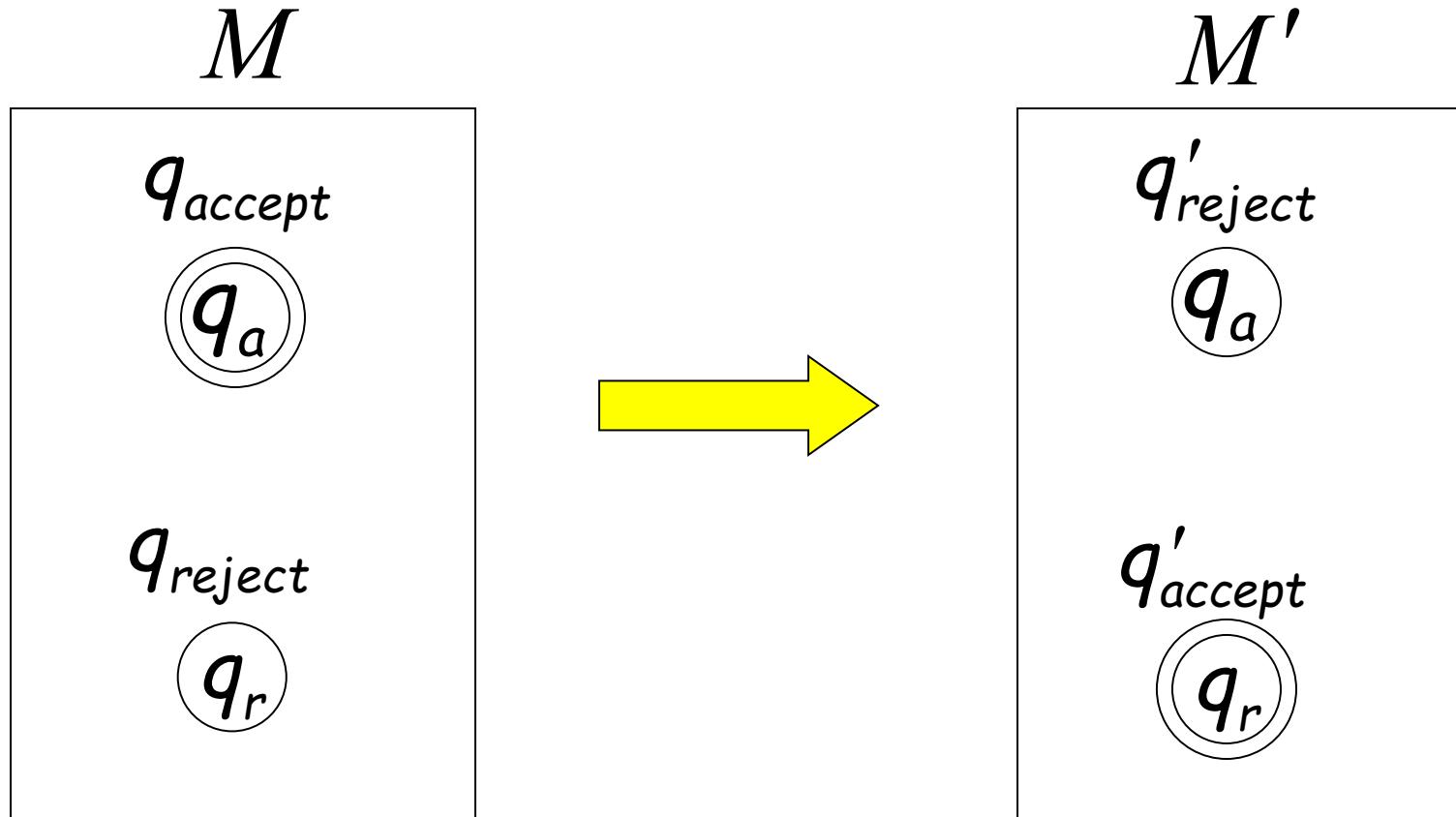
Accept
Statement: Accept \bar{L} using L

Proof:

Build a Turing machine M' that
accepts \bar{L} and halts on every input string

(M' is decider for \bar{L})

Transform accept state to reject and vice-versa



Turing Machine M'

On each input string w do:

1. Let M be the decider for L
2. Run M with input string w
 - If M accepts then reject
 - If M rejects then accept

Accepts \overline{L} and halts on every input string

Undecidable Languages

Undecidable Languages

An undecidable language has no decider:

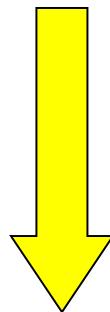
Any Turing machine that accepts L
does not halt on some input string

We will show that:

There is a language which is
Turing-Acceptable and undecidable

We will prove that there is a language L :

- L is Turing-acceptable
- \overline{L} is **not** Turing-acceptable
(not accepted by any Turing Machine)



the complement of a
decidable language is decidable

Therefore, L is undecidable

Non Turing-Acceptable \overline{L}

Turing-Acceptable L

Decidable

Consider alphabet $\{a\}$

Strings of $\{a\}^+$:

$a, aa, aaa, aaaa, \dots$

$a^1 \quad a^2 \quad a^3 \quad a^4 \quad \dots$

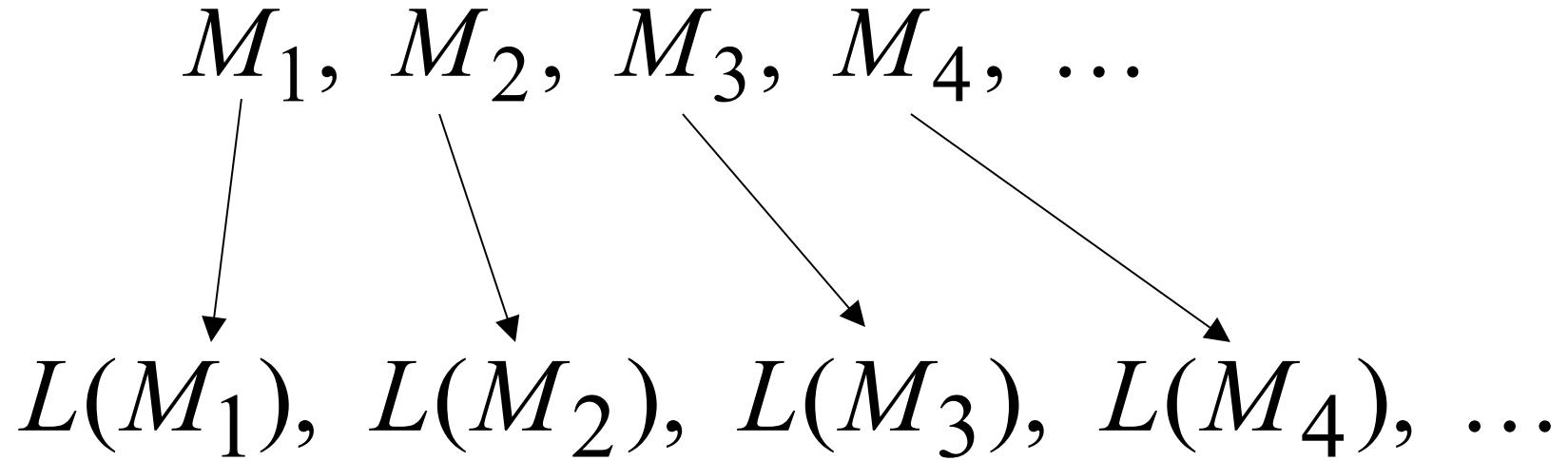
Consider Turing Machines
that accept languages over alphabet $\{a\}$

They are countable:

$M_1, M_2, M_3, M_4, \dots$

(There is an enumerator that generates them)

Each machine accepts some language over $\{a\}$



Note that it is possible to have

$$L(M_i) = L(M_j) \quad \text{for} \quad i \neq j$$

Since, a language could be accepted by more than one Turing machine

Example language accepted by M_i

$$L(M_i) = \{aa, aaaa, aaaaaaa\}$$

$$L(M_i) = \{a^2, a^4, a^6\}$$

Binary representation

	a^1	a^2	a^3	a^4	a^5	a^6	a^7	\dots
$L(M_i)$	0	1	0	1	0	1	0	\dots

Example of binary representations

	a^1	a^2	a^3	a^4	...
$L(M_1)$	0	1	0	1	...
$L(M_2)$	1	0	0	1	...
$L(M_3)$	0	1	1	1	...
$L(M_4)$	0	0	0	1	...

Consider the language

$$L = \{a^i : a^i \in L(M_i)\}$$

L consists of the 1's in the diagonal

	a^1	a^2	a^3	a^4	\dots
$L(M_1)$	0	1	0	1	\dots
$L(M_2)$	1	0	0	1	\dots
$L(M_3)$	0	1	1	1	\dots
$L(M_4)$	0	0	0	1	\dots



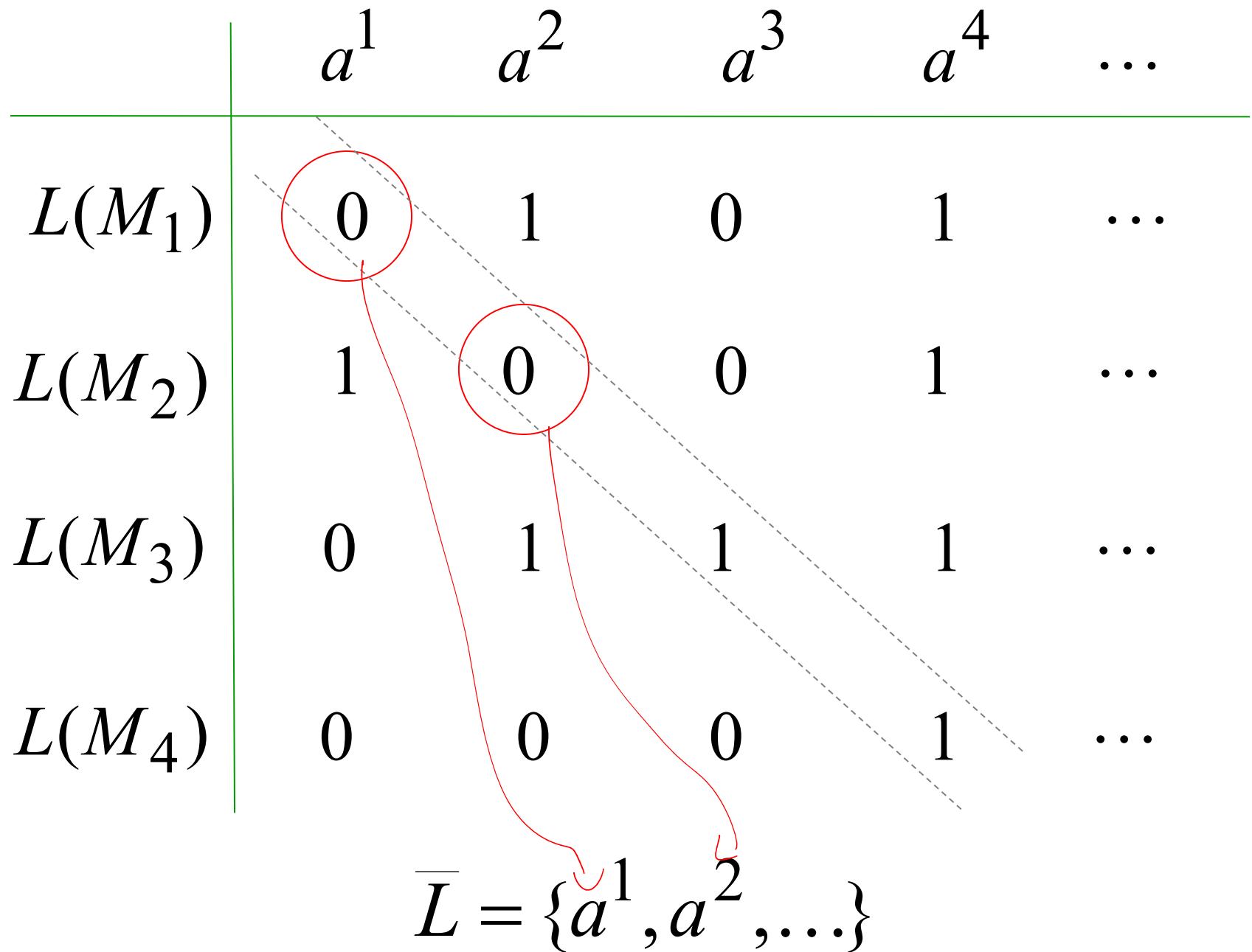
 $L = \{a^3, a^4, \dots\}$

Consider the language \overline{L}

$$\overline{L} = \{a^i : a^i \notin L(M_i)\}$$

$$L = \{a^i : a^i \in L(M_i)\}$$

\overline{L} consists of the 0's in the diagonal



Theorem:

Language \overline{L} is not Turing-Acceptable

Proof:

Assume for contradiction that

\overline{L} is Turing-Acceptable

Let M_k be the Turing machine
that accepts \overline{L} : $L(M_k) = \overline{L}$

	a^1	a^2	a^3	a^4	\dots
$L(M_1)$	0	1	0	1	\dots
$L(M_2)$	1	0	0	1	\dots
$L(M_3)$	0	1	1	1	\dots
$L(M_4)$	0	0	0	1	\dots
$L(M_k) = \overline{L} = 1100\dots$					

Question: $M_k = M_1$?

	a^1	a^2	a^3	a^4	\dots
$L(M_1)$	0	1	0	1	\dots
$L(M_2)$	1	0	0	1	\dots
$L(M_3)$	0	1	1	1	\dots
$L(M_4)$	0	0	0	1	\dots
$L(M_k) = \bar{L} =$	1	1	0	0	\dots

$$L(M_k) = \bar{L} = \underline{1}100\dots \quad a^1 \in L(M_k)$$

Answer: $M_k \neq M_1$

$a^1 \notin L(M_1)$

	a^1	a^2	a^3	a^4	\dots
$L(M_1)$	0	1	0	1	\dots
$L(M_2)$	1	0	0	1	\dots
$L(M_3)$	0	1	1	1	\dots
$L(M_4)$	0	0	0	1	\dots
$L(M_k) = \overline{L} = 1100\dots$					

Question: $M_k = M_2$?

	a^1	a^2	a^3	a^4	\dots
$L(M_1)$	0	1	0	1	\dots
$L(M_2)$	1	0	0	1	\dots
$L(M_3)$	0	1	1	1	\dots
$L(M_4)$	0	0	0	1	\dots

$$L(M_k) = \bar{L} = 1\textcircled{1}00\dots \quad a^2 \in L(M_k)$$

Answer: $M_k \neq M_2$

$a^2 \notin L(M_2)$

	a^1	a^2	a^3	a^4	\dots
$L(M_1)$	0	1	0	1	\dots
$L(M_2)$	1	0	0	1	\dots
$L(M_3)$	0	1	1	1	\dots
$L(M_4)$	0	0	0	1	\dots
$L(M_k) = \overline{L} = 1100\dots$					

Question: $M_k = M_3$?

	a^1	a^2	a^3	a^4	\dots
$L(M_1)$	0	1	0	1	\dots
$L(M_2)$	1	0	0	1	\dots
$L(M_3)$	0	1	1	1	\dots
$L(M_4)$	0	0	0	1	\dots

$$L(M_k) = \bar{L} = 11\textcircled{0}0\dots \quad a^3 \notin L(M_k)$$

Answer: $M_k \neq M_3$

$$a^3 \in L(M_3)$$

Similarly: $M_k \neq M_i$ for any i

Because either:

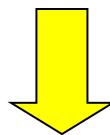
$$a^i \in L(M_k)$$

or

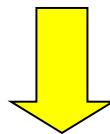
$$a^i \notin L(M_i)$$

$$a^i \notin L(M_k)$$

$$a^i \in L(M_i)$$



the machine M_k cannot exist



\overline{L} is not Turing-Acceptable

End of Proof

Non Turing-Acceptable

\overline{L}

Turing-Acceptable

Decidable

We will prove that the language

$$L = \{a^i : a^i \in L(M_i)\}$$

Is Turing-Acceptable

Brendan

Yogi

Sorras

Undecidable

There is a
Turing machine
that accepts L

Each machine
that accepts L
doesn't halt
on some input string

Theorem: The language

$$L = \{a^i : a^i \in L(M_i)\}$$

Is Turing-Acceptable

Proof: We will give a Turing Machine that
accepts L

Turing Machine that accepts L

For any input string w

- Suppose $w = a^i$
- Find Turing machine M_i
(using the enumerator for Turing Machines)
- Simulate M_i on input string a^i
- If M_i accepts, then accept w

End of Proof

Therefore:

Turing-Acceptable

$$L = \{a^i : a^i \in L(M_i)\}$$

Not Turing-acceptable

$$\overline{L} = \{a^i : a^i \notin L(M_i)\}$$

Non Turing-Acceptable \overline{L}

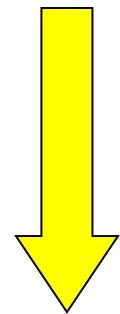
Turing-Acceptable L

Decidable

$L ?$

Theorem: $L = \{a^i : a^i \in L(M_i)\}$
is undecidable

Proof: If L is decidable



the complement of a
decidable language is decidable

Then \overline{L} is decidable

However, \overline{L} is not Turing-Acceptable!
Contradiction!!!!

Not Turing-Acceptable \overline{L}

Turing-Acceptable L

Decidable



BLM2502

The Theory of Computation

Spring 2022

Lecturer: Dr. Oğuz Altun

Email: oaltun@yildiz.edu.tr

Office: D036

Office Hours: Tuesday 12:00-13:00

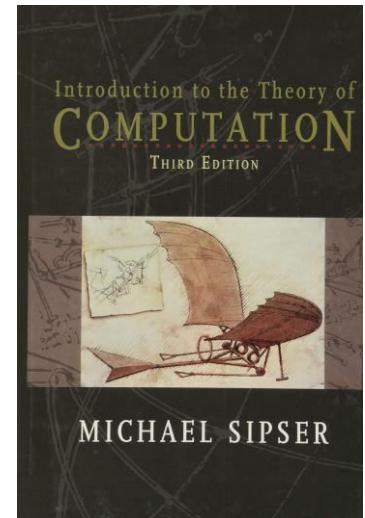
» **Course: BLM 2502 The theory of Computation**

- > **Tuesday 09:00-12:00**
- > **ECTS 3 credits**

BLM2502 Theory of Computation

» Book

- > Michael Sipser, **Introduction to the Theory of Computation (3E)**, Thomson
- > No handouts, its your responsibily to take notes.



- > Lectures from the author:
<https://ocw.mit.edu/courses/mathematics/18-404j-theory-of-computation-fall-2020/>

» **Course Goals:**

- » To introduce fundamental theoretical concepts for the computational complexity of algorithmic problems, and also to introduce basic techniques used for lexical analysis and syntax parsing in programming languages.

» **Main Topics Covered:**

- » Finite Automata
 - Deterministic and nondeterministic finite automata
 - Regular expressions
- » Pushdown Automata
 - Context-free languages and grammars
 - Parsing
- » Turing Machines
 - Turing recognizable languages
 - Decidability
 - Time and Space complexity
 - NP-completeness

» **Grading:**

- » Homework assignments: 15%
- » Pop-up quizzes (~8 quizzes): 15%
- » Midterm: 30%
- » Final: 40%



BLM2502

Theory of Computation

Spring 2017

BLM2502 Theory of Computation

» Book

- > Michael Sipser, Introduction to the Theory of Computation (3E), Thomson
- > No handouts, its your responsibility to take notes.

BLM2502 Theory of Computation

- » In this Theory of Computation course we will try to answer the following questions:
- » What are the mathematical properties of computer hardware and software?
- » What is a computation and what is an algorithm? Can we give rigorous mathematical definitions of these notions?
- » What are the limitations of computers? Can “everything” be computed? (As we will see, the answer to this question is “no”).
- » Purpose of the Theory of Computation:
Develop formal mathematical models of computation that reflect real-world computers.

BLM2502 Theory of Computation

- » Complexity Theory
- » The main question asked in this area is “What makes some problems computationally hard and other problems easy?”
- » Informally, a problem is called “easy”, if it is efficiently solvable. Examples of “easy” problems are
 - > Sorting a sequence of, e.g, 1,000,000 numbers,
 - > Searching for a name in a telephone directory
 - > Computing the fastest way to drive from Esenler to Beşiktaş.

BLM2502 Theory of Computation

- » Complexity Theory
- » On the other hand, a problem is called “hard”, if it cannot be solved efficiently, or if we don’t know whether it can be solved efficiently. Examples of “hard” problems are
 - > Time table scheduling for all courses
 - > Factoring a 300-digit integer into its prime factors
 - > Computing a layout for chips in VLSI.
- » Central Question in Complexity Theory:
Classify problems according to their degree of “difficulty”. Give a rigorous proof that problems that seem to be “hard” are really “hard”.

BLM2502 Theory of Computation

- » Computability Theory
- » In the 1930's, scientists discovered that some of the fundamental mathematical problems cannot be solved by a "computer". (computers were invented in 1940s).
For example: "Is an arbitrary mathematical statement true or false?"
- » To attack such a problem, we need formal definitions of the notions of
 - > computer
 - > algorithm
 - > computation

BLM2502 Theory of Computation

- » Computability Theory
- » The theoretical models that were proposed in order to understand solvable and unsolvable problems led to the development of real computers.
- » Central Question in Computability Theory:
Classify problems as being solvable or unsolvable.

BLM2502 Theory of Computation

- » Automata Theory
- » Automata Theory deals with definitions and properties of different types of “computation models”. Examples :
 - > Finite Automata. These are used in text processing, compilers, and hardware design.
 - > Context-Free Grammars. These are used to define programming languages and in Artificial Intelligence.
 - > Turing Machines. These form a simple abstract model of a “real” computer, such as your PC at home.
- » Central Question in Automata Theory:

Do these models have the same power, or can one model solve more problems than the other?

BLM2502 Theory of Computation

» Set Theory

- » A set is a collection of well-defined objects.
 - > The set of **natural numbers** is $\mathbf{N} = \{1, 2, 3, \dots\}$.
 - > The set of **integers** is $\mathbf{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$.
 - > The set of **rational numbers** is $\mathbf{Q} = \{m/n : m \in \mathbf{Z}, n \in \mathbf{Z}, n \neq 0\}$.
 - + What is irrational numbers?
 - > The set of **real numbers** is denoted by \mathbf{R} .
 - > If A and B are sets, then A is a **subset** of B , written as $A \subseteq B$, if every element of A is also an element of B .
 - > If B is a set, then the **power set** $P(B)$ of B is defined to be the set of all subsets of B :
 - ❖ $P(B) = \{A : A \subseteq B\}$.
 - ❖ Observe $\emptyset \in P(B)$ and $B \in P(B)$.

BLM2502 Theory of Computation

» Set Theory

- > If A and B are two sets, then
 - + their union is defined as
 - $A \cup B = \{x : x \in A \text{ or } x \in B\}$,
 - + their intersection is defined as
 - $A \cap B = \{x : x \in A \text{ and } x \in B\}$,
 - + their difference is defined as
 - $A \setminus B = \{x : x \in A \text{ and } x \notin B\}$,
 - + the Cartesian product of A and B is defined as
 - $A \times B = \{(x, y) : x \in A \text{ and } y \in B\}$,
 - + the complement of A is defined as
 - $\bar{A} = \{x : x \notin A\}$.

BLM2502 Theory of Computation

L, O, O O, I I, I, I O, O I

» Set Theory

- > A **binary relation** on two sets A and B is a subset of $A \times B$.
- > A **function** f from A to B, denoted by $f : A \rightarrow B$, is a binary relation R , having the property that for each element $a \in A$, there is exactly one ordered pair in R , whose first component is a. We will also say that $f(a) = b$, or f maps a to b, or the image of a under f is b. The set A is called the **domain** of f, and the set $\{b \in B : \text{there is an } a \in A \text{ with } f(a) = b\}$ is called the **range** of f.
- > A binary relation $R \subseteq A \times A$ is an **equivalence relation**, if it satisfies the following three conditions:
 - + R is **reflexive**: For every element in $a \in A$, we have $(a, a) \in R$.
 - + R is **symmetric**: For all a and b in A, if $(a, b) \in R$, then $(b, a) \in R$.
 - + R is **transitive**: For all a, b, and c in A, if $(a, b) \in R$ and $(b, c) \in R$, then also $(a, c) \in R$.

BLM2502 Theory of Computation

- » Boolean Logic
- » The Boolean values are 1 and 0, that represent true and false, respectively. The basic Boolean operations:
- » **negation** (or NOT), represented by \neg ,
- » **conjunction** (or AND), represented by Λ ,
- » **disjunction** (or OR), represented by \vee ,
- » **exclusive-or** (or XOR), represented by \oplus ,
- » **equivalence**, represented by \leftrightarrow or \Leftrightarrow ,
- » **implication**, represented by \rightarrow or \Rightarrow .

BLM2502 Theory of Computation

» Truth Table (0=false, 1=true)

NOT	AND	OR	XOR	equivalence	implication
$\neg 0 = 1$	$0 \wedge 0 = 0$	$0 \vee 0 = 0$	$0 \oplus 0 = 0$	$0 \Leftrightarrow 0 = 1$	$0 \Rightarrow 0 = 1$
$\neg 1 = 0$	$0 \wedge 1 = 0$	$0 \vee 1 = 1$	$0 \oplus 1 = 1$	$0 \Leftrightarrow 1 = 0$	$0 \Rightarrow 1 = 1$
	$1 \wedge 0 = 0$	$1 \vee 0 = 1$	$1 \oplus 0 = 1$	$1 \Leftrightarrow 0 = 0$	$1 \Rightarrow 0 = 0$
	$1 \wedge 1 = 1$	$1 \vee 1 = 1$	$1 \oplus 1 = 0$	$1 \Leftrightarrow 1 = 1$	$1 \Rightarrow 1 = 1$

» Implication (if ... then ...)

> antecedent (condition)->consequence (promise)

» E.g.

> p: "you take out the trash".

> q:"you get a dollar"

> $p \Rightarrow q$ is false only if you take out the trash but don't get a dollar.

BLM2502 Theory of Computation

» Proof Techniques

- » In mathematics, a **theorem** is a statement that is true. A **proof** is a sequence of mathematical statements that form an argument to show that a theorem is true.
 - > **Axioms**: assumptions about the underlying mathematical structures
 - > **Hypotheses**: a supposition or proposed explanation made based on limited evidence as a starting point for further investigation.
 - > **Theorem**: described above
 - > **Lemmas**: previously proved theorems
 - > **Corollaries**: Special cases of theorem
- » There is no specified way of producing a proof, but there are some generic strategies that could be of help.

BLM2502 Theory of Computation

» Proof Techniques

» Tips:

- > **Read** and completely understand the statement of the theorem to be proved. Most often this is the hardest part. **Rewrite** the statement in your own words.
- > Sometimes, theorems contain theorems inside them. For example, “Property A if and only if property B”, requires showing **two statements**:
 - + (a) If property A is true, then property B is true ($A \rightarrow B$).
 - + (b) If property B is true, then property A is true ($B \rightarrow A$).
- > Another example is the theorem “Set A equals set B.” To prove this, we need to prove that $A \subseteq B$ and $B \subseteq A$. That is, we need to show that each element of set A is in set B, and that each element of set B is in set A.

BLM2502 Theory of Computation

» Proof Techniques

» Tips:

- > Try to **work out a few simple cases** of the theorem just to get a grip on it (i.e., crack a few simple cases first).
- > Try to **write down the proof** once you have it. This is to ensure the correctness of your proof. Often, mistakes are found at the time of writing.
- > Finding proofs takes time, we do not come prewired to produce proofs. **Be patient**, think, express and write clearly and try to be precise as much as possible.

BLM2502 Theory of Computation

» Proof Techniques

- > Direct Proofs or Constructive Proofs or Proof by Construction
- > Nonconstructive Proofs
- > Proofs by Contradiction
- > Proofs by Induction
- > Pigeon Principle

BLM2502 Theory of Computation

- » Proof Techniques - Direct Proofs
- » As the name suggests, in a direct proof of a theorem, we just approach the theorem directly.
- » **Theorem:** If n is an odd positive integer, then n^2 is odd as well.
- » **Proof:** An odd positive integer n can be written as:
 $n = 2k + 1$, for some integer $k \geq 0$. Then:
 - ❖ $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$.
 - ❖ Since $2(2k^2 + 2k)$ is even, and “even plus one is odd”, we can conclude that
 - ❖ n^2 is odd.

BLM2502 Theory of Computation

» Proof Techniques - Constructive Proofs

(Proof By Construction)

- » Many theorems state that a particular type of object exists
- » One way to prove is to find a way to construct one such object
- » This technique is called proof by construction
- » **Theorem:** There exists a rational number p which can be expressed as a^b , with a and b both irrational.

A constructive proof of the above theorem on irrational powers of irrationals would give an actual example, such as:

$$a = \sqrt{2}, \quad b = \log_2 9, \quad a^b = 3.$$

The square root of 2 is irrational, and 3 is rational. $\log_2 9$ is also irrational: if it were equal to $\frac{m}{n}$, then, by the properties of logarithms, 9^n would be equal to 2^m , but the former is odd, and the latter is even.

BLM2502 Theory of Computation

- » Proof Techniques - **Proof by Contradiction**
- » The proof by contradiction is grounded in the fact that **any proposition must be either true or false, but not both true and false at the same time.**
- » One common way to prove a theorem is to assume that the theorem is false, and then show that this assumption leads to an obviously false consequence (also called a contradiction)
- » This type of reasoning is used frequently in everyday life.

BLM2502 Theory of Computation

- » Proof Techniques - Proof by Contradiction
- » Let us define a number is rational if it can be expressed as p/q where p and q are integers; if it cannot, then the number is called irrational.
- » **Theorem:** $\sqrt{2}$ (the square root of 2) is irrational.
- » **Proof:** Assume that $\sqrt{2}$ is rational. Then, it can be written as p/q for some positive integers p and q such that **p and q does not have a common factor.**
- » Then, we have $p^2/q^2 = 2$, or $2q^2 = p^2$
- » (continued in next page)

BLM2502 Theory of Computation

» Proof Techniques - Proof by Contradiction

- » Since $2q^2$ is an even number, p^2 is also an even number
- » This implies that **p is an even number** (why?)
- » So, $p = 2r$ for some integer r , and so, $2q^2 = p^2 = (2r)^2 = 4r^2$
- » This implies $2r^2 = q^2$
- » **So, q is an even number**
- » Something wrong happens!.. We now have:
 - » “p and q does not have common factor”
 - » AND
 - » “**p and q have common factor**”
 - » **This is a contradiction**
 - » Thus, the assumption is wrong, so that $\sqrt{2}$ is irrational

BLM2502 Theory of Computation

- » Proof Techniques - **Proof by Induction**
- » For each positive integer n , let $P(n)$ be a mathematical statement that depends on n . Assume we wish to prove that $P(n)$ is true for all positive integers n . A proof by induction of such a statement is carried out as follows:
- » **Basis:** Prove that $P(1)$ is true.
- » **Induction step:** Prove that for all $n \geq 1$, the following holds: If $P(n)$ is true, then $P(n + 1)$ is also true.
- » In the induction step, we choose an arbitrary integer $n \geq 1$ and assume that $P(n)$ is true; this is called the induction hypothesis. Then we prove that $P(n + 1)$ is also true.

BLM2502 Theory of Computation

- » Proof Techniques - **Proof by Induction**

- » **Theorem:** For all positive integers n , we have

$$1 + 2 + 3 + \dots + n = n(n + 1)/2$$

- » **Proof:** Start with the basis of the induction. If $n = 1$, then the left-hand side is equal to 1, and so is the right-hand side. So the theorem is true for $n = 1$.
- » For the induction step, let $n \geq 1$ and assume that the theorem is true for n , i.e., assume that

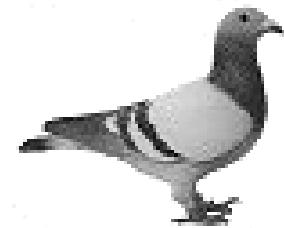
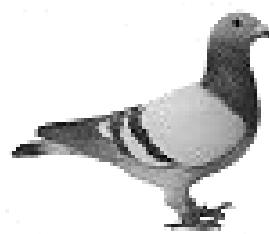
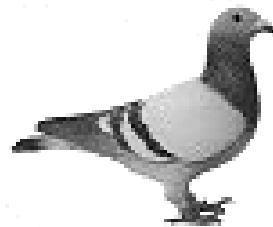
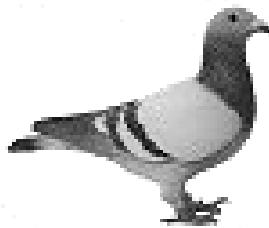
$$1 + 2 + 3 + \dots + n = n(n + 1)/2$$

- » We have to prove that the theorem is true for $n + 1$

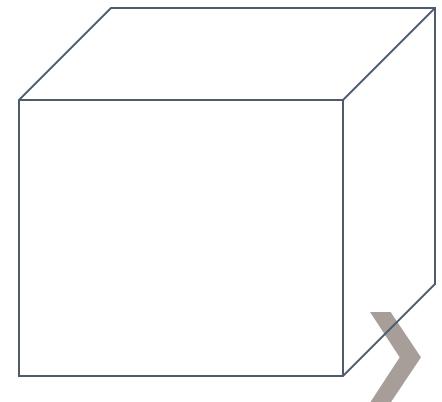
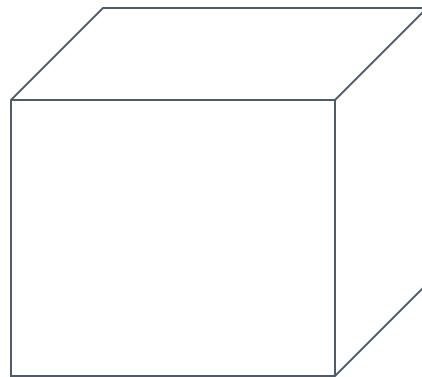
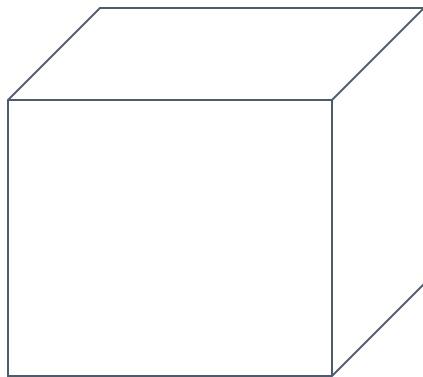
BLM2502 Theory of Computation

- » Proof Techniques – **Pigeonhole Principle**
- » If $n + 1$ or more objects are placed into n boxes, then there is at least one box containing two or more objects.
- » In other words, if A and B are two sets such that $|A| > |B|$, then there is no one-to-one function from A to B .

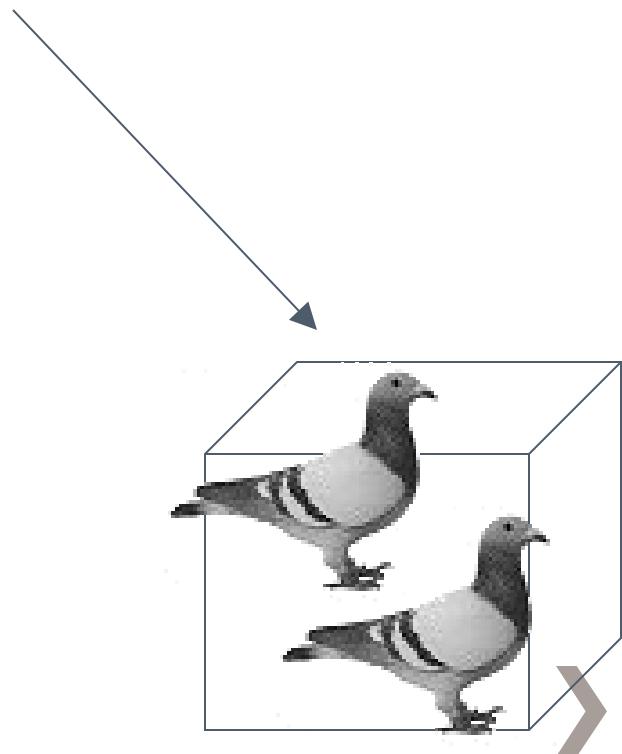
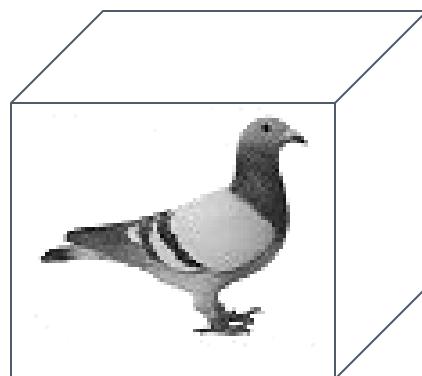
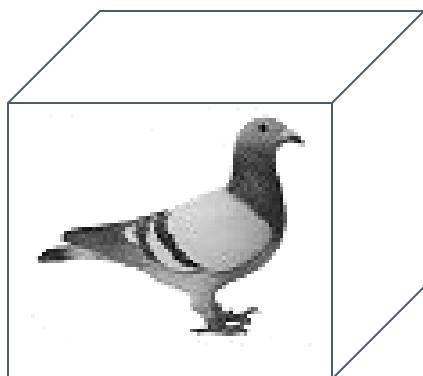
4 pigeons



3 pigeonholes



A pigeonhole must
contain at least two pigeons



- » Proof techniques: Pigeonhole Principle.
- » Example: Prove that if seven distinct numbers are selected from $\{1, 2, \dots, 11\}$, then some two of these numbers sum to 12
- » All numbers from 1 to 11 can be put into following **6 pigeonholes**: $\{1, 11\}, \{2, 10\}, \{3, 9\}, \{4, 8\}, \{5, 7\}, \{6\}$.
- » We select **7** distinct numbers (**pigeons**). First 6 pigeons can be put to different pigeonholes. But after that we have to put to an existing pigeonhole. The pigeonhole of 6 can hold only one pigeon 😊.

BLM2502 Theory of Computation

- » Proof Techniques – Pigeonhole Principle
- » **Theorem:** Let n be a positive integer. Every sequence of $n^2 + 1$ distinct natural numbers contains a subsequence of length $n + 1$ that is either increasing or decreasing.
- » **Proof:** For example consider the sequence

(8, 11, 9, 1, 4, 6, 12, 10, 5, 7)

of $10 = 3^2 + 1$ numbers. This sequence contains a decreasing subsequence of length $4 = 3 + 1$, shown. There are other subsequences of length 4, too.

Proof: Let $a_1, a_2, \dots, a_{n^2+1}$ be a sequence of $n^2 + 1$ distinct real numbers. Associate an ordered pair with each term of the sequence, namely, associate (i_k, d_k) to the term a_k , where i_k is the length of the longest increasing subsequence starting at a_k , and d_k is the length of the longest decreasing subsequence starting at a_k .

Suppose that there are no increasing or decreasing subsequences of length $n + 1$. Then i_k and d_k are both positive integers less than or equal to n , for $k = 1, 2, \dots, n^2 + 1$. Hence, by the product rule there are n^2 possible ordered pairs for (i_k, d_k) . By the pigeonhole principle, two of these $n^2 + 1$ ordered pairs are equal. In other words, there exist terms a_s and a_t , with $s < t$ such that $i_s = i_t$ and $d_s = d_t$. We will show that this is impossible. Because the terms of the sequence are distinct, either $a_s < a_t$ or $a_s > a_t$. If $a_s < a_t$, then, because $i_s = i_t$, an increasing subsequence of length $i_t + 1$ can be built starting at a_s , by taking a_s followed by an increasing subsequence of length i_t beginning at a_t . This is a contradiction. Similarly, if $a_s > a_t$, the same reasoning shows that d_s must be greater than d_t , which is a contradiction. 

Sequence=(8,11,9,1,4,6,12,10,5,7)

$$a_1 = 8, (i_1 = 3, d_1 = 3)$$

$$a_2 = 11, (i_2 = 2, d_2 = 4)$$

$$a_3 = 9, (i_3 = 2, d_3 = 3)$$

...

If there are no sequences of length $n + 1$, there are only n^2 pairs. Then since we have $n^2 + 1$ distinct numbers. At least 2 pairs are equal by pigeonhole principle. But this is not possible as numbers are **distinct**.



BLM2502

Theory of Computation

Spring 2017

BLM2502 Theory of Computation

» Course Outline

» Week	Content
» 1	Introduction to Course
» 2	Computability Theory, Complexity Theory, Automata Theory, Set Theory, Relations, Proofs, Pigeonhole Principle
» 3	Regular Languages
» 4	Finite Automata
» 5	Deterministic and Nondeterministic Finite Automata
» 6	Epsilon Transition, Equivalence of Automata
» 7	Pumping Theorem
»	
» 9	Context Free Grammars
» 10	Parse Tree, Ambiguity,
» 11	Pumping Theorem
» 13	Turing Machines, Recognition and Computation, Church-Turing Hypothesis
» 14	Turing Machines, Recognition and Computation, Church-Turing Hypothesis
» 15	Review



Regular Expressions

BLM2502 Theory of Computation

Regular Languages

» Keywords / Definitions:

- > **Alphabet:** A finite nonempty set of symbols. The members of the alphabet are the **symbols** of the alphabet. We generally use capital Greek letters Σ and Γ to designate alphabets. The following are a few examples of alphabets.

$$\Sigma_1 = \{0,1\}$$

$$\Sigma_2 = \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\}$$

$$\Gamma = \{0, 1, x, y, z\}$$

- > A **string** over an alphabet is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas. If $\Sigma_1 = \{0,1\}$, then 01101 is a string over Σ_1 . If $\Sigma_2 = \{a, b, c, \dots, z\}$, then abracadabra is a string over Σ_2 .

BLM2502 Theory of Computation

Regular Expressions

» Keywords / Definitions:

> If w is a string over Σ , the **length** of w , written $|w|$, is the number of symbols that it contains. The string of length zero is called the **empty string** and is written as ϵ (or λ). The empty string plays the role similar to 0 in a number system.

> If w has length n , we can write

$w = w_1 w_2 \dots w_n$ where each $w_i \in \Sigma$.

$$abc = w$$

$$cba = w^R$$

The reverse of w , written as w^R , is the string obtained by writing w in the opposite order (i.e., $w_n w_{n-1} \dots w_1$).

String z is a **substring** of w if z appears consecutively within w . For example, cad is a substring of abracadabra.

BLM2502 Theory of Computation

Regular Expressions



» Keywords / Definitions:

- > If we have string x of length m and string y of length n , the **concatenation** of x and y , written xy , is the string obtained by appending y to the end of x , as in $x_1 \dots x_m y_1 \dots y_n$. To concatenate a string with itself many times we use the superscript notation:

$$x^3 = xxx, \text{ } \downarrow \text{ times}$$

$$(x^n) = xx \dots x \text{ (n times)}$$

$$\begin{array}{l} x = abc \\ y = hello \end{array}$$

$$xy = abc hello$$

To indicate all possible recurrences, a special superscript (in fact a special operator) is used:

$$* \text{ (kleene star)} \rightarrow x^* = \{ x^0, x^1, \dots, x^n \}$$

- > **Language**: A set of strings (finite or infinite?) over an alphabet. Languages are used to describe computation problems.

$$L_1 = \{ 01, \infty, 11 \} \quad \Sigma_1 = \{ 0, 1 \}$$

BLM2502 Theory of Computation

Regular Expressions

» Keywords / Definitions:

$$A^* = \{1_1, 2_2, \dots\}$$

regular or not.

> Best way is using *Finite Automata*. If some finite automata recognizes the language, then the language is regular.

> Regular (set) operations are used to build up regular sets:

> Union, concatenation, and kleene star operations are as follows.

$$A = \{1_1, 2_2\}$$

> *Union*: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.

$$B = \{ab, aa, ba\}$$

< > *Concatenation*: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.

$$AB = A \circ B = \{1ab, 1aa, 1ba, 2ab, 2aa, 2ba, 22ab, 22aa, 22ba\}$$

> *Star*: $A^* = \{x_1 x_2 \dots x_k \mid k > 0 \text{ and each } x_i \in A\}$.

BLM2502 Theory of Computation



Regular Expressions

- » Keywords / Definitions:
 - > Class of regular languages is closed under **union**
 - > Class of regular languages is closed under **intersection**
 - > Class of regular languages is closed under **complement**
 - > Class of regular languages is closed under **concatenation**
 - > Class of regular languages is closed under (kleene) **star**

BLM2502 Theory of Computation

Alphabet: $= \{a, b\}$

Strings:

a

ab

$abba$

$aaabbbaabab$

$u = ab$

$v = bbbaaa$

$w = abba$

BLM2502 Theory of Computation

Decimal numbers

Alphabet: $\Sigma = \{0,1,2,\dots,9\}$

Strings:

102345 567463386 ...

Binary numbers

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

Strings:

100010001 101101111 ...

BLM2502 Theory of Computation

Unary numbers

Alphabet: $\Sigma = \{1\}$

Strings:

1 11 111111 ...

Decimal equivalent:

1,2,6...

BLM2502 Theory of Computation

» Examples on String Operations

$w = a_1a_2\dots a_n, v = b_1b_2\dots b_m \quad a, b \in \{x, y\}$

eg, $w = xyxy, v = xxxxyy$

Concenation:

$wv = a_1a_2\dots a_nb_1b_2\dots b_n$

eg, $xyxyxxxxyy$

Reverse:

$w^R = a_na_{n-1}\dots a_1$

eg, $yxyyx$

BLM2502 Theory of Computation

» Examples on String Operations

$w = a_1a_2\dots a_n, v = b_1b_2\dots b_m \quad a, b \in \{x, y\}$

Length:

$$|w| = n, |v| = m$$

eg, $|yxyyx| = 5; |xxxxyy| = 6$

$$|wv| = |w| + |v|$$

eg, $|yxyyxxxxyy| = 11$ (recall example above)

Empty String:

$$|\epsilon| = 0$$

BLM2502 Theory of Computation

» Examples on String Operations

$w = a_1a_2\dots a_n, v = b_1b_2\dots b_m \quad a,b \in \{x, y\}$

Substring:

a_1, a_2, \dots, a_n (each symbol of the string are its substrings)

$a_1, a_1a_2, a_1a_2a_3\dots$ (these are prefix substrings)

$a_2, a_2a_3, a_2a_3a_4, \dots$

$a_n, a_{n-1}a_n, a_{n-2}a_{n-1}a_n, \dots$ (these are suffix substrings)

special cases:

ϵ is prefix and suffix of each string

any string is prefix and suffix of itself

BLM2502 Theory of Computation

» Examples on String Operations

Special cases:

ϵ is prefix and suffix of each string

any string is prefix and suffix of itself

String: abba 

Prefix	Suffix
ϵ	abba
a	bba
ab	ba
abb	a
abba	ϵ

Observe that:
 $\epsilon w = w\epsilon = w$

BLM2502 Theory of Computation

» Examples on String Operations

$w = a_1a_2\dots a_n, v = b_1b_2\dots b_m \quad a, b \in \{x, y\}$

Self Concatenation:

$ww = w^2, www = w^3, \text{etc}$

$w^0 = \epsilon$

e.g., $w = abba$;

$(abba)^0 = \epsilon$

$(abba)^1 = abba$

$(abba)^2 = abbaabba$

$(abba)^3 = abbaabbaabba$

Kleene Star:

$w^* = \{w^0, w^1, w^2, w^3, \dots\}$

BLM2502 Theory of Computation

» The * operation:

The set of all possible strings from the alphabet

$$\Sigma = \{ \overset{\rightarrow}{\text{a}}, \overset{\rightarrow}{\text{b}} \}$$

$$\Sigma^* = \{ \varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots \} \rightarrow \text{it is infinite}$$

» The + operation:

The set of all possible strings from the alphabet excluding ε

$$\Sigma = \{ \overset{\rightarrow}{\text{a}}, \overset{\rightarrow}{\text{b}} \}$$

$$\Sigma^+ = \boxed{\Sigma^* - \{ \varepsilon \}} = \{ a, b, aa, ab, ba, bb, aaa, aab, \dots \}$$

BLM2502 Theory of Computation

» Language:

A Language over an alphabet $\Sigma = \{a, b\}$

is a subset of $\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$

Examples:

$$L_1 = \{\epsilon\}$$

$$L_2 = \{a, b, aa, ab, ba, bb\}$$

$$L_3 = \{\epsilon, a, aa, aaa, aaaa, \dots\}$$

$$L_4 = \{a^n b^n, n \geq 0\} = \{\epsilon, ab, aabb, aaabbbb, \dots\}$$

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\text{PRIME_NUMBERS} = \{x \mid x \in \Sigma^* \text{ and } x \text{ is prime}\} \\ = \{2, 3, 5, 7, 11, \dots\}$$

$$\text{EVEN_NUMBERS} = \{x \mid x \in \Sigma^* \text{ and } x \text{ is even}\} = \{0, 2, 4, \dots\}$$

$$\text{ODD_NUMBERS} = \{x \mid x \in \Sigma^* \text{ and } x \text{ is odd}\} = \{1, 3, 5, \dots\}$$

BLM2502 Theory of Computation

» Unary Addition

Alphabet $\Sigma = \{1, +, =\}$

ADDITION = $\{x + y = z : x = 1^m, y = 1^n, z = 1^k; k = m + n\}$

$111 + 11 = 11111 \in \text{ADDITION}$

$111 + 111 = 1111 \notin \text{ADDITION}$

» Squaring

Alphabet $\Sigma = \{1, \#\}$

SQUARES = $\{x \# y : x = 1^m, y = 1^n, n = m^2\}$

$111 \# 1111111111 \in \text{SQUARES}$

$1111 \# 111111111 \notin \text{SQUARES}$

BLM2502 Theory of Computation

» Operations On Languages

Set Operations: Regular sets are closed under union, intersection negation and complement.

$$A = \{a, ab, aaaa\}$$

$$B = \{ab, bb\}$$

$$A \cup B = \{a, ab, bb, aaaa\}$$

$$A \cap B = \{ab\}$$

$$A - B = \{a, \cancel{bb}, aaaa\}$$

$$\bar{A} = \Sigma^* - A = \{\overline{a}, \overline{ab}, \overline{aaa}\} = \{\epsilon, aa, ba, bb, aba, \dots\}$$

ϵ

↖ ↗

•

Note that :

$$\emptyset = \{\} \neq \{\epsilon\}$$

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

$$|\{\epsilon\}| = 1 \quad * \text{ This is set size}$$

$$|\epsilon| = 0 \quad * \text{ This is string length}$$

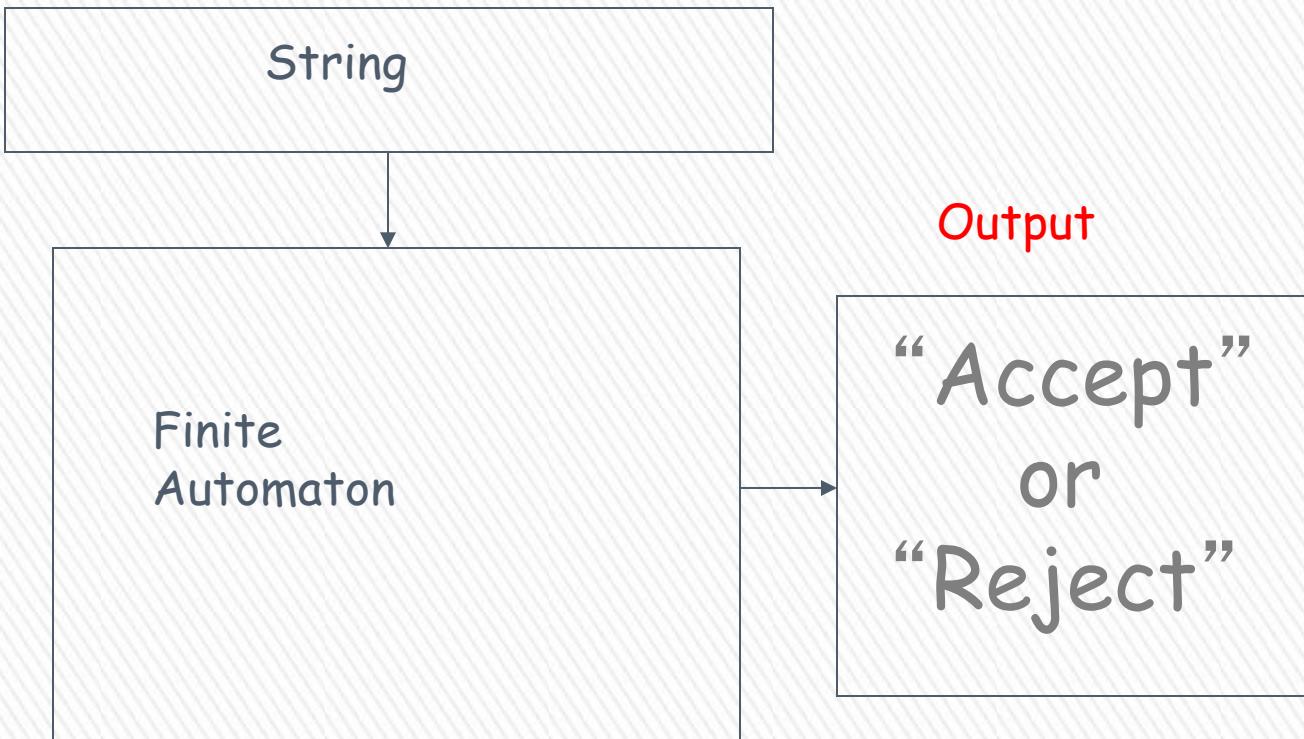


Finite Automata

BLM2502 Theory of Computation

»

Input Tape



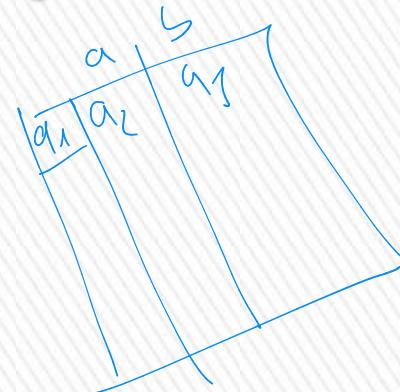
BLM2502 Theory of Computation

Finite Automata

» A finite automaton is a 5-tuple:

$(Q, \Sigma, \delta, q_0, F)$ where;

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \rightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and $q_f \in F = \{q_0, v\}$
5. $F \subseteq Q$ is the ***set of accept states***.



$$F = \{q_0, v\}$$

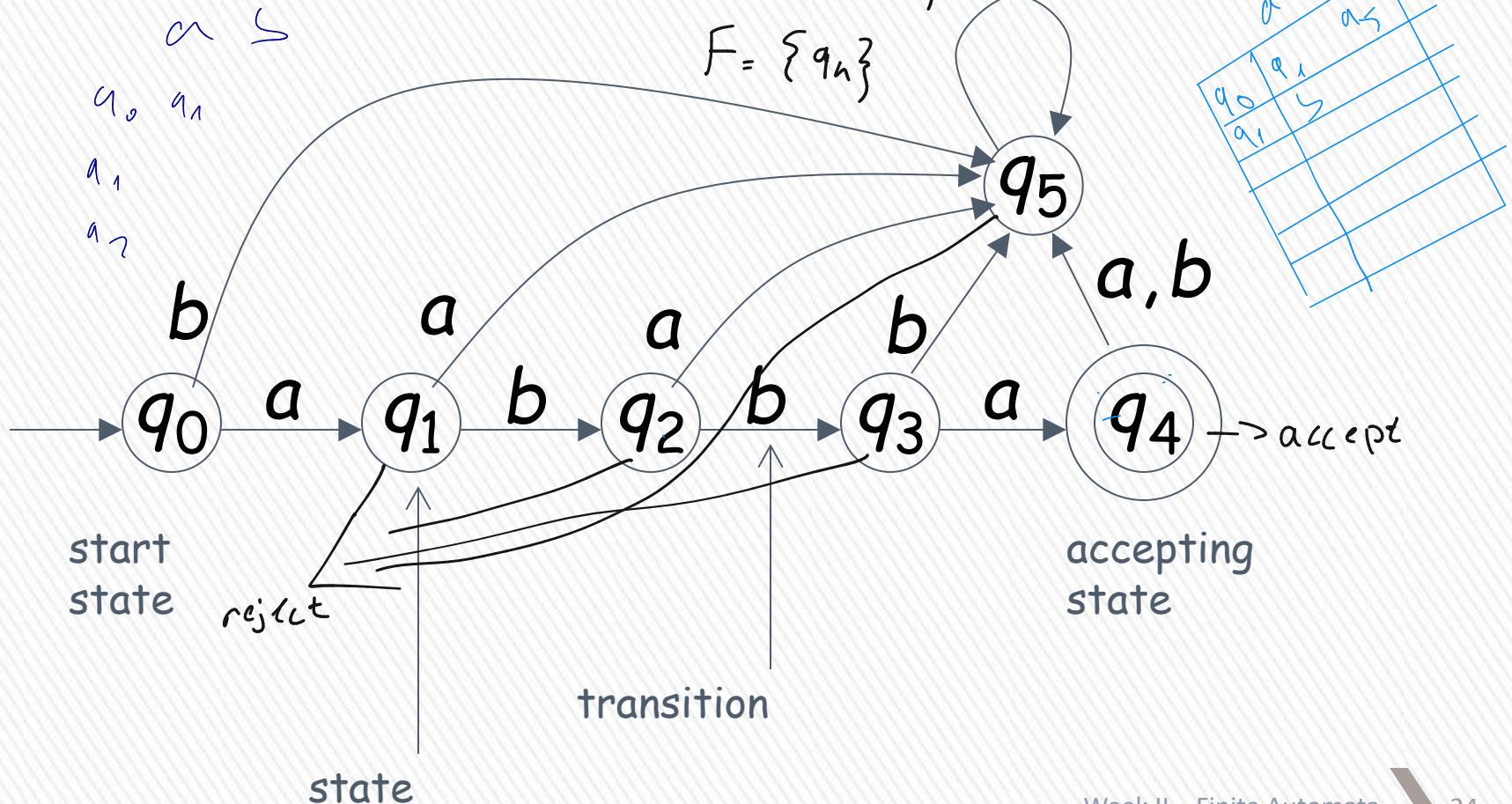
BLM2502 Theory of Computation

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

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

$$F = \{q_n\}$$

start state q_0



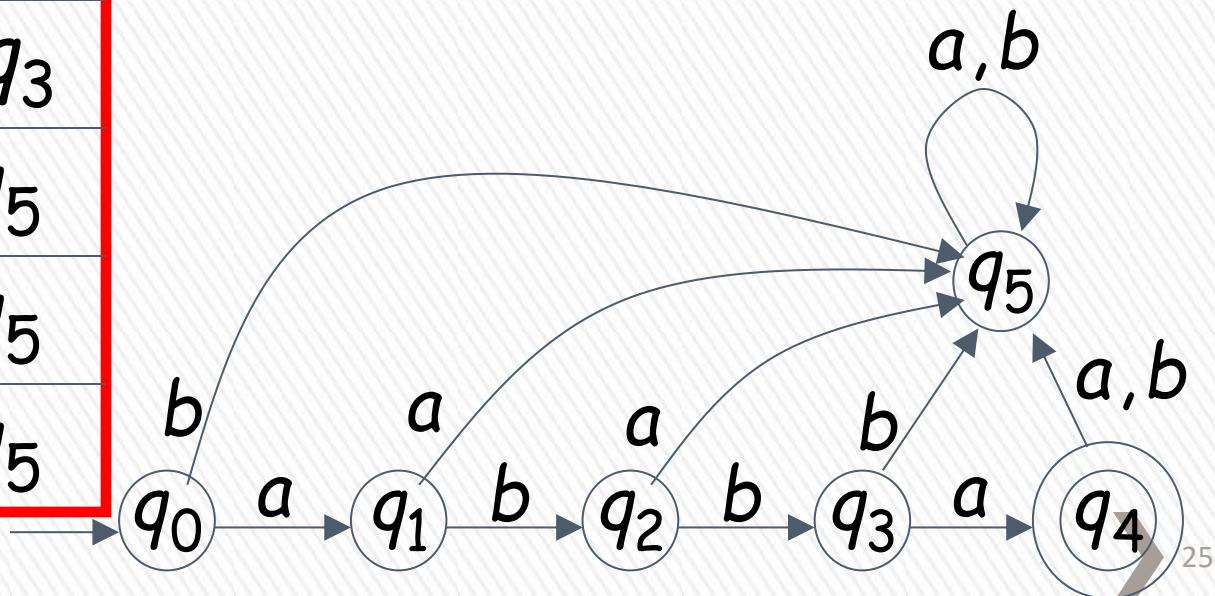
BLM2502 Theory of Computation

symbols

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

states

Transition Table



BLM2502 Theory of Computation

- » You can think of the transition function as being the “program” of the finite automaton M. This function tells us what M can do in “one step”:
 - » Let r be a state of Q and let a be a symbol of the alphabet Σ . If the finite automaton M is in state r and reads the symbol a , then it switches from state r to state $\delta(r, a)$. (In fact, $\delta(r, a)$ may be equal to r .)
- » Example:
 - » $A = \{w : w \text{ is a binary string containing an odd number of } 1\text{s}\}$.

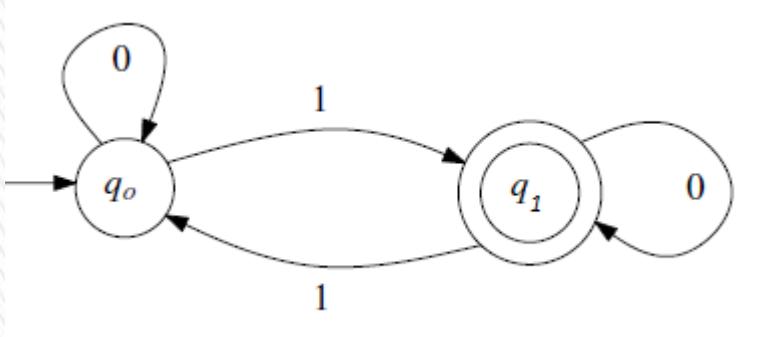
BLM2502 Theory of Computation

Design:

- » The finite automaton reads the input string w from left to right and keeps track of the number of 1s it has seen. After having read the entire string w , it checks whether this number is odd (in which case w is accepted) or even (in which case w is rejected).
- » Using this approach, the finite automaton needs a state for every integer $i \geq 0$, indicating that the number of 1s read so far is equal to i .

BLM2502 Theory of Computation

- » Design – Continued
- » However, this design is not feasible since FA have finite number of states.
- » ???
- » A better, and correct approach, is to keep track of whether the number of 1s read so far is even or odd.



BLM2502 Theory of Computation

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

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

$\Sigma = \{0, 1\}$ (trivial from the problem)

$$q_0 \in Q$$

$$F = \{q_1\}$$

δ :

$$(q_0, 0) \rightarrow q_0$$

$$(q_0, 1) \rightarrow q_1$$

$$(q_1, 0) \rightarrow q_1$$

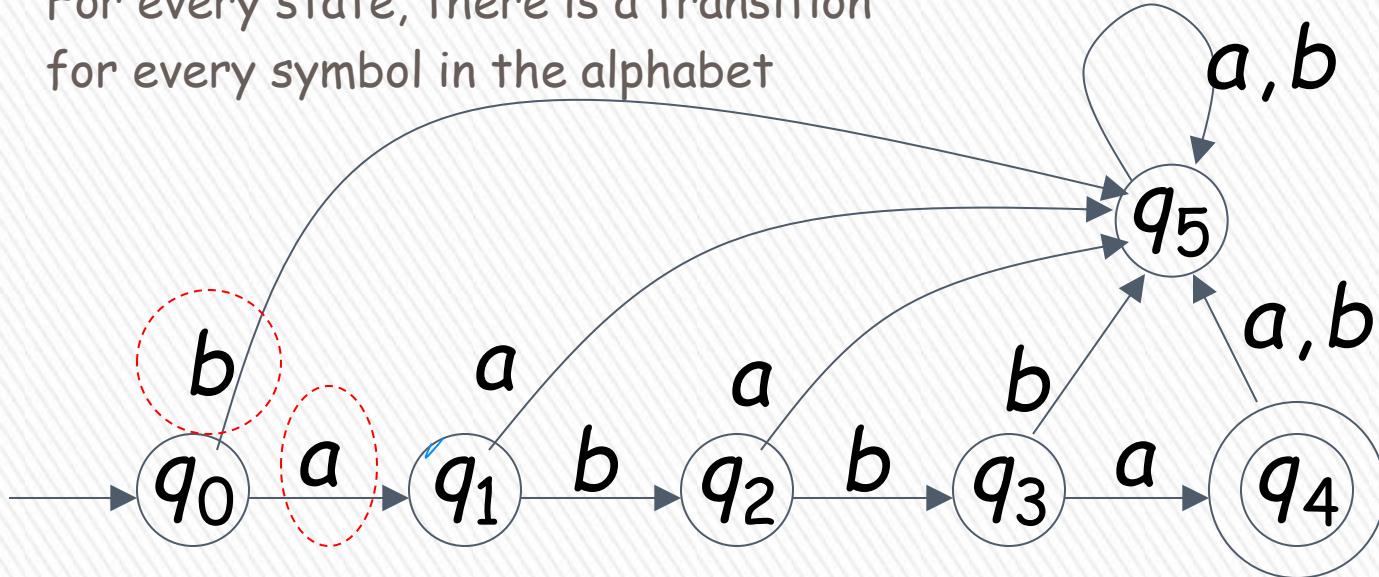
$$(q_1, 1) \rightarrow q_0$$

δ :	0	1
q ₀	q ₀	q ₁
q ₁	q ₁	q ₀

BLM2502 Theory of Computation

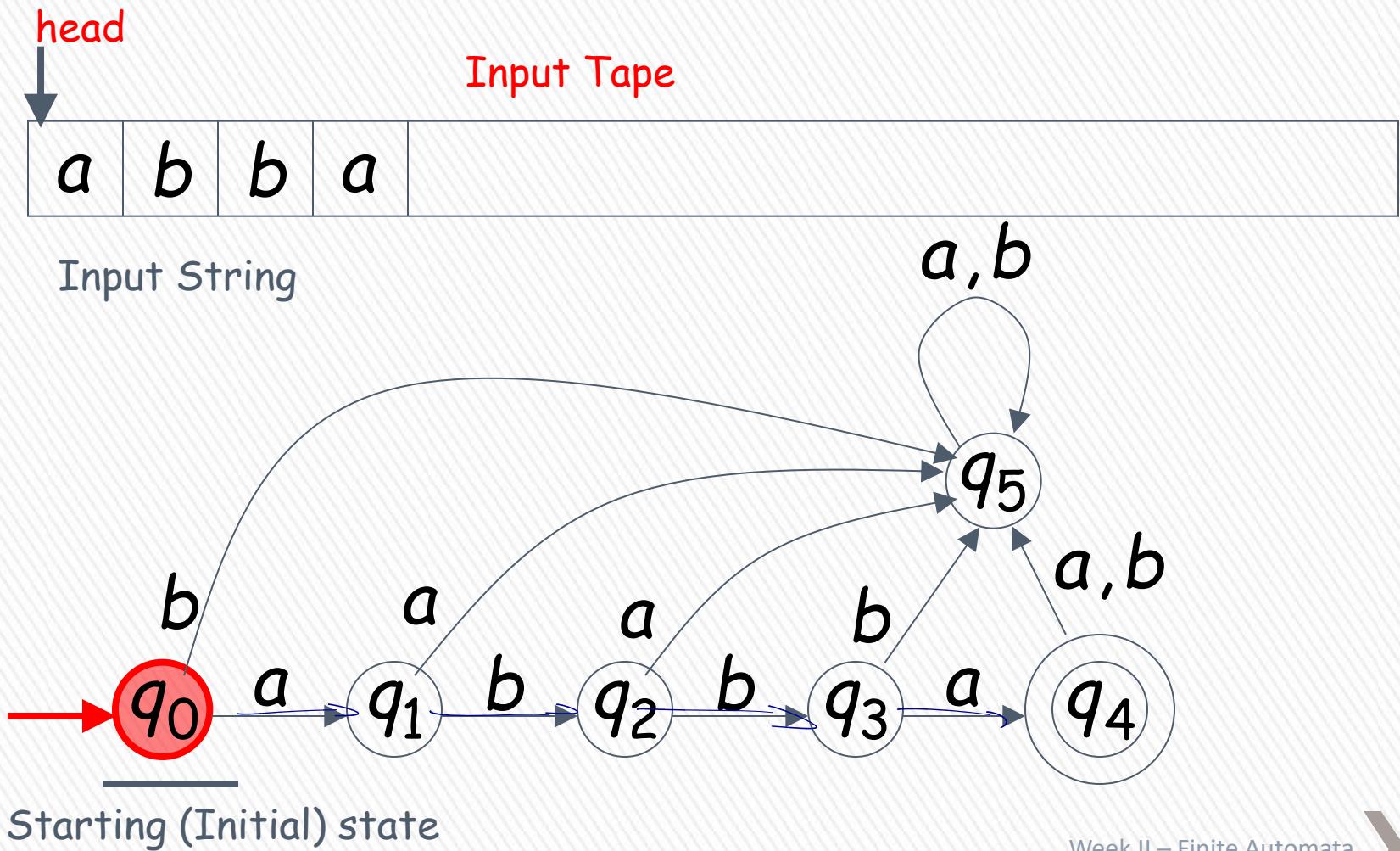
» DFA - Deterministic Finite Automata

For every state, there is a transition
for every symbol in the alphabet

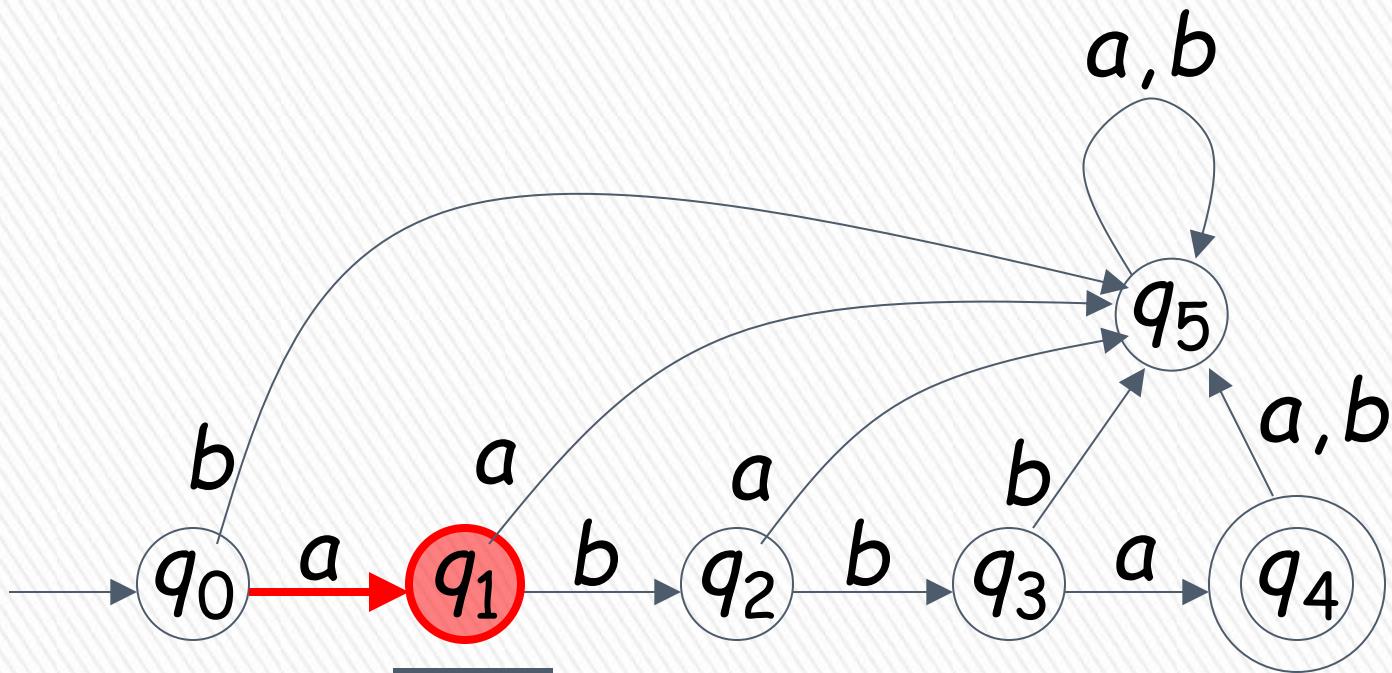


Alphabet $\Sigma = \{a, b\}$

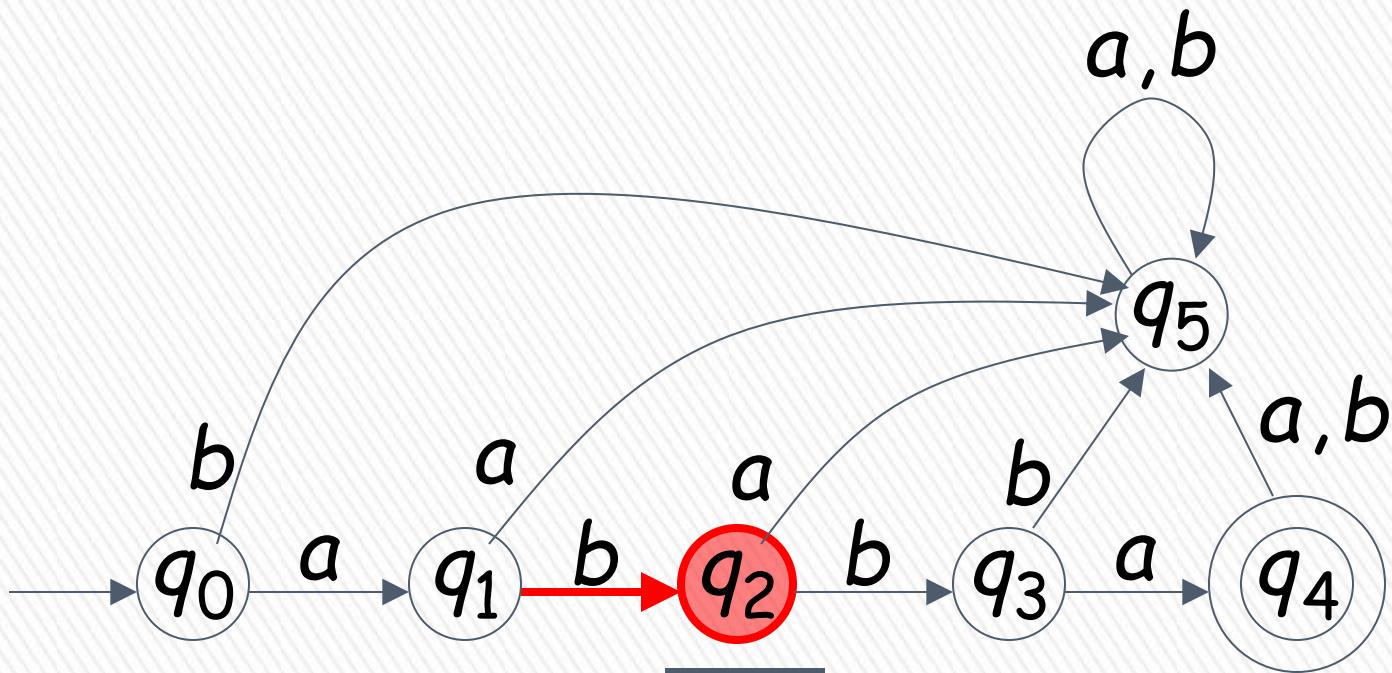
BLM2502 Theory of Computation



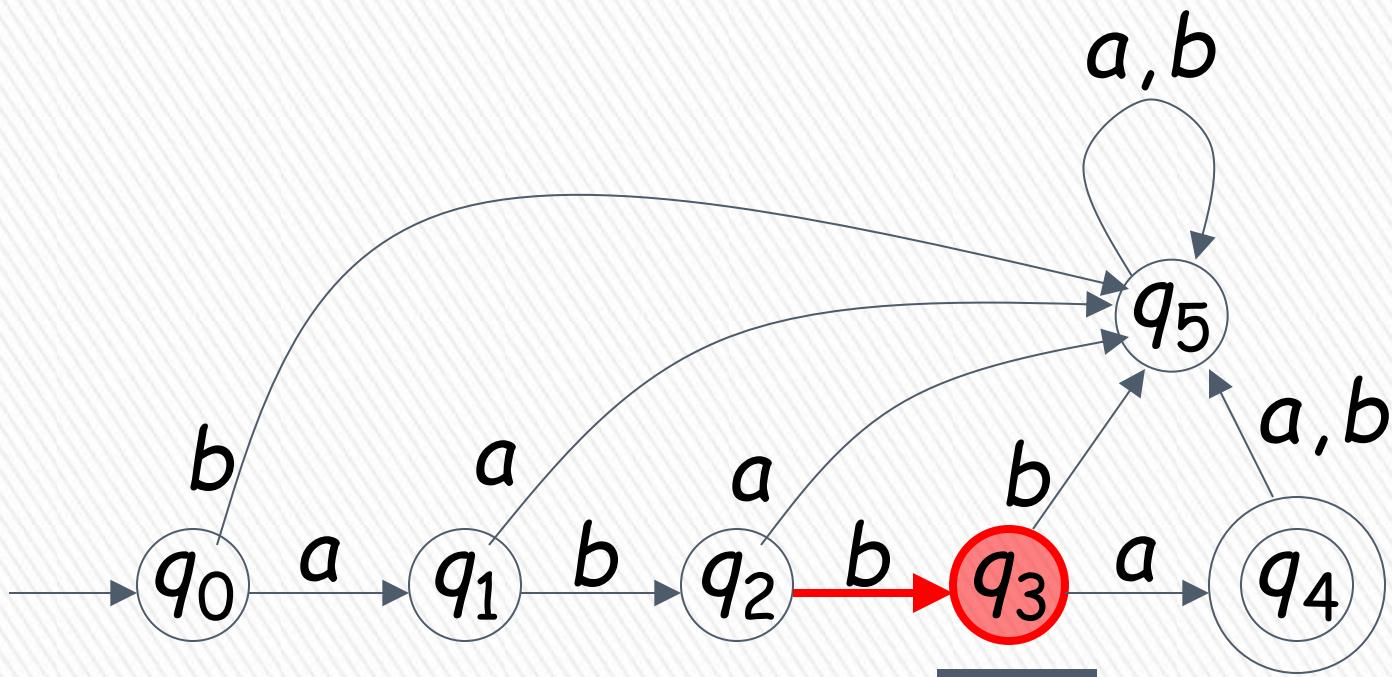
BLM2502 Theory of Computation



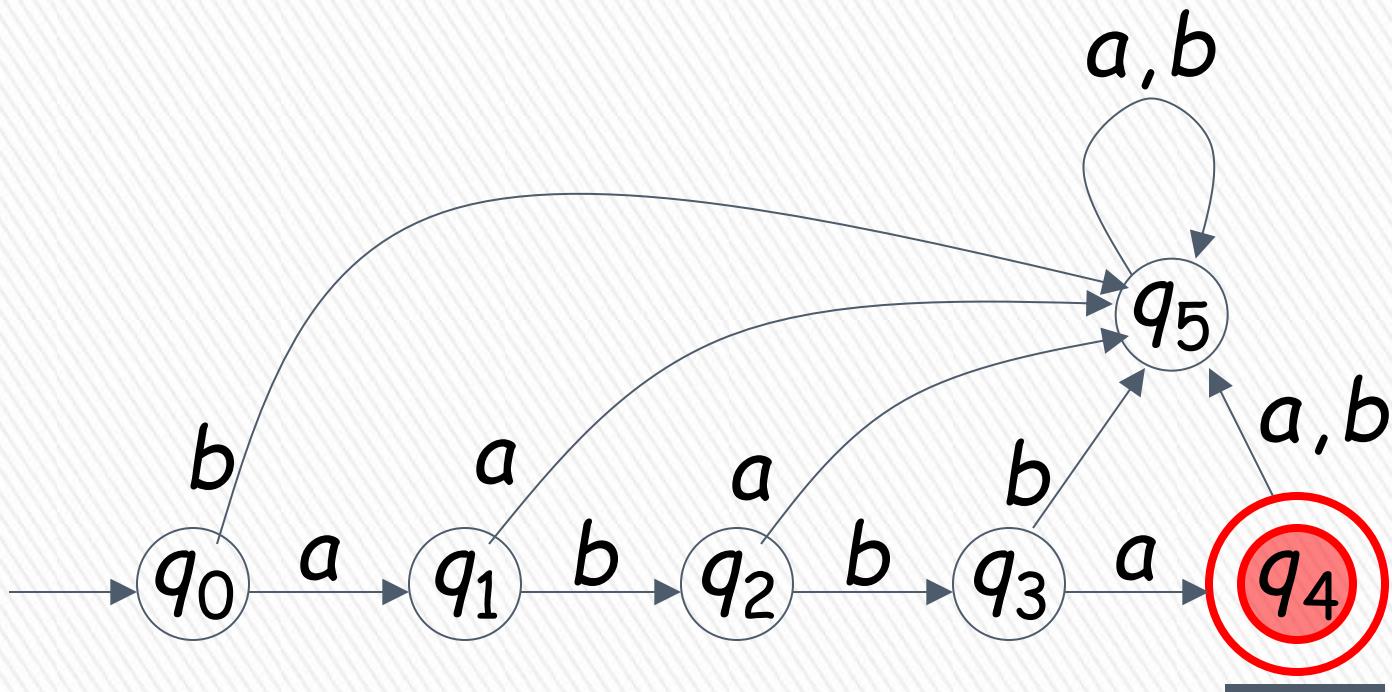
BLM2502 Theory of Computation



BLM2502 Theory of Computation



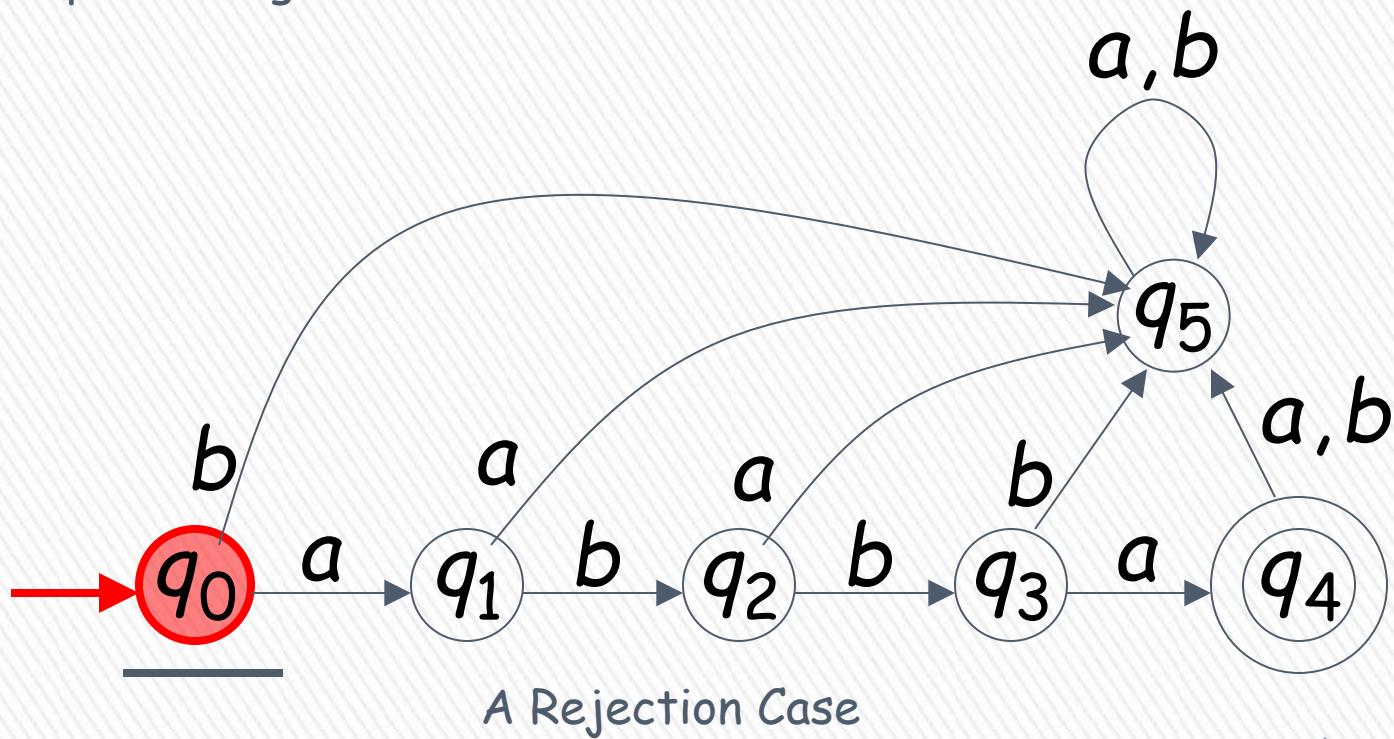
BLM2502 Theory of Computation



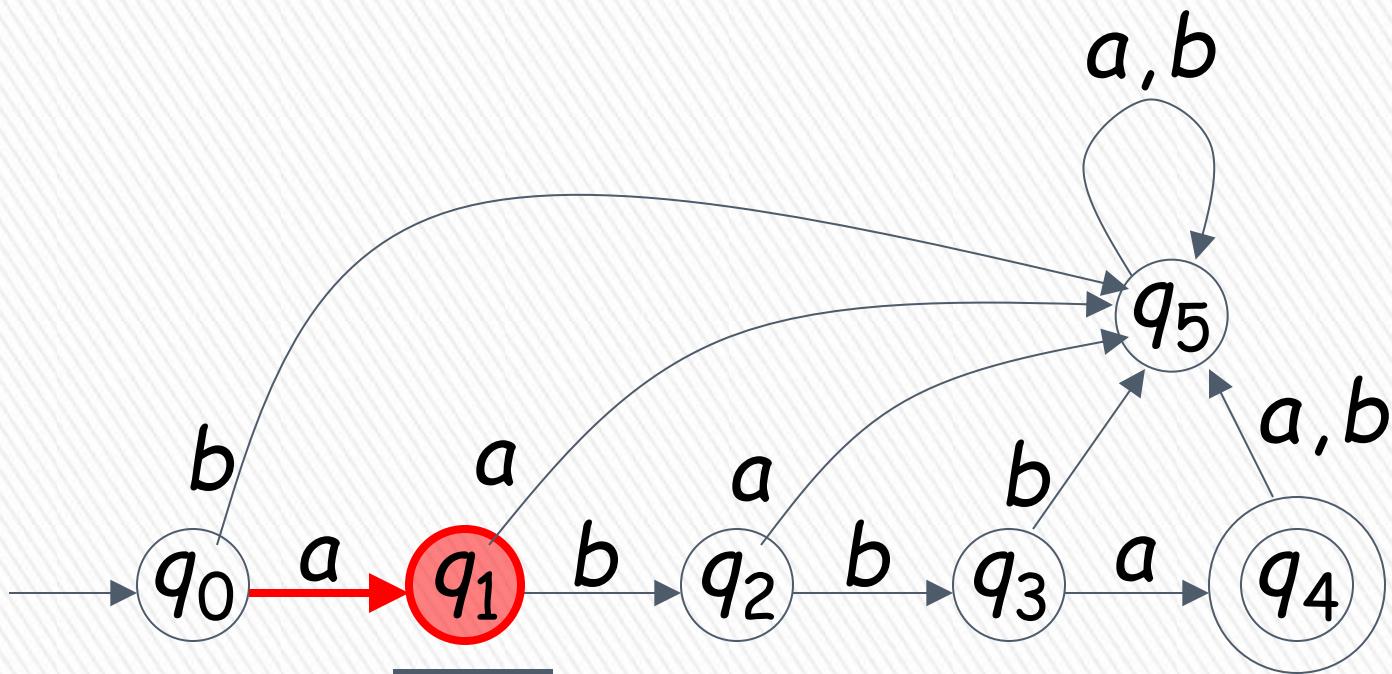
BLM2502 Theory of Computation



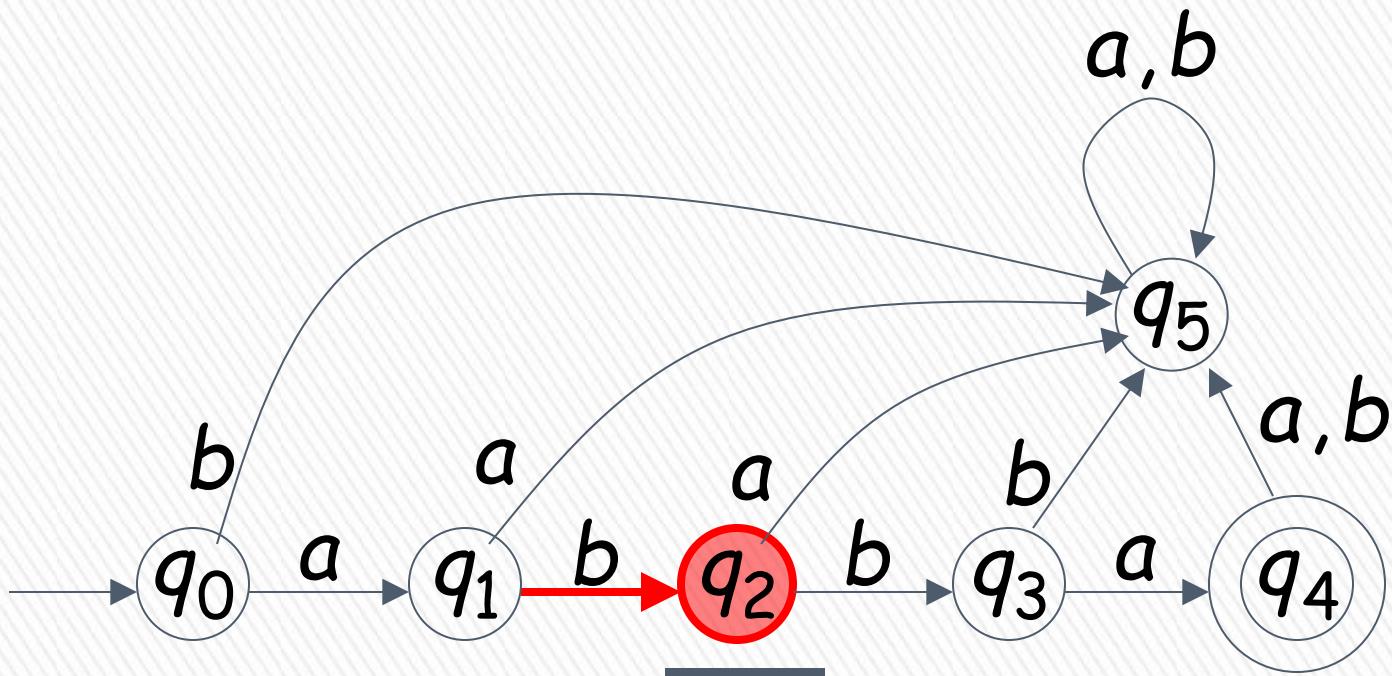
Input String



BLM2502 Theory of Computation



BLM2502 Theory of Computation

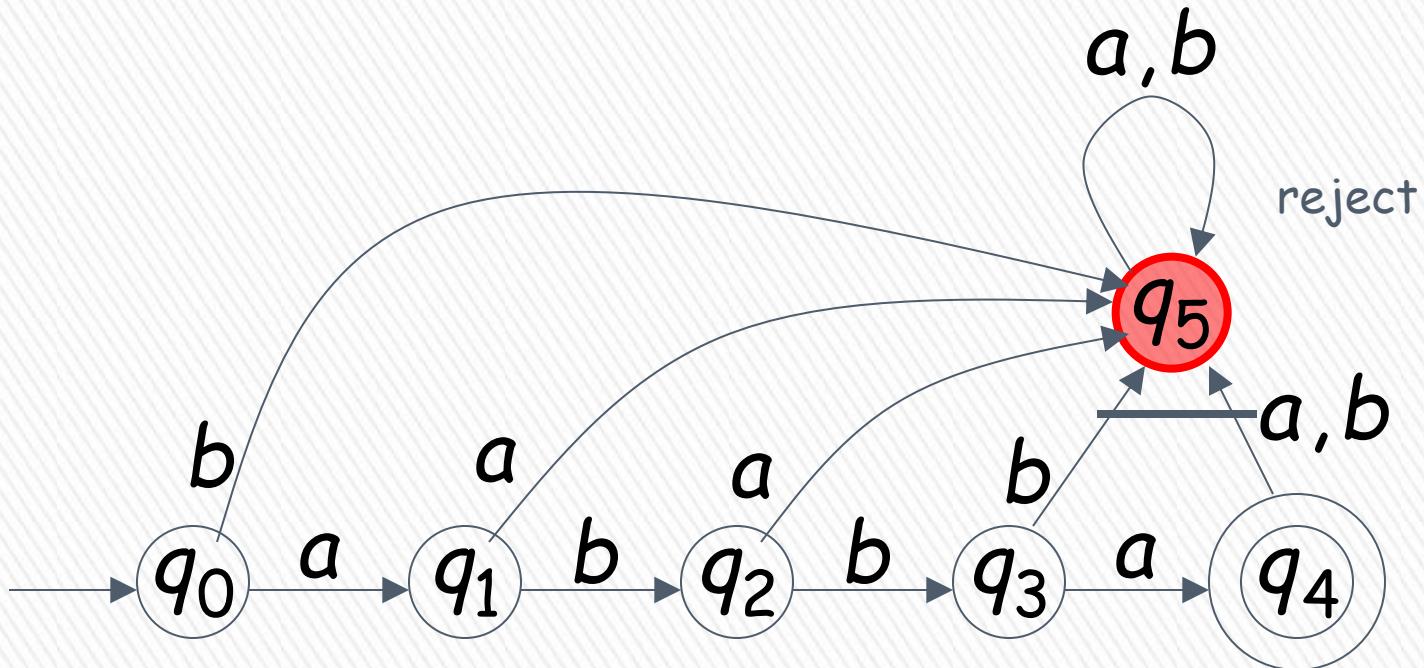


BLM2502 Theory of Computation



Input finished

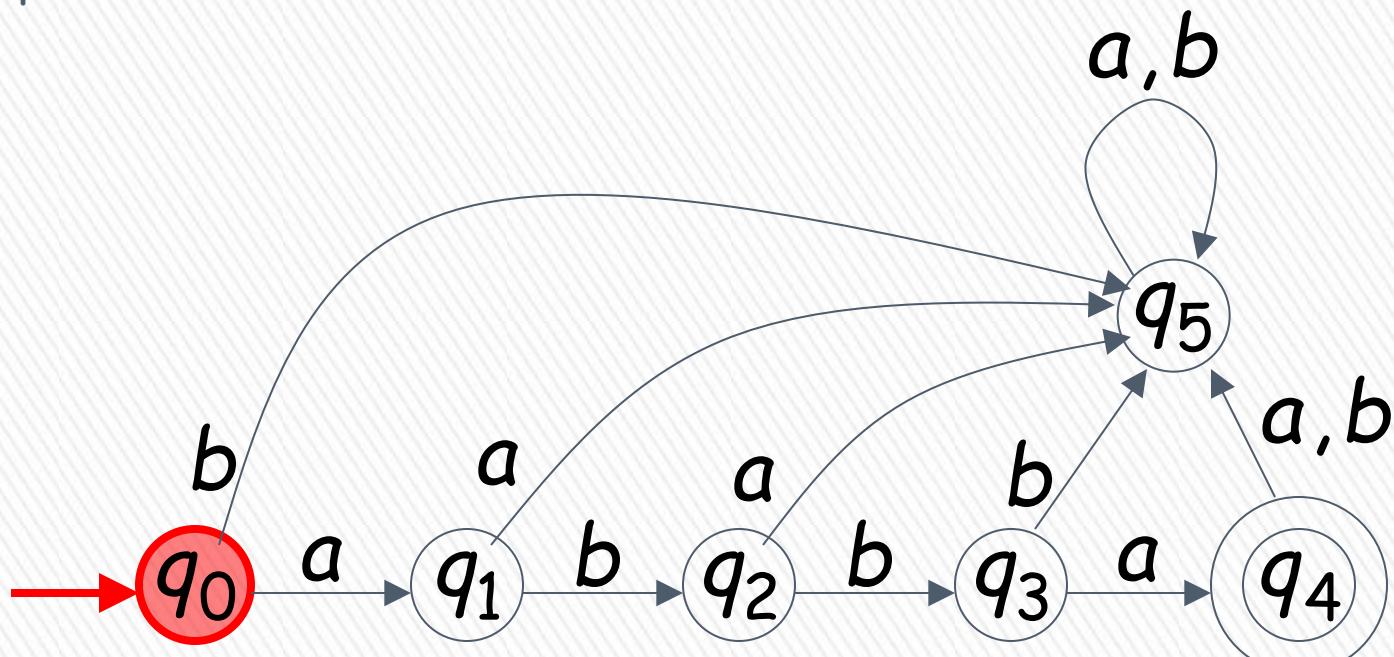
a	b	a	
---	---	---	--



BLM2502 Theory of Computation

↓ Tape is empty

Input Finished



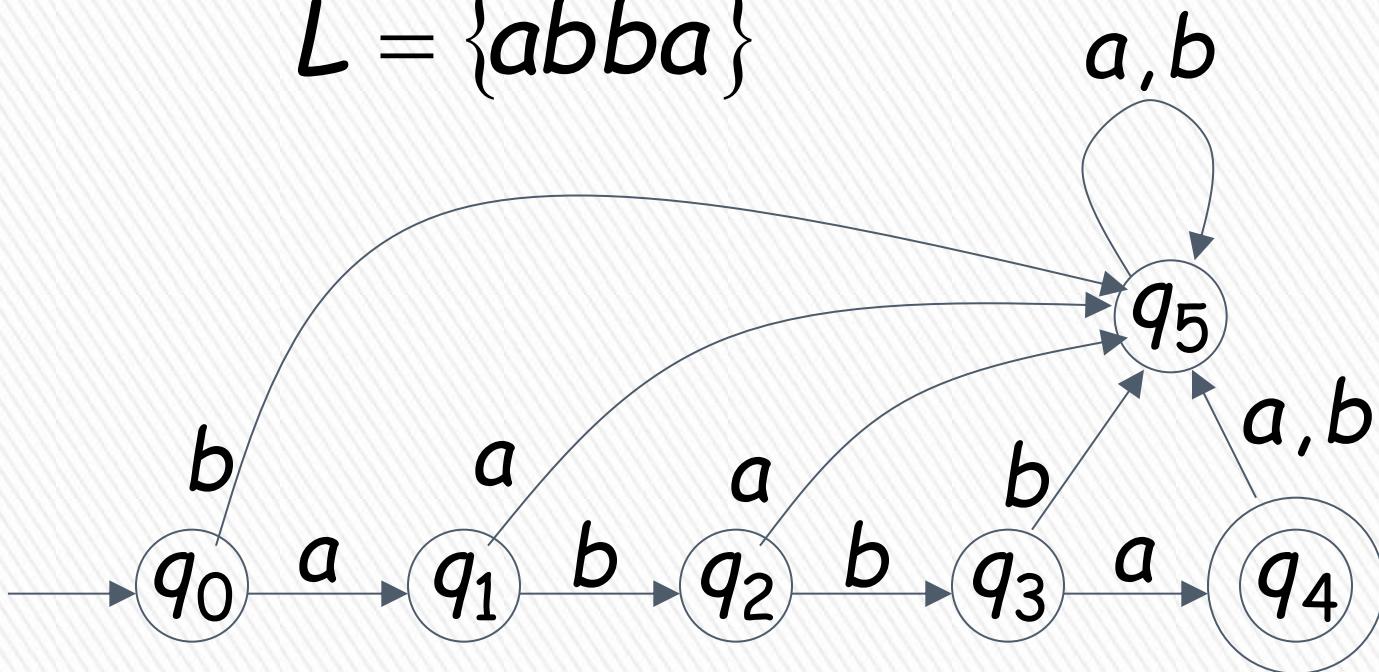
reject

Another Rejection Case

BLM2502 Theory of Computation

Language Accepted:

$$L = \{abba\}$$



BLM2502 Theory of Computation

To accept a string:

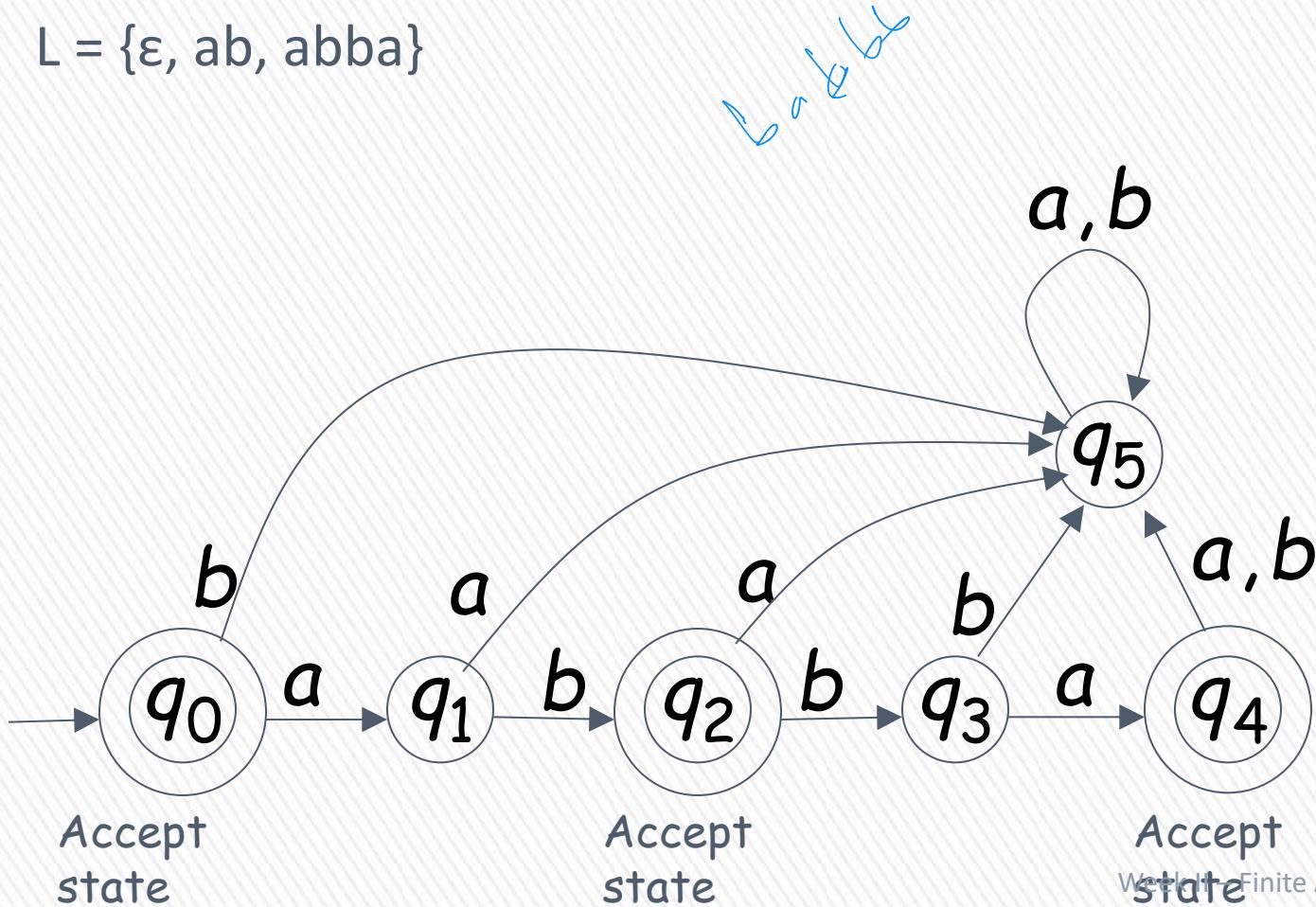
- all the input string is scanned and
- the last state is accepting

To reject a string:

- all the input string is scanned and
- the last state is non-accepting

BLM2502 Theory of Computation

$L = \{\epsilon, ab, abba\}$



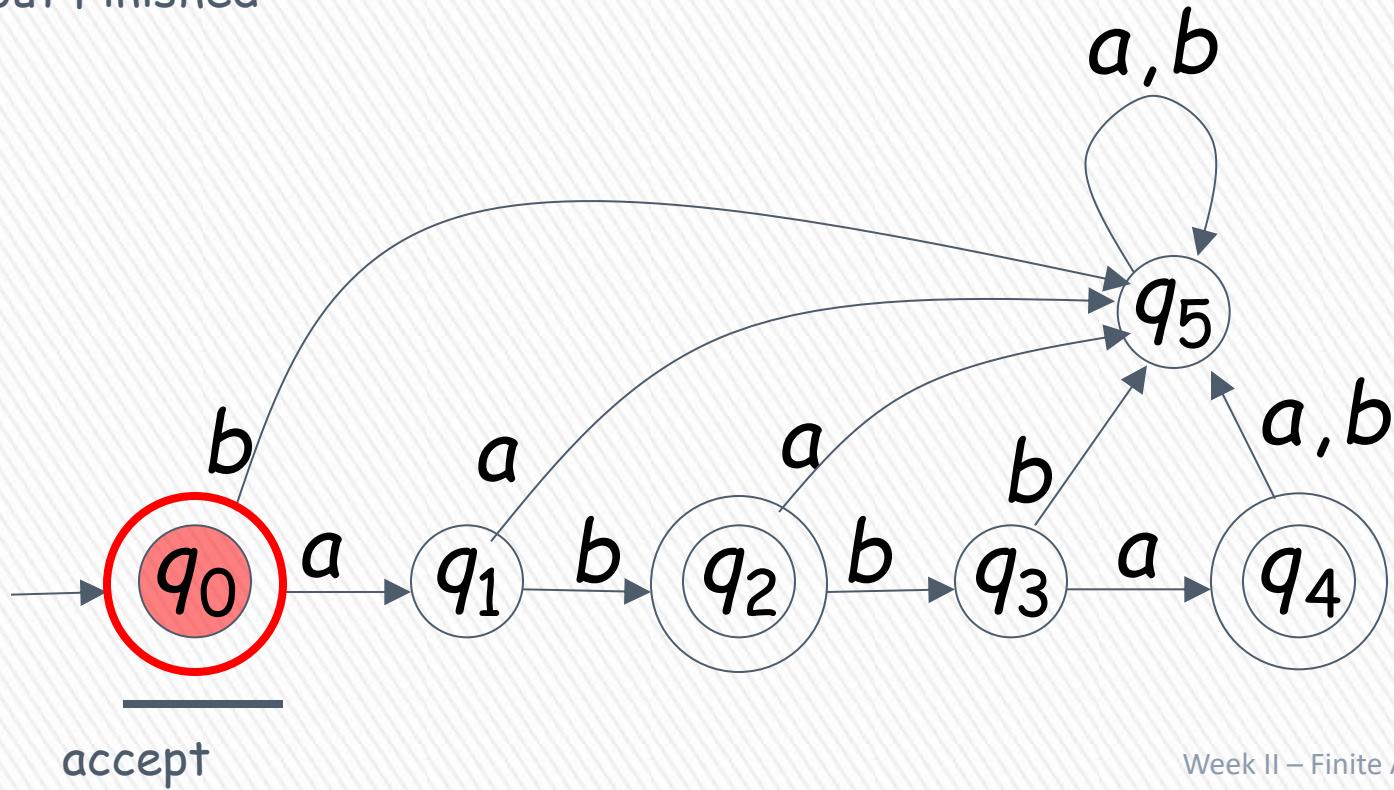
BLM2502 Theory of Computation



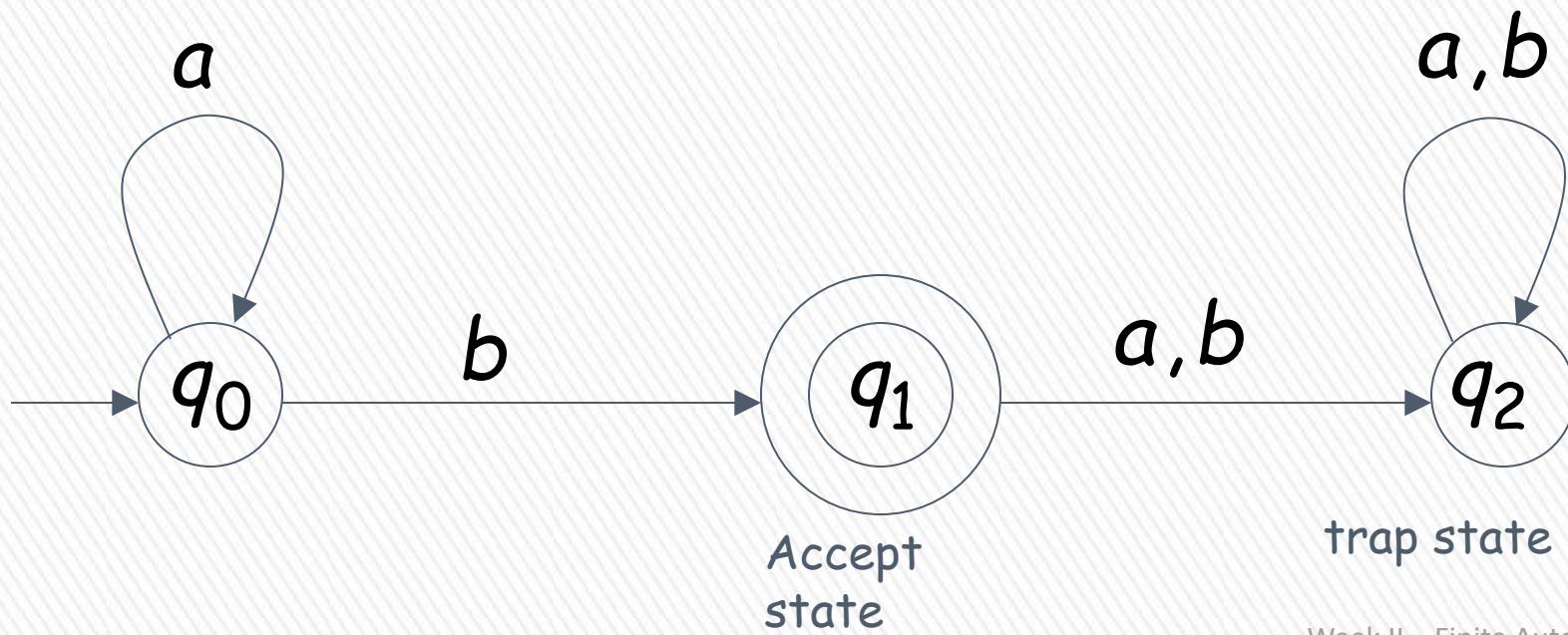
Empty Tape



Input Finished



BLM2502 Theory of Computation

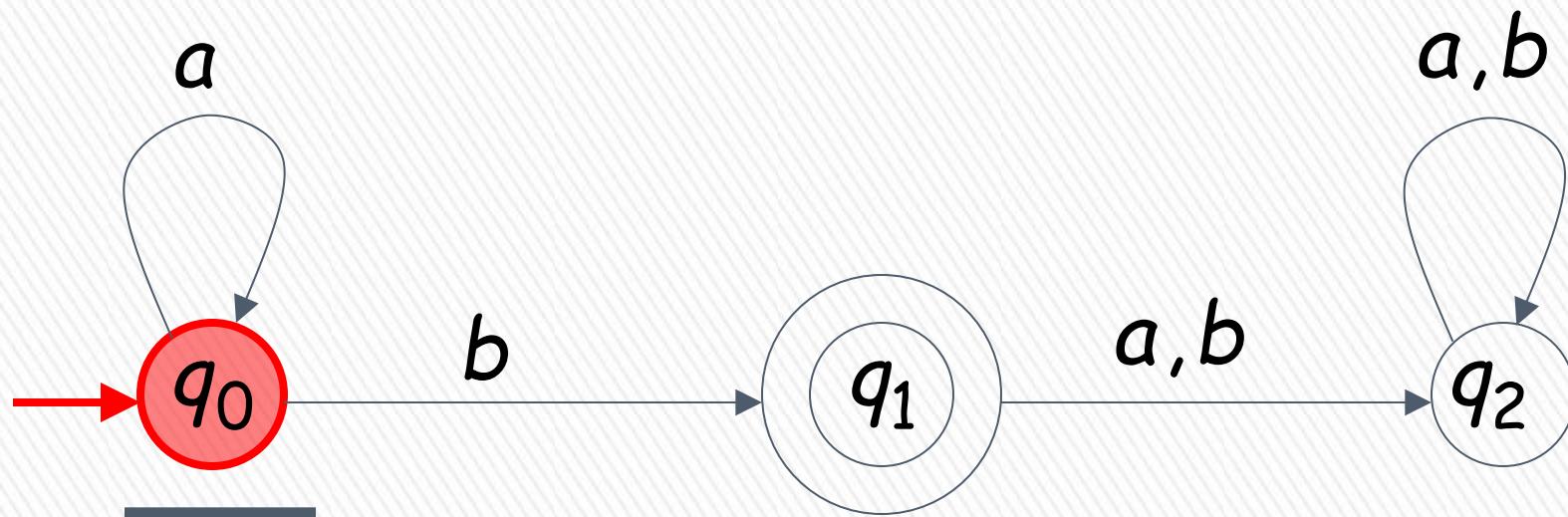


BLM2502 Theory of Computation

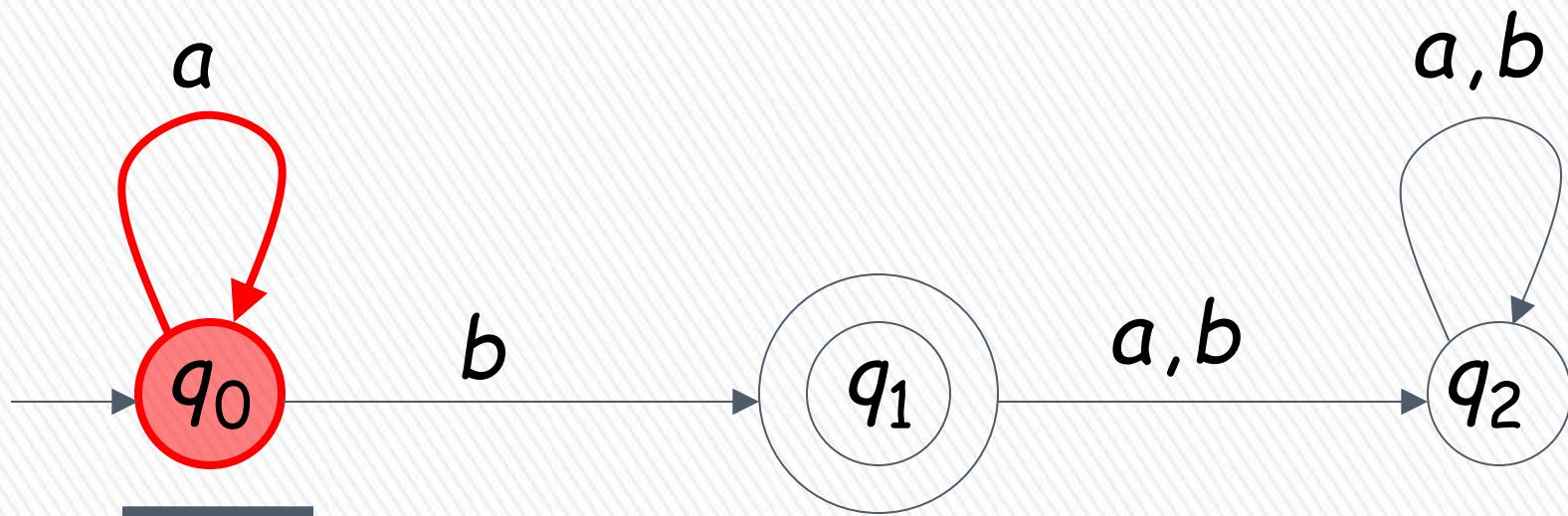
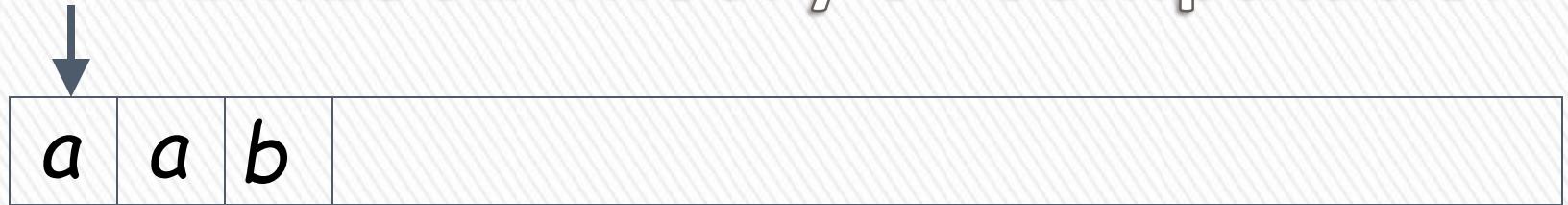


a	a	b	

Input String



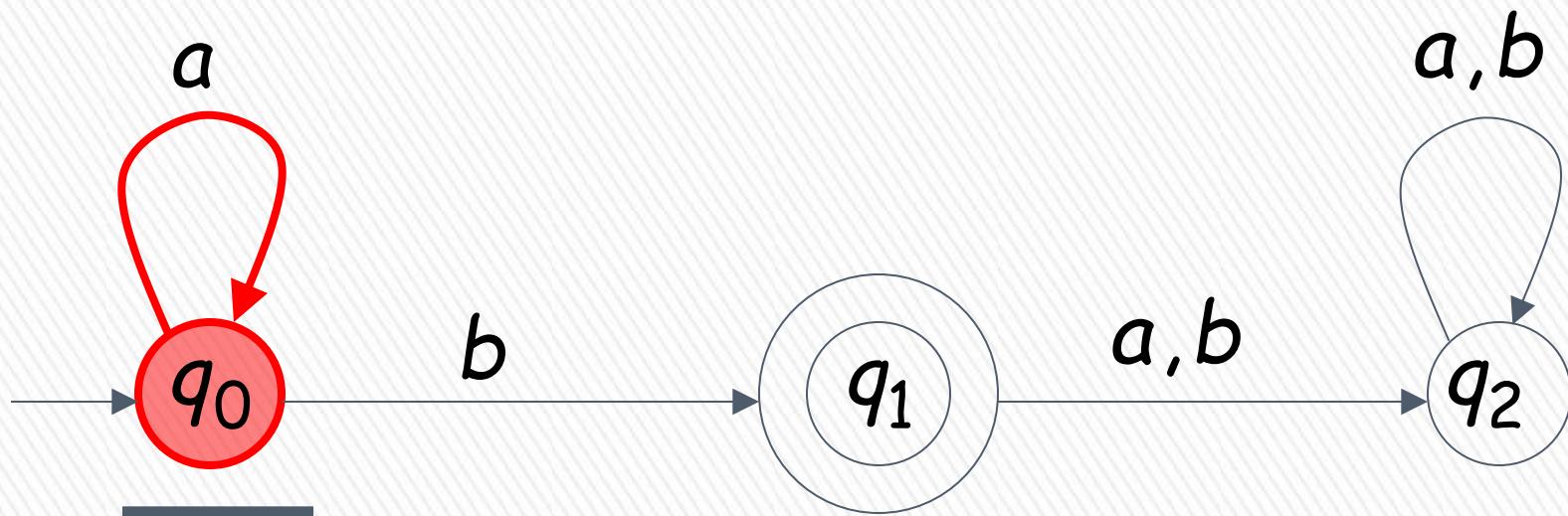
BLM2502 Theory of Computation



BLM2502 Theory of Computation



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

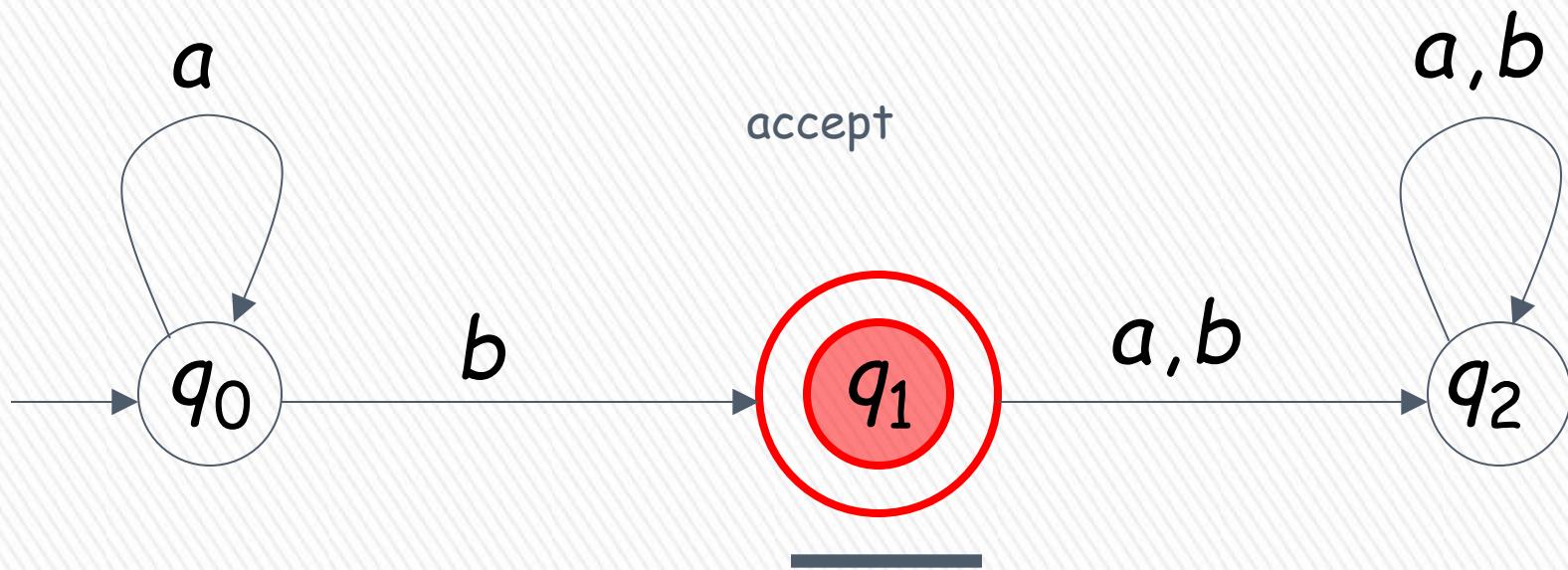


BLM2502 Theory of Computation



Input finished

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

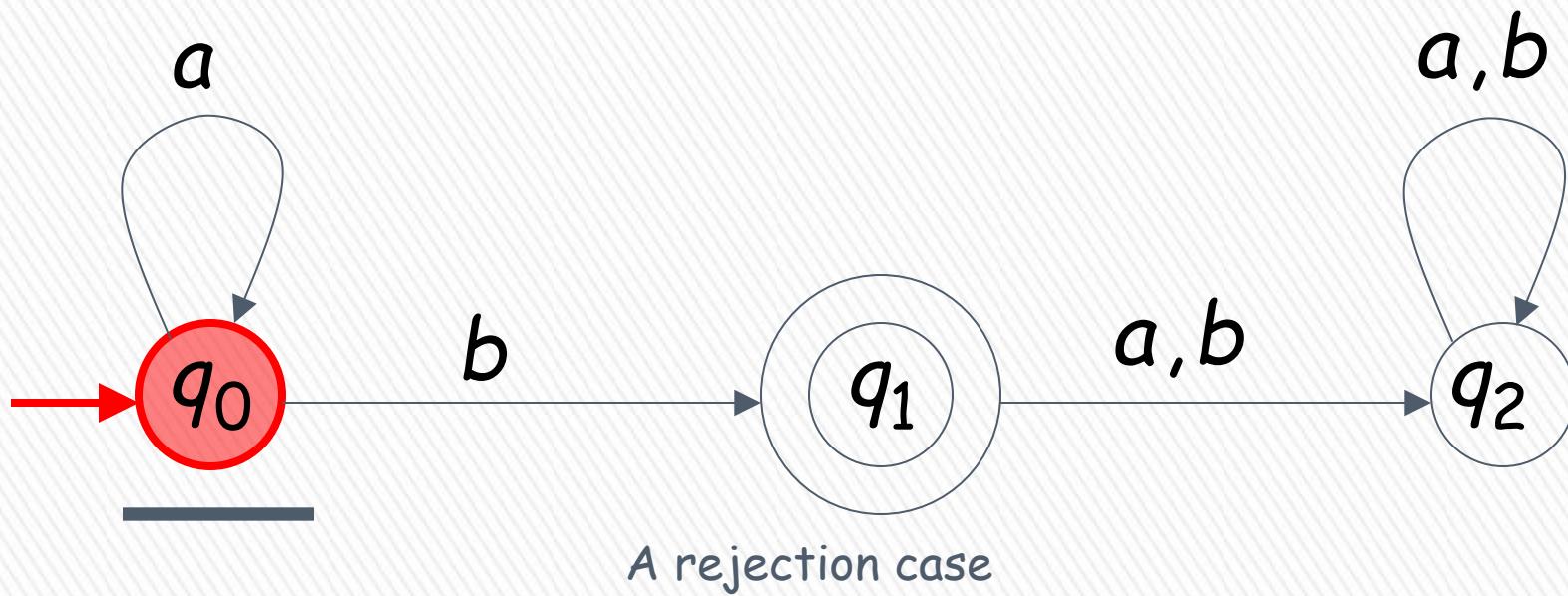


BLM2502 Theory of Computation

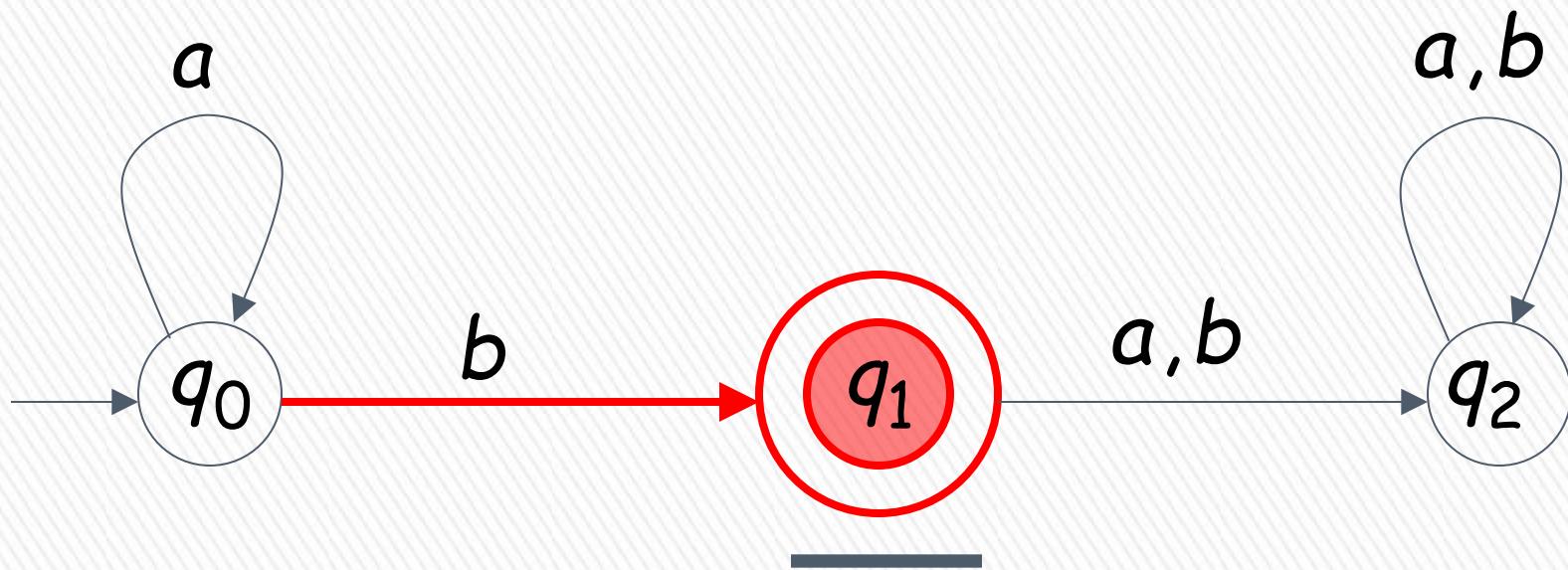


b	a	b	
---	---	---	--

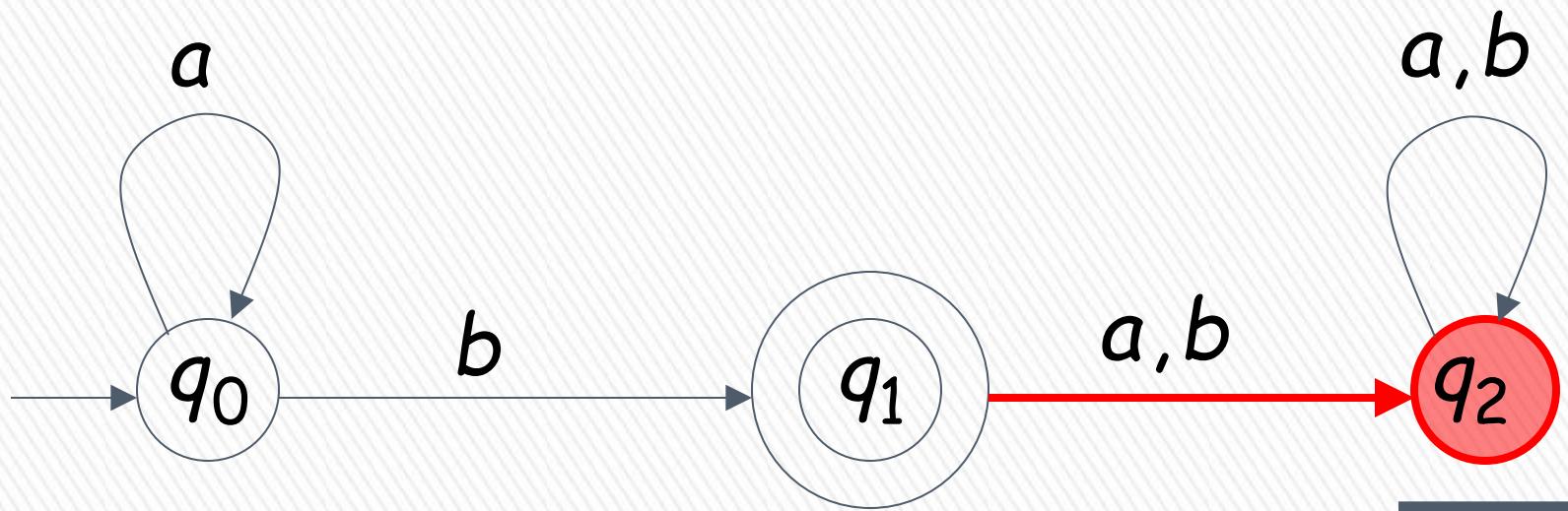
Input String



BLM2502 Theory of Computation



BLM2502 Theory of Computation

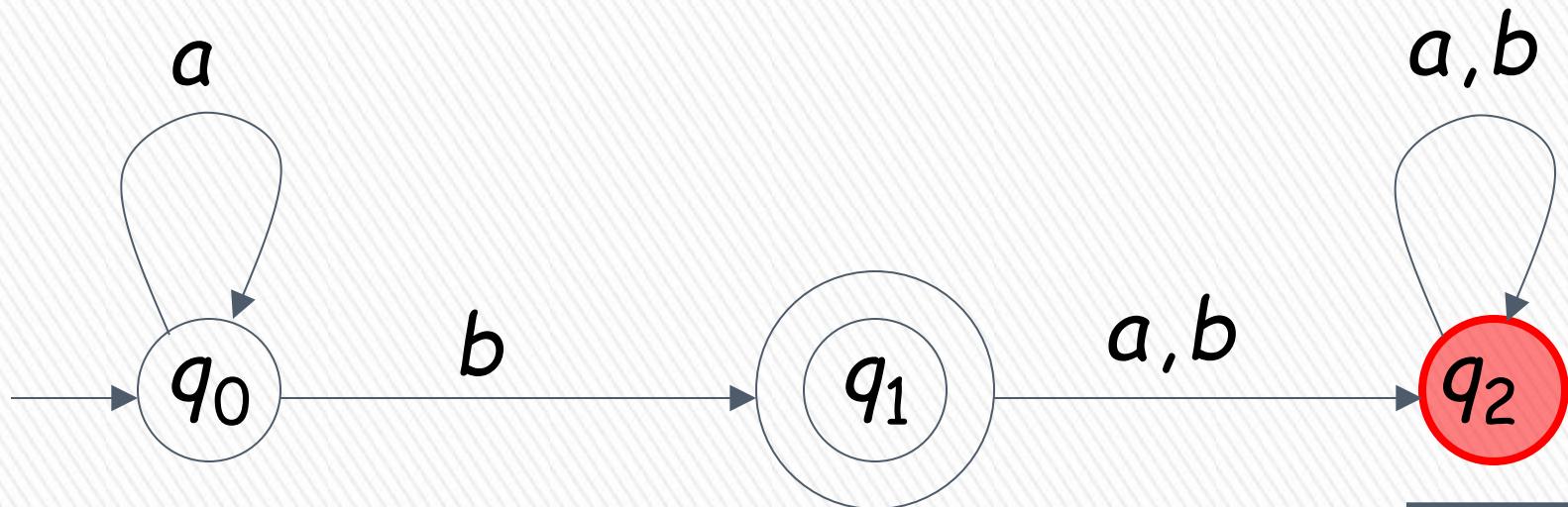


BLM2502 Theory of Computation



Input finished

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

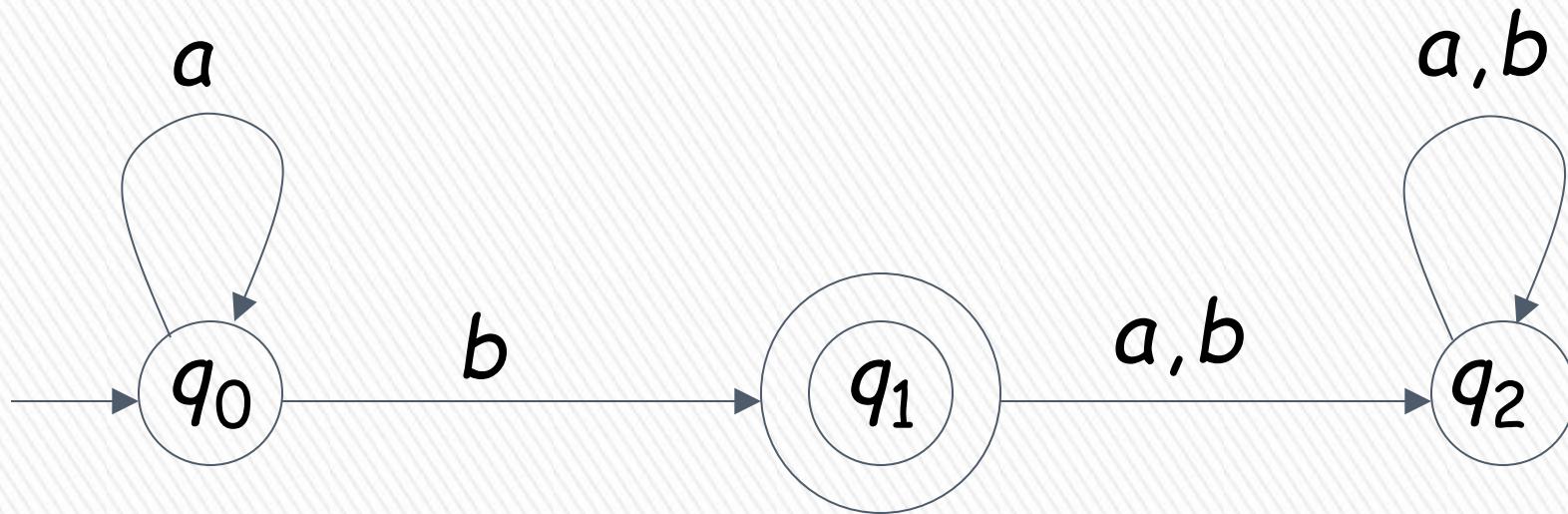


reject

BLM2502 Theory of Computation

Language Accepted:

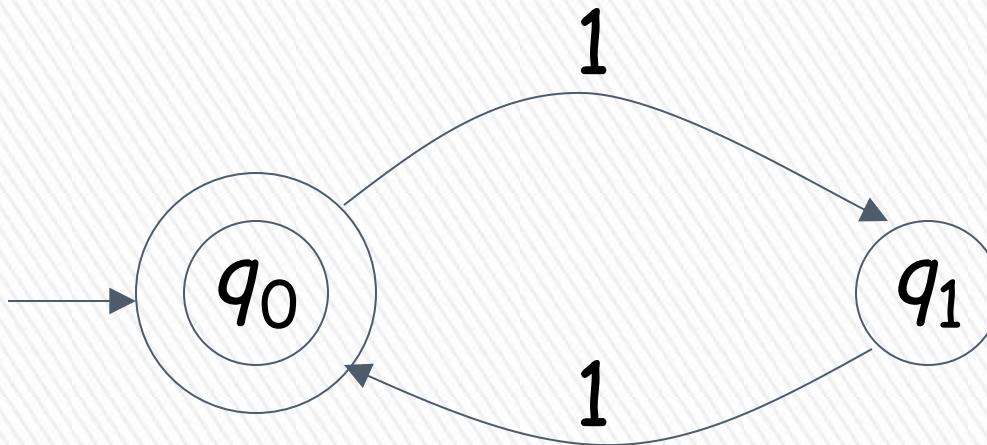
$$L = \{a^n b : n \geq 0\}$$



BLM2502 Theory of Computation

$$L = \{1^n : n=2k, k \text{ is integer}\}$$

Alphabet: $\Sigma = \{1\}$



Language Accepted:

$$\begin{aligned} \text{EVEN} &= \{x : x \text{ is in } \Sigma^* \text{ and } x \text{ is even}\} \\ &= \{\epsilon, 11, 1111, 111111, \dots\} \end{aligned}$$

BLM2502 Theory of Computation

» Extended Transition Function

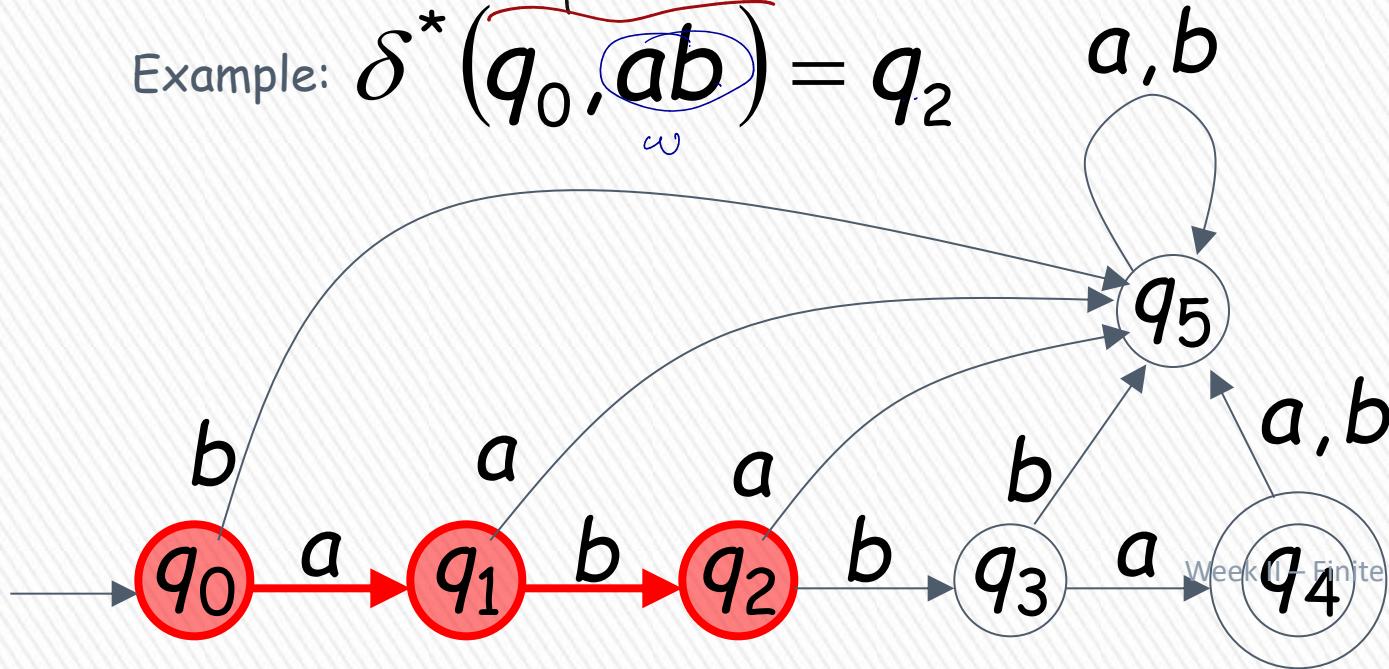
$$\delta^* : Q \times \Sigma^* \rightarrow Q$$

$$\delta(q, w) \rightarrow q'$$

> Describes the resulting state after scanning string w from state q

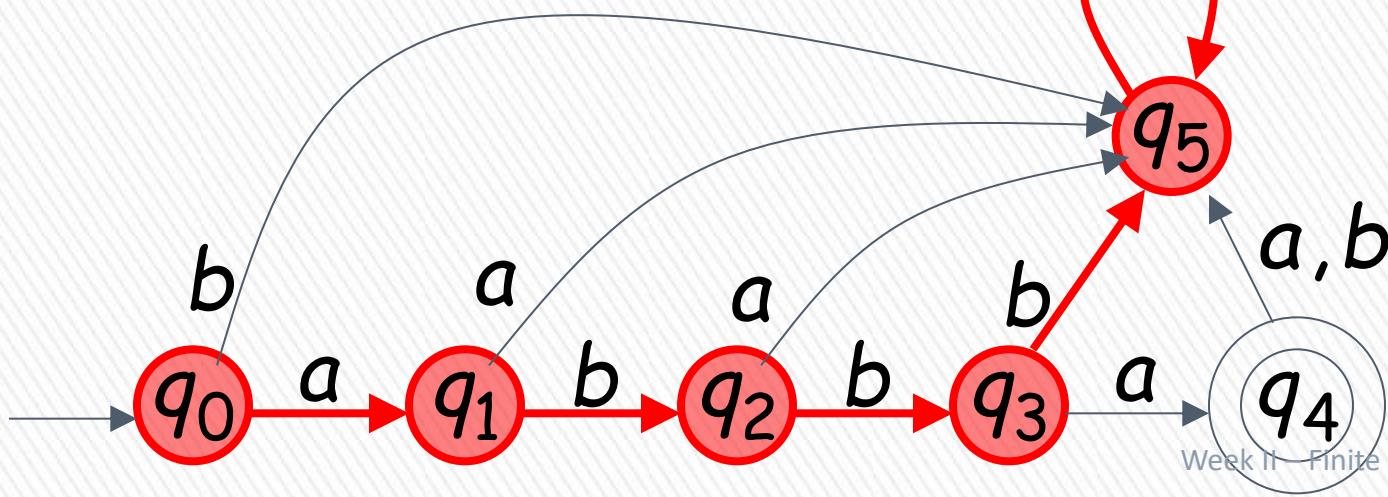
Example: $\delta^*(q_0, ab) = q_2$

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



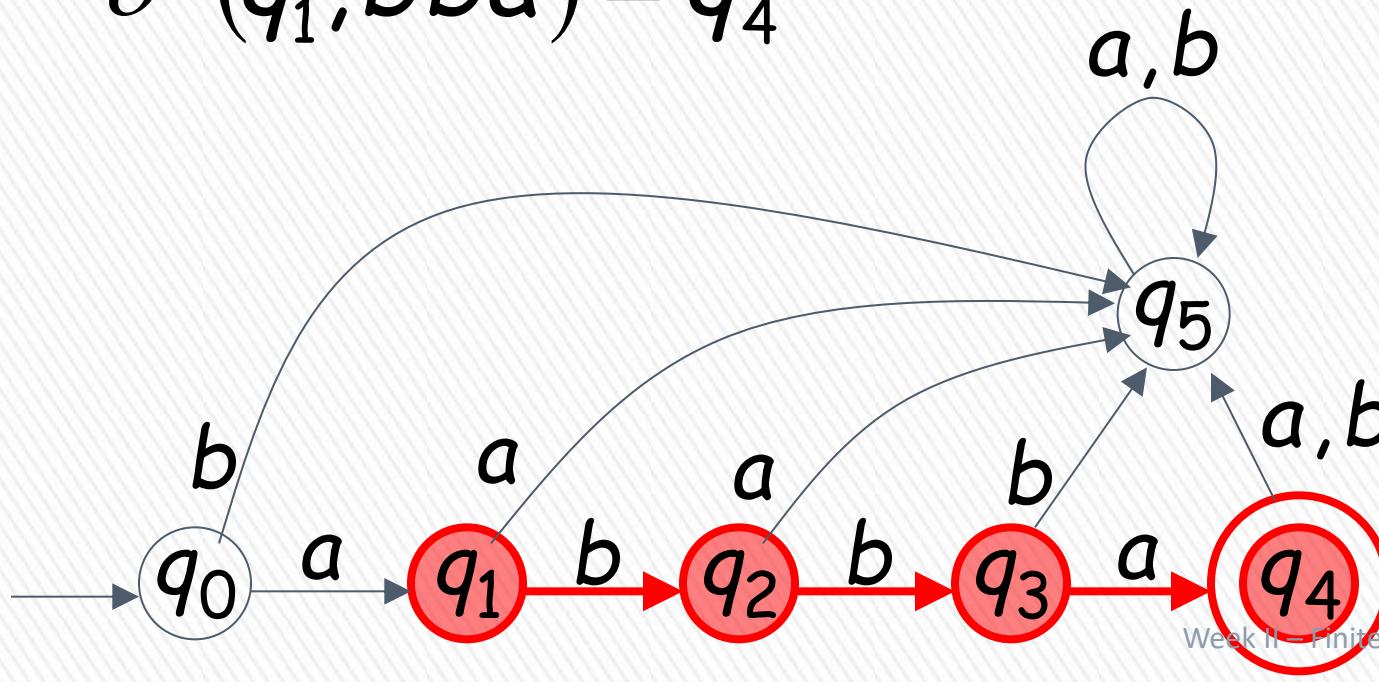
BLM2502 Theory of Computation

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



BLM2502 Theory of Computation

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



Week II – Finite Automata

BLM2502 Theory of Computation

In general:

$\delta^*(q, w) \rightarrow q'$ implies that there is a walk of transitions

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



states may be repeated



BLM2502 Theory of Computation

Language Accepted By DFA

The **language** accepted by an automaton M , is denoted as $L(M)$ and contains all the strings accepted by M

We say that a language L' is accepted (or recognized) by the DFA M if

$$L(M) = L'$$

An automaton accepts one and only one language.
A language can be accepted by a number of automata

BLM2502 Theory of Computation

- » For a DFA $M = (Q, \Sigma, \delta, q_0, F)$
- » Language accepted by M :

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



BLM2502 Theory of Computation

» Language rejected by **M** :

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



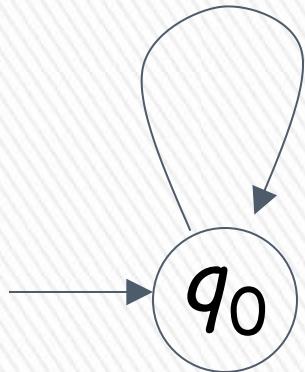
BLM2502 Theory of Computation



BLM2502 Theory of Computation

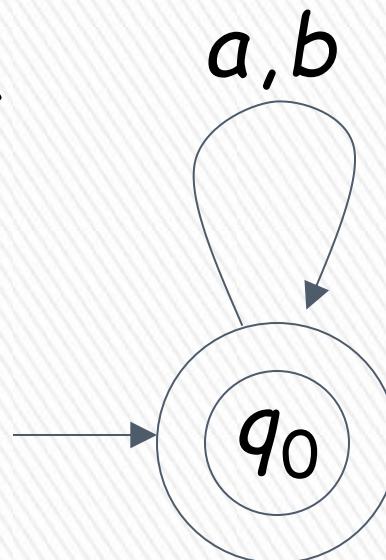
More DFA Examples

a, b



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

a, b



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

Empty language

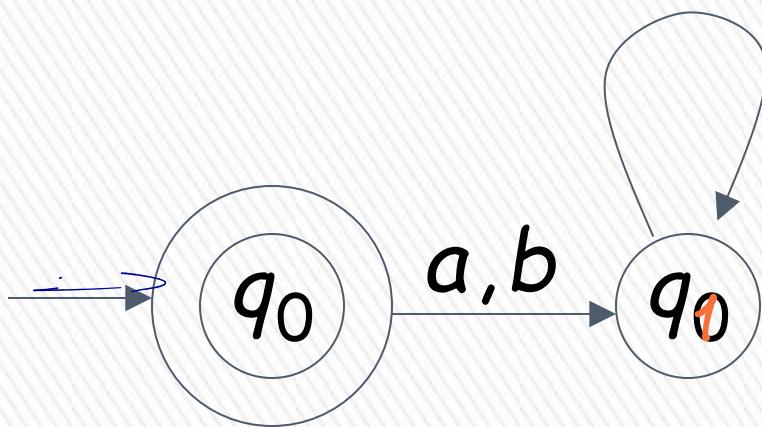
$$L(M) = \Sigma^*$$

All strings

BLM2502 Theory of Computation

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

a, b



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

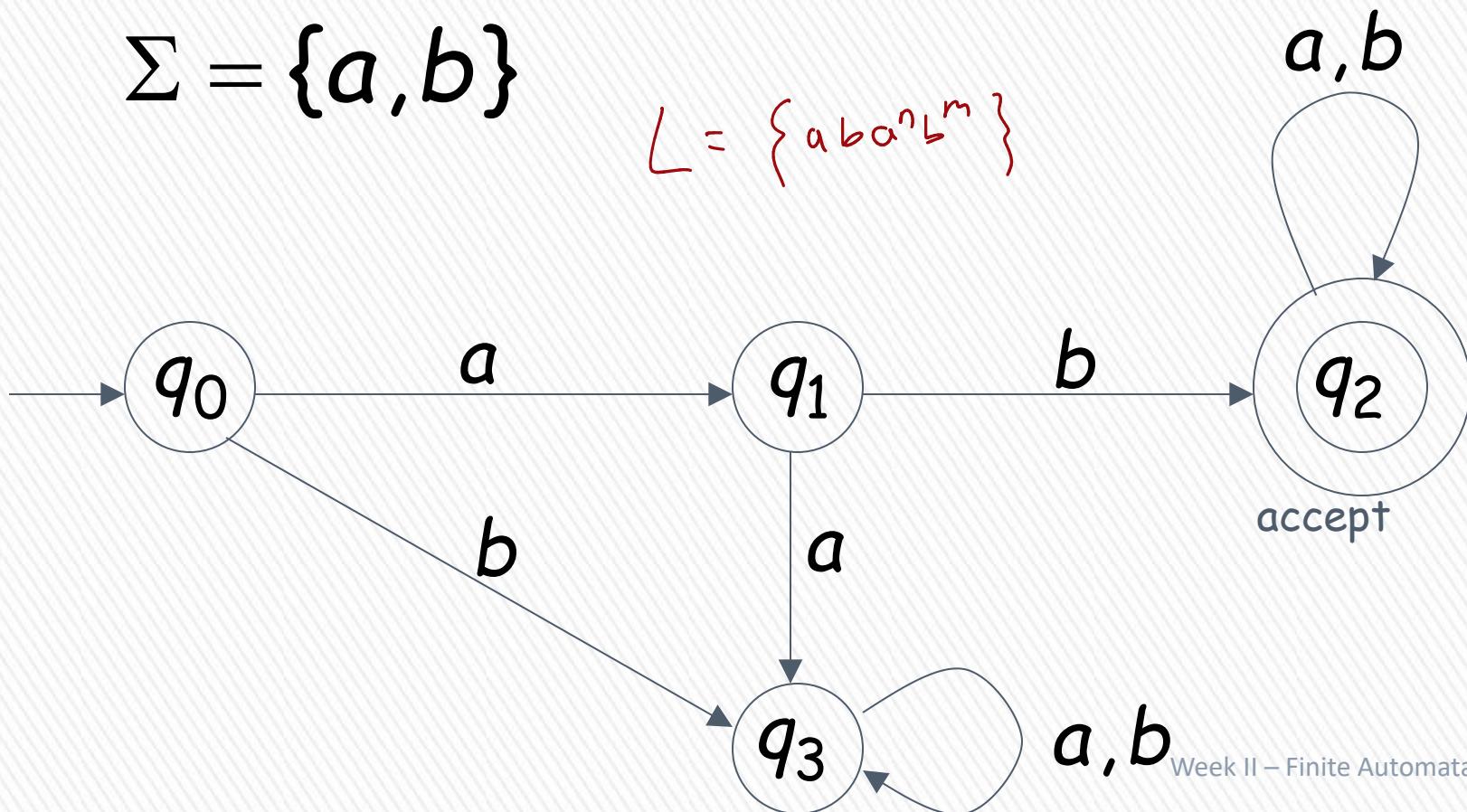
Language of the empty string

BLM2502 Theory of Computation

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

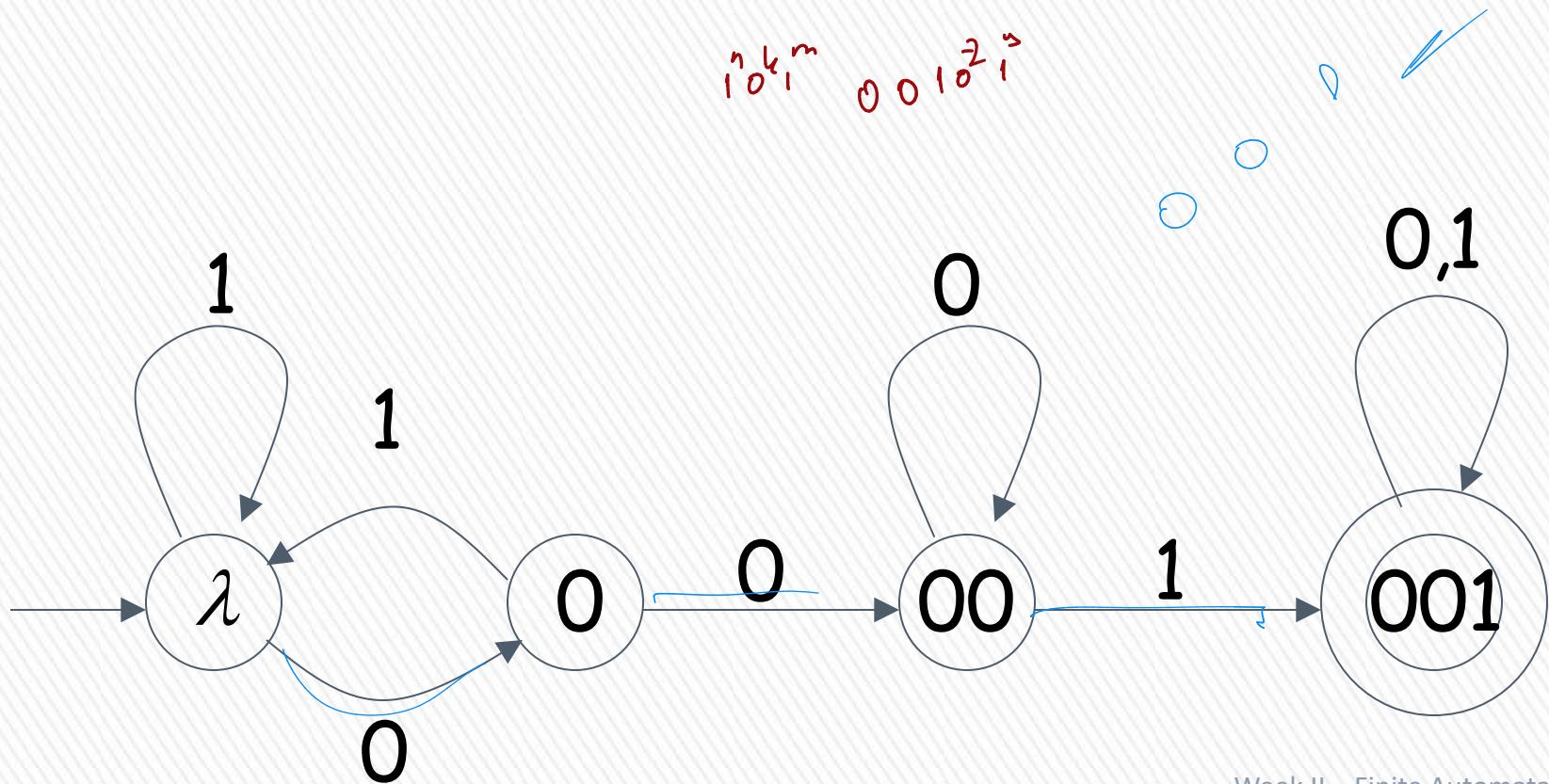
$$\Sigma = \{a, b\}$$

$$L = \{ab a^n b^m\}$$



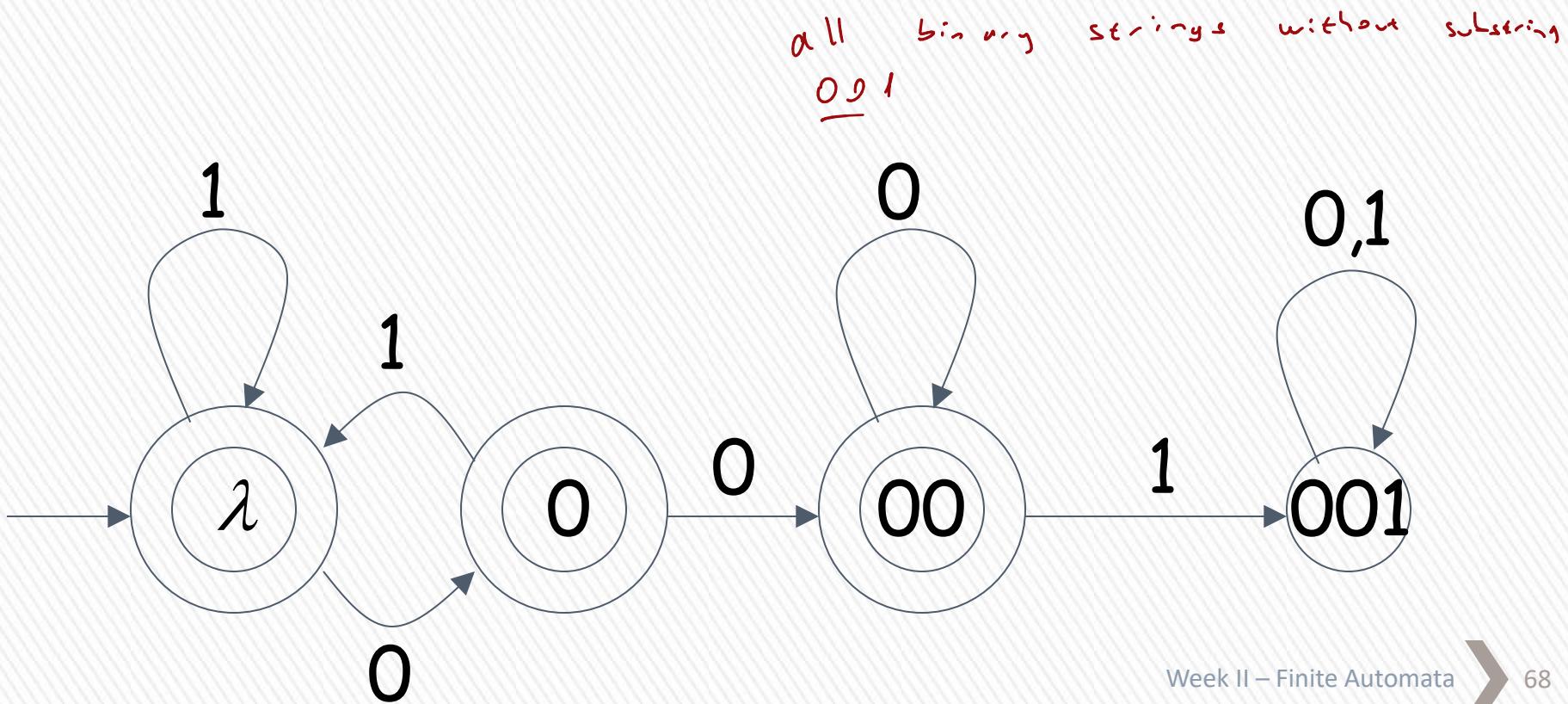
BLM2502 Theory of Computation

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



BLM2502 Theory of Computation

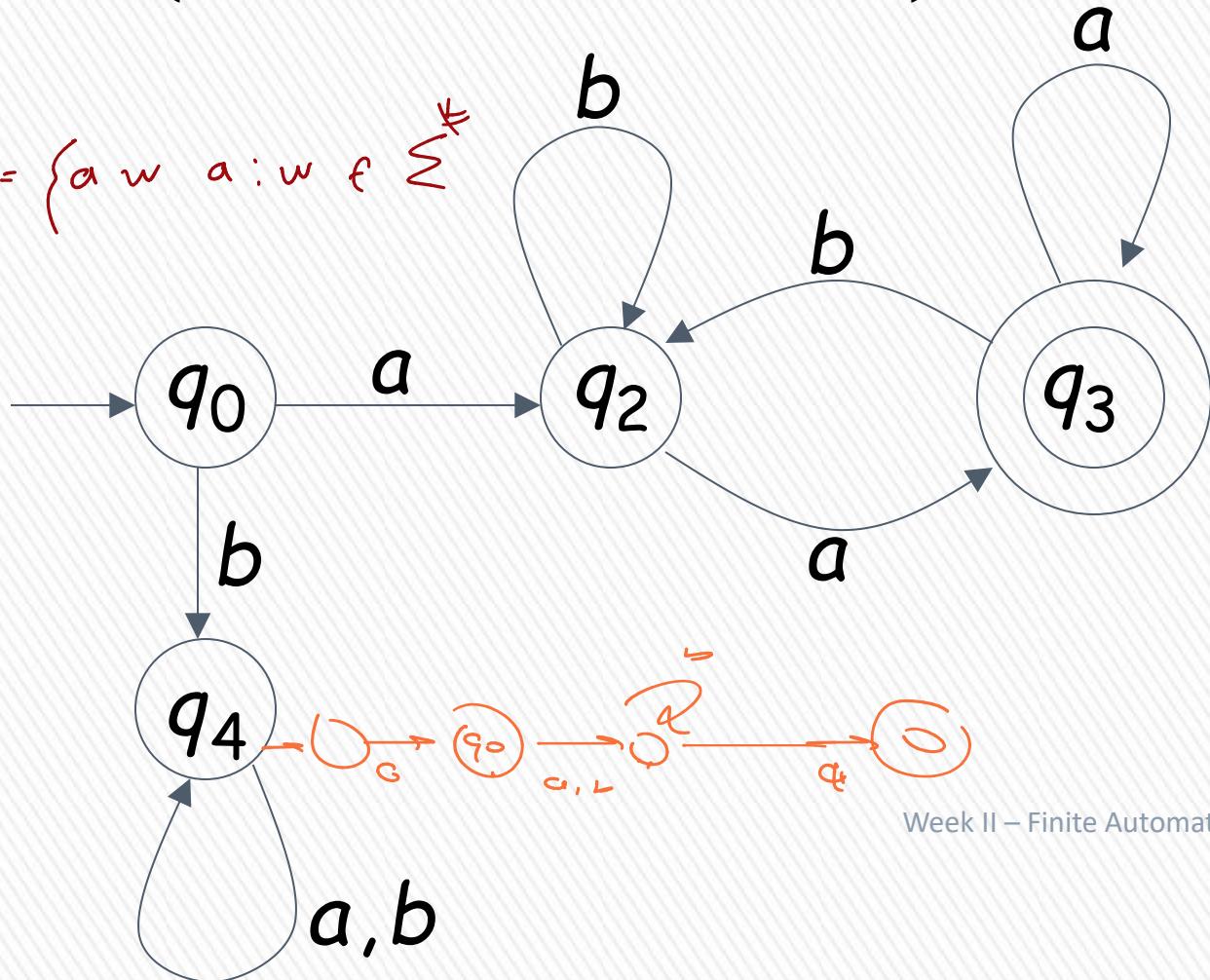
$L(M) = \{ \text{all binary strings without substring } 001 \}$



BLM2502 Theory of Computation

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

$$L(M) = \{awba : w \in \Sigma^*\}$$



BLM2502 Theory of Computation

$\{abba\}$ $\{\lambda, ab, abba\}$

$\{a^n b : n \geq 0\}$ $\{awa : w \in \{a,b\}^*\}$

$\{x : x \in \{1\}^* \text{ and } x \text{ is even}\}$

$\{ \}$ $\{\lambda\}$ $\{a,b\}^*$

{ all binary strings without substring 001 }

{ all strings in $\{a,b\}^*$ with prefix ab }

There exist automata that accept these languages (see previous slides).

BLM2502 Theory of Computation

There exist languages which are not Regular:



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

$$ADDITION = \{x + y = z : x = 1^n, y = 1^m,$$

$$z = 1^k, n + m = k\}$$

There is no DFA that accepts these languages



BLM2502

Theory of

Computation

BLM2502 Theory of Computation

» Course Outline

Week	Content
1.	Introduction to Course
2.	Computability Theory, Complexity Theory, Automata Theory, Set Theory, Relations, Proofs, Pigeonhole Principle
3.	Regular Expressions
4.	Finite Automata
5.	Deterministic and Nondeterministic Finite Automata
6.	Epsilon Transition, Equivalence of Automata
7.	Pumping Theorem
8.	Context Free Grammars
9.	Parse Tree, Ambiguity,
10.	Pumping Theorem
11.	Turing Machines, Recognition and Computation, Church-Turing Hypothesis
12.	Turing Machines, Recognition and Computation, Church-Turing Hypothesis
13.	Review

NFA

Non-Deterministic Finite Automata



Deterministic Same inputs always,
Same outputs

Formal Definition of NFA

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

Q : Set of states, i.e. $\{q_0, q_1, q_2\}$

Σ : Input alphabet, i.e. $\{a, b\}$ $\epsilon \notin \Sigma$

δ : Transition function $Q \times \Sigma \xrightarrow{\delta} 2^Q$

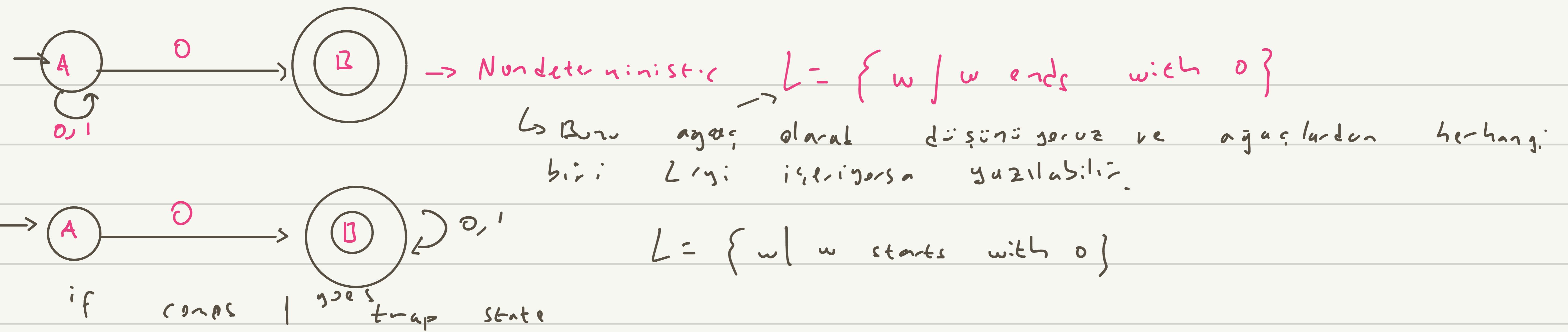
q_0 : Initial state

F : Accepting states

A	A	B	A	B
A B	ϵ	A B	ϵ	
A	B	A	B	
A B	ϵ	A B	ϵ	

B

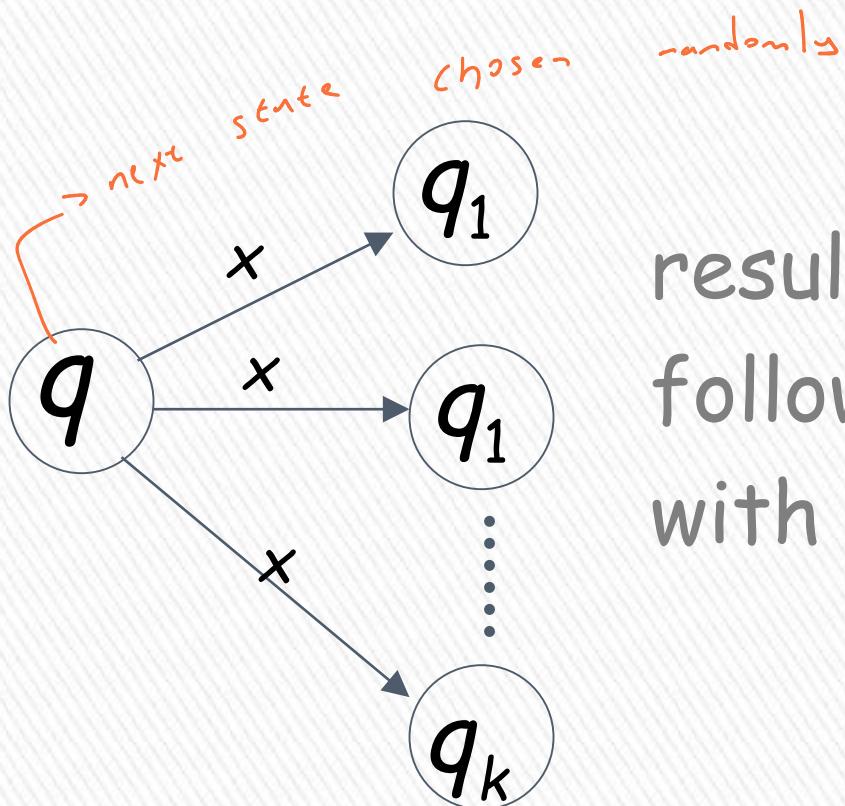




Transition Function δ

non-deterministic

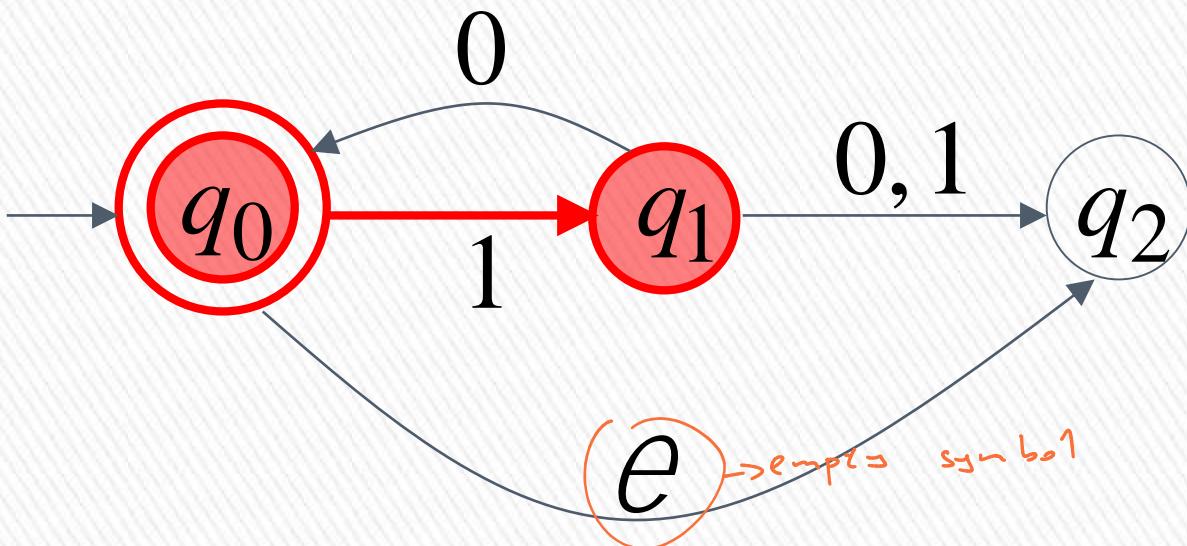
$$\delta(q, x) = \{q_1, q_2, \dots, q_k\}$$



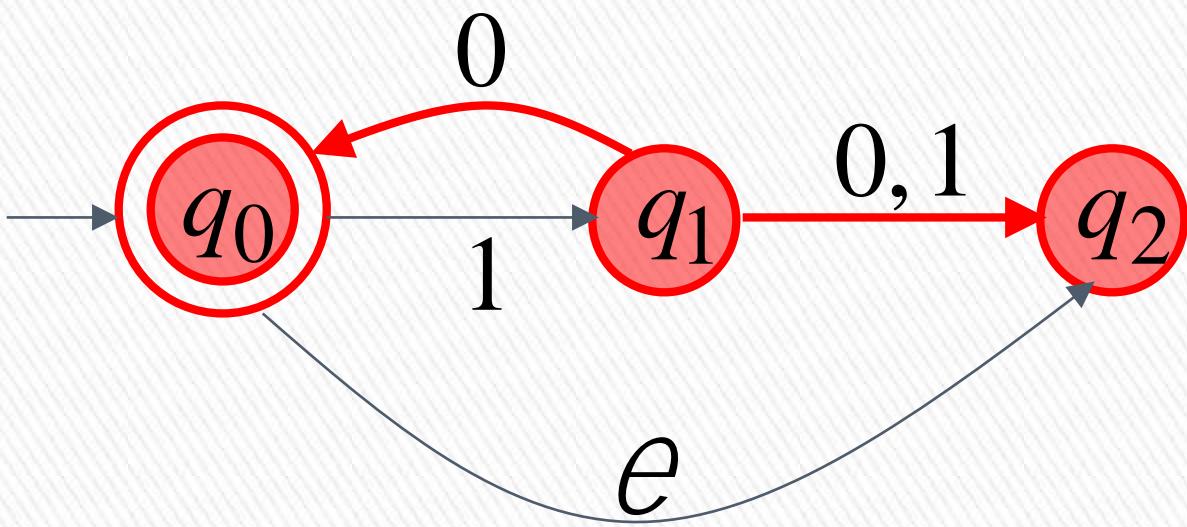
resulting states with
following **one** transition
with symbol x



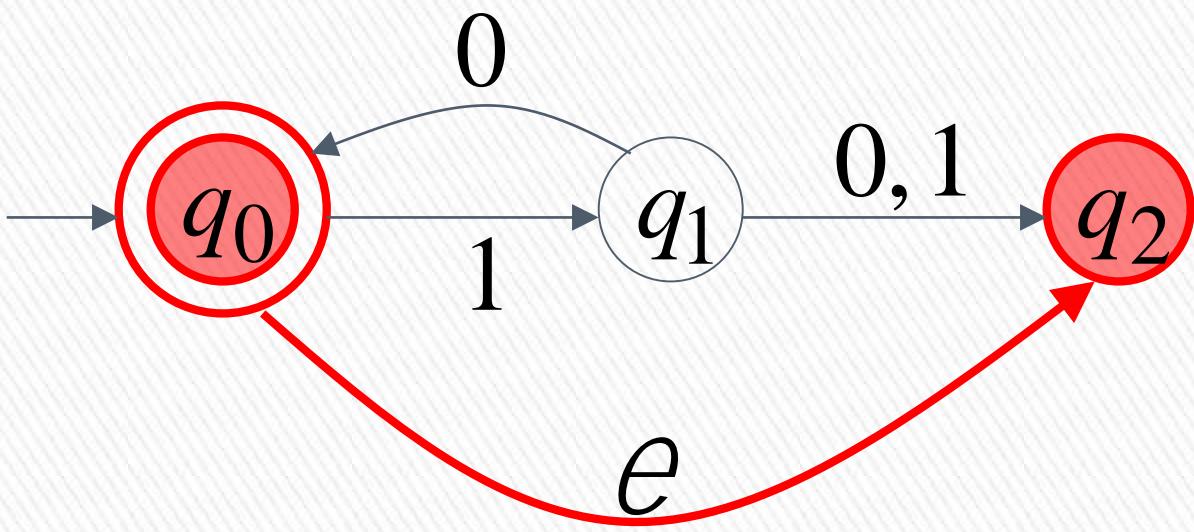
$$\delta(q_0, 1) = \{q_1\}$$



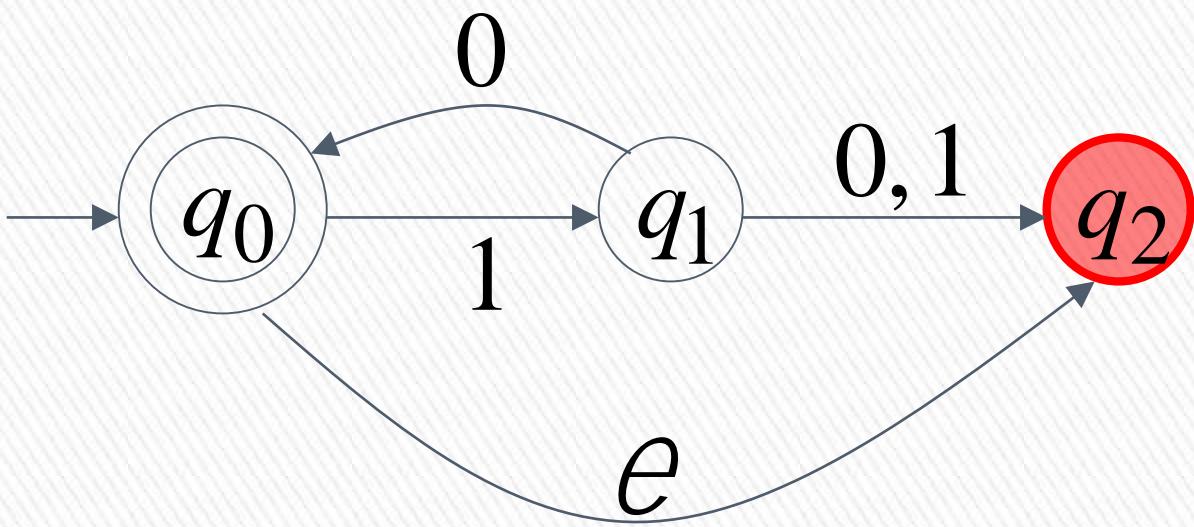
$$\delta(q_1, 0) = \{q_0, q_2\}$$



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

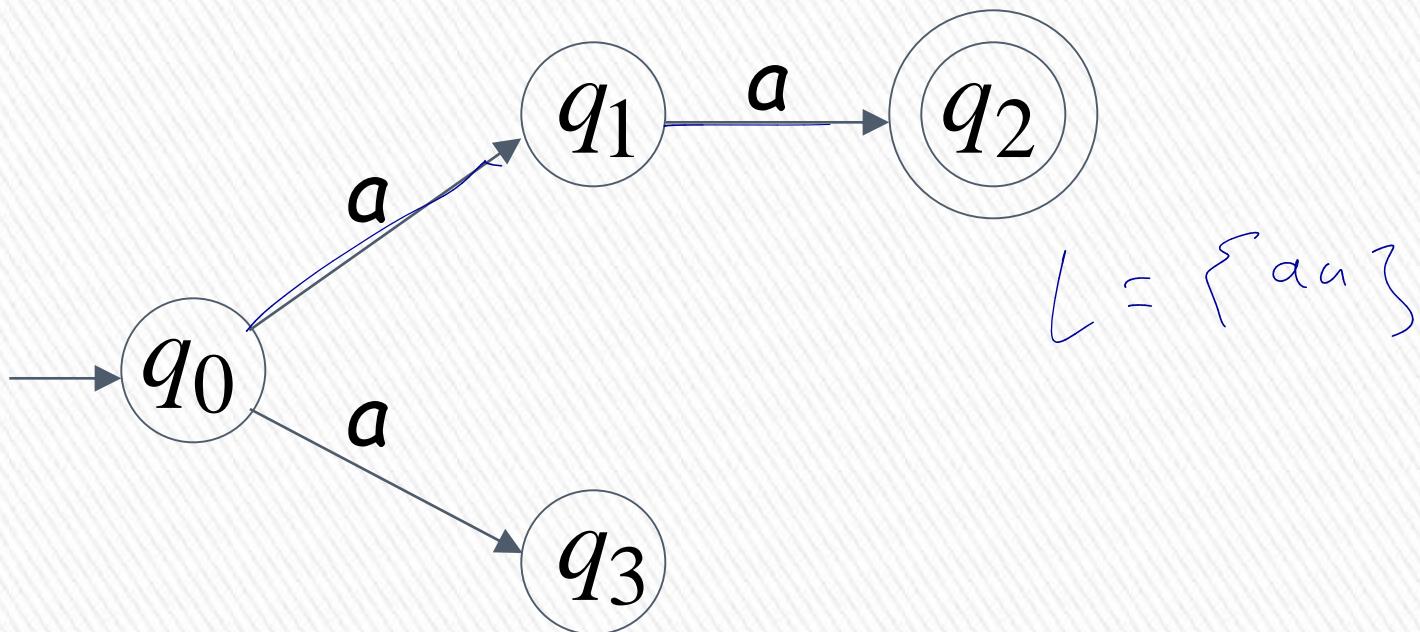


$$\delta(q_2, 1) = \emptyset$$

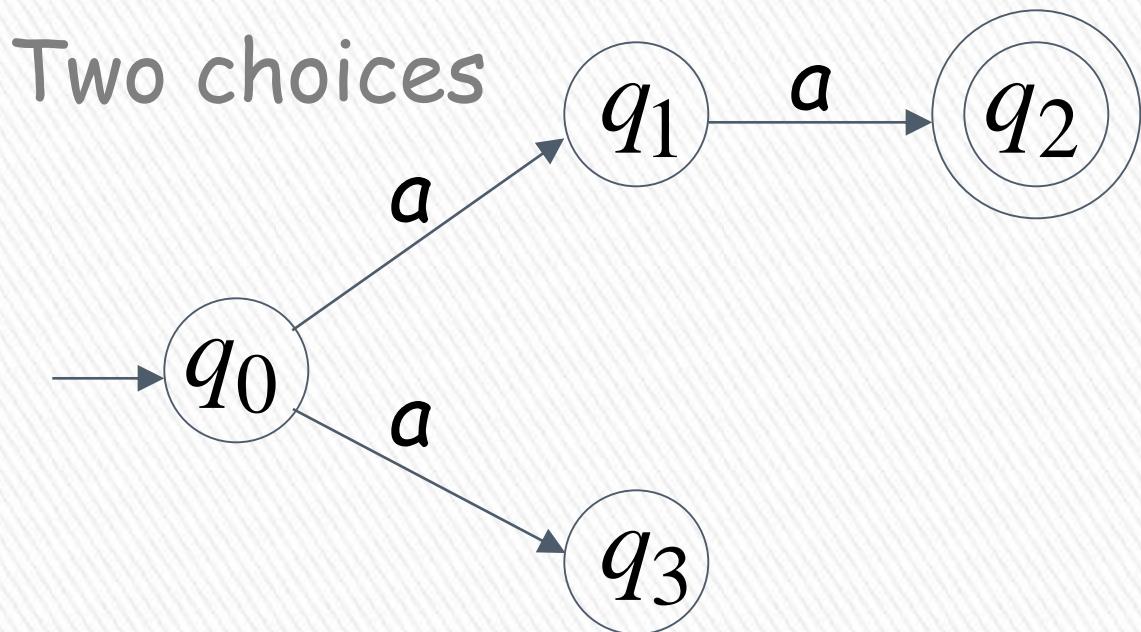


Nondeterministic Finite Automaton (NFA)

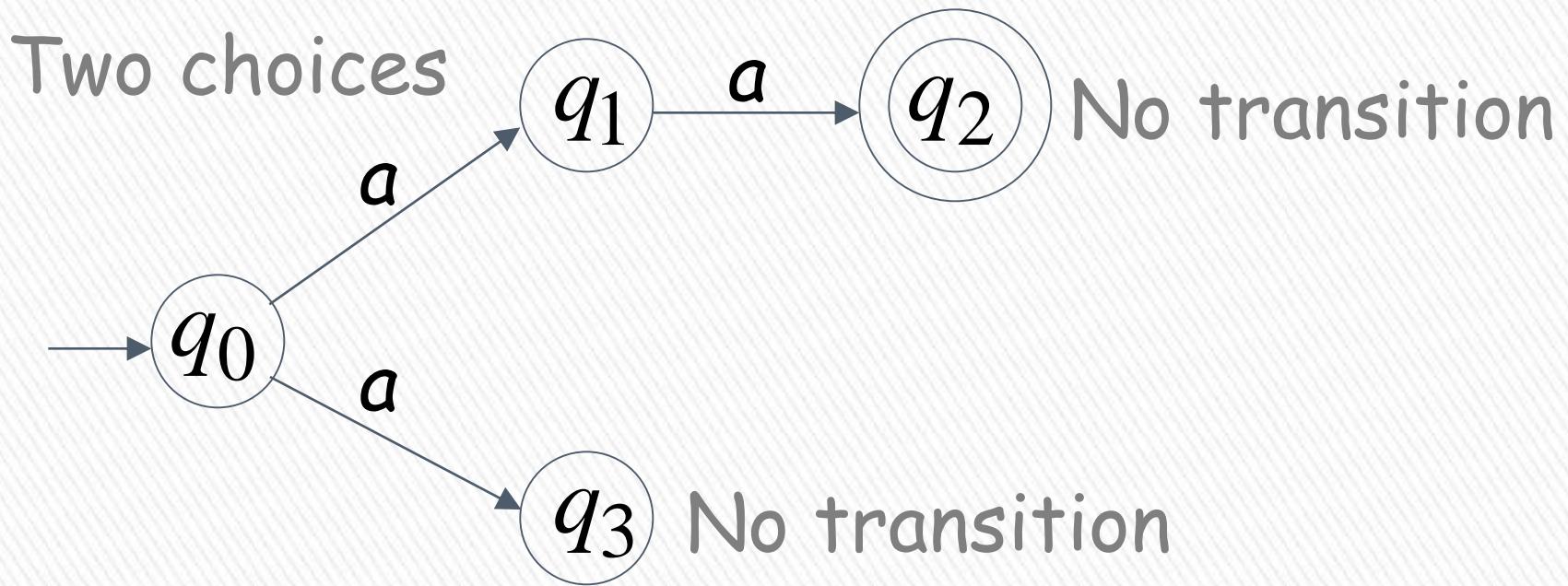
Alphabet = $\{a\}$



Alphabet = $\{a\}$



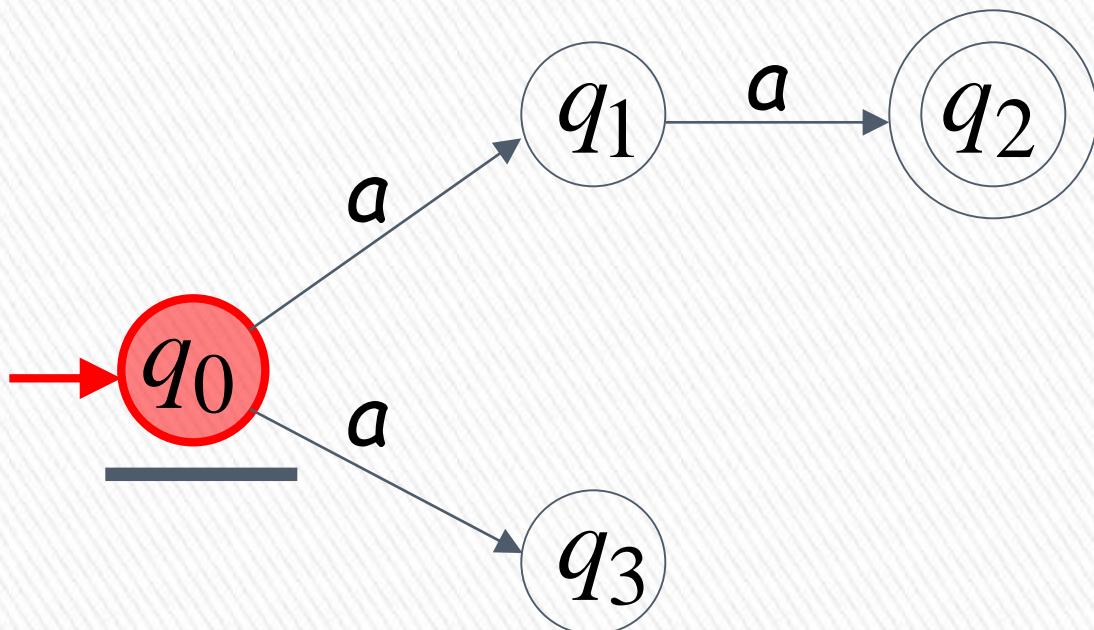
Alphabet = $\{a\}$



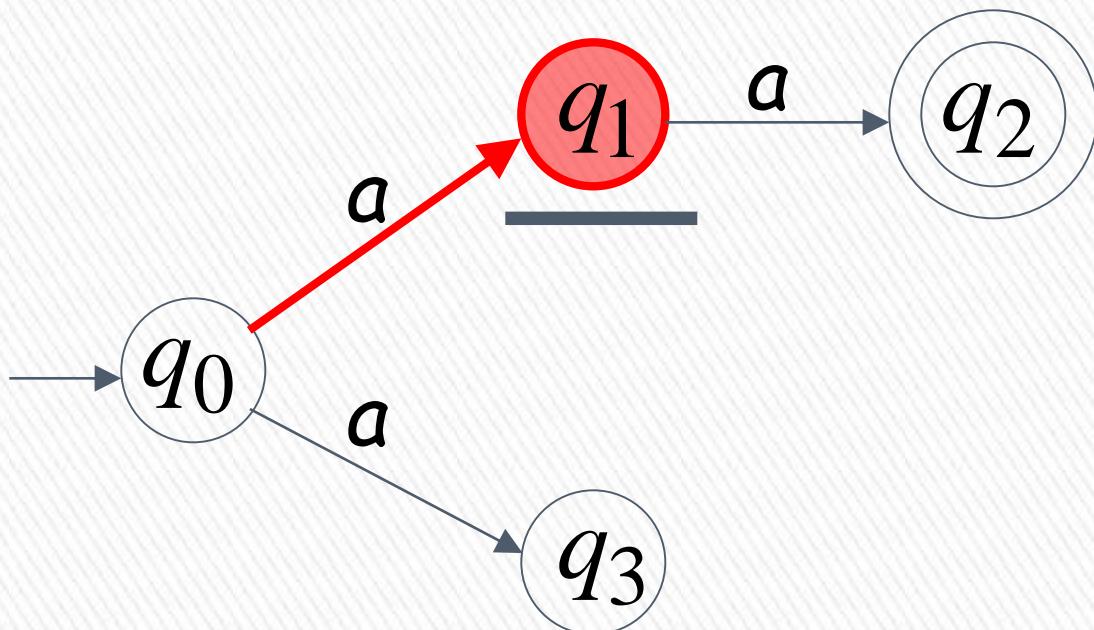
First Choice



a	a	
-----	-----	--



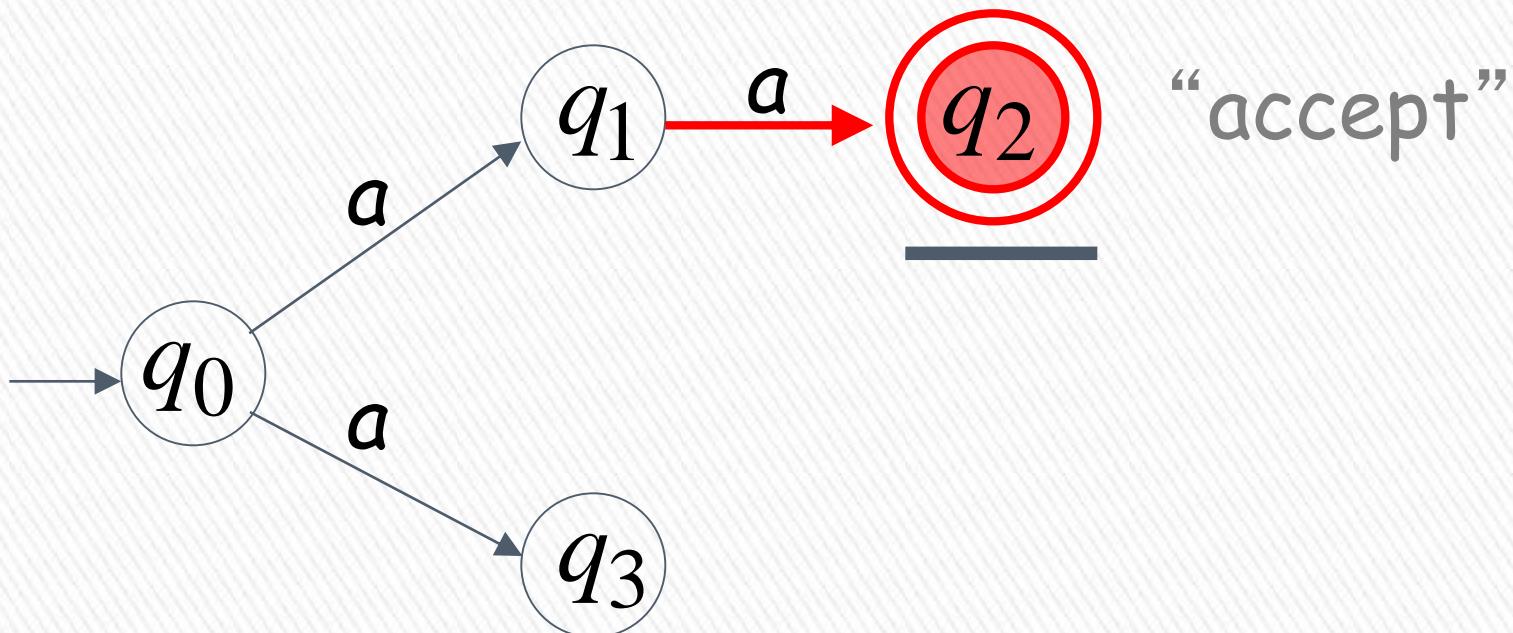
First Choice



First Choice



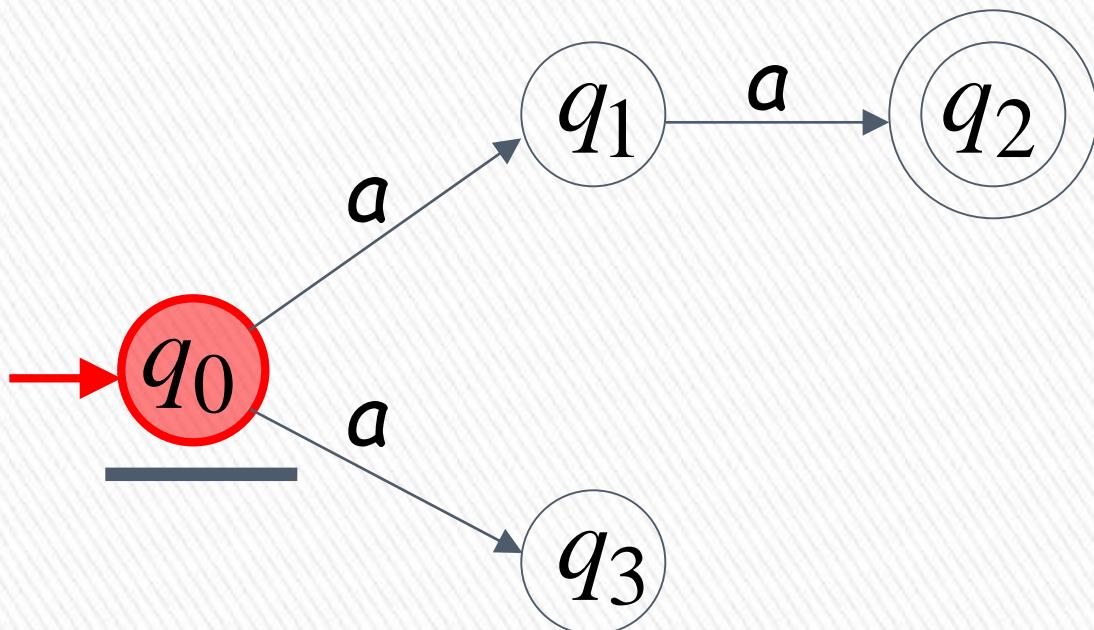
All input is consumed



Second Choice



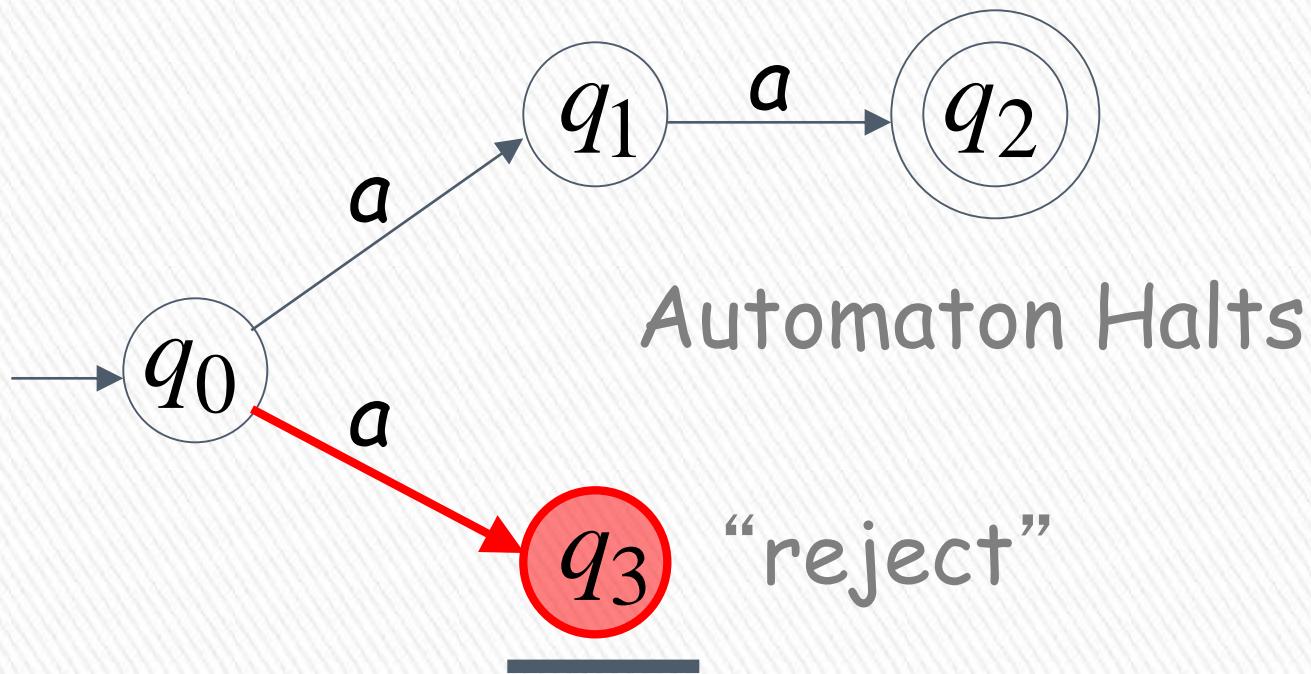
a	a	
-----	-----	--



Second Choice



Input cannot be consumed



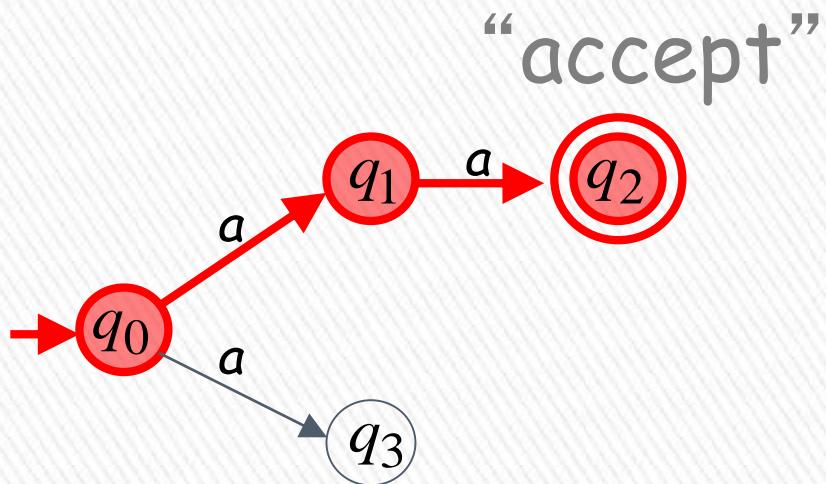
An NFA accepts a string:

if there exists a computation of the NFA
that accepts the string

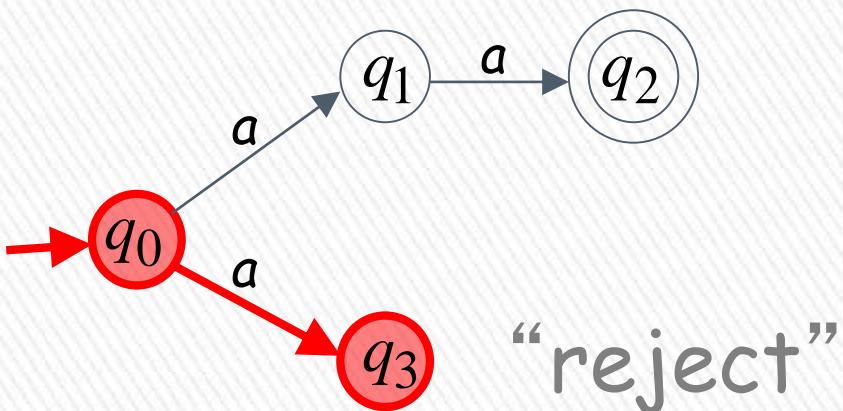
i.e., all the input string is processed
and the automaton is in an accepting
state



aa is accepted by the NFA:

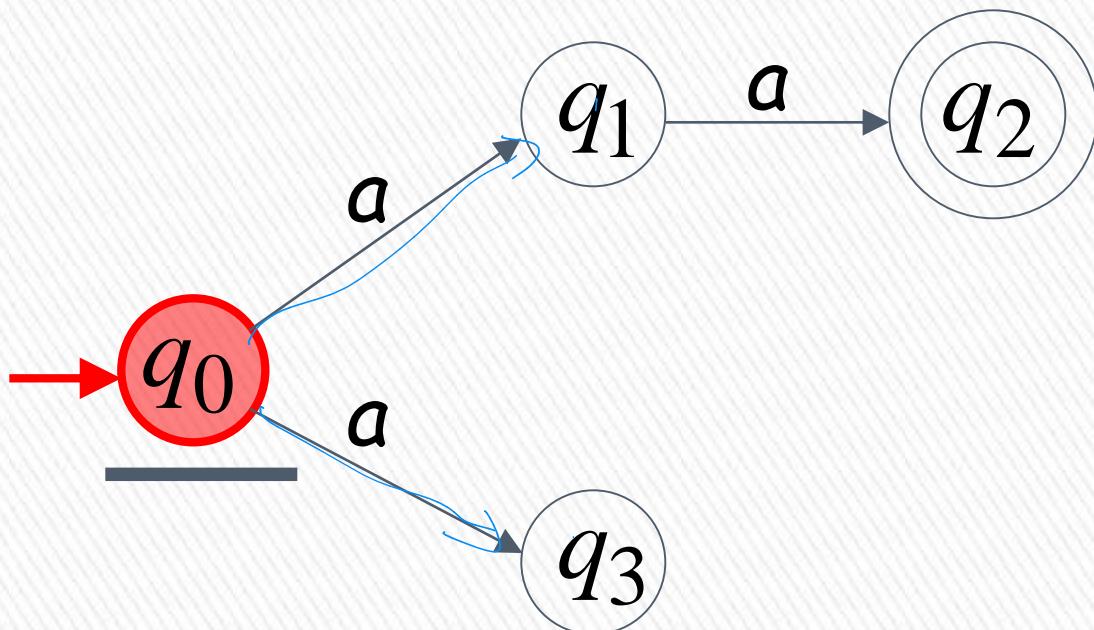


because this
computation
accepts aa

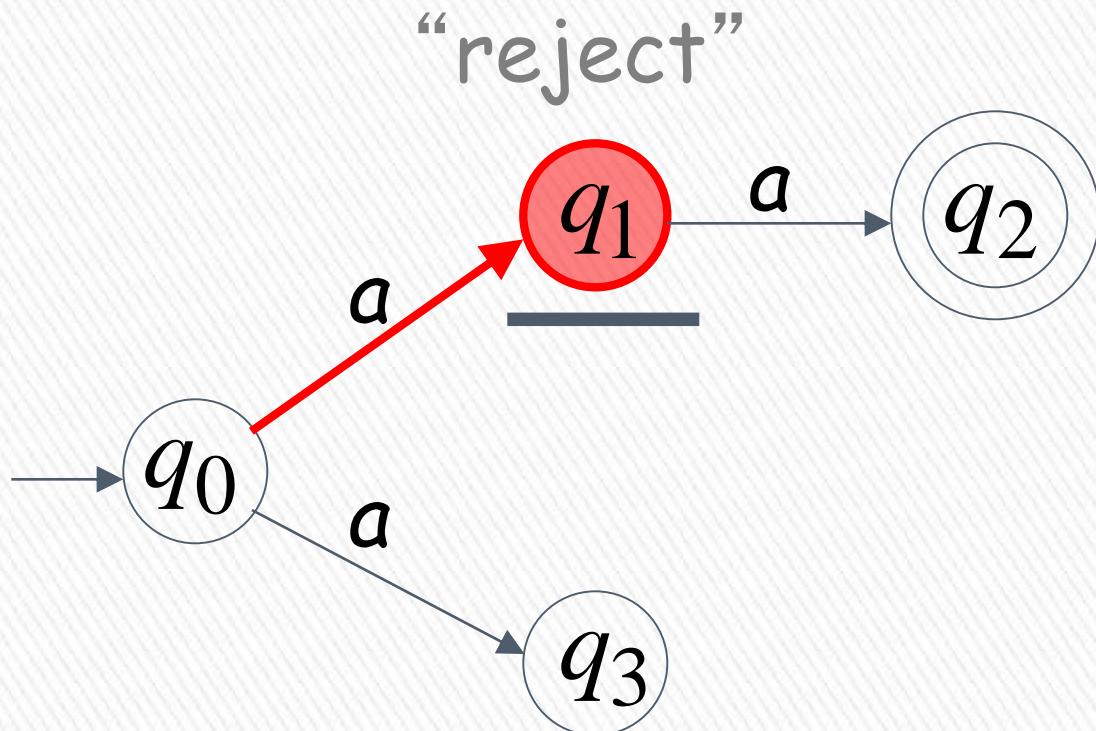


this computation
is ignored

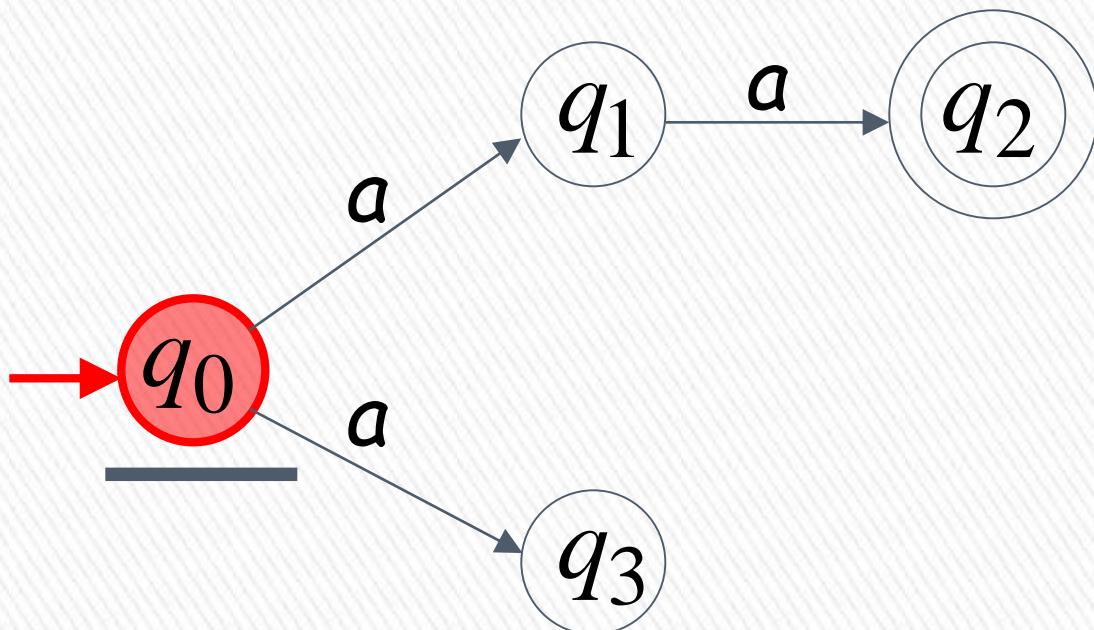
Rejection example



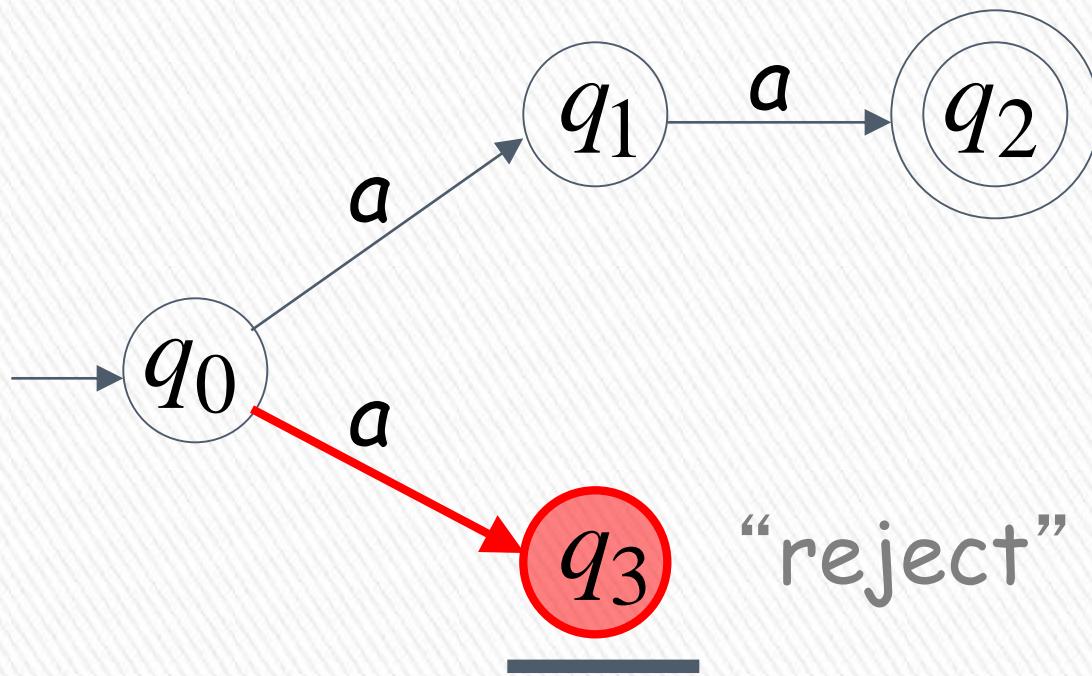
First Choice



Second Choice



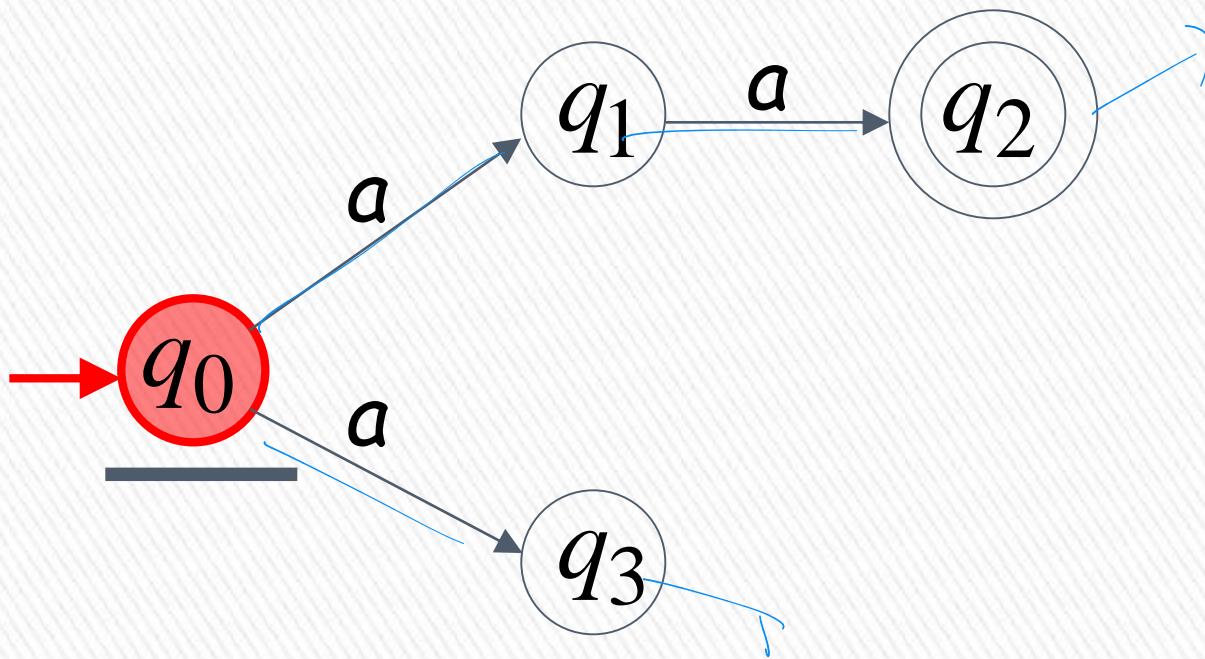
Second Choice



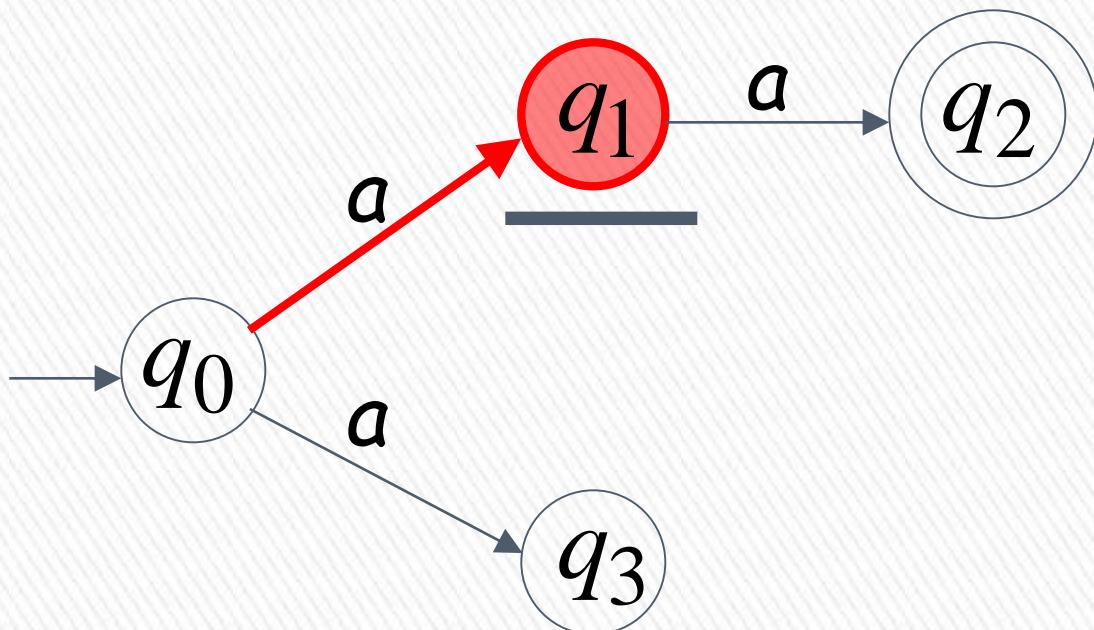
Another Rejection example



a	a	a	
---	---	---	--



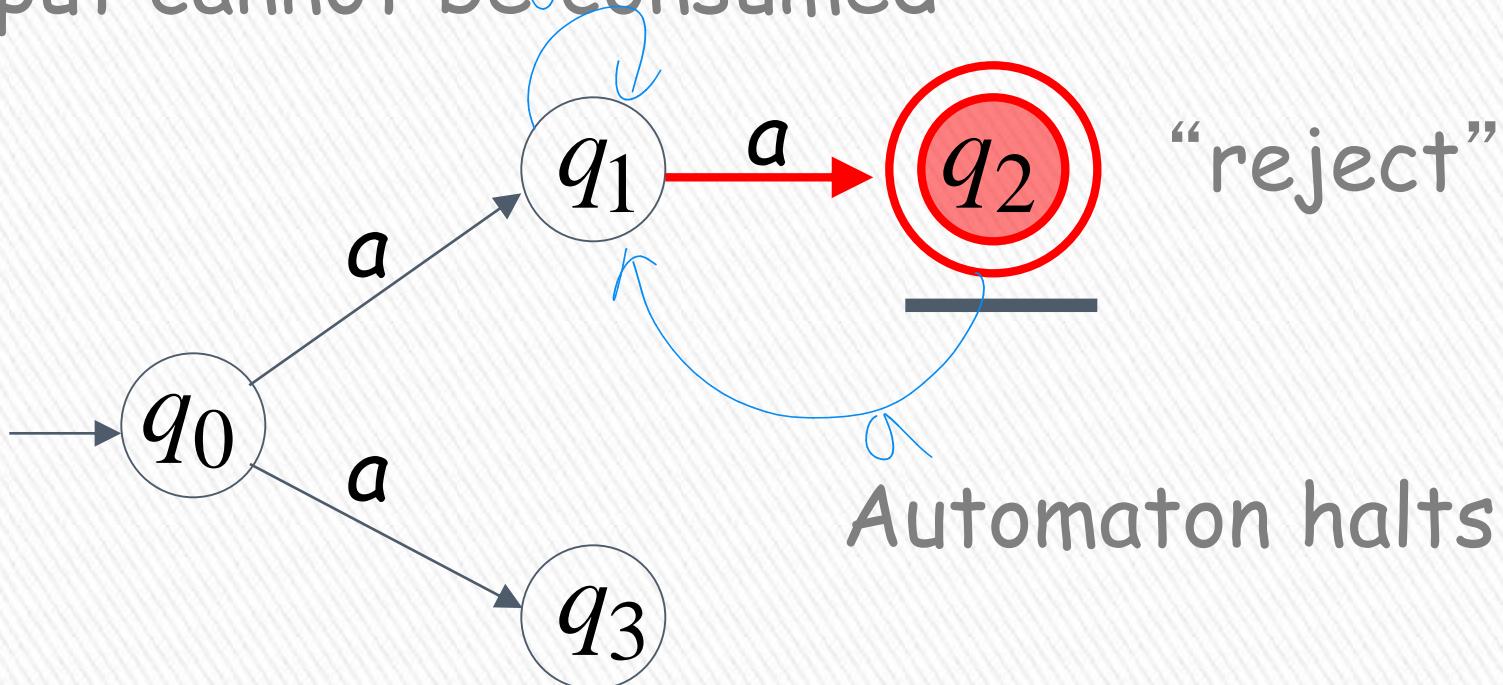
First Choice



First Choice



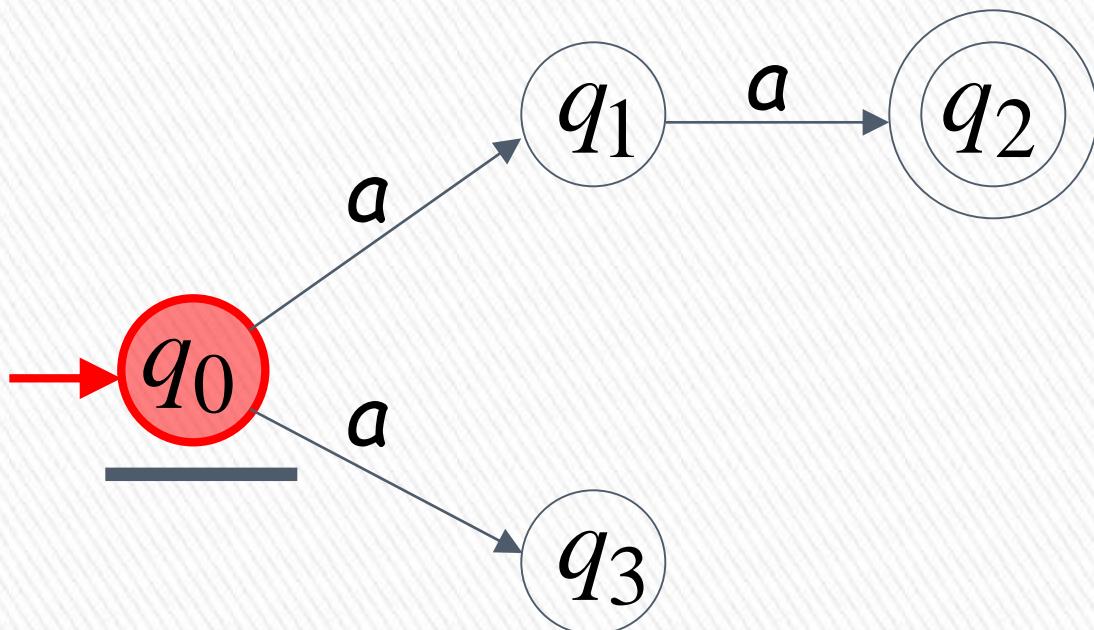
Input cannot be consumed



Second Choice



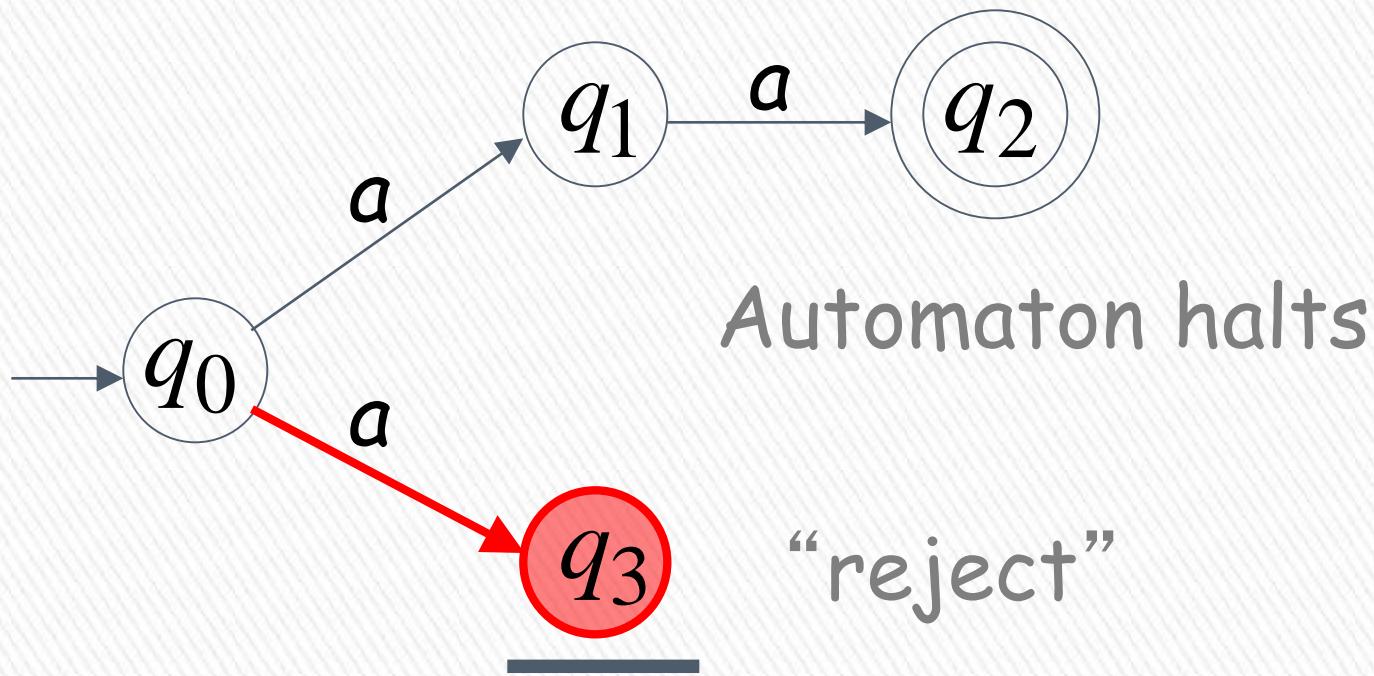
a	a	a	
-----	-----	-----	--



Second Choice



Input cannot be consumed



An NFA rejects a string:

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

For each computation:

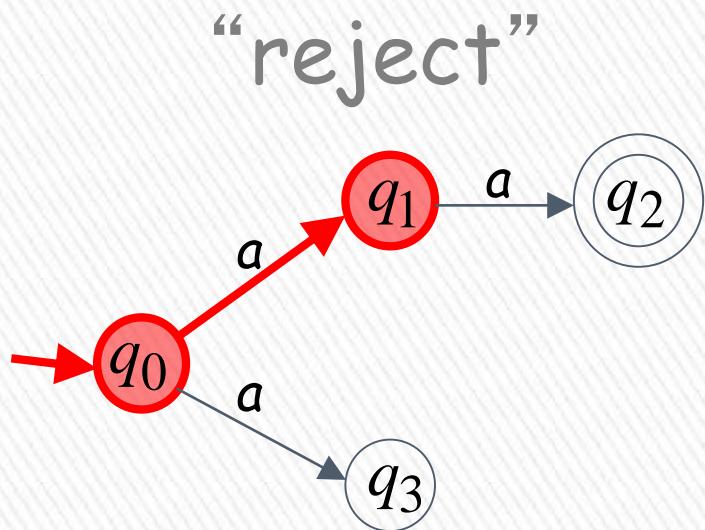
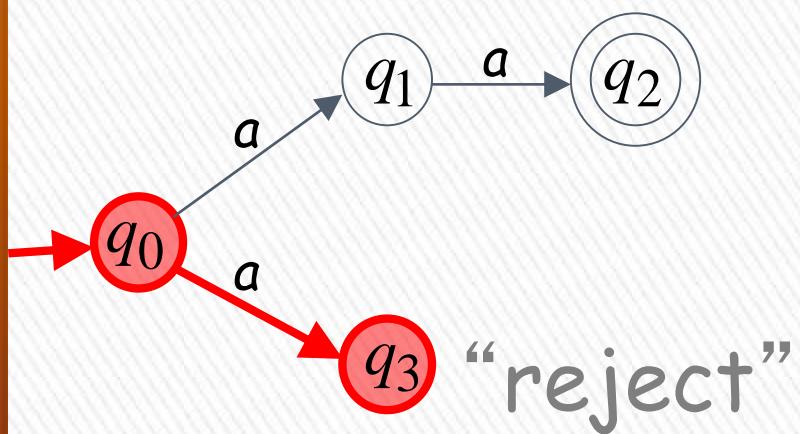
- All the input is consumed and the automaton is in a non final state

OR

- The input cannot be consumed

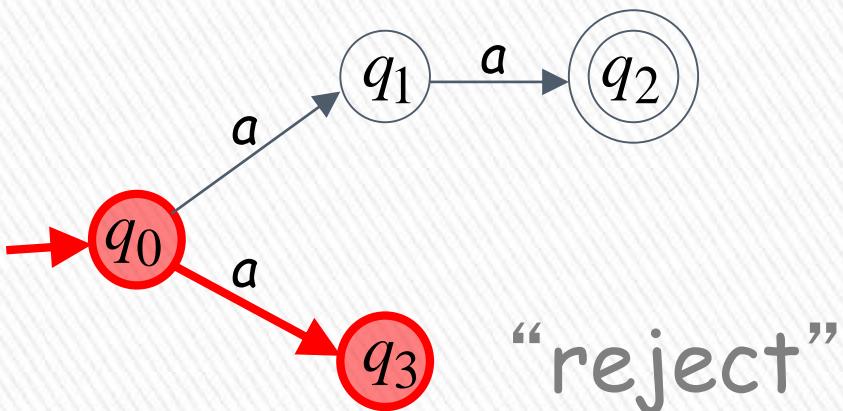
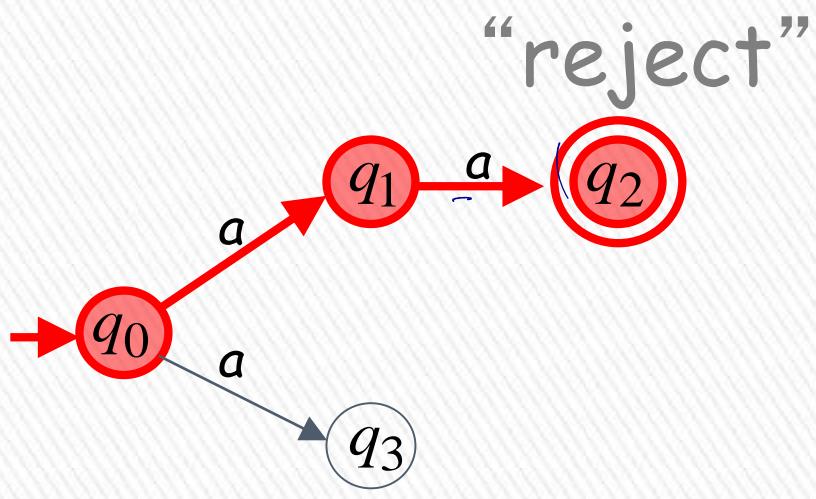


a is rejected by the NFA:



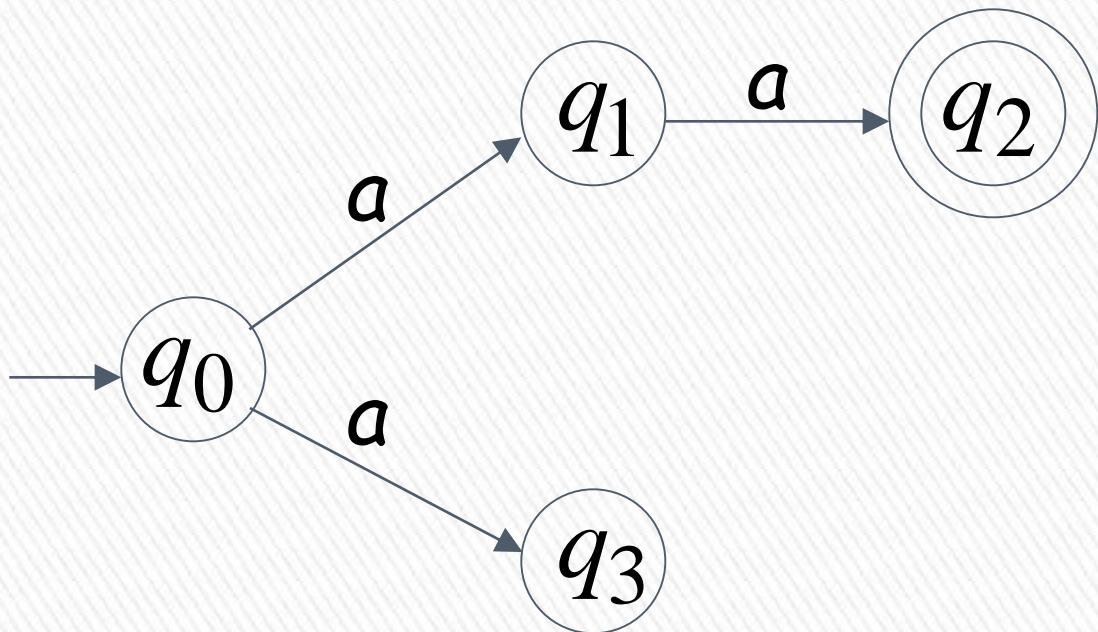
All possible computations lead to rejection

aaa is rejected by the NFA:

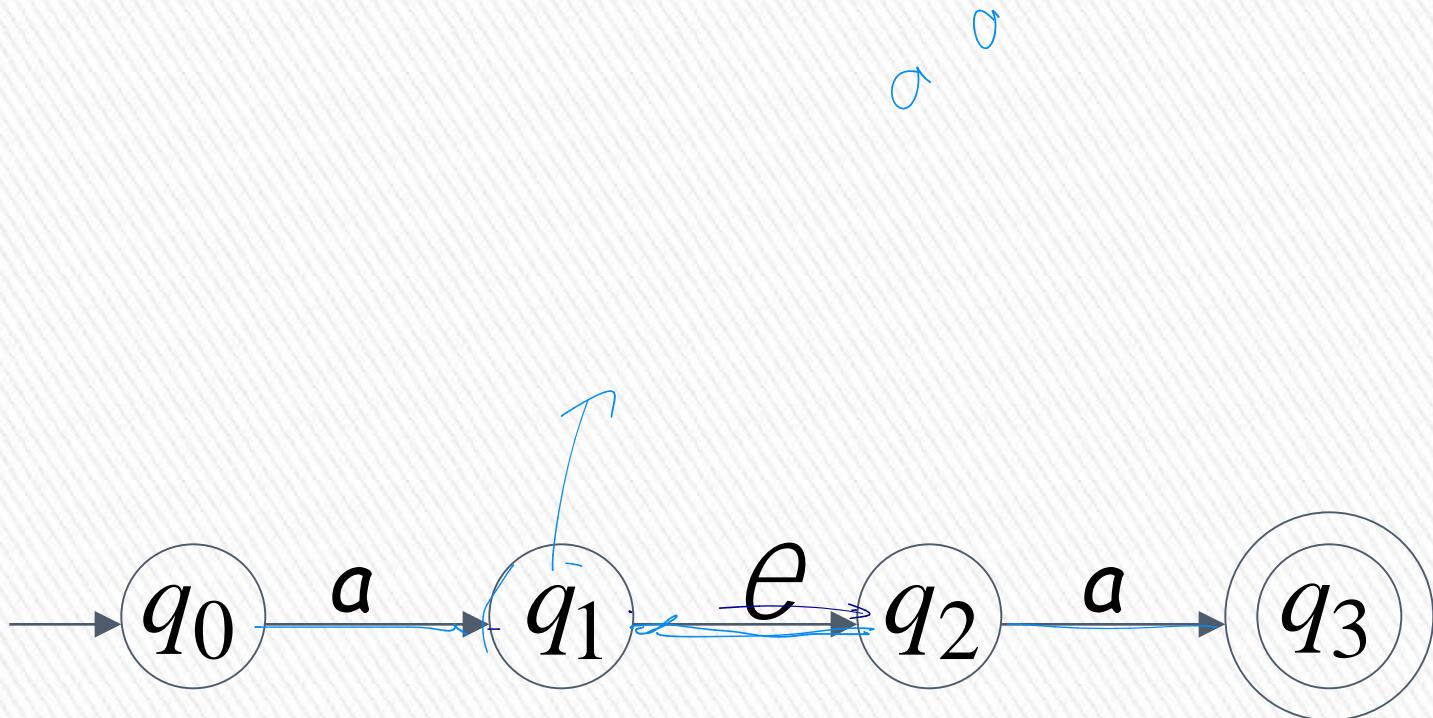


All possible computations lead to rejection

Language accepted: $L = \{aa\}$

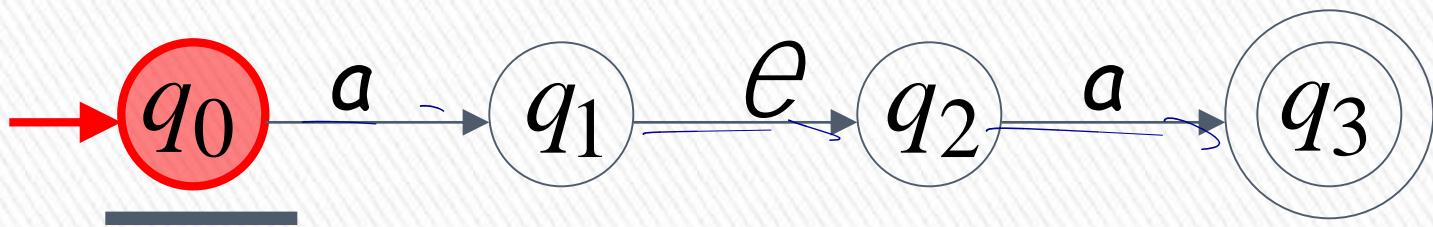


Epsilon Transition



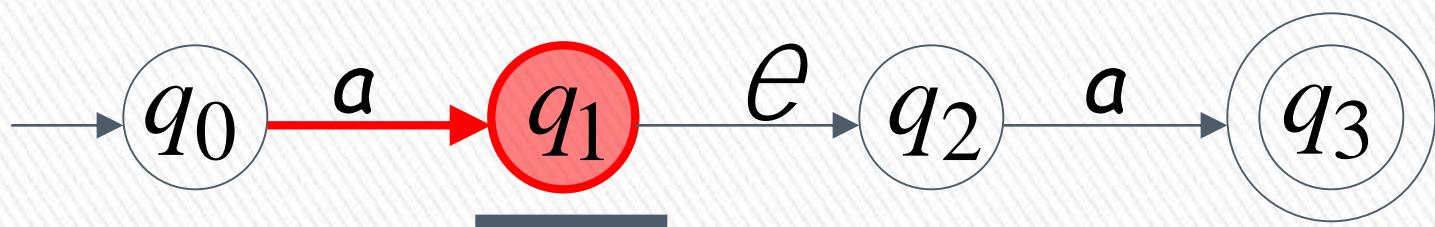


a	a	
-----	-----	--

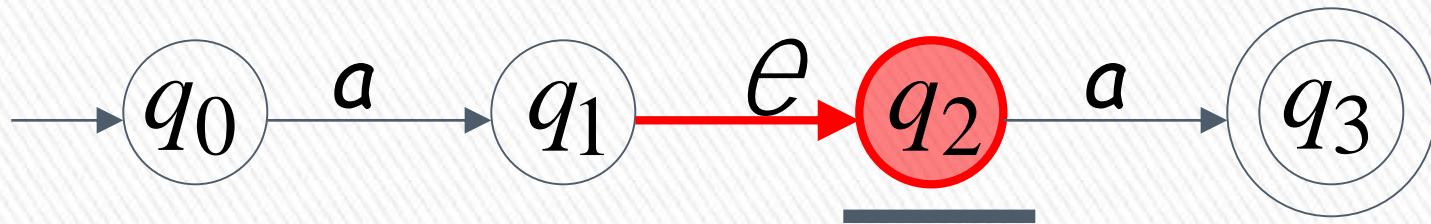


↓

a	a	
-----	-----	--



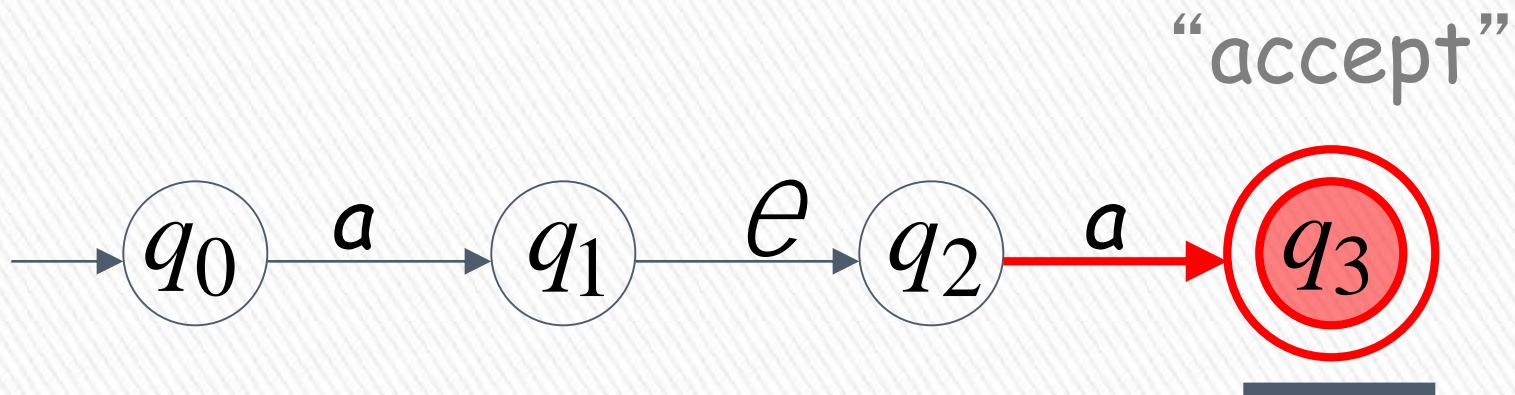
input tape head does not move



all input is consumed



a	a	
---	---	--



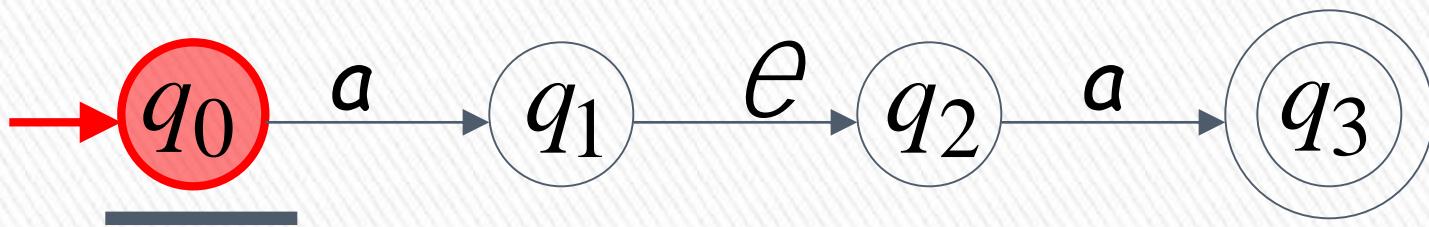
String aa is accepted

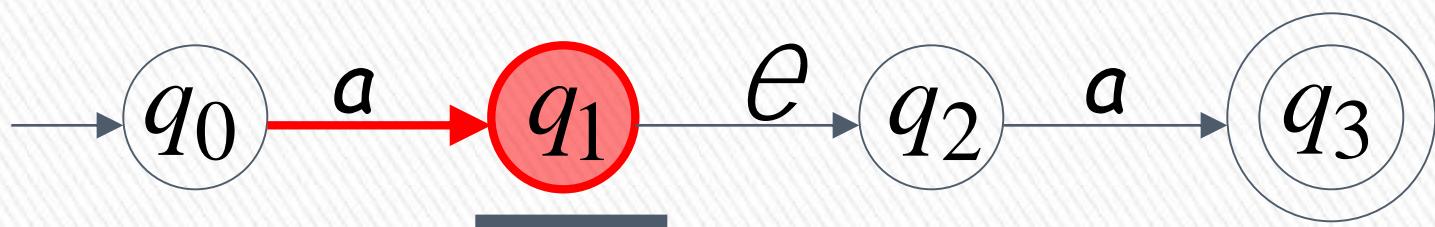


Rejection Example

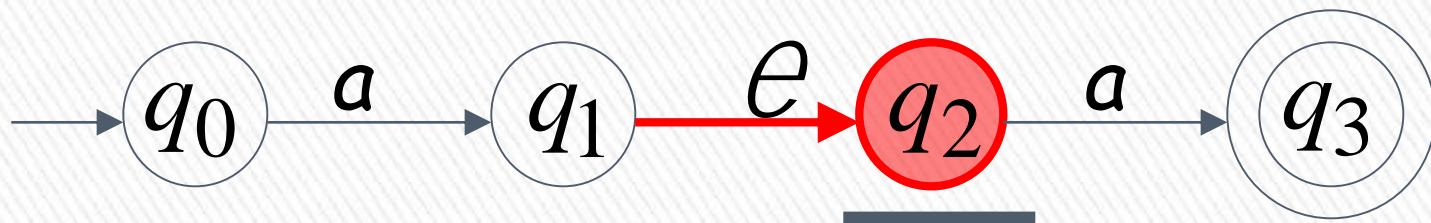


a	a	a	
---	---	---	--





(read head doesn't move)



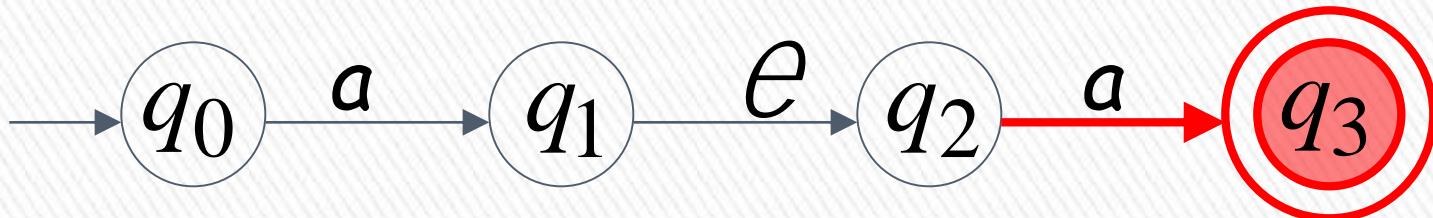
Input cannot be consumed



a	a	a	
---	---	---	--

Automaton halts

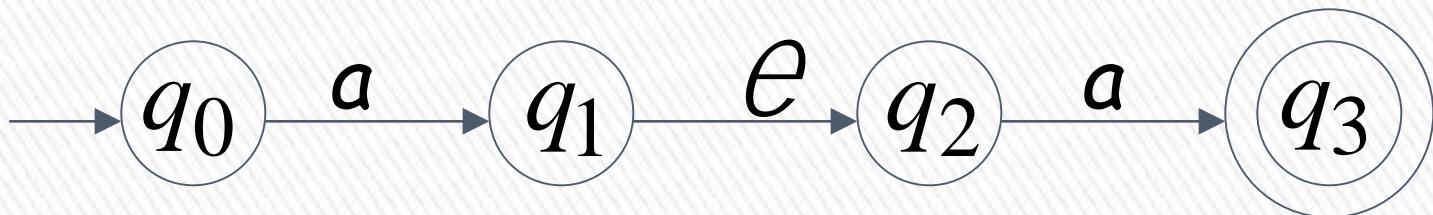
“reject”



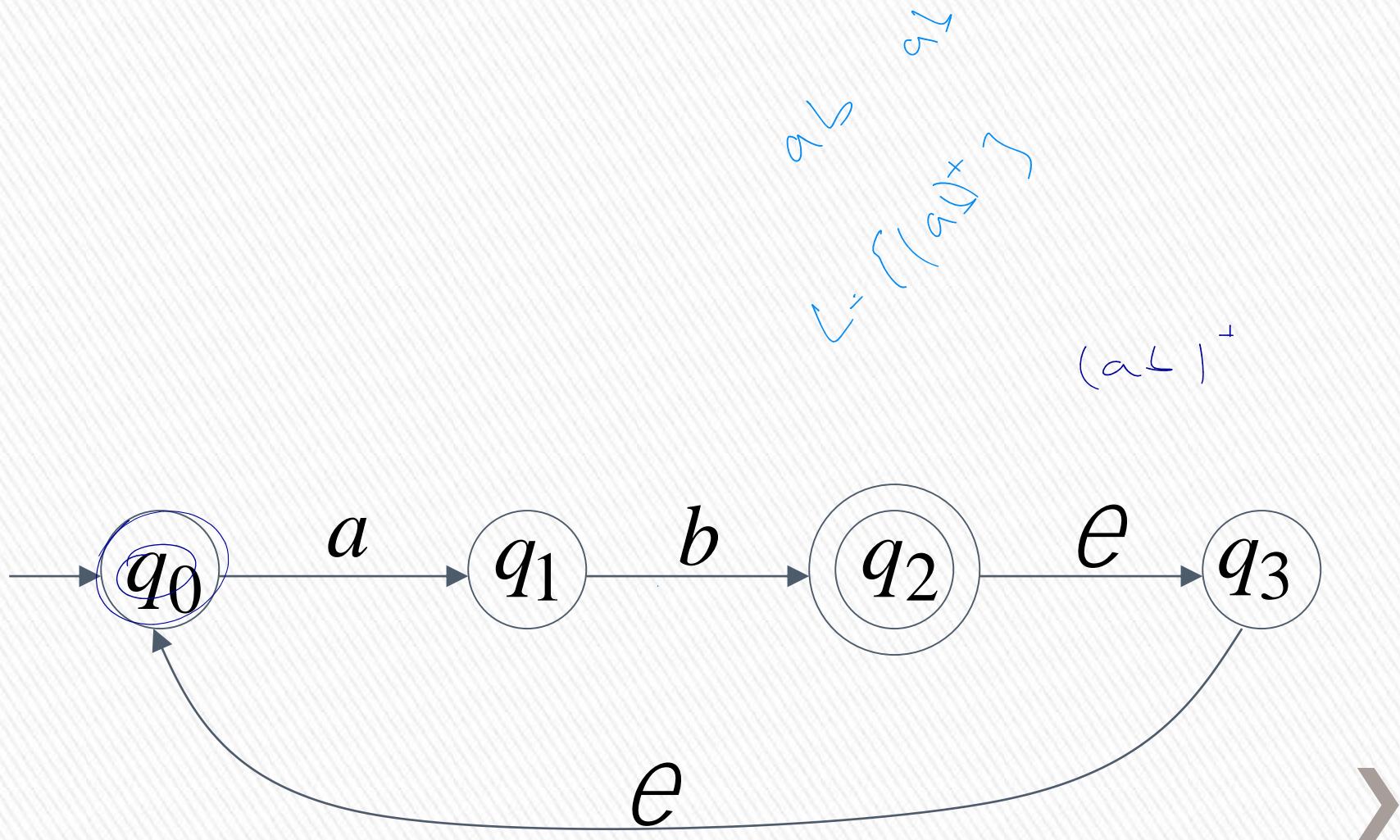
String **aaa** is rejected



Language accepted: $L = \{aa\}$

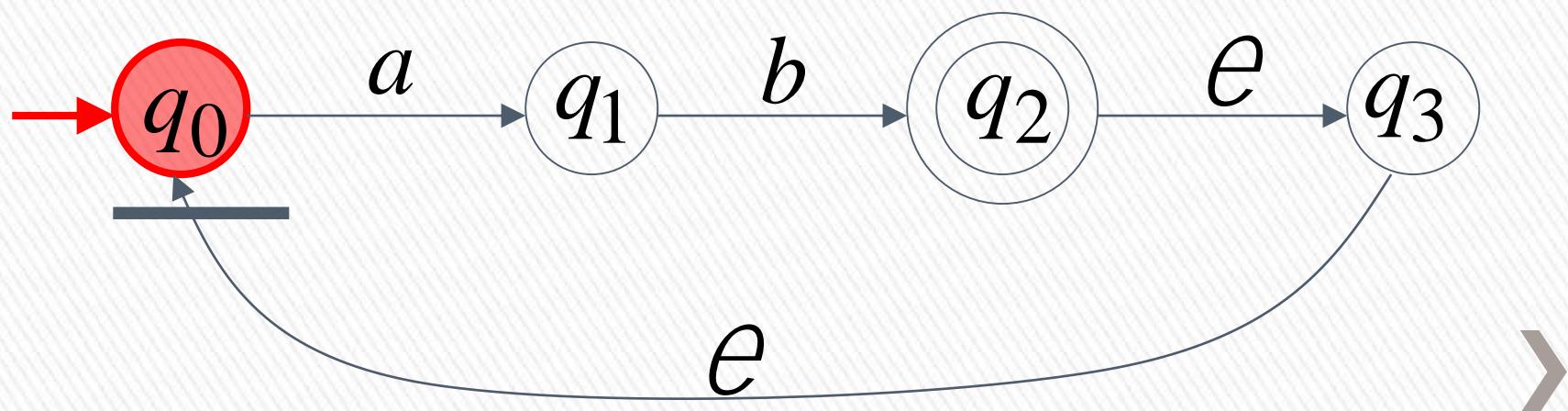


Another NFA Example

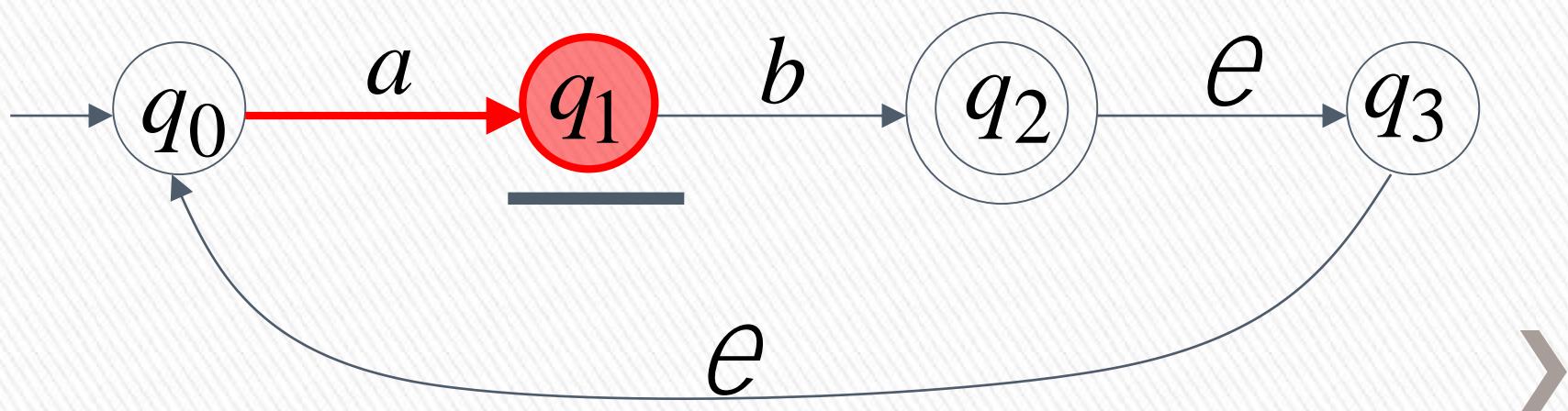




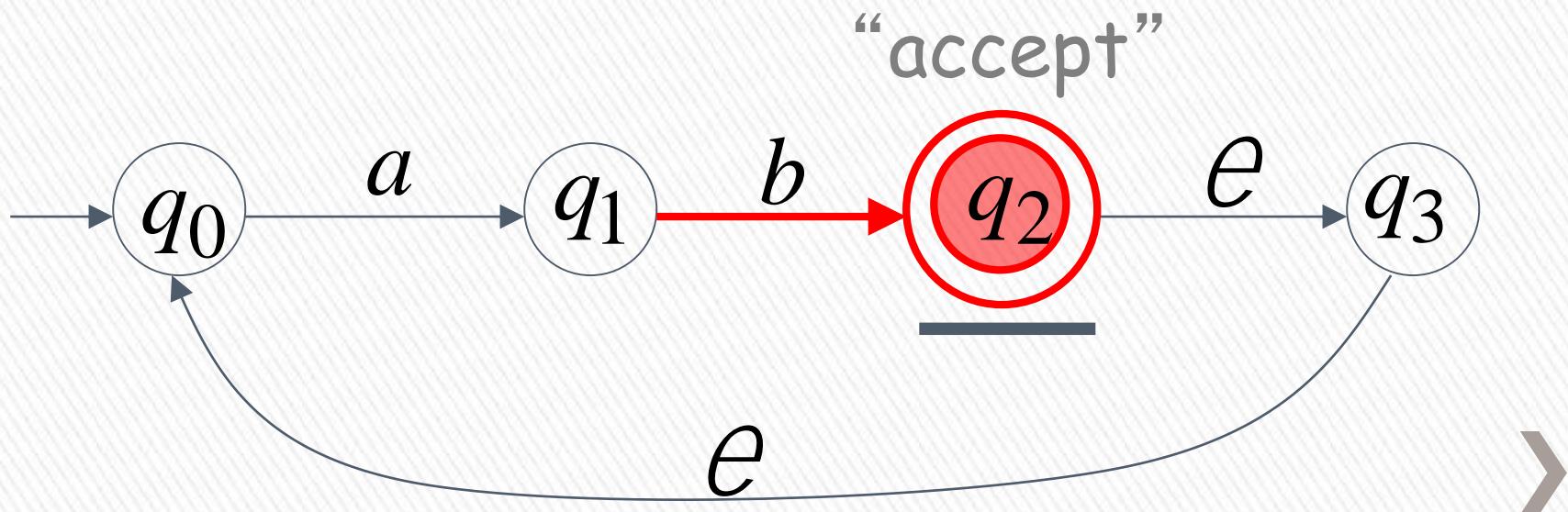
a	b	
-----	-----	--



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



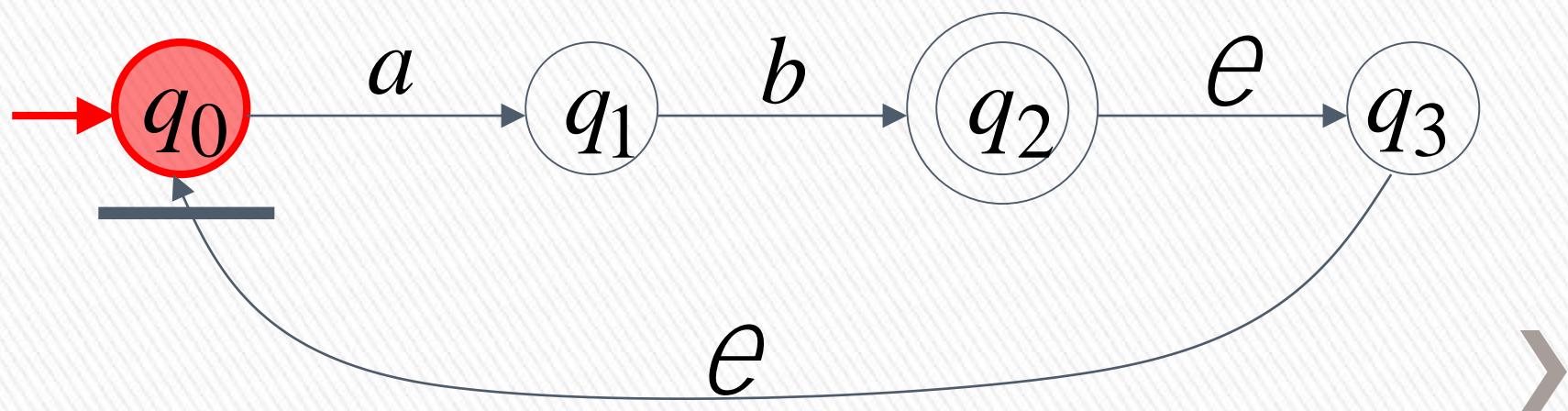
a	b	
-----	-----	--

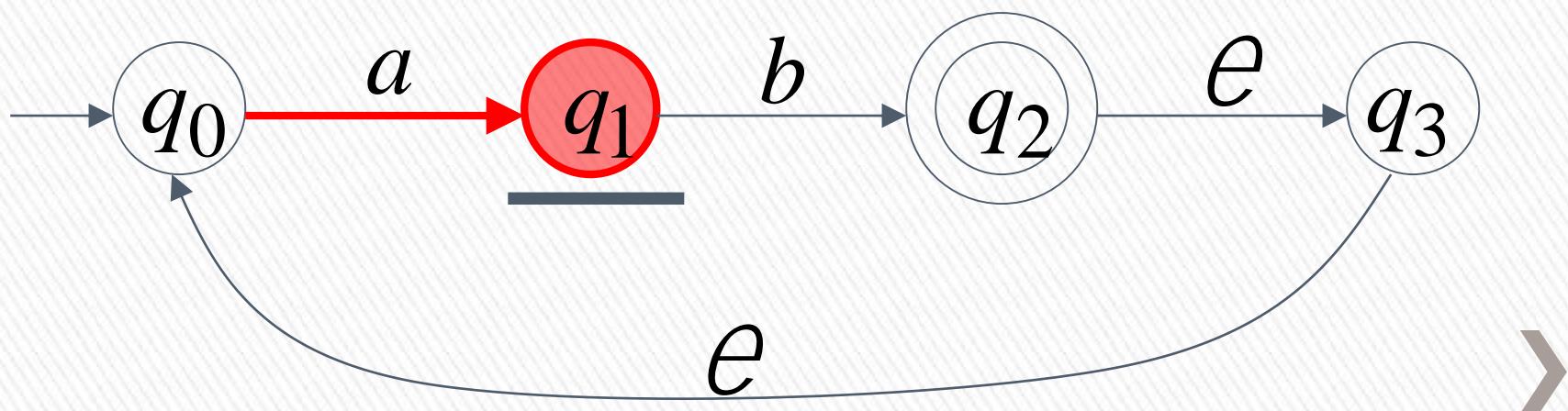


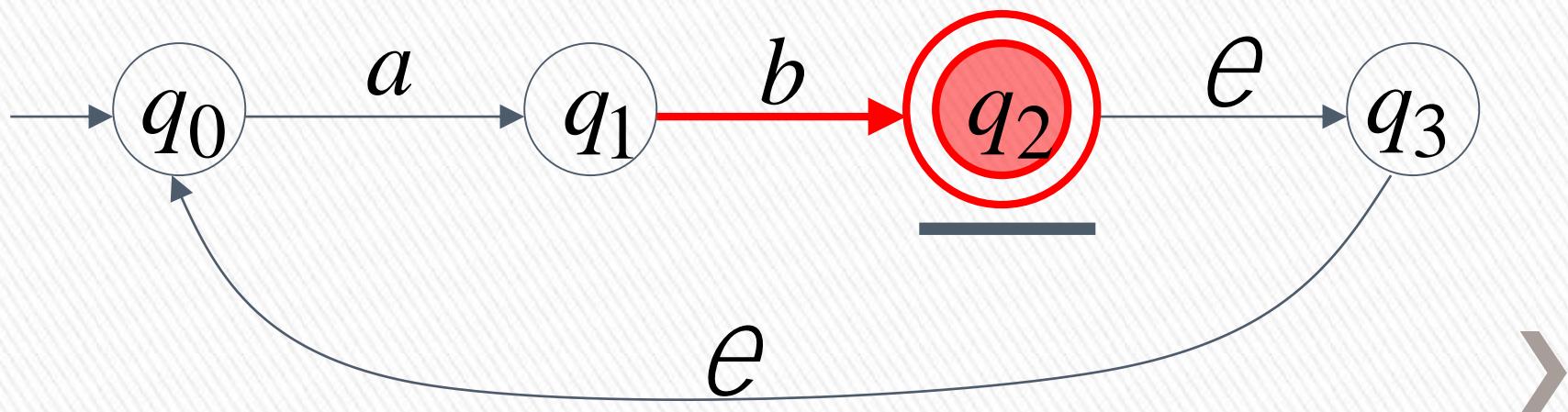
Another String

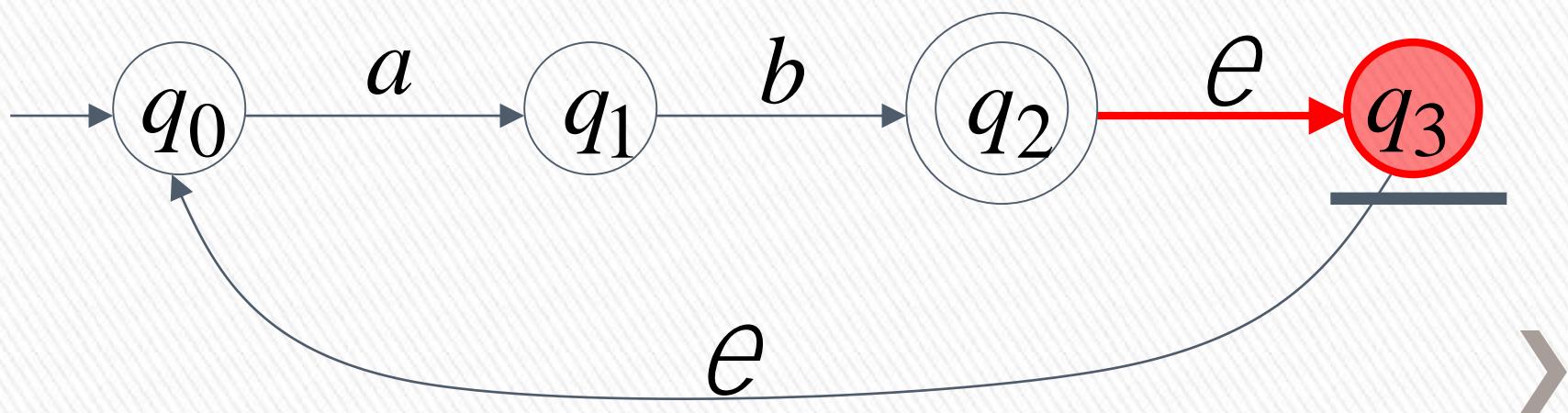


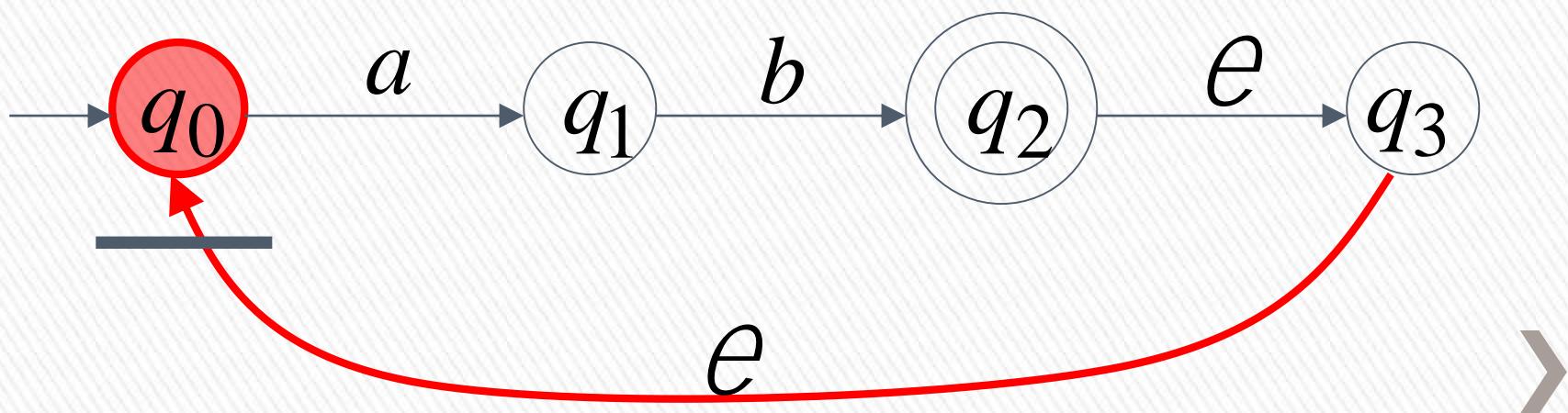
a	b	a	b	
---	---	---	---	--

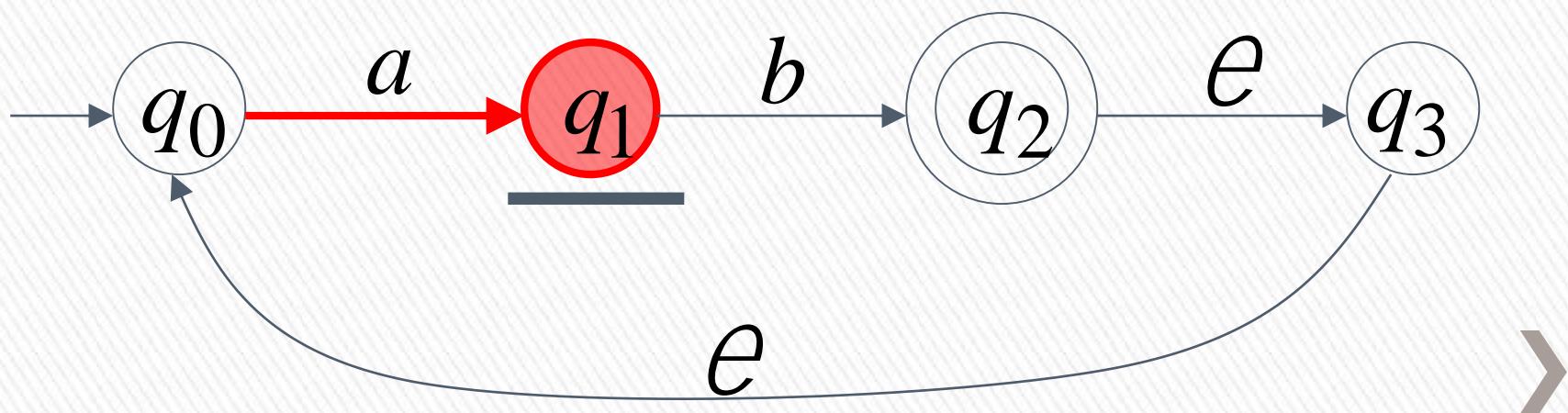


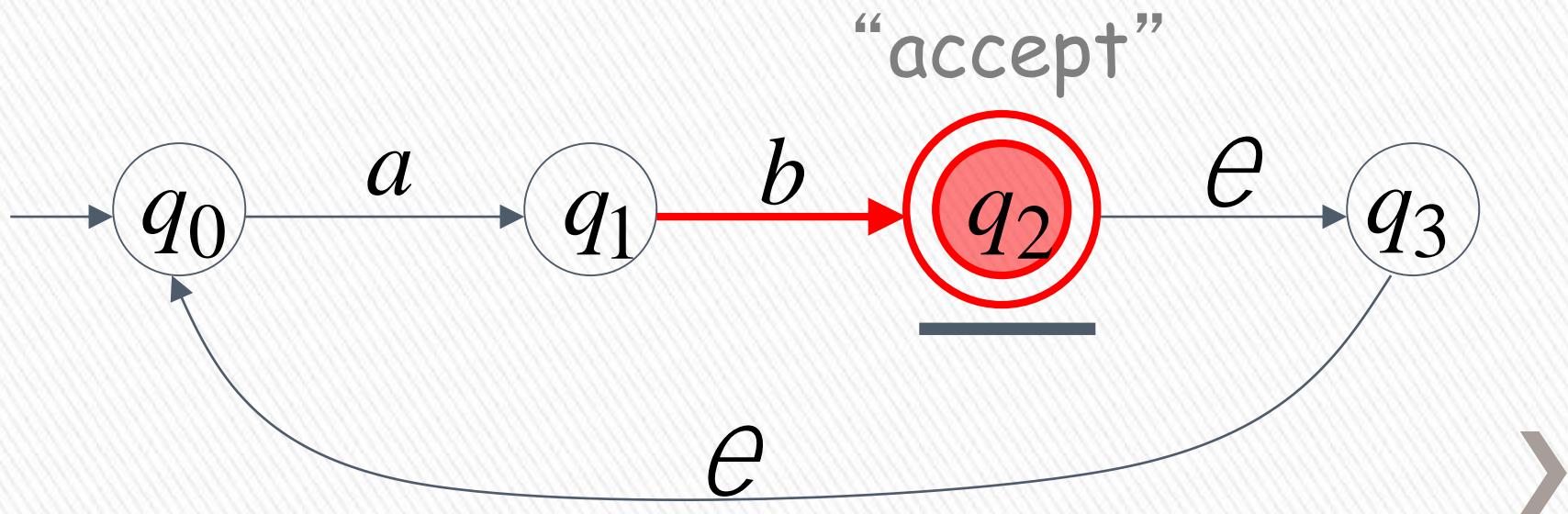








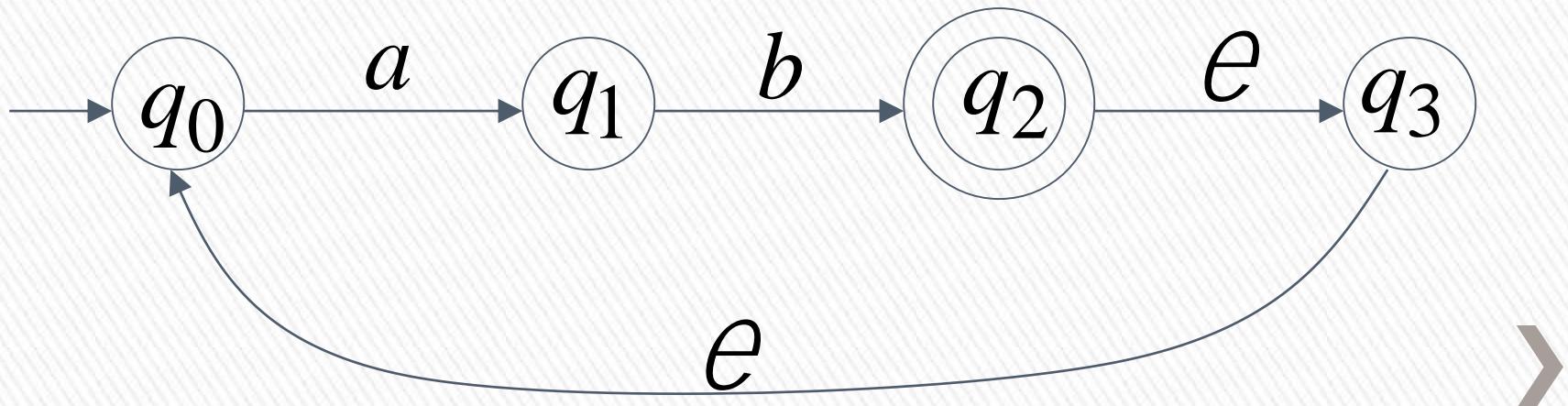




Language accepted

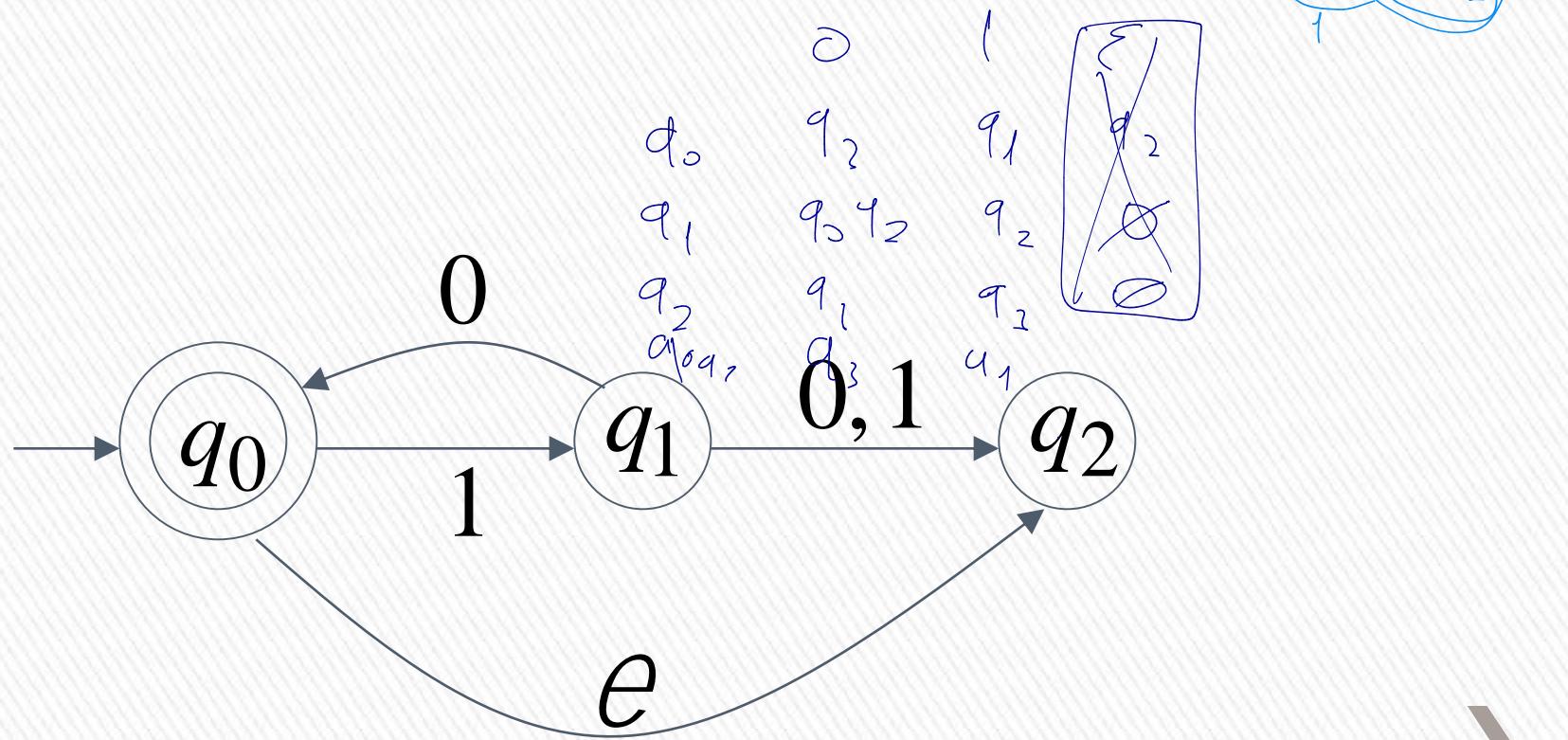
$$L = \{ab, abab, ababab, \dots\}$$

$$= \{ab\}^+$$

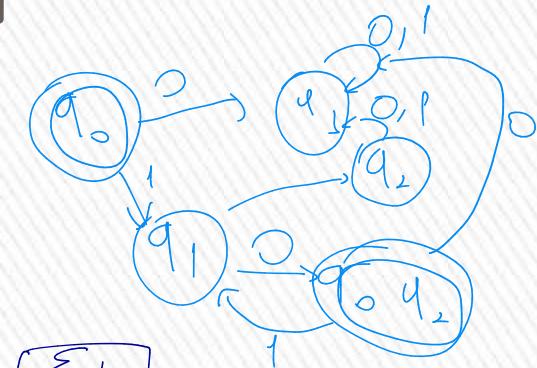


Another NFA Example

	0	1	ϵ
q_0	q_5	q_1	q_2
q_1	q_0, q_2	q_2	q_1
q_2	q_3	q_1	q_2

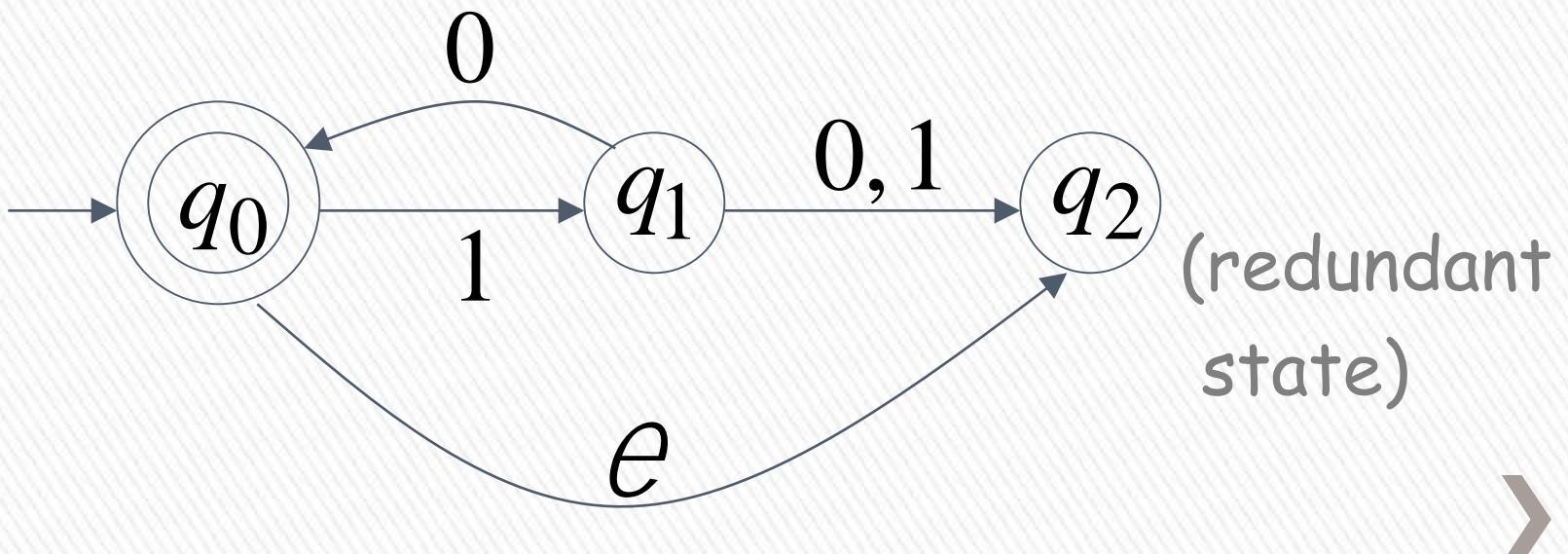


0	1	ϵ
q_0	q_2	q_1
q_1	q_0, q_2	q_2
q_2	q_1	q_1
q_3	q_1	q_1



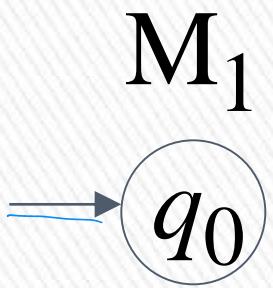
Language accepted

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

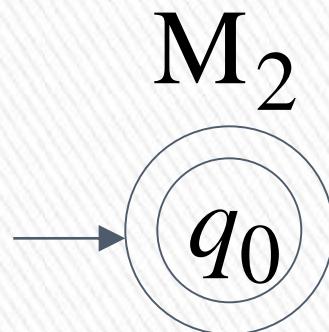


Remarks:

- The ϵ symbol never appears on the input tape
- Simple automata:



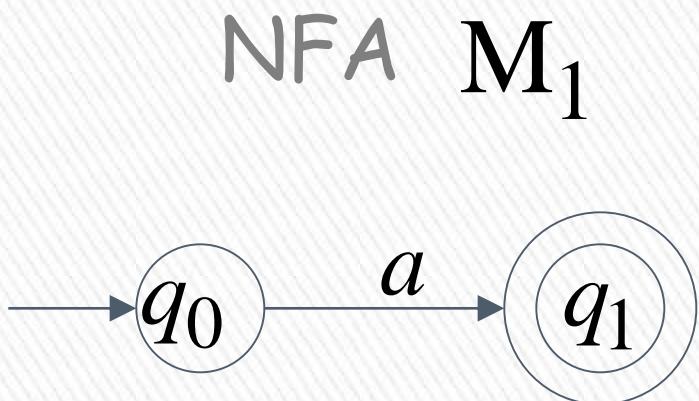
$$L(M_1) = \{ \}$$



$$L(M_2) = \{ \epsilon \}$$



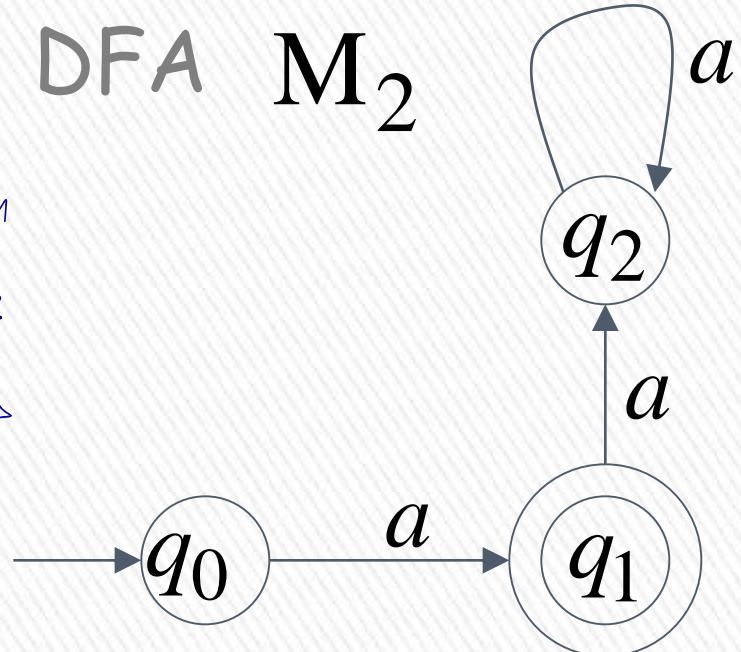
•NFAs are interesting because we can express languages easier than DFAs



DFA M_2

The diagram shows a Deterministic Finite Automaton (DFA) M_2 . It has four states: q_0 , q_1 , q_2 , and q_3 . q_0 is the start state (indicated by an incoming arrow). q_2 is the accepting state (indicated by a double circle). Transitions are as follows: $q_0 \xrightarrow{a} q_1$, $q_1 \xrightarrow{a} q_2$, $q_2 \xrightarrow{a} q_3$, and $q_3 \xrightarrow{a} q_2$.

q_0 q_1
 q_1 q_3
 q_2 q_2



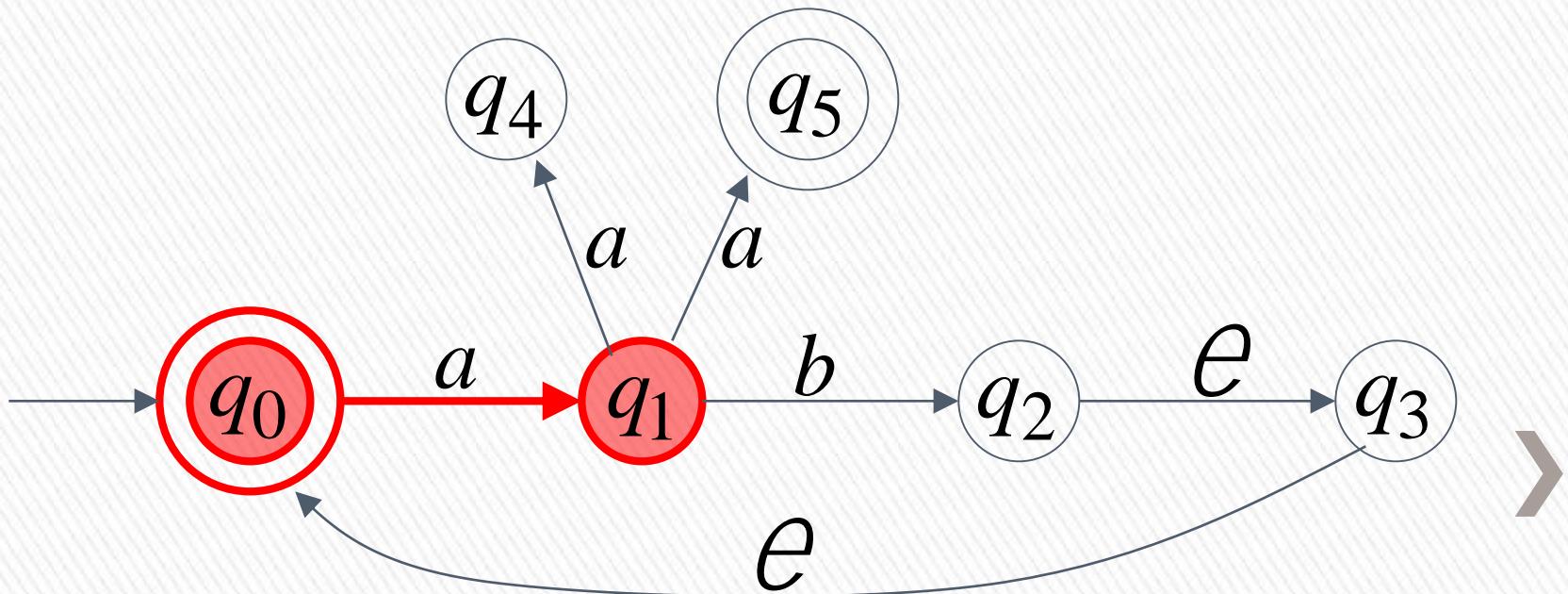
$$L(M_1) = \{a\}$$

$$L(M_2) = \{a\}$$

Extended Transition Function δ^*

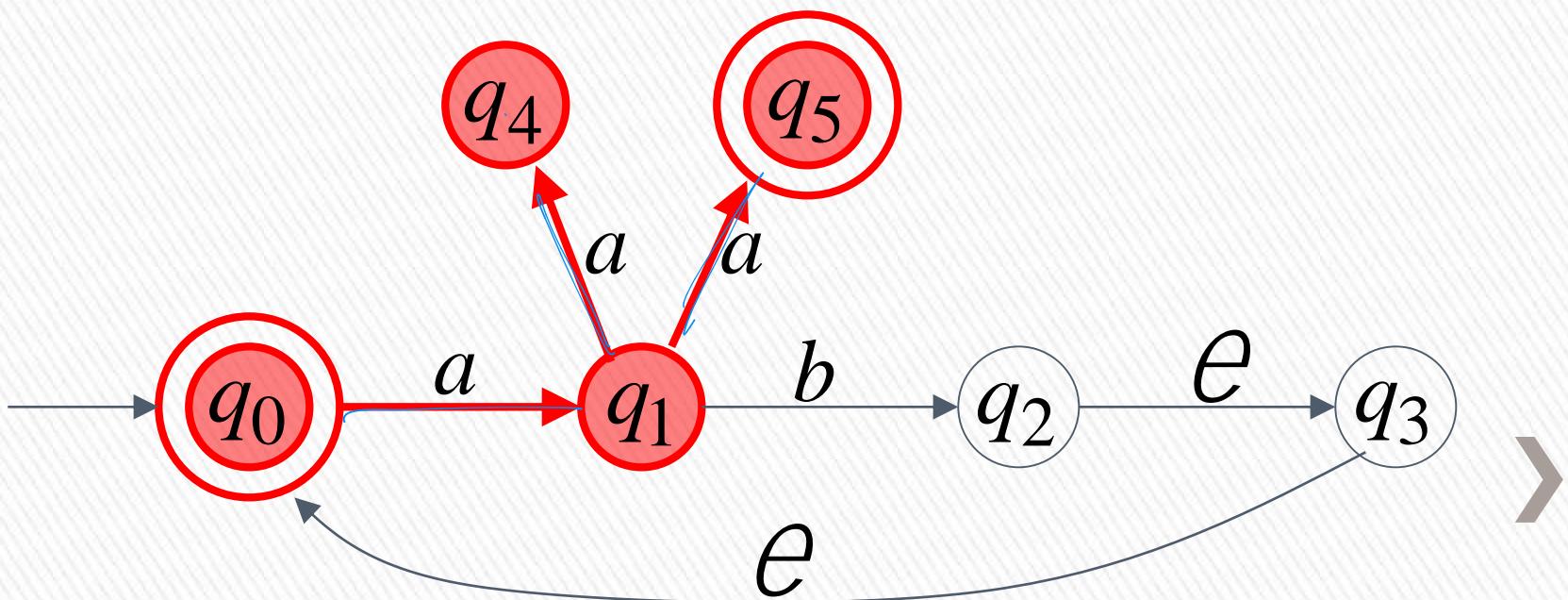
Same with δ but applied on strings

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

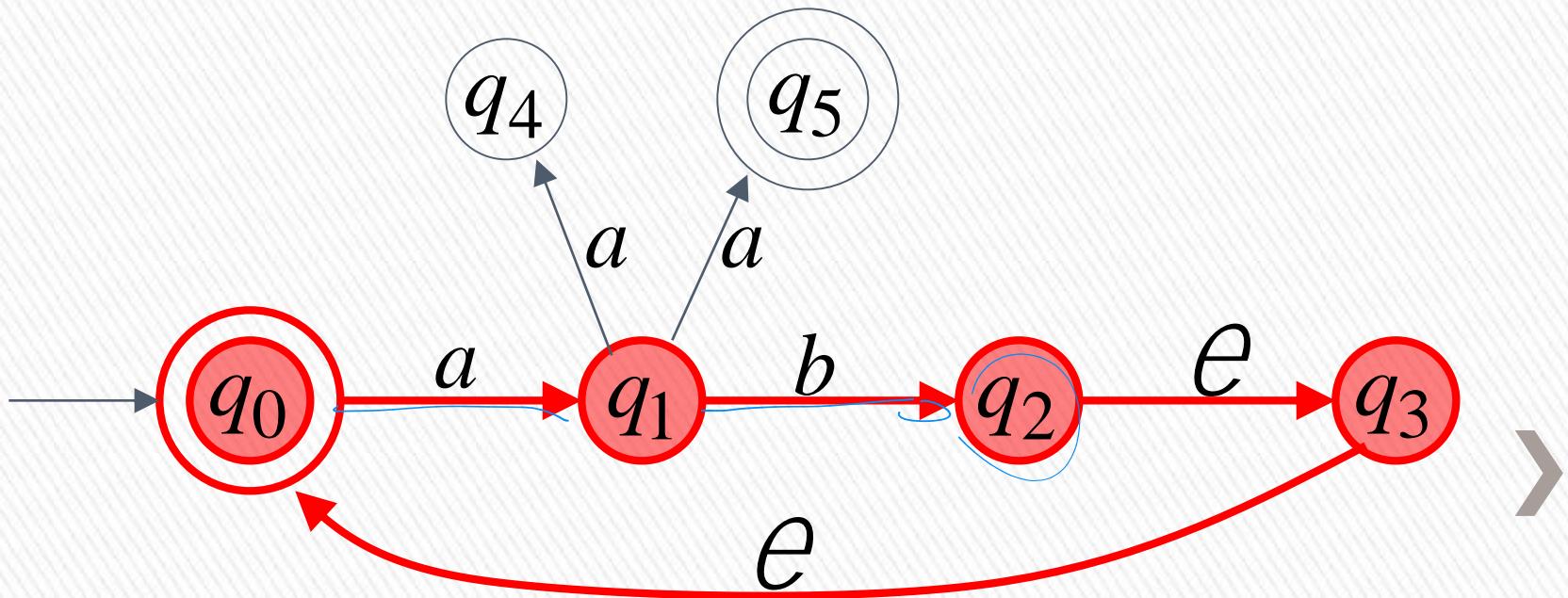


$$\delta^*(q_0, aa) = \{q_4, q_5\}$$

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



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



In general

$q_j \in \delta^*(q_i, w)$: there is a walk from q_i to q_j
with label w



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



The Language of an NFA

» The language accepted by M is:

$$L(M) = \{w_1, w_2, \dots, w_n\}$$

» where

$$\delta^*(q_0, w_m) = \{q_i, \dots, q_k, \dots, q_j\}$$

» and there is some $q_k \in F$

(accepting state)

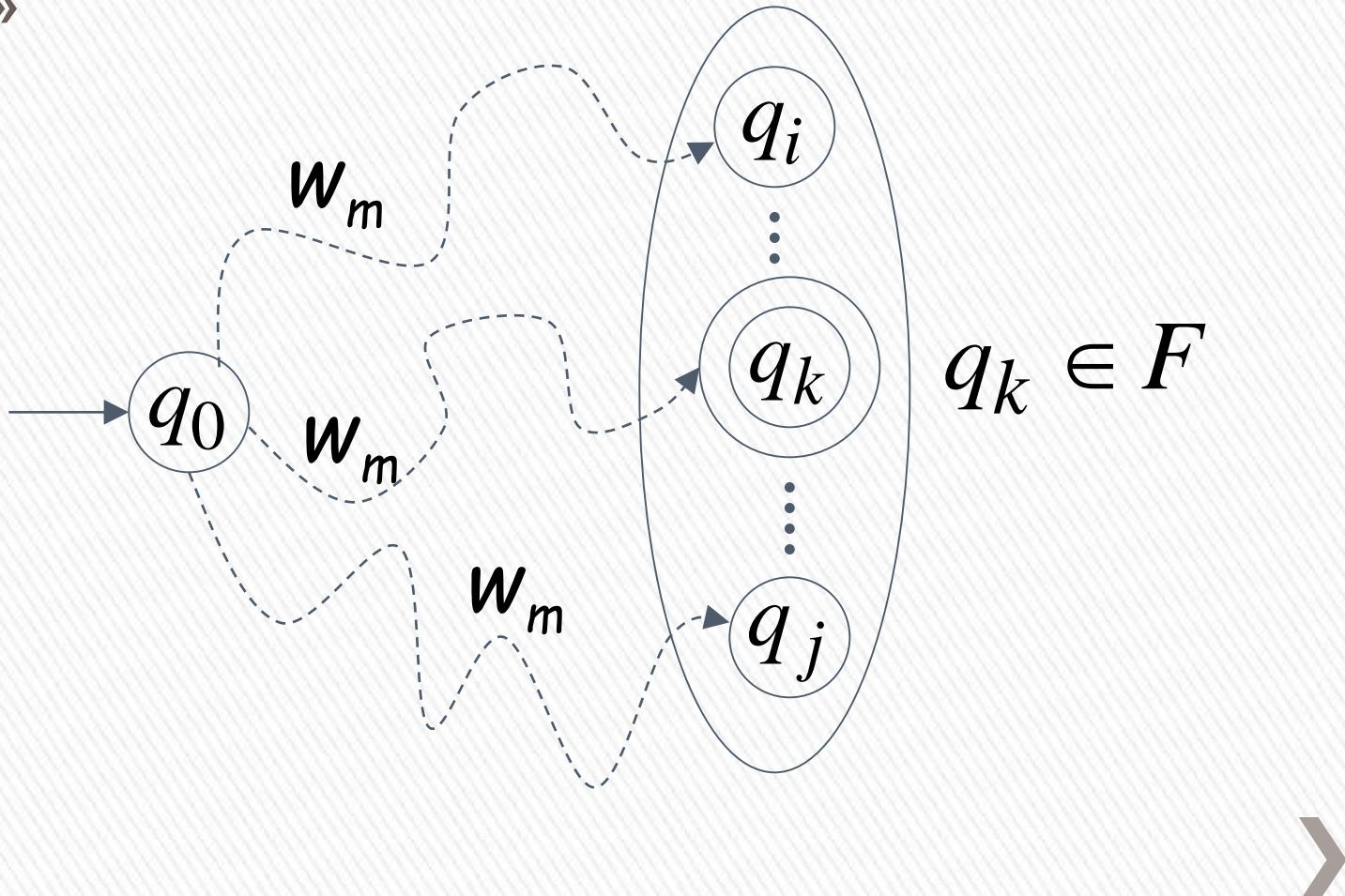


$w_m \in L(M)$

»

»

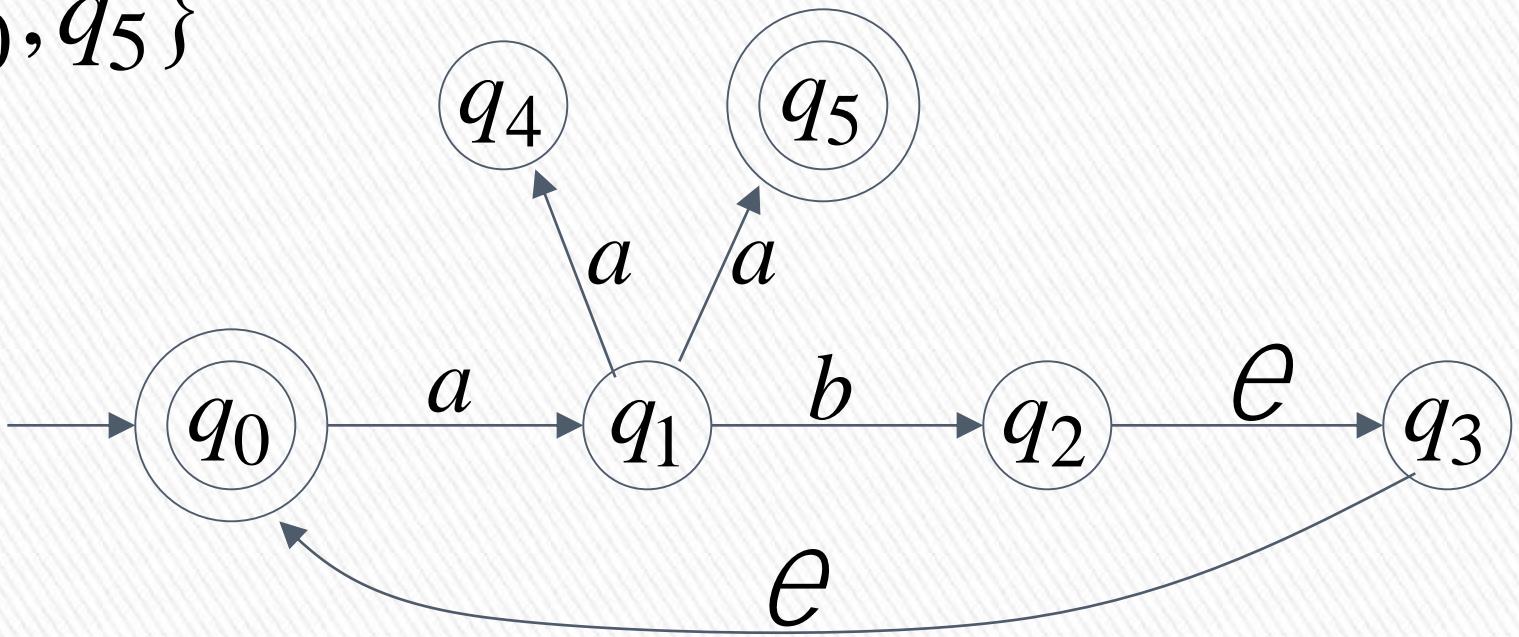
$\delta^*(q_0, w_m)$



$q_k \in F$

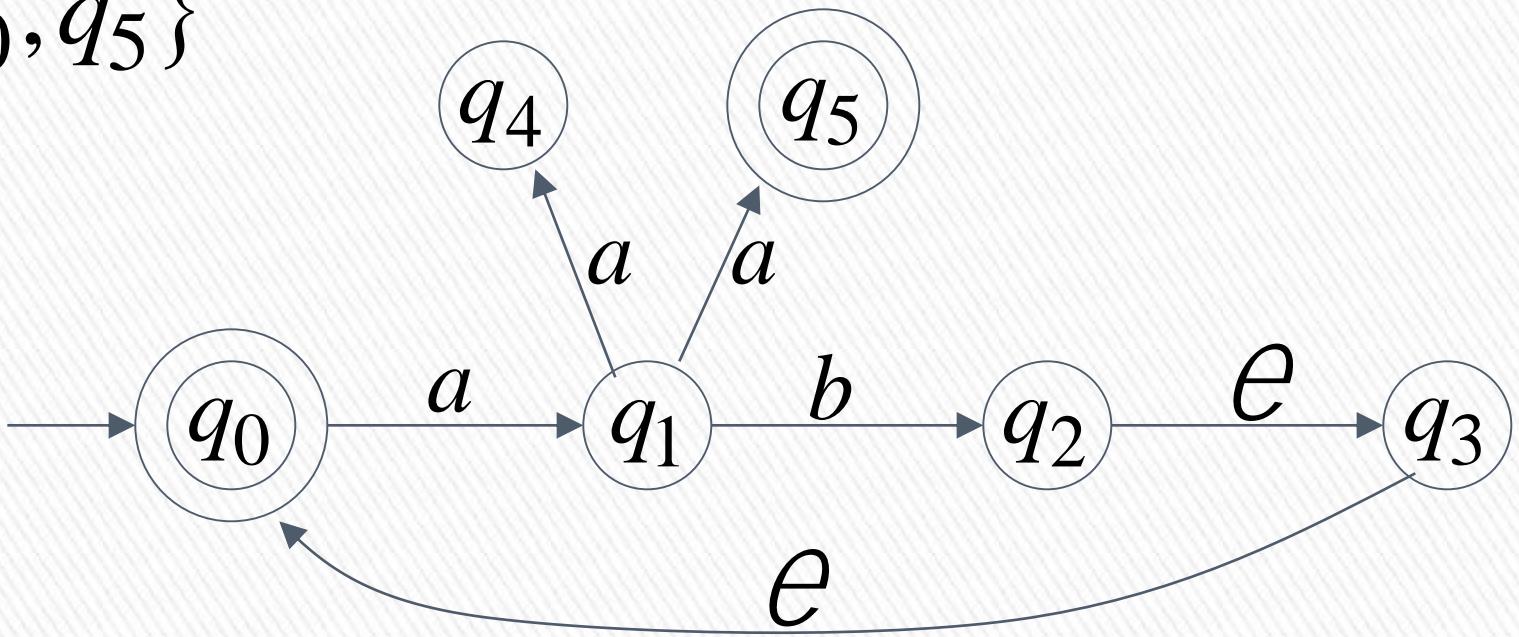


$$F = \{q_0, q_5\}$$



$$\delta^*(q_0, aa) = \{q_4, \underline{q_5}\} \xrightarrow[\vdash F]{} aa \in L(M)$$

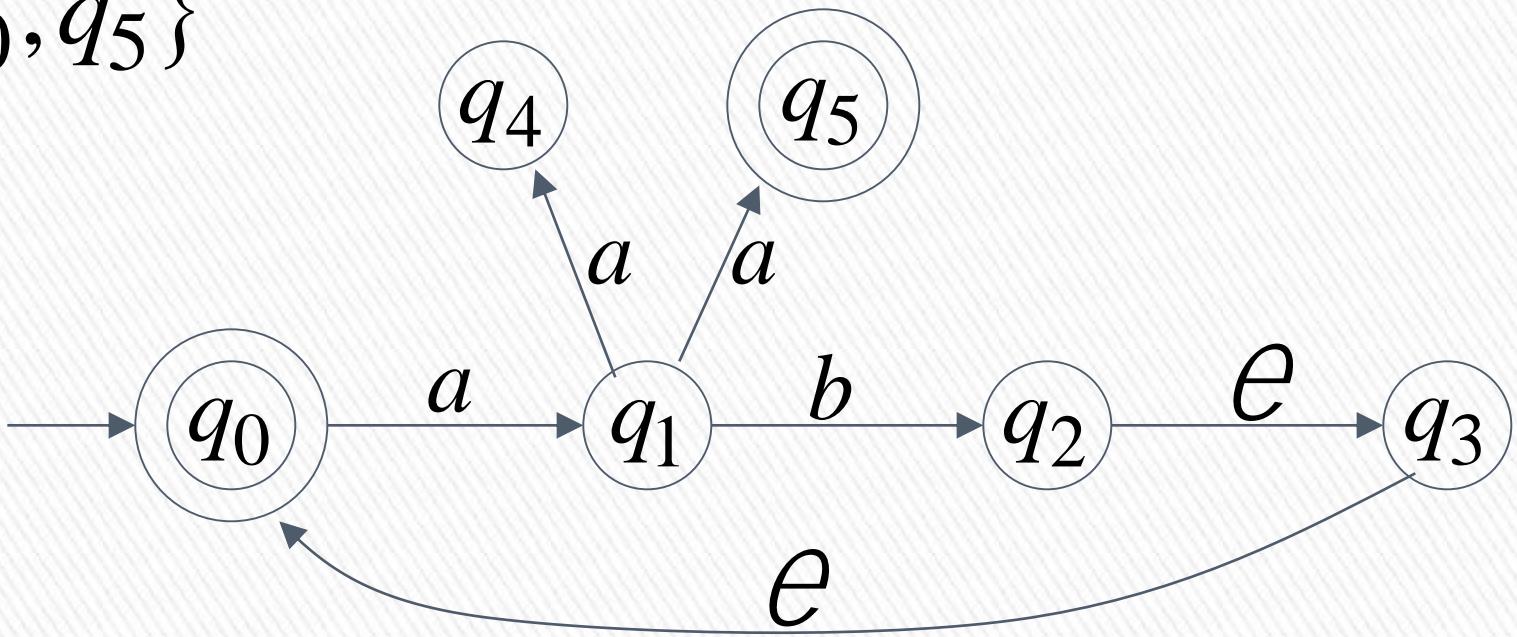
$$F = \{q_0, q_5\}$$



$$\delta^*(q_0, ab) = \{q_2, q_3, \underline{q_0}\} \xrightarrow{\text{yellow arrow}} ab \in L(M)$$

$\nwarrow \mid F$

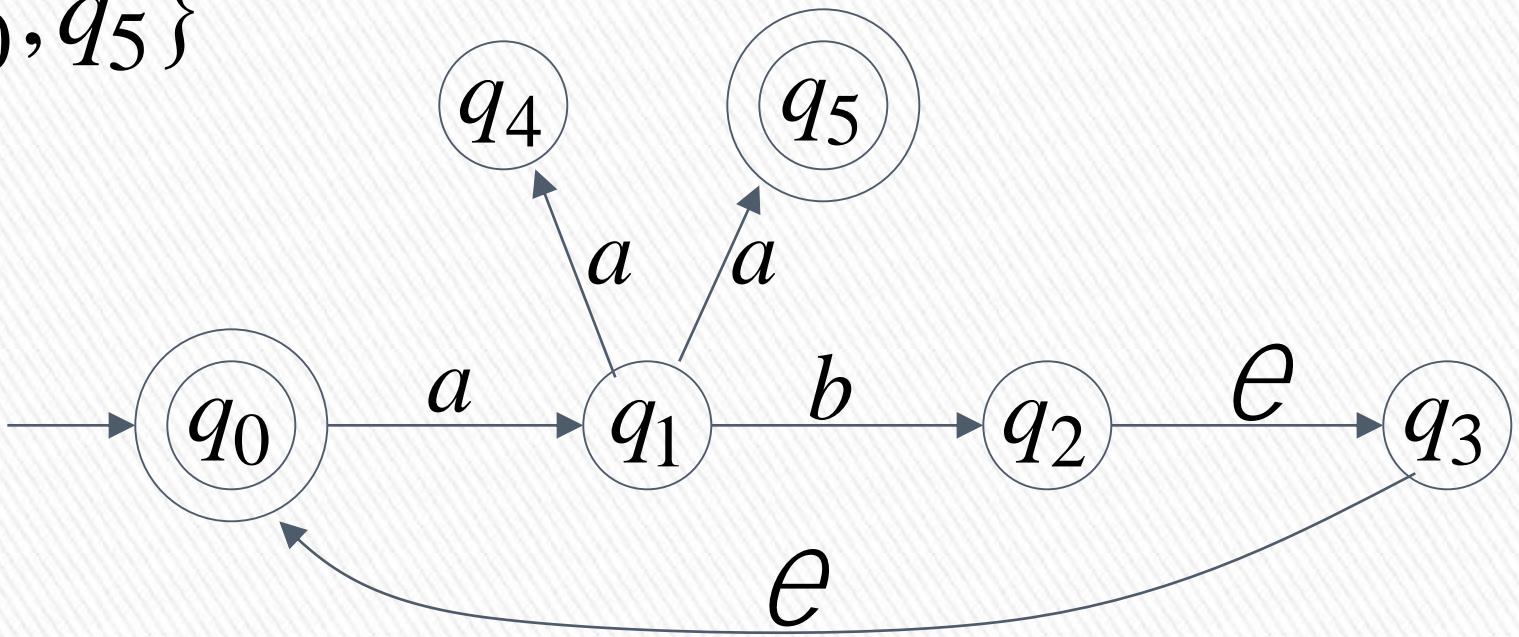
$$F = \{q_0, q_5\}$$



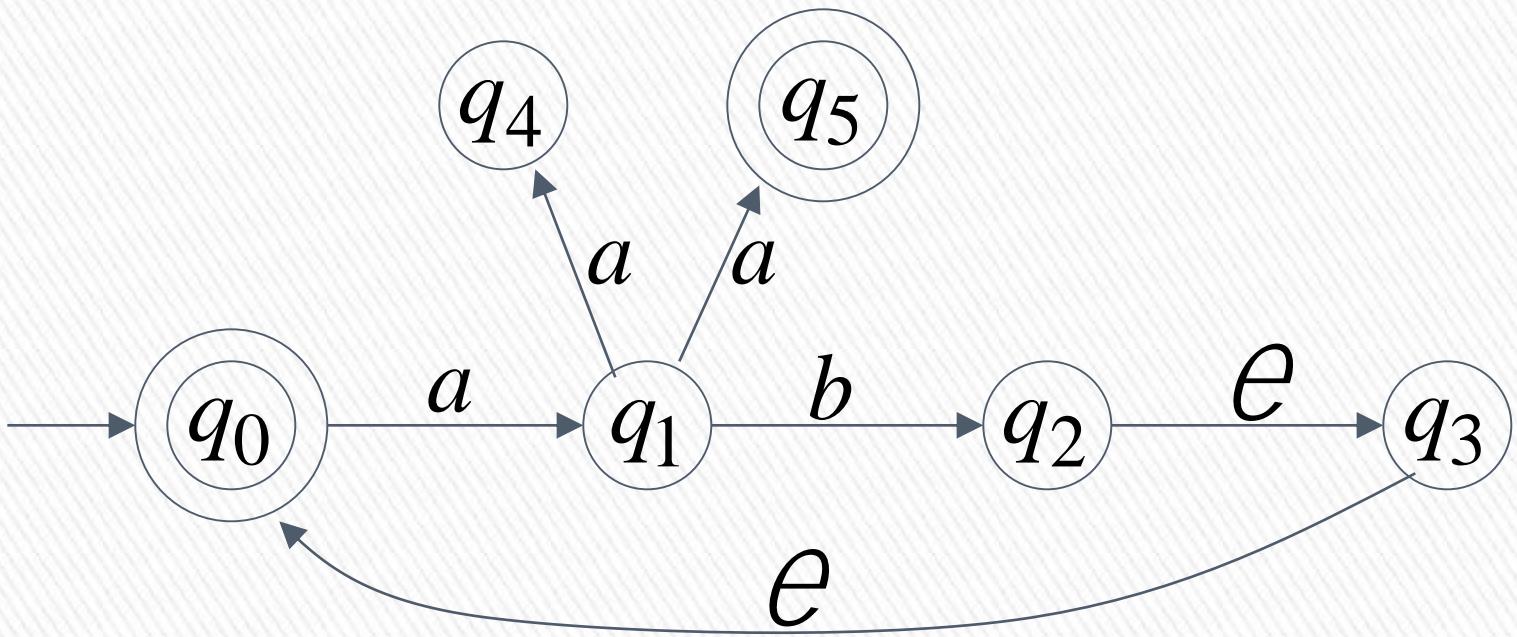
$$\delta^*(q_0, abaa) = \{q_4, \underline{q_5}\} \quad \xrightarrow{\text{abaa}} abaa \in L(M)$$

$\vdash F$

$$F = \{q_0, q_5\}$$



$$\delta^*(q_0, aba) = \{q_1\} \quad \xrightarrow{\text{yellow arrow}} \quad aba \notin L(M) \quad \Rightarrow \quad q_1 \notin F$$



$$L(M) = \{ab\}^* \textcircled{ } \{ab\}^* \{aa\}$$

or



Equivalence of Machines

» Definition:

if machine M_1 and machine M_2
accept ϵ -languages $L(M_1) = L(M_2)$

» Machine M_1 is equivalent to machine M_2

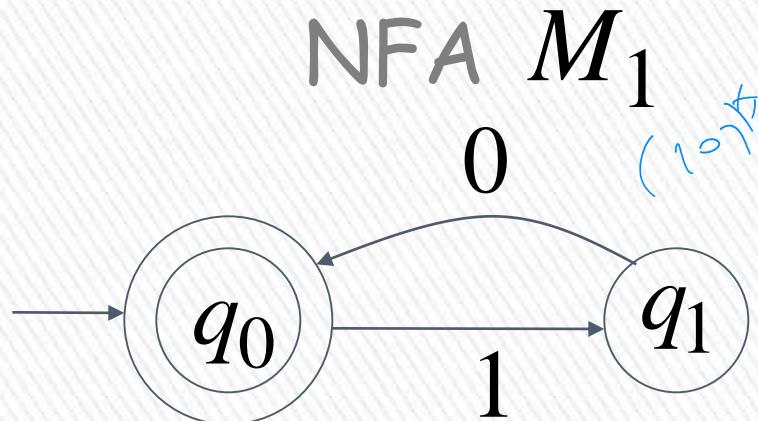
if $L(M_1) = L(M_2)$



Example of equivalent machines

»

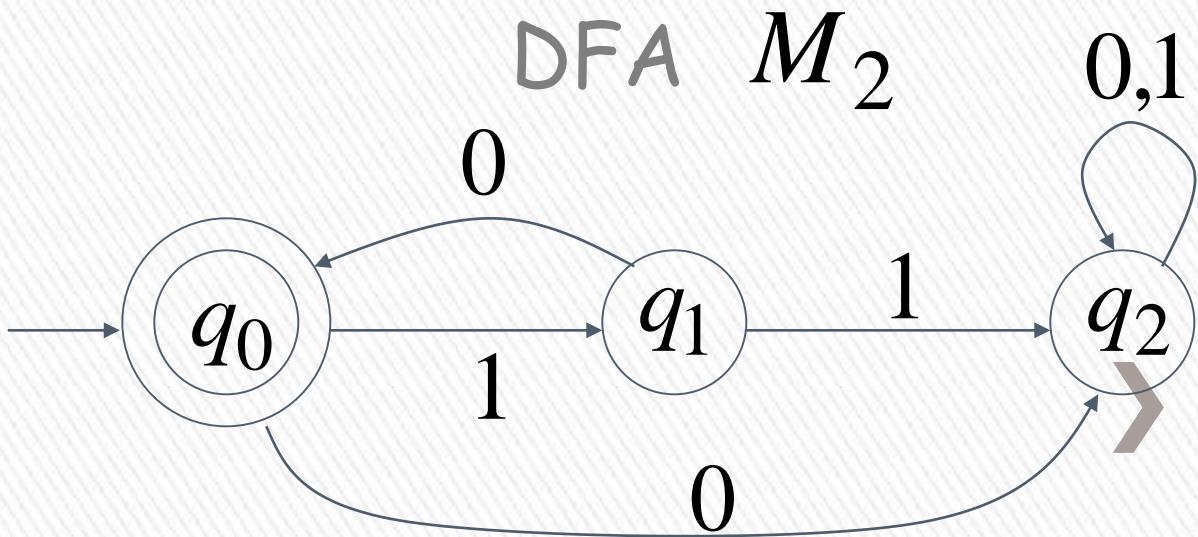
$$L(M_1) = \{10\}^*$$



2. Enkele verklager van equivalent zijn? dan voor

bij de $L(M_1) = \{10\}^*$ verschillende machinies DFA en NFA
beschrijven.

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



NFAs accept Regular Languages

Theorem:

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{accepted} \\ \text{by NFAs} \end{array} \right\} = \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Languages Accepted by DFAs

*↓↓↓ NFA regular languages
↑↑↑ DFA regular languages*

NFAs and DFAs have the same computation power,
accept the same set of languages

Proof: we only need to show

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{accepted} \\ \text{by NFAs} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

NFA - DFA's
forall L

DFA \subseteq NFA's

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{accepted} \\ \text{by NFAs} \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

AND



Proof-Step 1

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{accepted} \\ \text{by NFAs} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Every DFA is trivially an NFA



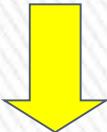
Any language L accepted by a DFA
is also accepted by an NFA



Proof-Step 2

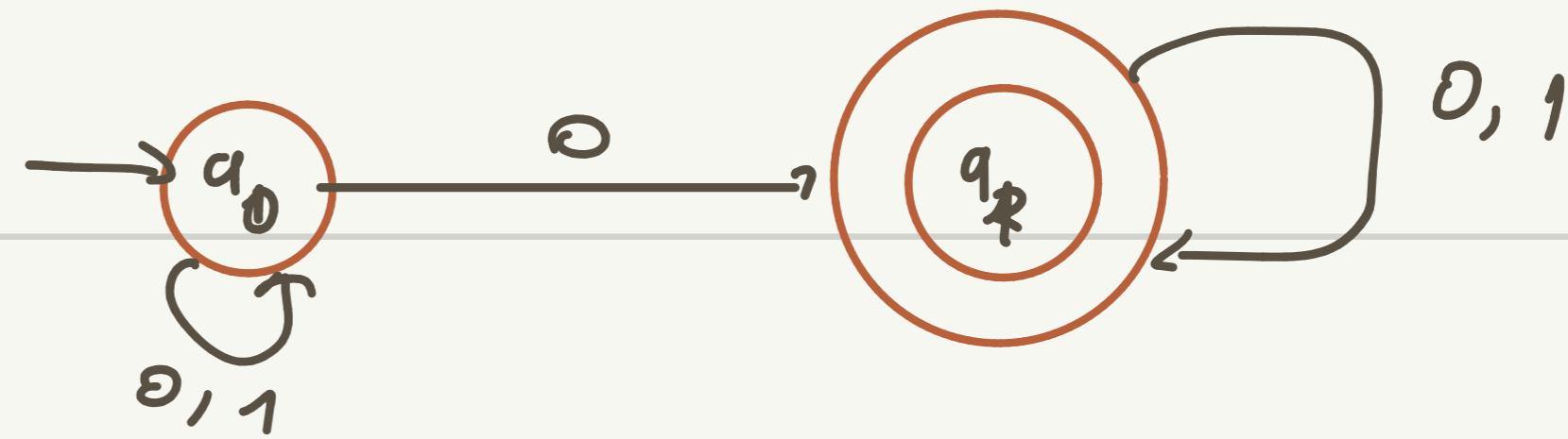
$$\left\{ \text{Languages accepted by NFAs} \right\} \subseteq \left\{ \text{Regular Languages} \right\}$$

Any NFA can be converted to an equivalent DFA



Any language L accepted by an NFA
is also accepted by a DFA

$L_1 = \{ \text{set of all strings that contain } 0 \}$



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

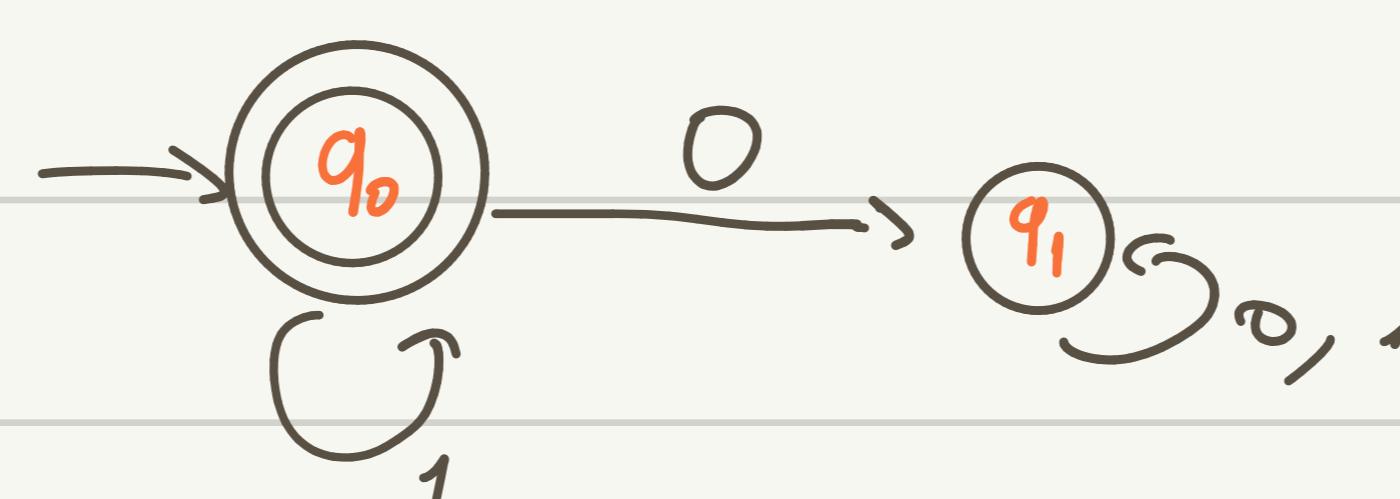
$$Q = \{q_0, q_1\}$$

q_0 initial state

$$F = \{q_1\}$$

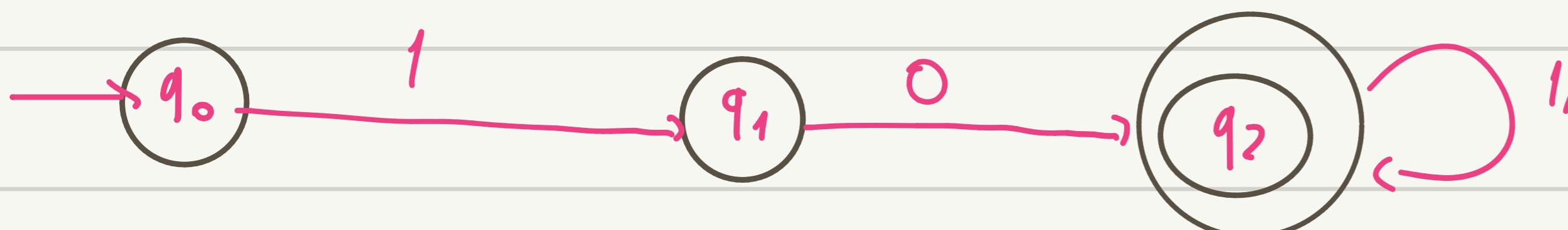
	0	1	ϵ
q_0	q_0, q_1	q_0	\emptyset
q_1	q_1	q_1	\emptyset

$L_2 = \{ \text{string not containing } 0 \}$

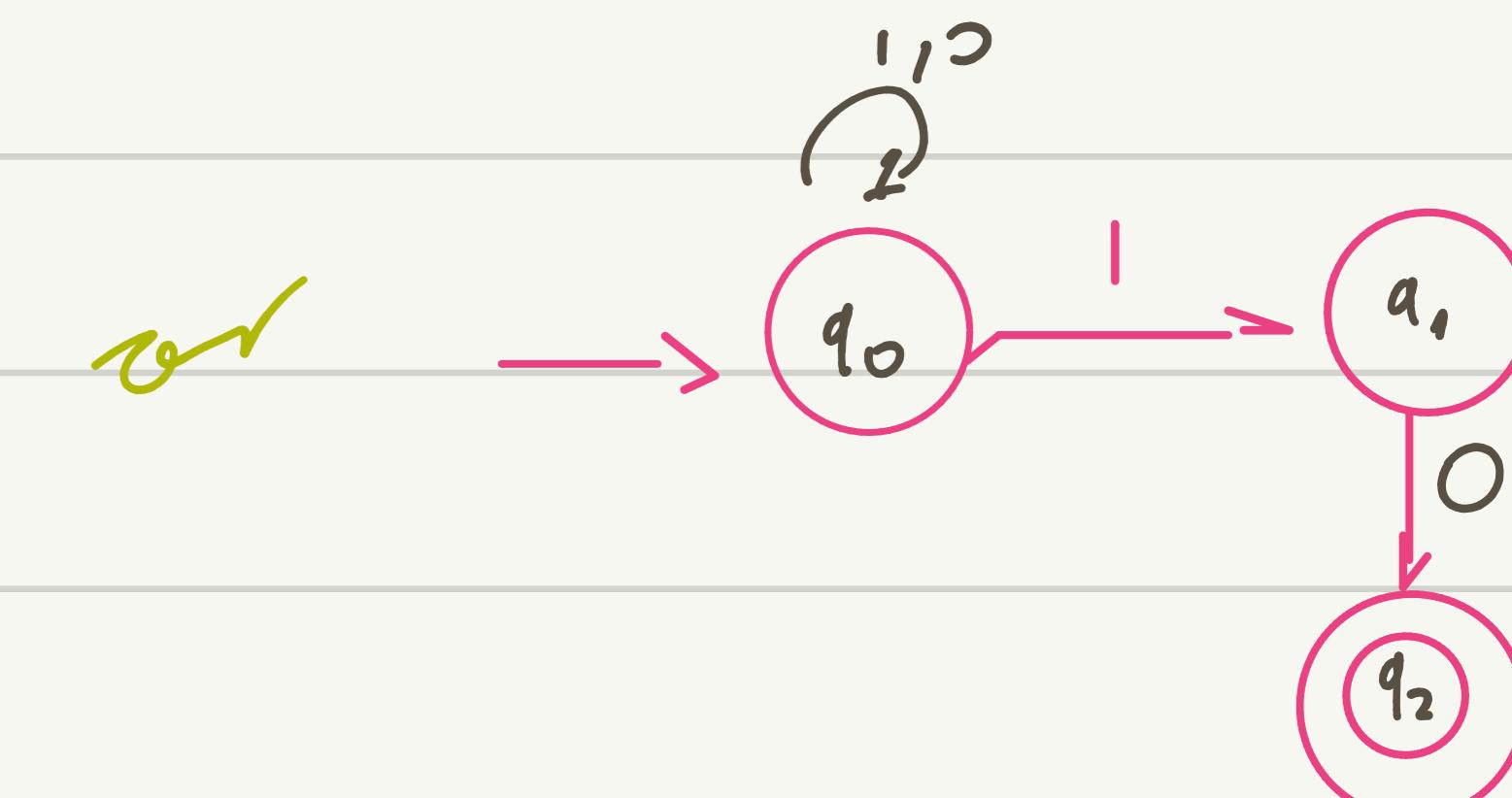
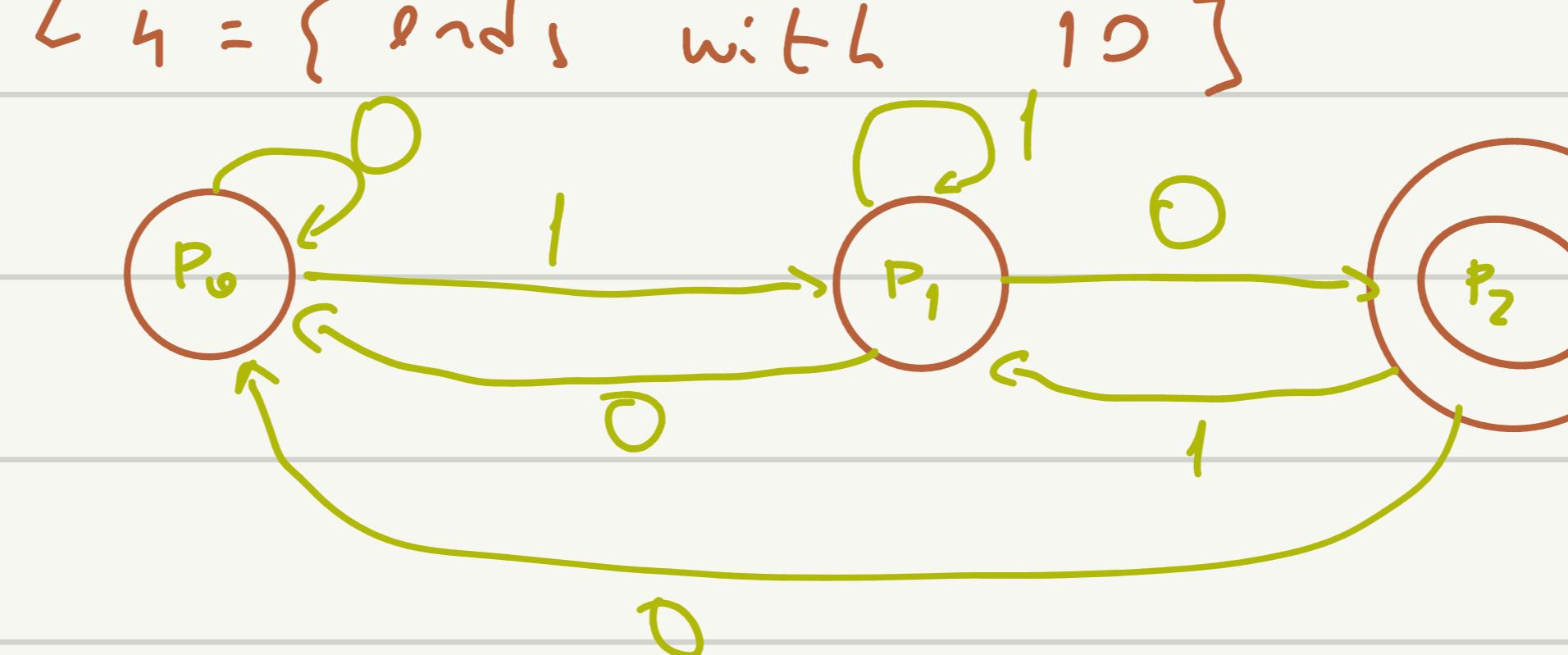


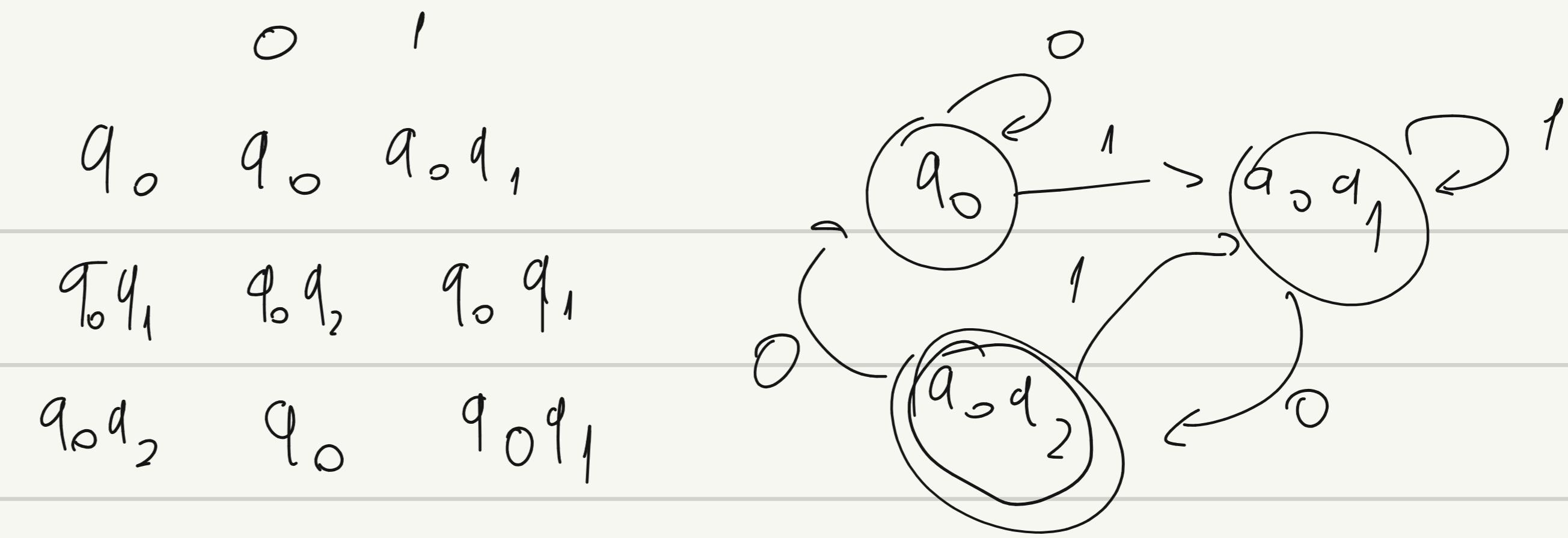
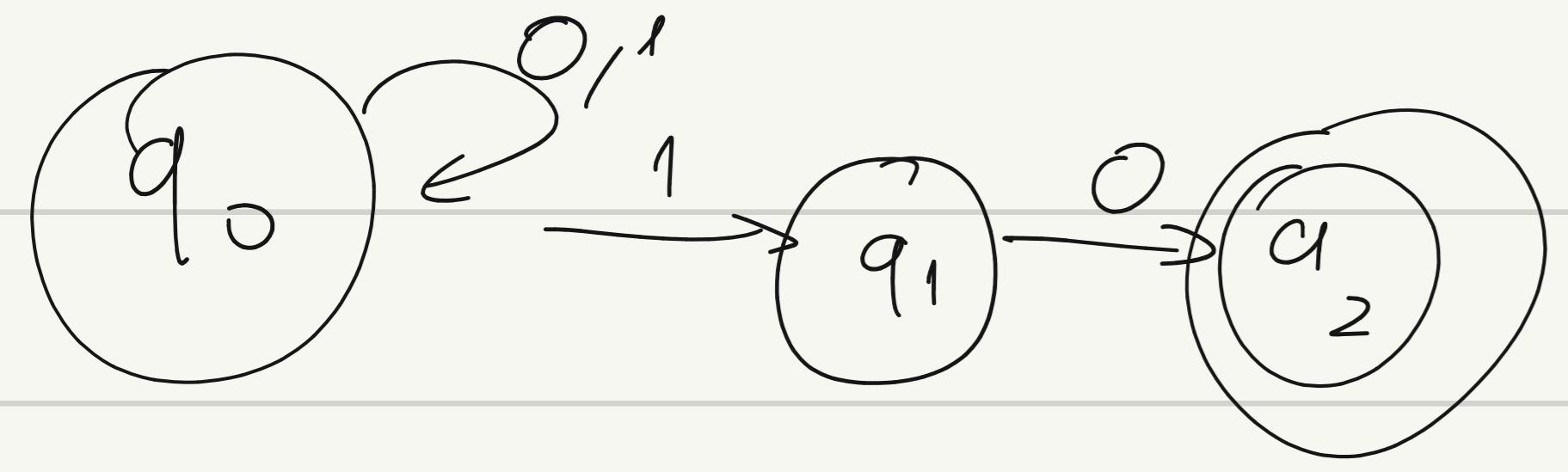
	0	1	ϵ
q_0	q_1	q_0	q_0
q_1	q_1	q_1	\emptyset

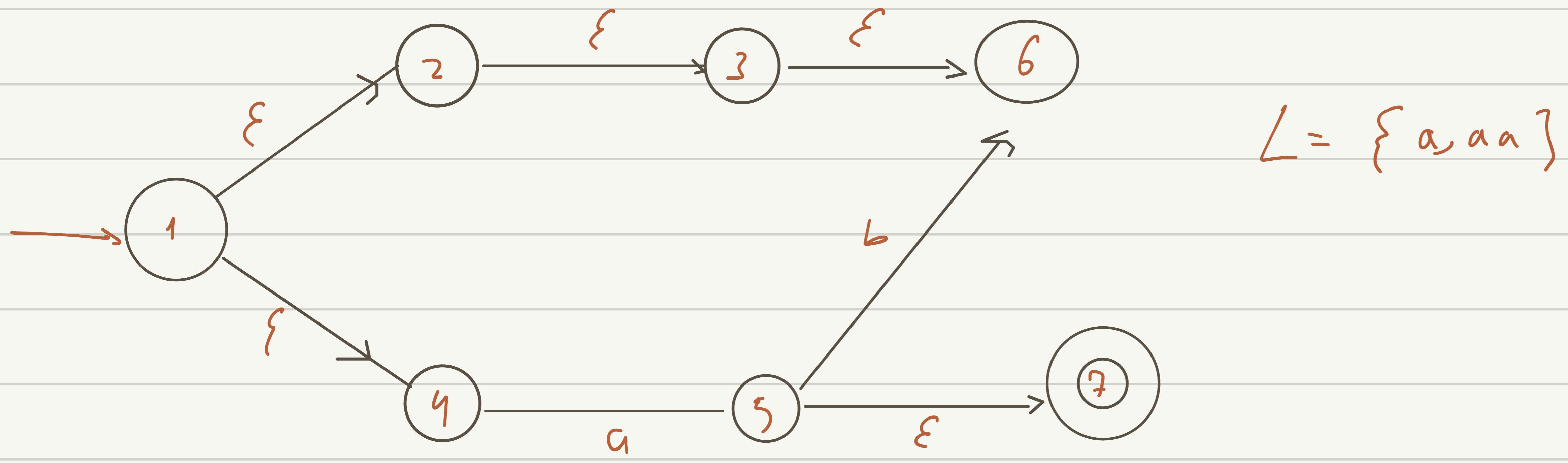
$L_3 = \{ \text{set of strings starts with } 10 \}$



$L_4 = \{ \text{ends with } 10 \}$







$$L = \{a, aa\}$$

Lemma:

If we convert NFA M to DFA M'
then the two automata are equivalent:

$$L(M) = L(M')$$

Proof:

We only need to show: $L(M) \subseteq L(M')$

AND

$$L(M) \supseteq L(M') \quad >$$

First we show: $L(M) \subseteq L(M')$

We only need to prove:

$$w \in L(M)$$



$$w \in L(M')$$



NFA

Consider $w \in L(M)$



symbols

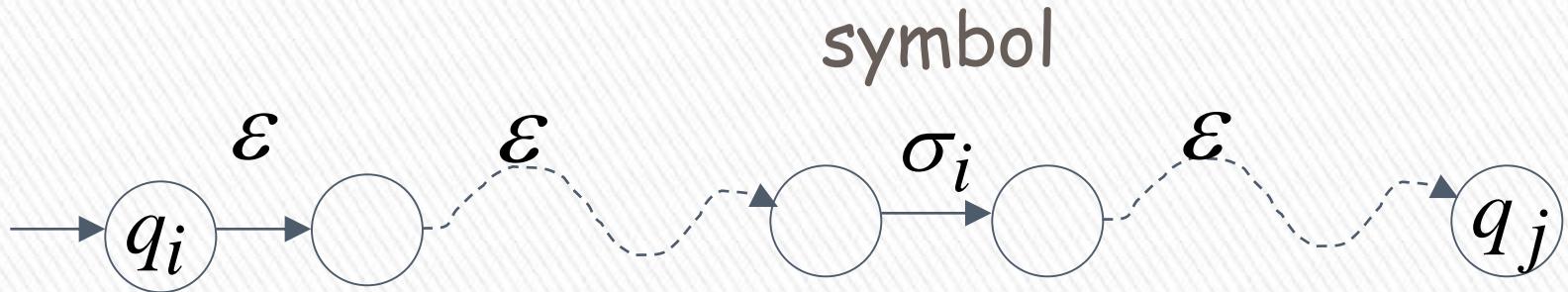
$$w = \sigma_1 \sigma_2 \cdots \sigma_k$$



symbol

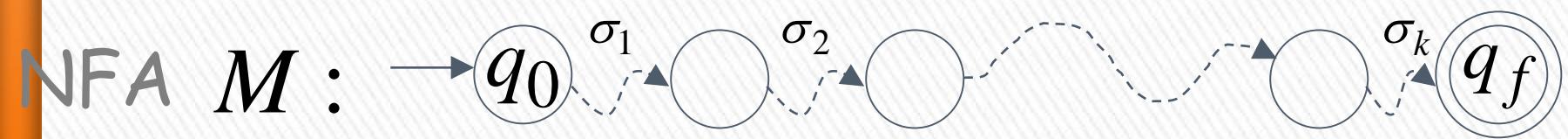


denotes a possible sub-path like

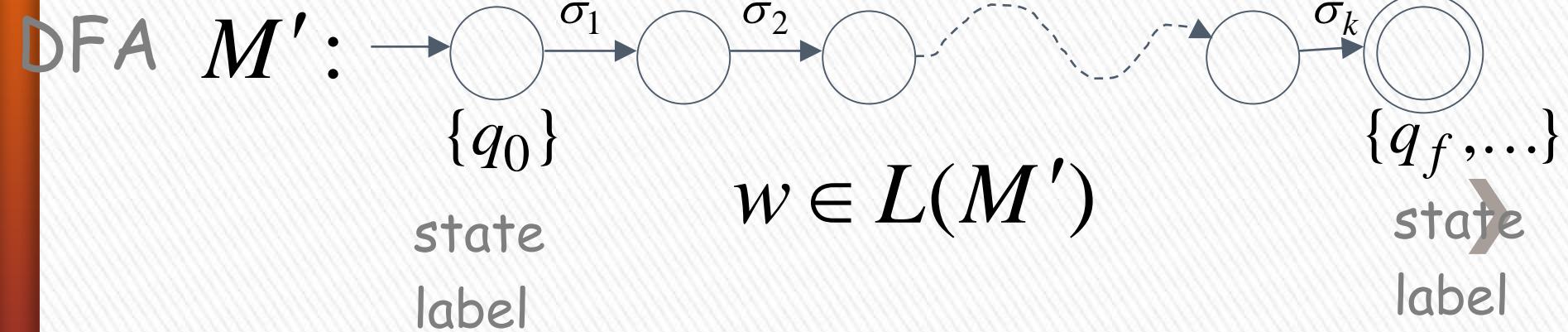


We will show that if $w \in L(M)$

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



then

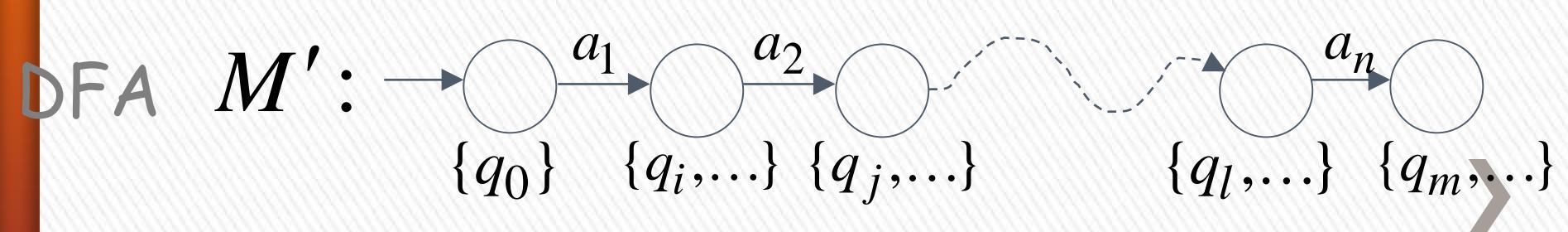


More generally, we will show that if in M :

(arbitrary string) $\mathcal{V} = a_1 a_2 \cdots a_n$

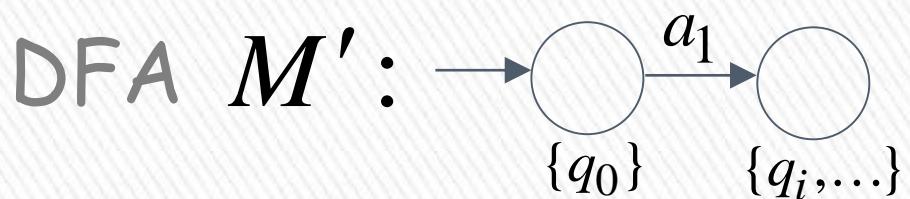


then



Proof by induction on $|v|$

Induction Basis: $|v| = 1 \quad v = a_1$



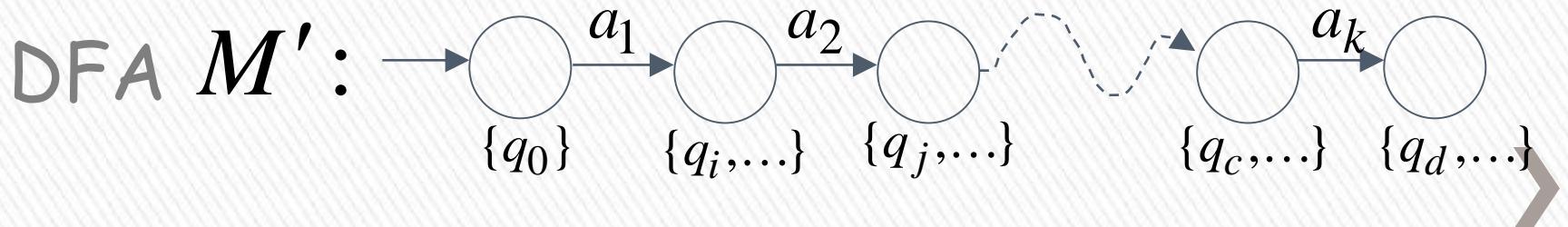
is true by construction of M'



Induction hypothesis: $1 \leq |v| \leq k$

$$v = a_1 a_2 \cdots a_k$$

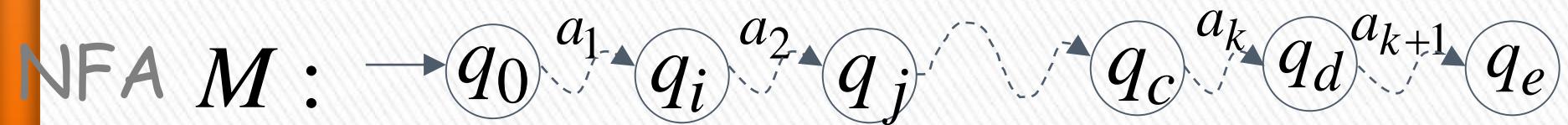
Suppose that the following hold



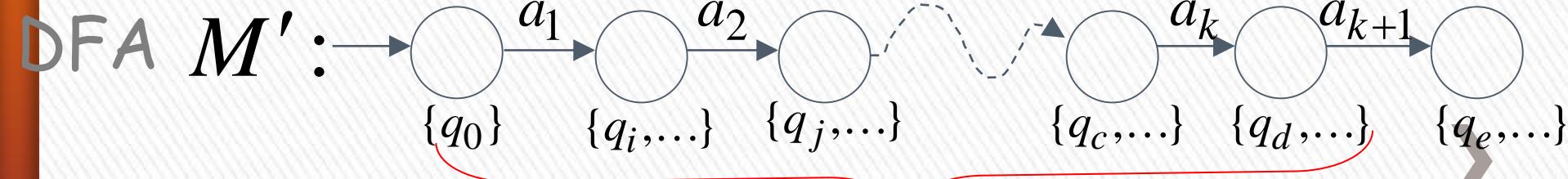
Induction Step: $|v| = k + 1$

$$v = \underbrace{a_1 a_2 \cdots a_k}_{v'} a_{k+1} = v' a_{k+1}$$

Then this is true by construction of M'



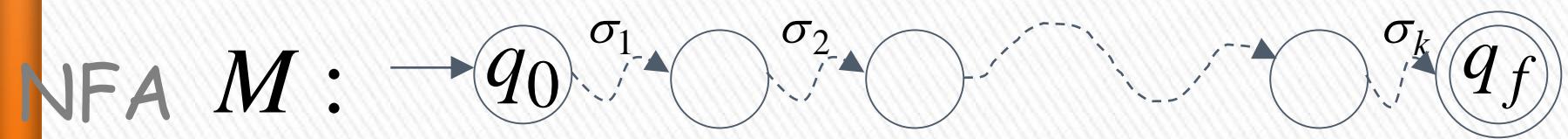
v'



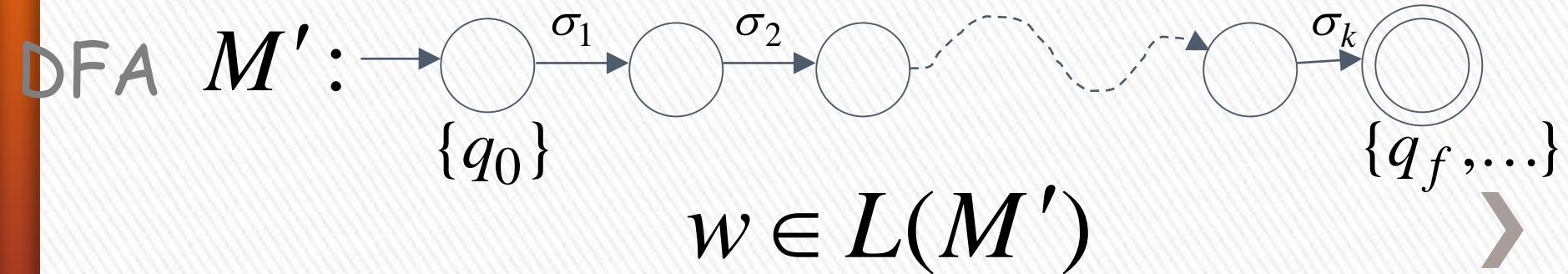
v'

Therefore if $w \in L(M)$

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



then



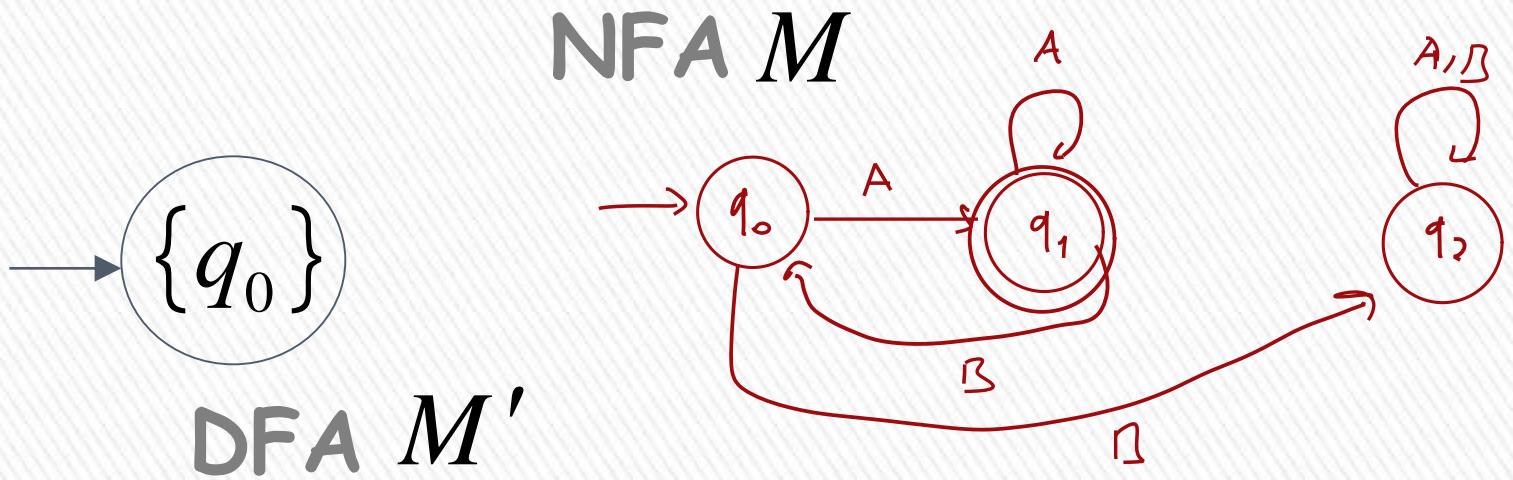
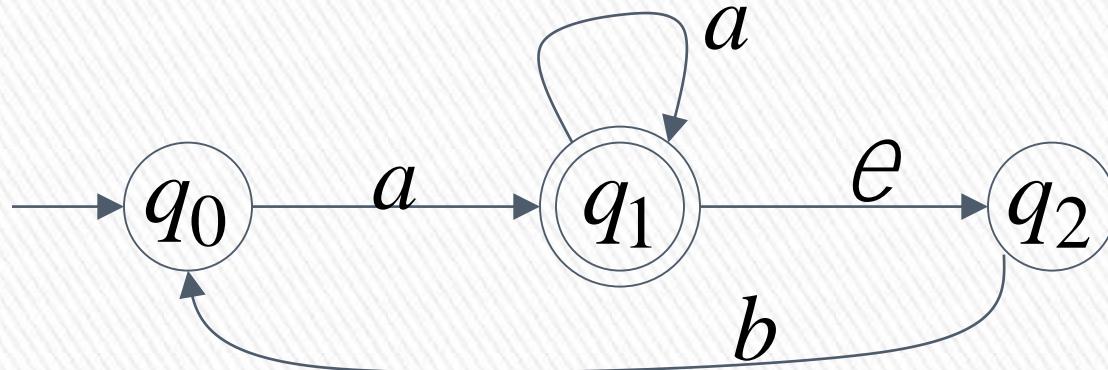
We have shown: $L(M) \subseteq L(M')$

With a similar proof
we can show: $L(M) \supseteq L(M')$

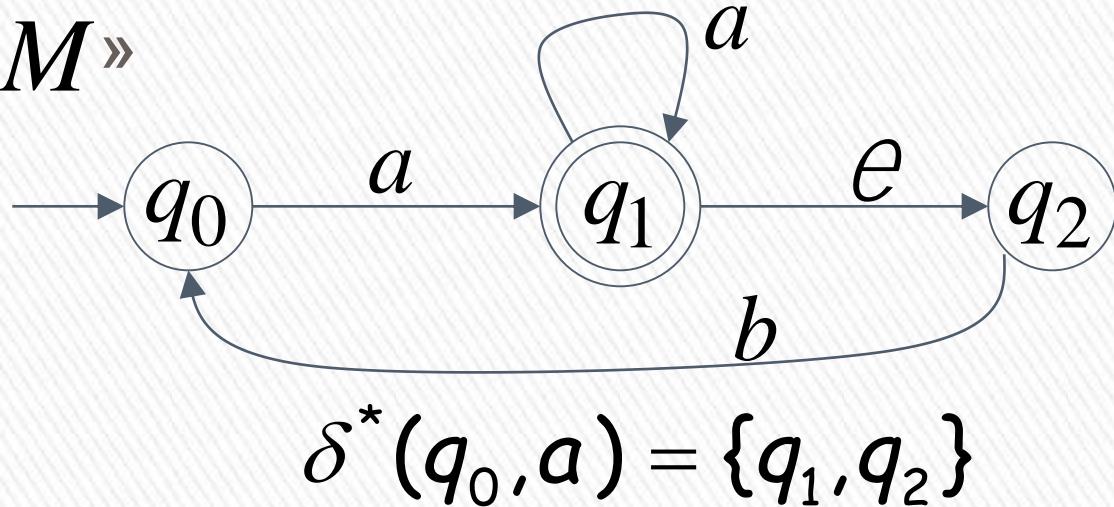
Therefore: $L(M) = L(M')$

END OF LEMMA PROOF

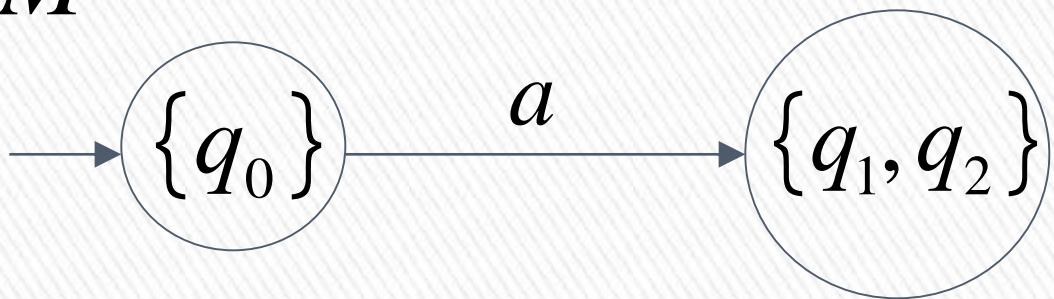
Conversion NFA to DFA



NFA $M \gg$

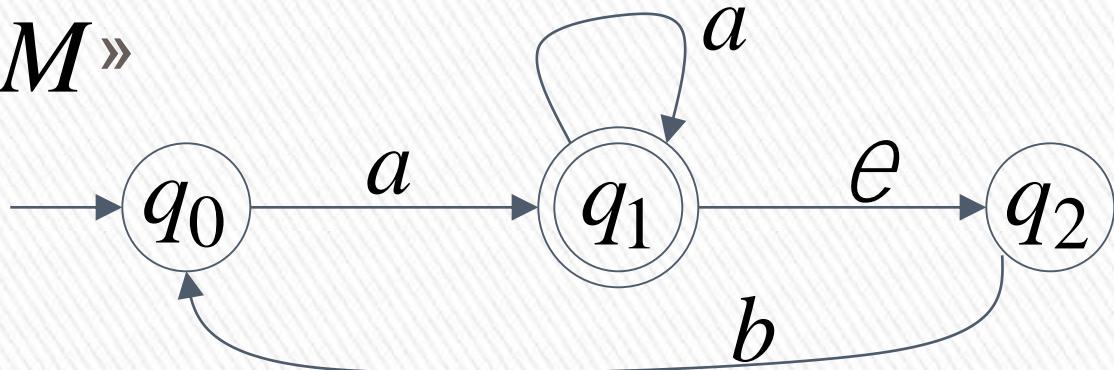


DFA M'

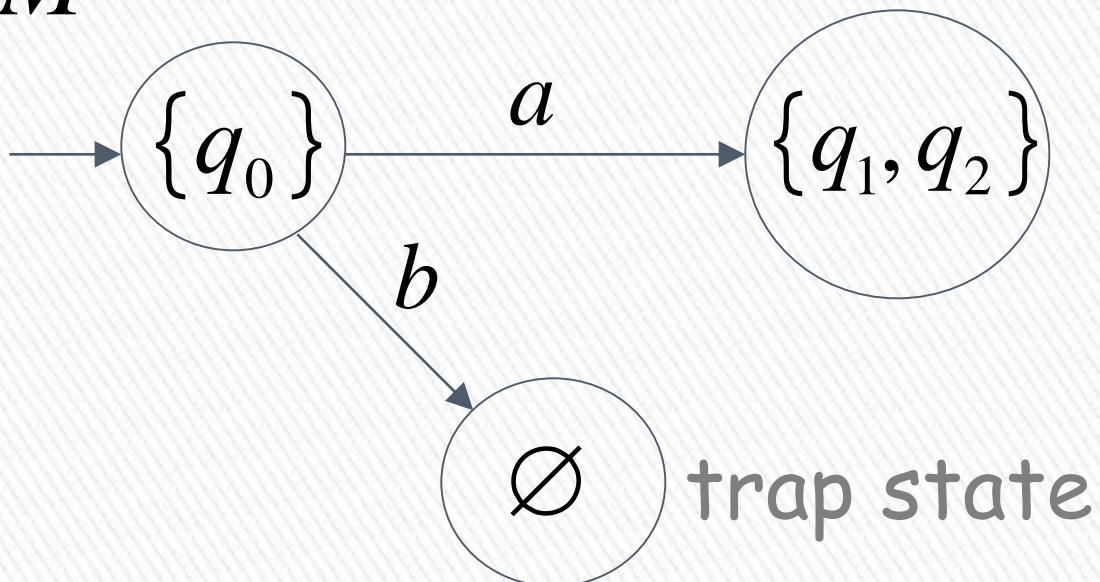


$$\delta^*(q_0, b) = \emptyset \quad \text{empty set}$$

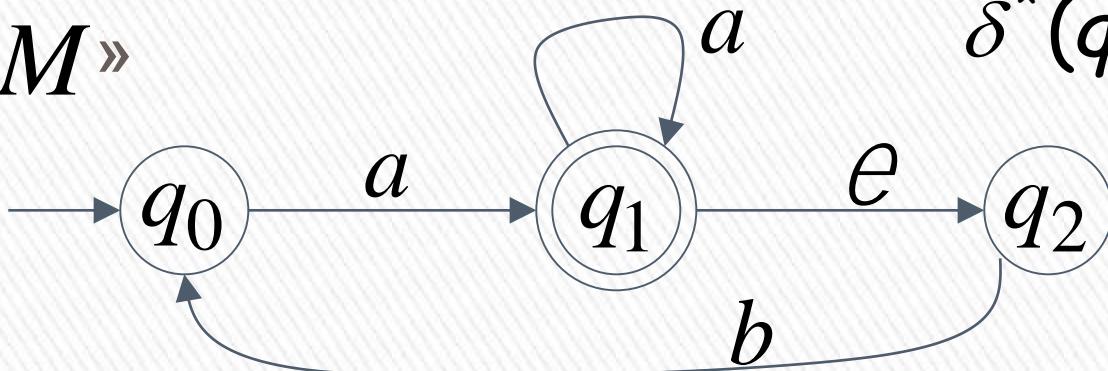
NFA $M»$



DFA M'



NFA $M \gg$

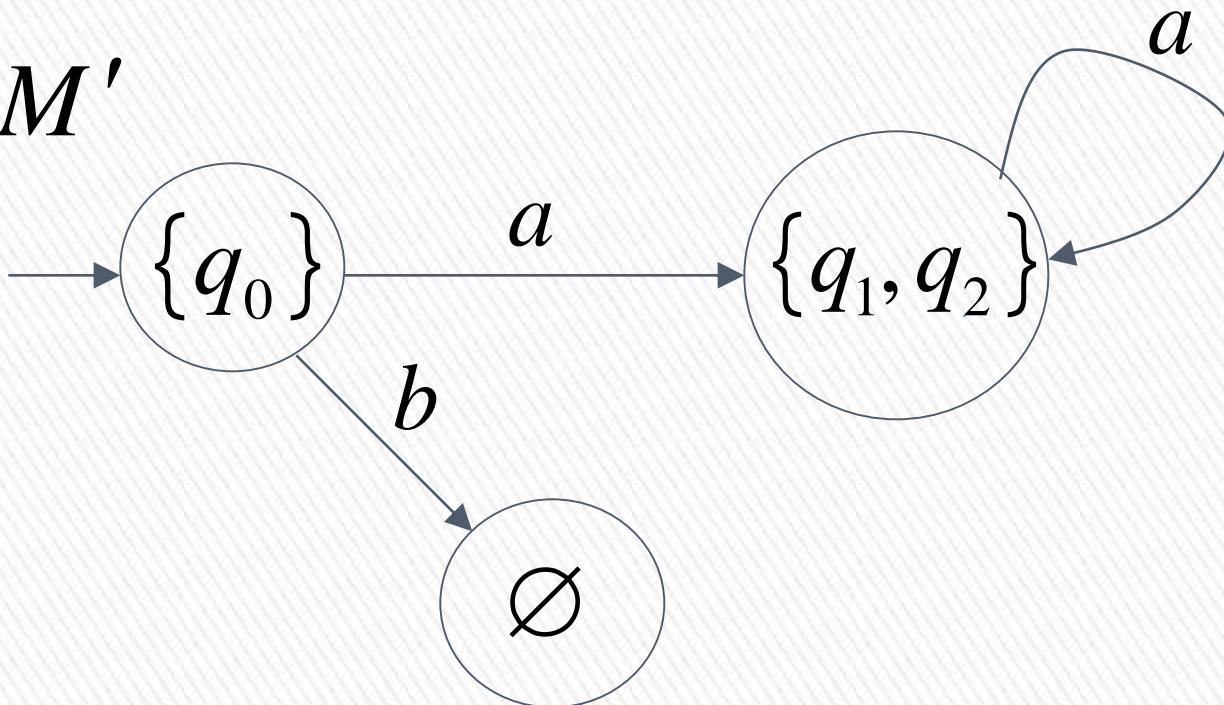


$$\delta^*(q_1, a) = \{q_1, q_2\}$$
$$\delta^*(q_2, a) = \emptyset$$

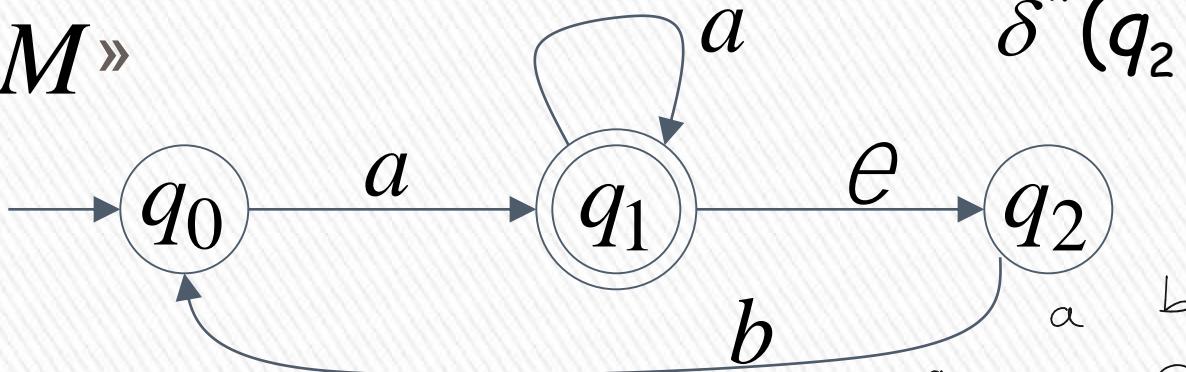
union

$\{q_1, q_2\}$

DFA M'



NFA $M \gg$

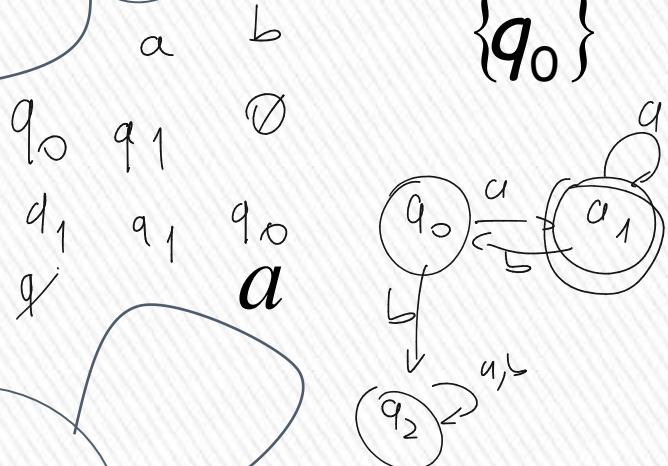
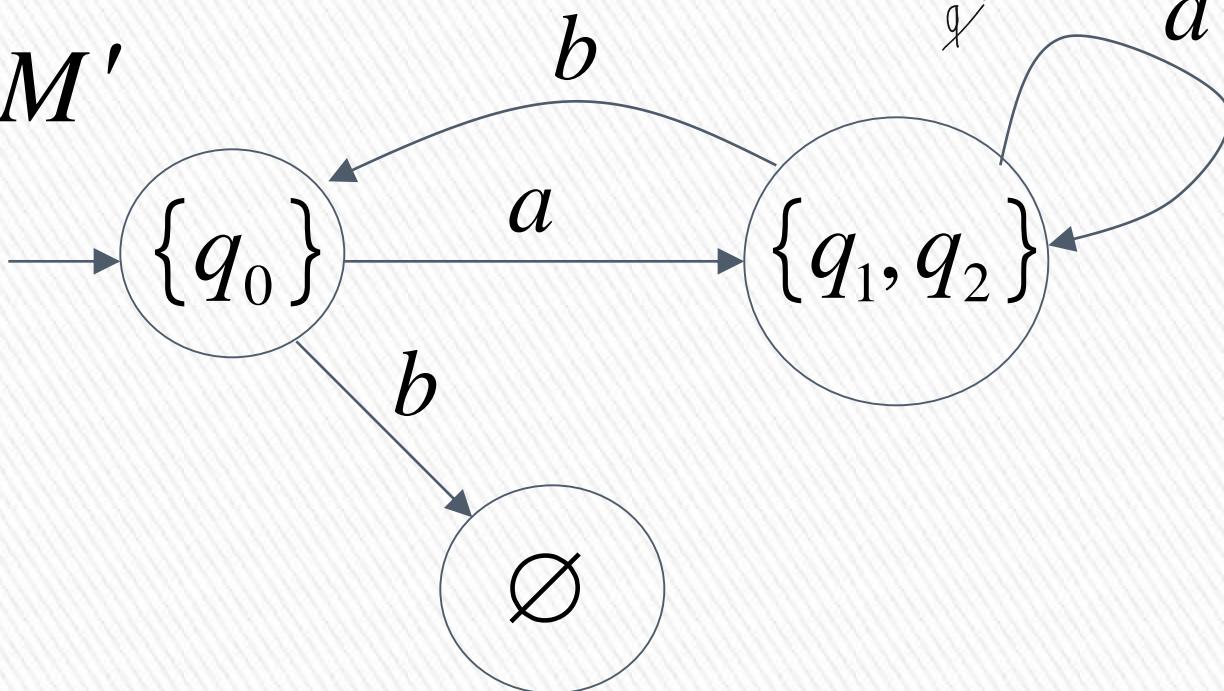


$$\delta^*(q_1, b) = \{q_0\}$$
$$\delta^*(q_2, b) = \{q_0\}$$

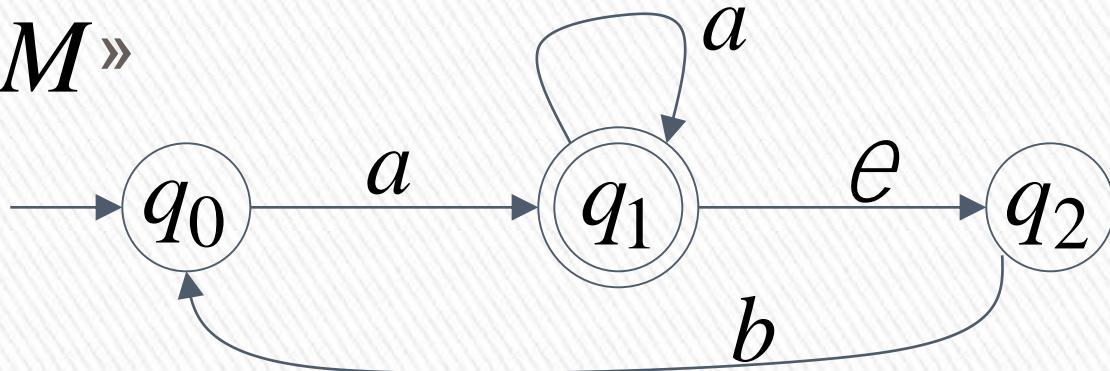
union

$\{q_0\}$

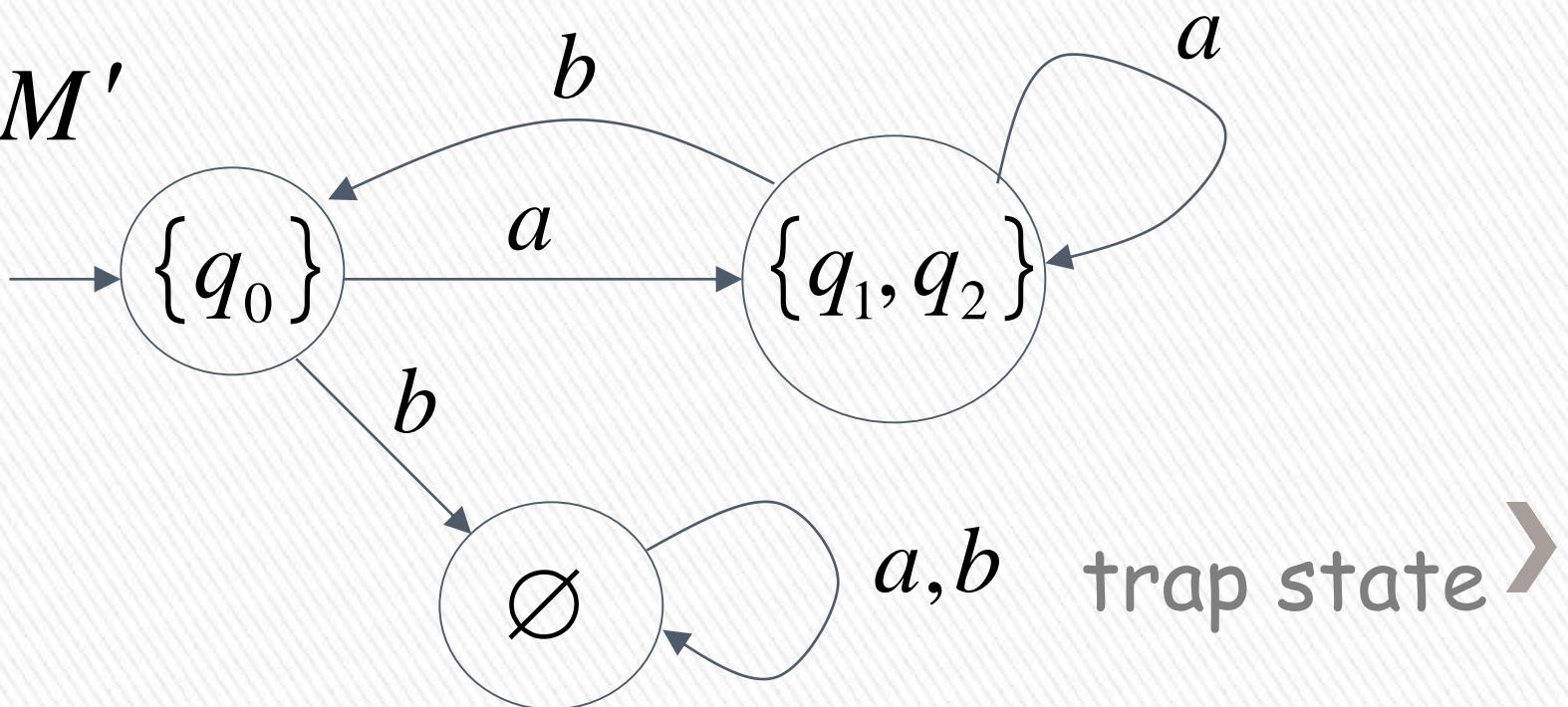
DFA M'



NFA $M \gg$

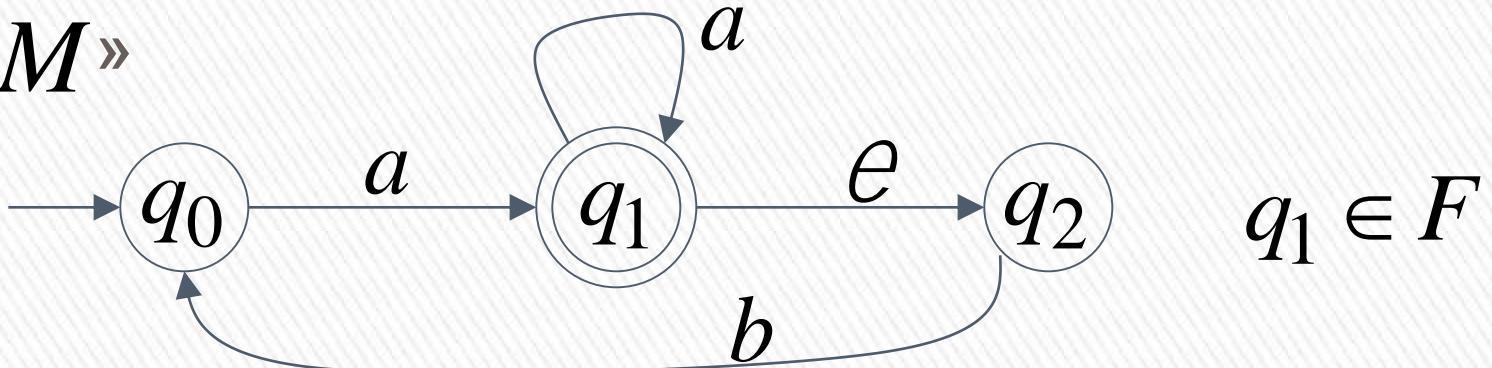


DFA M'

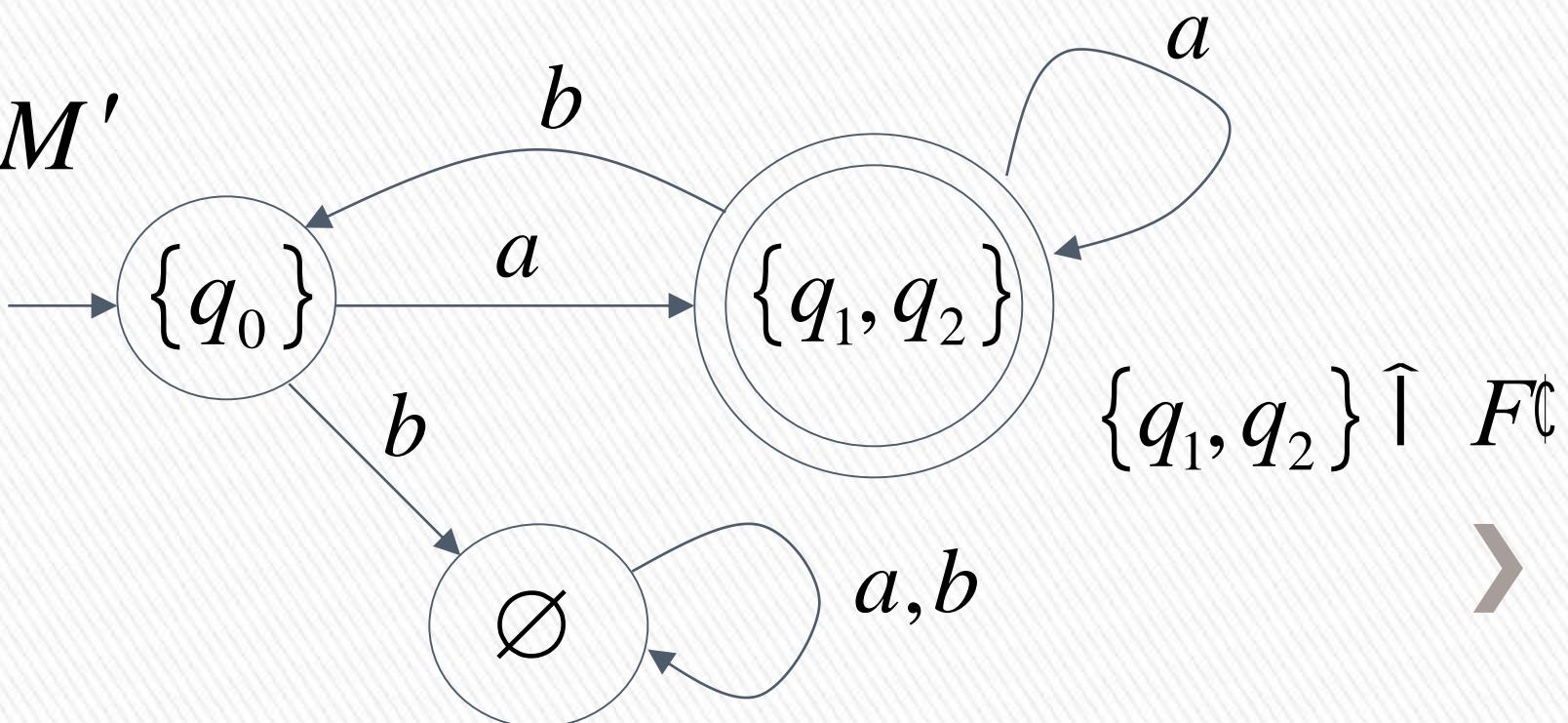


END OF CONSTRUCTION

NFA $M \gg$



DFA M'



$\{q_1, q_2\} \upharpoonright F \complement$



General Conversion Procedure

- » **Input:** an NFA M
- » **Output:** an equivalent DFA M'
with $L(M) = L(M')$



» The NFA has states

q_0, q_1, q_2, \dots

» The DFA has states from the power set

$\emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}, \{q_1, q_2, q_3\}, \dots$



Conversion Procedure Steps

»

Step 1

» Initial state of NFA: q_0

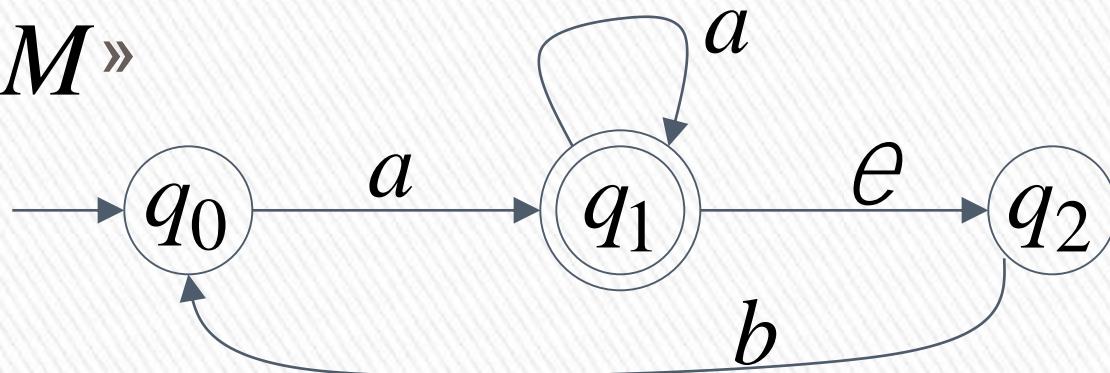


» Initial state of DFA: $\{q_0\}$

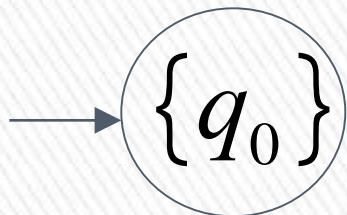


Example

NFA $M \gg$



DFA M'



Step 2

For every DFA's state $\{q_i, q_j, \dots, q_m\}$

compute in the NFA

$$\delta^*(q_i, a) \cup \delta^*(q_j, a)$$

...

$$\cup \delta^*(q_m, a)$$

add transition to DFA

Union

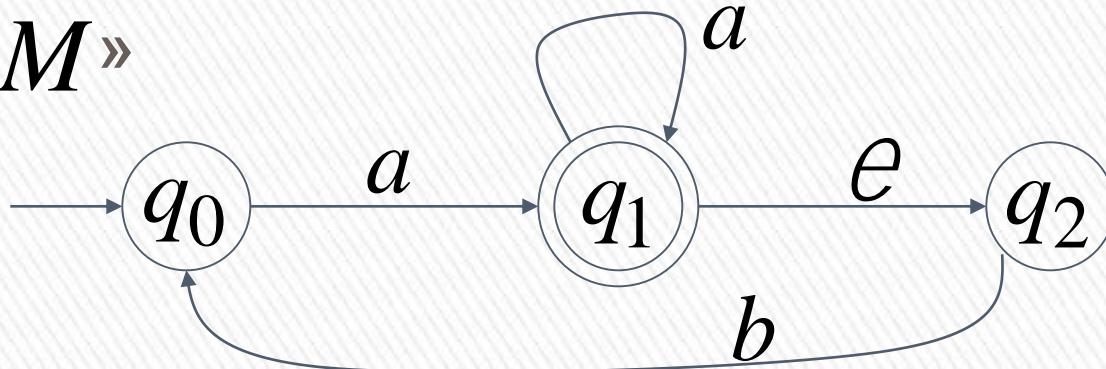
$$\{q'_k, q'_l, \dots, q'_n\}$$

$$\delta(\{q_i, q_j, \dots, q_m\}, a) = \{q'_k, q'_l, \dots, q'_n\}$$

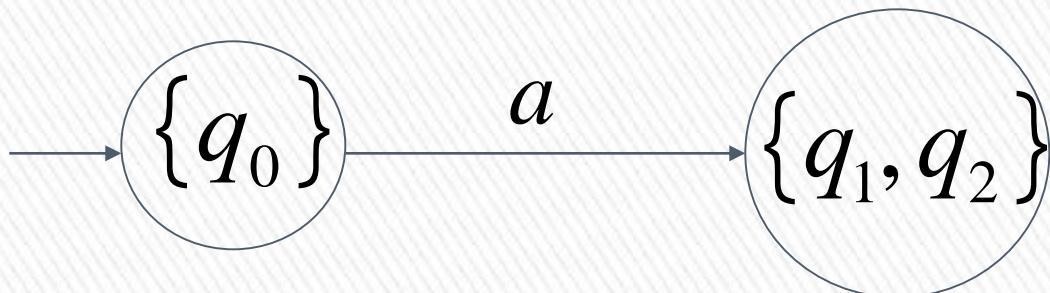


Example $\delta^*(q_0, a) = \{q_1, q_2\}$

NFA M »



DFA M' $d(\{q_0\}, a) = \{q_1, q_2\}$



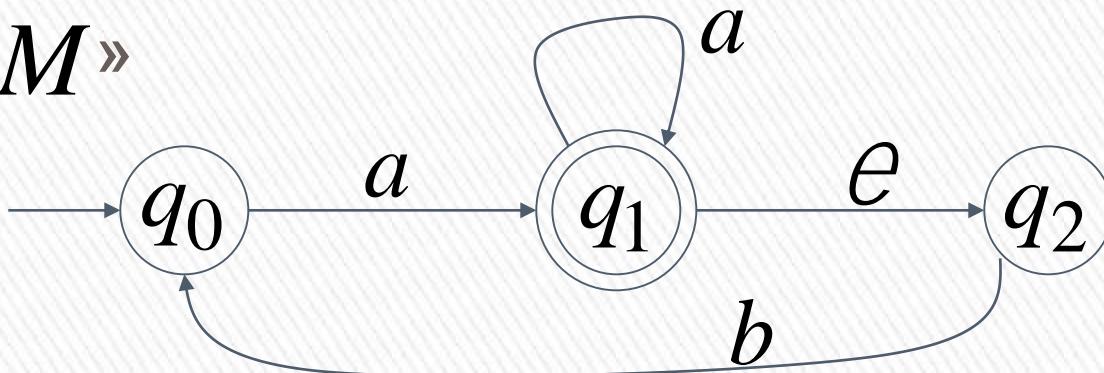
Step 3

- » Repeat Step 2 for every state in DFA and symbols in alphabet until no more states can be added in the DFA

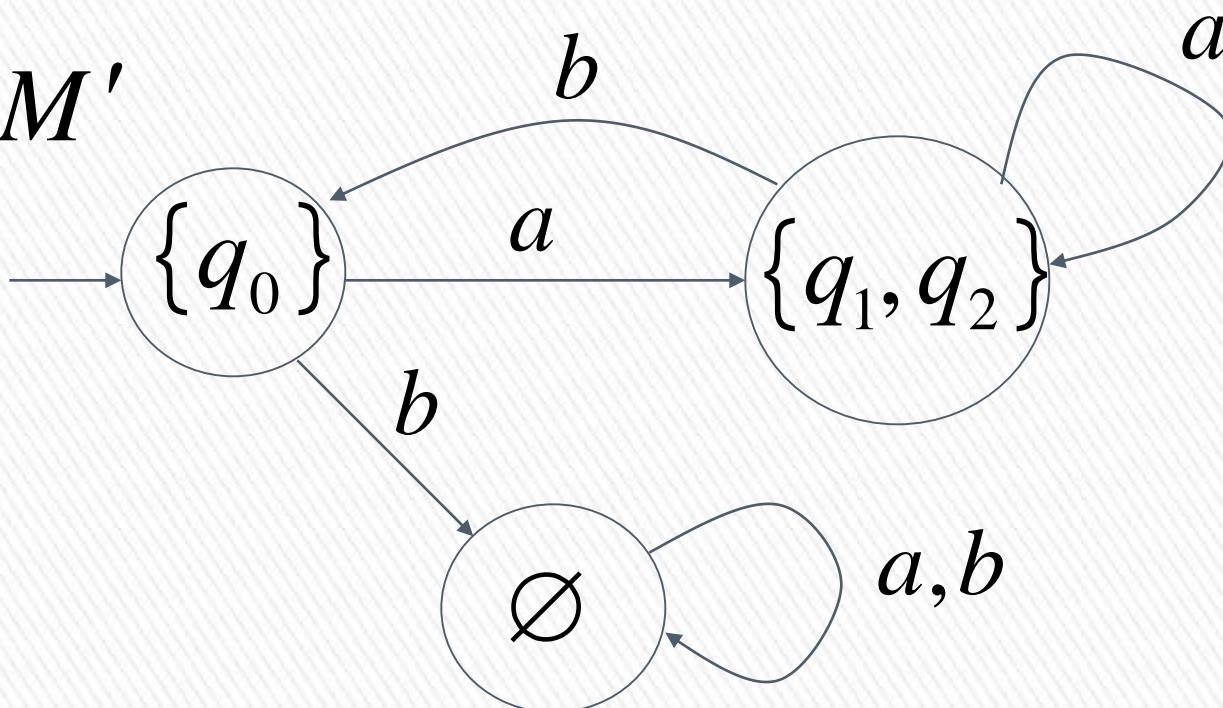


Example

NFA $M \gg$



DFA M'



Step 4

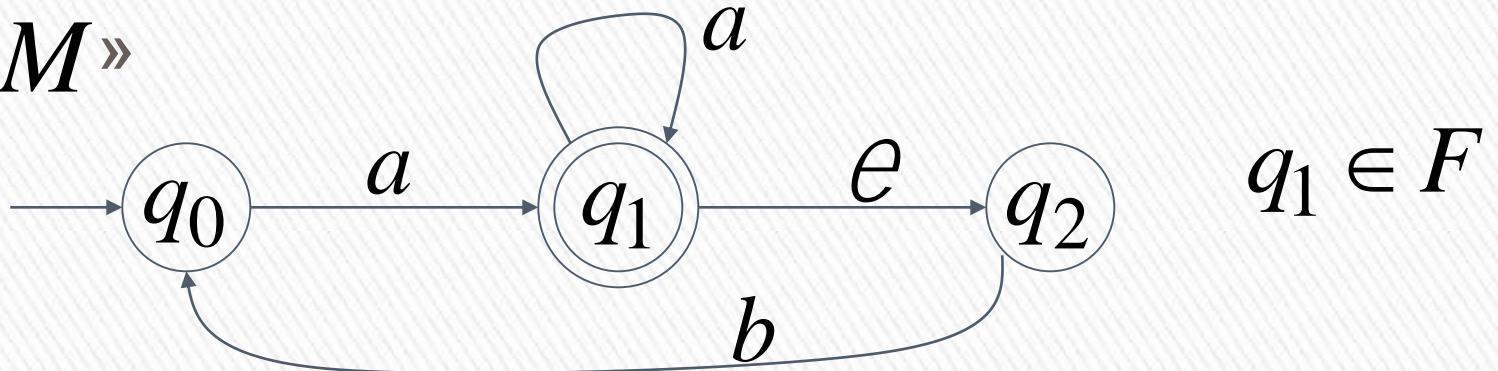
- » For any DFA state $\{q_i, q_j, \dots, q_m\}$
- » if some q_j is accepting state in NFA
- » Then, $\{q_i, q_j, \dots, q_m\}$ is accepting state in DFA

*↳: minden accept state olunur
hepsi final state yur**



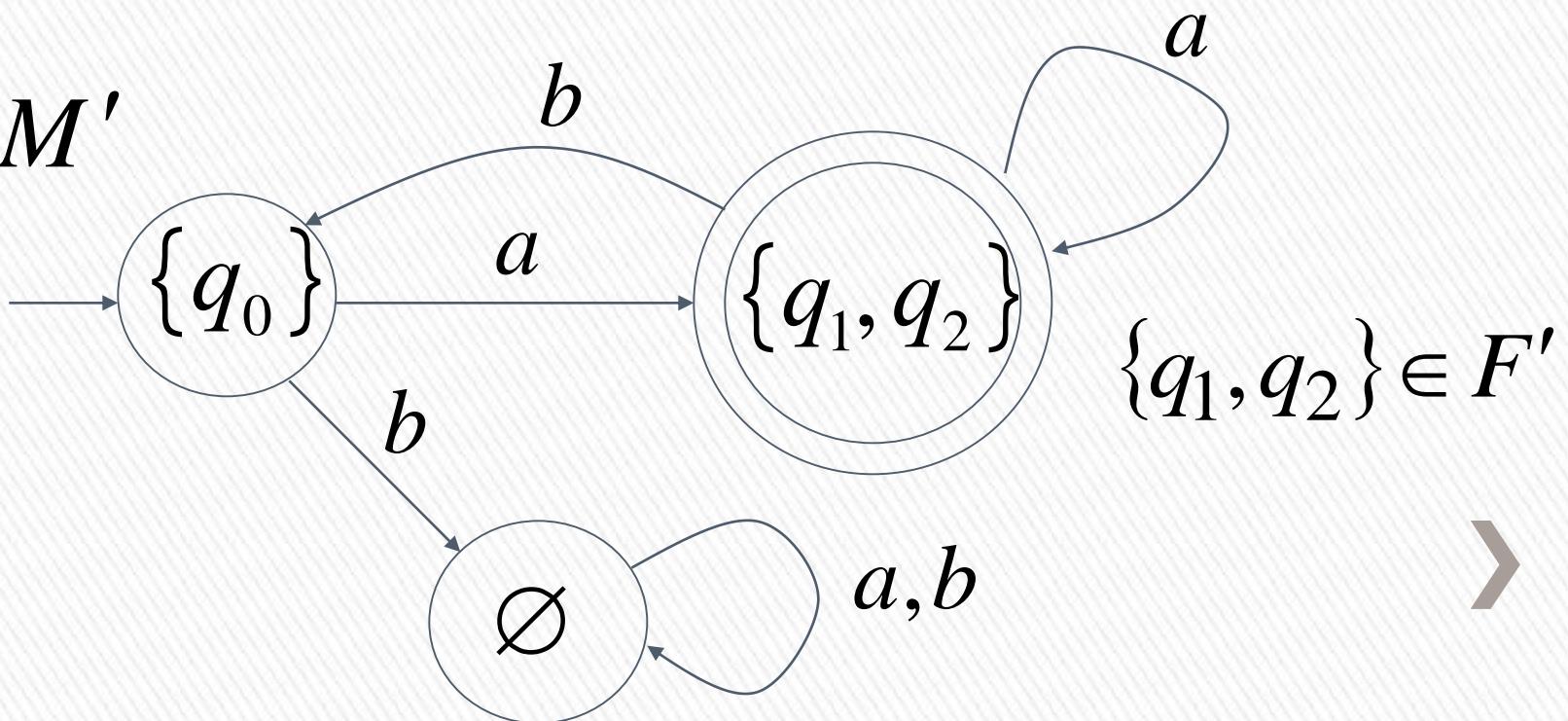
Example

NFA $M \gg$



$$q_1 \in F$$

DFA M'

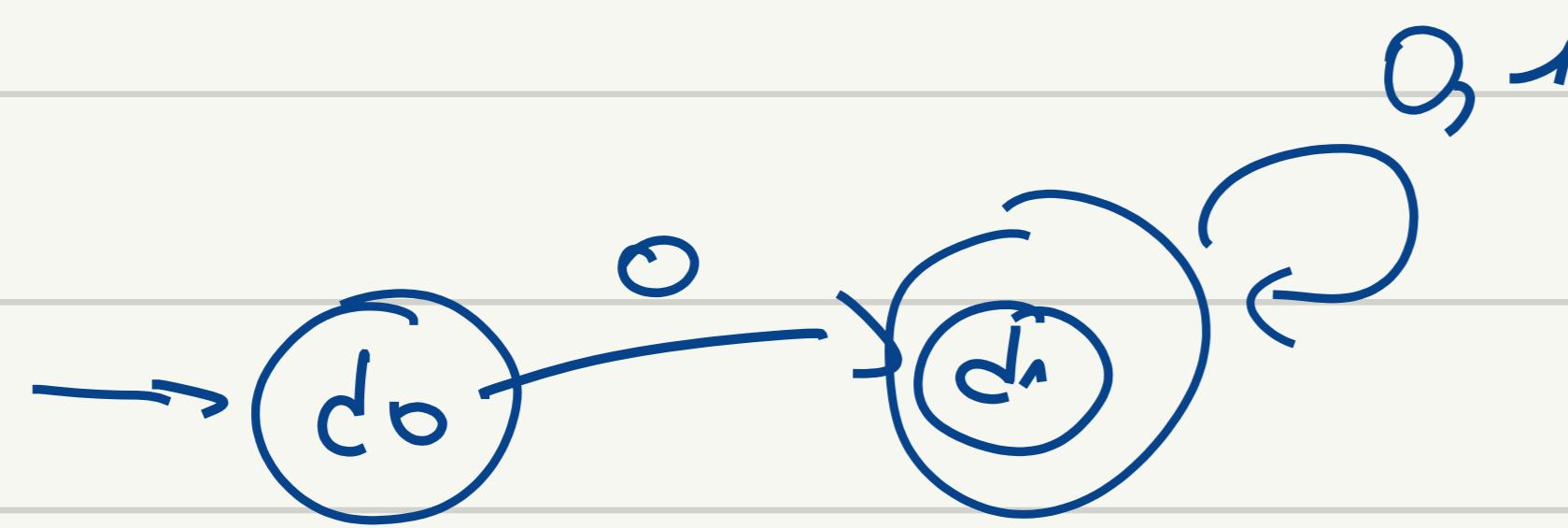


$$\{q_1, q_2\} \in F'$$



- An NFA accepts all strings over the alphabet $\Sigma = \{0, 1\}$ that begin with 0. Draw nfa, dfa

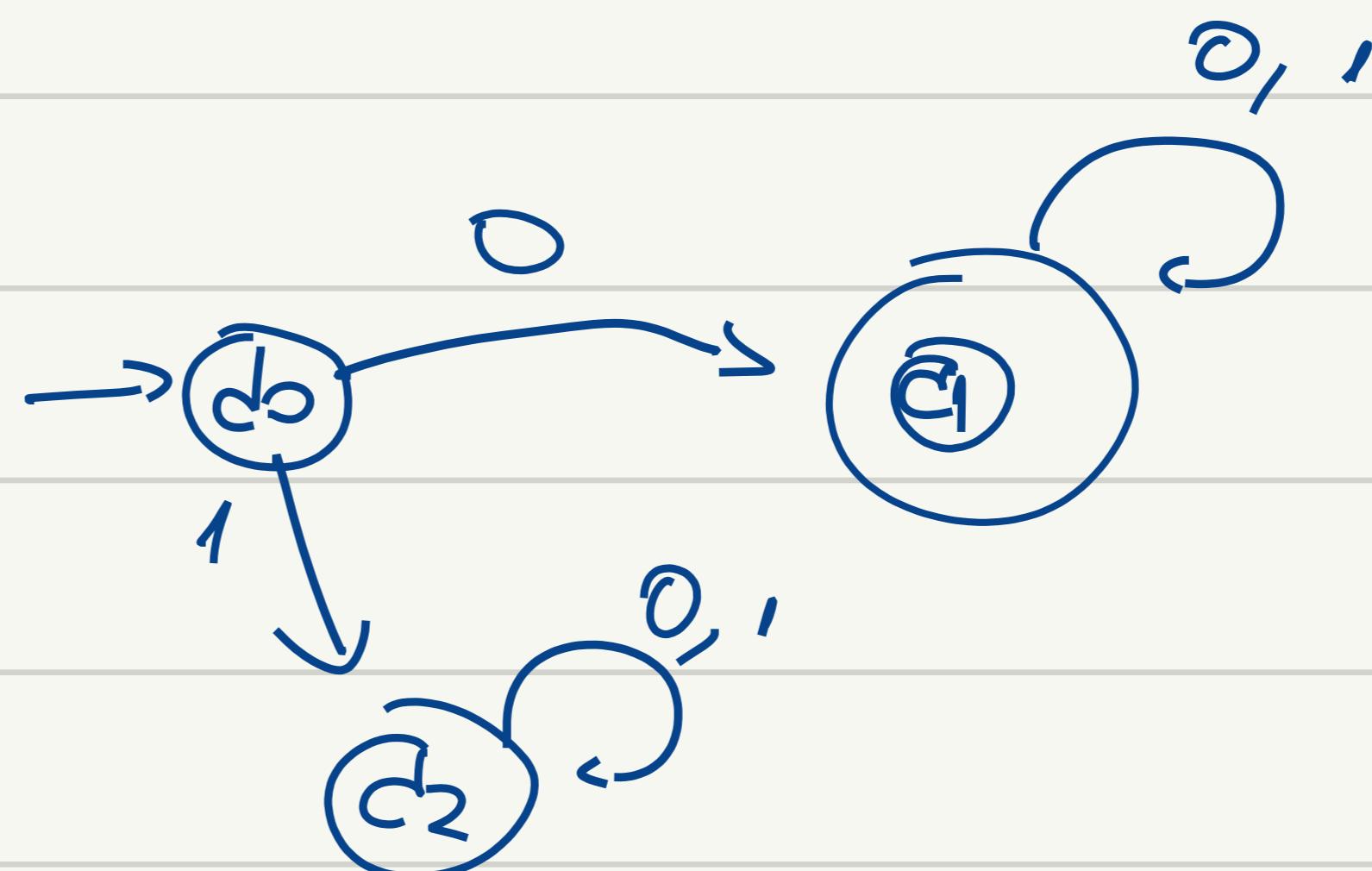
NFA



	0	1
d ₀	d ₁	\emptyset
d ₁	d ₁	d ₁

	0	1
d ₀	d ₁	d ₂
d ₁	d ₂	d ₂

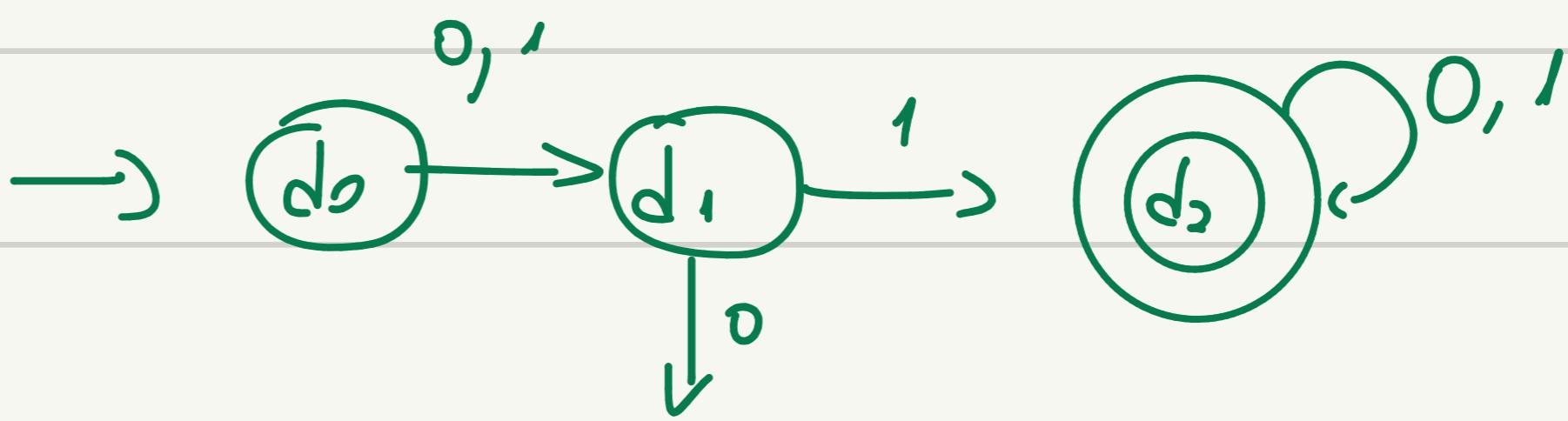
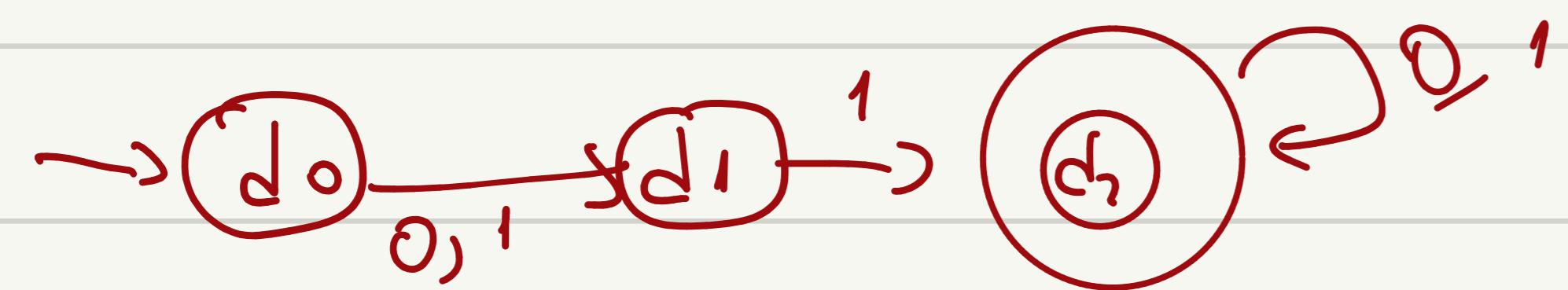
DFA



	0	1
d ₀	d ₁	d ₂
d ₁	d ₁	d ₁

- Design an NFA for the language that accepts strings over $\{0, 1\}$ second symbol 2.

NFA

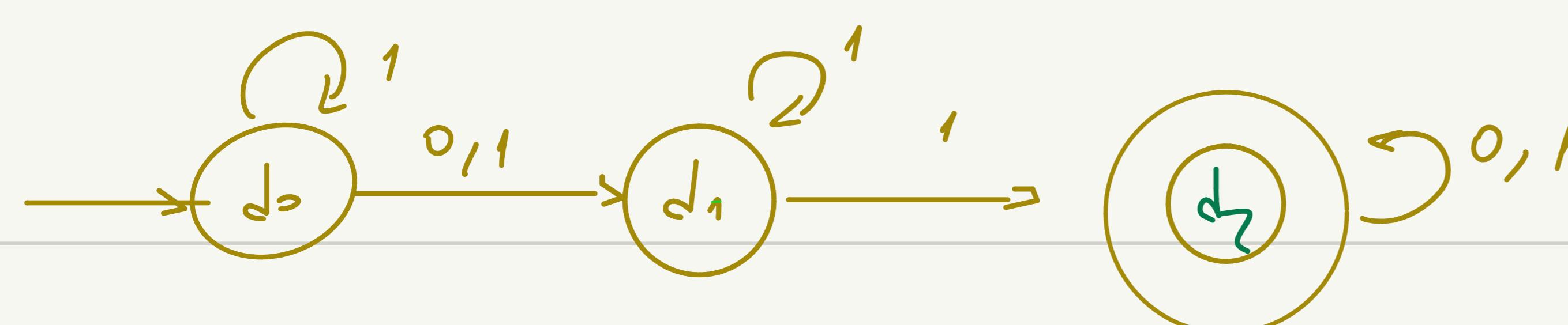


	0	1
d0	d1	d1
d1	\emptyset	d2
d2	d2	d2

```

graph LR
    start(( )) --> d0((d0))
    d0 -- "0, 1" --> d1((d1))
    d1 -- "1" --> d2((d2))
    d2 -- "0, 1" --> d0
    d2 -- "0, 1" --> d3((d3))
    style start fill:none,stroke:none
    style d0 fill:none,stroke:none
    style d1 fill:none,stroke:none
    style d2 fill:none,stroke:none
    style d3 fill:none,stroke:none
  
```

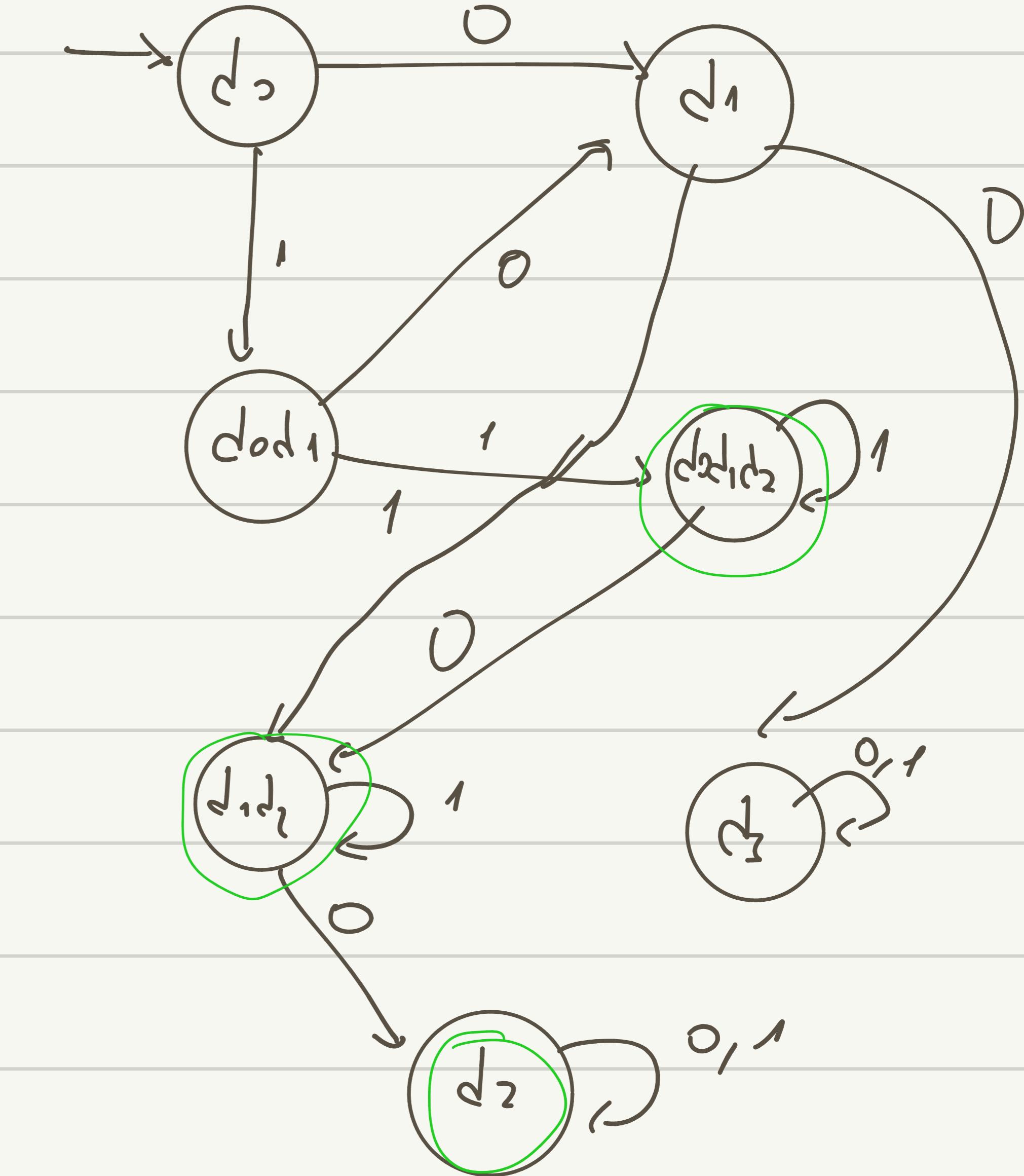
	0	1
d0	d1	d1
d1	d3	d2
d2	d2	d2
d3	d3	d1



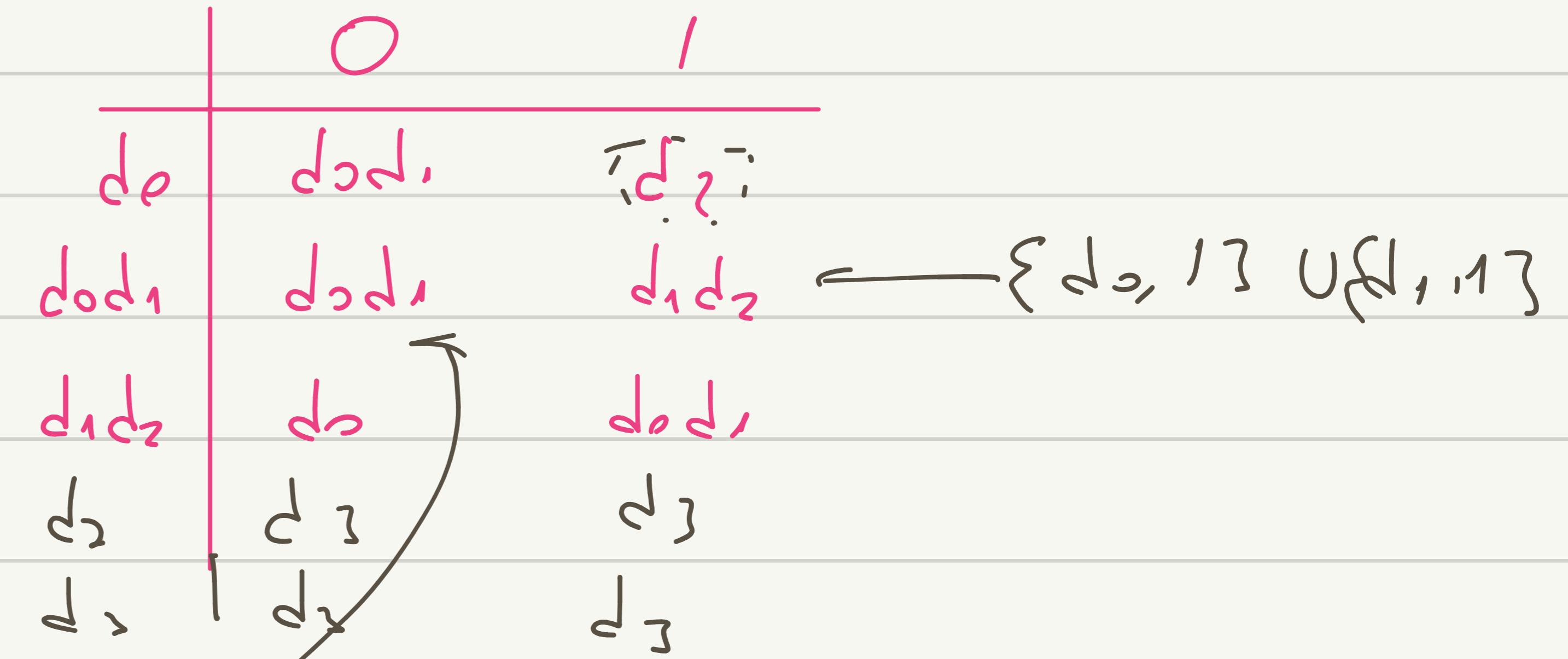
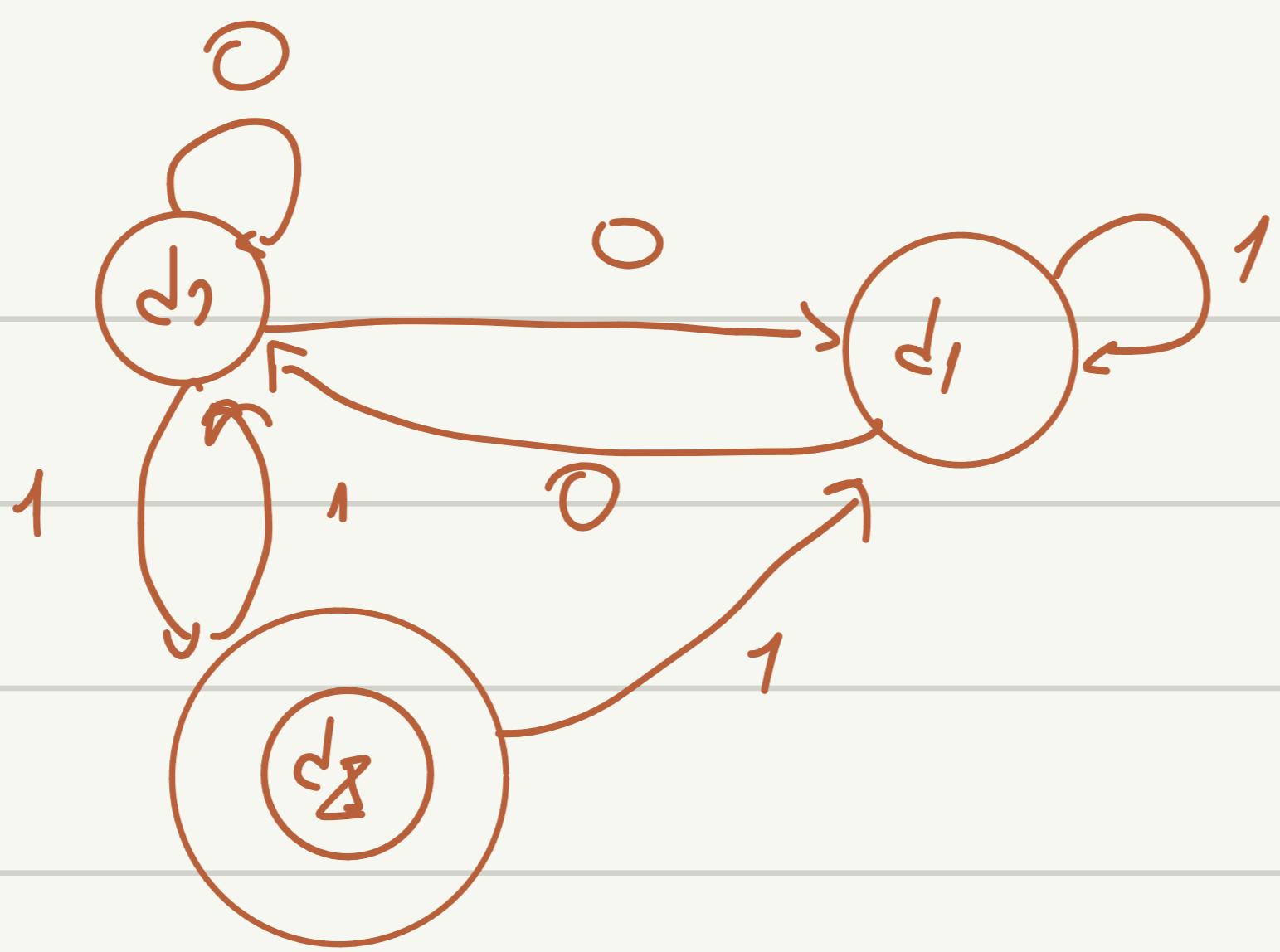
	d_0	d_1
d_0	d_0	d_0d_1
d_1	\emptyset	d_1d_2
d_2	d_2	d_2

$0 \quad 1$
1 1

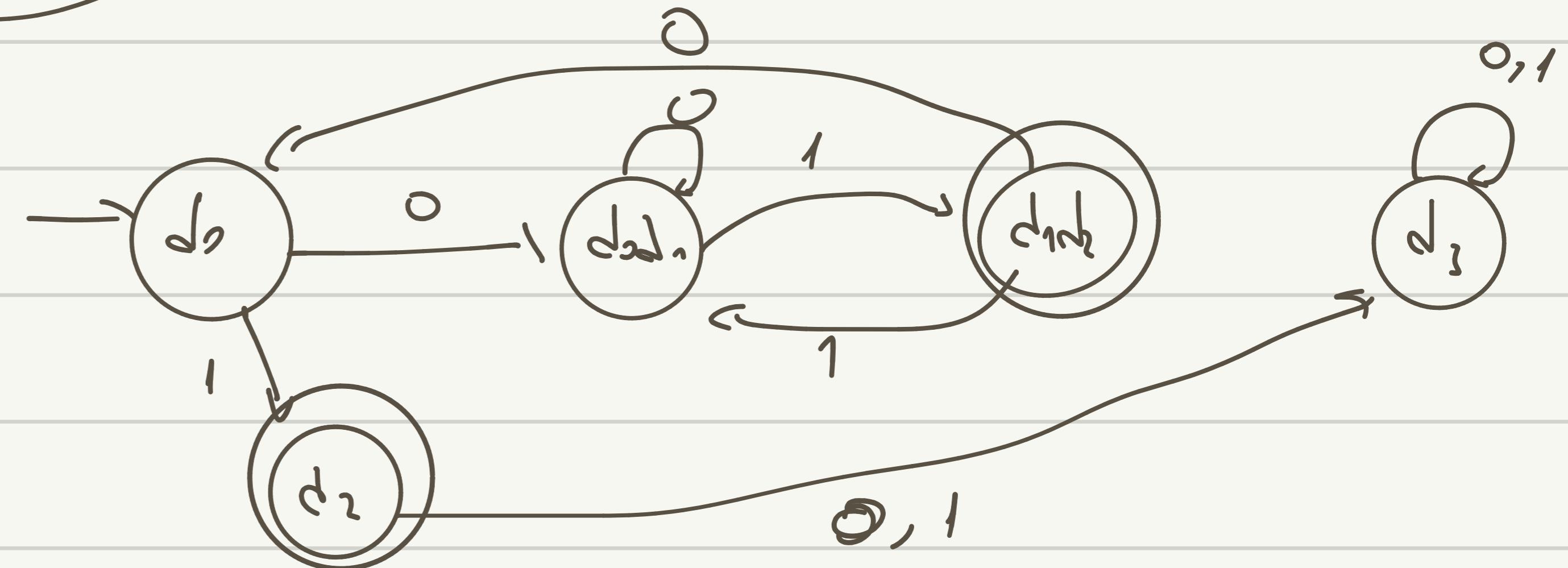
	d_0	d_1
d_0	d_0	d_0d_1
d_1	d_1	$d_0d_1d_2$
d_2	d_2	d_1d_2
d_3	d_3	d_1d_2



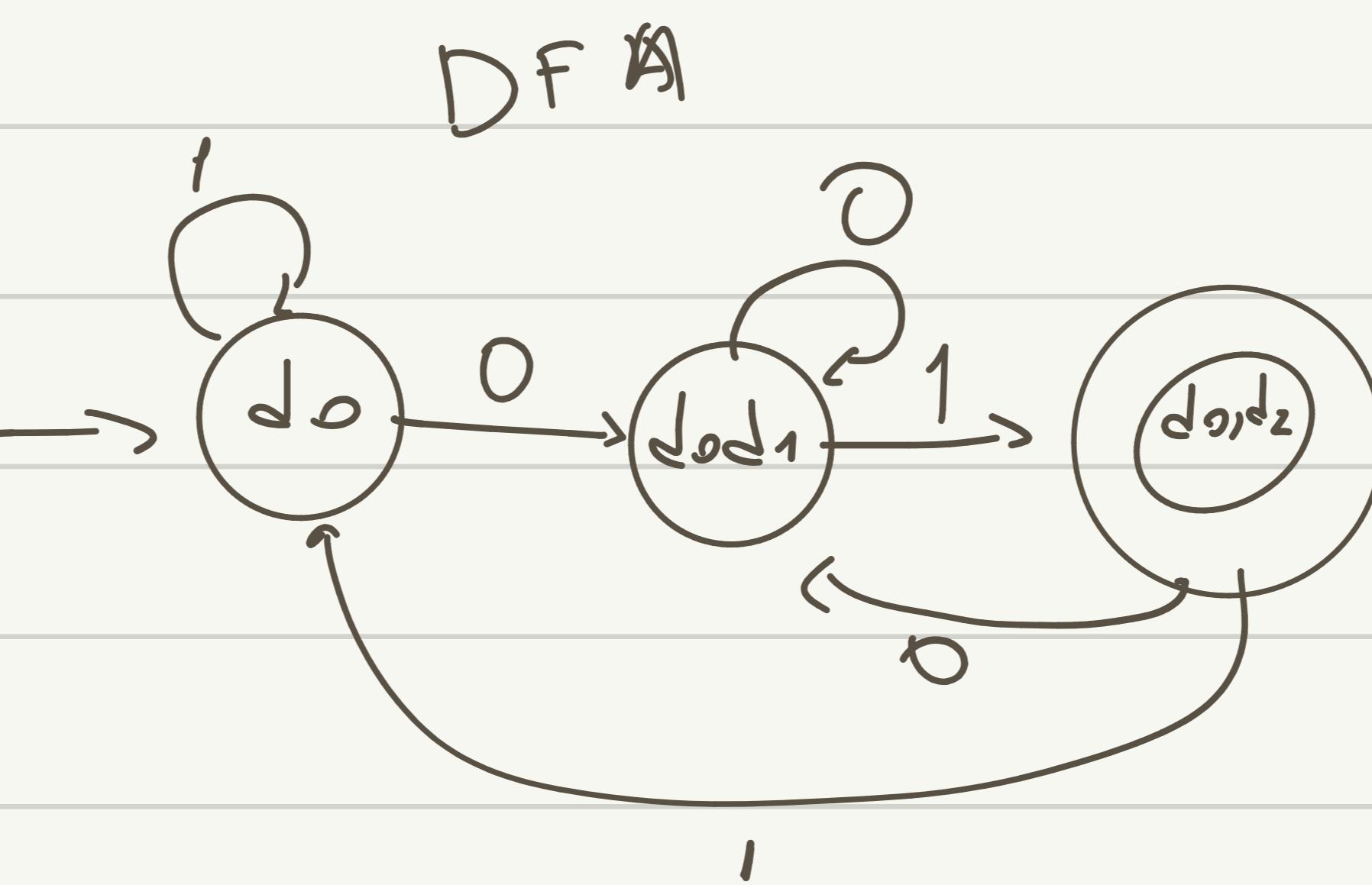
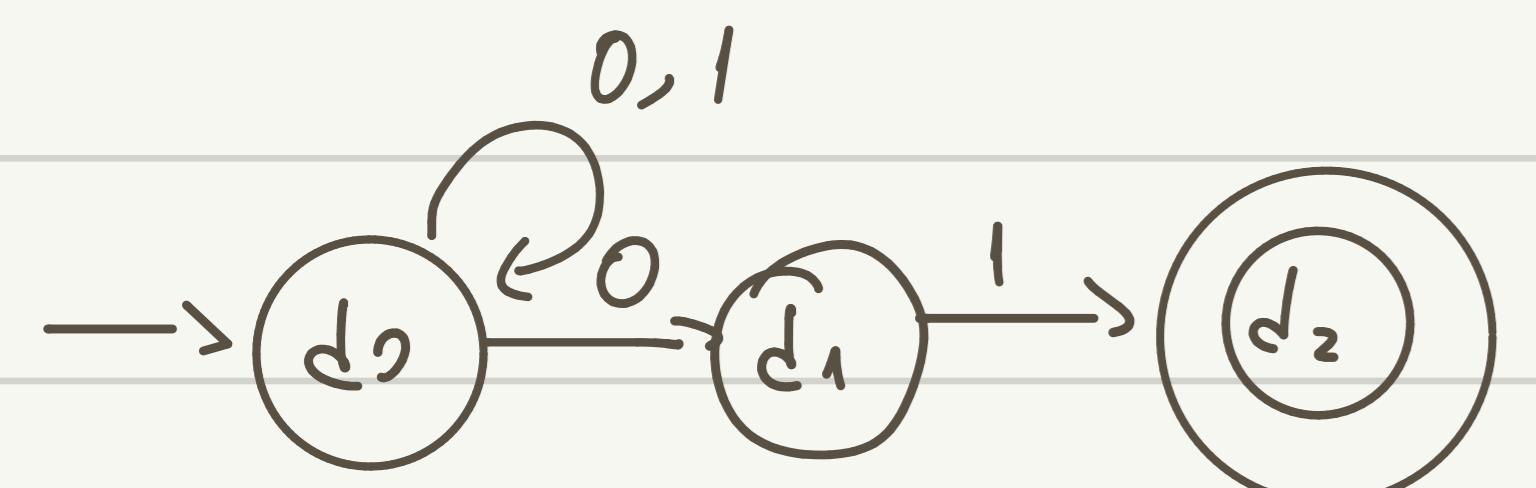
	0	1
d ₀	d ₀ d ₁	d ₂
d ₁	d ₀	d ₁
d ₂	∅	d ₀ d ₁



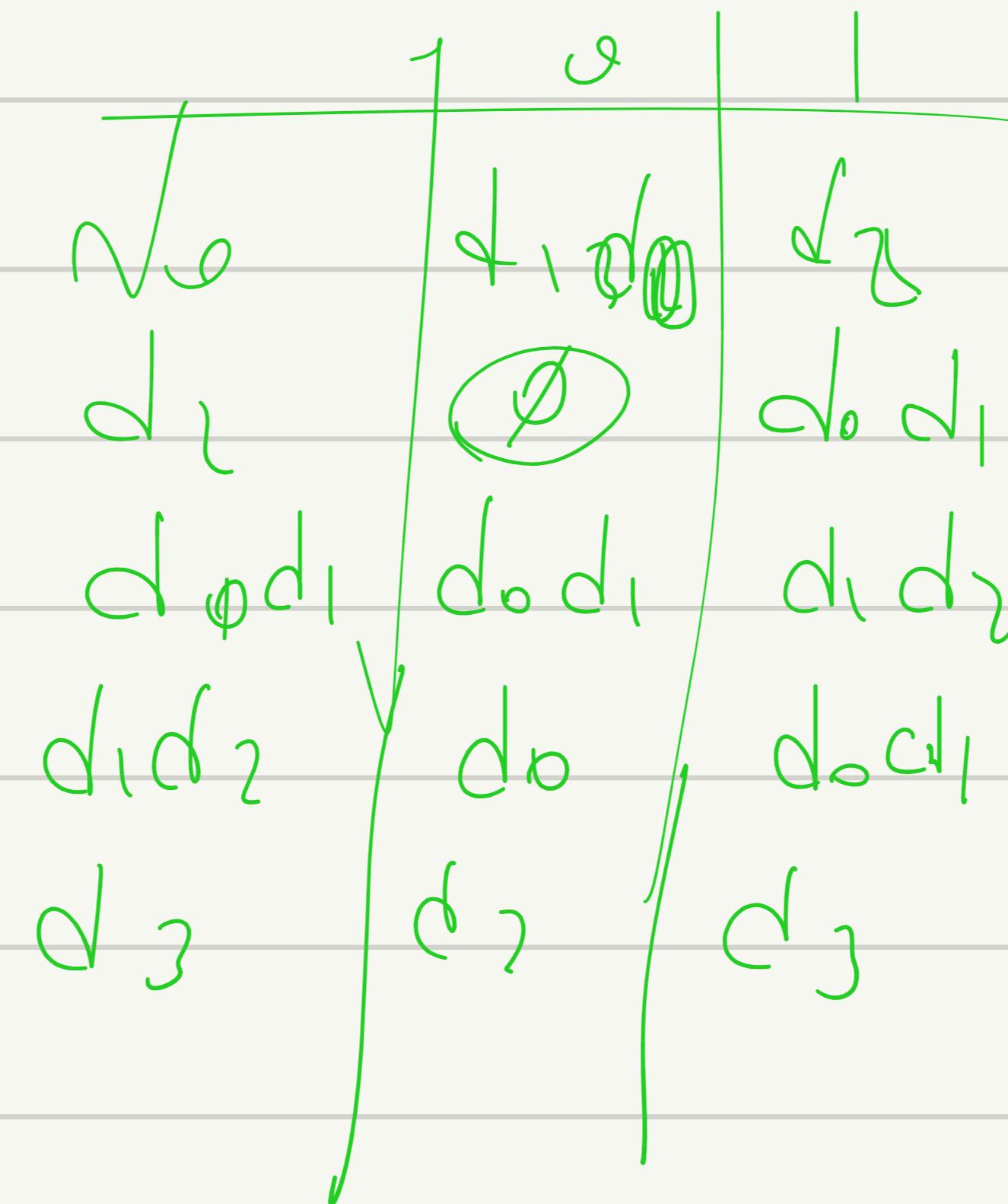
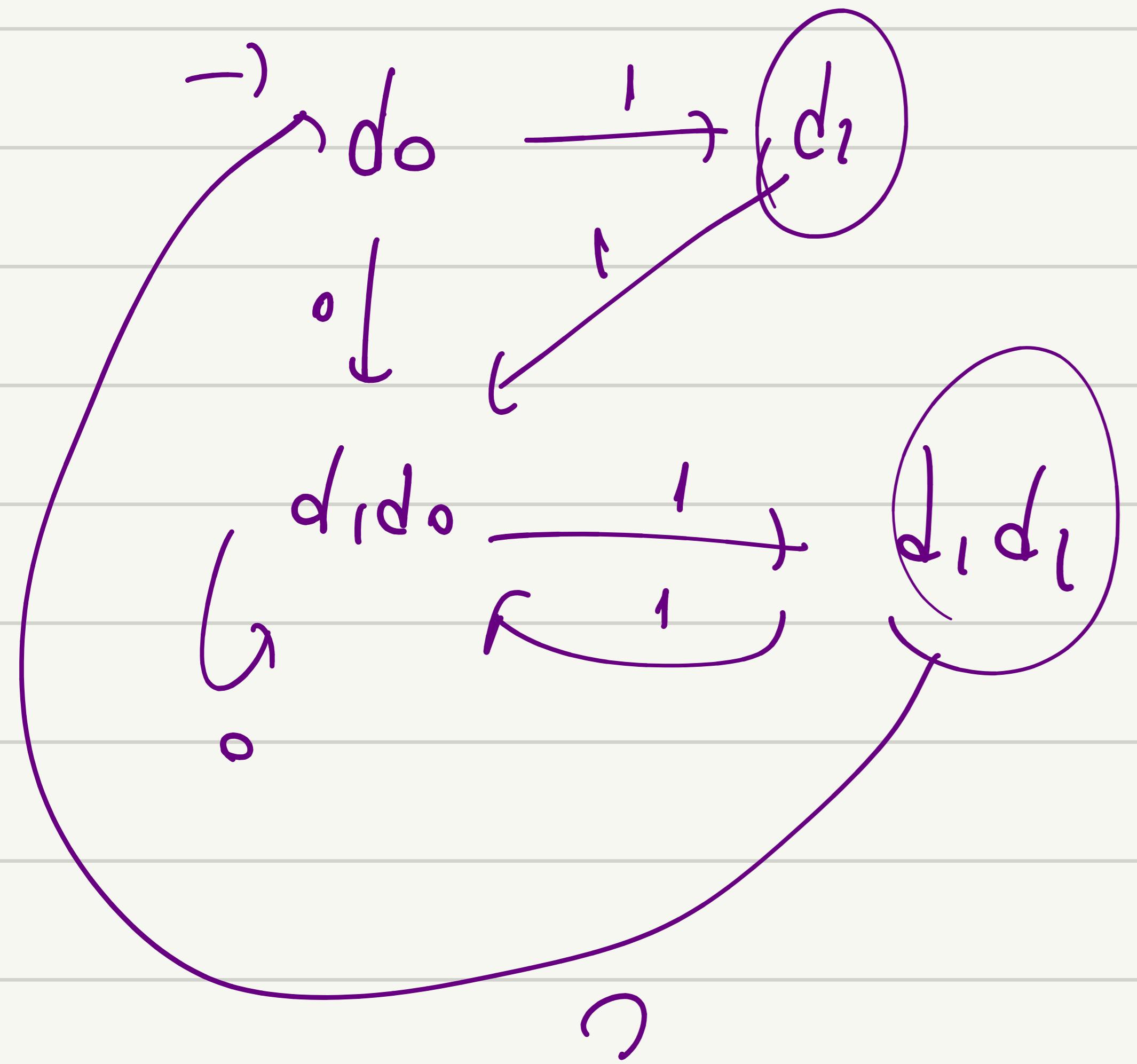
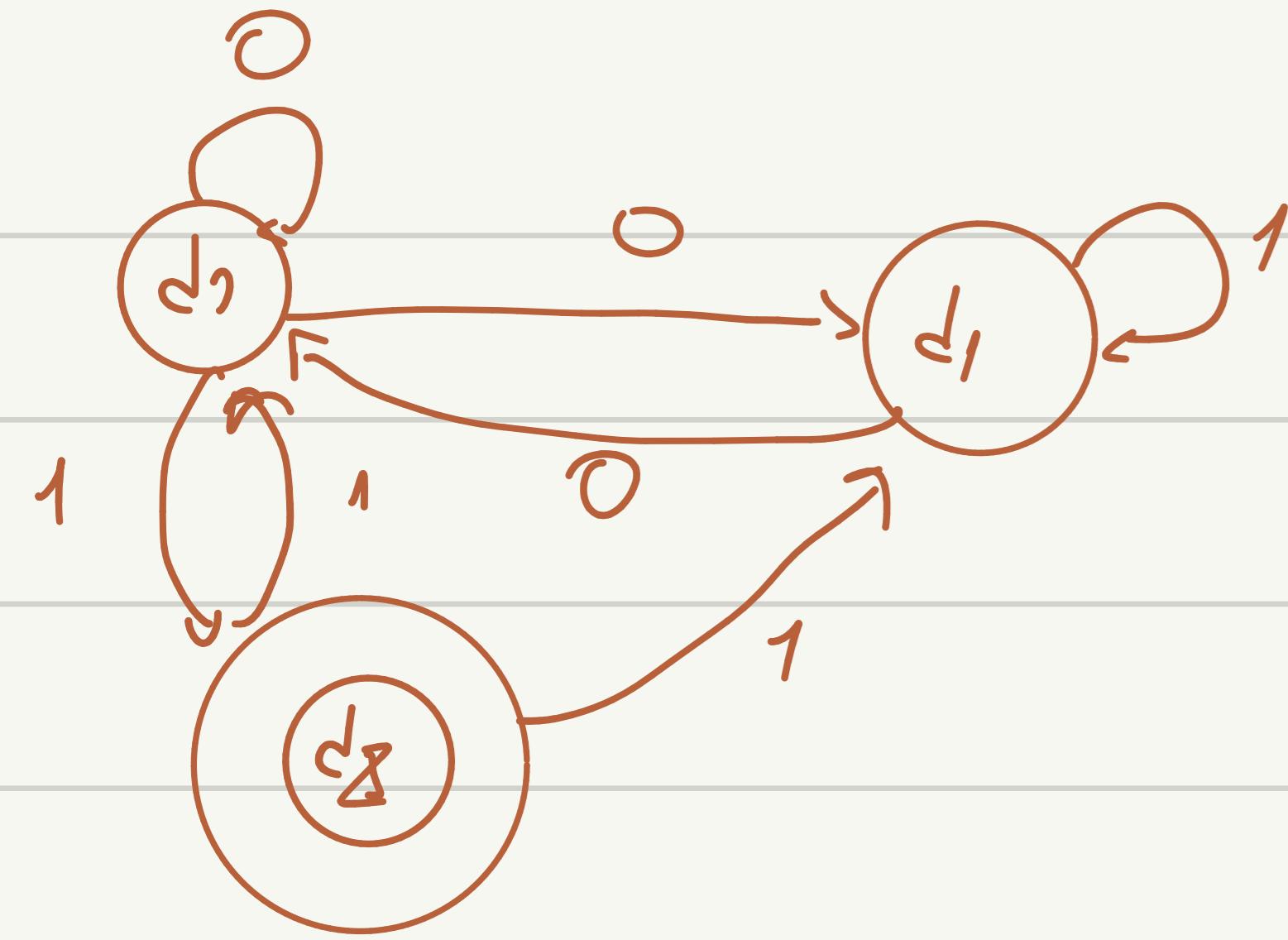
{d₀, 0} ∪ {d₁, 0}



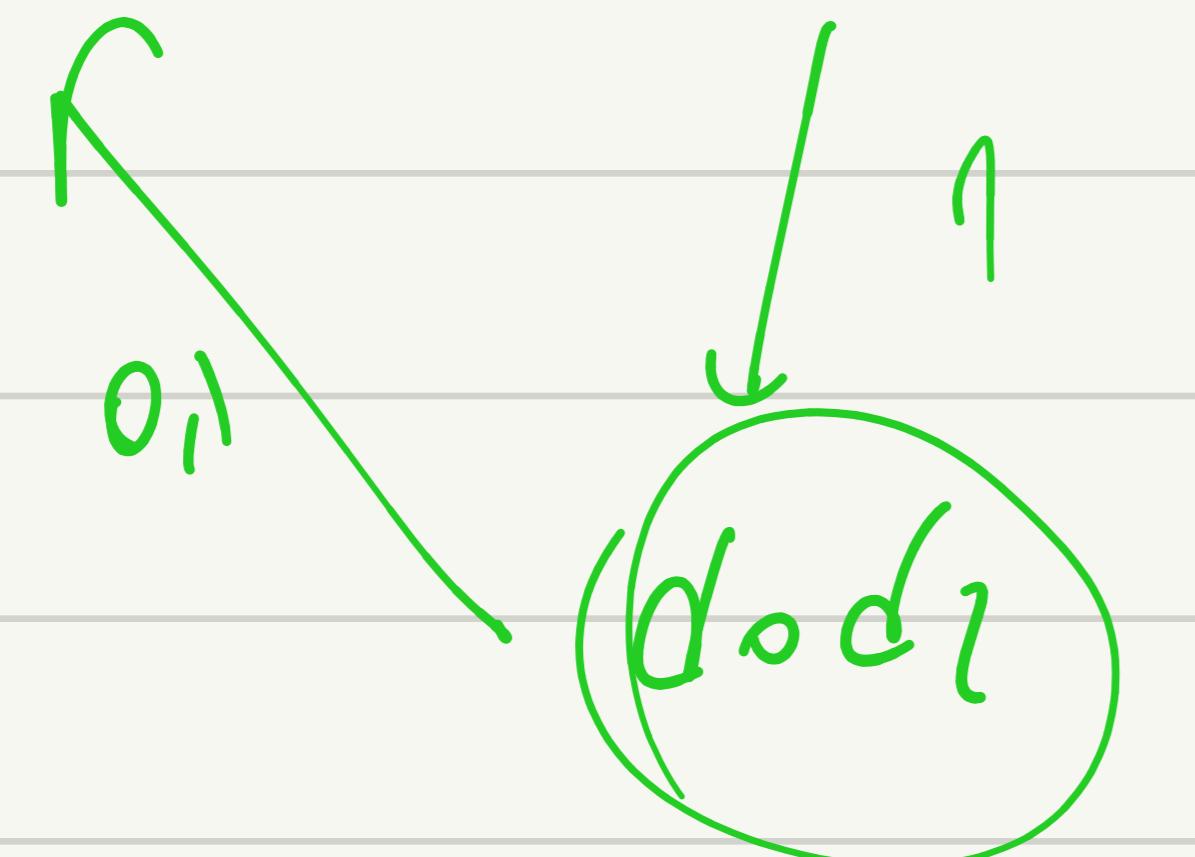
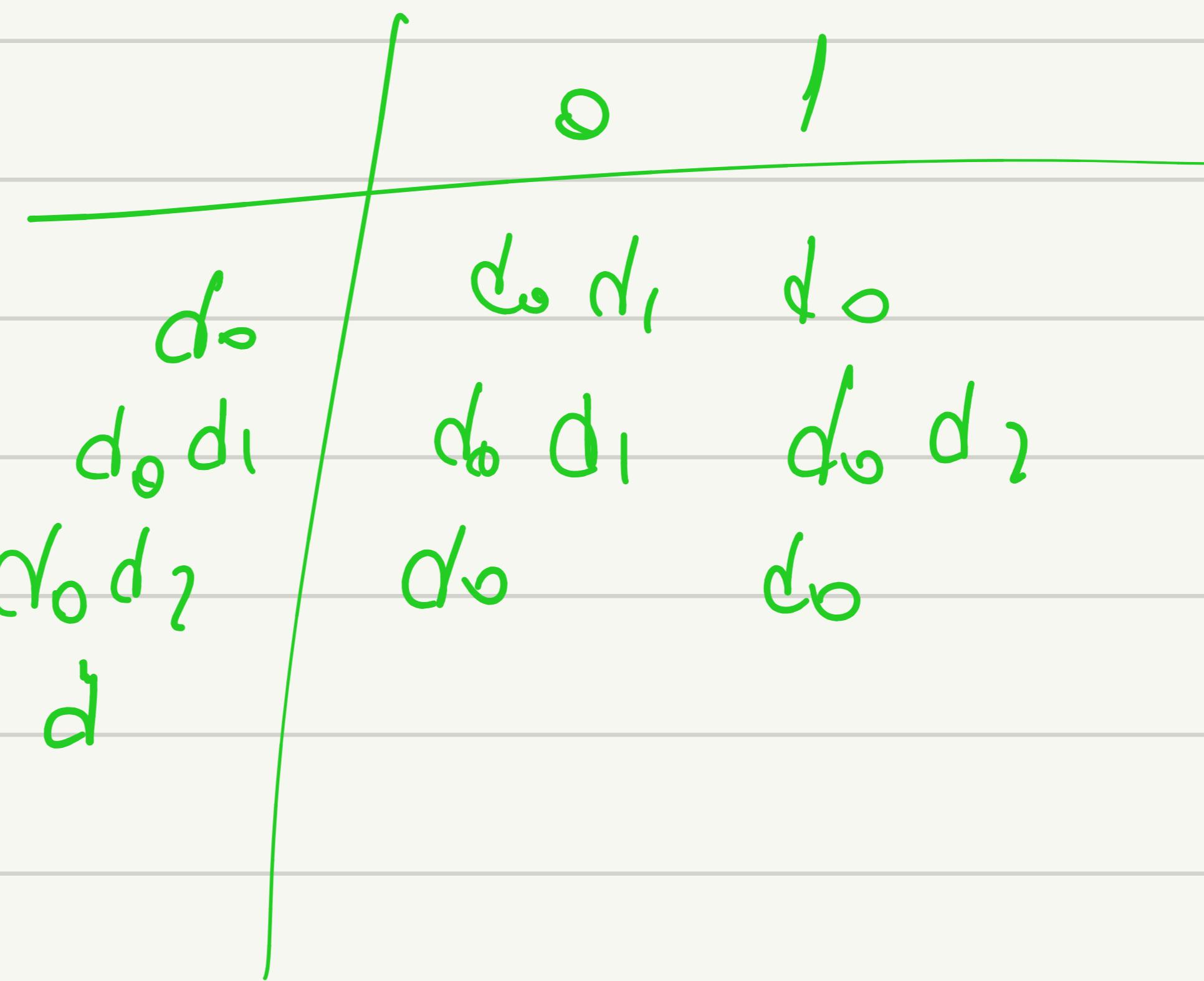
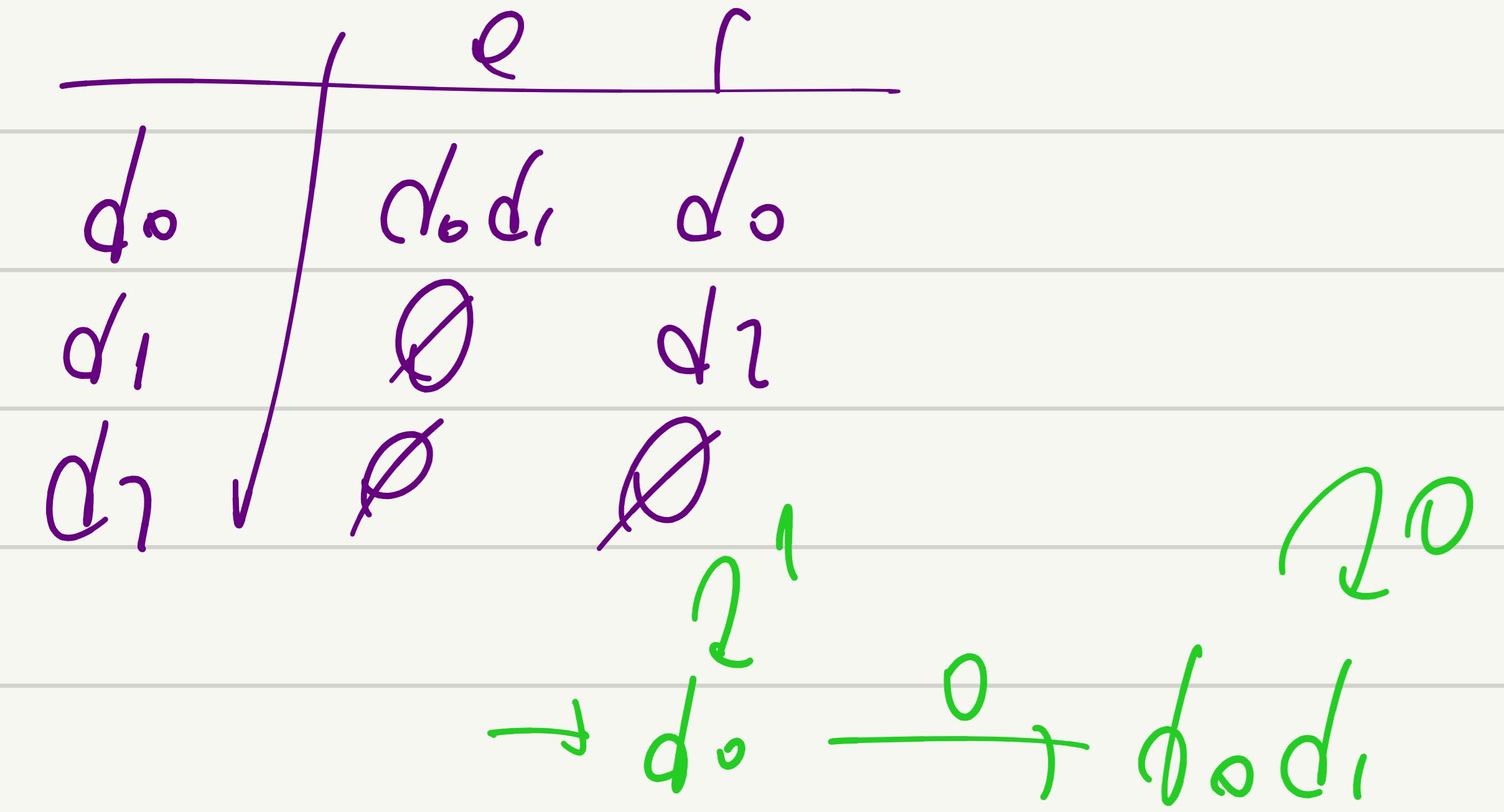
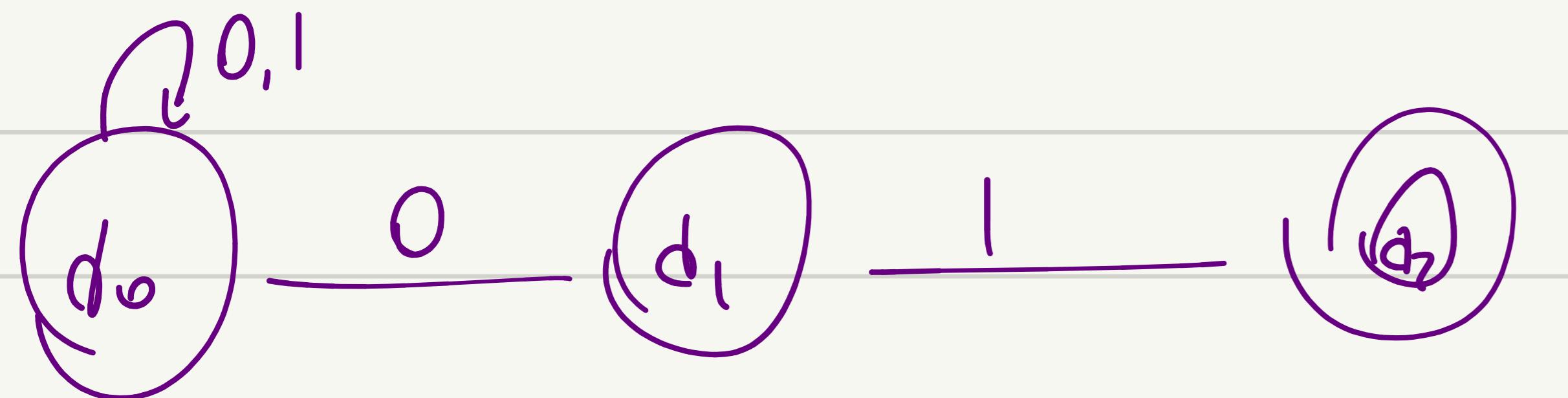
NFA ends with '01'



	0	1
d_0	d_0d_1	d_0
d_0d_1	d_0d_1	d_0d_2
d_0d_2	d_1	d_0



NFA ends with '01'

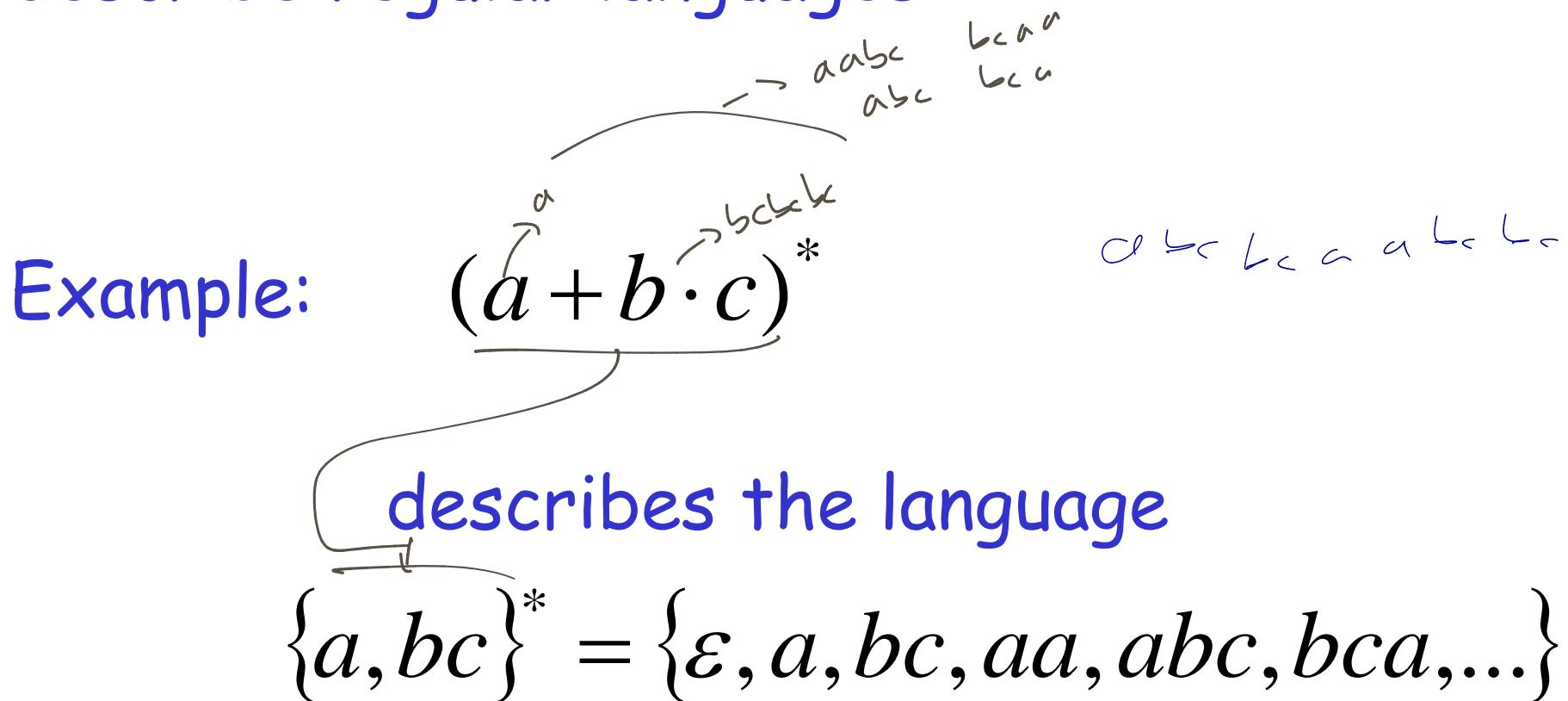


Regular Expressions

Regular Expressions

Regular expressions
describe regular languages

recognizes → terminal n.
↓
accept?



Recursive Definition

Primitive regular expressions:

\emptyset , ϵ , a

Given regular expressions r_1 and r_2

Operator precedence

* → highest

·

+

$r_1 + r_2$
 $r_1 \cdot r_2$
 r_1^*
 (r_1)

Are regular expressions

Examples

A regular expression:

$$(a + b \cdot c)^* \cdot (c + \emptyset)$$

d

Not a regular expression:

$$\cancel{(a + b^+)}$$

Languages of Regular Expressions

$L(r)$: language of regular expression r

Example

$$L((a+b \cdot c)^*) = \{\varepsilon, a, bc, aa, abc, bca, \dots\}$$

Definition

For primitive regular expressions:

$$L(\emptyset) = \emptyset$$

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

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

Definition (continued)

For regular expressions r_1 and r_2

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

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

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

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

Example

ϵ

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

$$\begin{aligned} L((a+b) \cdot a^*) &= L(a+b) L(a^*) \\ &= L(a+b) L(a^*) \\ &= (L(a) \cup L(b)) (L(a))^* \\ &= (\{a\} \cup \{b\}) (\{a\})^* \\ &= \{a, b\} \{\epsilon, a, aa, aaa, \dots\} \\ &= \{a, aa, aaa, \dots, b, ba, baa, \dots\} \end{aligned}$$

Example

Regular expression

$$r = (a + b)^* (a + bb)$$

a a a a a \rightarrow a a a a a $\perp L$

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

Example

Regular expression

$$r = (\cancel{aa})^* (bb)^* b$$

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

Example

Regular expression

$$r = (0+1)^* 00 (0+1)^*$$

$L(r) = \{ \text{all strings containing substring } 00 \}$

Example

Regular expression

$$r = (1 + 01)^*(0 + \epsilon)$$

1 0 1 1 0 0 1 0 1 0 0

$L(r) = \{ \text{all strings without substring } 00 \}$

Equivalent Regular Expressions

Definition:

Regular expressions r_1 and r_2

are **equivalent** if $L(r_1) = L(r_2)$

Example

$L = \{ \text{all strings without substring } 00 \}$

$$r_1 = (\cancel{1} + 01)^*(0 + \epsilon)$$

$$r_2 = (\cancel{1^*} \cancel{01} \cancel{1^*})^*(0 + \epsilon) + 1^*(0 + \epsilon)$$

$$L(r_1) = L(r_2) = L$$

r_1 and r_2
are equivalent
regular expressions

Regular Expressions and Regular Languages

Theorem

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} = \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Proof:

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} \cap \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} \cup \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Proof - Part 1

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} \subseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

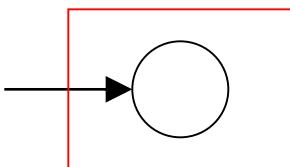
For any regular expression r
the language $L(r)$ is regular

Proof by induction on the size of r

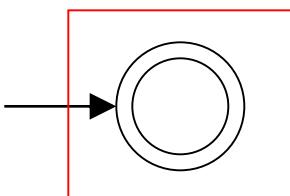
Induction Basis

Primitive Regular Expressions: \emptyset , ϵ , α

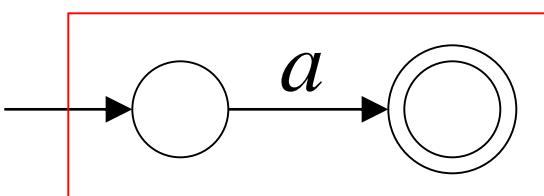
Corresponding
NFAs



$$L(M_1) = \emptyset = L(\emptyset)$$



$$L(M_2) = \{\epsilon\} = L(\epsilon)$$



$$L(M_3) = \{a\} = L(a)$$

regular
languages

Inductive Hypothesis

Suppose
that for regular expressions r_1 and r_2 ,
 $L(r_1)$ and $L(r_2)$ are regular languages

Inductive Step

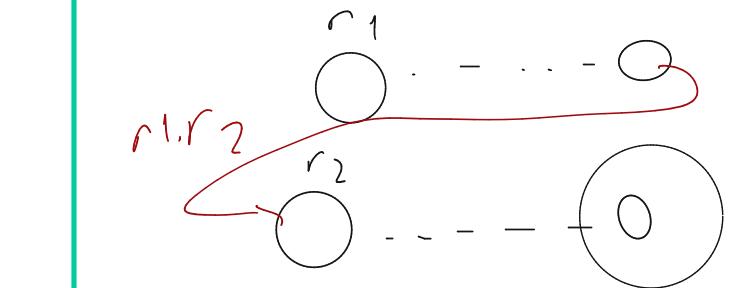
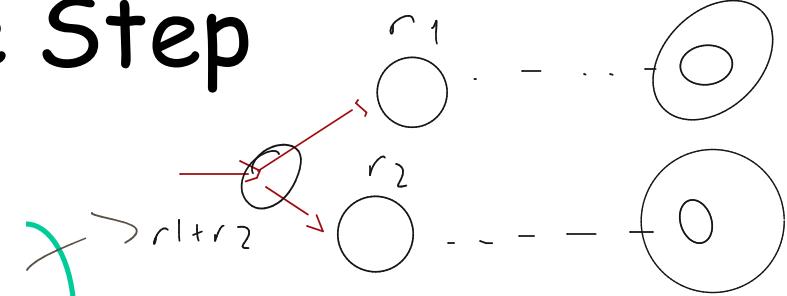
We will prove:

$$L(r_1 + r_2)$$

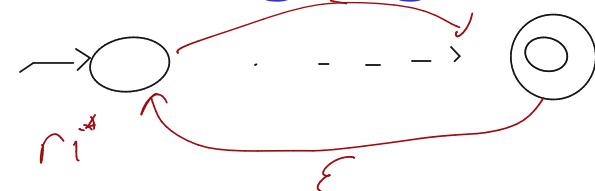
$$L(r_1 \cdot r_2)$$

$$L(r_1^*)$$

$$L((r_1))$$



Are regular
Languages



By definition of regular expressions:

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

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

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

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

By inductive hypothesis we know:

$L(r_1)$ and $L(r_2)$ are regular languages

We also know:

Regular languages are closed under:

Union

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

Concatenation

$$L(r_1) L(r_2)$$

Star

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

Therefore:

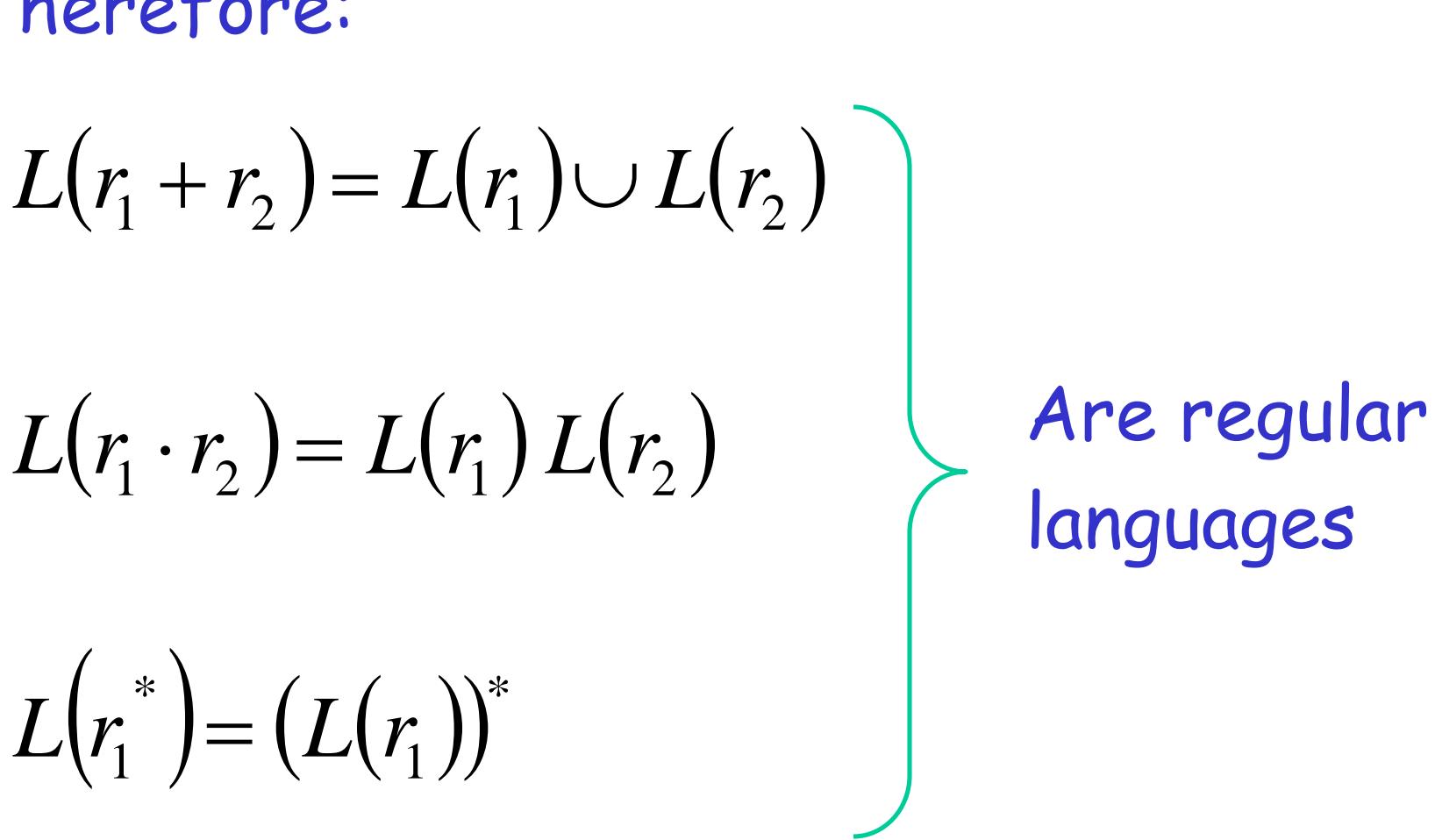
$$L(r_1 + r_2) = L(r_1) \cup L(r_2)$$

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

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

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

is trivially a regular language
(by induction hypothesis)

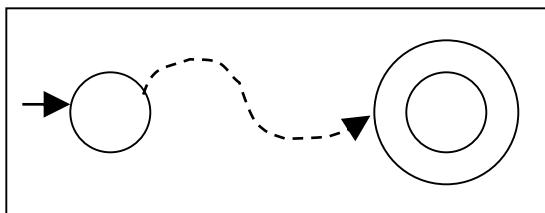


Are regular
languages

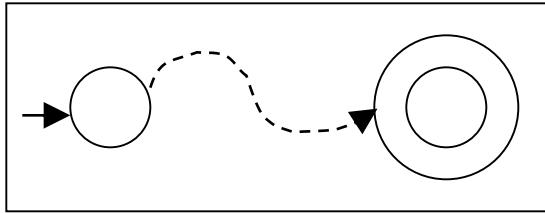
Using the regular closure of operations,
we can construct recursively the NFA M
that accepts $L(M) = L(r)$

Example: $r = r_1 + r_2$

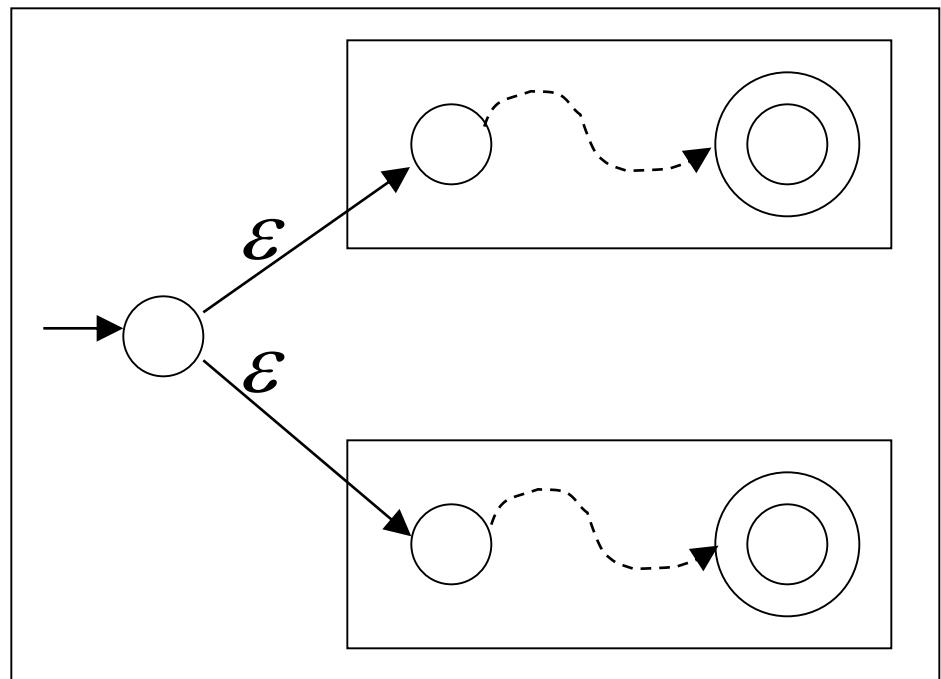
$$L(M_1) = L(r_1)$$



$$L(M_2) = L(r_2)$$



$$L(M) = L(r)$$



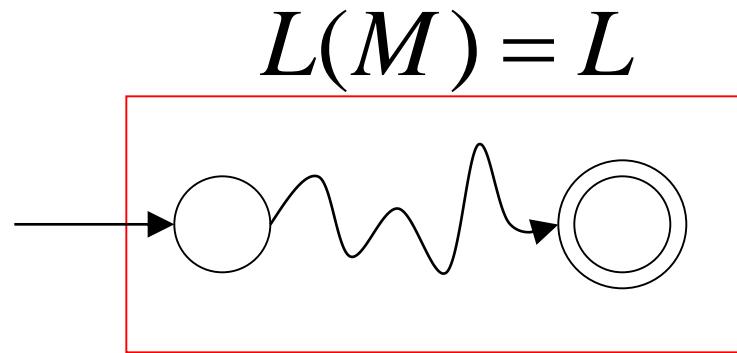
Proof - Part 2

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

For any regular language L there is a regular expression r with $L(r) = L$

We will convert an NFA that accepts L to a regular expression

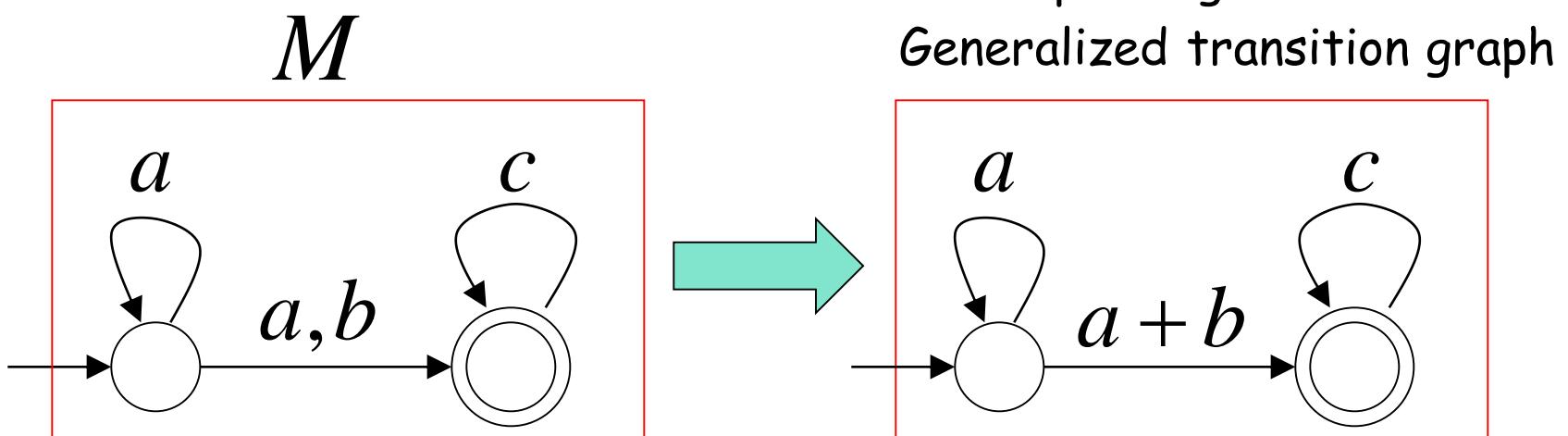
Since L is regular, there is a NFA M that accepts it



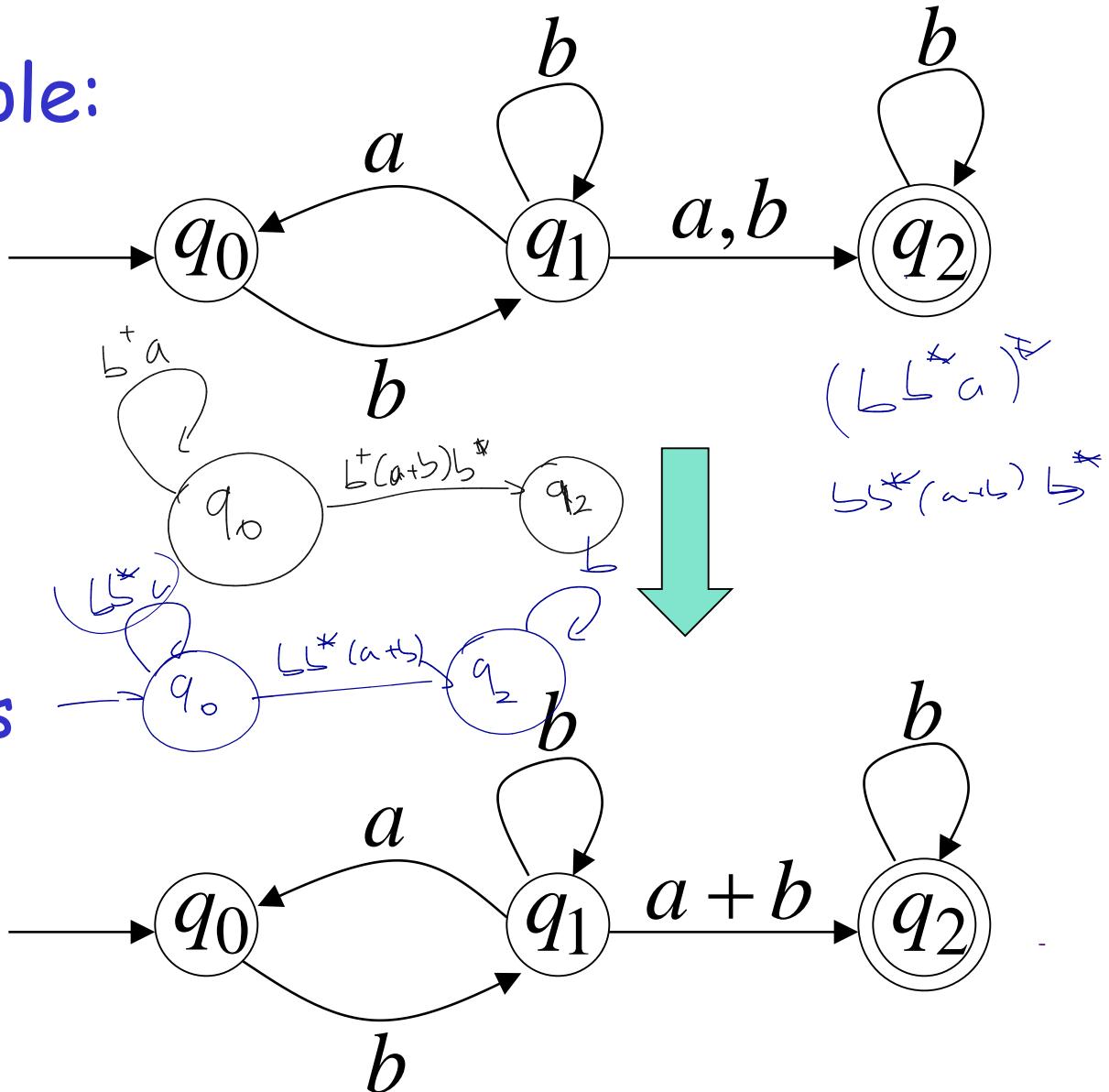
Take it with a single accept state

From M construct the equivalent
Generalized Transition Graph
in which transition labels are regular expressions

Example:

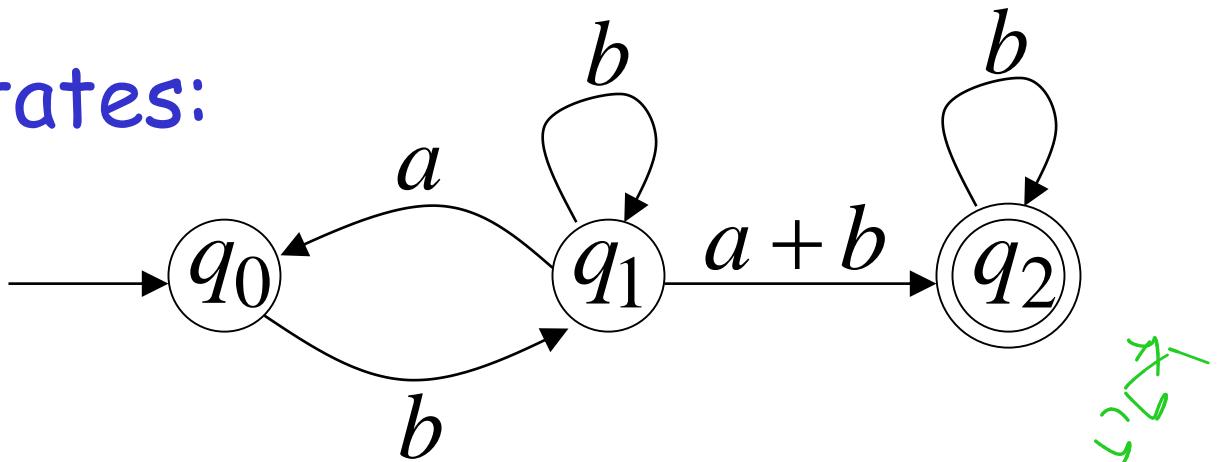


Another Example:



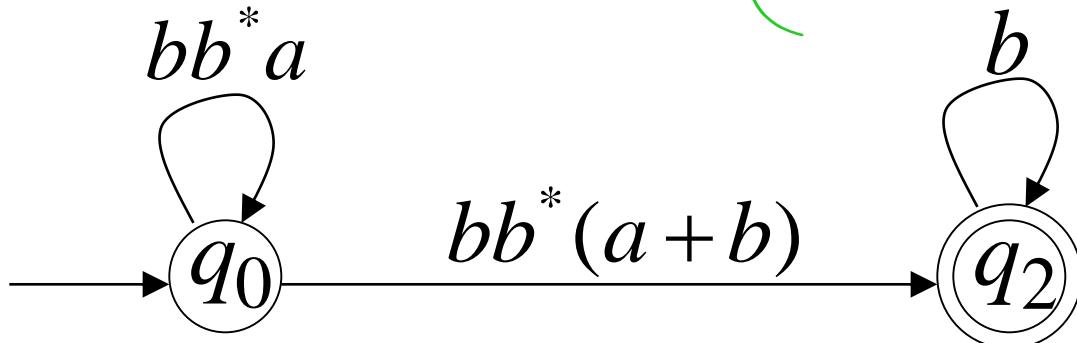
Transition labels
are regular
expressions

Reducing the states:

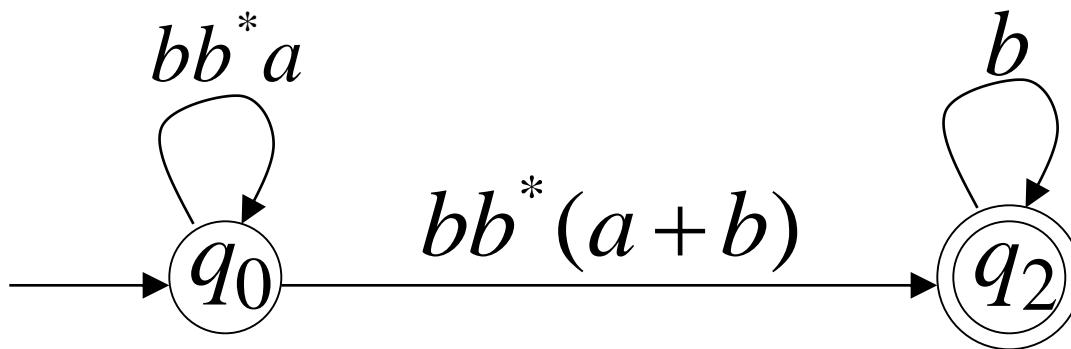


$$(L^*a)^* L^*(a+b) L^*$$

Transition labels
are regular
expressions



Resulting Regular Expression:

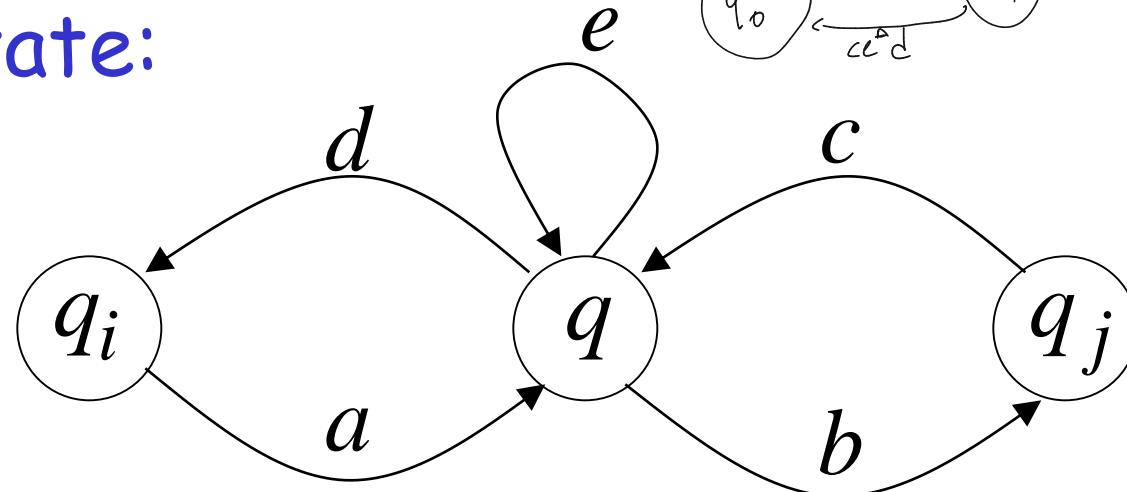
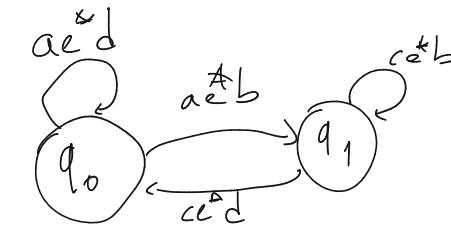


$$r = (bb^*a)^*bb^*(a+b)b^*$$

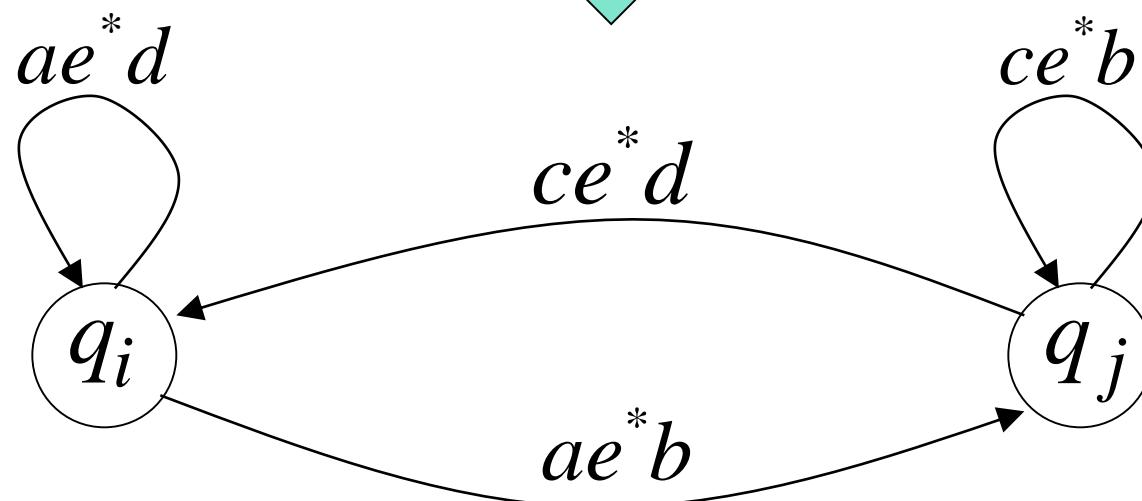
$$L(r) = L(M) = L$$

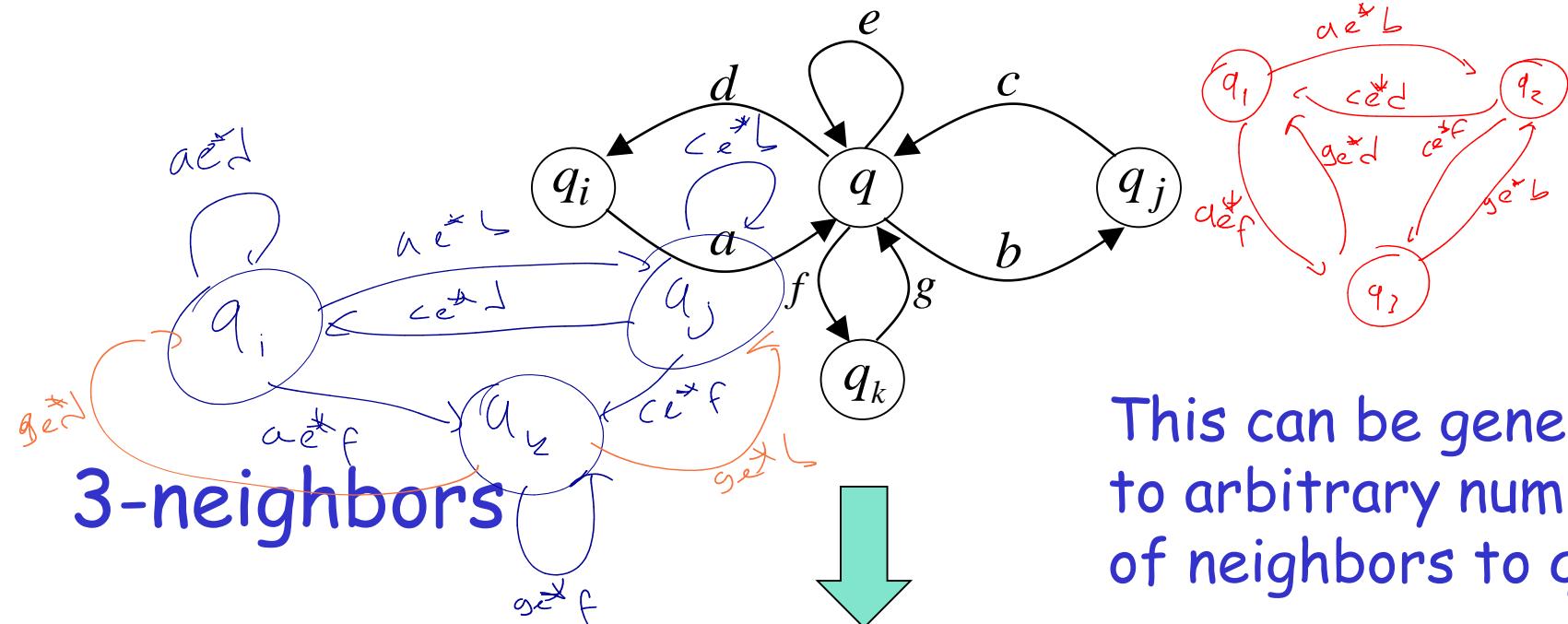
In General

Removing a state:

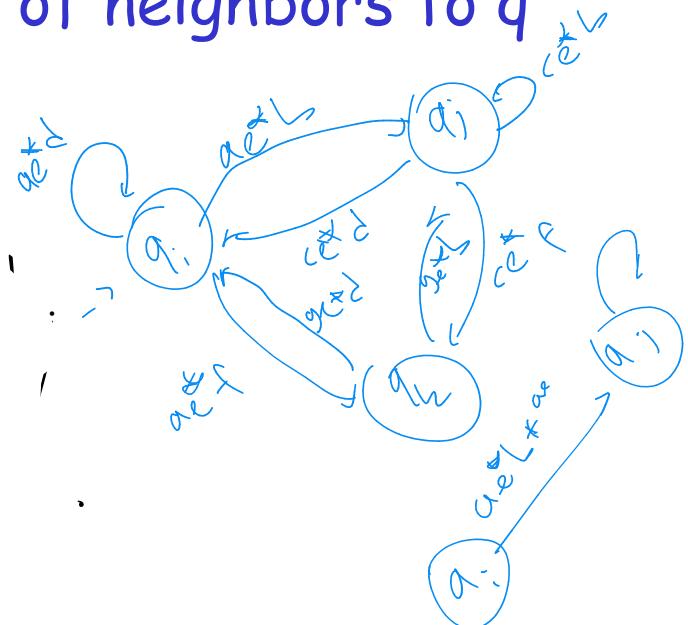


2-neighbors



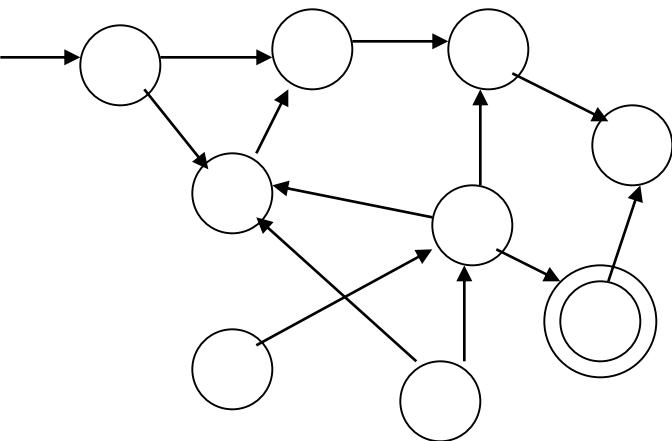


This can be generalized to arbitrary number of neighbors to q

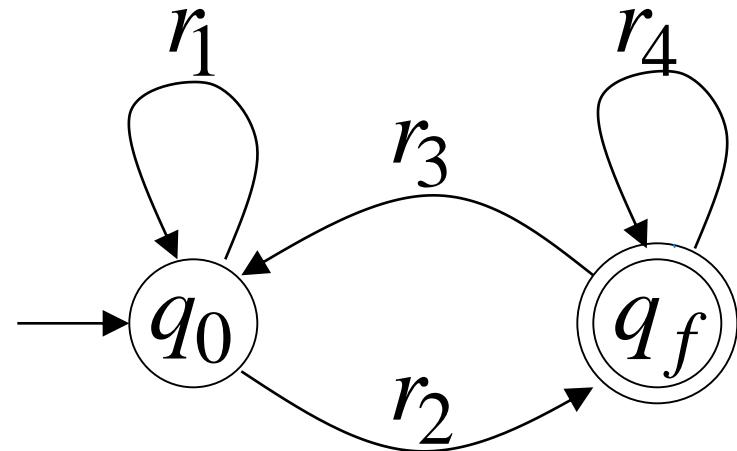


By repeating the process until two states are left, the resulting graph is

Initial graph



Resulting graph

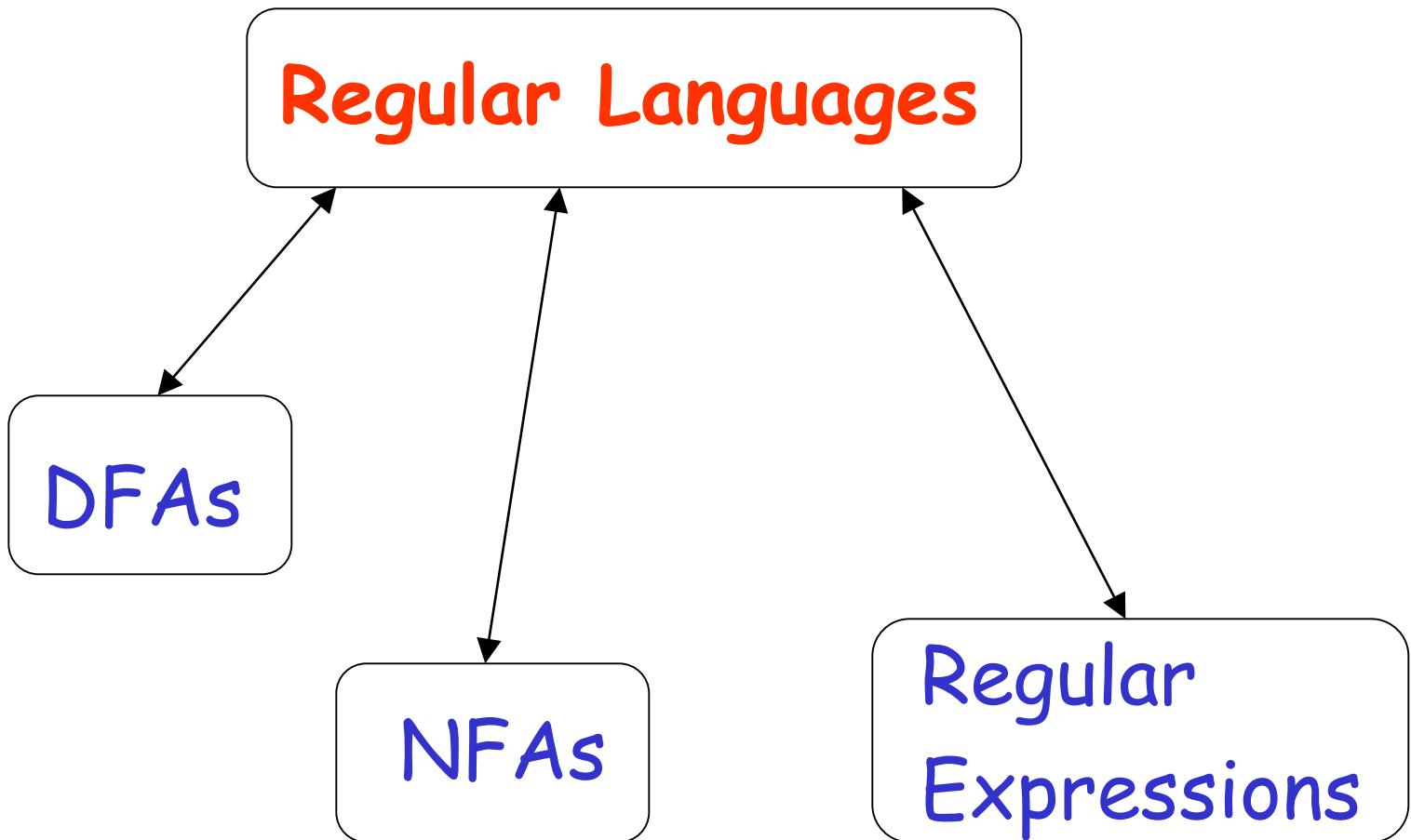


The resulting regular expression:

$$r_1^* r_2 \cdot (r_4 + r_3 r_1^* r_2)^*$$

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

Standard Representations of Regular Languages



When we say: We are given
a Regular Language L



We mean: Language L is in a standard
representation
(DFA, NFA, or Regular Expression)

BLM2502

Theory of

Computation

BLM2502 Theory of Computation

Course Outline

Week	Content
1	Introduction to Course
2	Computability Theory, Complexity Theory, Automata Theory, Set Theory, Relations, Proofs, Pigeonhole Principle
3	Regular Expressions
4	Finite Automata
5	Deterministic and Nondeterministic Finite Automata
6	Epsilon Transition, Equivalence of Automata
7	Pumping Theorem
8	April 10 - 14 week is the first midterm week
9	Context Free Grammars
10	Parse Tree, Ambiguity,
11	Pumping Theorem
12	Turing Machines, Recognition and Computation, Church-Turing Hypothesis
13	Turing Machines, Recognition and Computation, Church-Turing Hypothesis
14	May 22 - 27 week is the second midterm week
15	Review
16	Final Exam date will be announced

Week I - Introduction

Non-regular languages

(Pumping Lemma)

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 or NFA or RE that accepts L

Difficulty: this is not easy to prove
(since there is an infinite number of them)

Solution: use the Pumping Lemma !!!

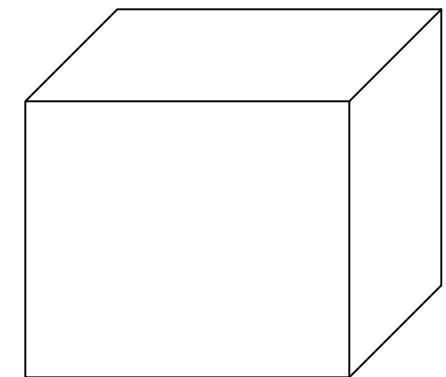
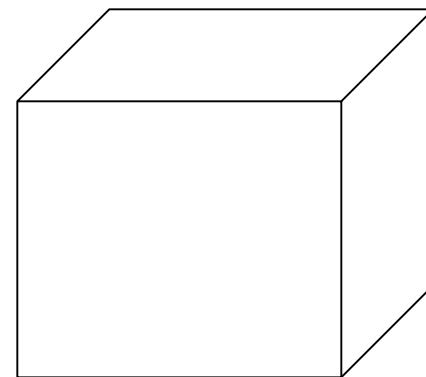
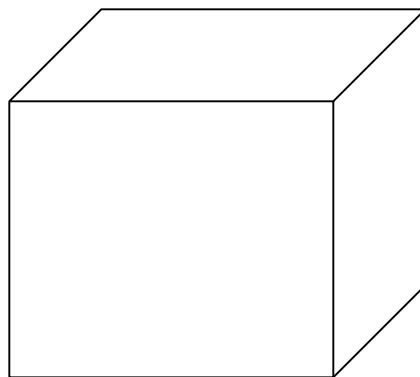


The Pigeonhole Principle

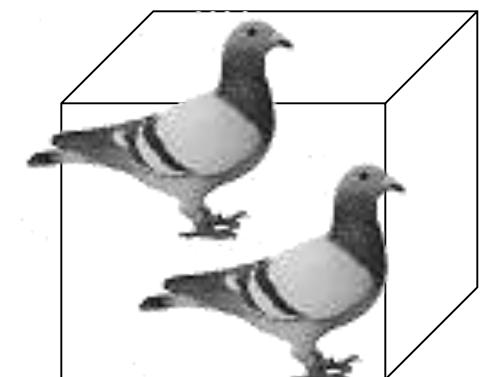
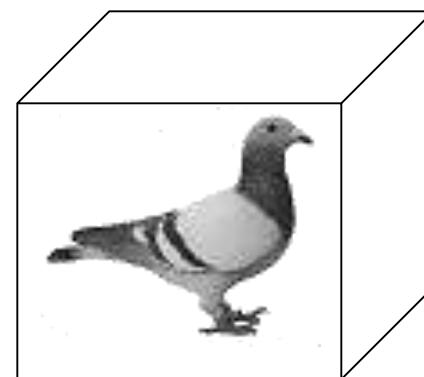
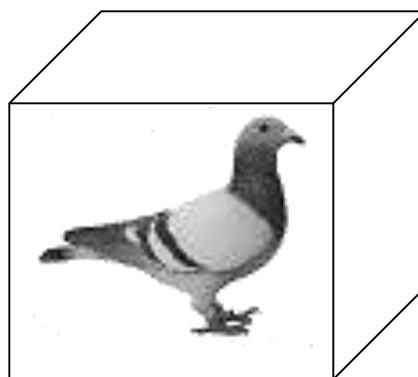
4 pigeons



3 pigeonholes



A pigeonhole must
contain at least two pigeons



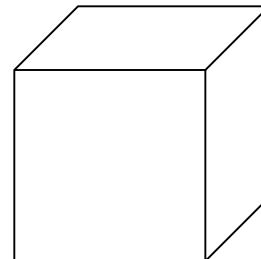
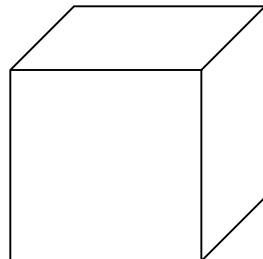
n pigeons



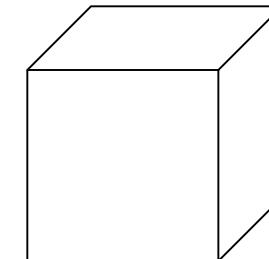
.....



m pigeonholes



.....



$n > m$

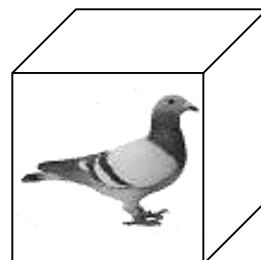
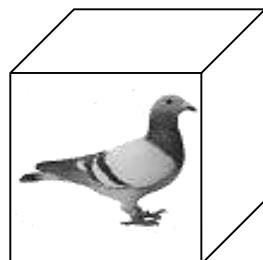
The Pigeonhole Principle

n pigeons

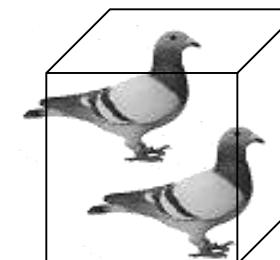
m pigeonholes

$$n > m$$

There is a pigeonhole
with at least 2 pigeons



.....

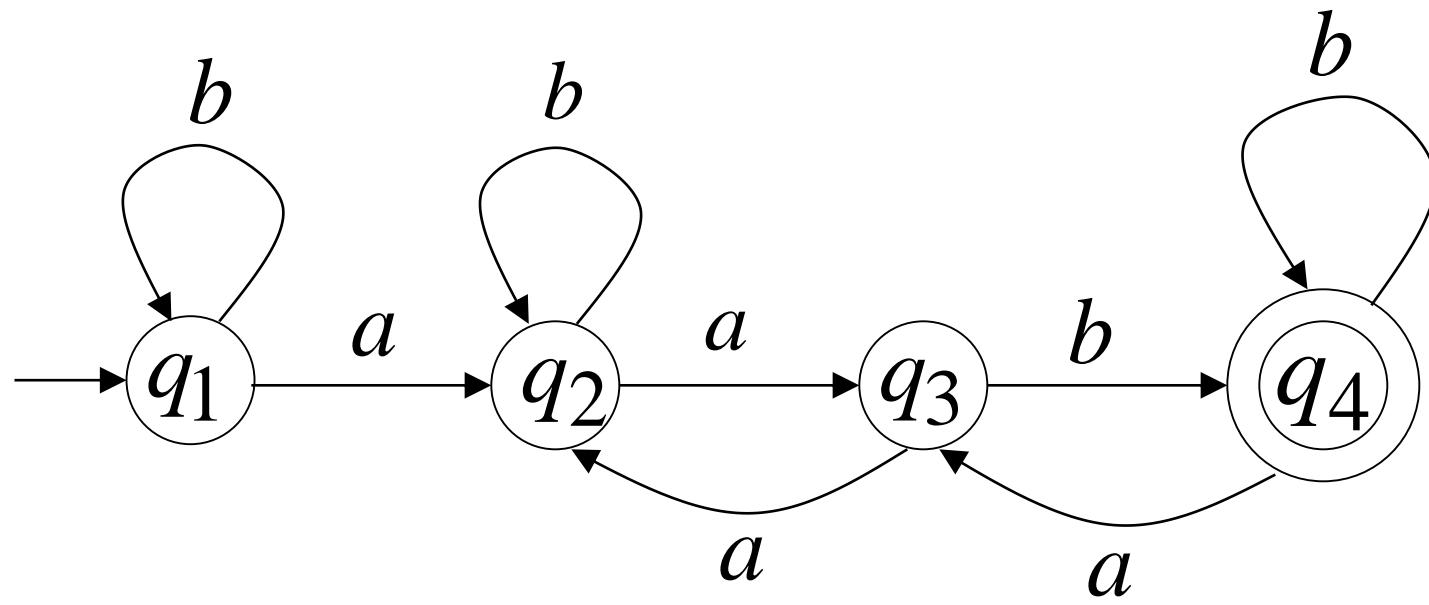


The Pigeonhole Principle

and

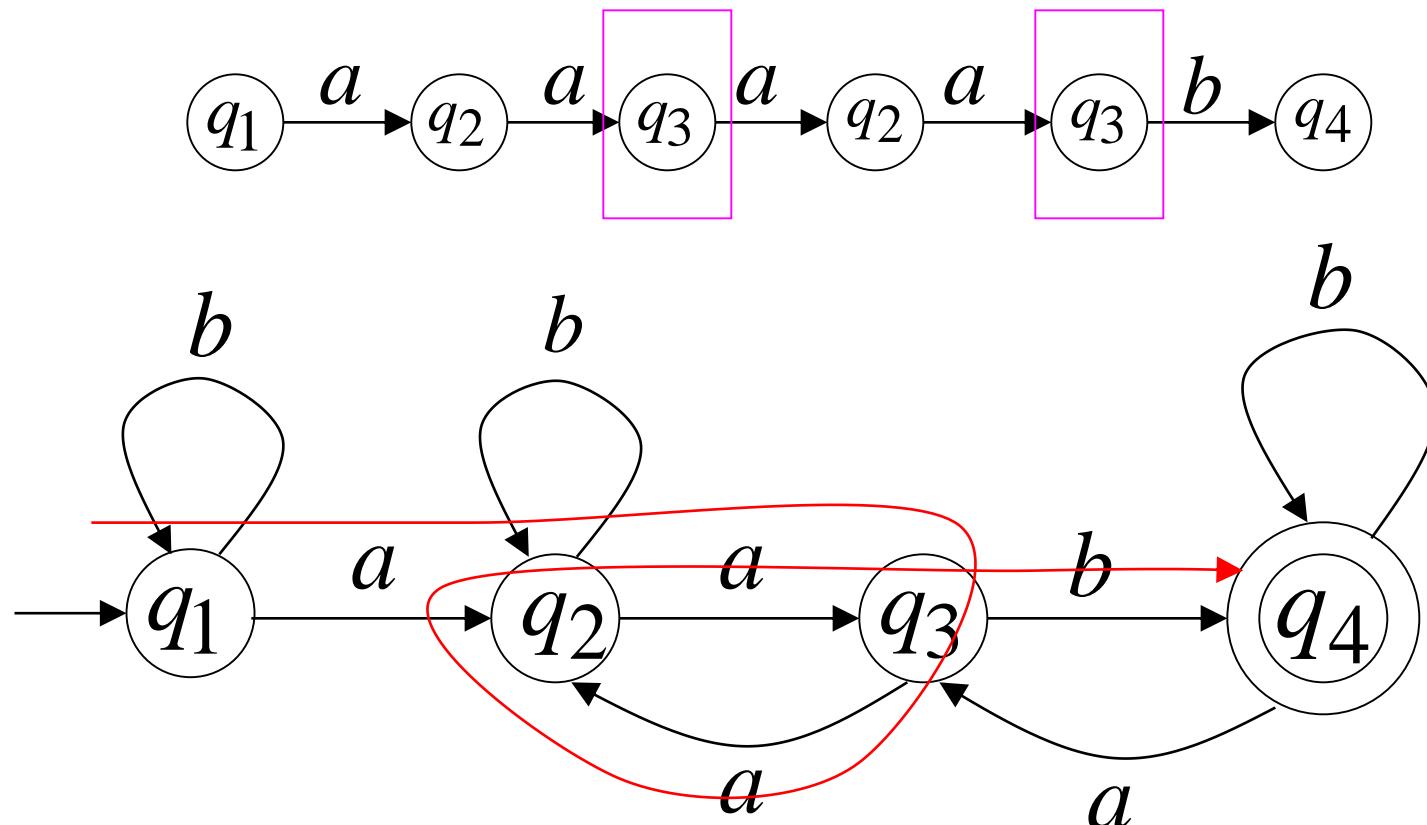
DFAs

Consider a DFA with 4 states



Consider the walk of a “long” string: $aaaab$
(length at least 4)

A state is repeated in the walk of $aaaab$



The state is repeated as a result of the pigeonhole principle

N states
 $N+1$: more
1 state > than

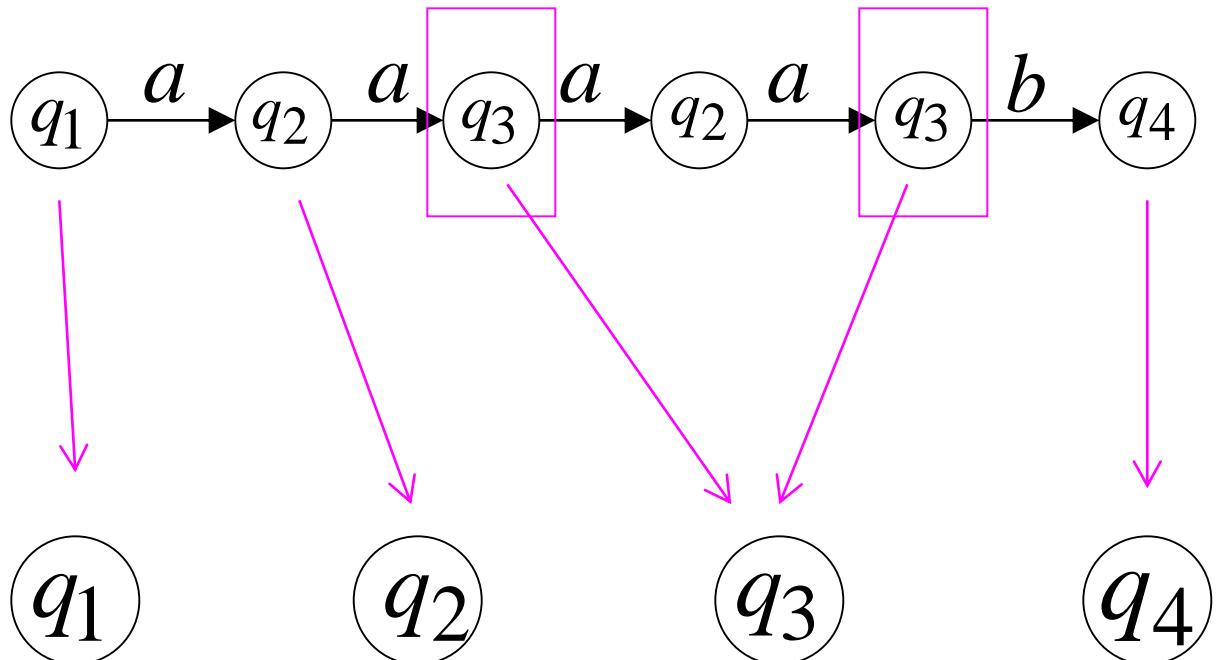
Walk of $^n adaab$

Pigeons:
(walk states)

Are more than

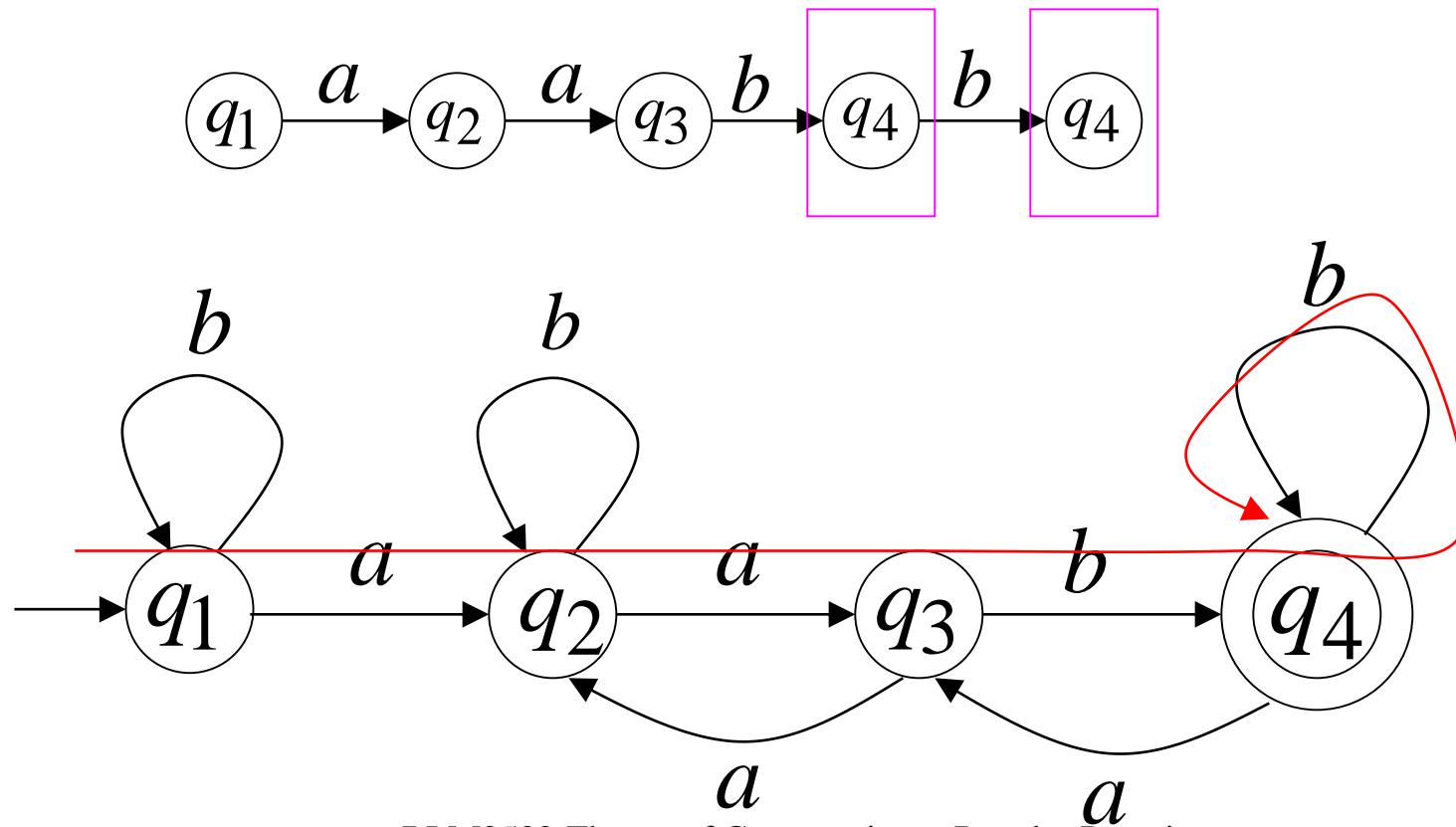
Nests:
(Automaton states)

Repeated state



Consider the walk of a “long” string: $aabb$
(length at least 4)

Due to the pigeonhole principle:
A state is repeated in the walk of $aabb$

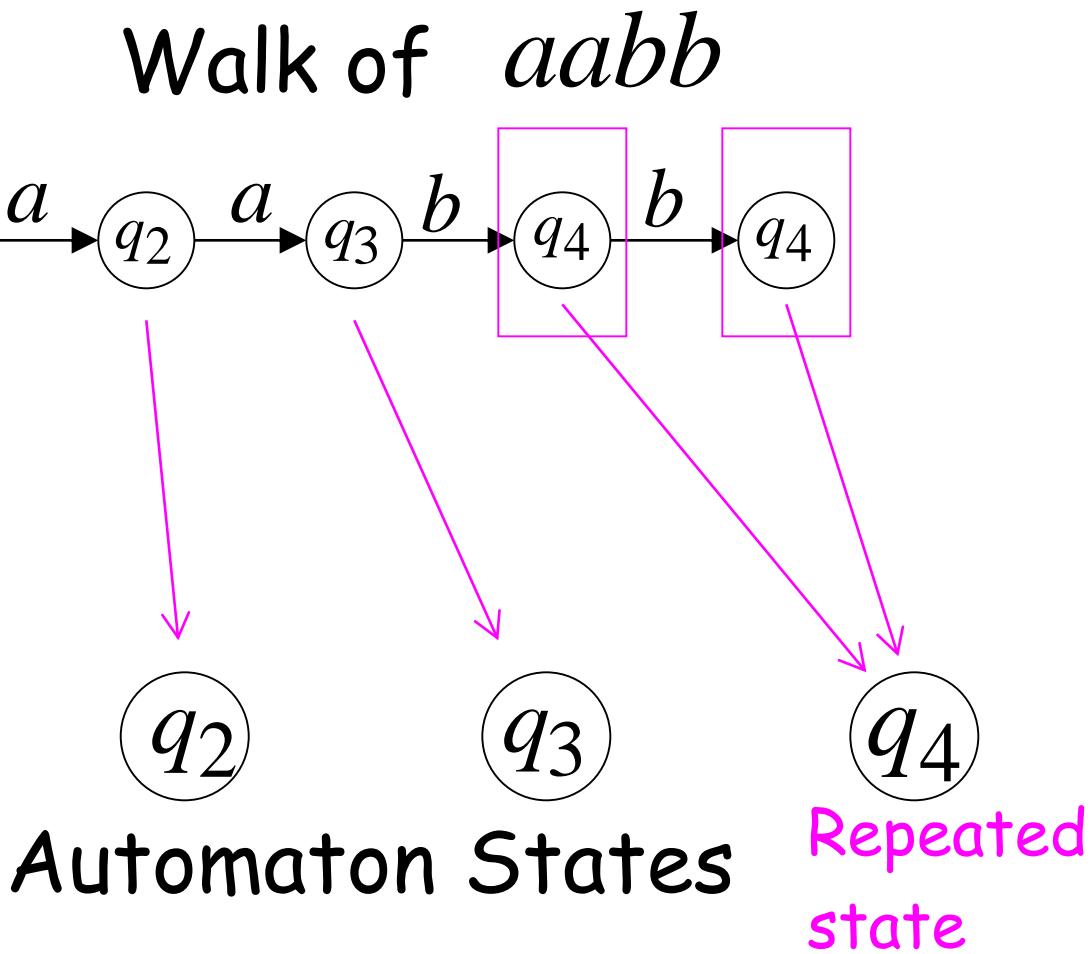


The state is repeated as a result of the pigeonhole principle

Pigeons:
(walk states)

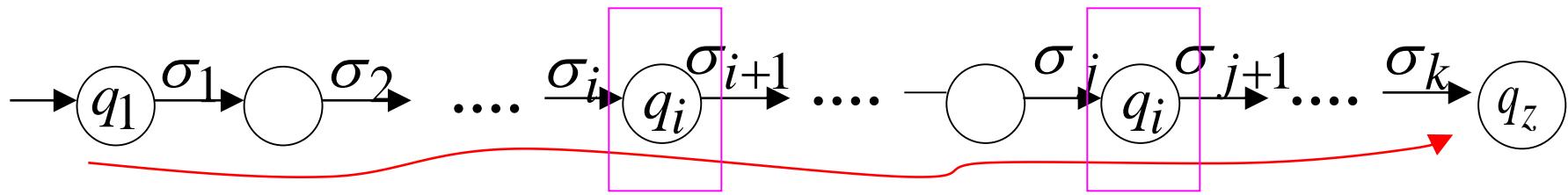
Are more than

Nests:
(Automaton states)

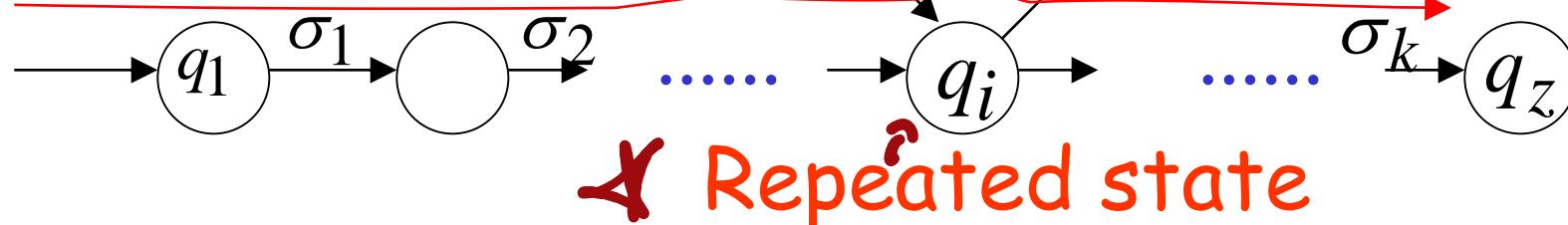


In General: If $|w| \geq \#\text{states of DFA}$,
 by the pigeonhole principle,
 a state is repeated in the walk w

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

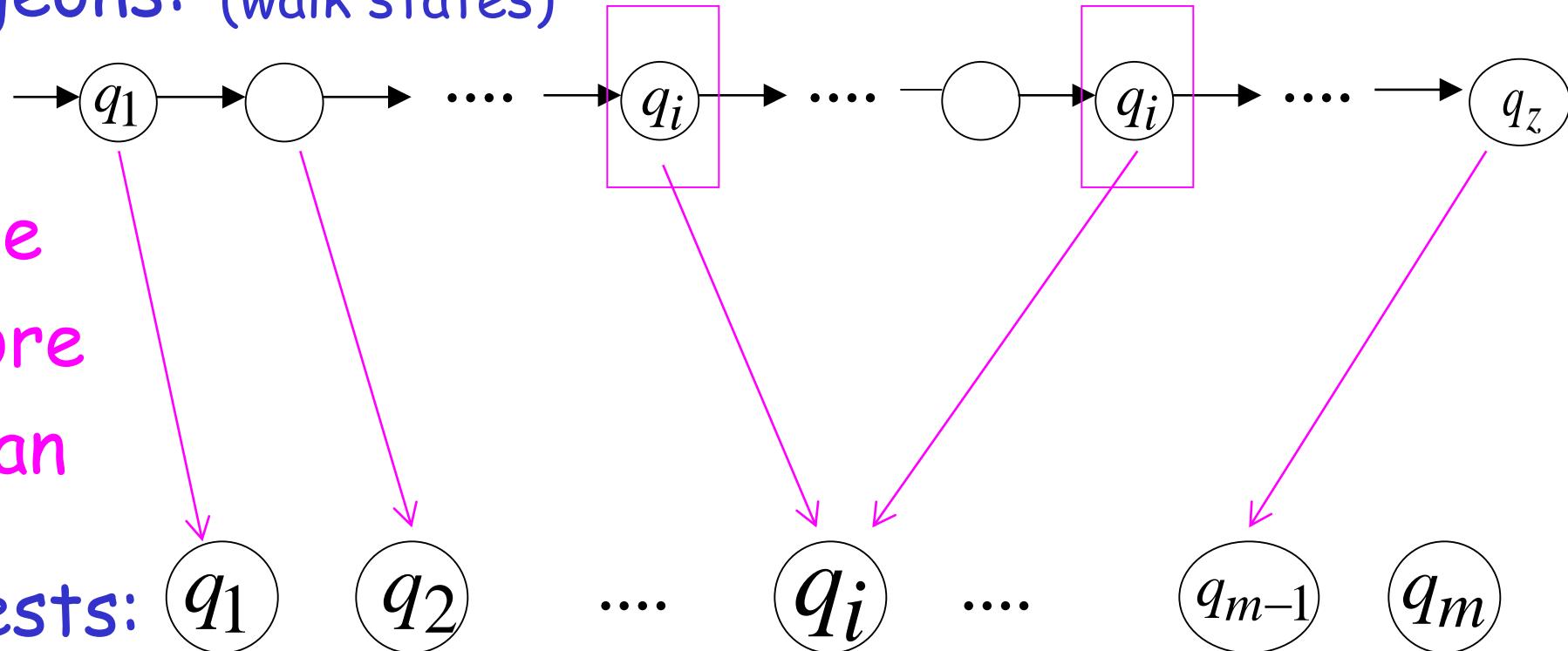


Arbitrary DFA



$$|w| \geq \#\text{states of DFA} = m$$

Pigeons: (walk states)



Walk of w

Are
more
than

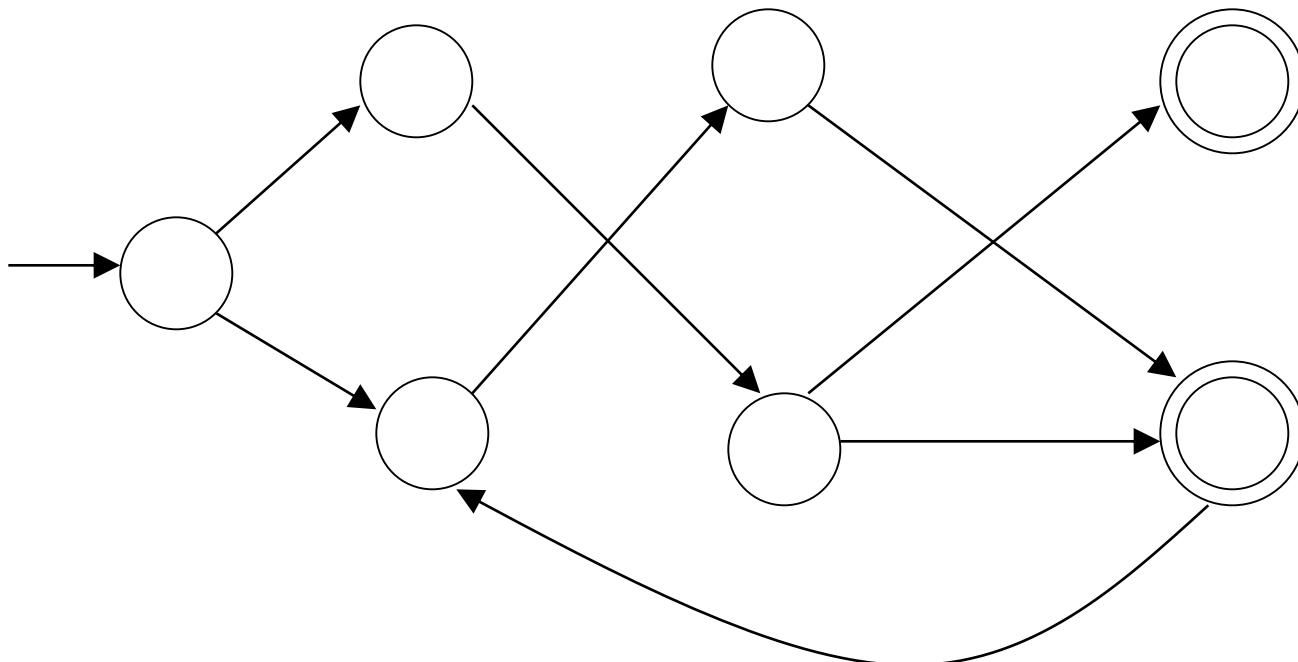
Nests: $q_1, q_2, \dots, q_i, \dots, q_{m-1}, q_m$
(Automaton states)

A state is
repeated

The Pumping Lemma

Take an **infinite** regular language L
(contains an infinite number of strings)

There exists a DFA that accepts L



m
states

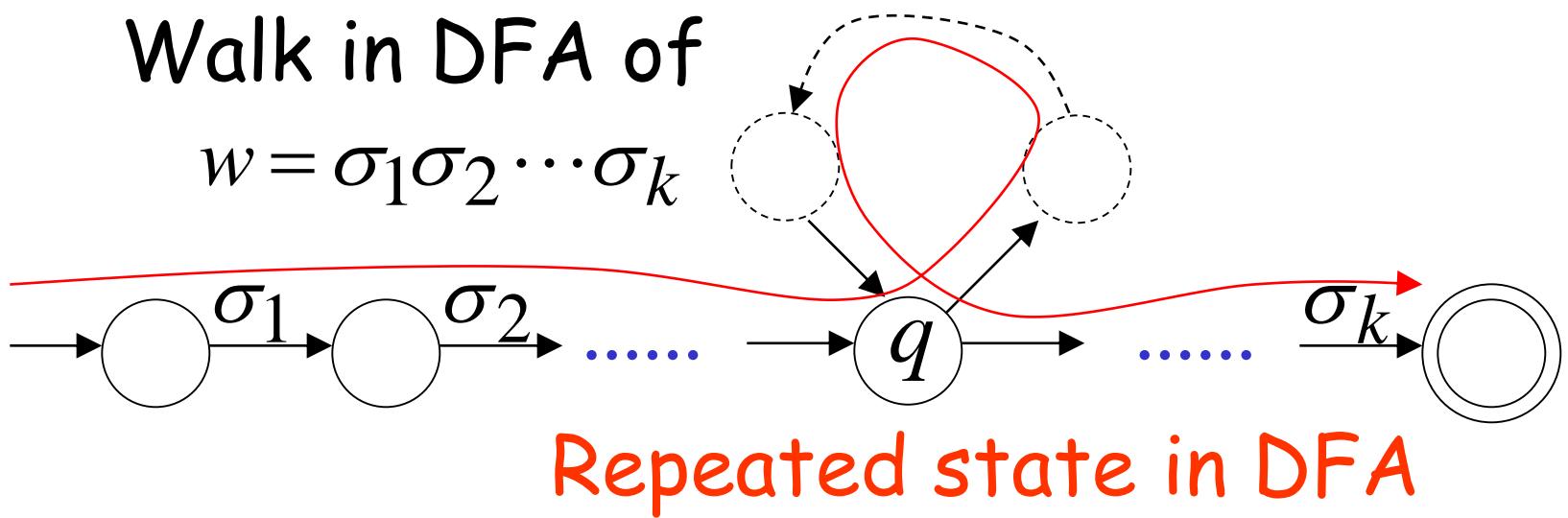
Take string $w \in L$ with $|w| \geq m$

(number of states of DFA)

then, at least one state is repeated in the walk of w

Walk in DFA of

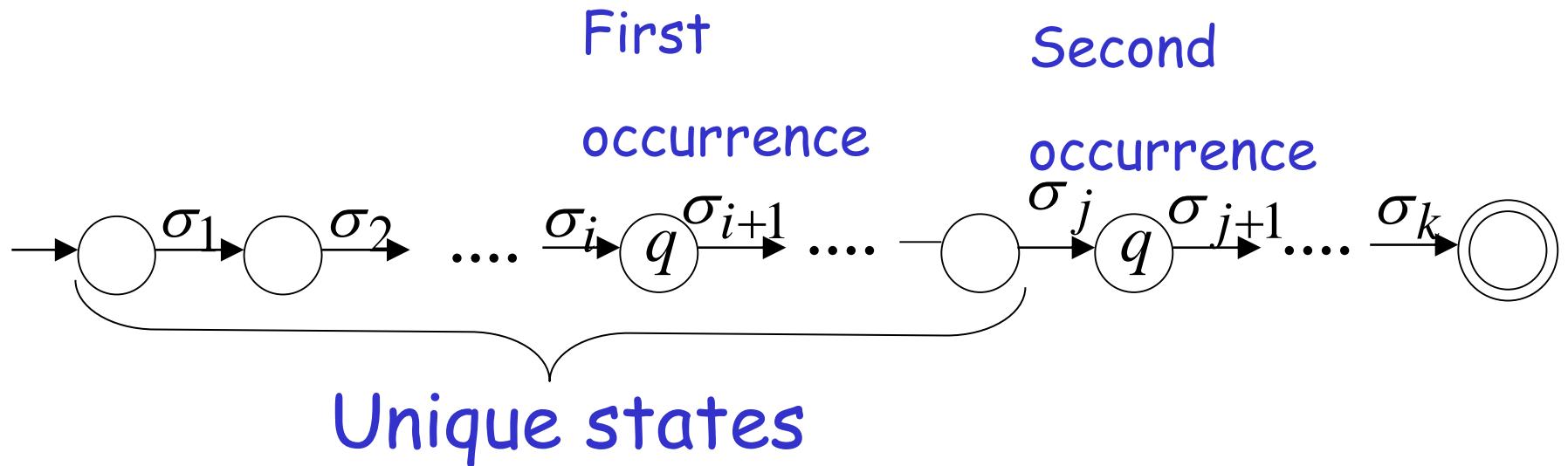
$$w = \sigma_1 \sigma_2 \cdots \sigma_k$$



There could be many states repeated

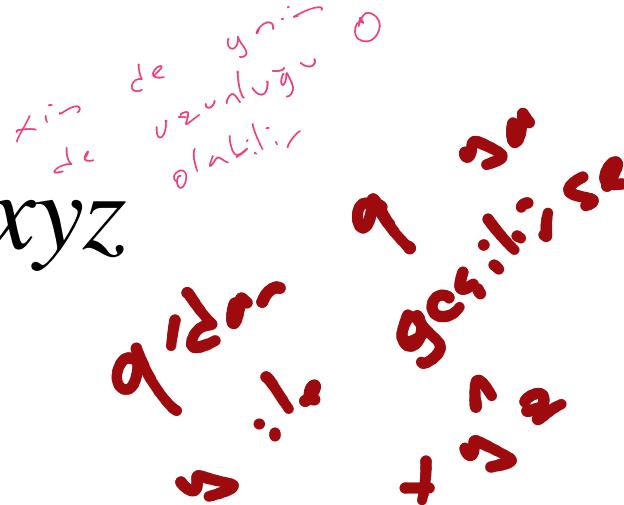
Take q to be the first state repeated

One dimensional projection of walk w :

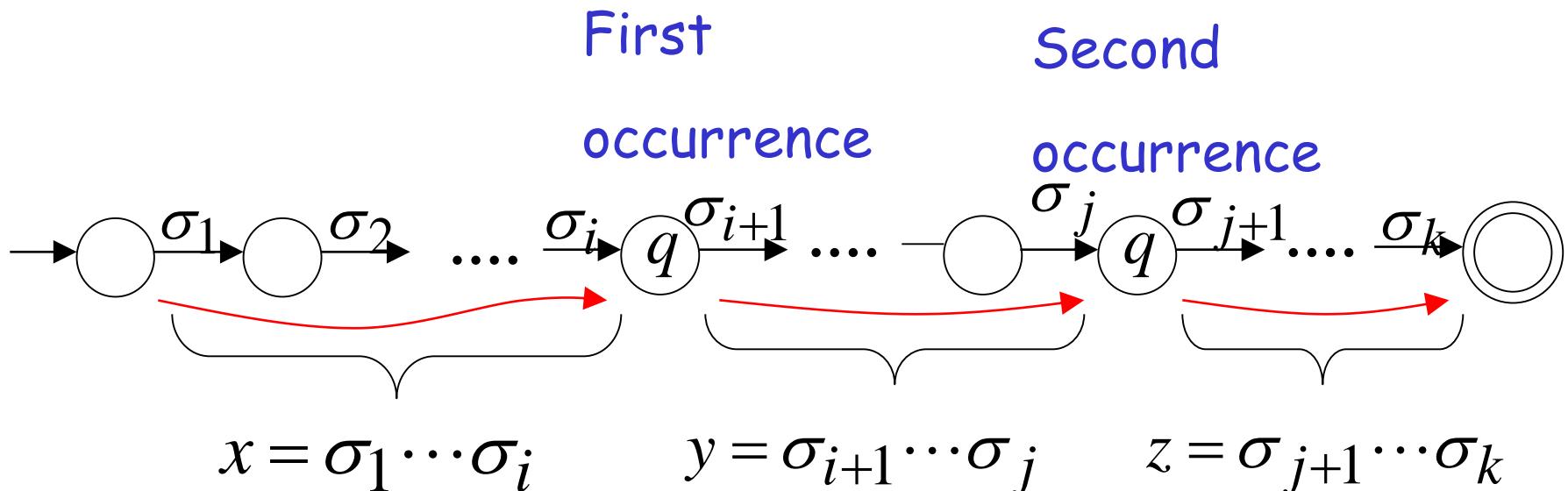


We can write

$$w = xyz$$



One dimensional projection of walk w :

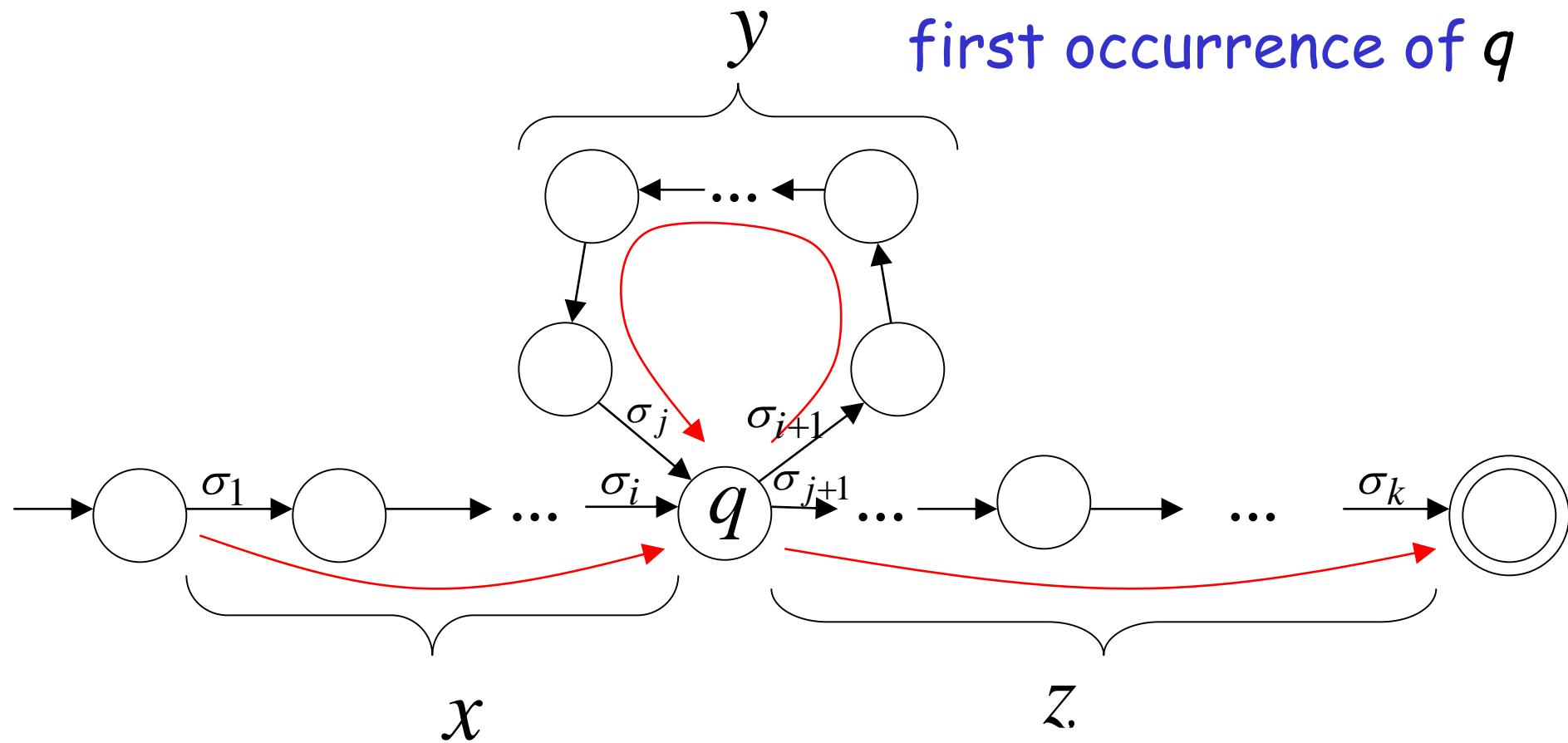


In DFA:

$$w = x \ y \ z$$

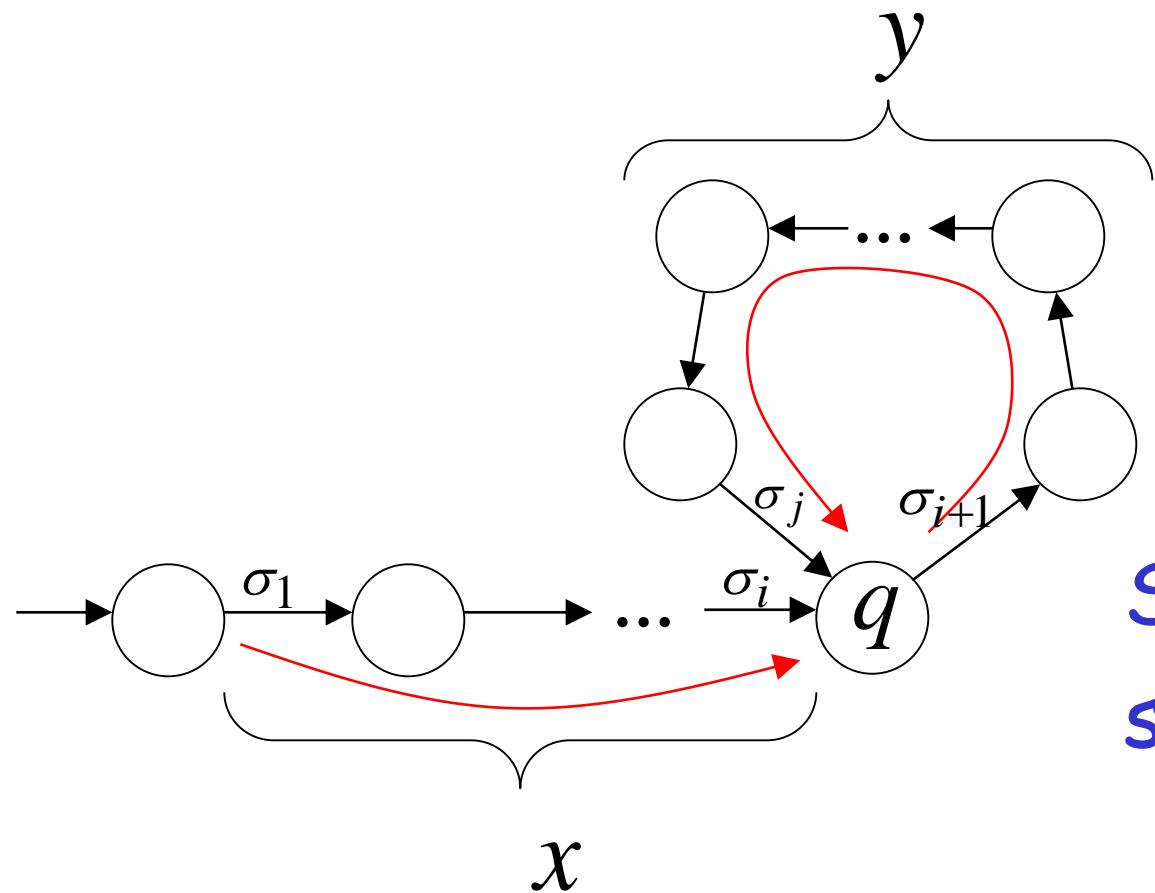
$x \circlearrowleft m$

contains only
first occurrence of q



Observation:

length $|x y| \leq m$ number
of states
of DFA



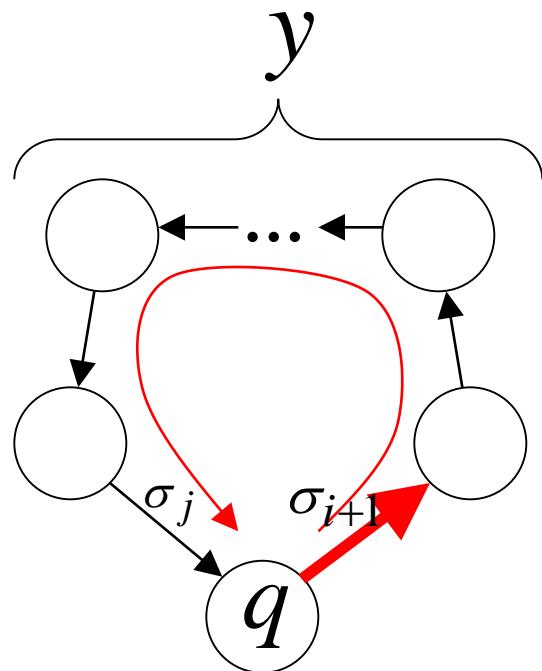
Unique States

Since, in xy no
state is repeated
(except q)

Observation:

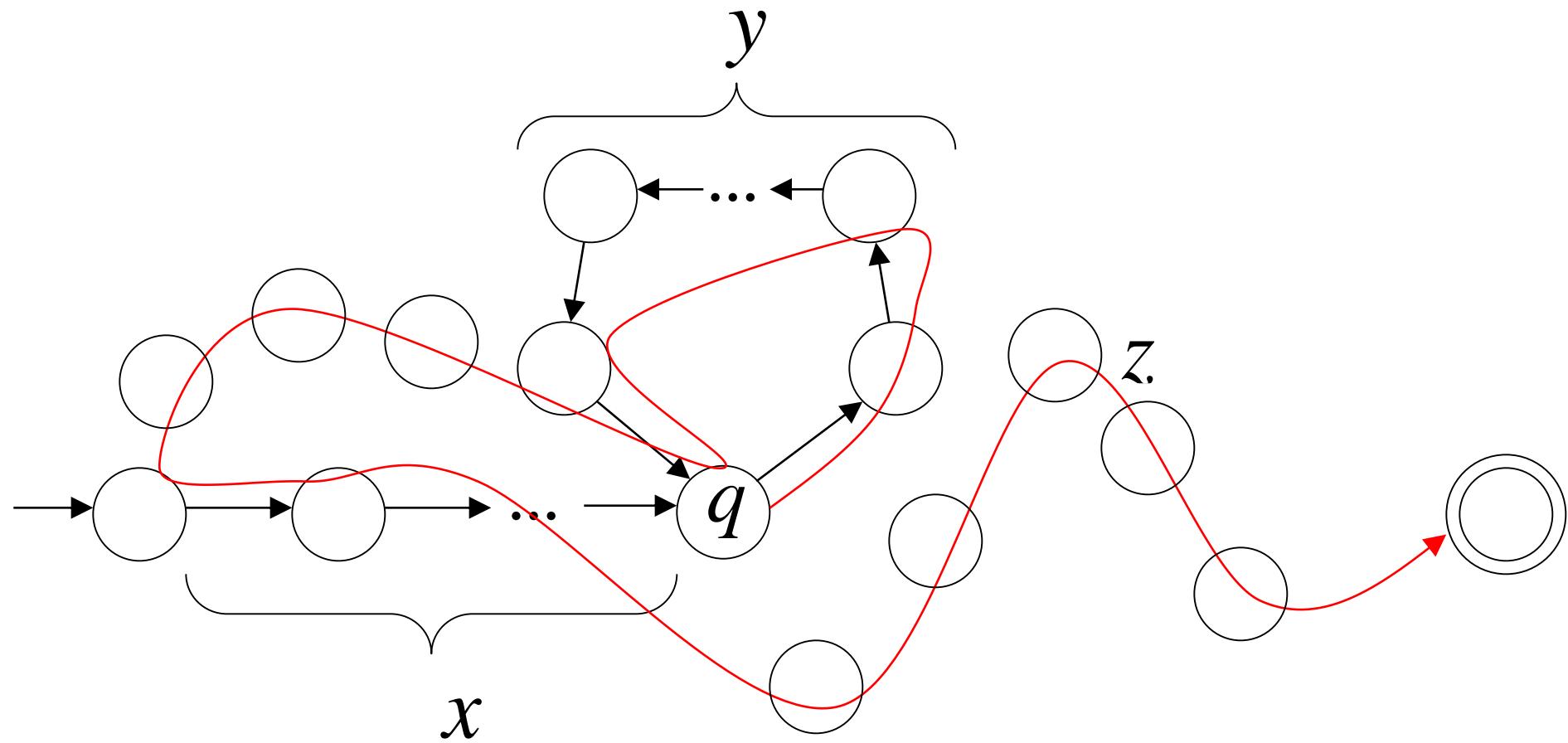
$$\text{length } |y| \geq 1$$

Since there is at least one transition in loop



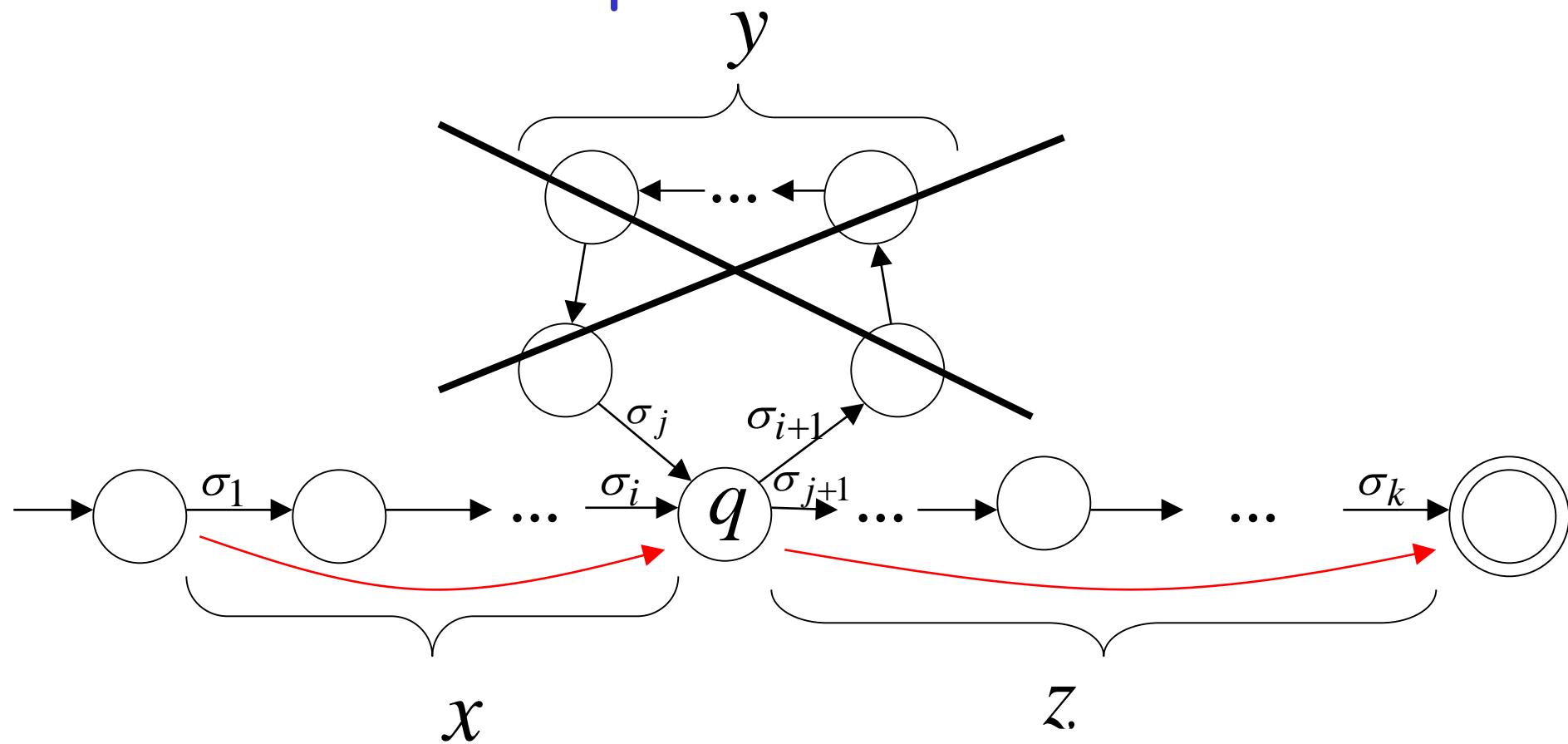
We do not care about the form of string z

z may actually overlap with the paths of x and y



Additional string: The string $x z$
is accepted

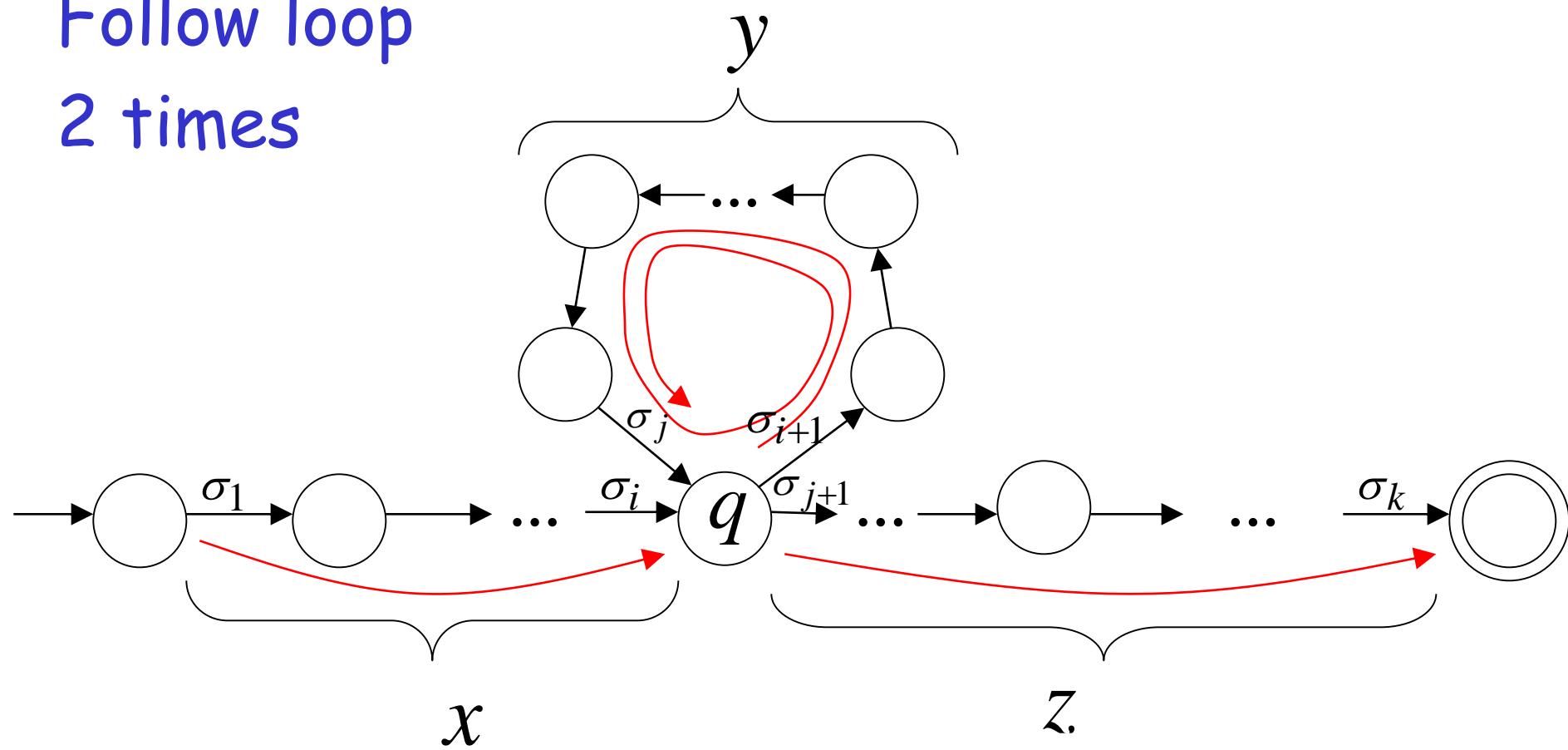
Do not follow loop



Additional string:

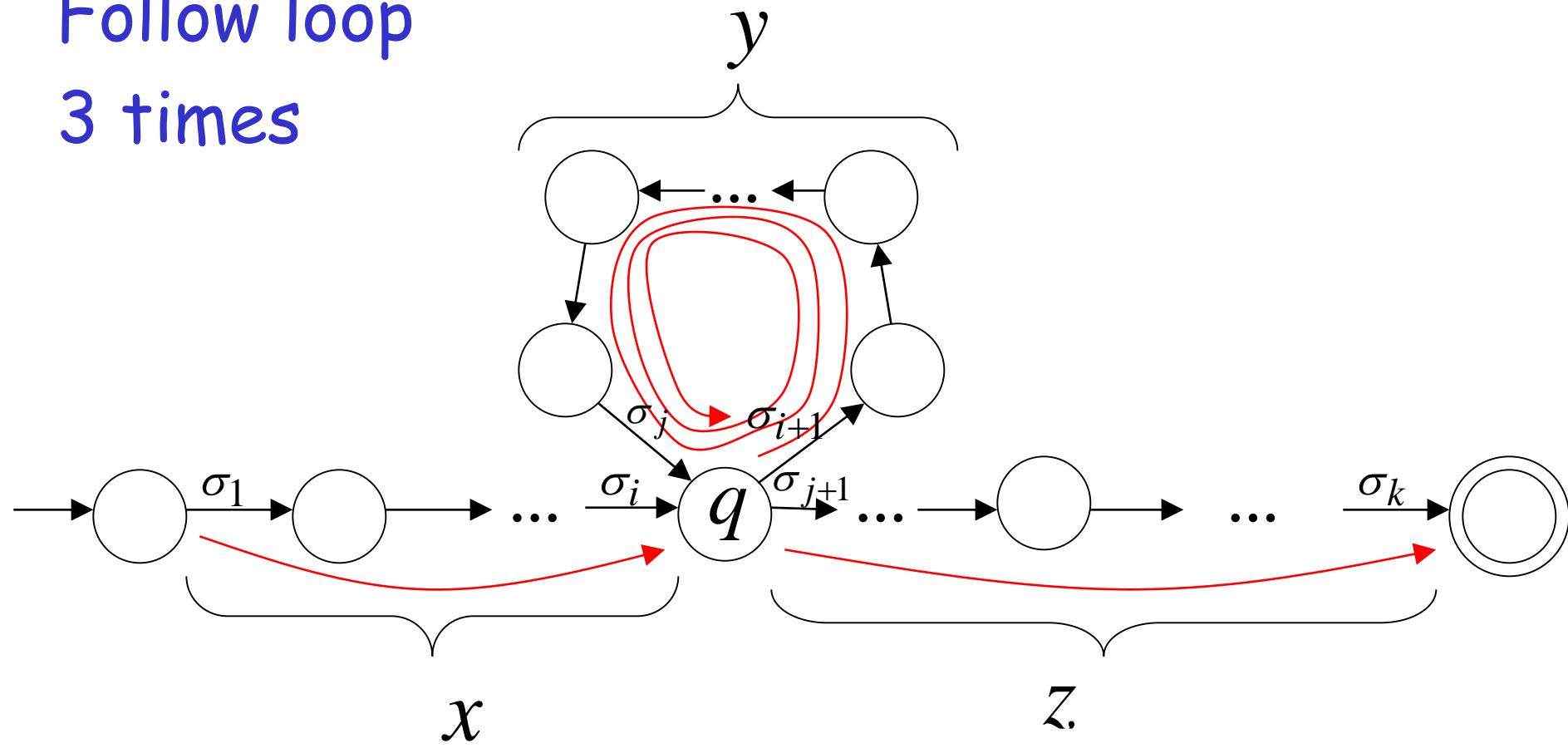
The string $x y y z$
is accepted

Follow loop
2 times



Additional string: The string $x y y y z$
is accepted

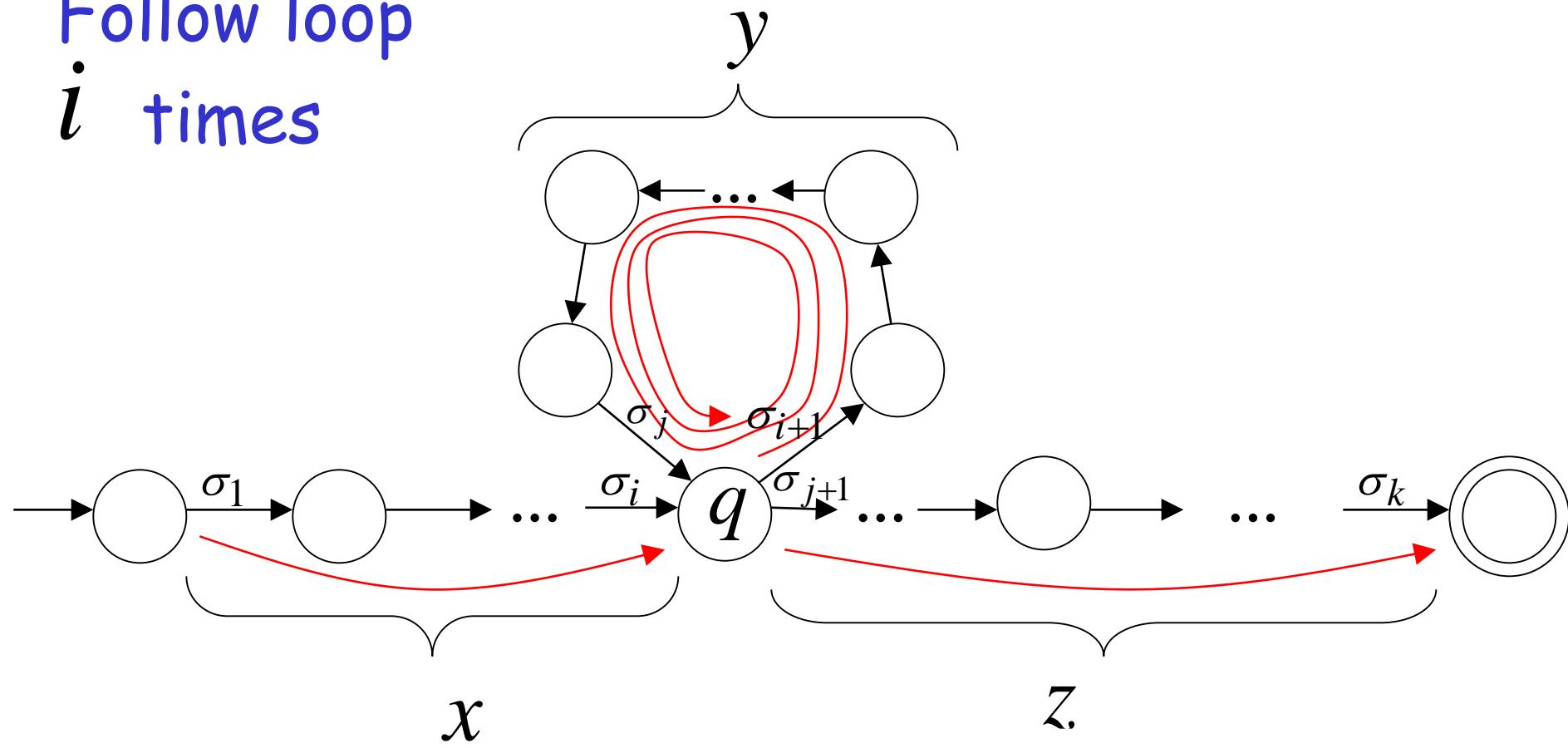
Follow loop
3 times



In General:

The string
is accepted $x \ y^i \ z$
 $i = 0, 1, 2, \dots$

Follow loop
 i times

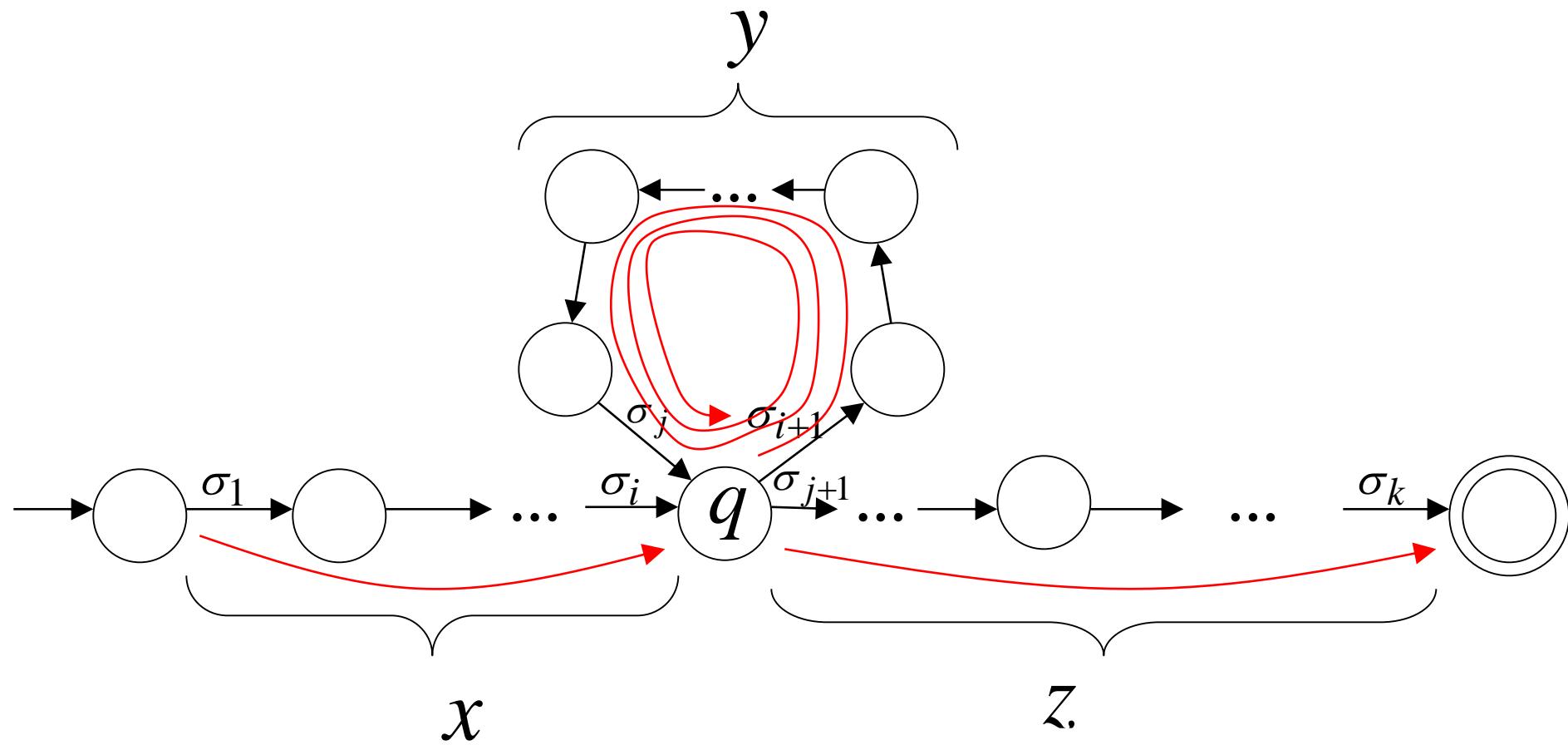


Therefore:

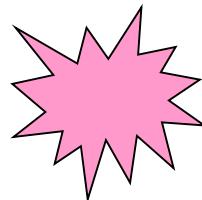
$$x \ y^i \ z \in L$$

$$i = 0, 1, 2, \dots$$

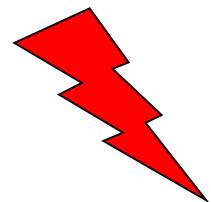
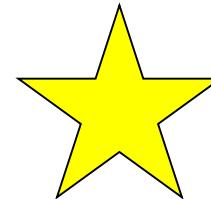
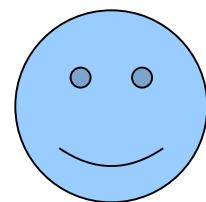
Language accepted by the DFA



In other words, we described:



The Pumping Lemma !!!



The Pumping Lemma:

- Given a infinite regular language L
- there exists an integer m (critical length)
- for any string $w \in L$ with length $|w| \geq m$
- we can write $w = x y z$
- with $|x y| \leq m$ and $|y| \geq 1$
- such that: $x y^i z \in L \quad i = 0, 1, 2, \dots$

In the book:

Critical length m = Pumping length p

Applications of the Pumping Lemma

Observation:

Every language of finite size has to be regular

(we can easily construct an NFA
that accepts every string in the language)

→ n finite
size languages.

Therefore, every non-regular language
has to be of infinite size

(contains an infinite number of strings)

Suppose you want to prove that
An infinite language L is not regular

1. Assume the opposite: L is regular
2. The pumping lemma should hold for L
3. Use the pumping lemma to obtain a contradiction
 \hookrightarrow Chelishki
4. Therefore, L is not regular

Explanation of Step 3: How to get a contradiction

1. Let m be the critical length for L
2. Choose a particular string $w \in L$ which satisfies the length condition $|w| \geq m$
3. Write $w = xyz$
not disjoint, normal, begins like
4. Show that $w' = xy^i z \notin L$ for some $i \neq 1$
5. This gives a contradiction, since from pumping lemma $w' = xy^i z \in L$

Note: It suffices to show that
only one string $w \in L$
gives a contradiction

You don't need to obtain
contradiction for every $w \in L$

Example of Pumping Lemma application

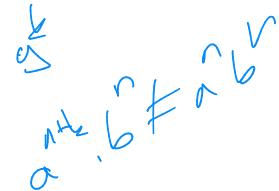
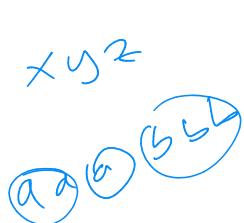
Theorem: The language $L = \{a^n b^n : n \geq 0\}$ is not regular

$$\begin{array}{c} a^m b^m \\ \vdots \\ xy^2 = a a b b \\ y = a \end{array}$$

$$x^{y^4} z^2 \\ a^{m+k} b^m \neq a^{\sim} b^{\sim}$$

Proof: Use the Pumping Lemma

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



$a a \underbrace{b}_2 L$

$x \rightarrow a$
 $y \rightarrow a$

Assume for contradiction
that L is a regular language

a
 $xy^i z \notin a^n L^n$

$a a a b b \notin a^7 b^5$

Since L is infinite
we can apply the Pumping Lemma

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

Let m be the critical length for L

Pick a string w such that: $w \in L$

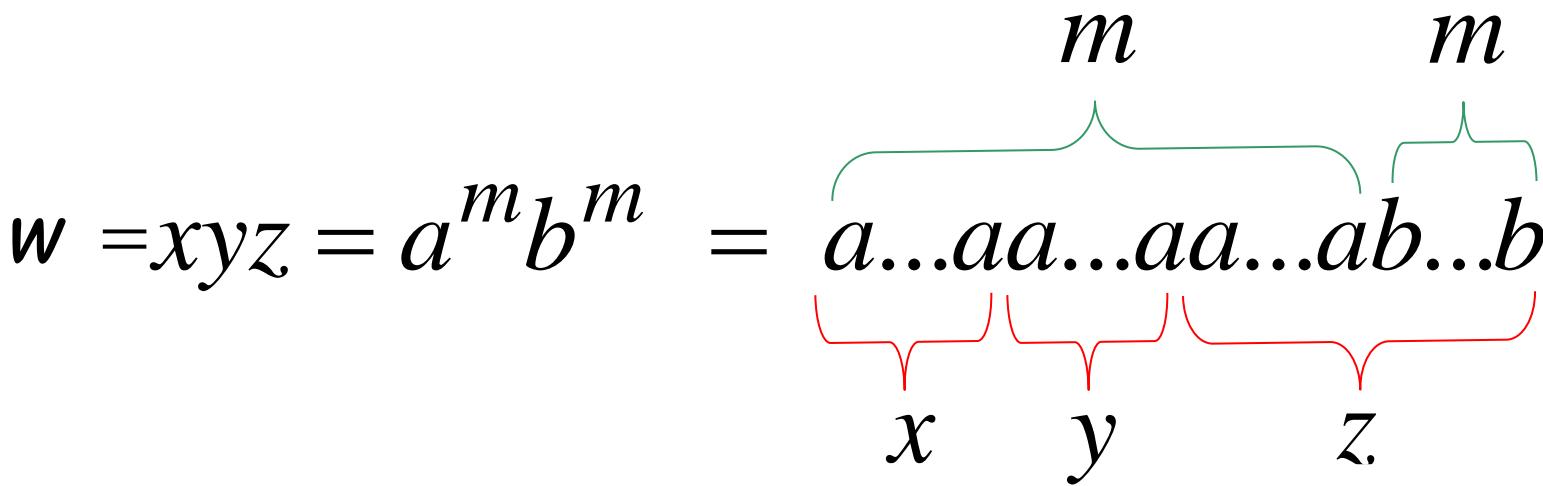
and length $|w| \geq m$

We pick $w = a^m b^m$

From the Pumping Lemma:

we can write $w = a^m b^m = x y z$

with lengths $|x y| \leq m$, $|y| \geq 1$



Thus: $y = a^k$, $1 \leq k \leq m$

$$x \ y \ z = a^m b^m \quad y = a^k, \quad 1 \leq k \leq m$$

From the Pumping Lemma: $x \ y^i \ z \in L$

$i = 0, 1, 2, \dots$

Thus: $x \ y^2 \ z \in L$

$$x \ y \ z = a^m b^m \quad y = a^k, \quad 1 \leq k \leq m$$

From the Pumping Lemma: $x \ y^2 \ z \in L$

$$xy^2z = \underbrace{a \dots a}_{x} \underbrace{a \dots a}_{y} \underbrace{a \dots a}_{y} \underbrace{ab \dots b}_{z} \in L$$

$m+k$ m

Thus: $a^{m+k} b^m \in L$

$$a^{m+k}b^m \in L \quad k \geq 1$$

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

END OF PROOF

Non-regular language $\{a^n b^n : n \geq 0\}$

Regular languages

$L(a^* b^*)$

More Applications

of

the Pumping Lemma

The Pumping Lemma:

- Given a infinite regular language L
- there exists an integer m (critical length)
- for any string $w \in L$ with length $|w| \geq m$
- we can write $w = x y z$
- with $|x y| \leq m$ and $|y| \geq 1$
- such that: $x y^i z \in L \quad i = 0, 1, 2, \dots$

Non-regular languages

$$L = \{vv^R : v \in \Sigma^*\}$$

$a^m b^m b^m a^m$

Regular languages

$x y z = a b b a a$

$y = a$

$x y z = a^m b^m a^k b^m a^m$

Theorem: The language

$$L = \{vv^R : v \in \Sigma^*\} \quad \Sigma = \{a,b\}$$

is not regular

Proof: Use the Pumping Lemma

$$L = \{vv^R : v \in \Sigma^*\}$$

Assume for contradiction
that L is a regular language

Since L is infinite
we can apply the Pumping Lemma

$$L = \{vv^R : v \in \Sigma^*\}$$

Let m be the critical length for L

Pick a string w such that: $w \in L$

$a^n a^m$
 $\underbrace{a}_x \underbrace{a}_y \underbrace{a}_z^m$

$a^{m+k} a^m \neq a^{2n}$
while k odd

and length $|w| \geq m$

We pick $w = a^m b^m b^m a^m$

From the Pumping Lemma:

we can write: $w = a^m b^m b^m a^m = x y z$

with lengths: $|x y| \leq m$, $|y| \geq 1$

$w = xyz = a \dots aa \dots a \dots ab \dots bb \dots ba \dots a$

m m m m
 x y $z.$

Thus: $y = a^k$, $1 \leq k \leq m$

$$x \ y \ z = a^m b^m b^m a^m \quad y = a^k, \quad 1 \leq k \leq m$$

From the Pumping Lemma: $x \ y^i \ z \in L$
 $i = 0, 1, 2, \dots$

Thus: $x \ y^2 \ z \in L$

$$x y z = a^m b^m b^m a^m \quad y = a^k, \quad 1 \leq k \leq m$$

From the Pumping Lemma: $x y^2 z \in L$

$$xy^2z = \underbrace{a \dots a}_{x} \underbrace{\dots a}_{y} \underbrace{\dots a}_{y} \underbrace{\dots ab \dots bb \dots ba \dots a}_{z} \underbrace{\dots}_{m+k} \underbrace{\dots}_{m} \underbrace{\dots}_{m} \underbrace{\dots}_{m} \in L$$

Thus: $a^{m+k} b^m b^m a^m \in L$

$$a^{m+k}b^mb^ma^m \in L \quad k \geq 1$$

BUT: $L = \{vv^R : v \in \Sigma^*\}$



$$a^{m+k}b^mb^ma^m \notin L$$

CONTRADICTION!!!

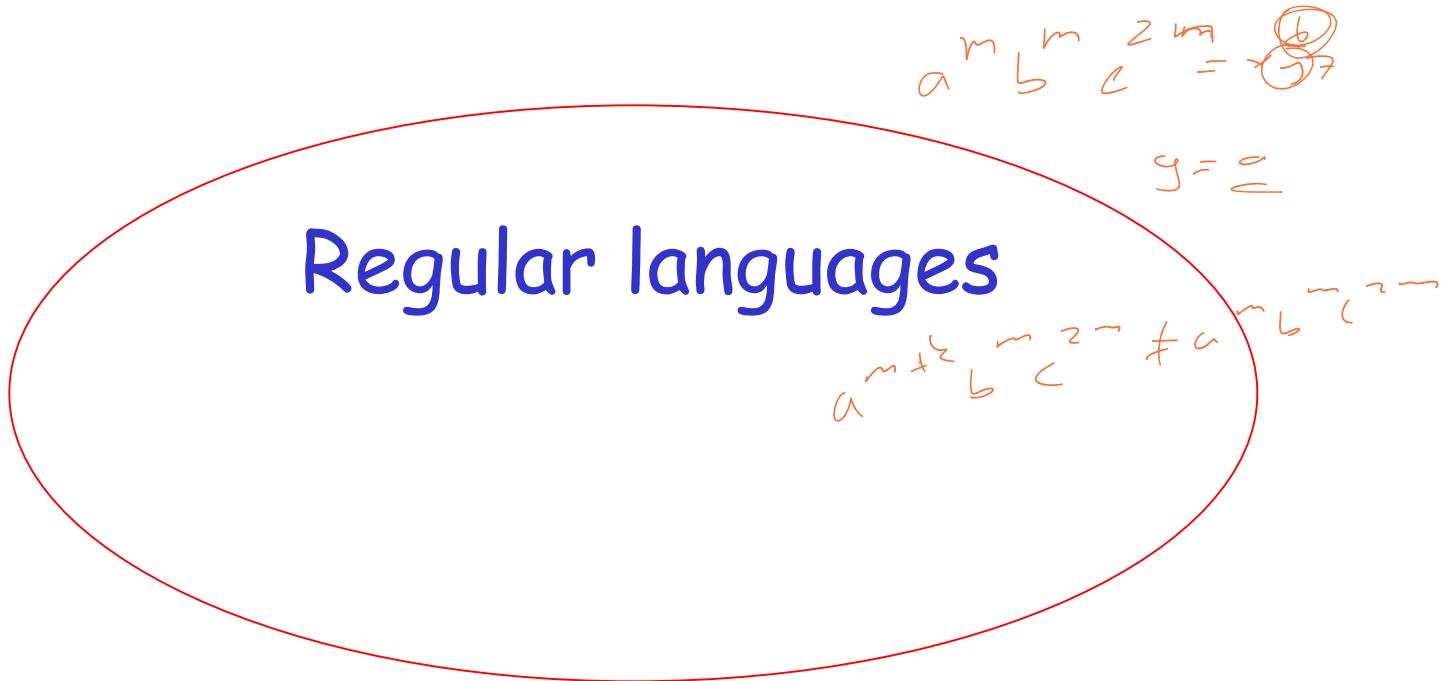
Therefore: Our assumption that L is a regular language is not true

Conclusion: L is not a regular language

END OF PROOF

Non-regular languages

$$L = \{a^n b^l c^{n+l} : n, l \geq 0\}$$





Theorem: The language

$$L = \{a^n b^l c^{n+l} : n, l \geq 0\}$$

Handwritten notes in yellow:
"language"
"or many other words"
"but not all"
"yup"
"z>1"

is not regular

Proof:

Use the Pumping Lemma

$$L = \{a^n b^l c^{n+l} : n, l \geq 0\}$$

Assume for contradiction
that L is a regular language

Since L is infinite
we can apply the Pumping Lemma

$$L = \{a^n b^l c^{n+l} : n, l \geq 0\}$$

$(xy)^2$

Let m be the critical length of L

$\leq m$

$a^m b^m c^{2m}$
 $y = a$
 x^2
 $a^m b^m c^{2m} x^2$

Pick a string w such that: $w \in L$ and

length $|w| \geq m$

We pick $w = a^m b^m c^{2m}$

From the Pumping Lemma:

$\leq m$

We can write $w = \underbrace{a^m}_{\text{length } \leq m} b^m c^{2m} = \underbrace{x y z}$

With lengths $|x y| \leq m, |y| \geq 1$

$a^m b^m c^{2m}$

$w = xyz = \overbrace{a \dots aa \dots aa \dots ab \dots bc \dots cc \dots c}^{2m} = \underbrace{x}_{m \text{ length}} \underbrace{y}_{m \text{ length}} \underbrace{z}_{2m \text{ length}}$

Thus: $y = a^k, 1 \leq k \leq m$

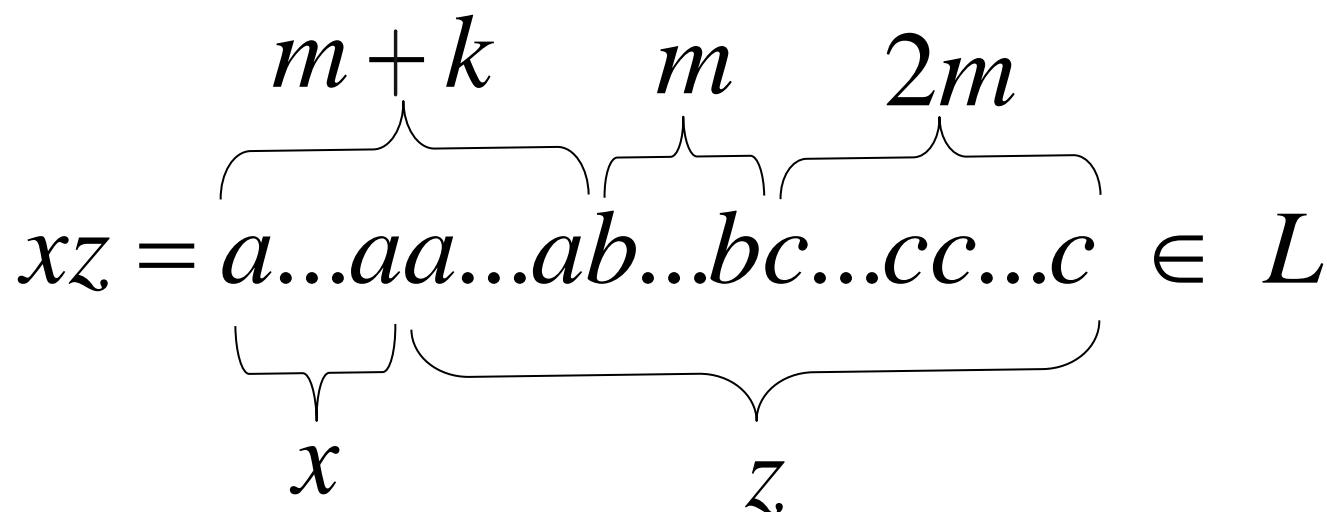
$$x \ y \ z = a^m b^m c^{2m} \quad y = a^k, \quad 1 \leq k \leq m$$

From the Pumping Lemma: $x \ y^i \ z \in L$
 $i = 0, 1, 2, \dots$

Thus: $x \ y^0 \ z = xz \in L$

$$x \ y \ z = a^m b^m c^{2m} \quad y = a^k, \quad 1 \leq k \leq m$$

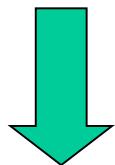
From the Pumping Lemma: $xz \in L$



Thus: $a^{m+k} b^m c^{2m} \in L$

$$a^{m+k} b^m c^{2m} \in L \quad k \geq 1$$

BUT: $L = \{a^n b^l c^{n+l} : n, l \geq 0\}$



$$a^{m+k} b^m c^{2m} \notin L$$

CONTRADICTION!!!

Therefore: Our assumption that L is a regular language is not true

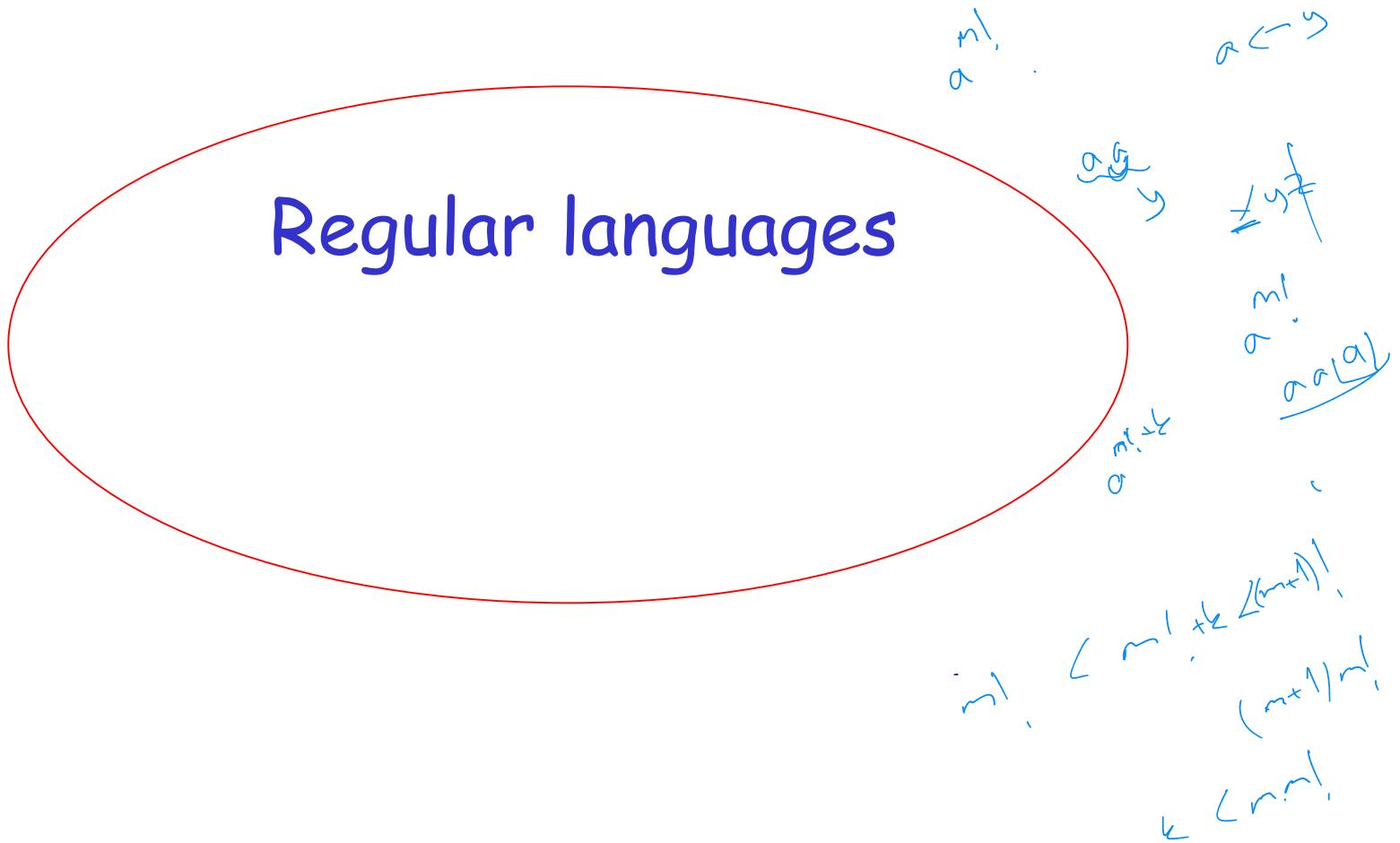
Conclusion: L is not a regular language

END OF PROOF

Non-regular languages

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

Regular languages



Theorem: The language $L = \{a^{n!} : n \geq 0\}$
is not regular

$$n! = 1 \cdot 2 \cdots (n-1) \cdot n$$

Proof: Use the Pumping Lemma

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

Assume for contradiction
that L is a regular language

Since L is infinite
we can apply the Pumping Lemma

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

Let m be the critical length of L

Pick a string w such that: $w \in L$

length $|w| \geq m$

We pick $w = a^{m!}$

From the Pumping Lemma:

We can write $w = a^{m!} = x y z$

With lengths $|x y| \leq m$, $|y| \geq 1$

$w = xyz = a^{m!} = \overbrace{a \dots a}^m \dots \overbrace{a \dots a}^{m!-m} \dots \overbrace{a \dots a}^m$

m $m!-m$

x y z

Thus: $y = a^k$, $1 \leq k \leq m$

$$x \ y \ z = a^{m!} \quad y = a^k, \quad 1 \leq k \leq m$$

From the Pumping Lemma: $x \ y^i \ z \in L$
 $i = 0, 1, 2, \dots$

Thus: $x \ y^2 \ z \in L$

$$x \ y \ z = a^{m!}$$

$$y = a^k, \quad 1 \leq k \leq m$$

From the Pumping Lemma: $x \ y^2 \ z \in L$

$$xy^2z = \overbrace{a \dots aa \dots aa \dots aa \dots aa \dots aa \dots a}^{m+k} \in L$$

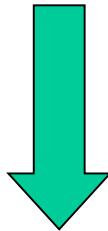
$m + k$
 $m! - m$

$x \quad y \quad y \quad z.$

Thus: $a^{m!+k} \in L$

$$a^{m!+k} \in L \quad 1 \leq k \leq m$$

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



There must exist p such that:

$$m!+k = p!$$

However: $m!+k \leq m!+m$ for $m > 1$

$$\leq m!+m!$$

$$< m!m + m!$$

$$= m!(m+1)$$

$$= (m+1)!$$



$$m!+k < (m+1)!$$



$$m!+k \neq p! \quad \text{for any } p$$

$$a^{m!+k} \in L \quad 1 \leq k \leq m$$

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



$$a^{m!+k} \notin L$$

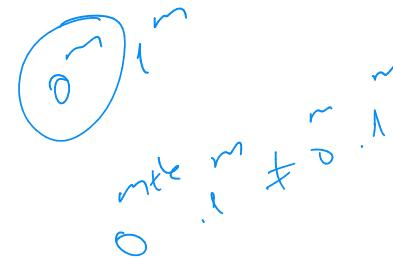
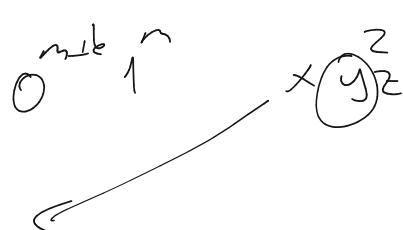
CONTRADICTION!!!

Therefore: Our assumption that L is a regular language is not true

Conclusion: L is not a regular language

END OF PROOF

Example 2



Theorem: $B = \{0^n 1^n \mid n \geq 0\}$ is not regular.

Proof. Assume the contrary that B is regular.

Let p be the pumping length. Then, $s = 0^p 1^p$ can be decomposed as $s = xyz$ with $|y| \geq 1$, and $xy^n z \in B$ for any $n \geq 0$.

There can be three cases $y = 0^k$, $y = 0^k 1^l$, and $y = 1^l$, for some nonzero k, l . In each case, we can easily see that $xy^n z \notin B$, which leads to contradiction. ($n \geq 2$)

Example 3



Theorem: $C = \{w \mid w \text{ has an equal number of 0s and 1s}\}$
is not regular.

Proof. Assume the contrary that C is regular.

Let p be the pumping length. Then, $s = 0^p 1^p$ can be decomposed as $s = xyz$ with $|y| \geq 1$ and $|xy| \leq p$, and $xy^n z \in C$ for any $n \geq 0$.

Then we must have $y = 0^k$ for some nonzero k .

We can easily see that $xy^n z \notin C$, which contradicts the assumption.

Alternative proof of Example 3

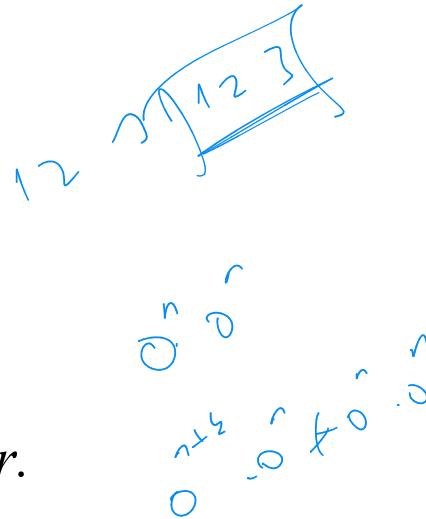
- The class of regular languages is closed under the intersection operation. This is easy to prove if we run two DFAs parallelly and accept only string which are accepted by both of the DFAs.
- Now, assume C is regular.
- Then, $C \cap 0^*1^* = B$ is also regular.
- This contradicts what we proved in Example 1.38.???

Example 4

$a^m 0^n 1^k$

$a^m a^m$

$a^{m+k} a^n \neq a^{m+k}$ for k odd



Theorem: $F = \{ww \mid w \in \{0,1\}^*\}$ is not regular.

Proof. Assume the contrary that F is regular.

Let p be the pumping length and let $s = \underline{0^p} \underline{1^p} \underline{1^k} \in F$. This s can be split into pieces like $s = xyz$ with $|y| \geq 1$ and $|xy| \leq p$, and $xy^n z \in F$ for any $n \geq 0$. Then we must have $y = 0^k$ for some nonzero k .

We can easily see that $xy^n z \notin F$, which contradicts the assumption.

Example 5

1^{m^2+k}

$y = \underbrace{1 \dots 1}_{n^2}$

$n^2 \times k \neq p^2$

Theorem : $D = \{1^{\circled{n^2}} \mid n \geq 0\}$ is not regular.

Proof. Assume the contrary that D is regular.

Let p be the pumping length and let $s = 1^{p^2} \in D$. This s can be split into pieces like $s = xyz$ with $|y| \geq 1$ and $|xy| \leq p$, and $xy^n z \in D$ for any $n \geq 0$.

The length of $xy^n z$ grows linearly with n , while the lengths of strings in D grows as $0, 1, 4, 9, 16, 25, 36, 49, \dots$

These two facts are incompatible as can be easily seen.

Example 6 (Pumping Down)

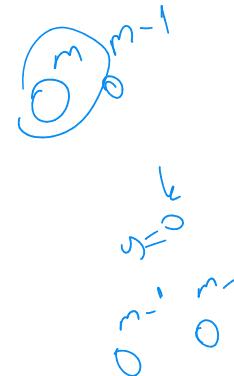
Theorem: $E = \{0^i 1^j \mid i > j\}$ is not regular.

Proof. Assume the contrary that E is regular.

Let p be the pumping length and let $s = \underline{0^{p+1} 1^p}$. This s can be split into $s = xyz$ with $|y| \geq 1$ and $|xy| \leq p$, and $xy^n z \in E$ for any $n \geq 0$.

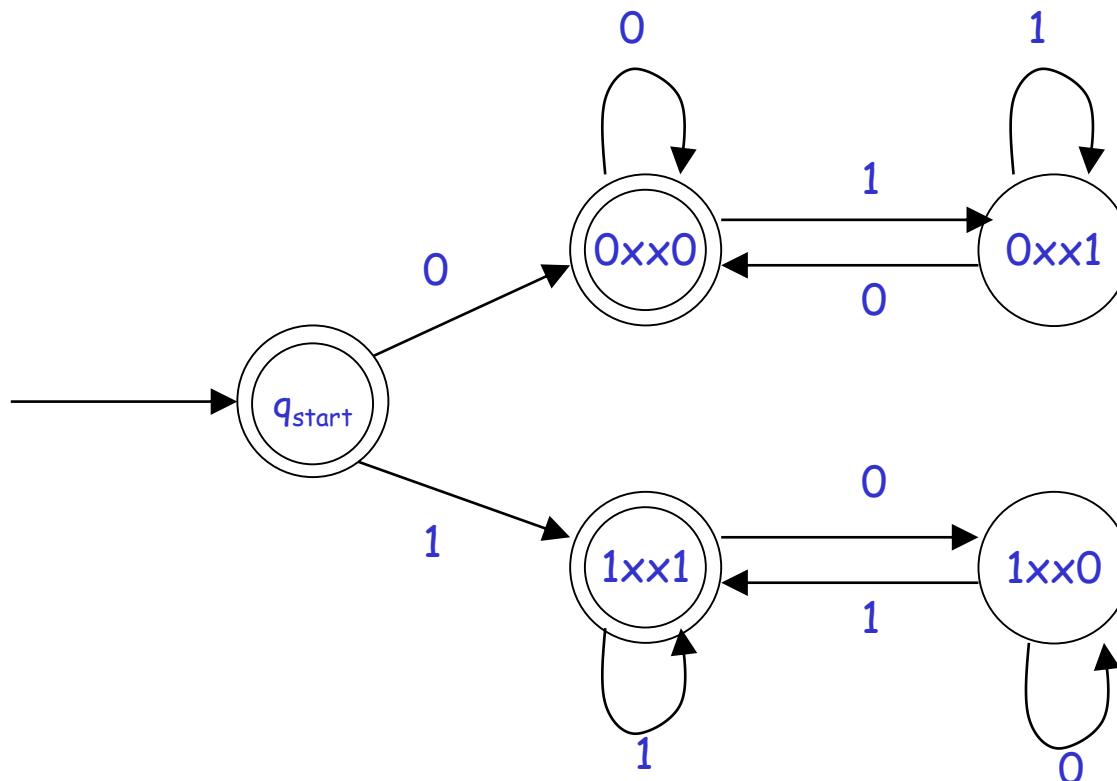
Then we must have $y = 0^k$ for some nonzero k .

We can easily see that $\cancel{xy^0z} = xz \notin E$, which contradicts the assumption.



Example 6 (Differential Encoding)

Claim : $D = \{w \mid w \text{ contains equal number of occurrences of the substrings } 01 \text{ and } 10\}$ is regular.





BLM2502

Theory of Computation

Spring 2015

BLM2502 Theory of Computation

» Course Outline

- | » Week | Content |
|--------|---|
| » 1 | Introduction to Course |
| » 2 | Computability Theory, Complexity Theory, Automata Theory, Set Theory, Relations, Proofs, Pigeonhole Principle |
| » 3 | Regular Expressions |
| » 4 | Finite Automata |
| » 5 | Deterministic and Nondeterministic Finite Automata |
| » 6 | Epsilon Transition, Equivalence of Automata |
| » 7 | Pumping Theorem |
| » 8 | April 10 - 14 week is the first midterm week |
| » 9 | Context Free Grammars |
| » 10 | Parse Tree, Ambiguity, |
| » 11 | Pumping Theorem |
| » 12 | Turing Machines, Recognition and Computation, Church-Turing Hypothesis |
| » 13 | Turing Machines, Recognition and Computation, Church-Turing Hypothesis |
| » 14 | May 22 – 27 week is the second midterm week |
| » 15 | Review |
| » 16 | Final Exam date will be announced |



Context-Free Languages

»

Context-Free Languages

$\{\underline{a}^n b^n : n \geq 0\}$

$\{\underline{w}w^R\}$

Regular Languages

$a^* b^*$

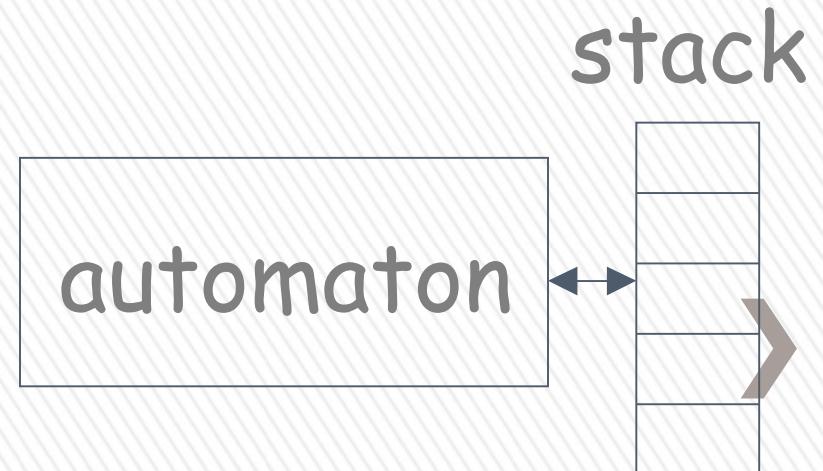
$(a + b)^*$



Context-Free Languages

Context-Free
Grammars

Pushdown
Automata

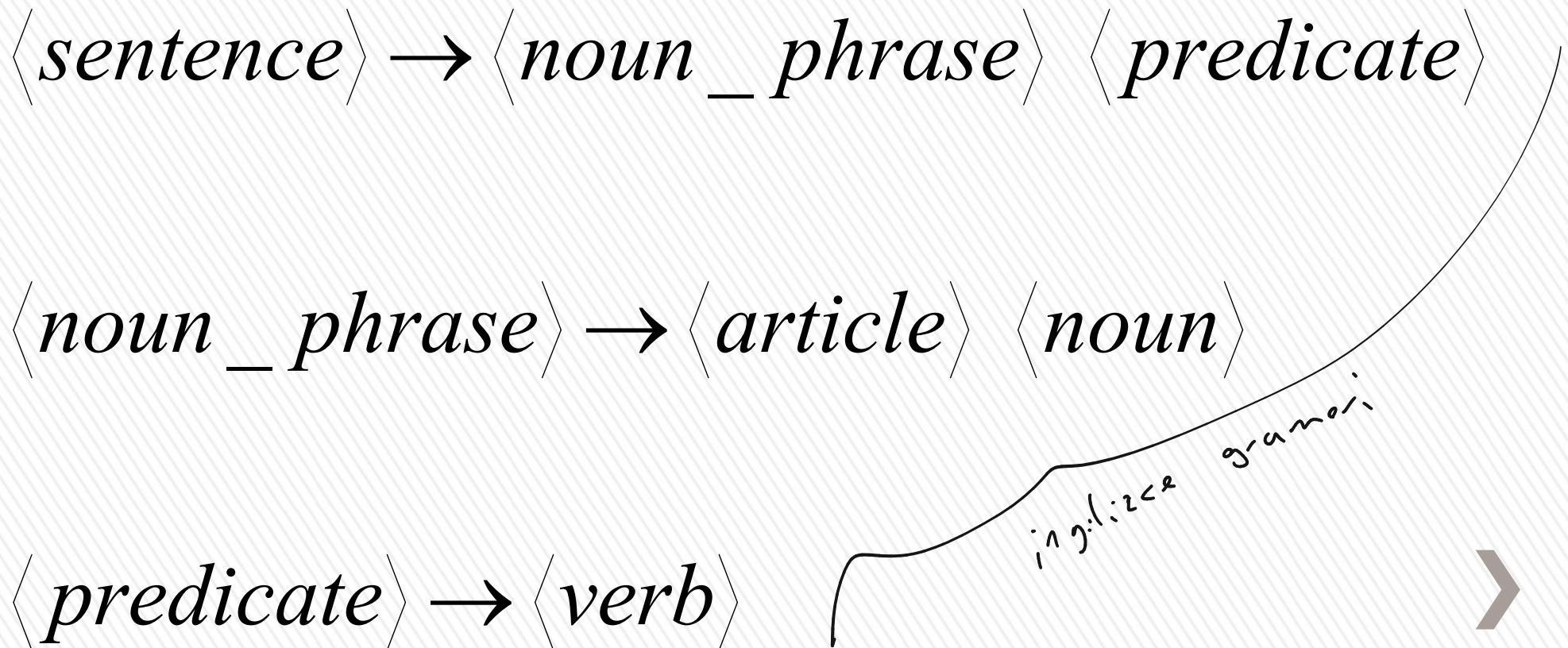




Context-Free Grammars

Grammars

- » Grammars express languages
- » Example: the English language grammar



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

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

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

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

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

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



» Derivation of string “the dog walks” :

$\langle sentence \rangle \Rightarrow \langle noun_phrase \rangle \langle predicate \rangle$
 $\Rightarrow \langle noun_phrase \rangle \langle verb \rangle$
 $\Rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle$
 $\Rightarrow the \langle noun \rangle \langle verb \rangle$
 $\Rightarrow the \ dog \langle verb \rangle$
 $\Rightarrow the \ dog \ sleeps$



» Derivation of string “a cat runs” :

$$\begin{aligned}\langle \textit{sentence} \rangle &\Rightarrow \langle \textit{noun_phrase} \rangle \langle \textit{predicate} \rangle \\&\Rightarrow \langle \textit{noun_phrase} \rangle \langle \textit{verb} \rangle \\&\Rightarrow \langle \textit{article} \rangle \langle \textit{noun} \rangle \langle \textit{verb} \rangle \\&\Rightarrow a \langle \textit{noun} \rangle \langle \textit{verb} \rangle \\&\Rightarrow a \textit{ cat } \langle \textit{verb} \rangle \\&\Rightarrow a \textit{ cat runs}\end{aligned}$$


» Language of the grammar:

$L = \{ \text{"a cat runs"},$
 $\text{"a cat sleeps"},$
 $\text{"the cat runs"},$
 $\text{"the cat sleeps"},$
 $\text{"a dog runs"},$
 $\text{"a dog sleeps"},$
 $\text{"the dog runs"},$
 $\text{"the dog sleeps"} \}$



Productions

Sequence of
Terminals (symbols)

$$\langle \text{noun} \rangle \rightarrow \overbrace{\text{cat}}$$
$$\langle \text{sentence} \rangle \rightarrow \langle \text{noun_phrase} \rangle \langle \text{predicate} \rangle$$

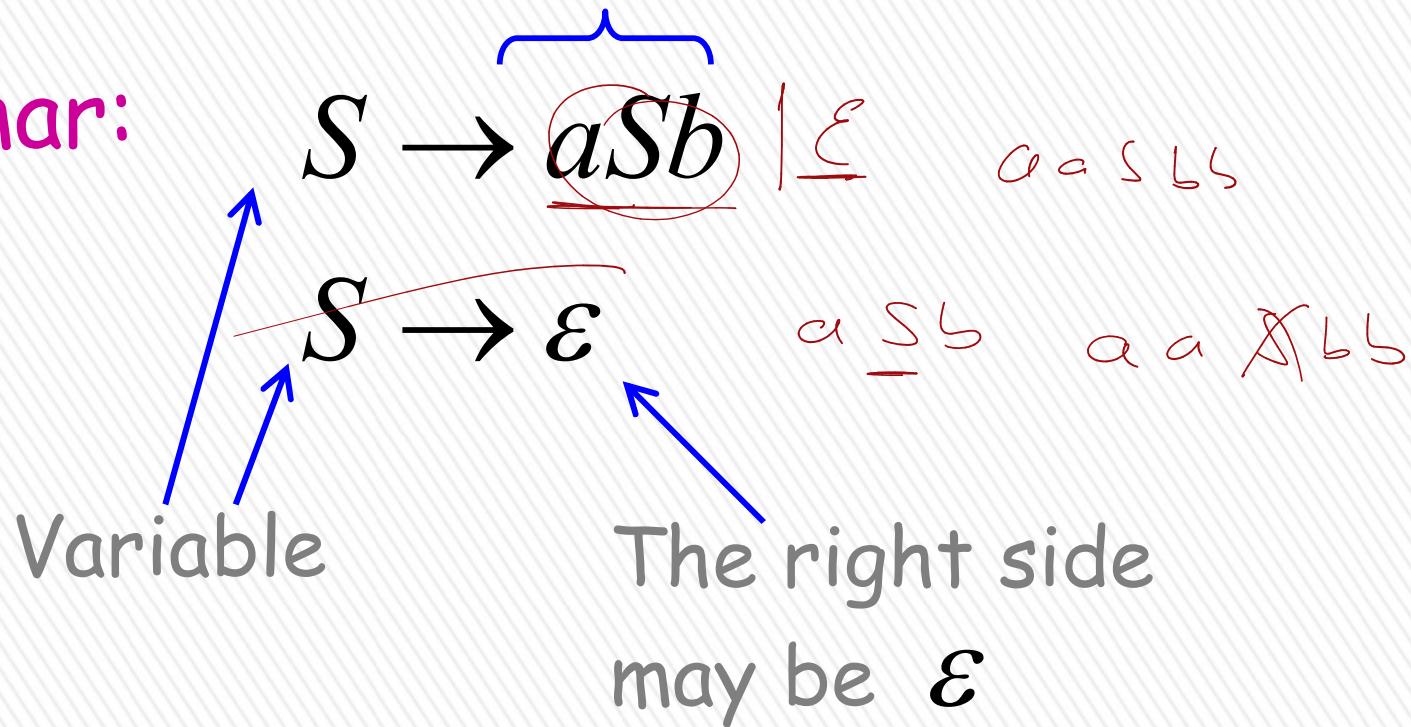
Variables
sent, noun, phrase, predicate

Sequence of Variables



Another Example

Grammar:



» Grammar:

$$S \rightarrow aSb$$

$$S \rightarrow \epsilon \Rightarrow$$

aSb
acSbb
abc

» Derivation of string : *ab*

$$\begin{array}{c} S \Rightarrow aSb \Rightarrow ab \\ S \rightarrow aSb \qquad \qquad \qquad S \rightarrow \epsilon \end{array}$$



» Grammar:

$$S \rightarrow aSb$$

$$S \rightarrow \epsilon$$

» Derivation of string :

aabb

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$



$$S \rightarrow aSb$$

$$S \rightarrow \epsilon$$



Grammar: $S \rightarrow aSb$

$S \rightarrow \epsilon$

Other derivations:

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

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

$\Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$



Grammar:

$$S \rightarrow aSb$$

$$S \rightarrow \epsilon$$

Language of the grammar:

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



A Convenient Notation

» We write:

$$S \xrightarrow{*} aaabbb$$

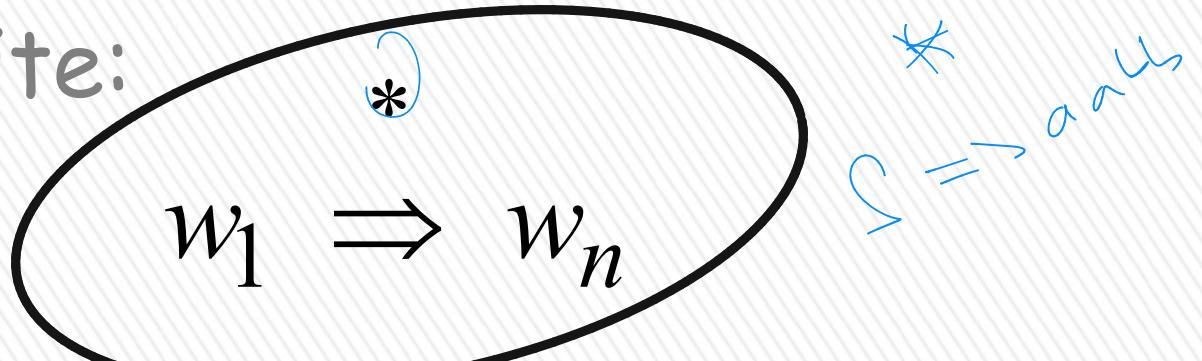
for one or more derivation steps

» Instead of:

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



In general we write:

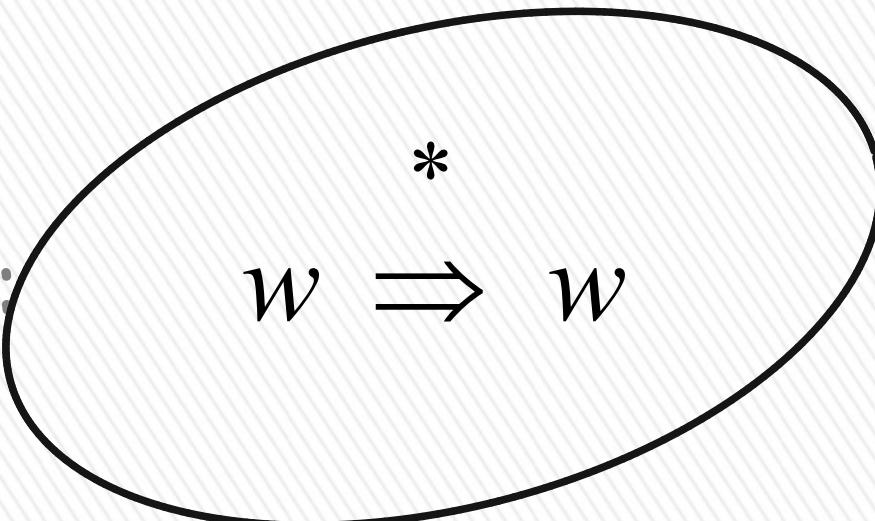


If:

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

in zero or more derivation steps

Trivially:



Example Grammar

$$S \rightarrow aSb$$

$$S \rightarrow \epsilon$$

$$S \xrightarrow{*} aaSbb \xrightarrow{*} aaaaSbbbb$$

Possible Derivations

$$S \xrightarrow{*} \epsilon$$

$$S \xrightarrow{*} ab$$

$$S \xrightarrow{*} aaabbb$$



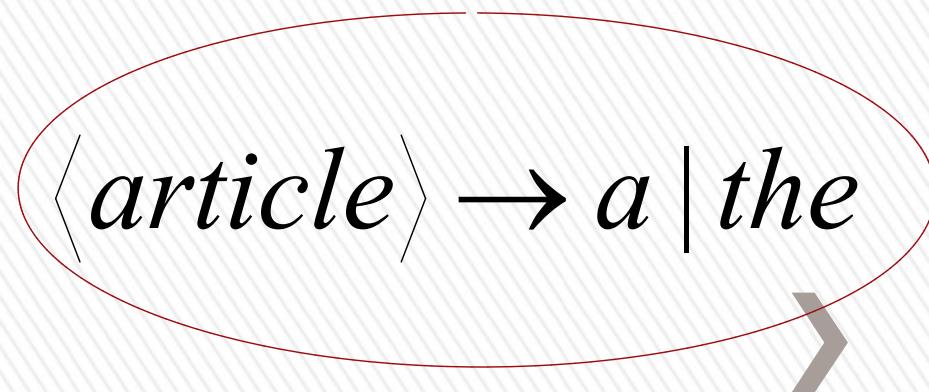
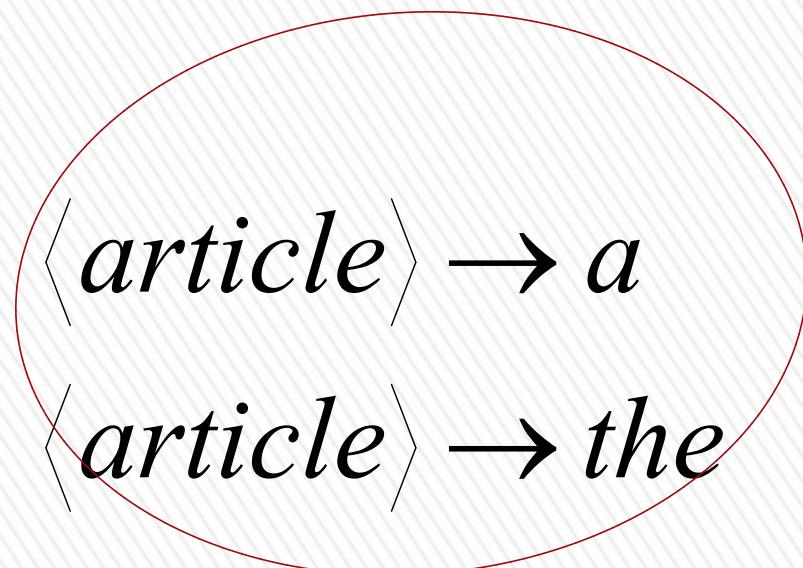
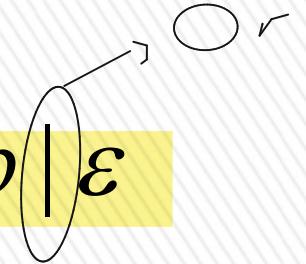
Another convenient notation:

$$S \rightarrow aSb \mid \epsilon$$

$$S \rightarrow \epsilon$$

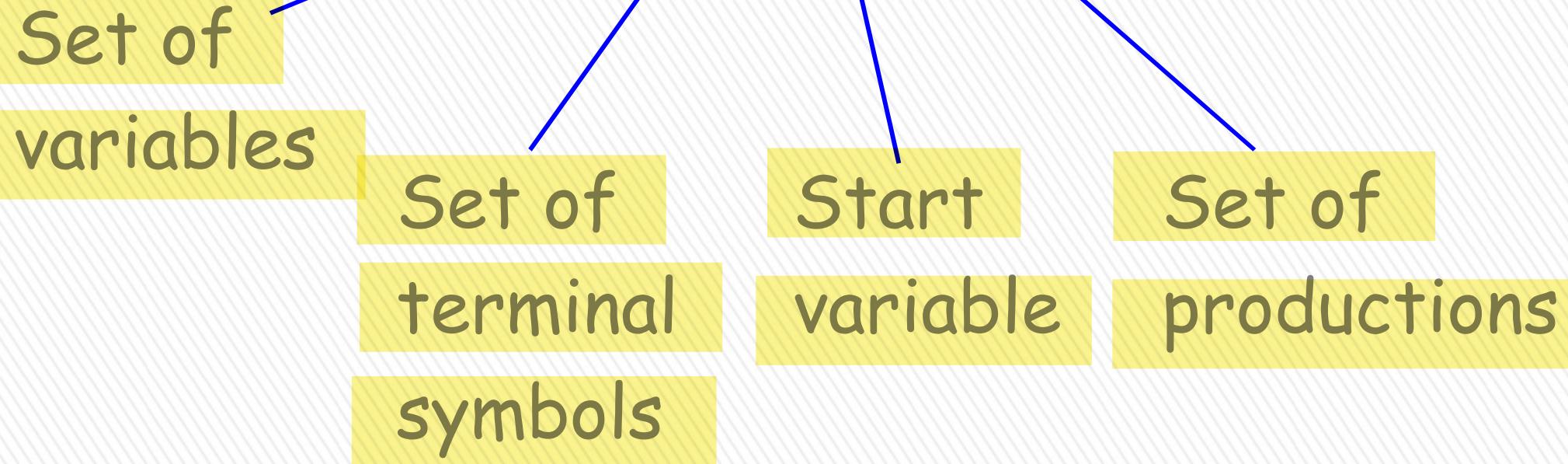


$$S \rightarrow aSb \mid \epsilon$$



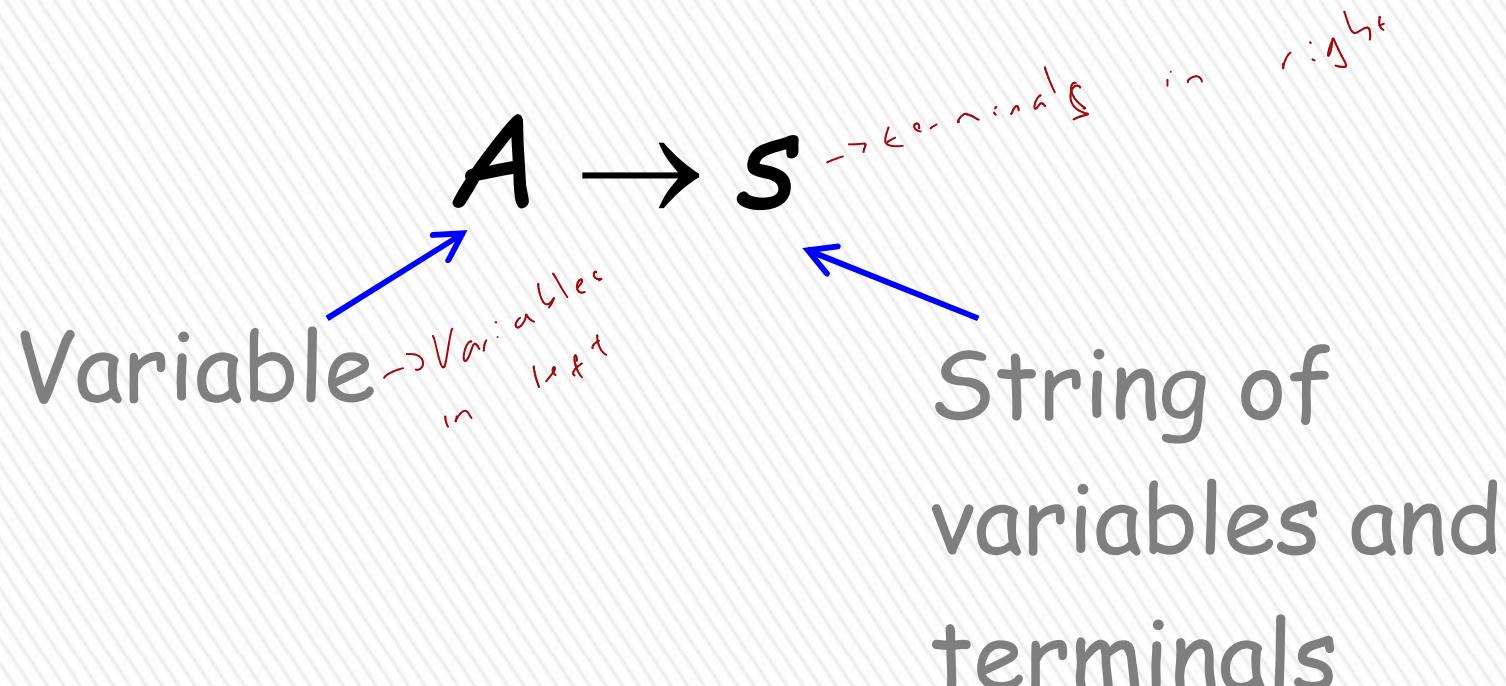
Formal Definition

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



Context-Free Grammar: $G = (V, T, S, P)$

All productions in P are of the form



Example for Context-Free Grammar

$$S \rightarrow aSb \mid \epsilon$$

$$V = S, L, P$$

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

$$S = L$$

$$P = \{S \rightarrow aSL, S \rightarrow aL, S \rightarrow \epsilon\}$$

productions

$$P = \{S \rightarrow aSb, S \rightarrow \epsilon\}$$

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

$$V = \{S\}$$

variables

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

terminals

start variable

list
set
mark

Language of a Grammar:

» For a grammar G with start variable S

$$L(G) = \{w : S \xrightarrow{*} w, w \in T^*\}$$

String of terminals or ϵ



Example:

context-free grammar G : $S \rightarrow aSb \mid \epsilon$

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

Since, there is derivation

$$S \xrightarrow{*} a^n b^n \quad \text{for any } n \geq 0 \Rightarrow$$

$$L = \{ 0^k 1^k \}$$

Context-Free Language:

regular language context-free language implies $L \rightarrow E \cup S \cup S^*$
all languages

- » A language L is context-free
- » if there is a context-free grammar G
- » with $L = L(G)$



Example:

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

is a context-free language
since context-free grammar G :

$$S \rightarrow aSb \mid \epsilon$$

generates $L(G) = L$



Another Example

Context-free grammar G :

$$S \rightarrow \underline{aSa} \mid bSb \mid \underline{\epsilon} \mid a \mid b$$

Example derivations:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$$

$$L(G) = \{ww^R : w \in \{a,b\}^*\}$$

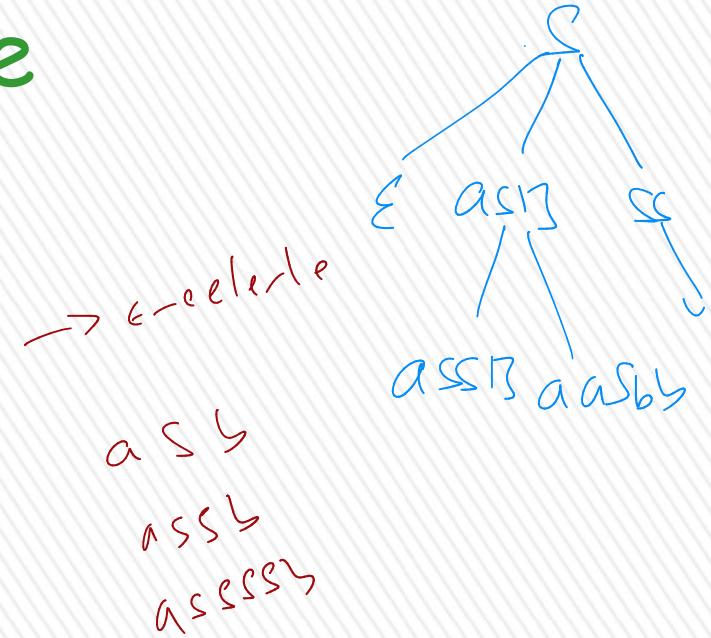
Palindromes of even length



Another Example

Context-free grammar G :

$$S \rightarrow aSb \mid SS \mid \epsilon \quad \xrightarrow{\text{euler}}$$



Example derivations:

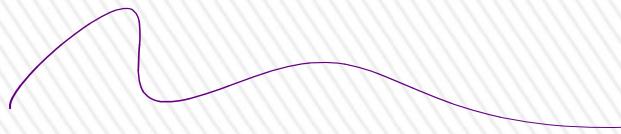
$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow ab$$

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$$

$$L(G) = \{w : n_a(w) = n_b(w), \\ \text{and } n_a(v) \geq n_b(v) \\ \text{in any prefix } v\}$$

Describes
matched
parentheses

() ((()))) (()) a = (, b =)



Derivation Trees

Derivation Order

Consider the following example grammar
with 5 productions:

- | | | |
|-----------------------|-----------------------------|-----------------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow Bb$ |
| | 3. $A \rightarrow \epsilon$ | 5. $B \rightarrow \epsilon$ |

aa A 



- | | | |
|-------------------------------------|---|-----------------------|
| 1. $S \rightarrow \textcircled{AB}$ | 2. $A \rightarrow aa\textcircled{A}$ | 4. $B \rightarrow Bb$ |
| 3. $A \rightarrow \epsilon$ | 5. $B \rightarrow \epsilon$
<small>soldan başlıyoruz</small> | |

Leftmost derivation order of string \overbrace{aab} :

$$S \xrightarrow{1} AB \xrightarrow{2} aaAB \xrightarrow{3} aaB \xrightarrow{4} aaBb \xrightarrow{5} aab$$

At each step, we substitute the leftmost variable



① Derivation trees

- ORDER

Left ↙ Right ↘

- | | | |
|-----------------------------|--|-----------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow Bb$ |
| 3. $A \rightarrow \epsilon$ | 5. $B \rightarrow \epsilon$
<small><i>fis s^e</i></small> | |

Rightmost derivation order of string aab :

$$\begin{array}{ccccccc}
 & 1 & & 4 & & 5 & \\
 S \Rightarrow & AB \Rightarrow & ABb \Rightarrow & Ab \Rightarrow & aaAb \Rightarrow & aab
 \end{array}$$

At each step, we substitute the rightmost variable



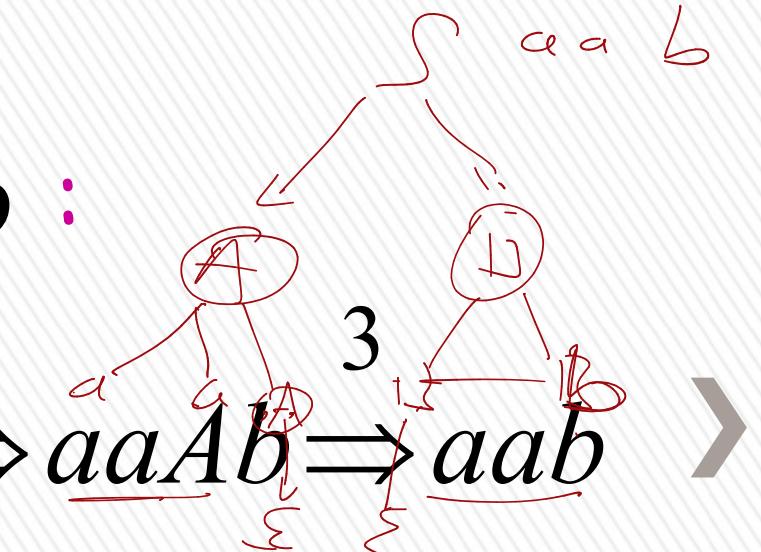
- | | | |
|---|---|-----------------------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow \underline{Bb}$ |
| 3. $A \rightarrow \underline{\epsilon}$ | 5. $B \rightarrow \underline{\epsilon}$ | |

Leftmost derivation of aab :

$$\begin{array}{ccccccc}
 1 & & 2 & & 3 & & 4 & & 5 \\
 S \Rightarrow \underline{AB} \Rightarrow aa\underline{AB} \Rightarrow \underline{aaB} \Rightarrow aa\underline{Bb} \Rightarrow aab
 \end{array}$$

Rightmost derivation of aab :

$$\begin{array}{ccccccc}
 1 & & 4 & & 5 & & 2 \\
 S \Rightarrow A\underline{B} \Rightarrow A\underline{Bb} \Rightarrow \underline{Ab} \Rightarrow aa\underline{Ab} \Rightarrow aab
 \end{array}$$



Consider the same example grammar:

$$S \rightarrow AB \quad A \rightarrow aaA \mid \epsilon \quad B \rightarrow Bb \mid \epsilon$$

And a derivation of *aab*:

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

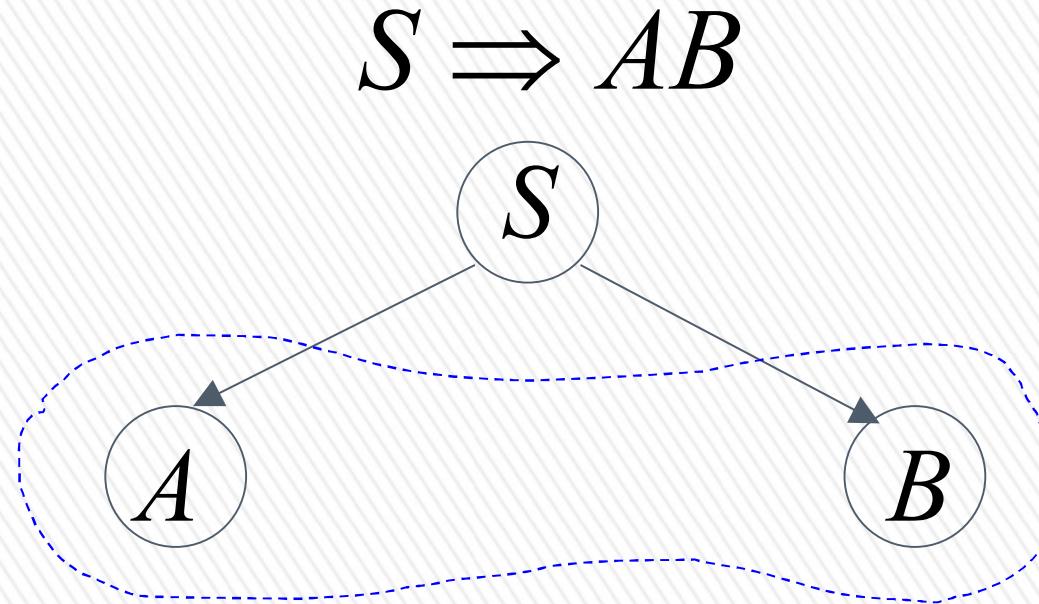


Derivation Tree

$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \epsilon$$

$$B \rightarrow Bb \mid \epsilon$$



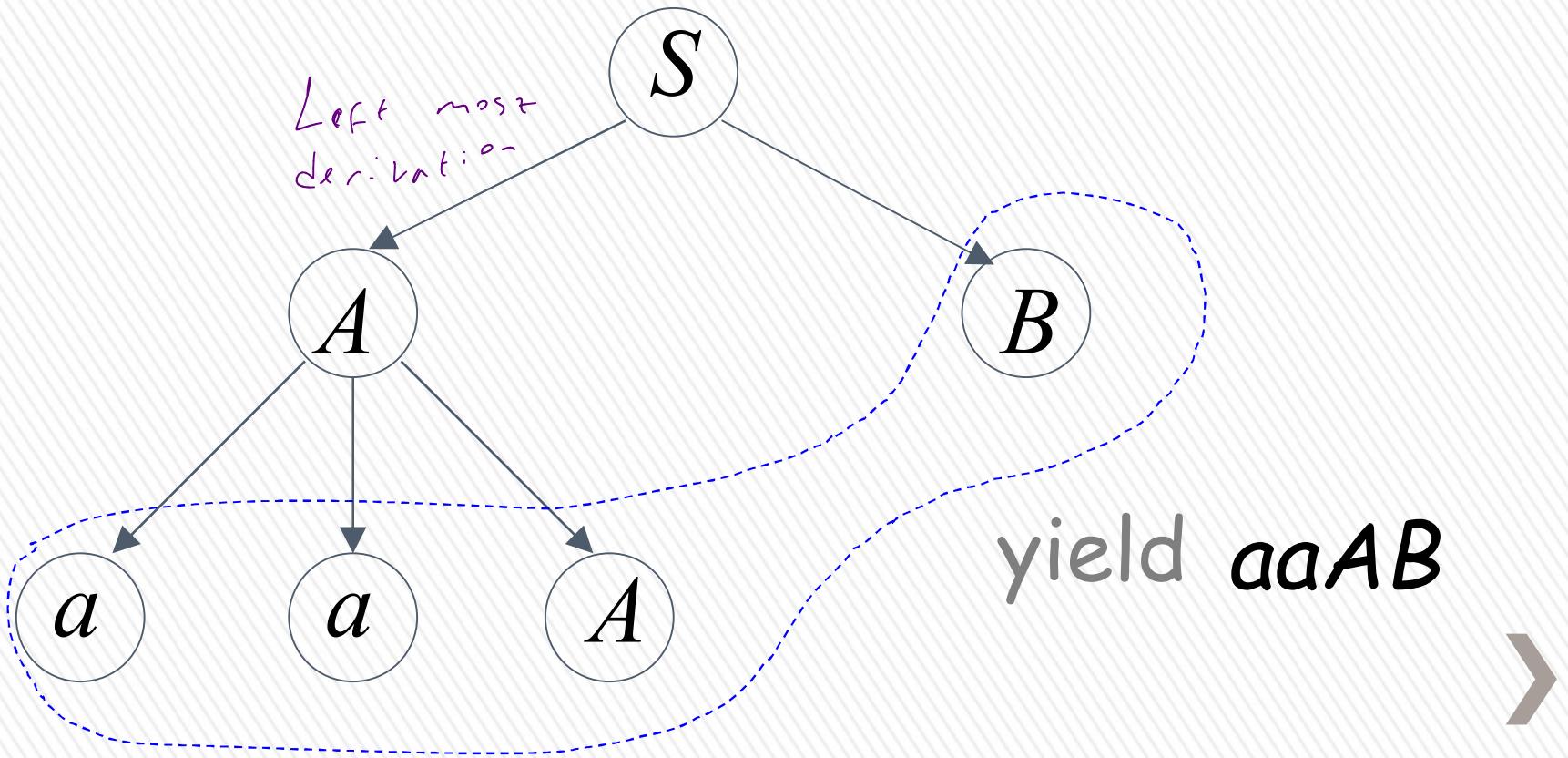
yield AB



$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \epsilon \quad B \rightarrow Bb \mid \epsilon$$

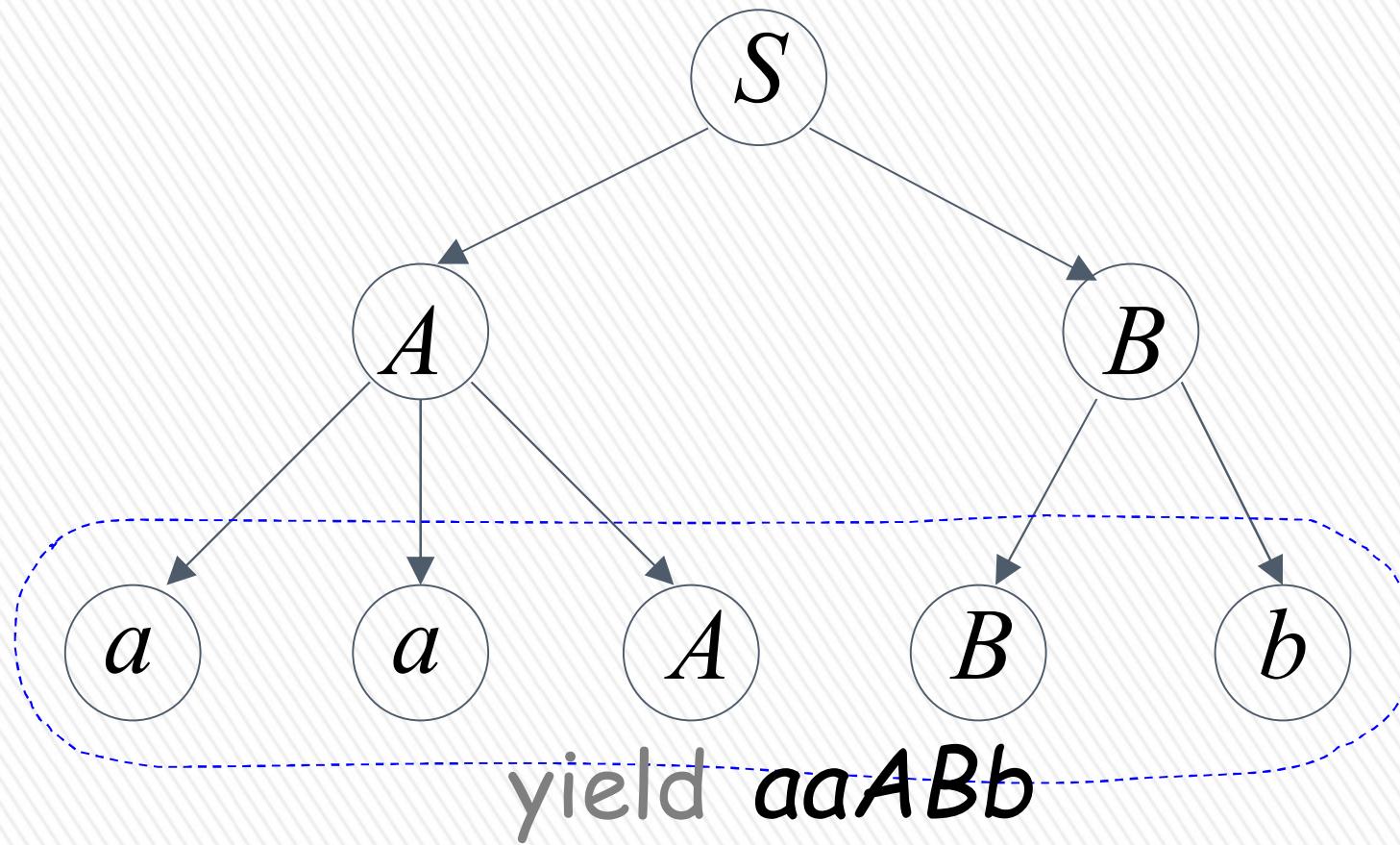
$$S \Rightarrow AB \Rightarrow aaAB$$



$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \epsilon \quad B \rightarrow Bb \mid \epsilon$$

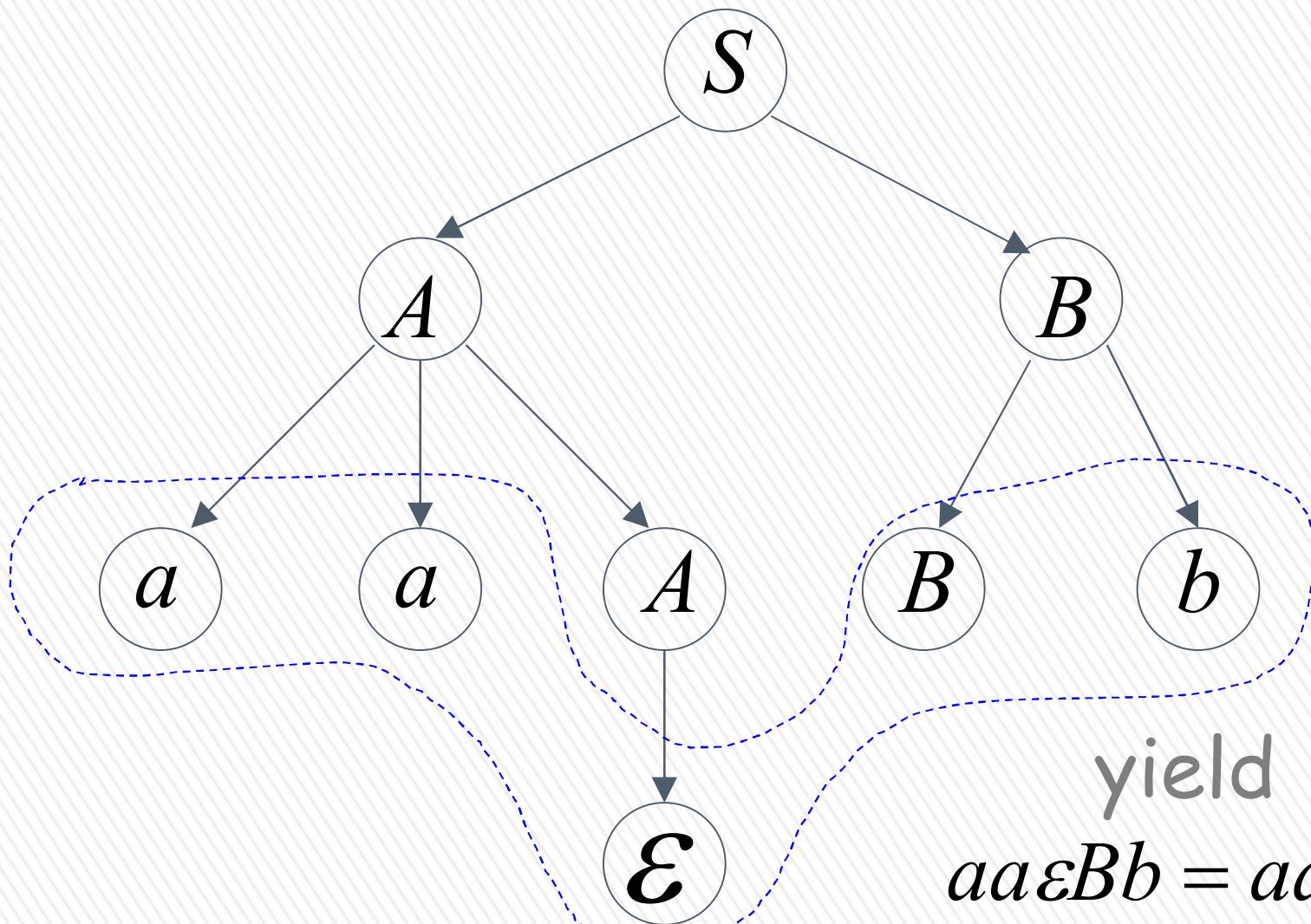
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb$$



$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \varepsilon \quad B \rightarrow Bb \mid \varepsilon$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb$$



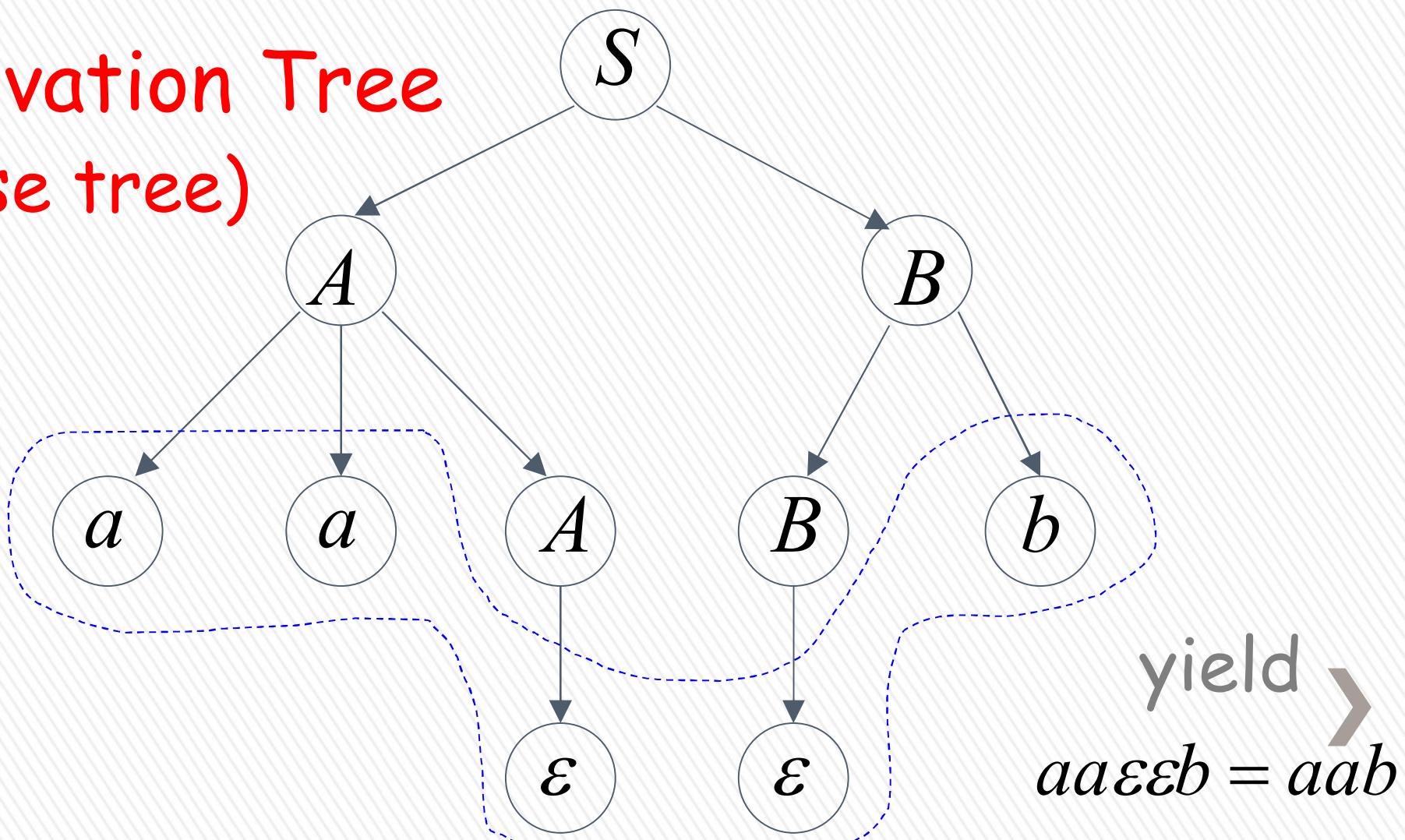
$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \varepsilon$$

$$B \rightarrow Bb \mid \varepsilon$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

Derivation Tree
(parse tree)



Sometimes, derivation order doesn't matter

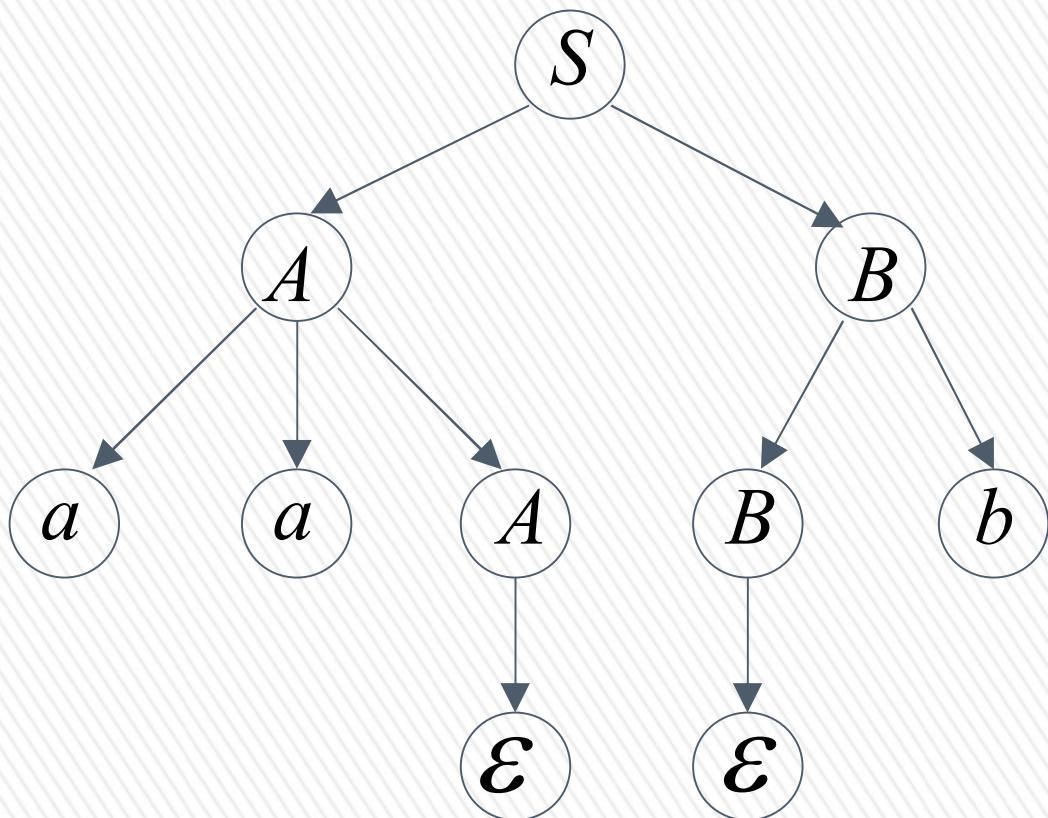
Leftmost derivation:

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aab$$

Rightmost derivation:

$$S \Rightarrow AB \Rightarrow ABb \Rightarrow Ab \Rightarrow aaAb \Rightarrow aab$$

Give same
derivation tree





→ Angular benzene tetro

Ambiguity

Grammar for mathematical expressions

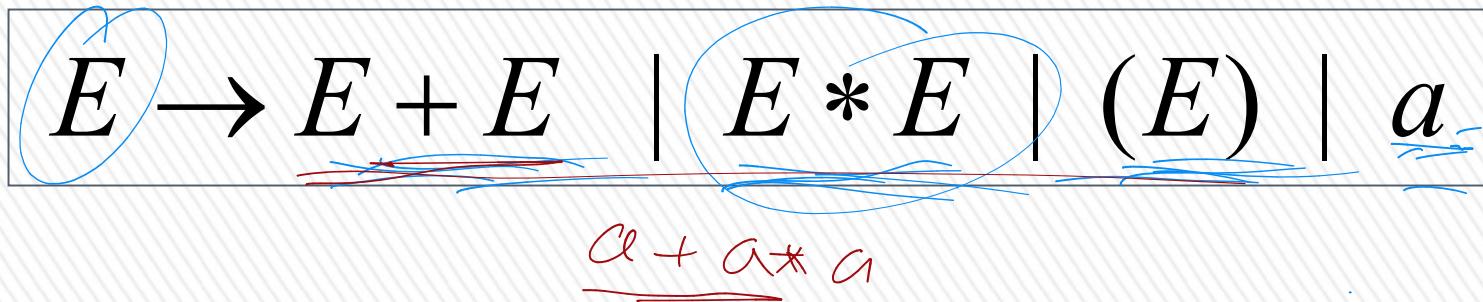
$$E \rightarrow E + E \quad | \quad E * E \quad | \quad (E) \quad | \quad a$$

Example strings:

$$(a + a) * a + (a + a * (a + a))$$

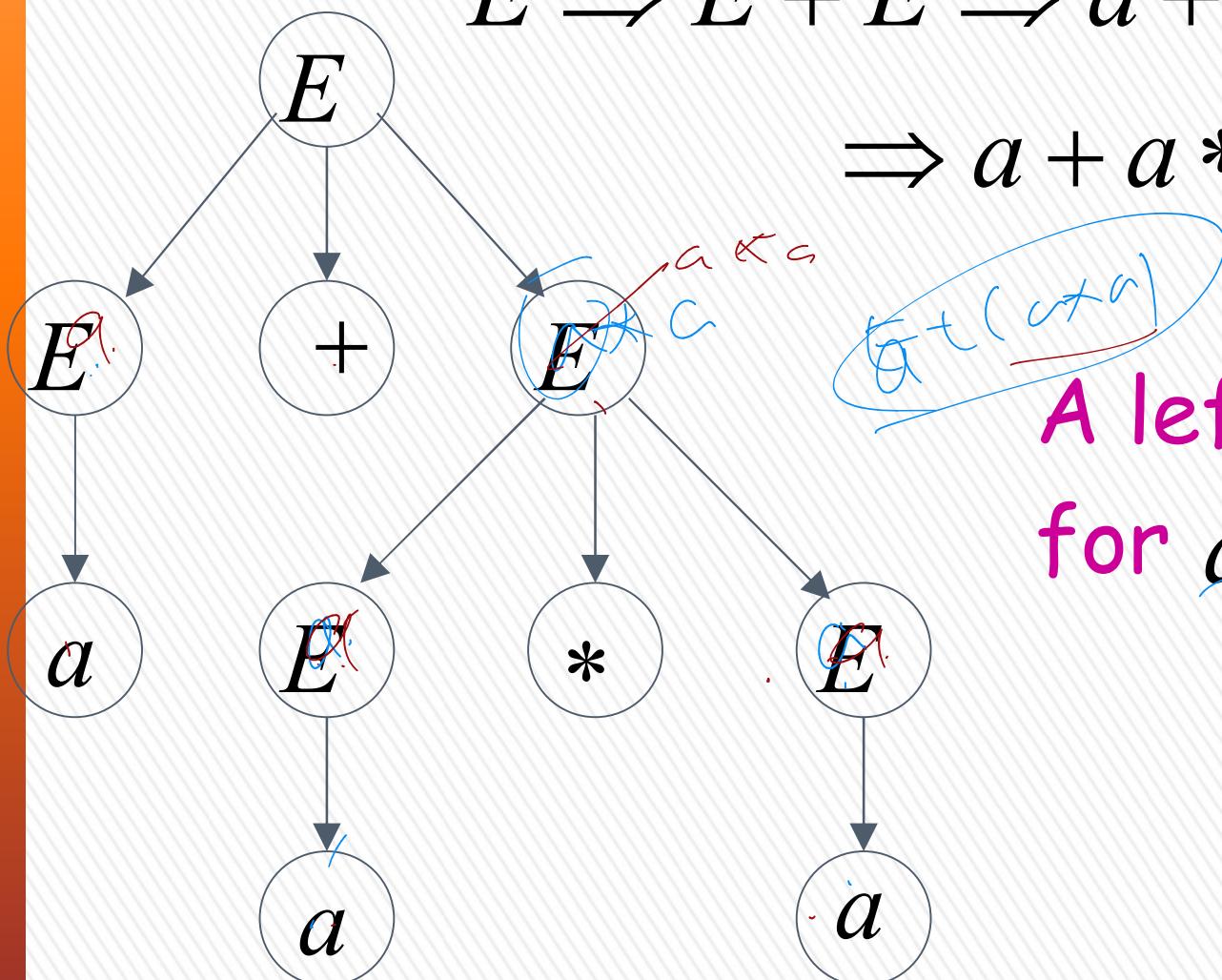

Denotes any number





$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E$

$\Rightarrow a + a * E \Rightarrow \underline{\cancel{a + a * a}}$



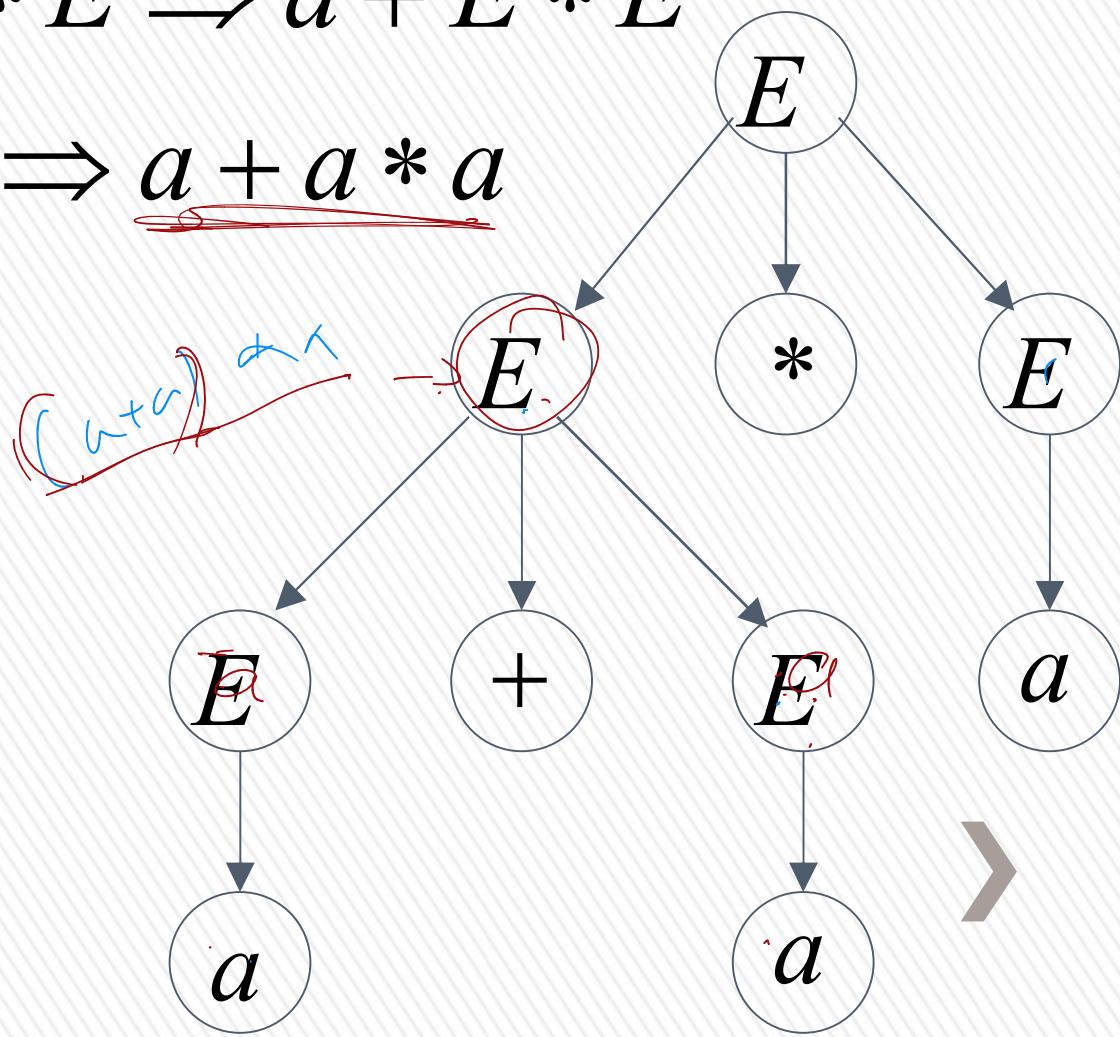
A leftmost derivation
for $a + a * a$



$$E \rightarrow E + E \quad | \quad E * E \quad | \quad (E) \quad | \quad a$$

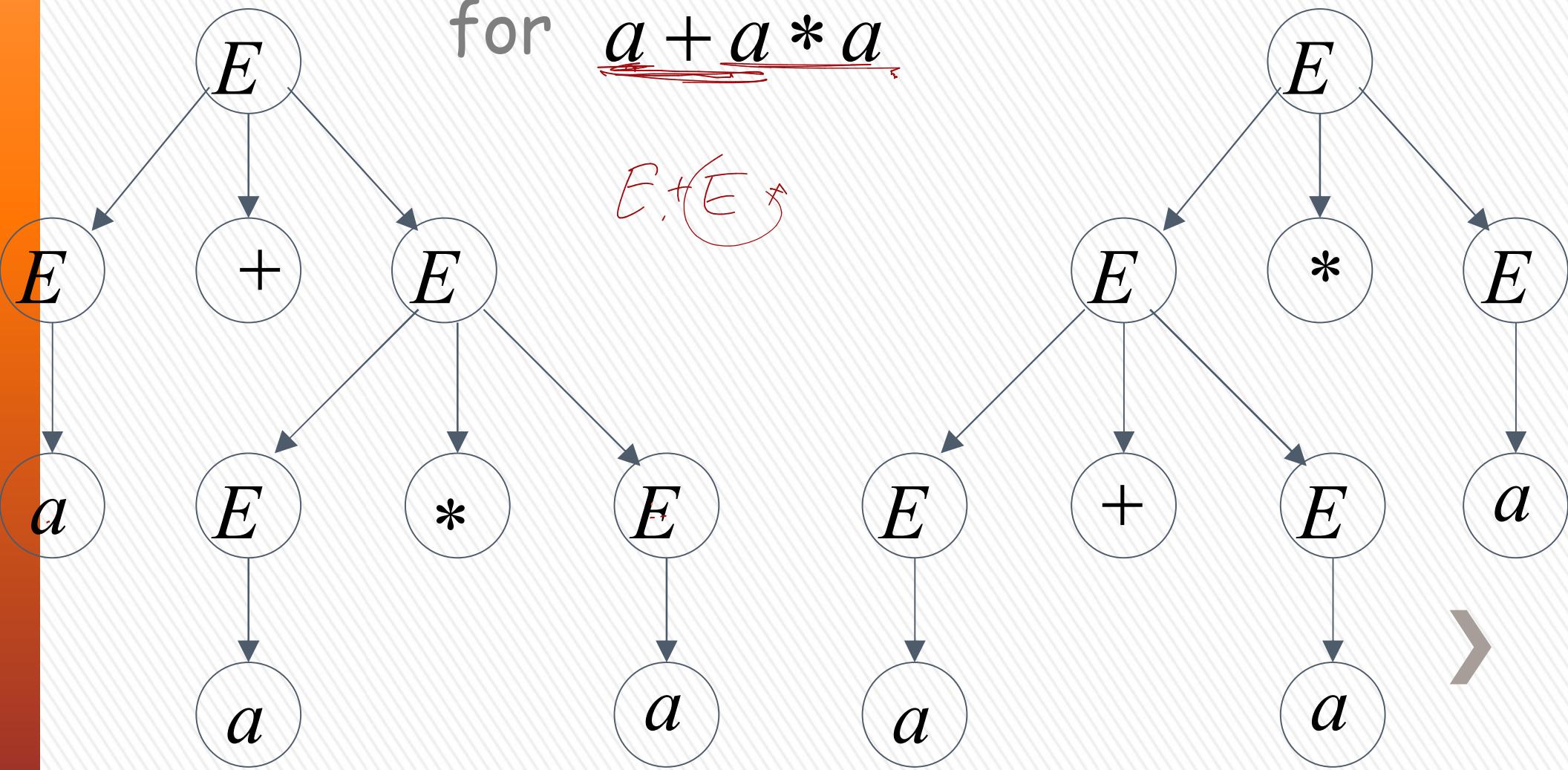
$$\begin{aligned} E &\Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \\ &\Rightarrow a + a * E \Rightarrow a + \cancel{a * a} \end{aligned}$$

Another
leftmost derivation
for $a + a * a$



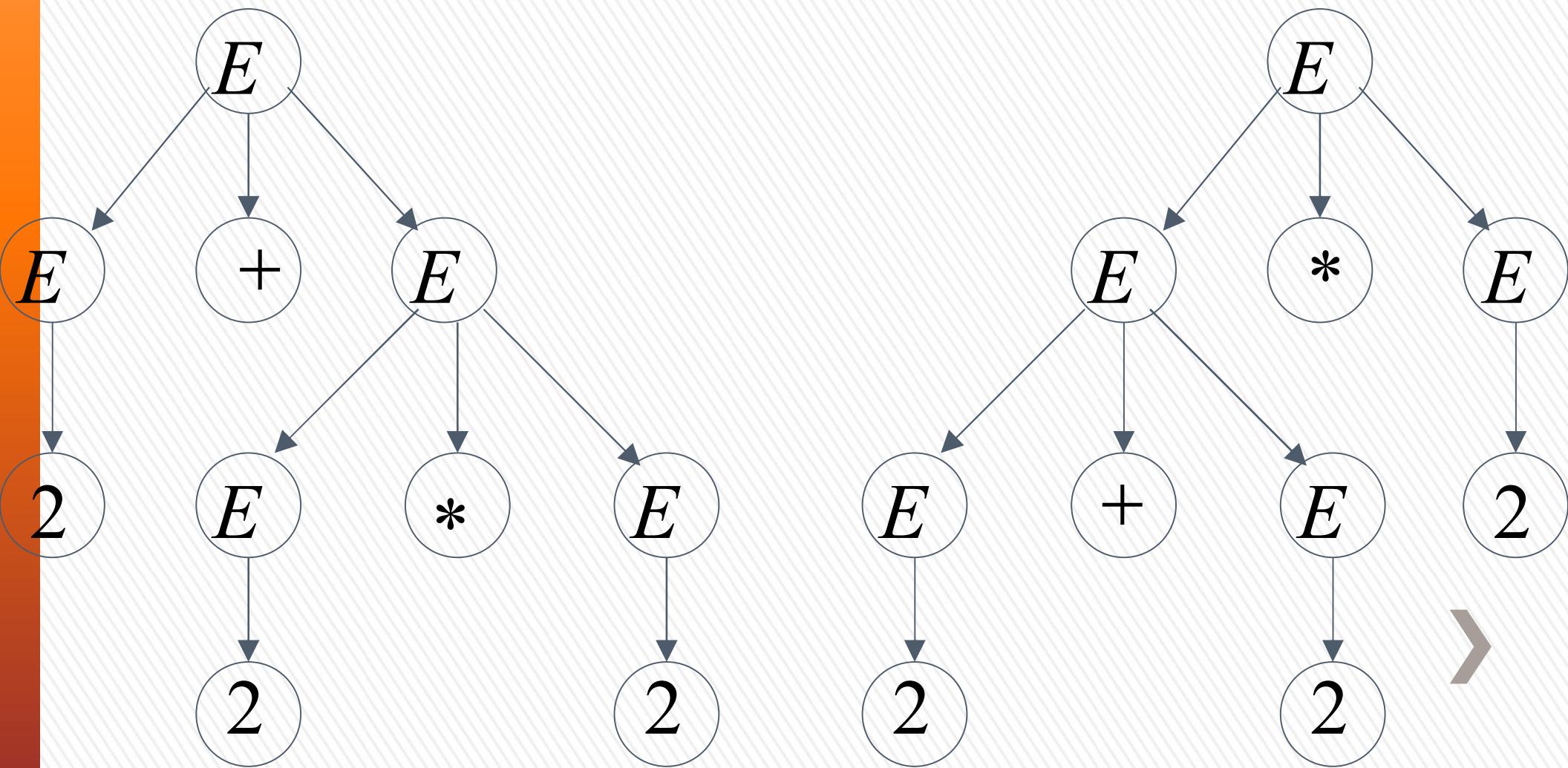
$$E \rightarrow E + E \quad | \quad E * E \quad | \quad (E) \quad | \quad a$$

Two derivation trees
for $a + a * a$



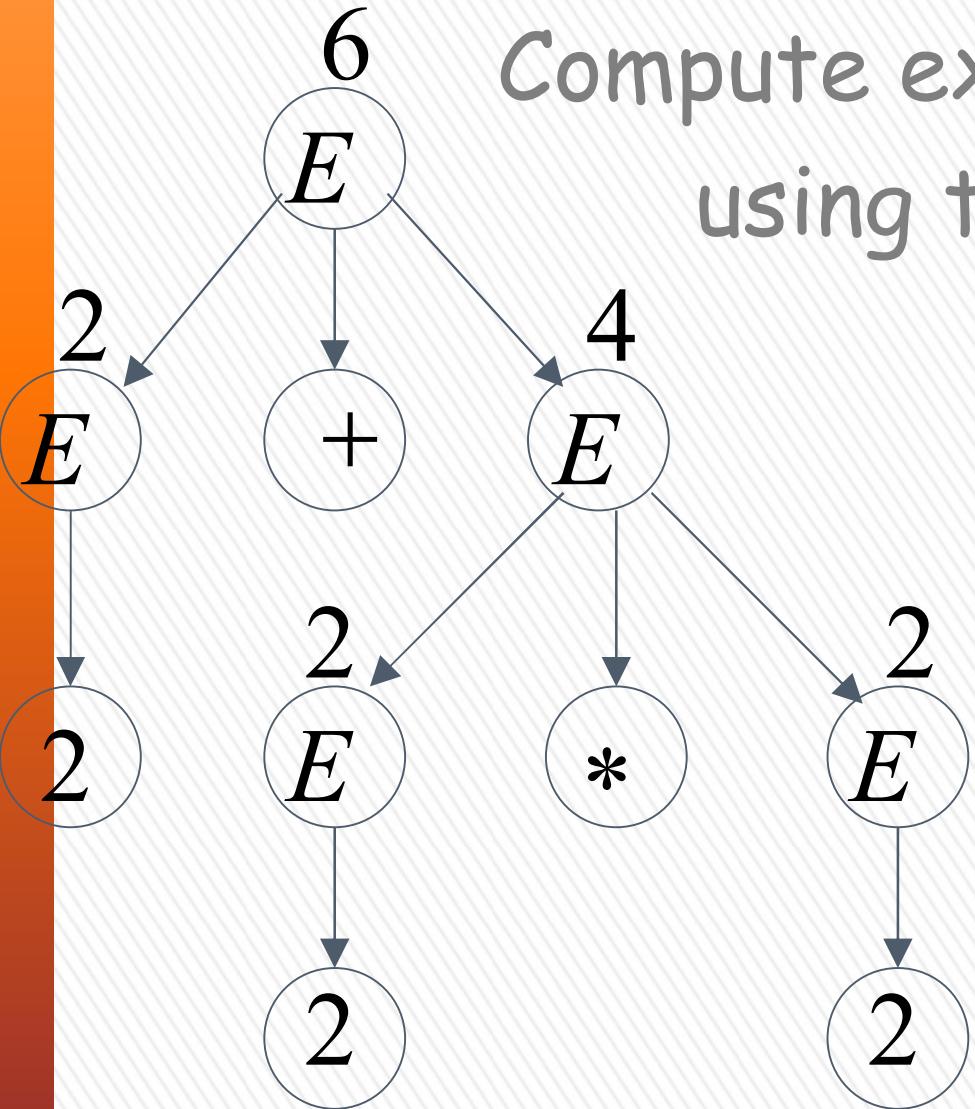
take $a = 2$

$$a + a * a = 2 + 2 * 2$$



Good Tree

$$2 + 2 * 2 = 6$$

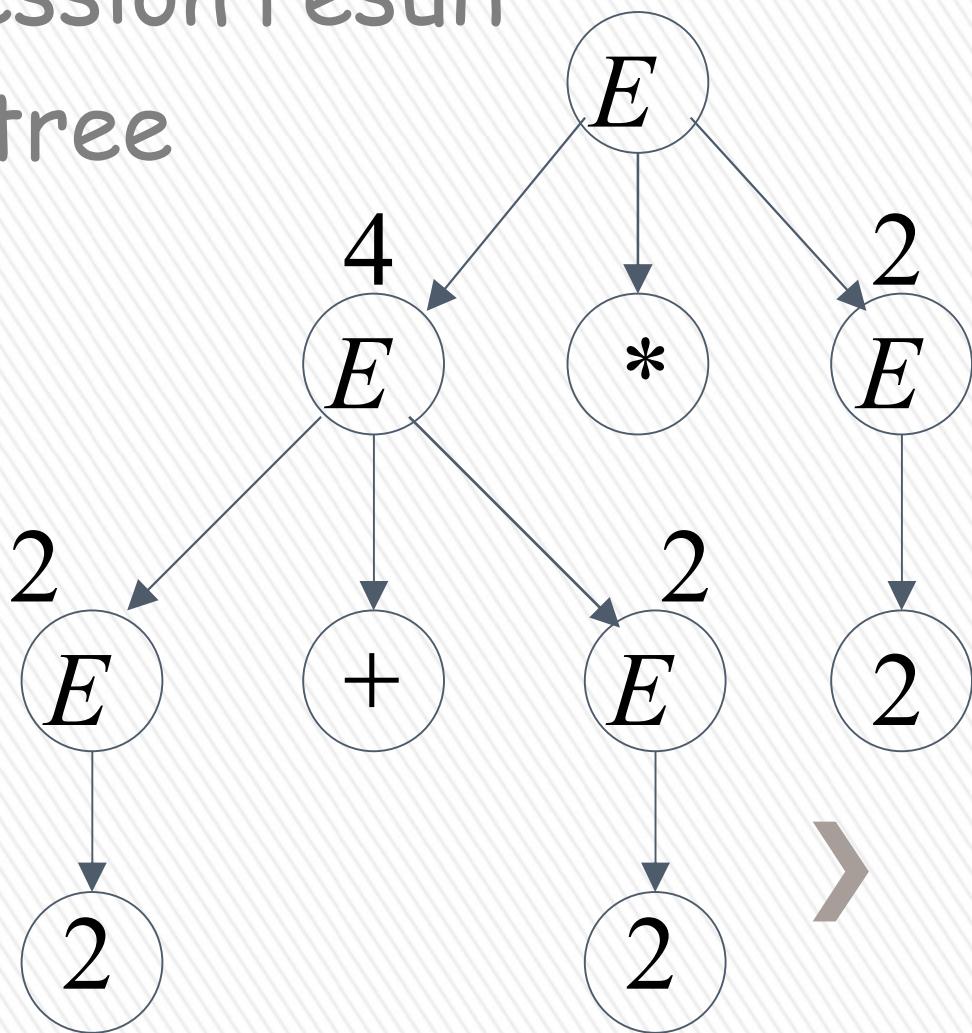


Compute expression result
using the tree

Bad Tree

$$2 + \underline{2 * 2} = 8$$

Önce deðlig:
en yilseð gecitme
8 başlað



Two different derivation trees
may cause problems in applications which
use the derivation trees:

- Evaluating expressions
- In general, in compilers
for programming languages



Ambiguous Grammar:

→ 2 farklı agac sözüne
Ambiguous

A context-free grammar G is ambiguous if there is a string $w \in L(G)$ which has:

two different derivation trees

or

two leftmost derivations

(Two different derivation trees give two different leftmost derivations and vice-versa)

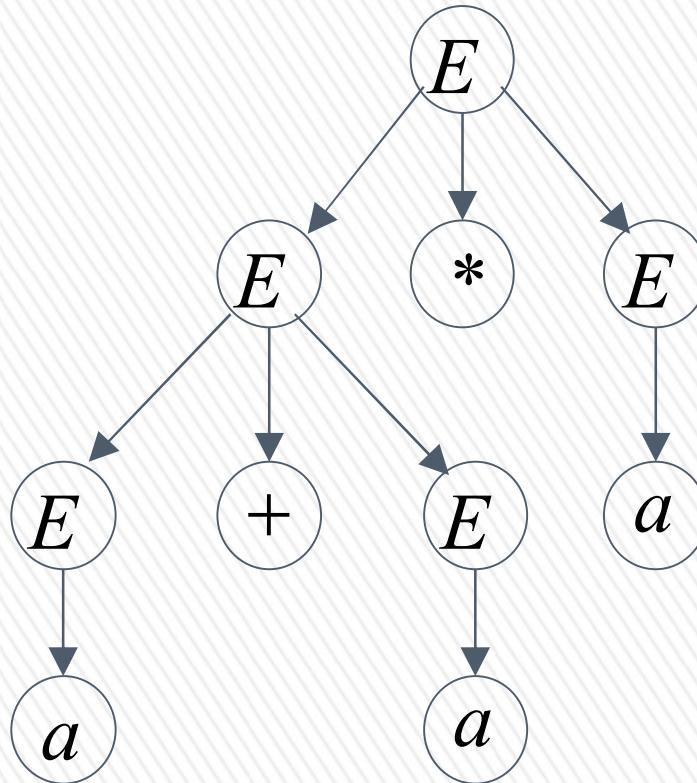
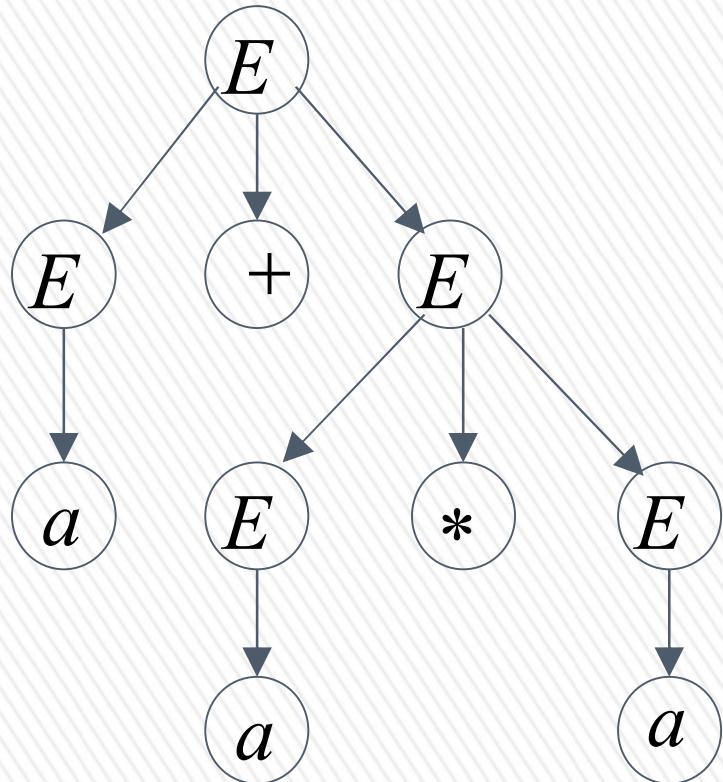


Example:

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

this grammar is ambiguous since

string $a + a * a$ has two derivation trees



$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

this grammar is ambiguous also because
string $a + a * a$ has two leftmost derivations

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E$$

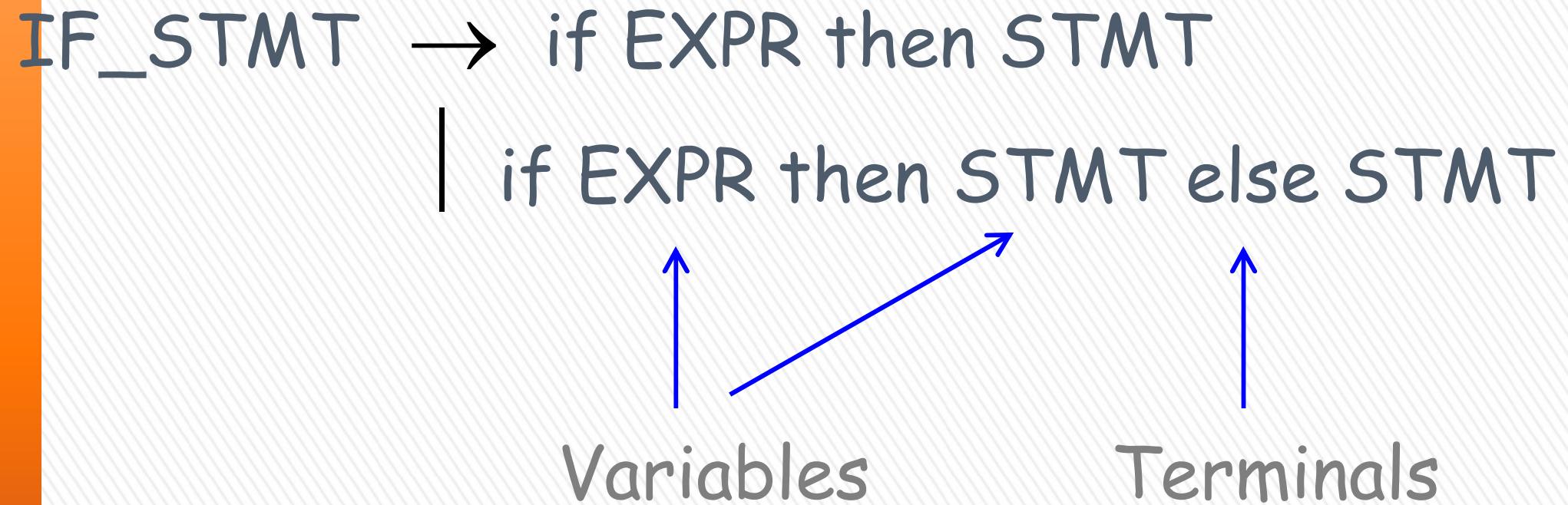
$$\Rightarrow a + a * E \Rightarrow a + a * a$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E$$

$$\Rightarrow a + a * E \Rightarrow a + a * a$$



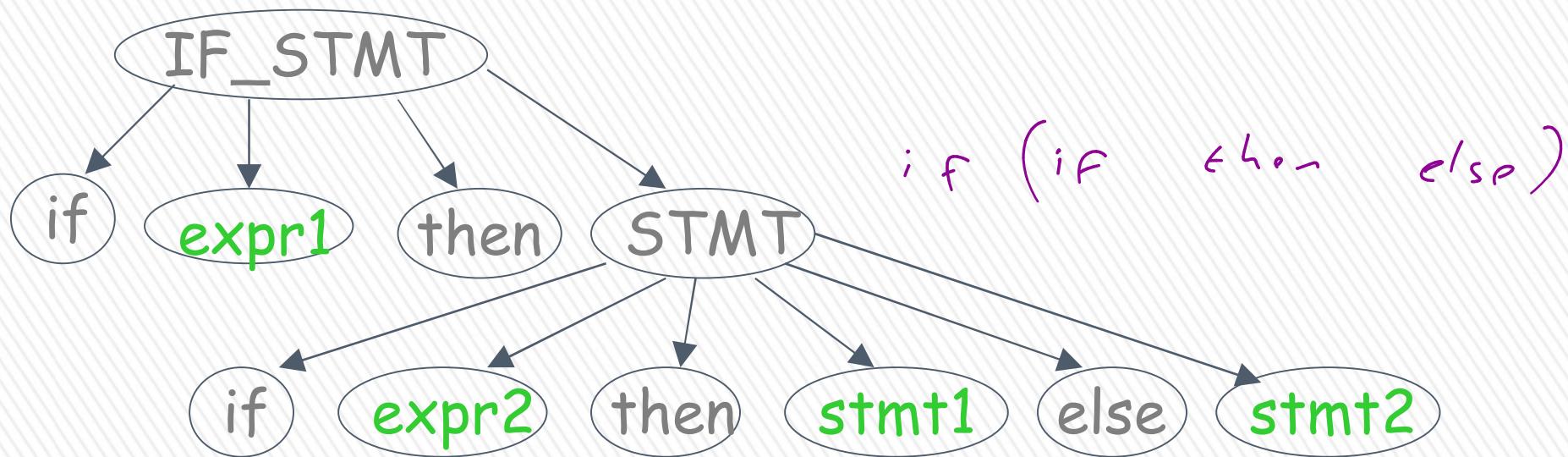
Another ambiguous grammar:



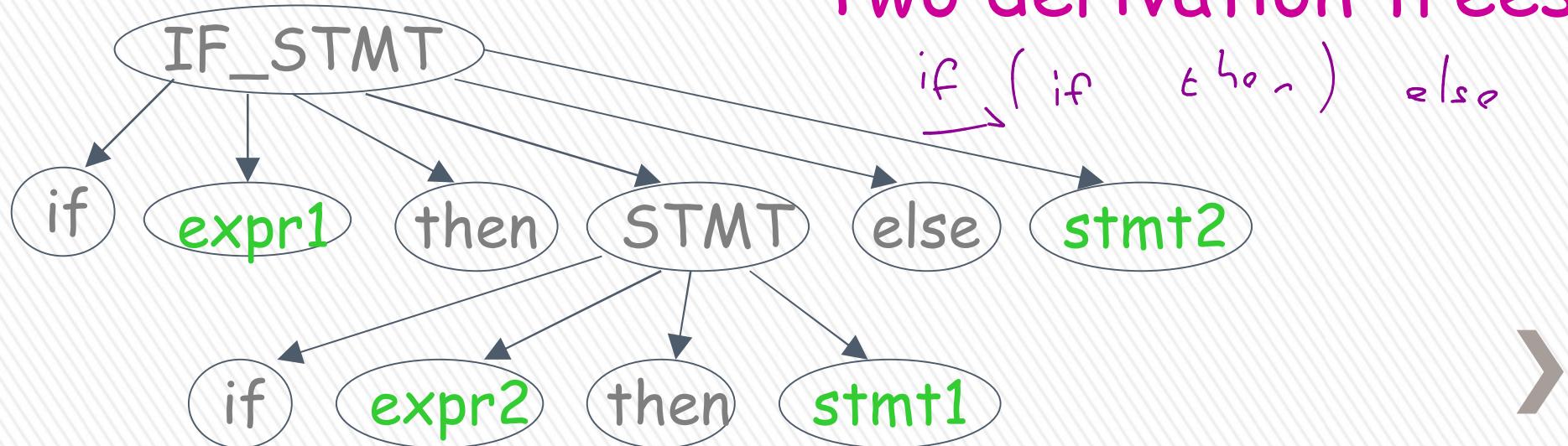
Very common piece of grammar
in programming languages



If expr1 then if expr2 then stmt1 else stmt2



Two derivation trees



In general, ambiguity is bad
and we want to remove it

Sometimes it is possible to find
a non-ambiguous grammar for a language

But, in general we cannot do so



A successful example:

Ambiguous Grammar

$$\begin{array}{l} E \rightarrow E + E \mid T \\ E \rightarrow E * E \mid F \\ E \rightarrow (E) \mid a \\ E \rightarrow a \end{array}$$

↳ degenerate state

Equivalent

Non-Ambiguous Grammar

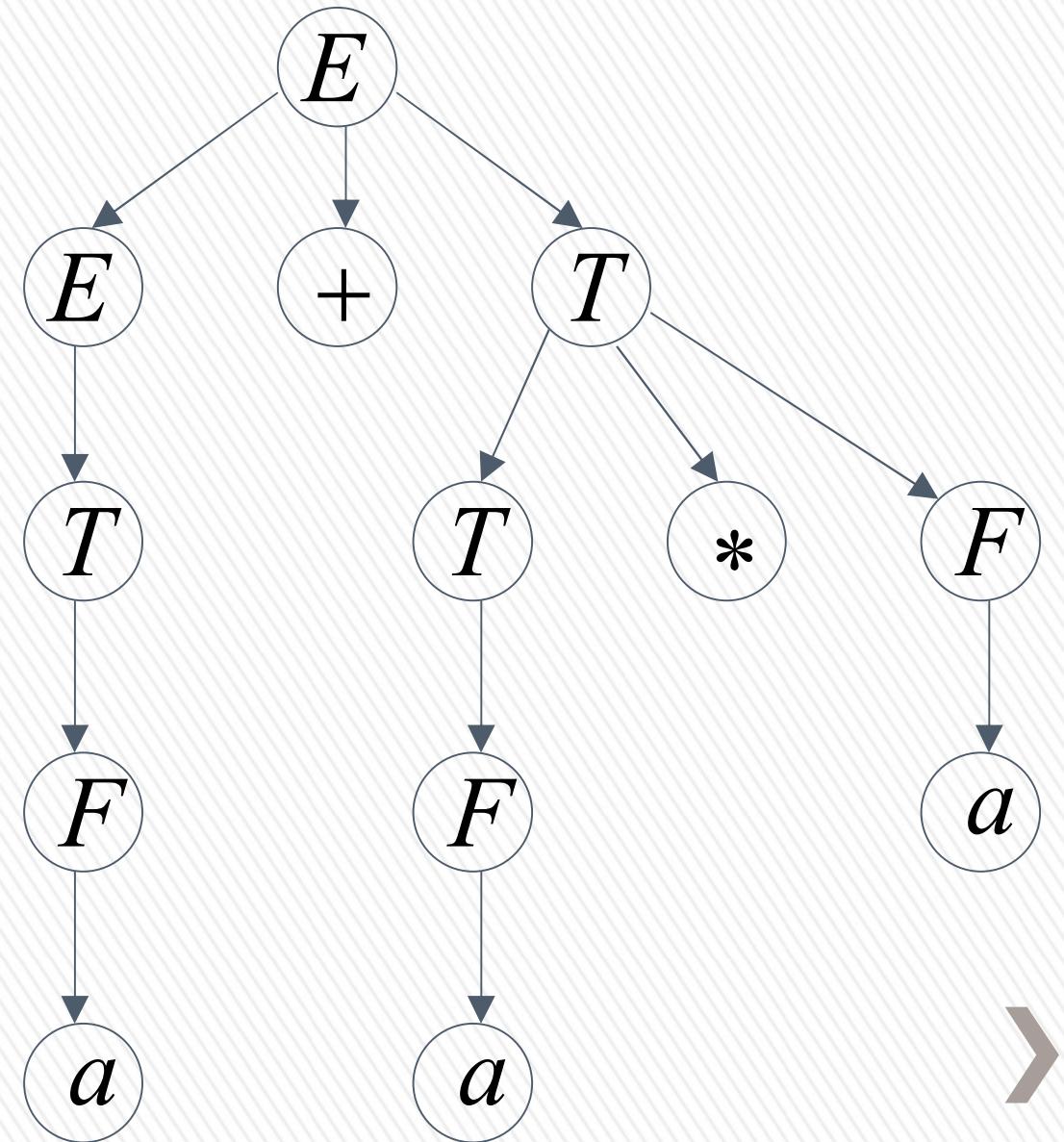
$$\begin{array}{l} E \rightarrow E + T \mid T \\ \hline T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid a \end{array}$$

generates the same language ➤

$$\begin{aligned}
 E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F \\
 &\Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a
 \end{aligned}$$

$$\begin{aligned}
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow (E) \mid a
 \end{aligned}$$

Unique
derivation tree
for $a + a * a$



An un-successful example:

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$$

$$n, m \geq 0$$

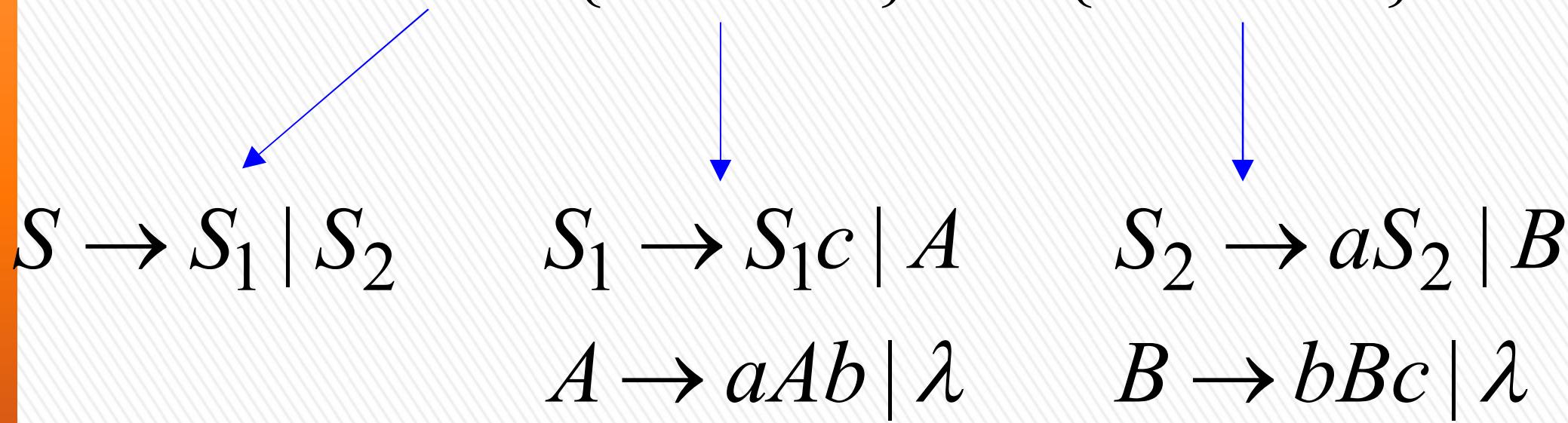
L is inherently ambiguous:

every grammar that generates this language is ambiguous



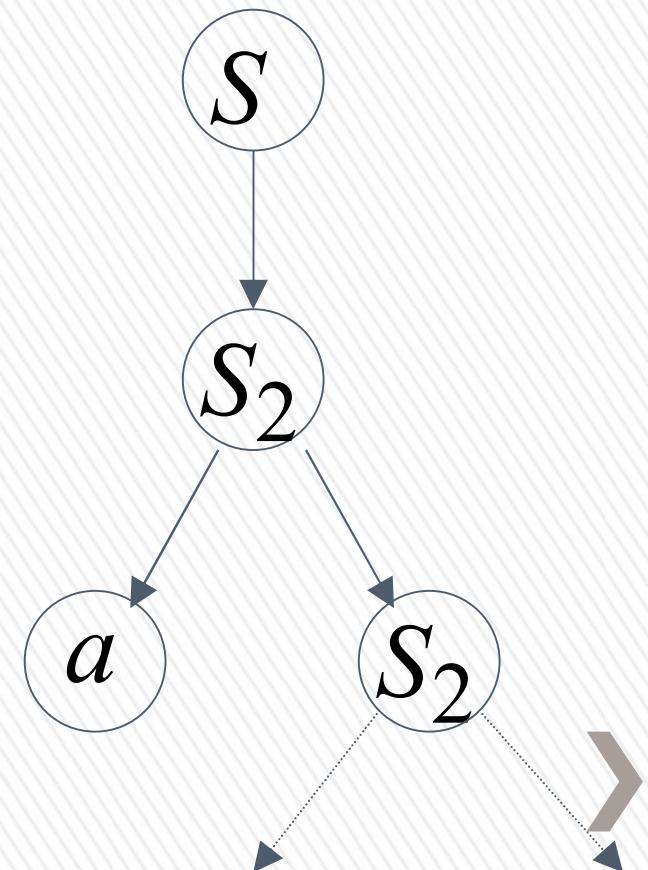
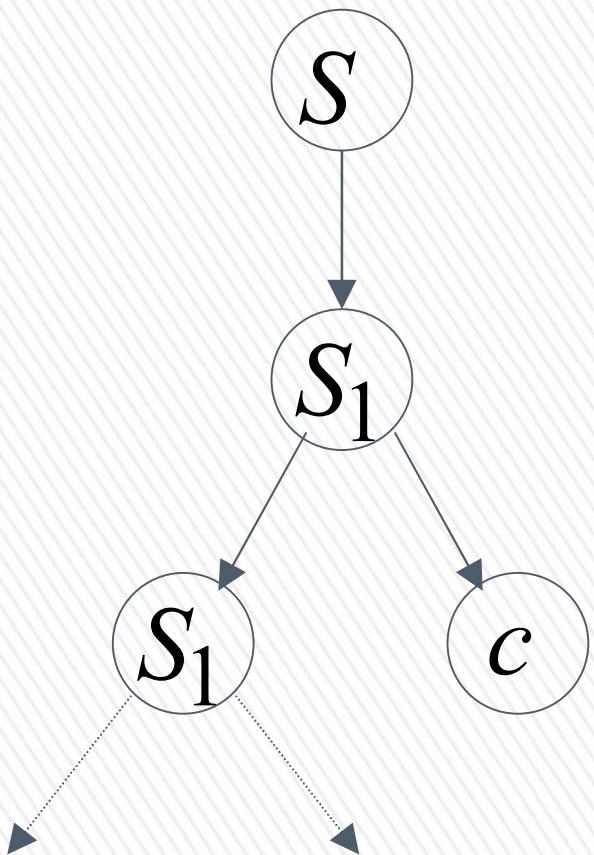
Example (ambiguous) grammar for L :

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$$



The string $a^n b^n c^n \in L$
has always two different derivation trees
(for any grammar)

For example



BLM2502 Theory of Computation





BLM2502

Theory of

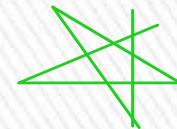
Computation

Spring 2016

BLM2502 Theory of Computation

» Course Outline

- | » Week | Content |
|--------|---|
| » 1 | Introduction to Course |
| » 2 | Computability Theory, Complexity Theory, Automata Theory, Set Theory, Relations, Proofs, Pigeonhole Principle |
| » 3 | Regular Expressions |
| » 4 | Finite Automata |
| » 5 | Deterministic and Nondeterministic Finite Automata |
| » 6 | Epsilon Transition, Equivalence of Automata |
| » 7 | Pumping Theorem |
| » 8 | April 10 - 14 week is the first midterm week |
| » 9 | Context Free Grammars |
| » 10 | Parse Tree, Ambiguity, |
| » 11 | Pumping Theorem |
| » 12 | Turing Machines, Recognition and Computation, Church-Turing Hypothesis |
| » 13 | Turing Machines, Recognition and Computation, Church-Turing Hypothesis |
| » 14 | May 22 – 27 week is the second midterm week |
| » 15 | Review |
| » 16 | Final Exam date will be announced |



The Pumping Lemma for **CFL's**

Pumping Lemma

- » Recall the pumping lemma for regular languages.
- » It told us that if there was a string long enough to cause a **cycle** in the DFA for the language, then we could "pump" the cycle and discover an infinite sequence of strings that had to be in the language.
- » For CFL's the situation is a little more complicated.
String in 2 lines is distinguishable under pump
- » We can always find two pieces of any sufficiently long string to "pump" in tandem.
accept LMB 2 words



Statement of the CFL Pumping Lemma

$$(2) \quad z \in L$$

- » For every context-free language L There is an integer n , such that For every string z in L of length $> n$ There exists $z = uvwxy$ such that:

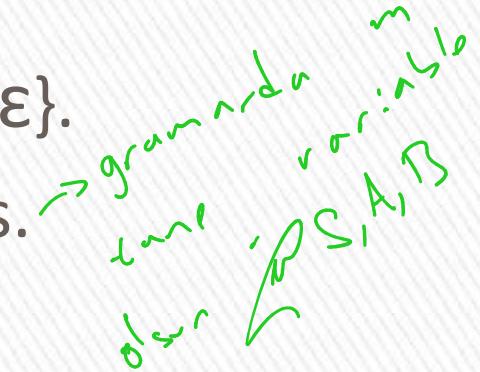
1. $|vwx| < n.$
 2. $|vx| > 0.$
 3. For all $i > 0$, $uv^iwx^i y$ is in L .

is in L.



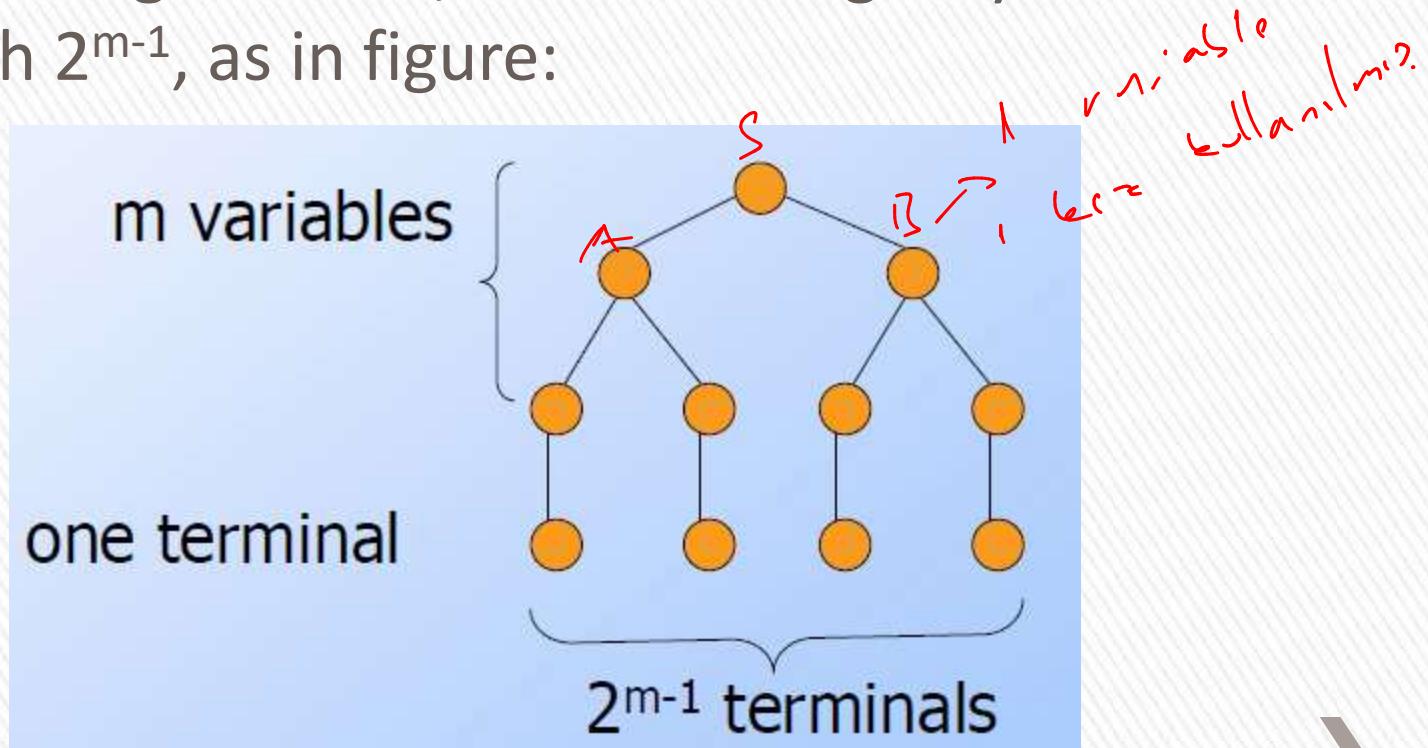
Proof of the Pumping Lemma

- » Start with a CNF grammar for $L - \{\epsilon\}$.
- » Let the grammar have m variables.
 - > Pick $n = 2^m$.
 - > Let $|z| > n$.
- » We claim ("Lemma 1") that a parse tree with yield z must have a path of length $m+2$ or more.



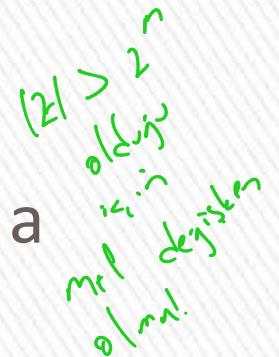
Proof of Lemma 1

- » If all paths in the parse tree of a CNF grammar are of length $< m+1$, then the longest yield has length 2^{m-1} , as in figure:

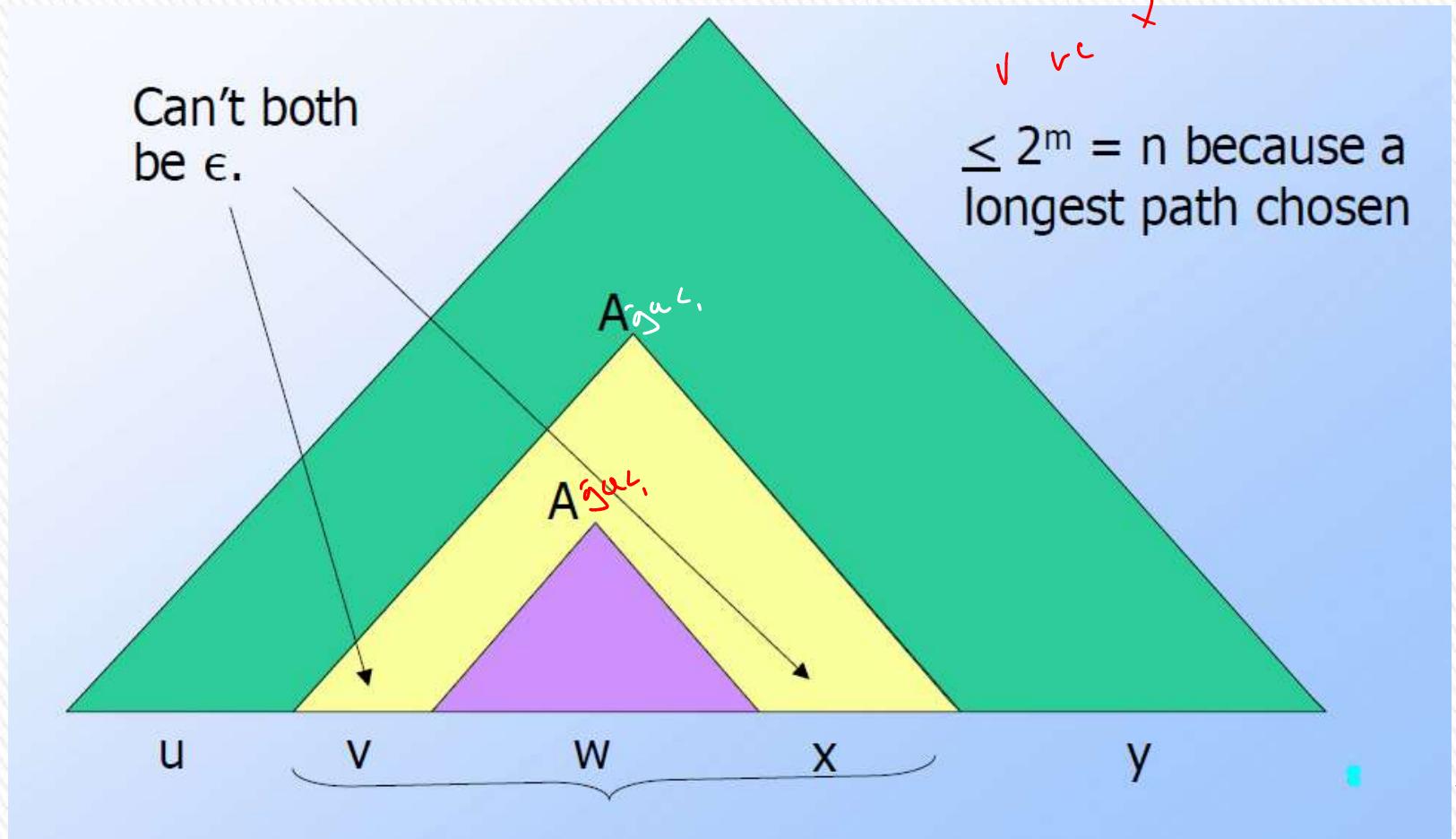


Proof of the Pumping Lemma

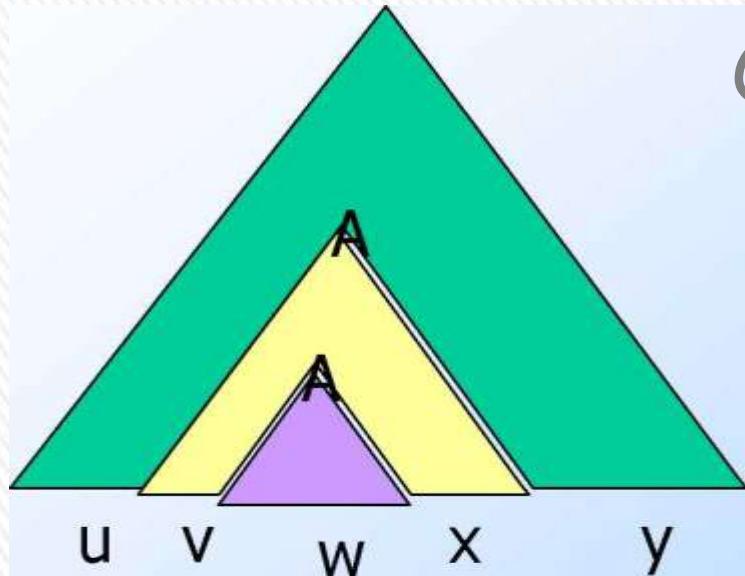
- » Now we know that the parse tree for z has a path with at least $m+1$ variables.
- » Consider some longest path.
- » There are only m different variables, so among the lowest $m+1$ we can find two nodes with the same label, say A .
- » The parse tree thus looks like:



Parse Tree in the Pumping-Lemma



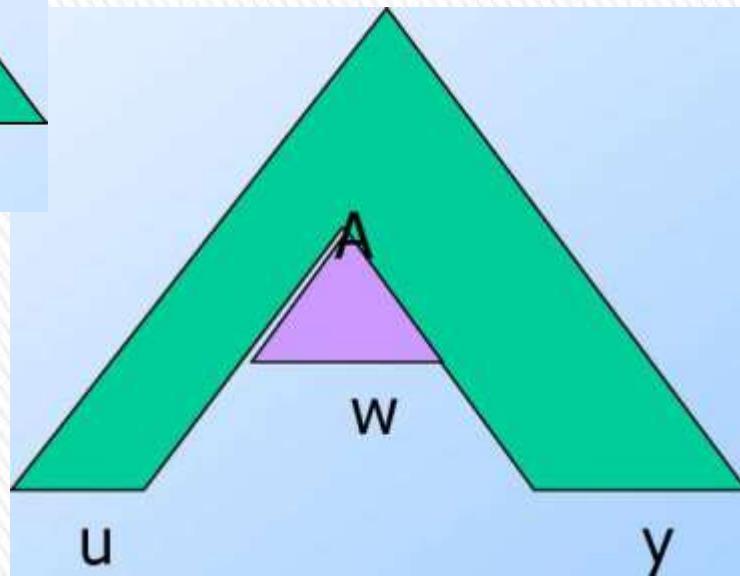
Pumping



Once

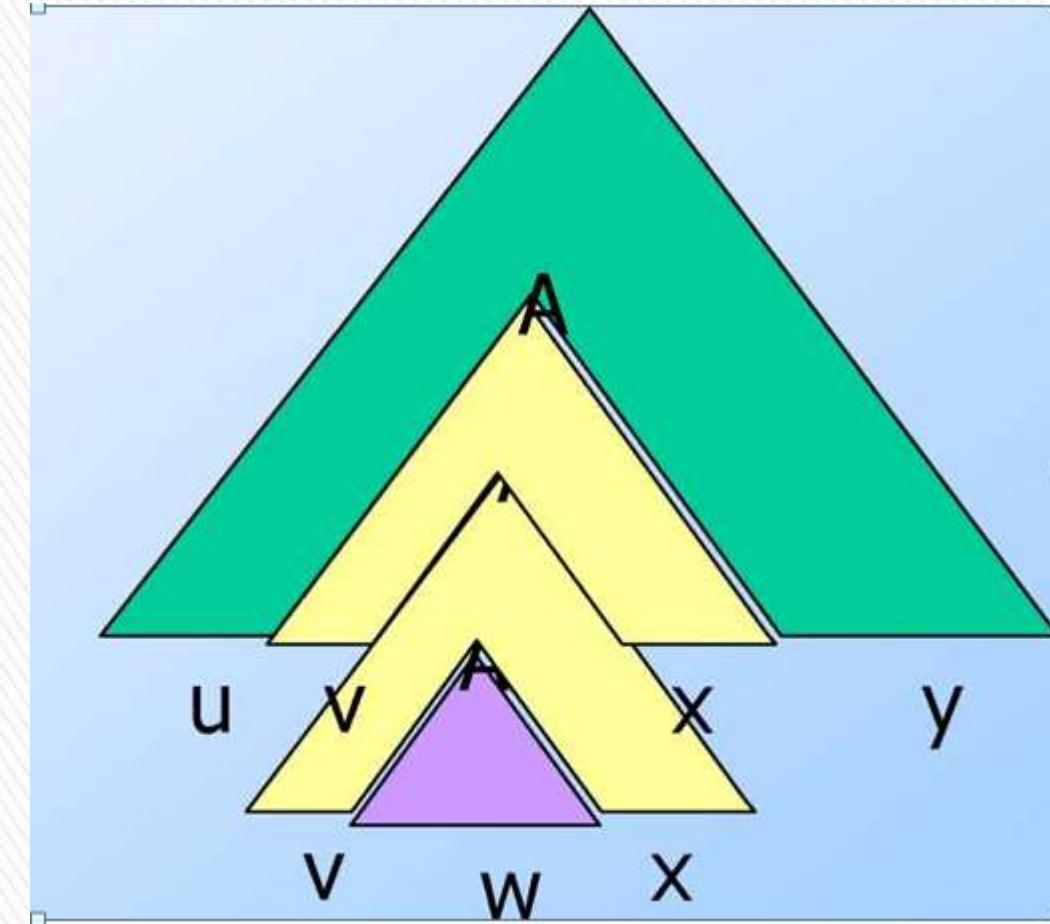
$CFL^{'e}$
 $a \rightarrow e$

Zero times



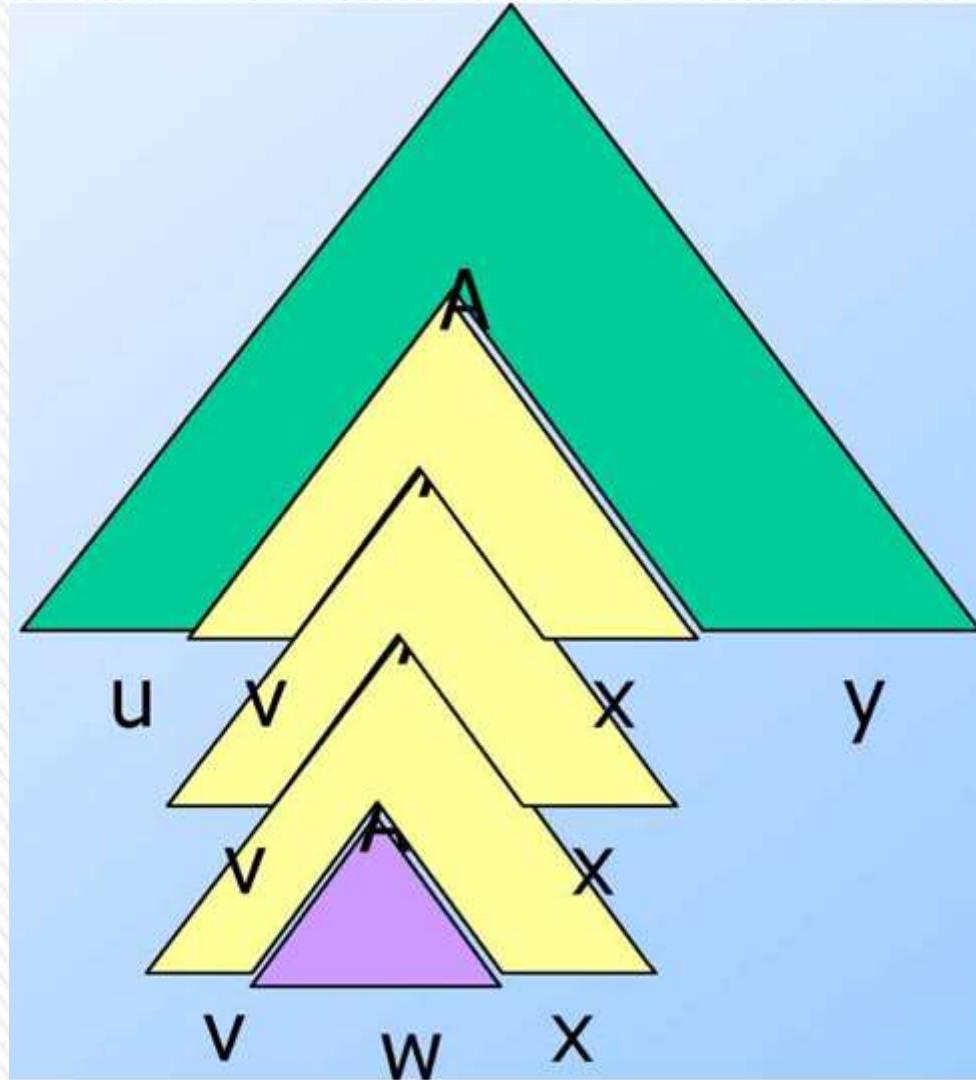
Pumping

Twice



Pumping

Thrice, ...



Using the Pumping Lemma

- » Non-CFL's typically involve trying to match two pairs of counts or match two strings.
- » Example: pumping lemma can be used to show that $L = \{ww \mid w \text{ in } (0+1)^*\}$ is not a CFL.
- » $\{0^i 1 0^i \mid i > 1\}$ is a CFL.
 - > We can match one pair of counts.

$\cup \textcircled{w} \textcircled{x} \textcircled{y}$

$\cup v^n w x^n y$

$a b^n c b^n \not\models$

$\underline{a b^{n+k} c b^n} \rightarrow$

Combining grammar
on \sim delay
CFL reg.
push pop



Using the Pumping Lemma

- » $L = \{0^i 1 0^i 1 0^i \mid i > 1\}$ is not a CFL
 - > We can't match two pairs, or three counts as a group.
 - > Proof using the pumping lemma.
 - > Suppose L were a CFL.
 - > Let n be L 's pumping-lemma constant.
 - > Consider $z = 0^n 1 0^n 1 0^n$.
 - > We can write $z = uvwxy$, where $|vwxy| < n$, and $|vx| > 1$.
 - > Case 1: vx has no 0's.
 - > Then at least one of them is a 1, and uw has at most one 1, which no string in L does.



Using the Pumping Lemma

- » Still considering $z = 0^n 1 0^n 1 0^n$.
- » Case 2: vx has at least one 0.
- » vwx is too short ($\text{length} < n$) to extend to all three blocks of 0's in $0^n 1 0^n 1 0^n$.
- » Thus, uwv has at least one block of n 0's, and at least one block with fewer than n 0's.
- » Thus, uwv is not in L .

↙ ↘ ↙ ↘ ↗





BLM2502

Theory of

Computation

Spring 2016

BLM2502 Theory of Computation

» Course Outline

- | » Week | Content |
|--------|---|
| » 1 | Introduction to Course |
| » 2 | Computability Theory, Complexity Theory, Automata Theory, Set Theory, Relations, Proofs, Pigeonhole Principle |
| » 3 | Regular Expressions |
| » 4 | Finite Automata |
| » 5 | Deterministic and Nondeterministic Finite Automata |
| » 6 | Epsilon Transition, Equivalence of Automata |
| » 7 | Pumping Theorem |
| » 8 | April 10 - 14 week is the first midterm week |
| » 9 | Context Free Grammars |
| » 10 | Parse Tree, Ambiguity, |
| » 11 | Pumping Theorem |
| » 12 | Turing Machines, Recognition and Computation, Church-Turing Hypothesis |
| » 13 | Turing Machines, Recognition and Computation, Church-Turing Hypothesis |
| » 14 | May 22 – 27 week is the second midterm week |
| » 15 | Review |
| » 16 | Final Exam date will be announced |



The Pumping Lemma for CFL's



Simplifications of Context-Free Grammars

A Substitution Rule

$S \rightarrow \cancel{aB} \mid ab \mid aaA$

$A \rightarrow aaA$

$A \rightarrow ab\cancel{Bc}$

$\cancel{B} \rightarrow aA$

$\cancel{B} \rightarrow b$

Substitute
 $\cancel{abbc} \mid \cancel{ab} \cancel{a} \cancel{Ac}$

Equivalent
grammar

$S \rightarrow \cancel{aB} \mid ab$

$A \rightarrow aaA$

$A \rightarrow abBc \mid abbc$

$B \rightarrow aA$



$$S \rightarrow aB \mid ab$$
$$A \rightarrow aaA$$
$$A \rightarrow abBc \mid abbc$$
$$B \rightarrow aA$$

Substitute

$$B \rightarrow aA$$
$$S \rightarrow \cancel{aB} \mid ab \mid aaA$$
$$A \rightarrow aaA$$
$$A \rightarrow \cancel{abBc} \mid abbc \mid abaAc$$

Equivalent
grammar ➤

In general: $A \rightarrow xBz$

$B \rightarrow y_1$

Substitute

$B \rightarrow y_1$

$A \rightarrow xBz \mid xy_1z$

equivalent
grammar ➤

Nullable Variables

ϵ – production :

$$X \rightarrow \epsilon$$

Nullable Variable:

$$Y \Rightarrow \dots \Rightarrow \epsilon$$

Example:

$$S \rightarrow aMb \quad | \text{ab}$$

$$M \rightarrow aMb \quad | \text{ab}$$

$$\cancel{M} \rightarrow \epsilon$$

Nullable variable

ϵ – production



Removing ε – productions

$$S \rightarrow aMb$$

$$M \rightarrow aMb$$

~~$$M \rightarrow \varepsilon$$~~

|
ab

Substitute

$$M \rightarrow \varepsilon$$

$$S \rightarrow aMb | ab$$

$$M \rightarrow aMb | ab$$

After we remove all the ε - productions
all the nullable variables disappear
(except for the start variable)



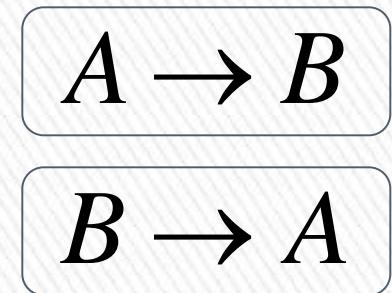
Unit-Productions

Unit Production: $X \rightarrow Y$

(a single variable in both sides)

Example: $S \rightarrow aA$

$A \rightarrow a$



Unit Productions

$B \rightarrow bb$



Removal of unit productions:

$S \rightarrow aA$ | ~~a~~ bB | ab | aa | ad

$A \rightarrow a$

~~$A \rightarrow B$~~

$B \rightarrow A$ | a

~~$B \rightarrow bb$~~

Substitute

$A \rightarrow B$

$S \rightarrow aA$ | aB

$A \rightarrow a$

$B \rightarrow A$ | B

$B \rightarrow bb$



Unit productions of form $X \rightarrow X$
can be removed immediately

$$S \rightarrow aA \mid aB$$

$$A \rightarrow a$$

$$B \rightarrow A \mid \cancel{B}$$

$$B \rightarrow bb$$

Remove

$$B \rightarrow B$$

$$S \rightarrow aA \mid aB$$

$$A \rightarrow a$$

$$B \rightarrow A$$

$$B \rightarrow bb$$

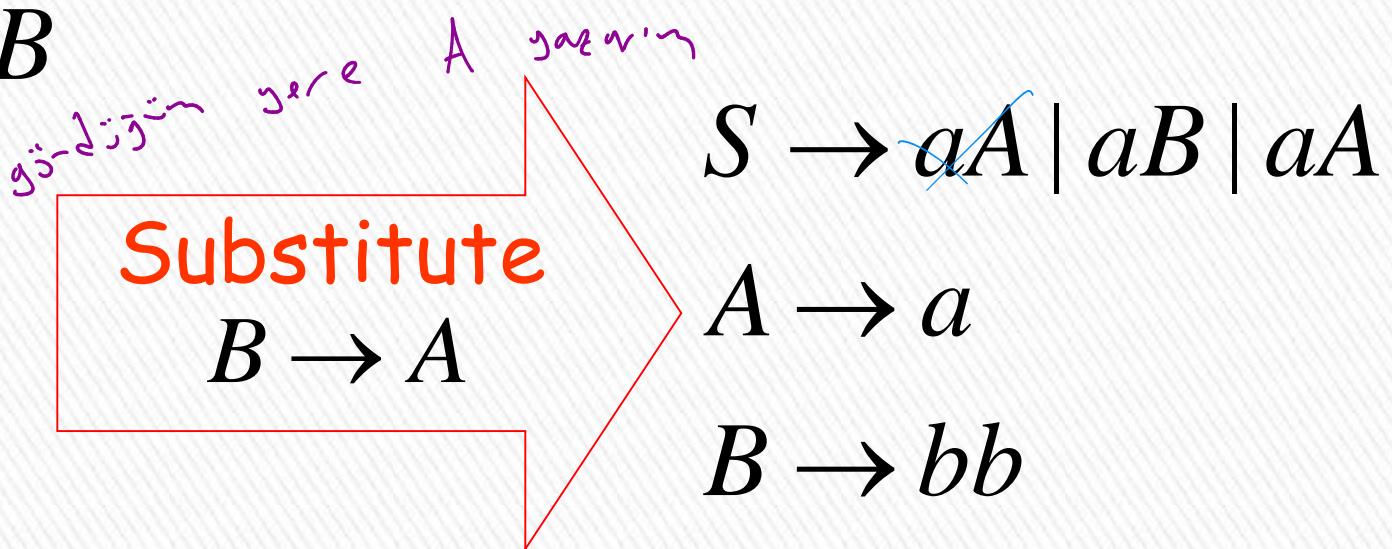


$$S \rightarrow aA \mid aB$$

$$A \rightarrow a$$

~~$$B \rightarrow A$$~~

$$B \rightarrow bb$$



Remove repeated productions

$$S \rightarrow aA \mid aB \mid \cancel{aA}$$
$$A \rightarrow a$$
$$B \rightarrow bb$$


Final grammar

$$S \rightarrow aA \mid aB$$
$$A \rightarrow a$$
$$\textcircled{B} \rightarrow bb$$


Useless Productions

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

$$S \rightarrow A$$

$A \rightarrow aA$

sonuçlanamaz

Useless Production

Some derivations never terminate...

$$S \Rightarrow A \Rightarrow aA \Rightarrow aaA \Rightarrow \dots \Rightarrow aa\dots aA \Rightarrow \dots$$

Another grammar:

$$S \rightarrow A$$

$$A \rightarrow aA$$

$$A \rightarrow \lambda$$

$$B \rightarrow bA$$

h: a gel magyel egysége?

Useless Production

Not reachable from S



In general:

If there is a derivation

$$S \Rightarrow \dots \Rightarrow xAy \Rightarrow \dots \Rightarrow w \in L(G)$$

consists of
terminals

Then variable A is useful

Otherwise, variable A is useless



A production $A \rightarrow x$ is useless
if any of its variables is useless

$$S \rightarrow aSb$$

$$S \rightarrow \epsilon$$

Variables

$$S \rightarrow A$$

useless

$$A \rightarrow aA$$

useless

$$B \rightarrow C$$

useless

$$C \rightarrow D$$

Productions

useless

useless

useless

useless



Removing Useless Variables and 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 or ϵ (possible useful variables)

$$S \rightarrow aS \mid A \mid C$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

$$C \rightarrow aCb$$

Round 1: $\{A, B\}$

(the right hand side of production that has only terminals)

Round 2: $\{A, B, S\}$

(the right hand side of a production has terminals and variables of previous round)

This process can be generalized

Then, remove productions that use variables other than $\{A, B, S\}$

$$S \rightarrow aS \mid A \mid \cancel{C}$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

~~$$C \rightarrow aCb$$~~



$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

$$B \rightarrow aa$$



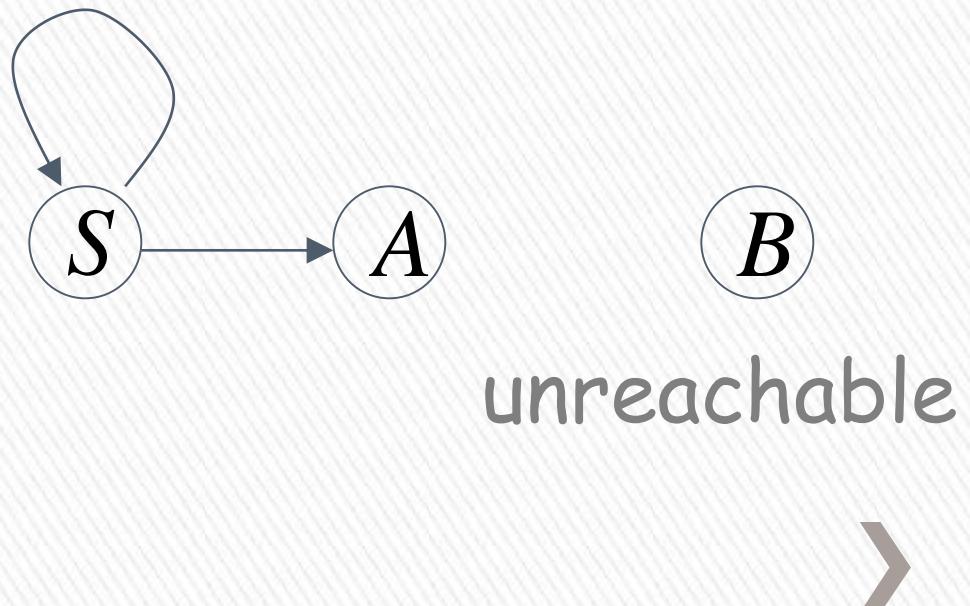
Second: Find all variables
reachable from S

Use a Dependency Graph
where nodes are variables

$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

$$B \rightarrow aa$$



Keep only the variables
reachable from S

Final Grammar

$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

~~$$B \rightarrow aa$$~~



$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

Contains only
useful variables

Removing All

» **Step 1:** Remove Nullable Variables

$$\hookrightarrow A \rightarrow \epsilon$$

» **Step 2:** Remove Unit-Productions

$$\hookrightarrow A \rightarrow B$$

» **Step 3:** Remove Useless Variables

$$\hookrightarrow A \rightarrow A_a$$

This sequence guarantees that unwanted variables and productions are removed



j
 $S \rightarrow ABAc$

$A \rightarrow qA | \epsilon$

$B \rightarrow bB | \epsilon$

$C \rightarrow c$

\vdots

① Remove null productions

$A \rightarrow \Sigma, B \rightarrow \Sigma$

↓

$S \rightarrow ABAC | ABC | BC | C | AAC | AC$

$A \rightarrow qA | q$

$B \rightarrow bB | b$

$\Sigma \cancel{\rightarrow} C$

$S \rightarrow ABAc | ABC | BAc | BC | C | AAC | AC$

$A \rightarrow qA | q$

$B \rightarrow bB | b$

$S \rightarrow ABAc | ABC | BAc | BC | C | AAC | AC$

$A \rightarrow qA | q$

$B \rightarrow bB | b$

\cancel{C}



Normal Forms for Context-free Grammars

Chomsky Normal Form

Each production has form:

$$A \rightarrow BC$$

variable

or

$$A \rightarrow a$$

terminal



variable



Examples:

$$\underline{S \rightarrow AS}$$

$$\underline{S \rightarrow a}$$

$$\underline{A \rightarrow SA}$$

$$\underline{A \rightarrow b}$$

Chomsky

Normal Form

$$\begin{array}{l} S \xrightarrow{\quad} \underline{A V_1} \\ V_1 \rightarrow A S \end{array}$$

$$T_1 \rightarrow q$$

$$\overbrace{S \rightarrow \underline{AS}}$$

$$S \rightarrow \overbrace{\underline{AAS}}$$

$$A \rightarrow SA$$

$$A \rightarrow \underline{aa}$$

$$A \rightarrow \widehat{T_1 T_1}$$

Not Chomsky

Normal Form



Conversion to Chomsky Normal Form

» Example:

$$S \rightarrow A B a$$

$$A \rightarrow a a b$$

$$B \rightarrow A c$$

$$V_1 \rightarrow a$$

$$V_2 \rightarrow B V_1$$

$$V_3 \rightarrow b$$

$$V_4 \rightarrow V_1 V_3$$

$$V_5 \rightarrow c$$

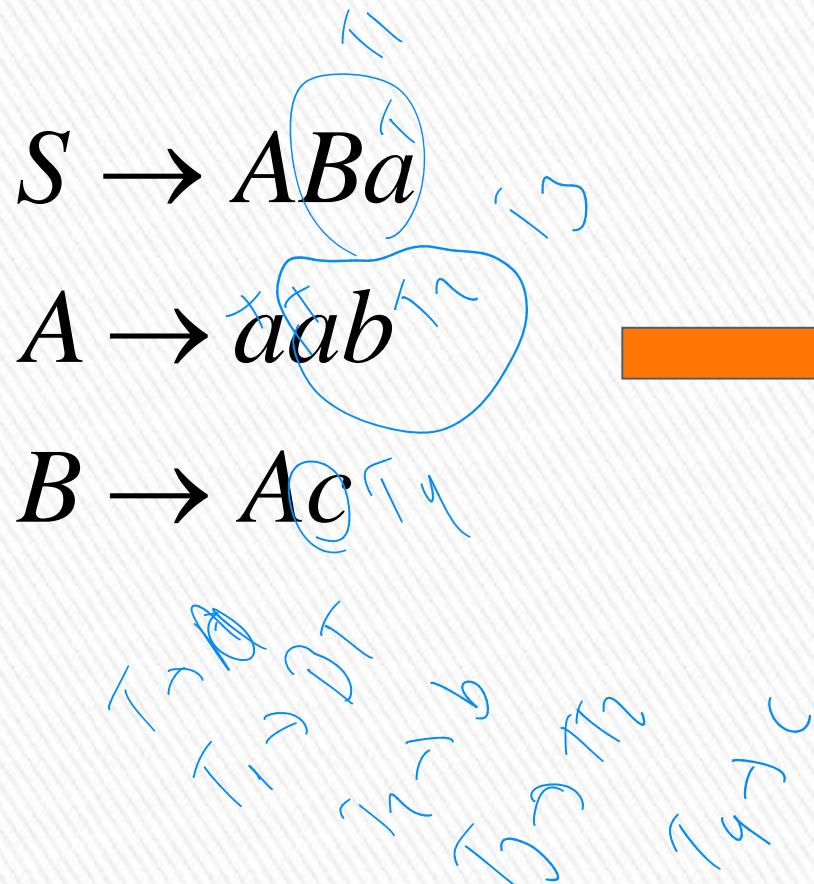
Not in Chomsky
Normal Form

We will convert it to Chomsky Normal Form



Introduce new variables for the terminals:

$$T_a, T_b, T_c$$



Introduce new intermediate variable V_1
to break first production:

$$S \rightarrow A \underbrace{BT_a}_c$$

$$A \rightarrow T_a \underbrace{T_a T_b}_c$$

$$B \rightarrow \underbrace{AT_c}_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$



$$\begin{aligned} S &\rightarrow A \underbrace{V_1}_c \\ V_1 &\rightarrow BT_a \end{aligned}$$

$$A \rightarrow T_a T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$



Introduce intermediate variable: V_2

$$S \rightarrow AV_1$$

$$V_1 \rightarrow BT_a$$

$$A \rightarrow T_a T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$

$$S \rightarrow AV_1$$

$$V_1 \rightarrow BT_a$$

$$A \rightarrow T_a V_2$$

$$V_2 \rightarrow T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$



Final grammar in Chomsky Normal Form:

$$S \rightarrow AV_1$$

$$V_1 \rightarrow BT_a$$

$$A \rightarrow T_a V_2$$

$$V_2 \rightarrow T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$

Initial grammar

$$S \rightarrow ABa$$

$$A \rightarrow aab$$

$$B \rightarrow Ac$$



In general:

From any context-free grammar
(which doesn't produce ϵ)
not in Chomsky Normal Form

we can obtain:

an equivalent grammar
in Chomsky Normal Form



The Procedure

First remove:

Nullable variables

Unit productions

(Useless variables optional)



Then, for every symbol a :

New variable: T_a

Add production $T_a \rightarrow a$

In productions with length at least 2
replace a with T_a

Productions of form $A \rightarrow a$
do not need to change!



Replace any production $A \rightarrow C_1C_2\cdots C_n$

with $A \rightarrow C_1V_1$

$V_1 \rightarrow C_2V_2$

...

$V_{n-2} \rightarrow C_{n-1}C_n$

New intermediate variables: V_1, V_2, \dots, V_{n-2} ➤

Observations

- Chomsky normal forms are good for parsing and proving theorems
- It is easy to find the Chomsky normal form for any context-free grammar



Greinbach Normal Form

All productions have form:

$$A \rightarrow a V_1 V_2 \cdots V_k \quad k \geq 0$$

symbol

variables



Examples:

$$S \rightarrow \underline{cAB}$$

$$A \rightarrow aA \mid bB \mid b$$

$$B \rightarrow b$$

Greinbach
Normal Form

b ✓

$$T_1 = b$$

$$S \rightarrow aT_1 ST_1$$

$$S \rightarrow ab \cancel{S} b$$

$$S \rightarrow \cancel{aa}$$

$$T_2 = a$$

$$S \rightarrow aT_2$$

Not Greinbach
Normal Form



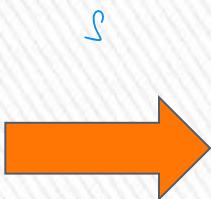
Conversion to Greinbach Normal Form:

$V_1 \quad V_2$

$$S \rightarrow ab\cancel{S}b$$
$$S \rightarrow aa$$

$V_1 \Rightarrow b$

$V_2 \Rightarrow a$


$$S \rightarrow aT_b S T_b$$
$$S \rightarrow a\underline{T_a}$$
$$T_a \rightarrow a$$
$$T_b \rightarrow b$$

Greinbach
Normal Form ➤

Observations

- Greinbach normal forms are very good for parsing strings (better than Chomsky Normal Forms)
- However, it is difficult to find the Greinbach normal of a grammar



BLM2502 Theory of Computation





BLM2502

Theory of Computation

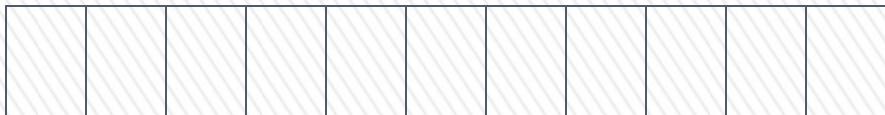


Pushdown Automata

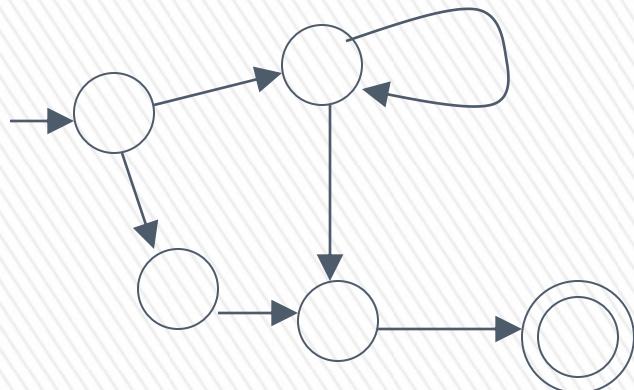
PDA

Pushdown Automaton -- PDA

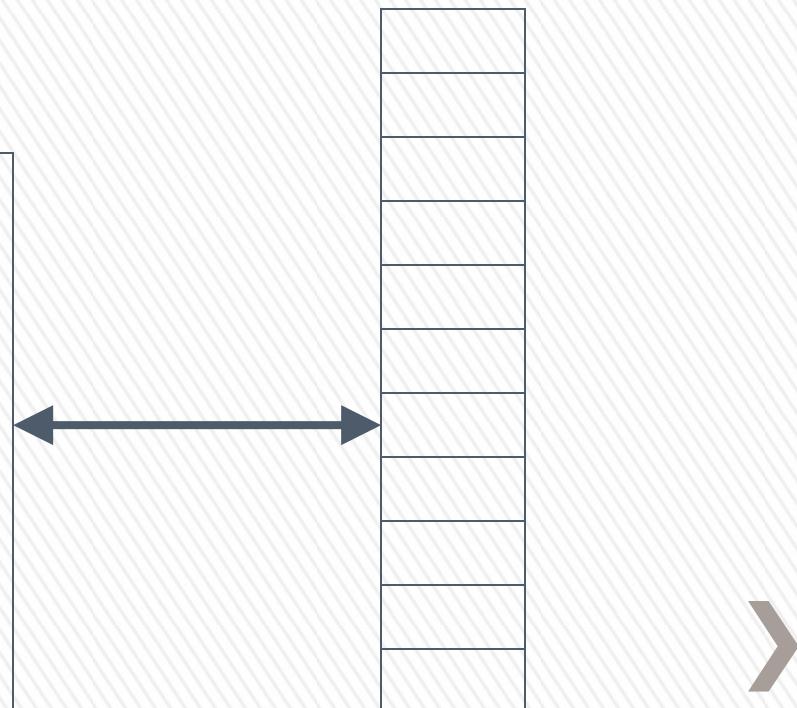
Input String



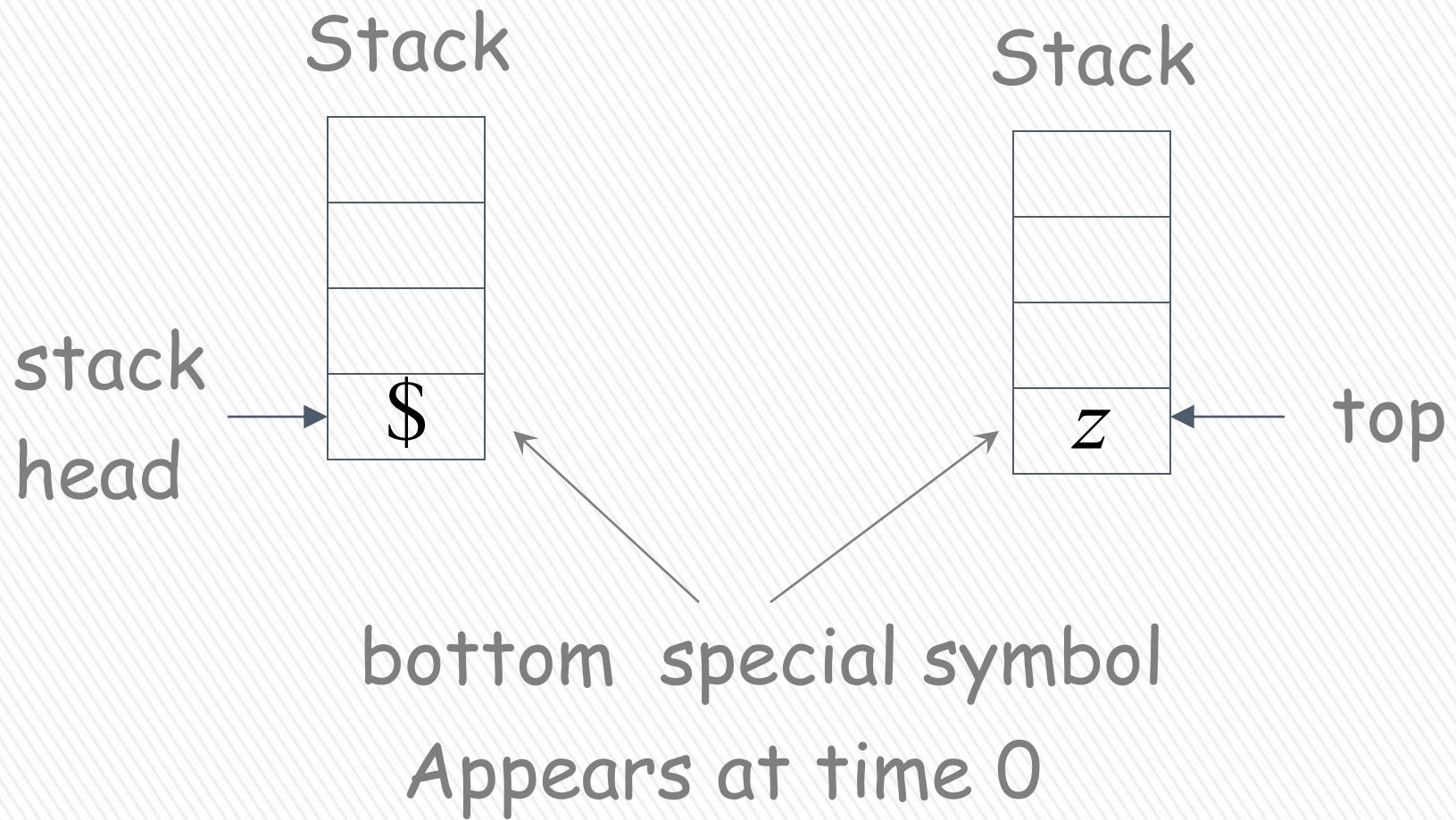
States



Stack



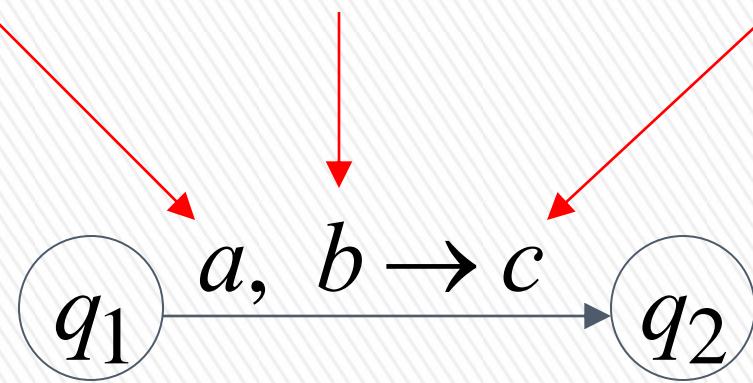
Initial Stack Symbol



Input
symbol

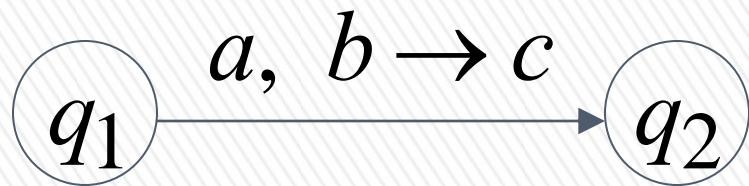
Pop
symbol

Push
symbol

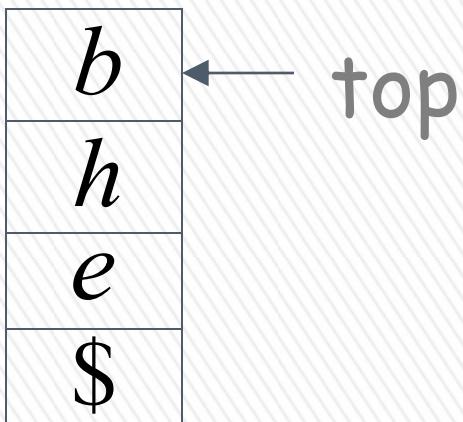


The States

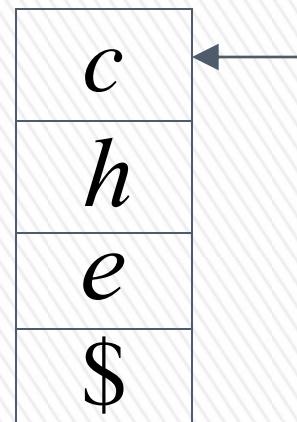


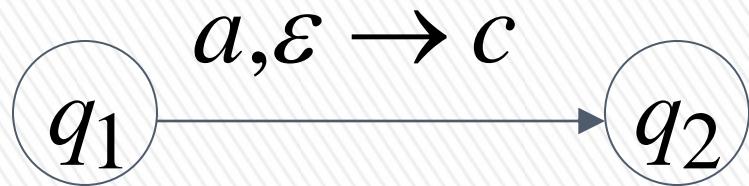


stack



Replace

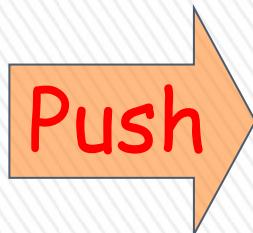




stack

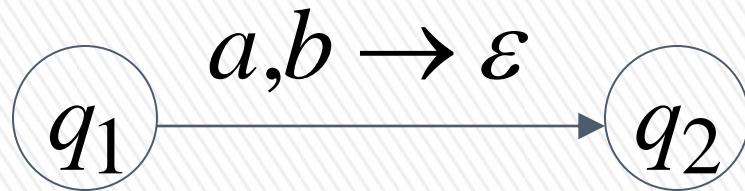
b
h
e
$\$$

top



c
b
h
e
$\$$





stack

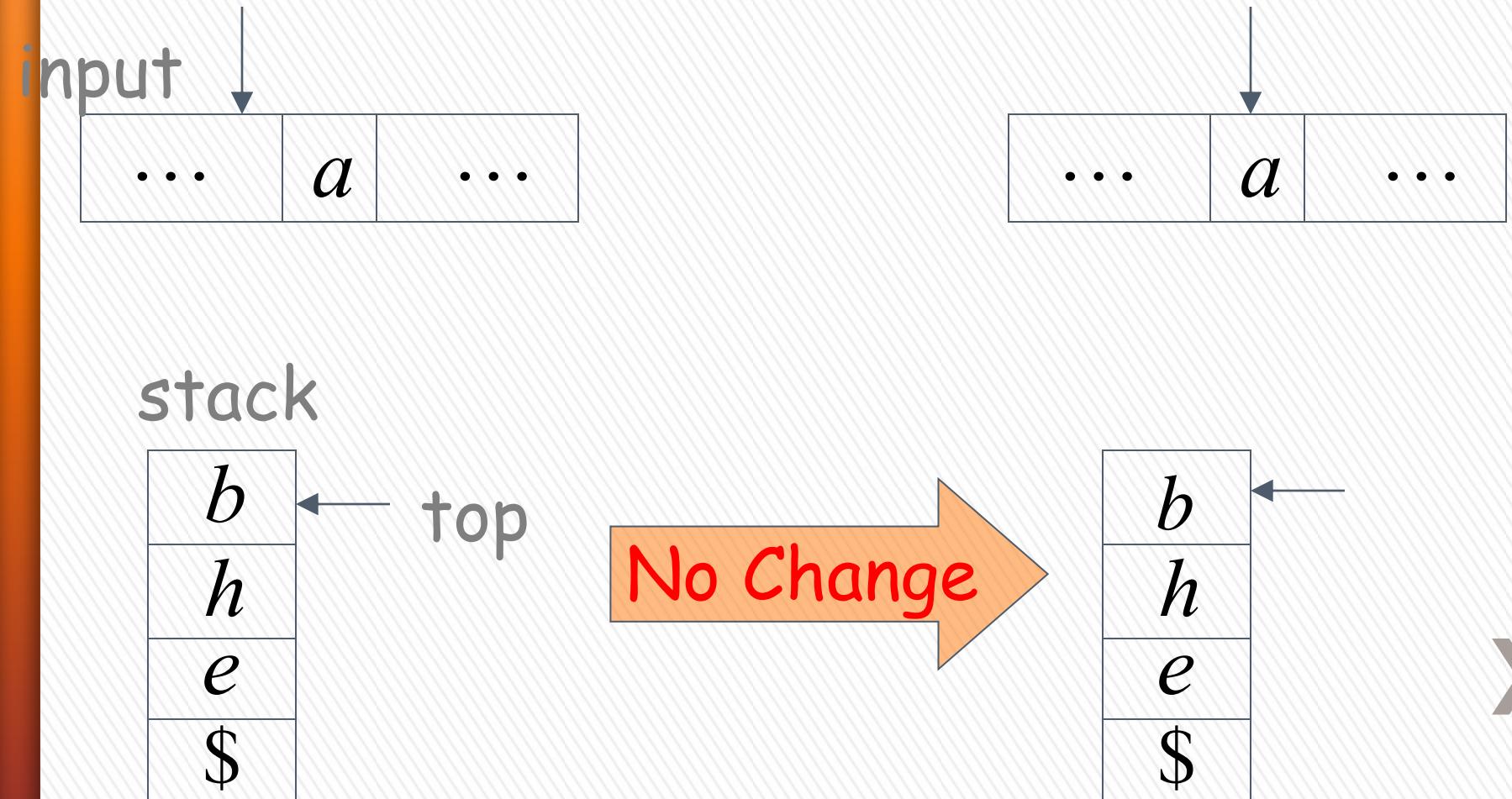
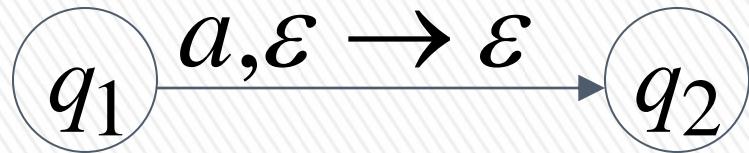
b
h
e
$\$$

top



h
e
$\$$



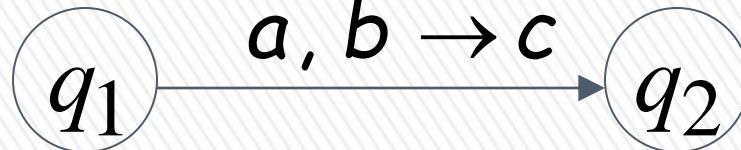


Pop from Empty Stack

input



stack



Pop

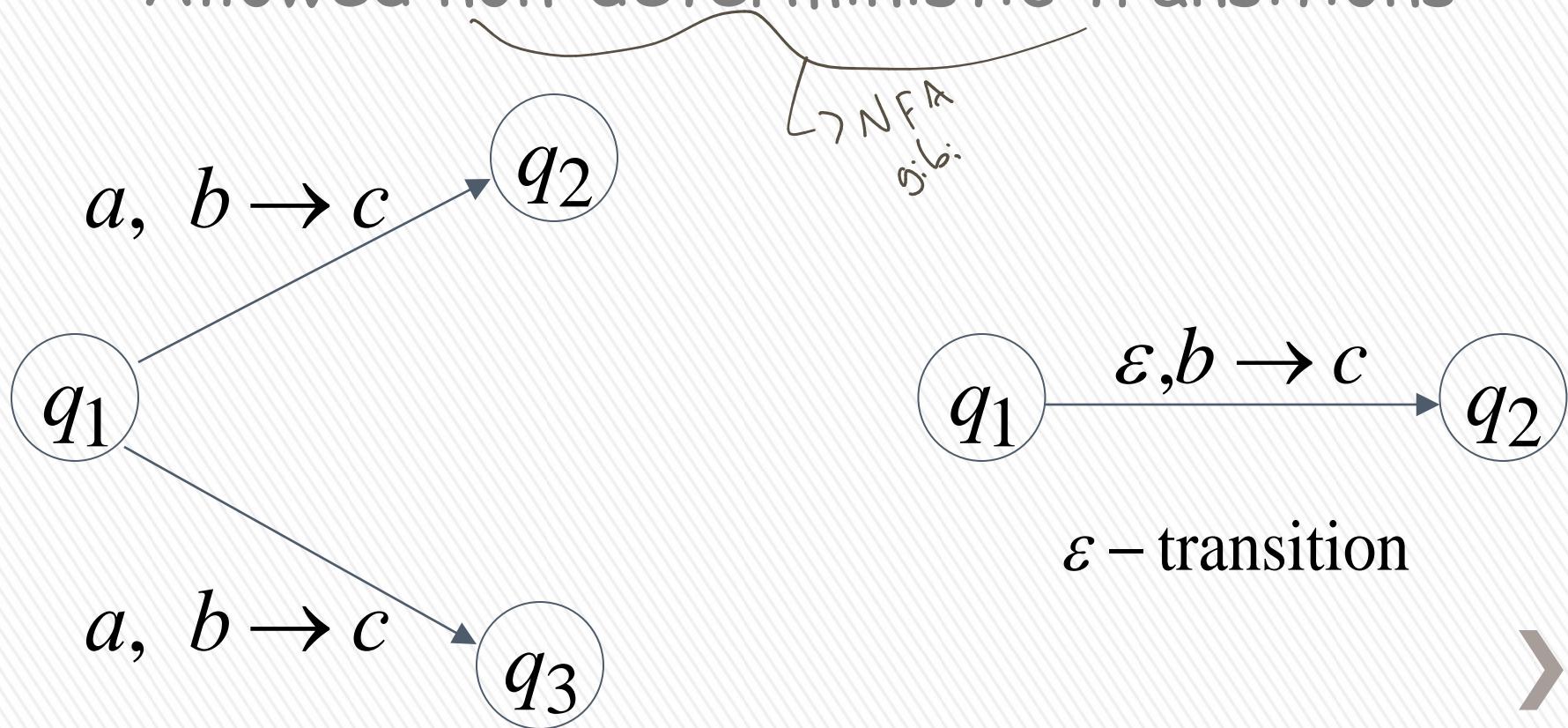
Automaton halts!

If the automaton attempts to pop from empty stack then it halts and rejects input

Non-Determinism

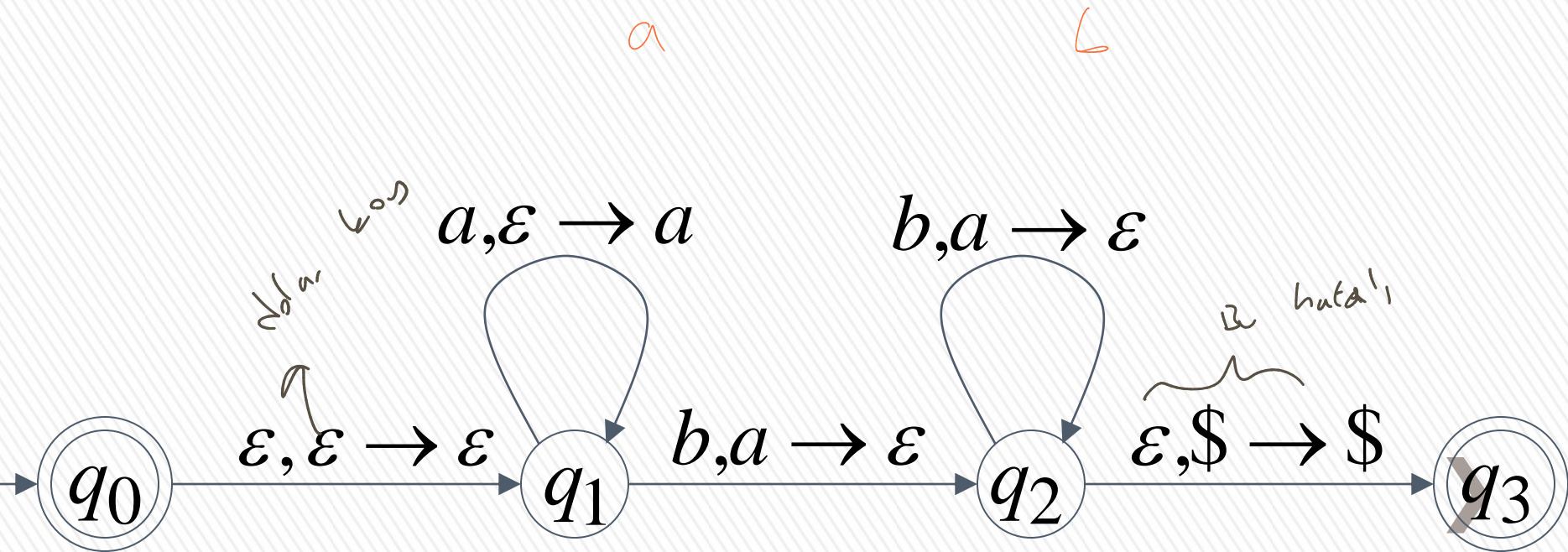
PDAs are non-deterministic

Allowed non-deterministic transitions



Example PDA

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



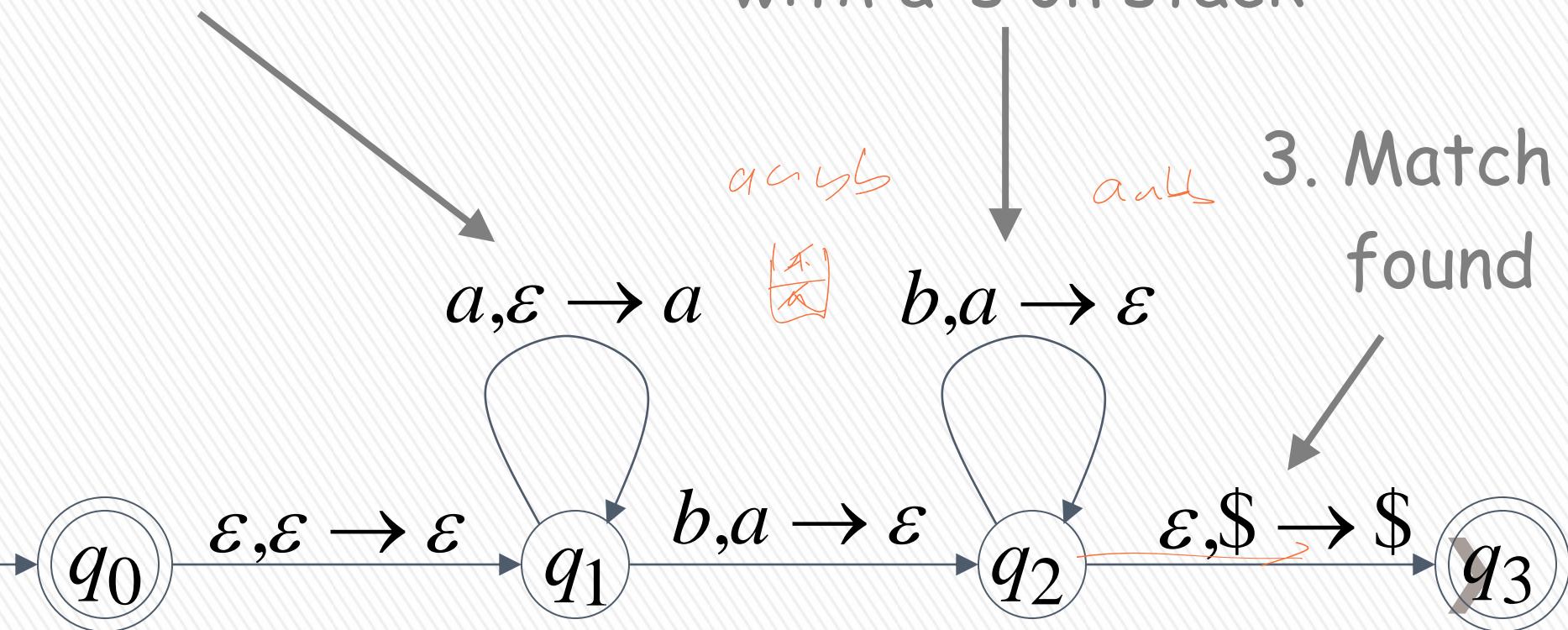
$$L(M) = \{a^n b^n : n \geq 0\}$$

Basic Idea:

1. Push the a's
on the stack

2. Match the b's on input
with a's on stack

3. Match
found



Execution Example: Time 0

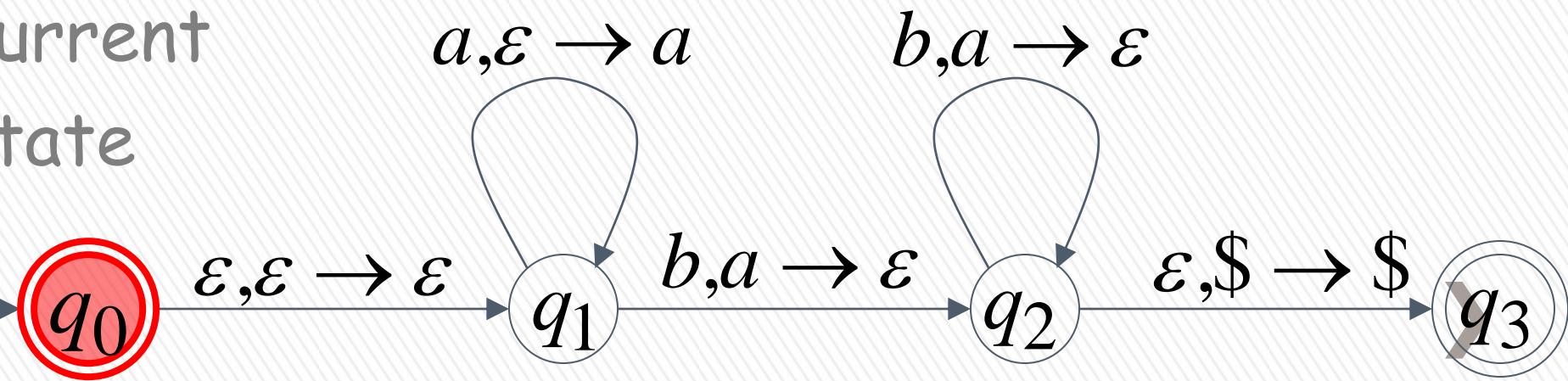
Input

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



Stack

current
state



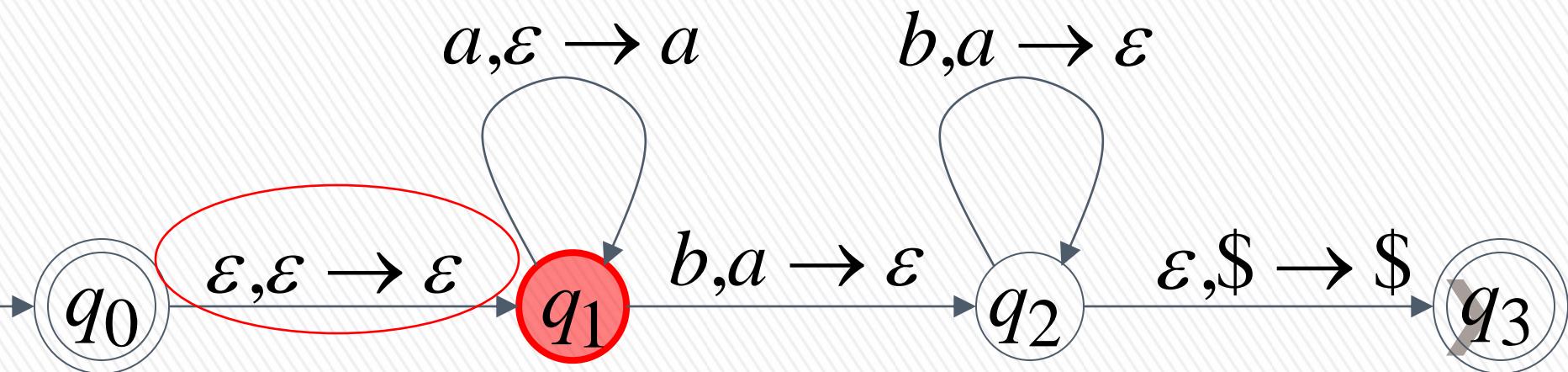
Time 1

Input

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



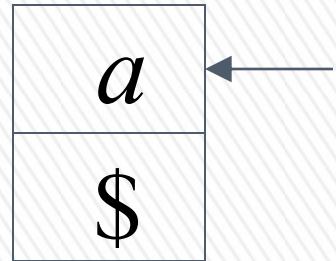
Stack



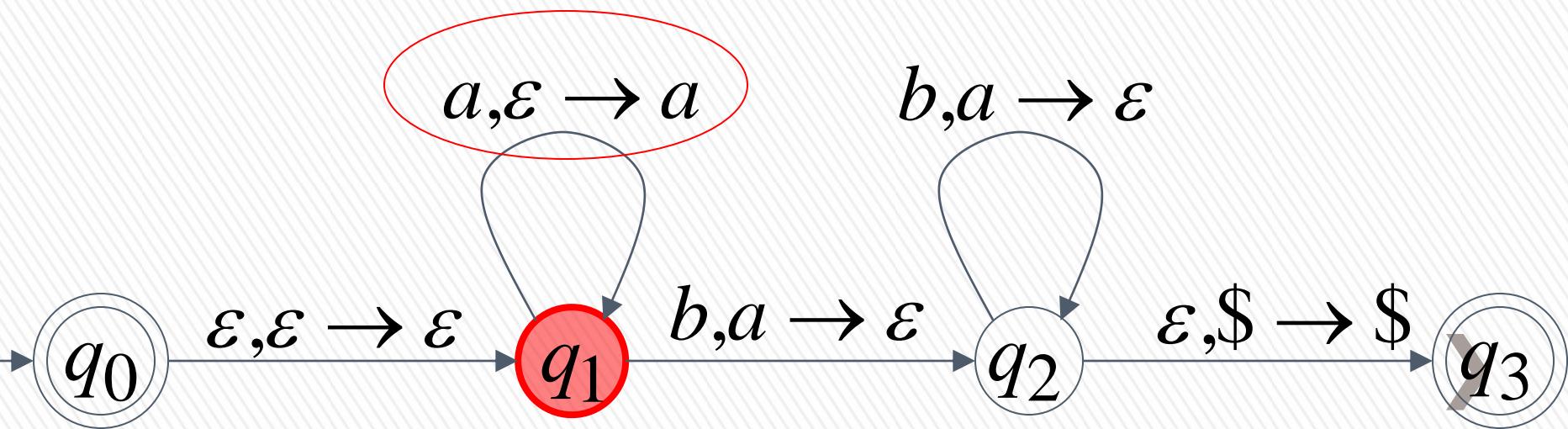
Time 2

Input

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



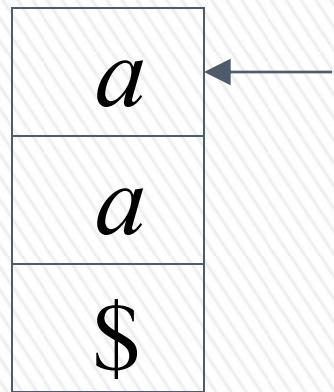
Stack



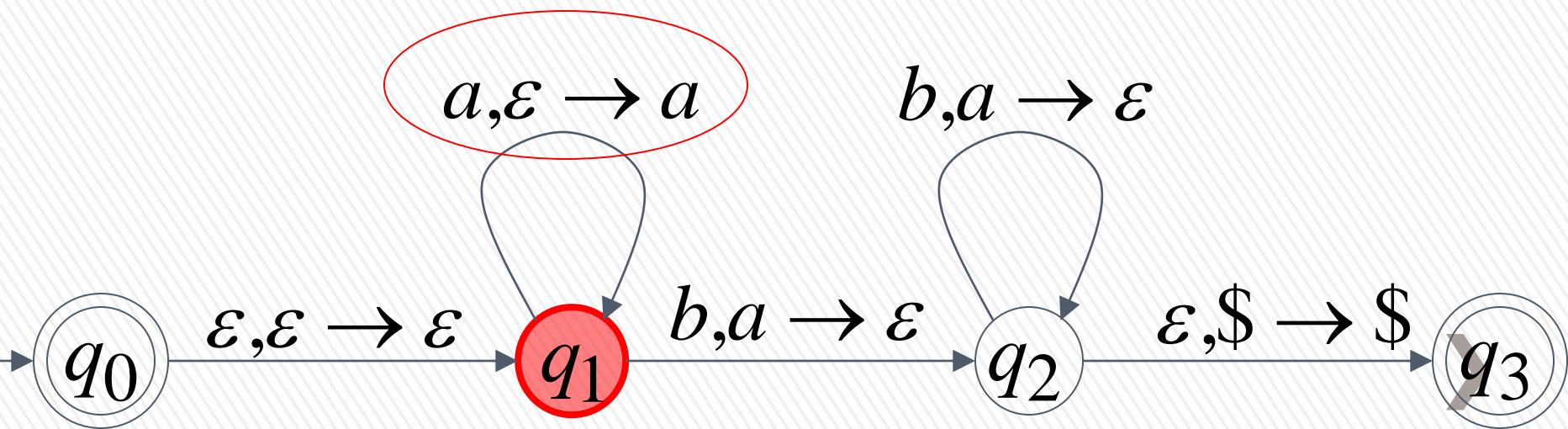
Time 3

Input

a	a	a	b	b	b



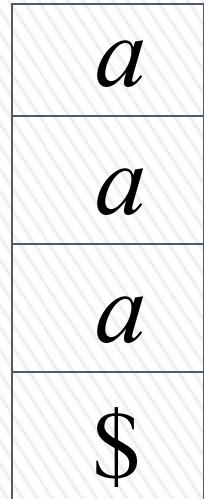
Stack



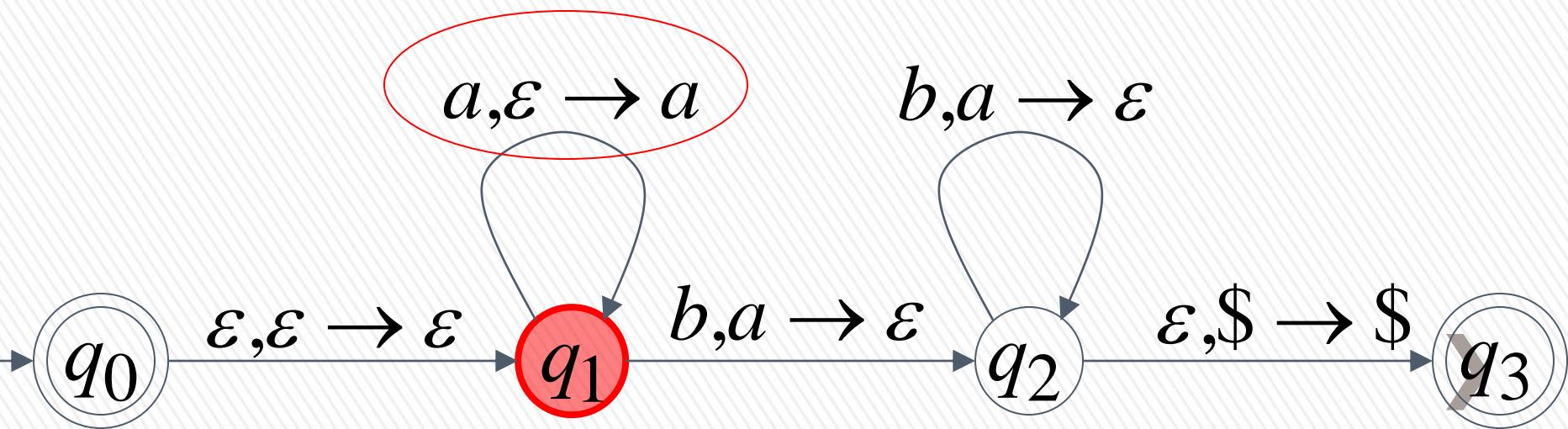
Time 4

Input

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



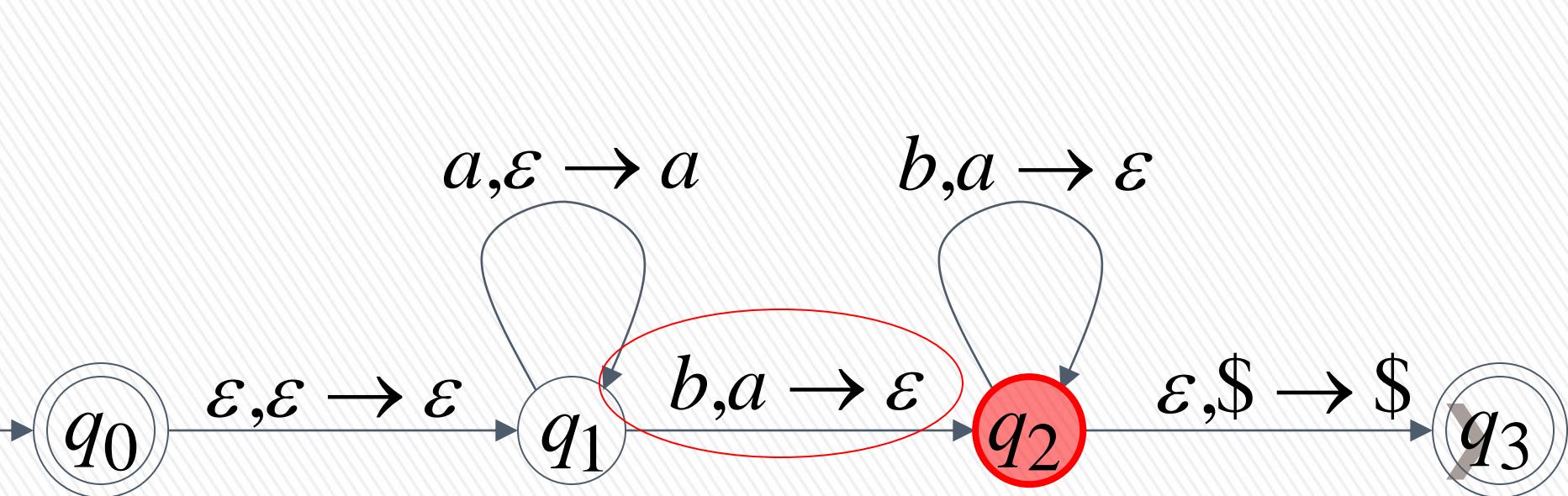
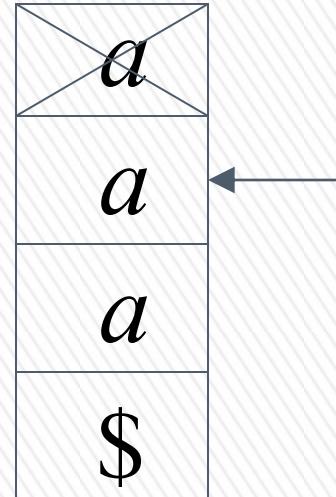
Stack



Time 5

Input

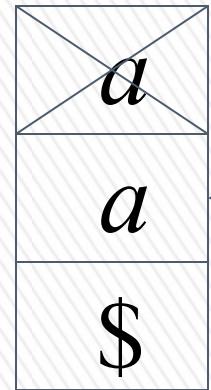
a	a	a	b	b	b
---	---	---	---	---	---



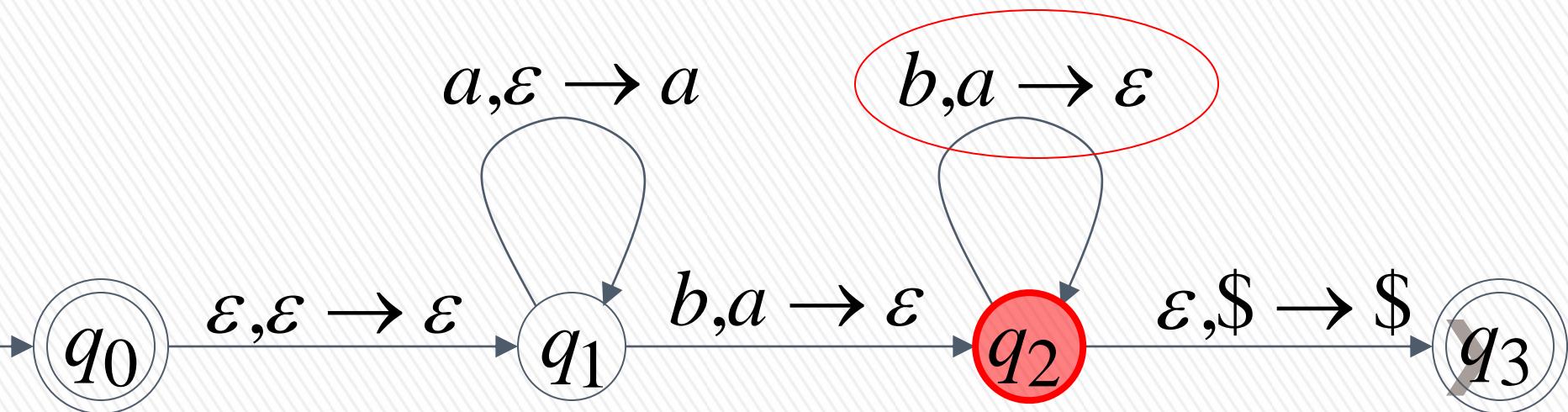
Time 6

Input

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



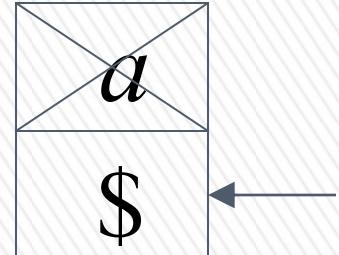
Stack



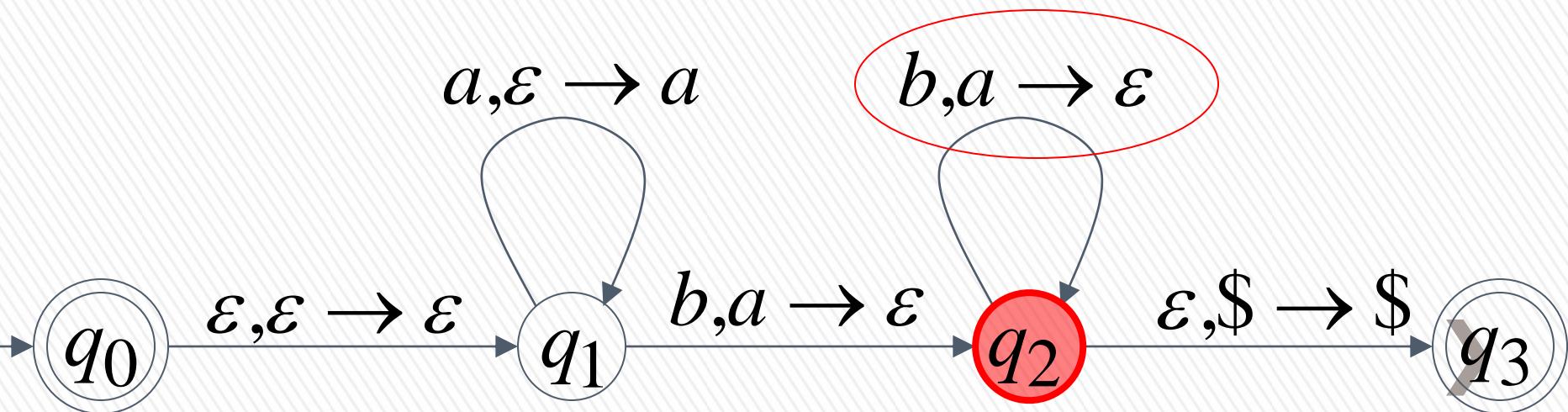
Time 7

Input

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



Stack



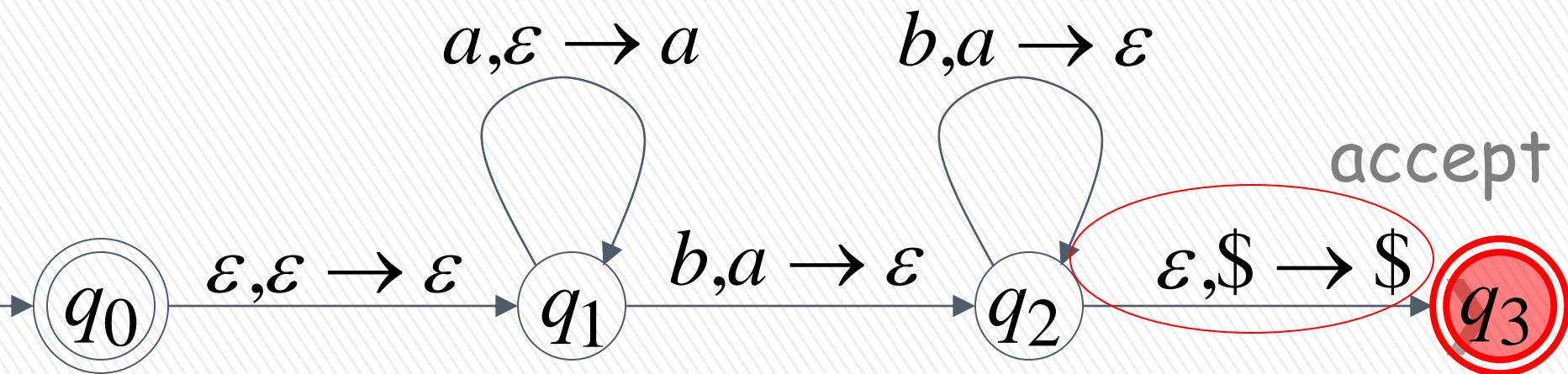
Time 8

Input

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



Stack



A string is accepted if there is
a computation such that:

All the input is consumed

AND

The last state is an accepting state

we do not care about the stack contents
at the end of the accepting computation

Rejection Example: Time 0

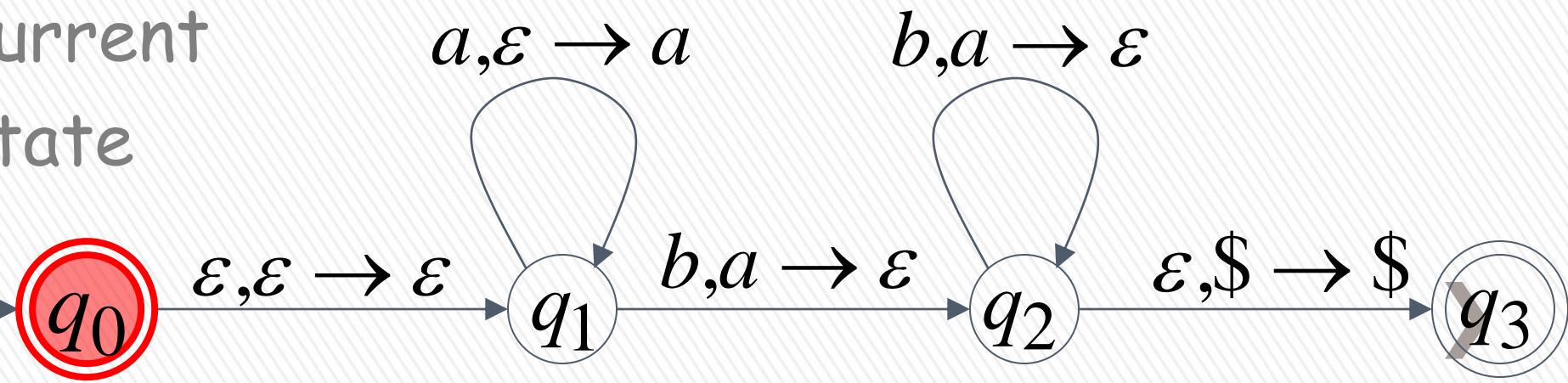
Input

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



Stack

current
state



Rejection Example: Time 1

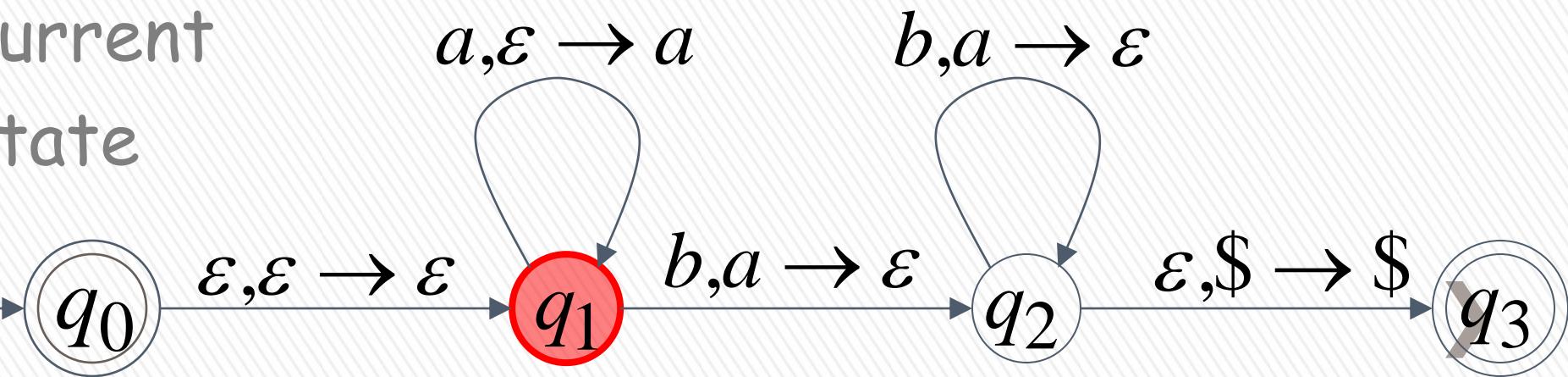
Input

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



Stack

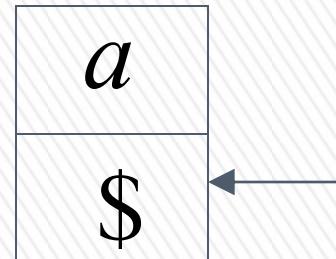
current
state



Rejection Example: Time 2

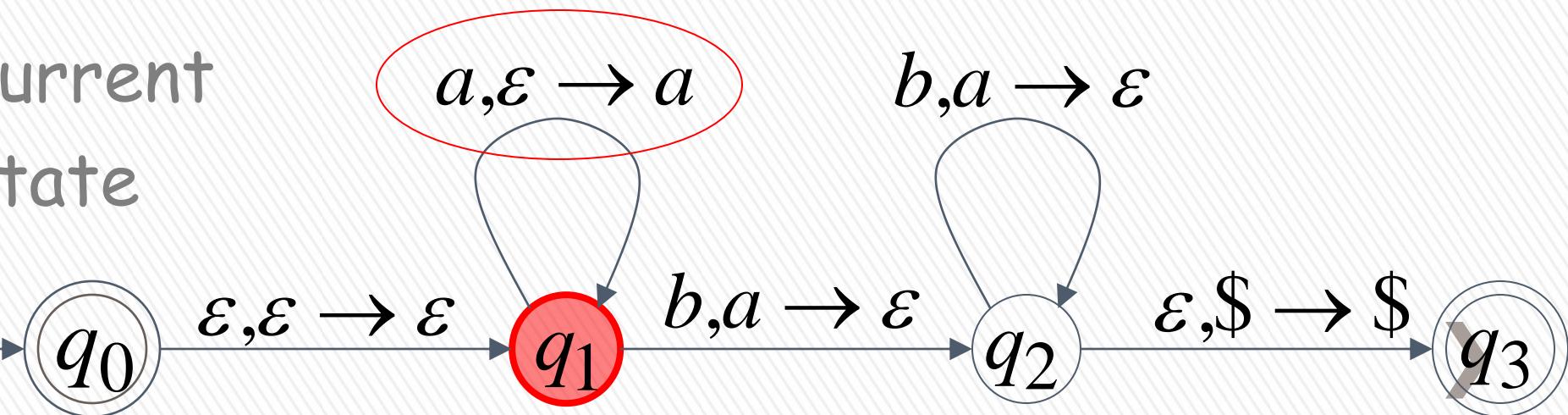
Input

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



Stack

current
state

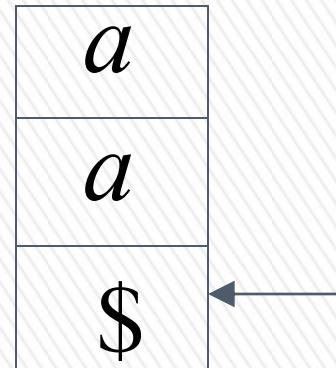


Rejection Example: Time 3

Input

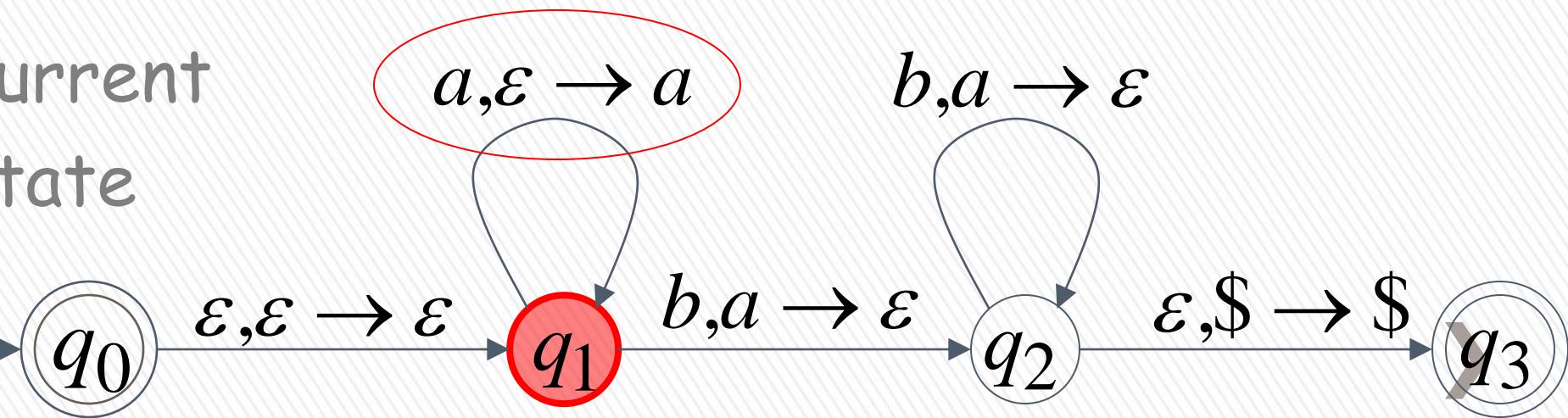
a	a	b

↑



Stack

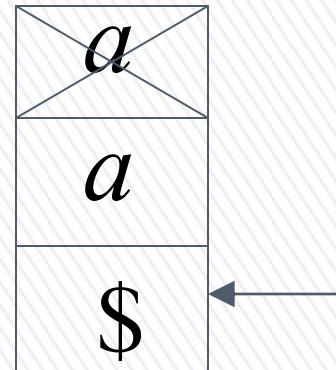
current
state



Rejection Example: Time 4

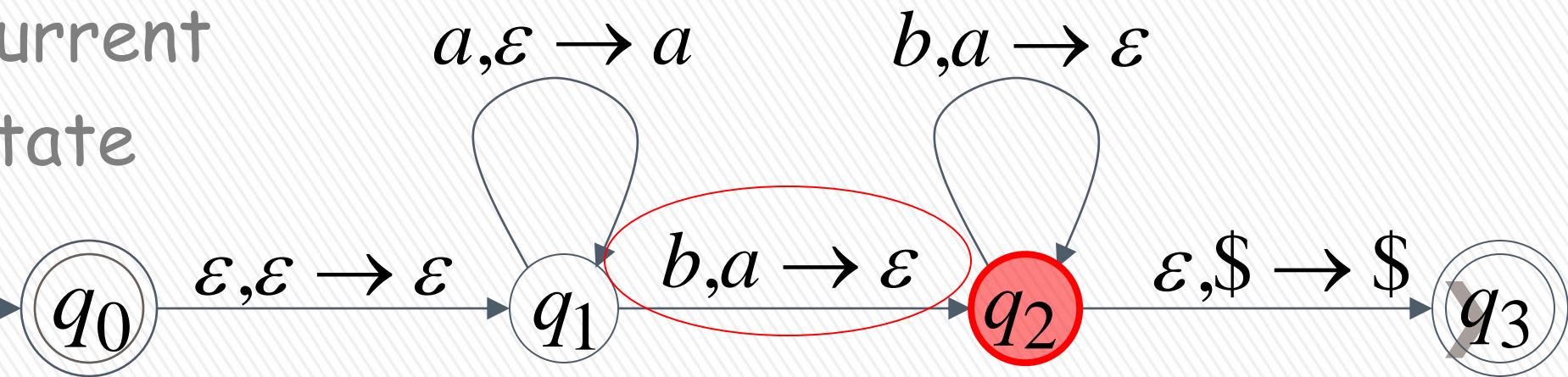
Input

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



Stack

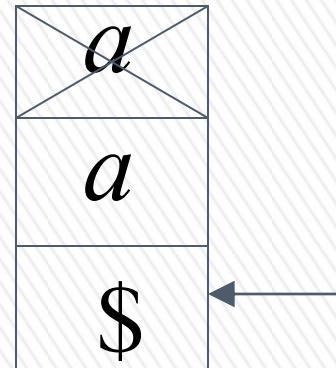
current
state



Rejection Example: Time 4

Input

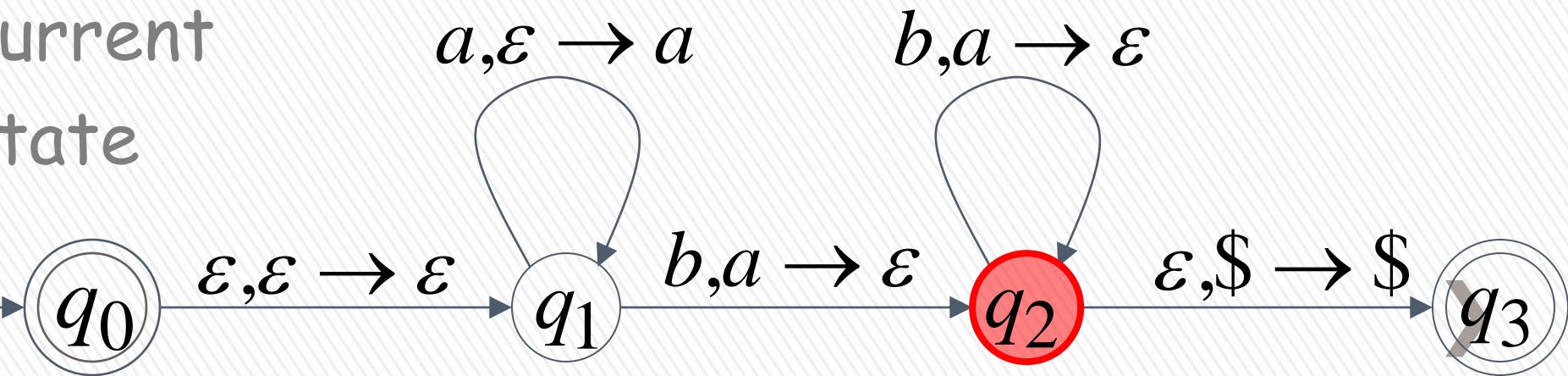
a	a	b
-----	-----	-----



Stack

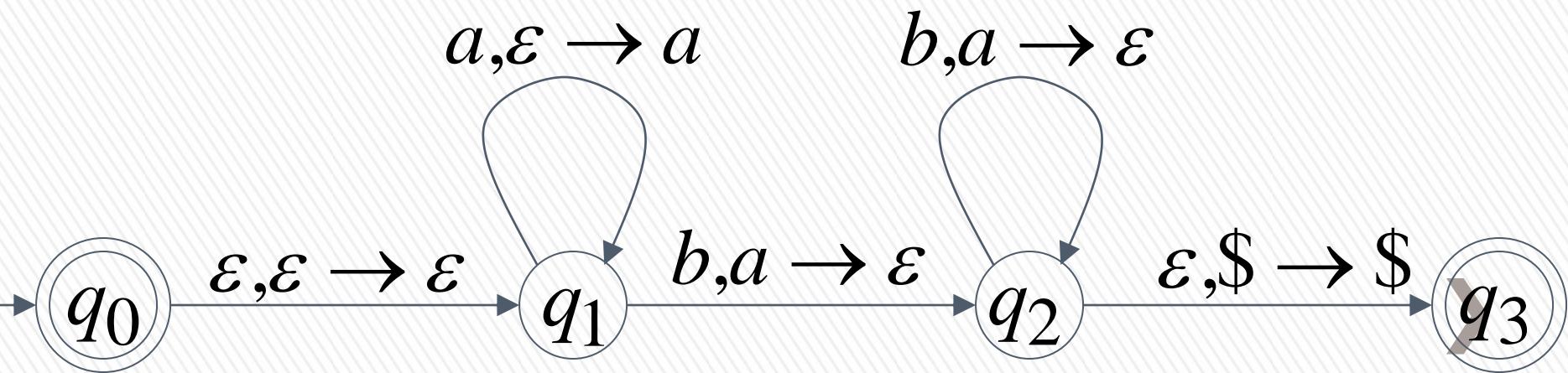
reject

current
state



There is no accepting computation for aab

The string aab is rejected by the PDA



Another PDA example

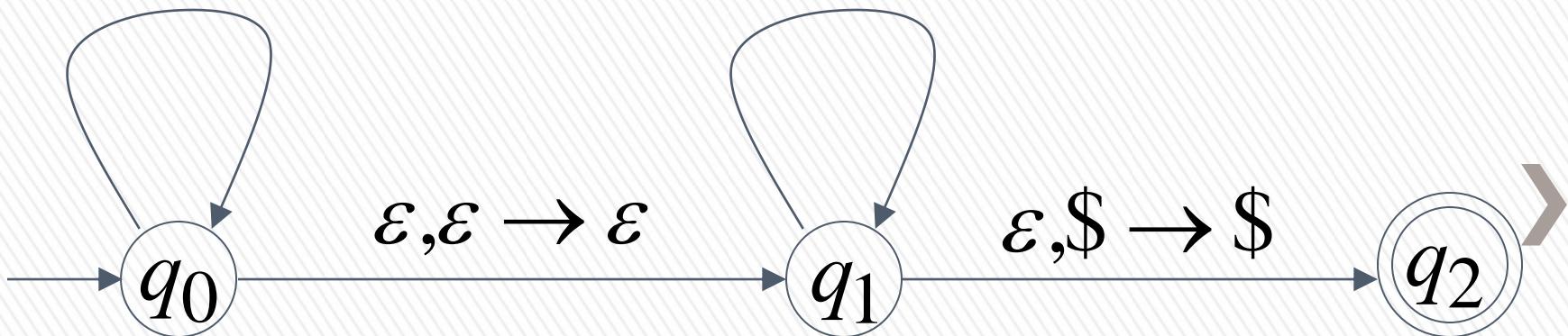
PDA M : $L(M) = \{vv^R : v \in \{a,b\}^*\}$

$a, \varepsilon \rightarrow a$

$b, \varepsilon \rightarrow b$

$a, a \rightarrow \varepsilon$

$b, b \rightarrow \varepsilon$



Basic Idea:

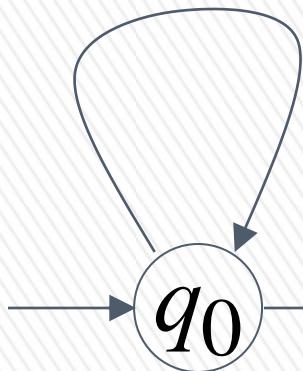
$$L(M) = \{vv^R : v \in \{a,b\}^*\}$$

1. Push v on stack



$$a, \varepsilon \rightarrow a$$

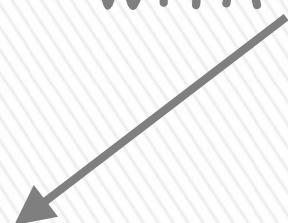
$$b, \varepsilon \rightarrow b$$



2. Guess middle of input

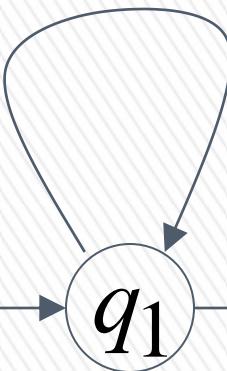
$$\varepsilon, \varepsilon \rightarrow \varepsilon$$

3. Match v^R on input with v on stack

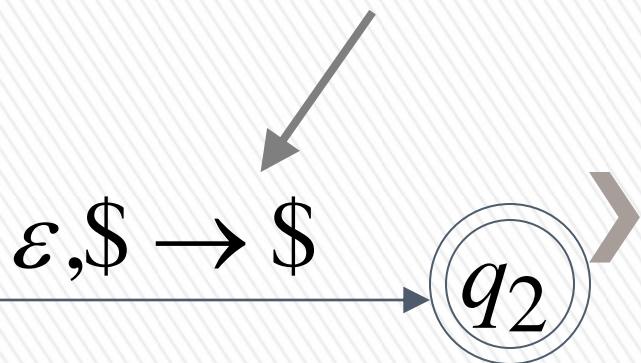


$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



4. Match found



Execution Example: Time 0

Input

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



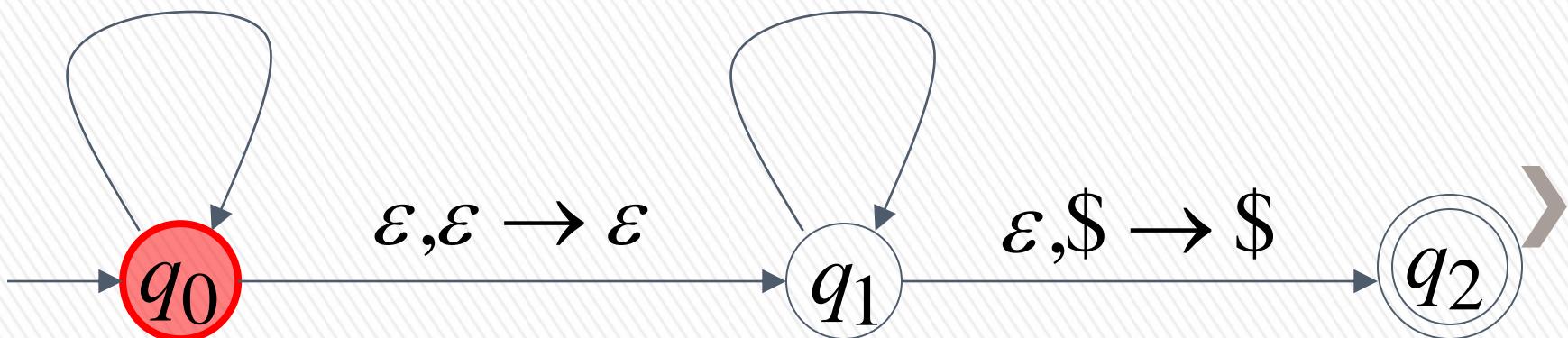
Stack

$$a, \varepsilon \rightarrow a$$

$$a, a \rightarrow \varepsilon$$

$$b, \varepsilon \rightarrow b$$

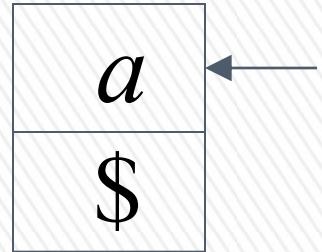
$$b, b \rightarrow \varepsilon$$



Time 1

Input

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



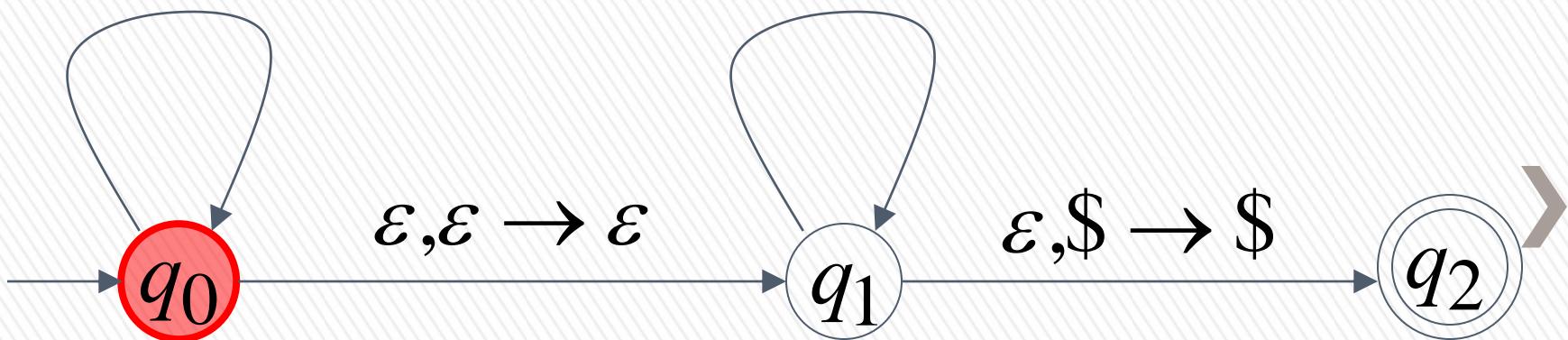
Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

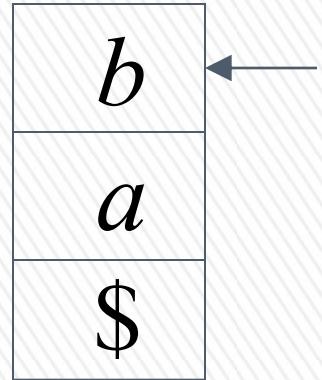
$$b, b \rightarrow \varepsilon$$



Time 2

Input

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



Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$

Transitions from q_0 :



$$\varepsilon, \varepsilon \rightarrow \varepsilon$$

q_0



$$\varepsilon, \$ \rightarrow \$$$

q_1

q_2

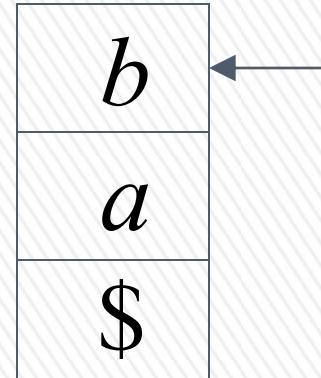


Time 3

Input

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

Guess the middle
of string



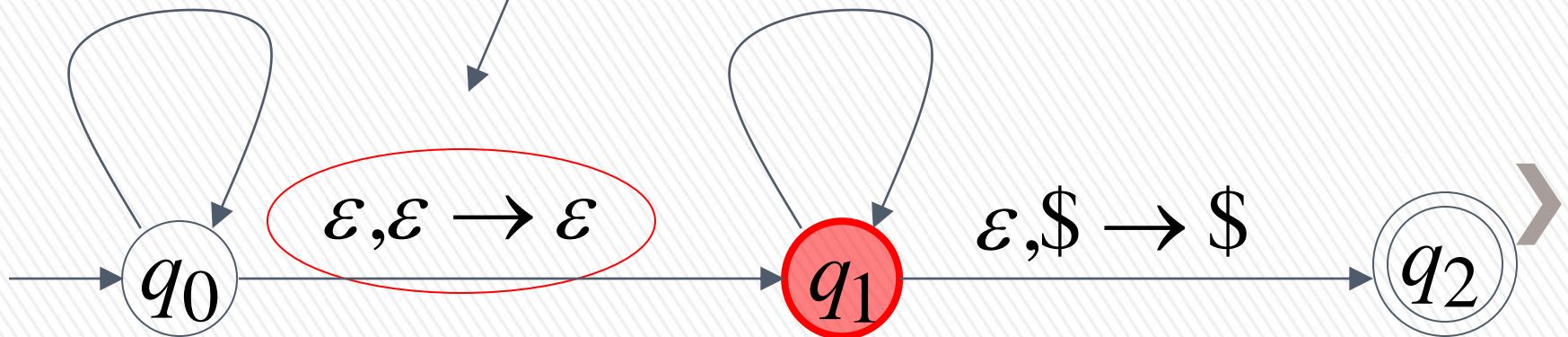
Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

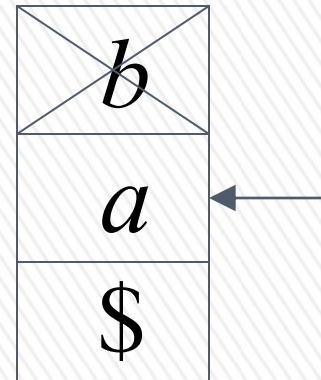
$$b, b \rightarrow \varepsilon$$



Time 4

Input

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



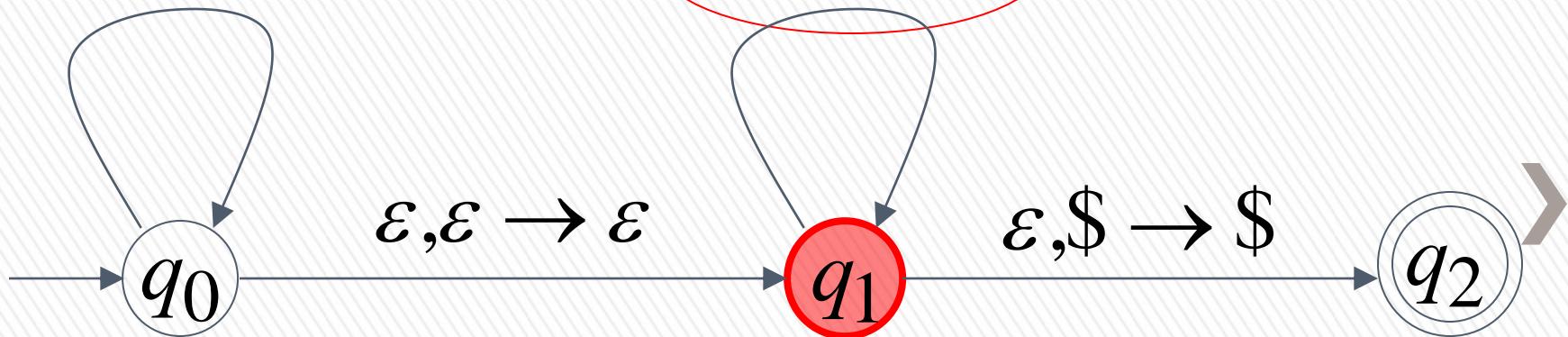
Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

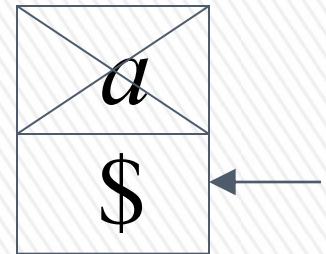
$$b, b \rightarrow \varepsilon$$



Time 5

Input

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



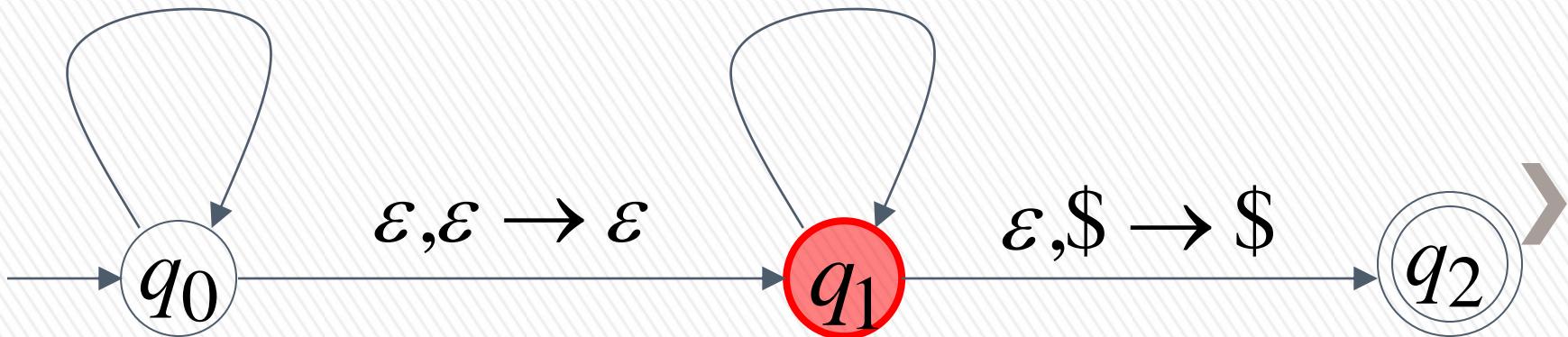
Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



Time 6

Input

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



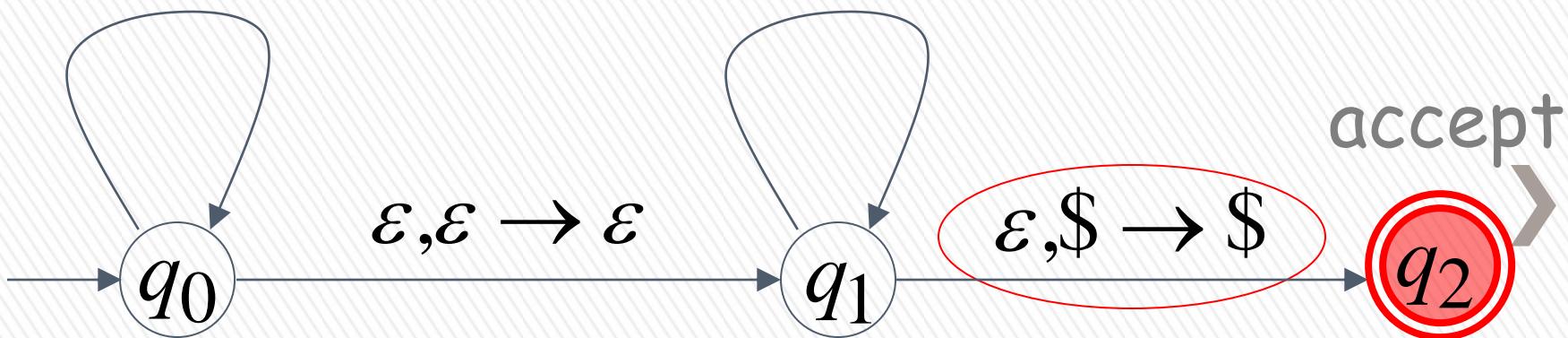
Stack

$$a, \varepsilon \rightarrow a$$

$$a, a \rightarrow \varepsilon$$

$$b, \varepsilon \rightarrow b$$

$$b, b \rightarrow \varepsilon$$



Rejection Example: Time 0

Input

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



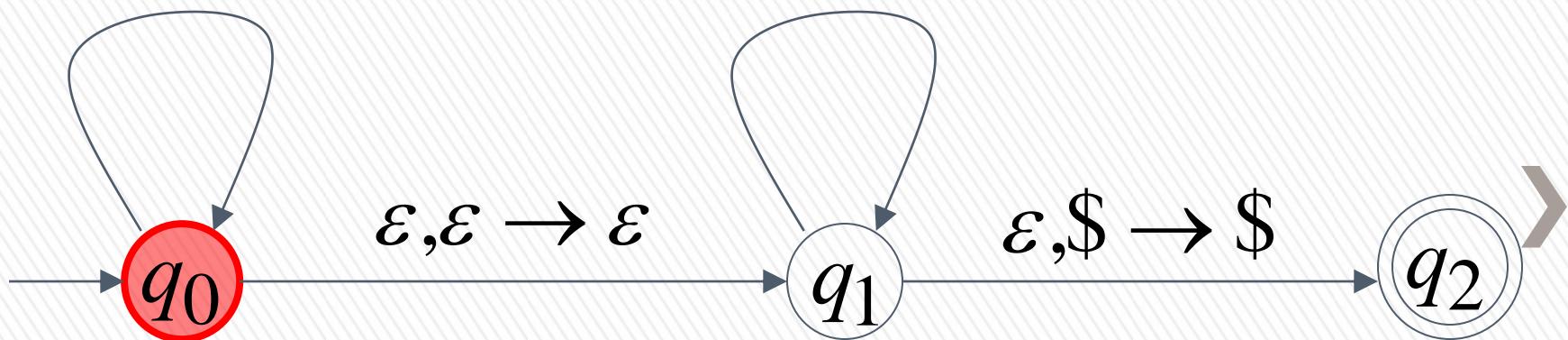
Stack

$$a, \varepsilon \rightarrow a$$

$$a, a \rightarrow \varepsilon$$

$$b, \varepsilon \rightarrow b$$

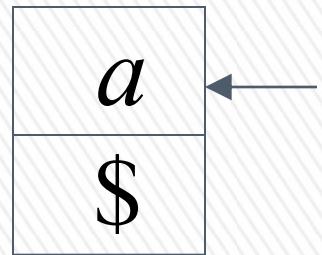
$$b, b \rightarrow \varepsilon$$



Time 1

Input

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



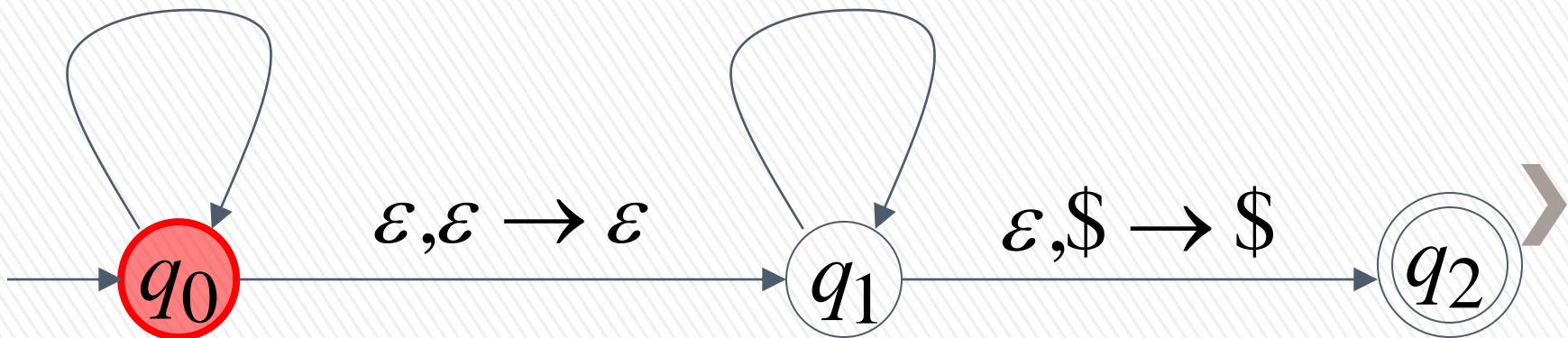
Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

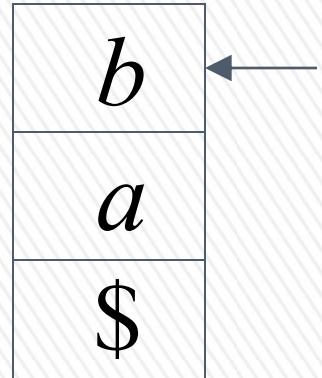
$$b, b \rightarrow \varepsilon$$



Time 2

Input

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



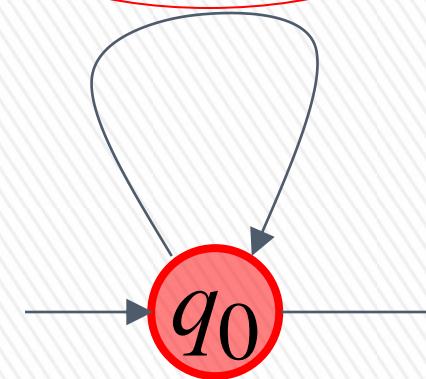
Stack

$$a, \varepsilon \rightarrow a$$

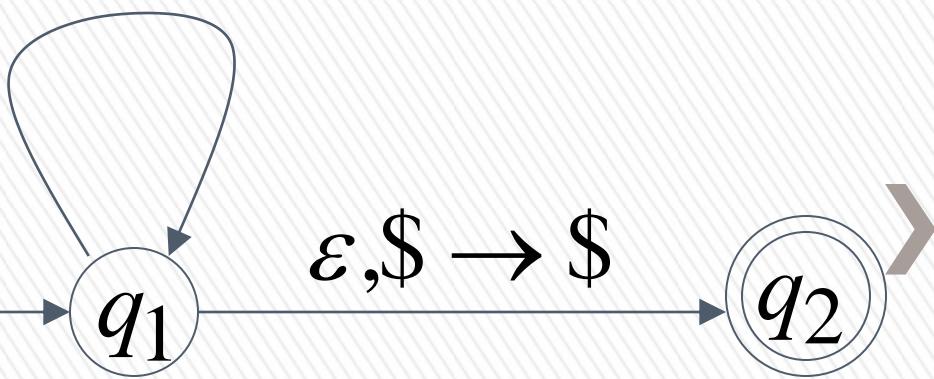
$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



$$\varepsilon, \varepsilon \rightarrow \varepsilon$$



$$\varepsilon, \$ \rightarrow \$$$

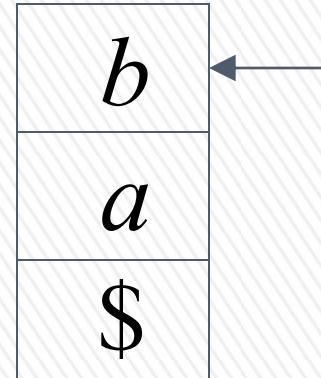
q_2

Time 3

Input

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

Guess the middle
of string



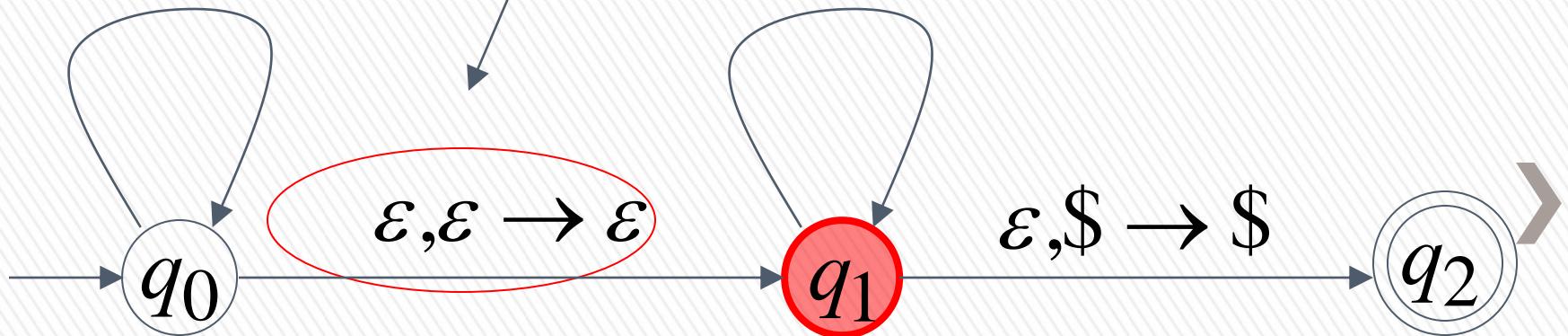
Stack

$$a, \epsilon \rightarrow a$$

$$b, \epsilon \rightarrow b$$

$$a, a \rightarrow \epsilon$$

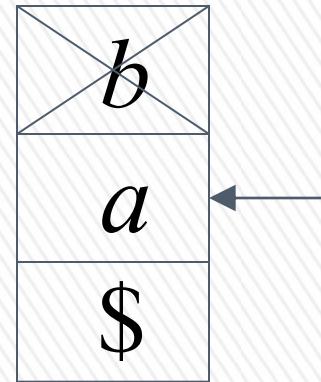
$$b, b \rightarrow \epsilon$$



Time 4

Input

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



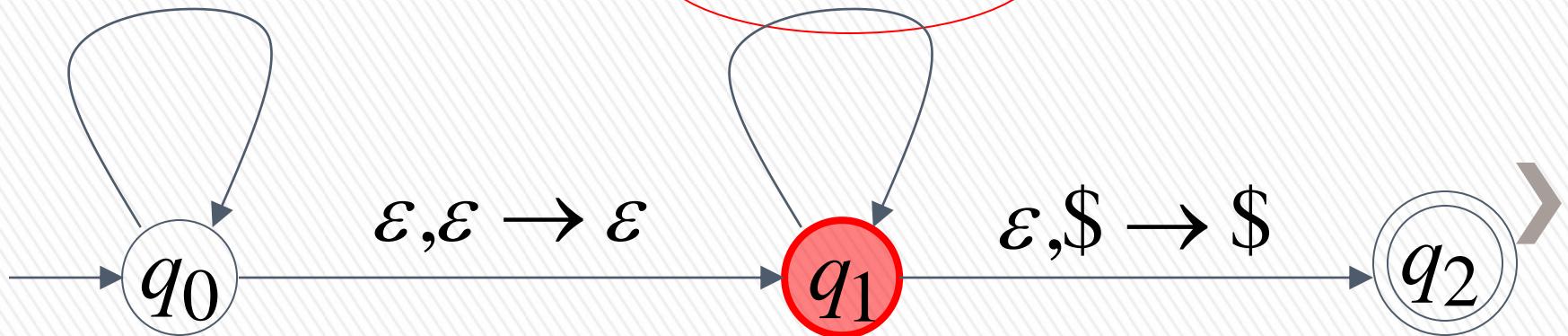
Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



Time 5

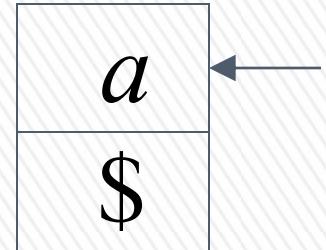
Input

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



There is no possible transition.

Input is not
consumed



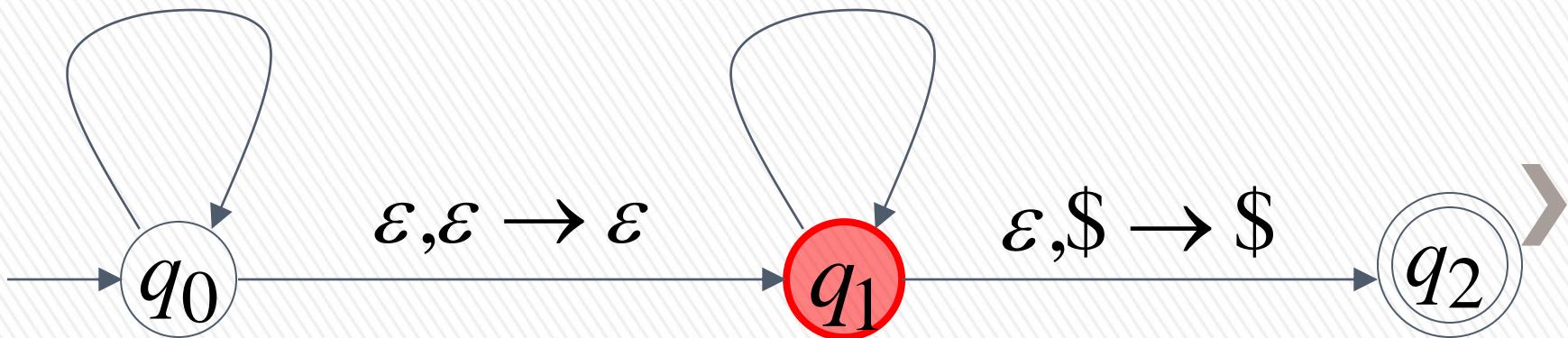
Stack

$$a, \varepsilon \rightarrow a$$

$$a, a \rightarrow \varepsilon$$

$$b, \varepsilon \rightarrow b$$

$$b, b \rightarrow \varepsilon$$



Another computation on same string:

Input

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

Time 0



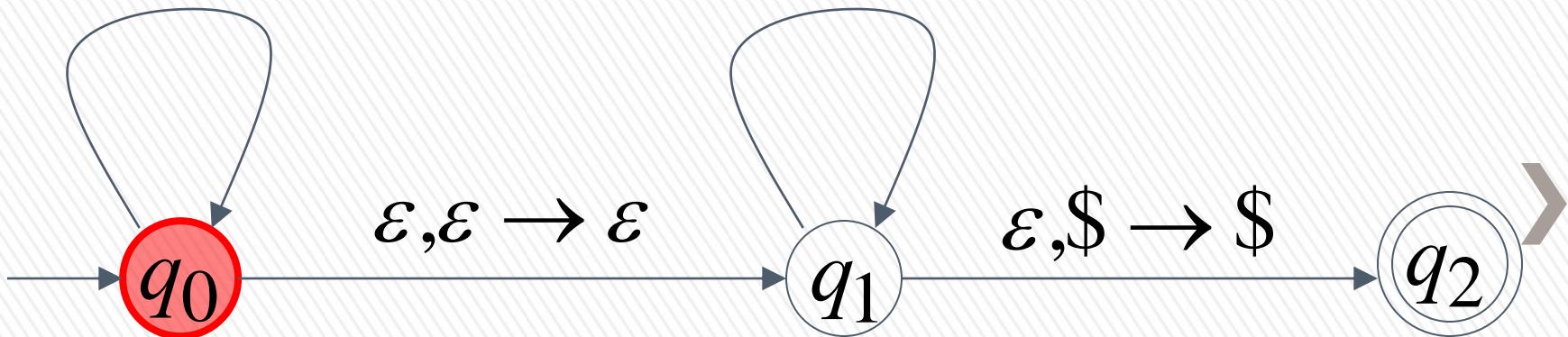
Stack

$$a, \epsilon \rightarrow a$$

$$a, a \rightarrow \epsilon$$

$$b, \epsilon \rightarrow b$$

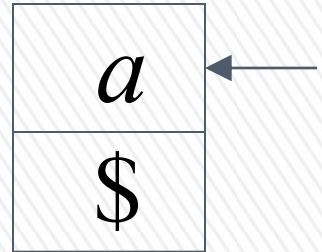
$$b, b \rightarrow \epsilon$$



Time 1

Input

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



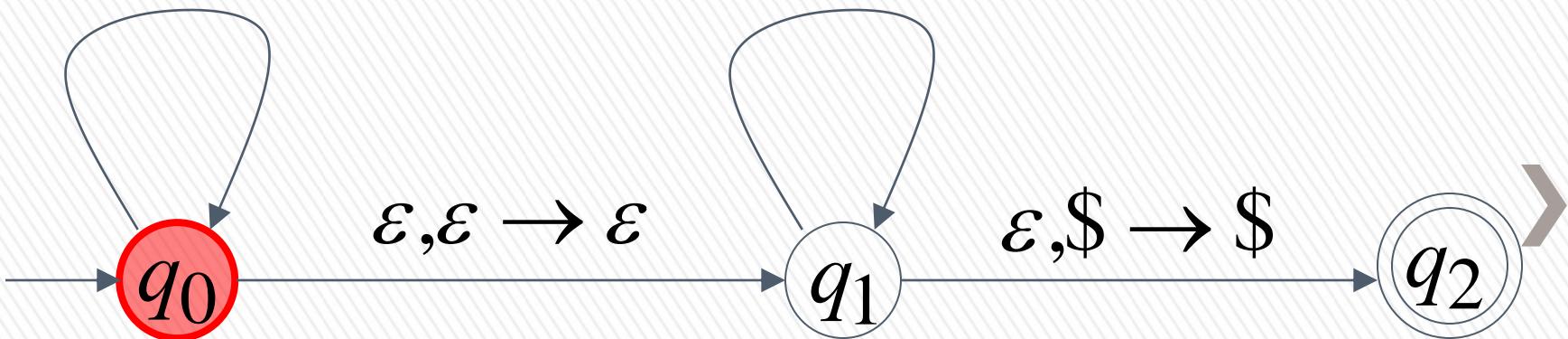
Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

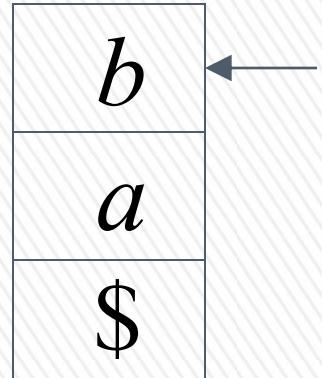
$$b, b \rightarrow \varepsilon$$



Time 2

Input

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



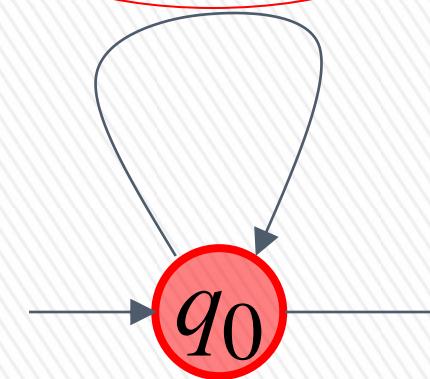
Stack

$$a, \varepsilon \rightarrow a$$

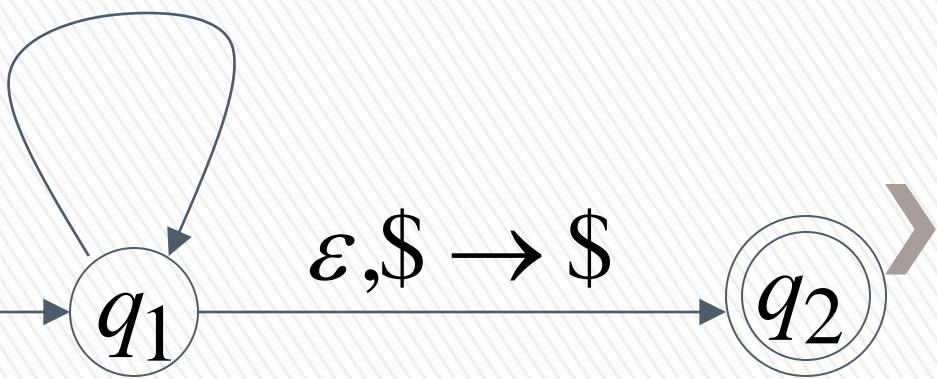
$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



$$\varepsilon, \varepsilon \rightarrow \varepsilon$$



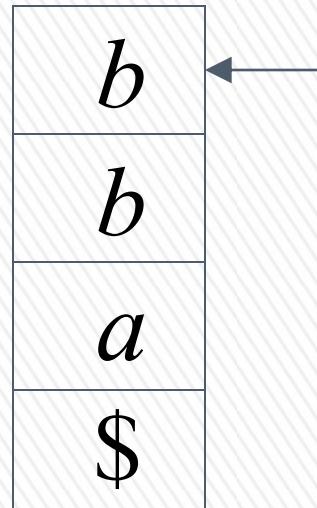
$$\varepsilon, \$ \rightarrow \$$$

q_2

Time 3

Input

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



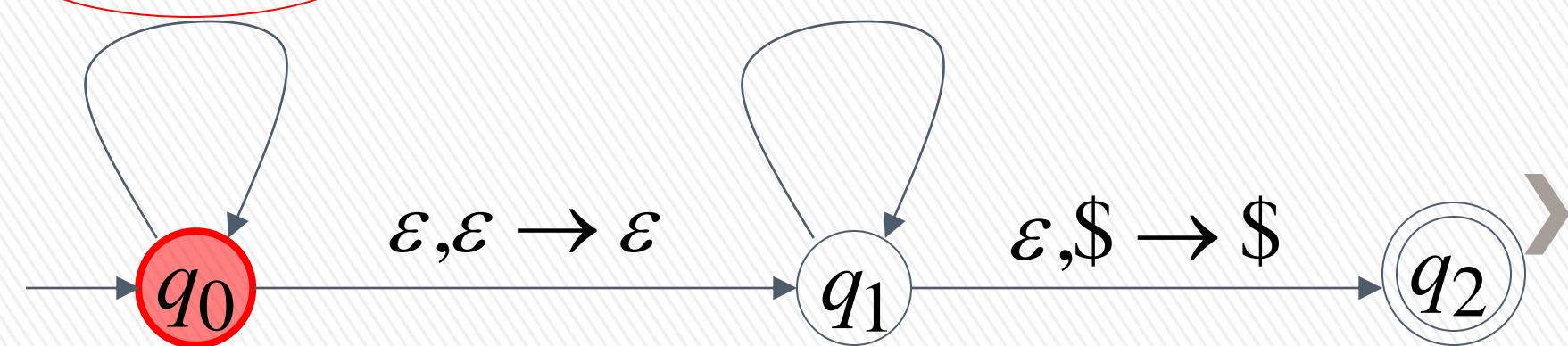
Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



Input

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

Time 4

b
b
b
a
\$

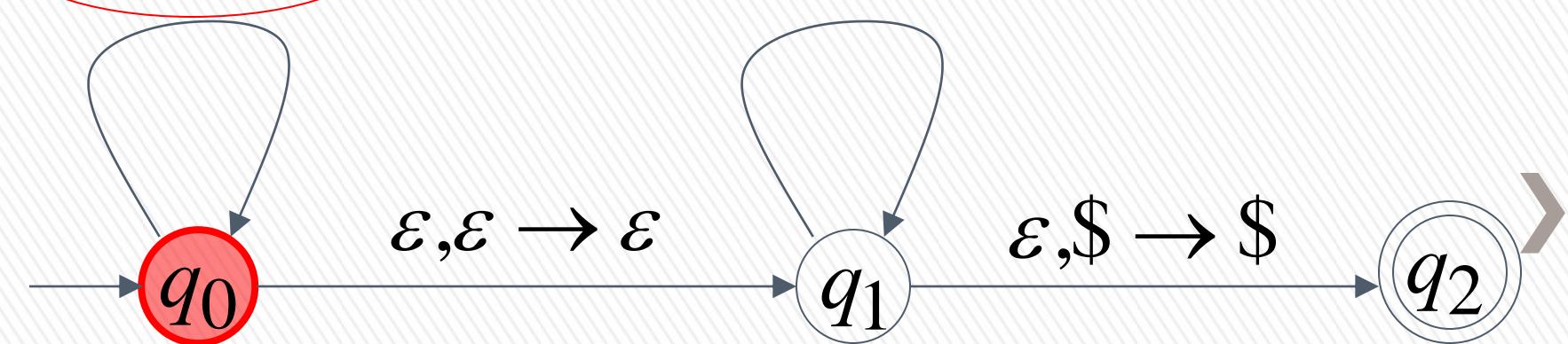
Stack

$$a, \varepsilon \rightarrow a$$

$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

$$b, b \rightarrow \varepsilon$$



Input

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

Time 5

No accept state
is reached

b
b
b
a
\$

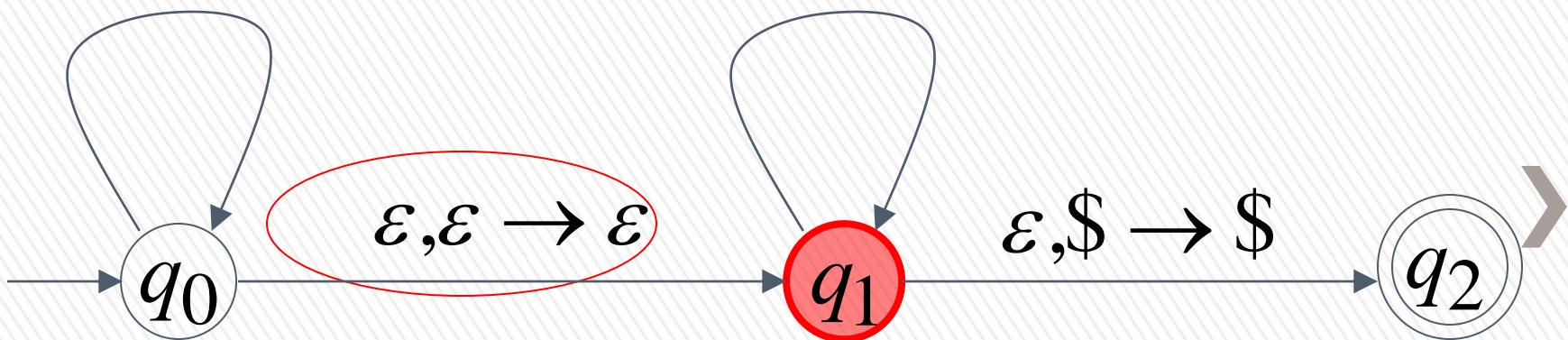
Stack

$$a, \varepsilon \rightarrow a$$

$$a, a \rightarrow \varepsilon$$

$$b, \varepsilon \rightarrow b$$

$$b, b \rightarrow \varepsilon$$



There is no computation
that accepts string $abbb$

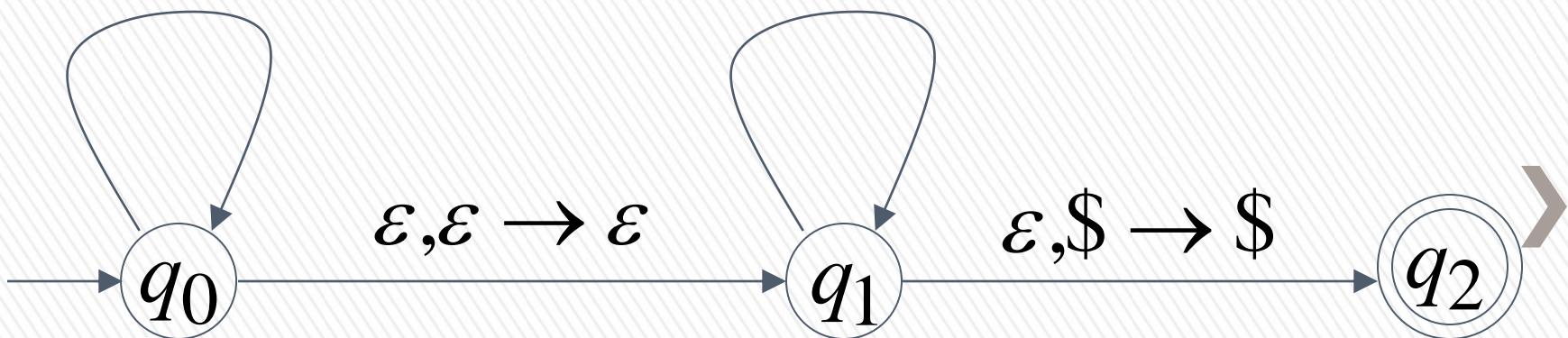
$$abbb \notin L(M)$$

$$a, \varepsilon \rightarrow a$$

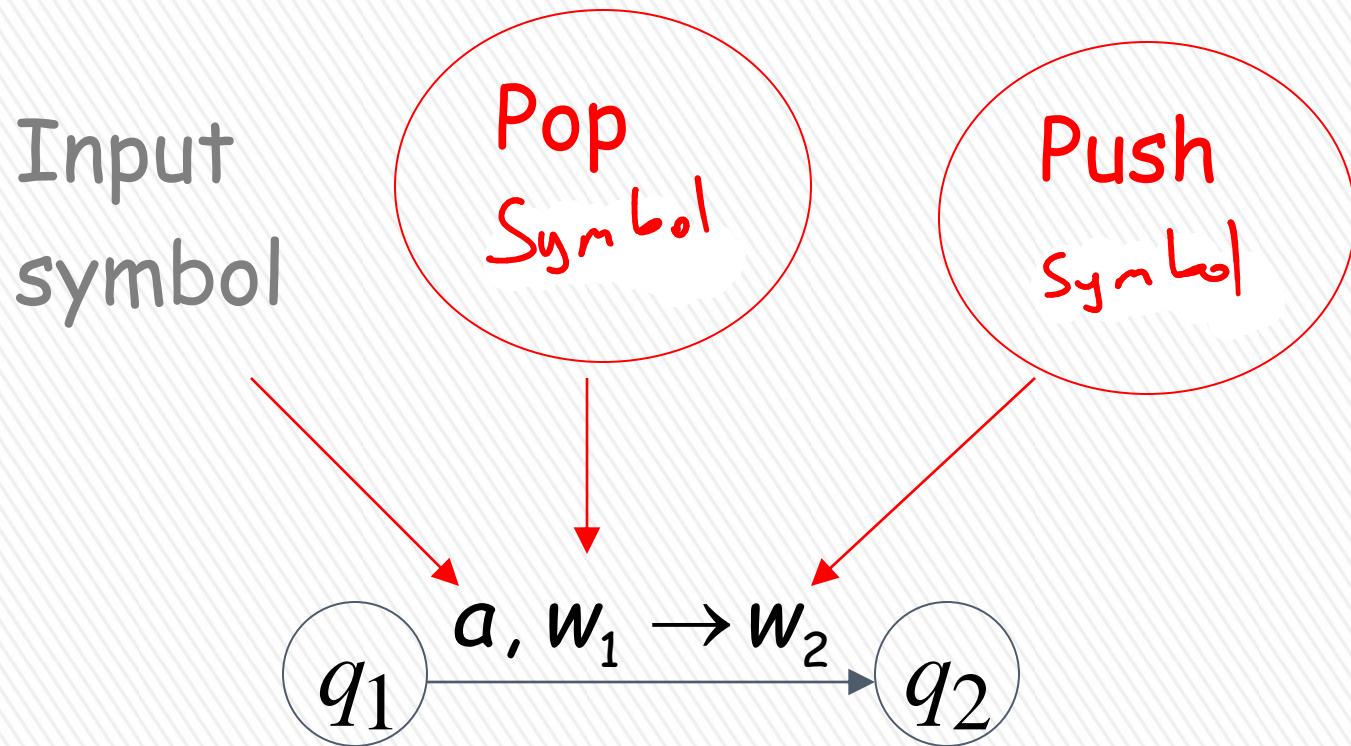
$$b, \varepsilon \rightarrow b$$

$$a, a \rightarrow \varepsilon$$

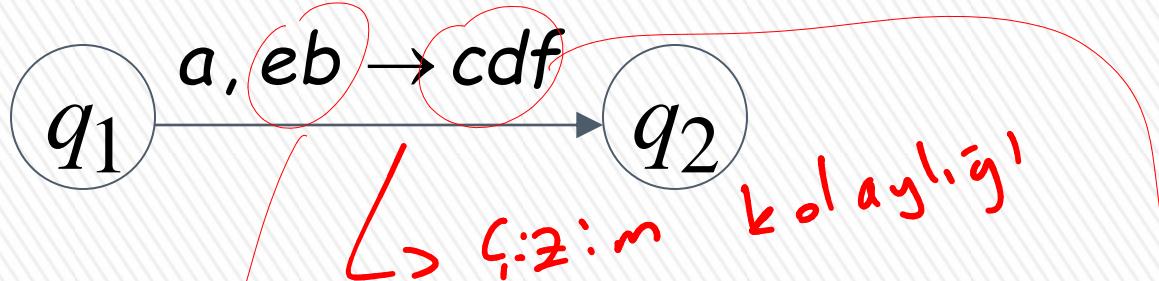
$$b, b \rightarrow \varepsilon$$



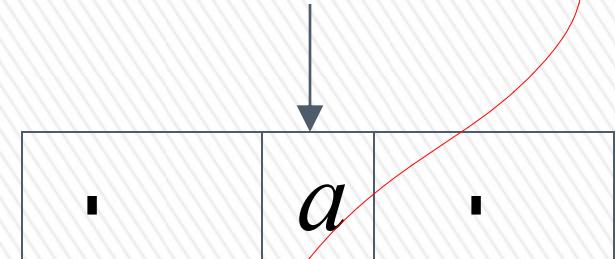
Pushing & Popping Strings



Example:



input



stack

pop
string

<i>e</i>
<i>b</i>
<i>h</i>
<i>e</i>
<i>\$</i>

top

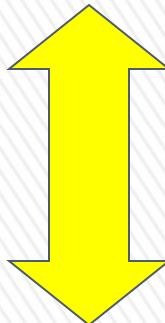
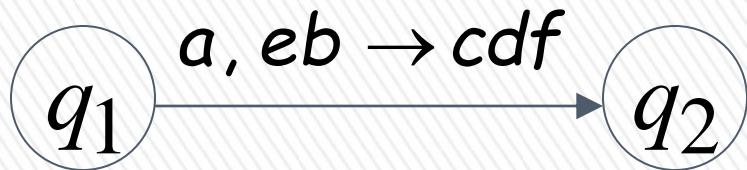
top

Replace

<i>c</i>
<i>d</i>
<i>f</i>
<i>h</i>
<i>e</i>
<i>\$</i>

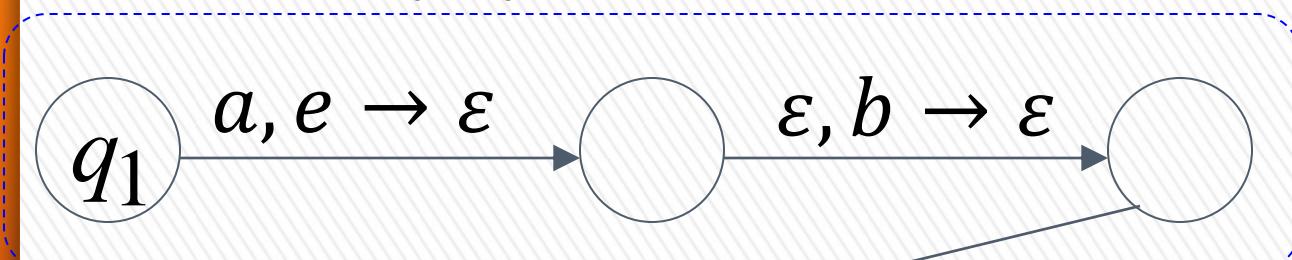
push
string





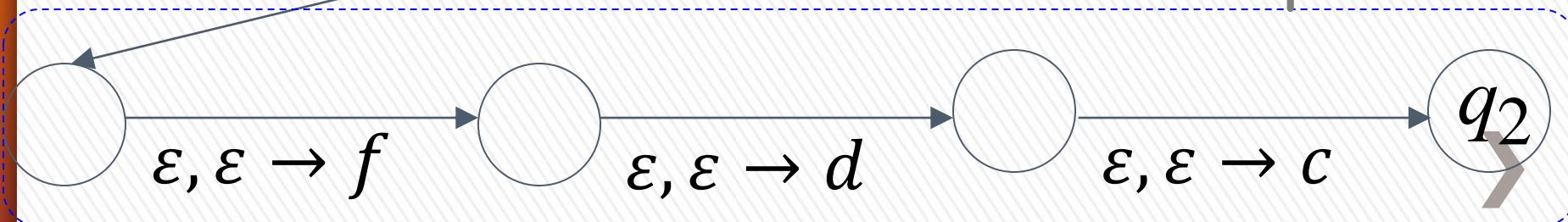
Equivalent
transitions

pop



$\varepsilon, \varepsilon \rightarrow \varepsilon$

push



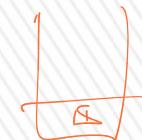
Another PDA example

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

PDA M

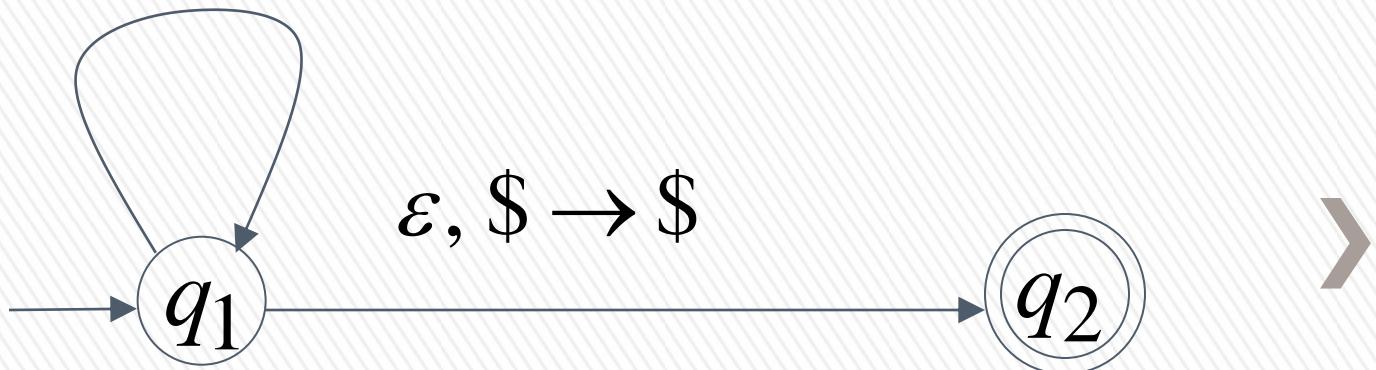
a a a a

$$a, \$ \rightarrow 0\$ \quad b, \$ \rightarrow 1\$$$



$$a, 0 \rightarrow 00 \quad b, 1 \rightarrow 11$$

$$a, 1 \rightarrow \epsilon \quad b, 0 \rightarrow \epsilon$$



Execution Example:

Time 0

Input

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



$a, \$ \rightarrow a\$$ $b, \$ \rightarrow b\$$

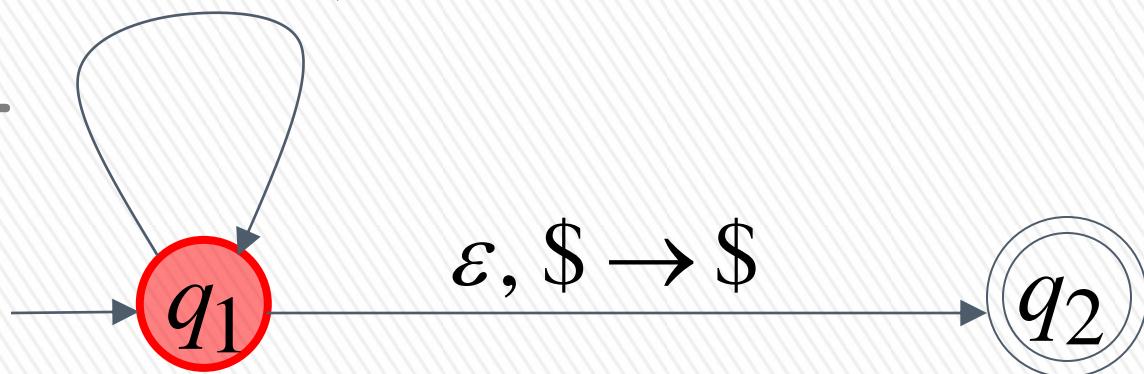
$a, a \rightarrow aa$ $b, b \rightarrow bb$

$a, b \rightarrow \epsilon$ $b, a \rightarrow \epsilon$

\$

Stack

current
state



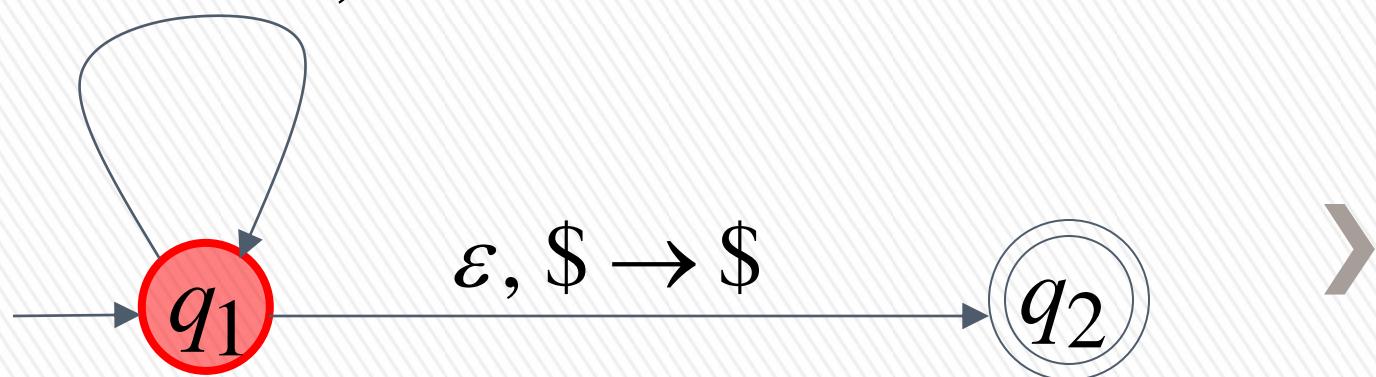
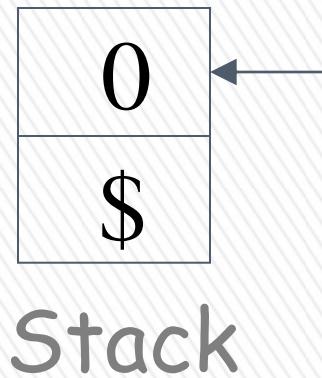
Time 1

Input

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



- | | |
|-----------------------------|-----------------------------|
| $a, \$ \rightarrow 0\$$ | $b, \$ \rightarrow 1\$$ |
| $a, 0 \rightarrow 00$ | $b, 1 \rightarrow 11$ |
| $a, 1 \rightarrow \epsilon$ | $b, 0 \rightarrow \epsilon$ |



Time 3

Input

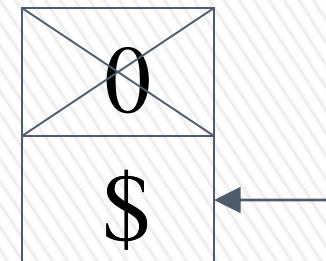
a	b	b	b	a	a
-----	-----	-----	-----	-----	-----



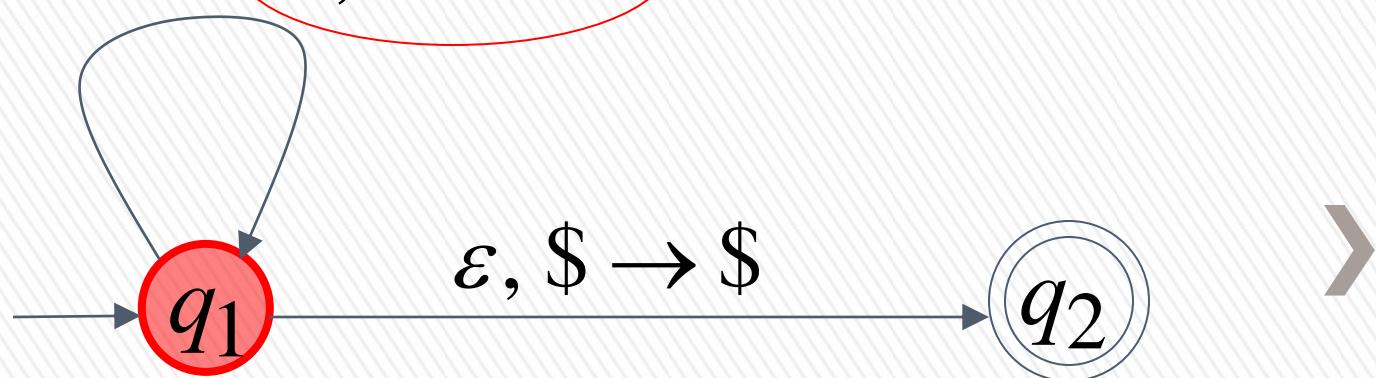
$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

$a, 1 \rightarrow \epsilon$ $b, 0 \rightarrow \epsilon$



Stack



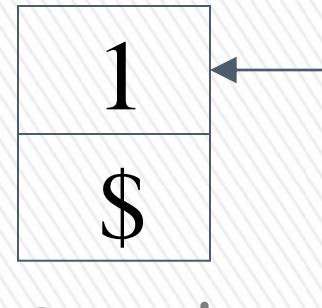
Time 4

Input

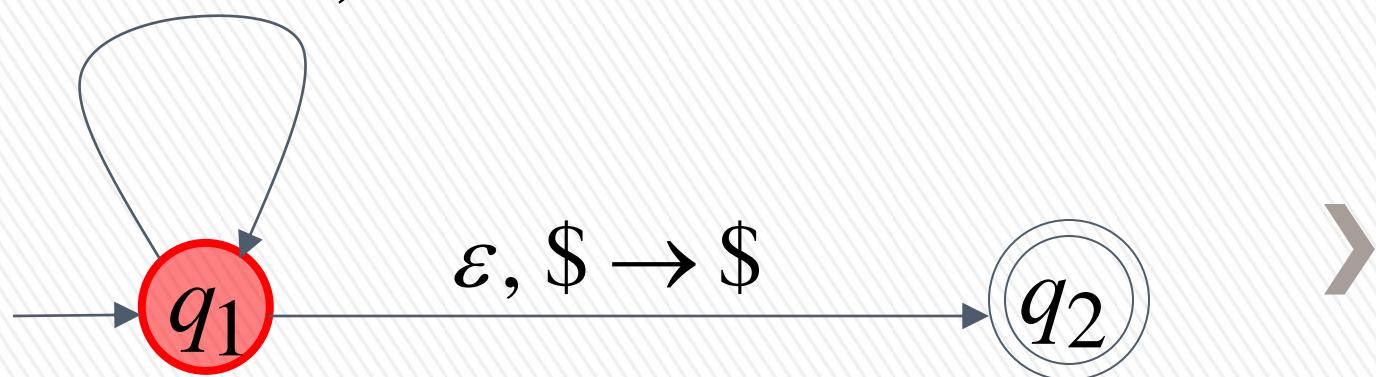
a	b	b	b	a	a
-----	-----	-----	-----	-----	-----



$$\begin{array}{ll} a, \$ \rightarrow 0\$ & b, \$ \rightarrow 1\$ \\ a, 0 \rightarrow 00 & b, 1 \rightarrow 11 \\ a, 1 \rightarrow \epsilon & b, 0 \rightarrow \epsilon \end{array}$$



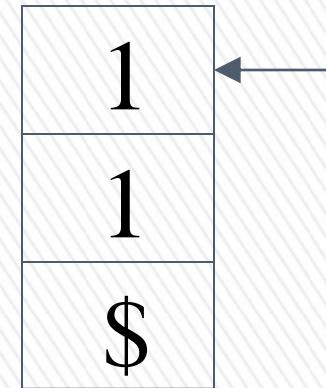
Stack



Time 5

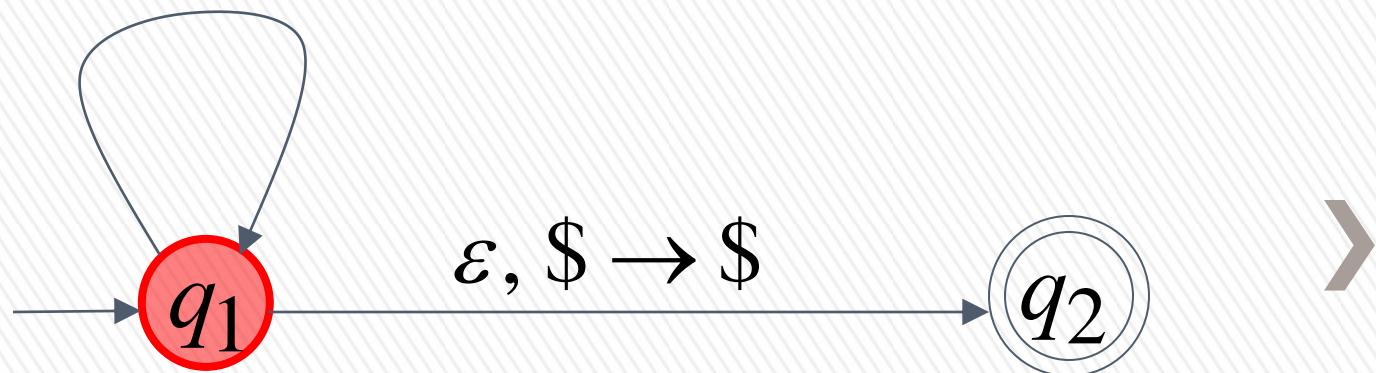
Input

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



Stack

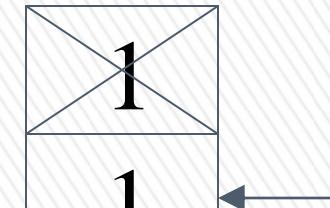
$$\begin{array}{ll} a, \$ \rightarrow 0\$ & b, \$ \rightarrow 1\$ \\ a, 0 \rightarrow 00 & b, 1 \rightarrow 11 \\ a, 1 \rightarrow \epsilon & b, 0 \rightarrow \epsilon \end{array}$$



Time 6

Input

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



Stack

$a, \$ \rightarrow 0\$$

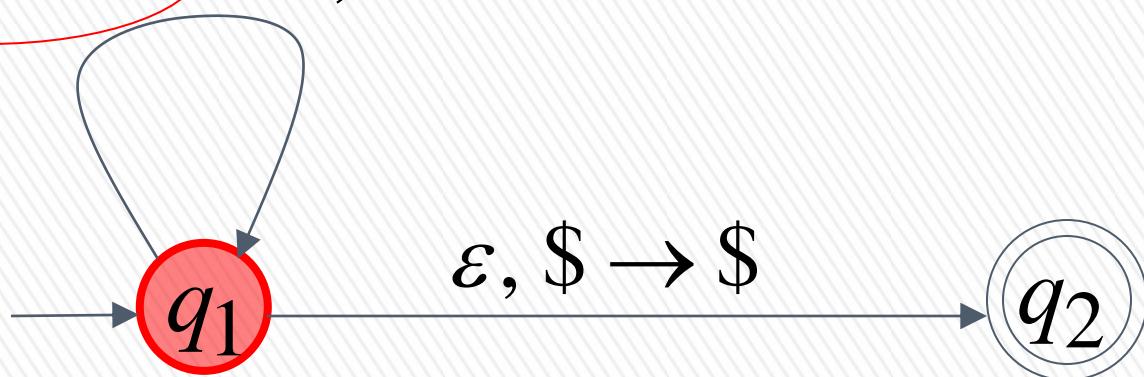
$b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$

$b, 1 \rightarrow 11$

$a, 1 \rightarrow \epsilon$

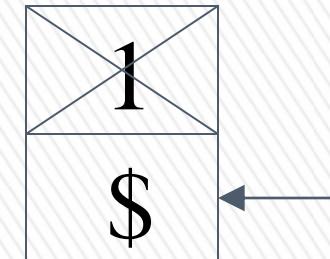
$b, 0 \rightarrow \epsilon$



Time 7

Input

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

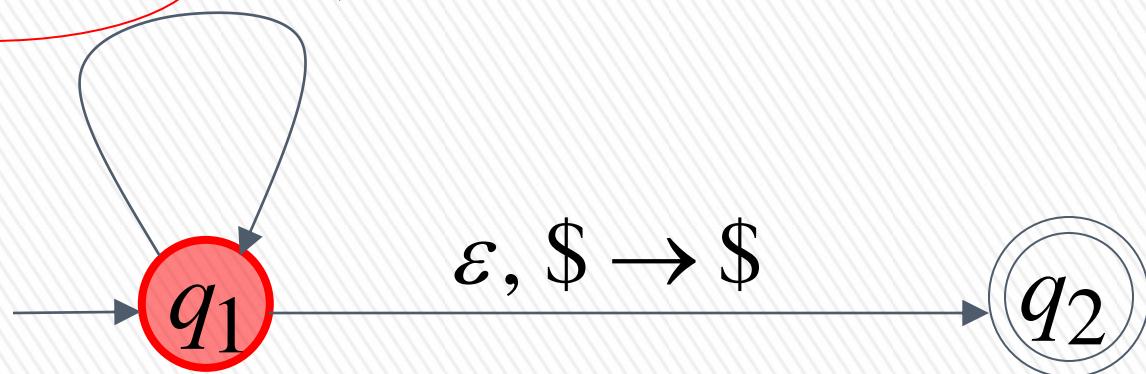


Stack

$a, \$ \rightarrow 0\$$ $b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$ $b, 1 \rightarrow 11$

$a, 1 \rightarrow \epsilon$ $b, 0 \rightarrow \epsilon$



Time 8

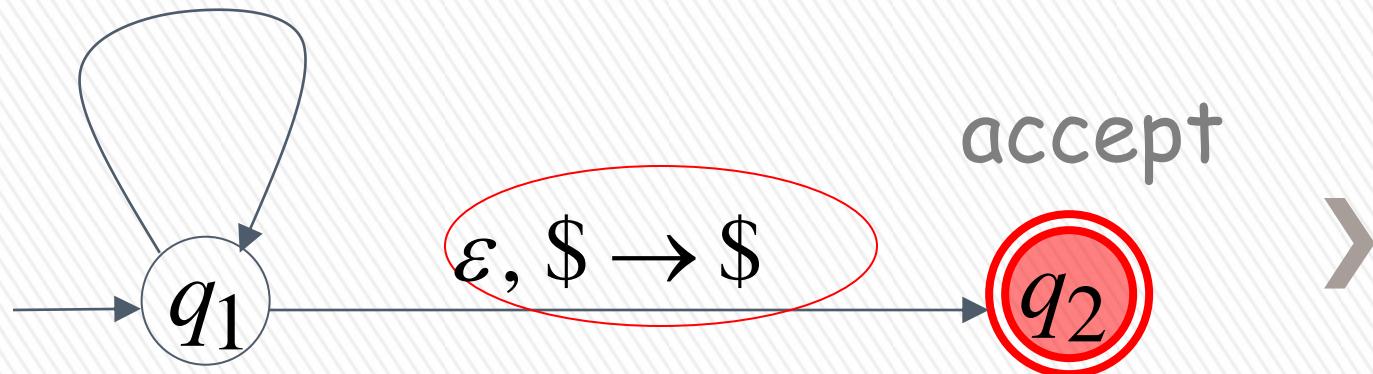
Input

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

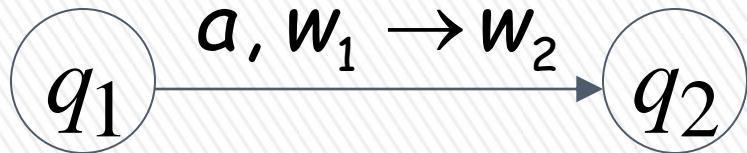


Stack

$$\begin{array}{ll} a, \$ \rightarrow 0\$ & b, \$ \rightarrow 1\$ \\ a, 0 \rightarrow 00 & b, 1 \rightarrow 11 \\ a, 1 \rightarrow \epsilon & b, 0 \rightarrow \epsilon \end{array}$$



Formalities for PDAs



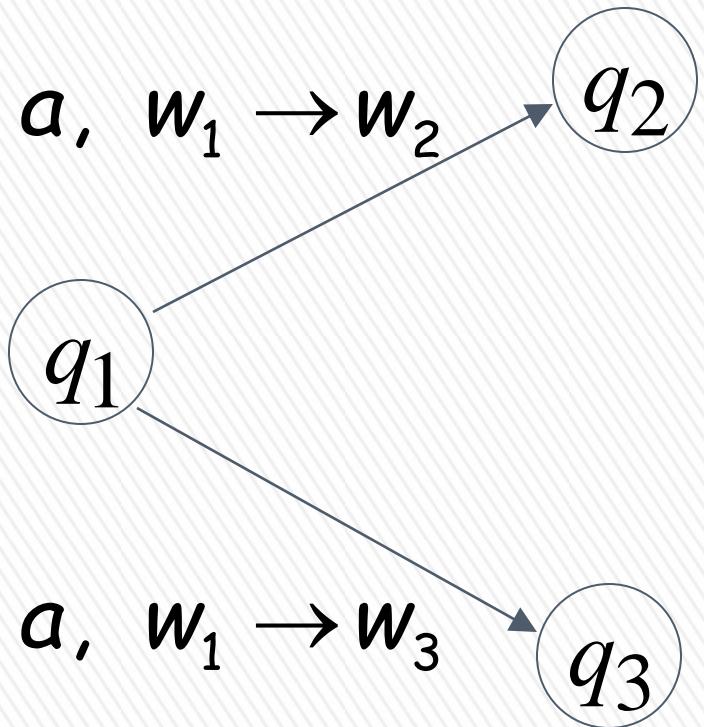
Transition function:

$\delta(q_1, a, w_1) = \{(q_2, w_2)\}$

Annotations in red:

- Red circles highlight the states q_1 , a , w_1 , q_2 , and w_2 .
- A red arrow points from q_1 to q_2 , labeled "state q ".
- A red arrow points from w_1 to w_2 , labeled "push - pop".
- A red arrow points from a to w_1 , labeled "push - pop".
- A red arrow points from w_2 back to w_1 , labeled "pop".
- A red arrow points from q_2 back to q_1 , labeled "state q ".





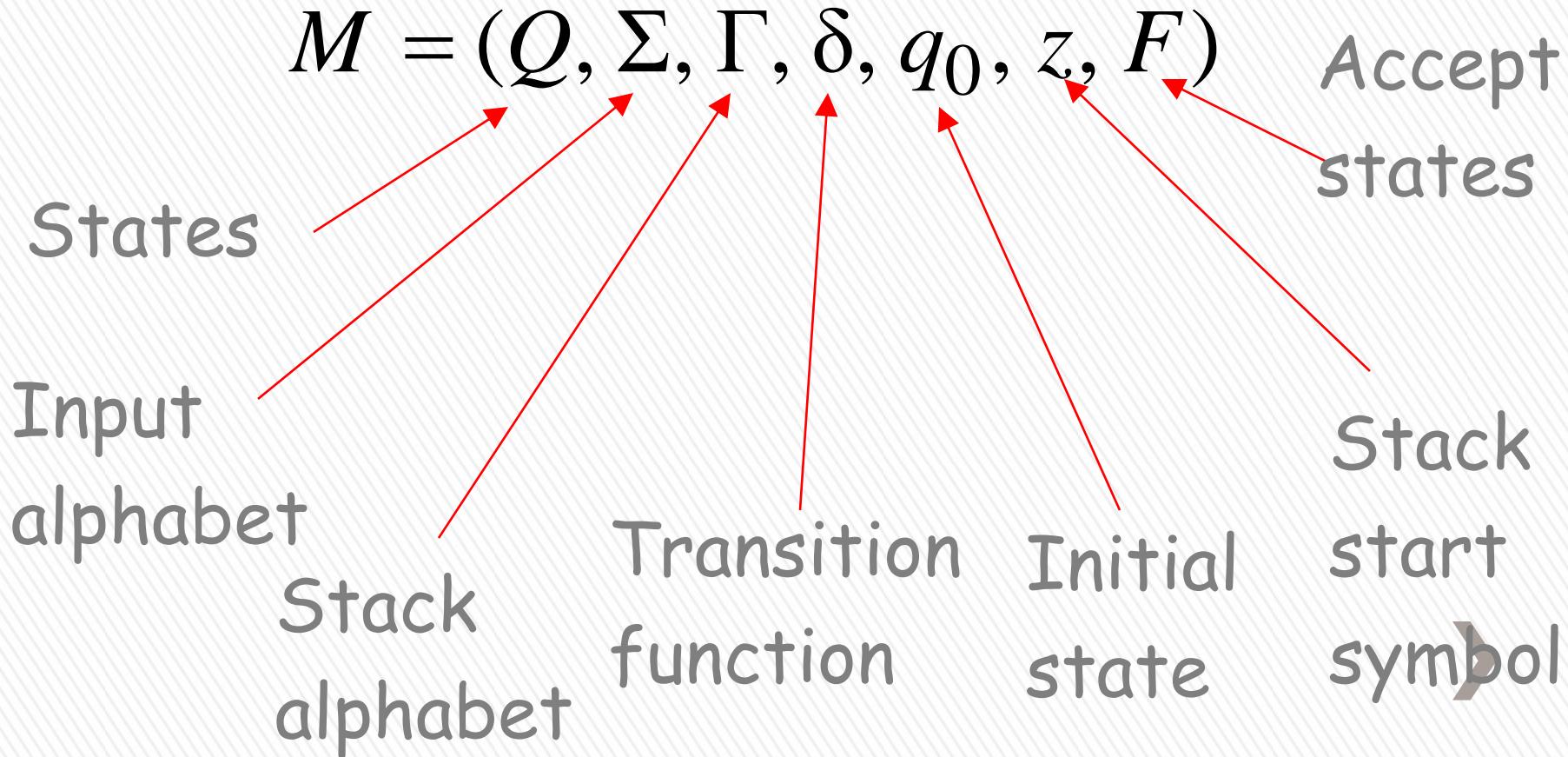
Transition function:

$$\delta(q_1, a, w_1) = \{(q_2, w_2), (q_3, w_3)\}$$

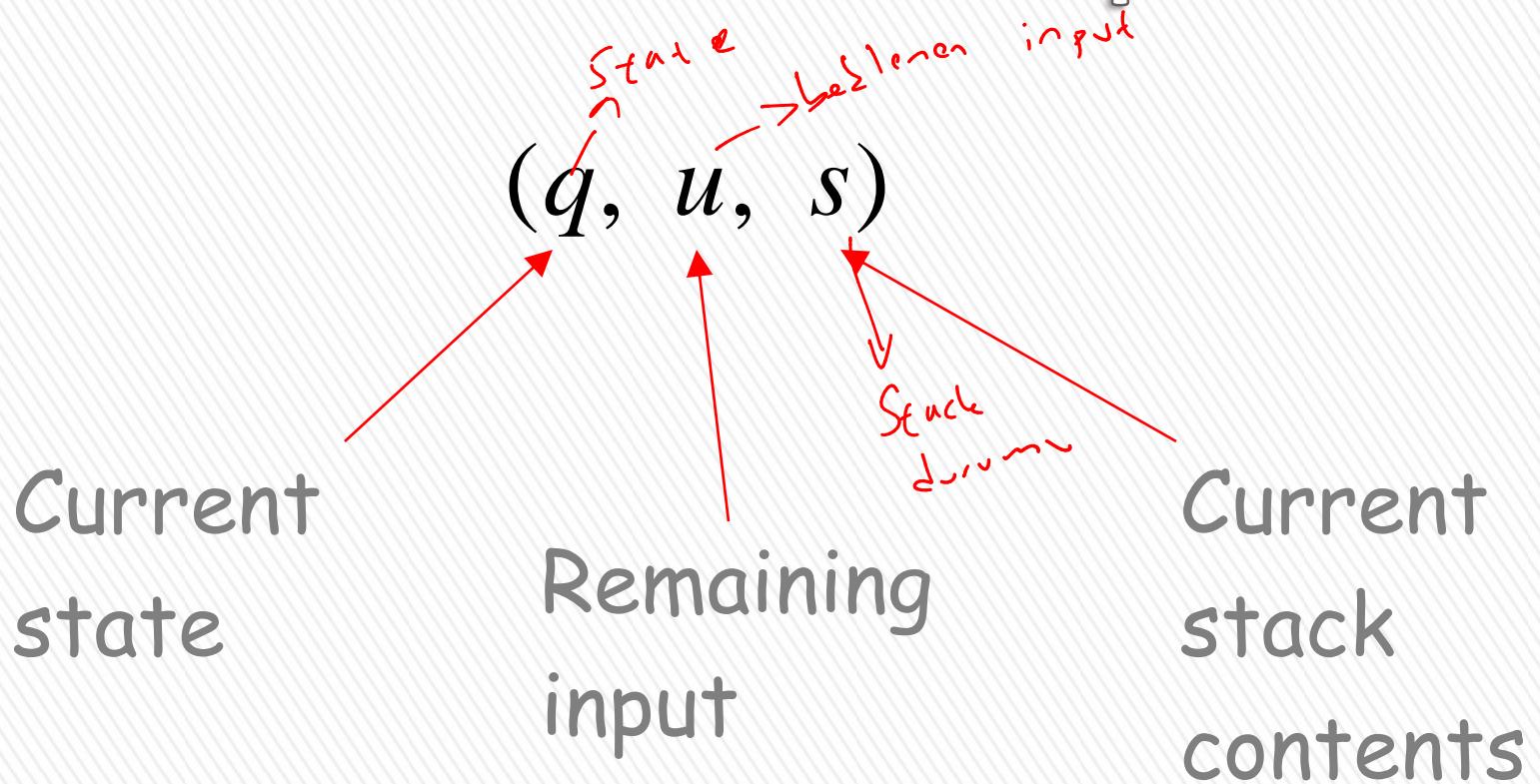


Formal Definition

Pushdown Automaton (PDA)



Instantaneous Description



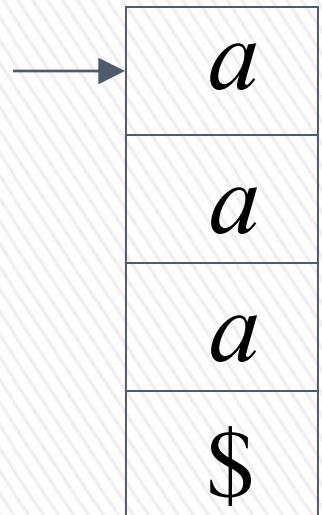
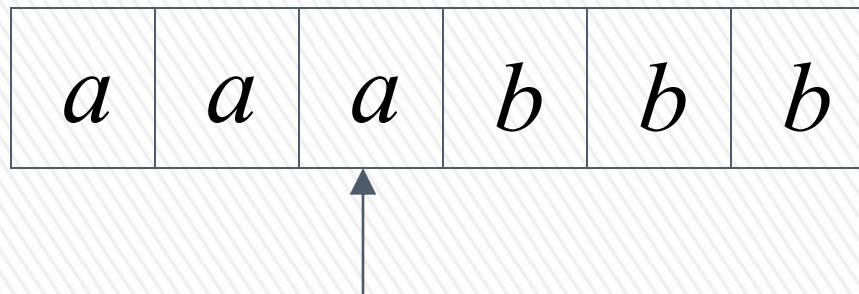
Example:

Instantaneous Description

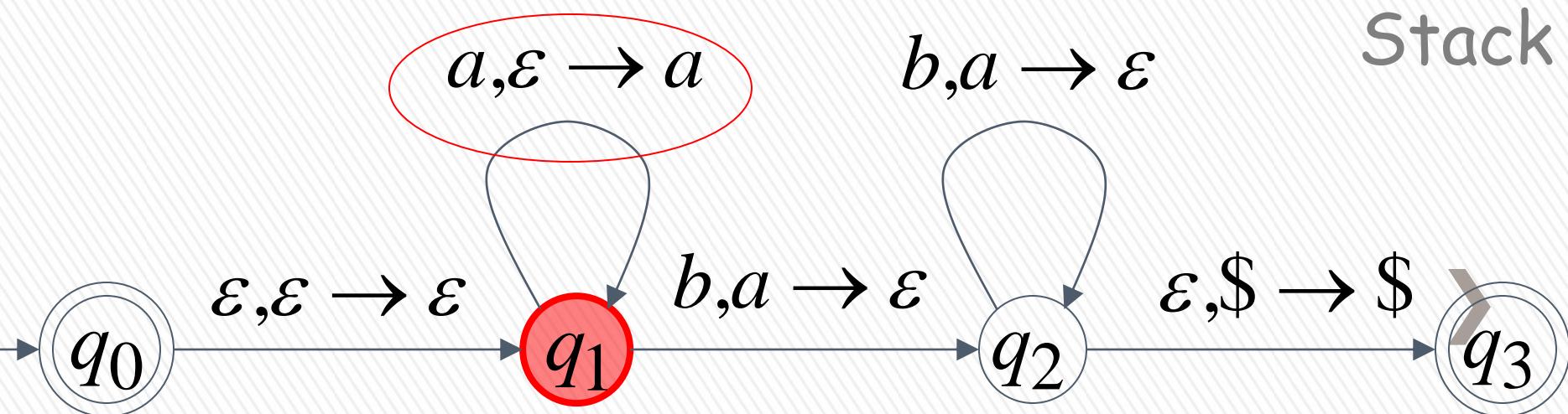
$$(q_1, bbb, aaa\$)$$

Time 4:

Input



Stack



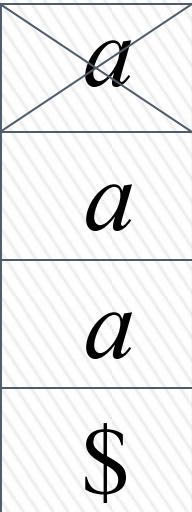
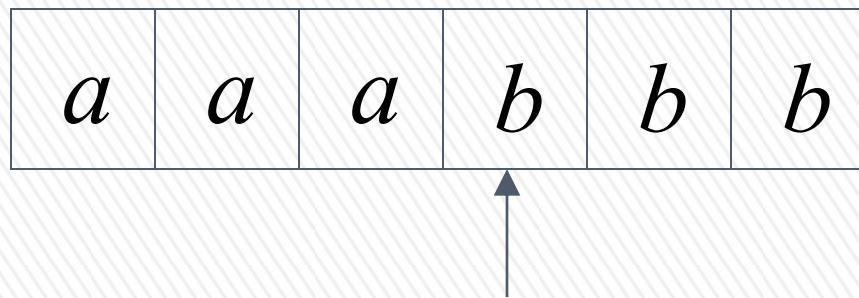
Example:

Instantaneous Description

$(q_2, bb, aa\$)$

Time 5:

Input



$a, \epsilon \rightarrow a$

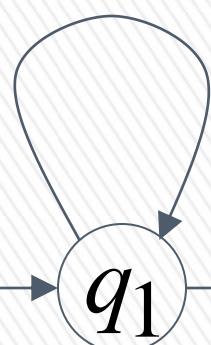
$b, a \rightarrow \epsilon$

Stack

$\epsilon, \epsilon \rightarrow \epsilon$

$b, a \rightarrow \epsilon$

$\epsilon, \$ \rightarrow \$$



ϵ

a

b

$\$$

We write:

$$(q_1, bbb, aaa\$) \succ (q_2, bb, aa\$)$$

Time 4

Time 5

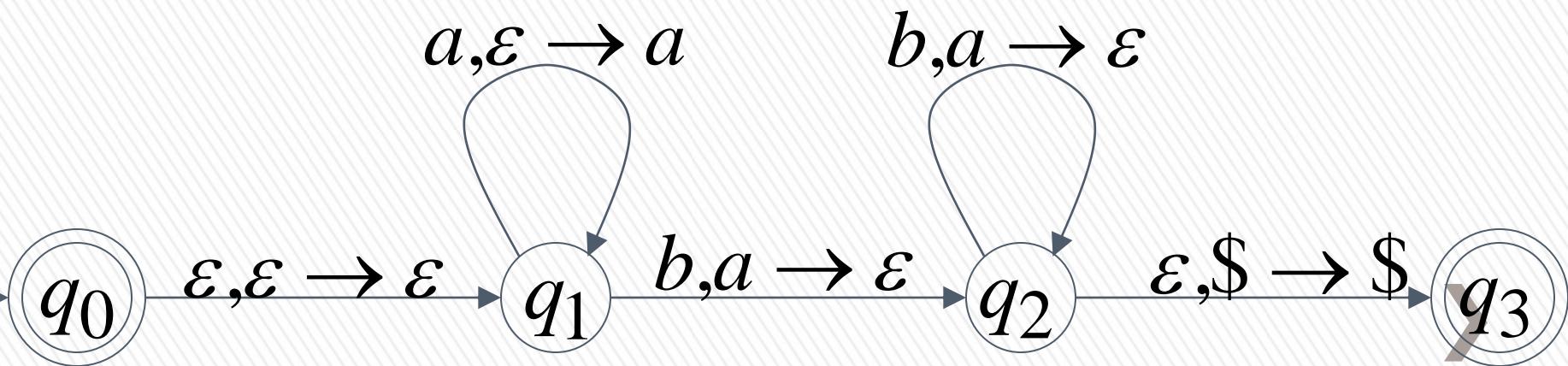


A computation:

$(q_0, aaabbb, \$) \succ (q_1, aaabbb, \$) \succ$

$(q_1, aabbb, a\$) \succ (q_1, abbb, aa\$) \succ (q_1, bbb, aaa\$) \succ$

$(q_2, bb, aa\$) \succ (q_2, b, a\$) \succ (q_2, \epsilon, \$) \succ (q_3, \epsilon, \$)$



$$(q_0, aaabbb, \$) \succ (q_1, aaabbb, \$) \succ$$
$$(q_1, aabbb, a\$) \succ (q_1, abbb, aa\$) \succ (q_1, bbb, aaa\$) \succ$$
$$(q_2, bb, aa\$) \succ (q_2, b, a\$) \succ (q_2, \varepsilon, \$) \succ (q_3, \varepsilon, \$)$$

For convenience we write:

$$(q_0, aaabbb, \$) \xrightarrow{*} (q_3, \varepsilon, \$)$$


Language of PDA

→ CFL
L ::= L₁ ∪ L₂ ∪ L₃

Language $L(M)$ accepted by PDA M :

$$L(M) = \{ w : (q_0, w, z) \xrightarrow{*} (q_f, \epsilon, s) \}$$

Initial state

Accept state



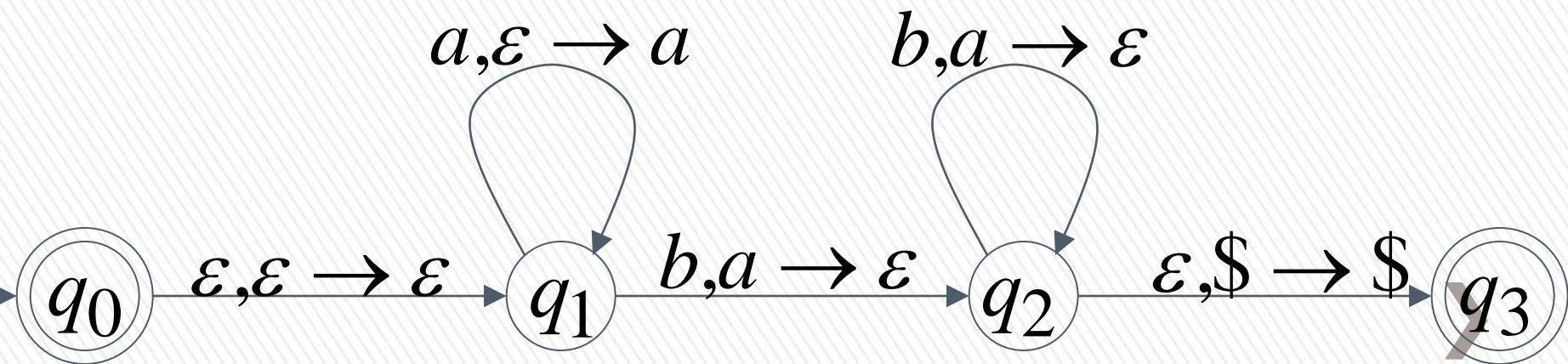
Example:

$$(q_0, aaabbb, \$) \xrightarrow{*} (q_3, \epsilon, \$)$$



$$aaabbb \in L(M)$$

PDA M :

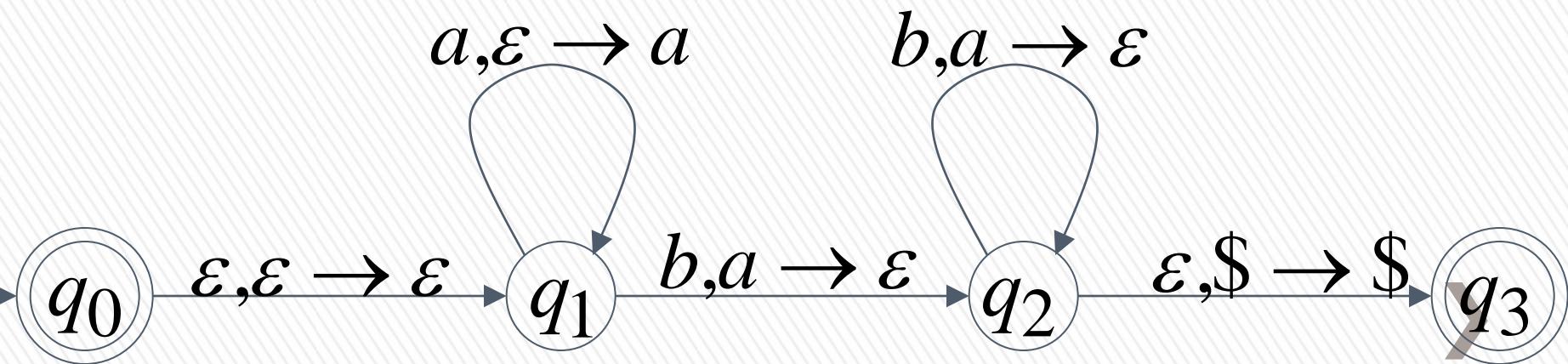


$$(q_0, a^n b^n, \$) \xsucc{*} (q_3, \epsilon, \$)$$



$$a^n b^n \in L(M)$$

PDA M :



Therefore:

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

PDA M :

