

Formal Language & Automata Theory (CS303)

Instructor: Abyayananda Maiti

Mail id: abyaym@iitp.ac.in

Phone: 8130

Syllabus

Alphabet, languages and grammars.

Regular languages and finite automata: Regular expressions and languages, deterministic finite automata (DFA), nondeterministic finite automata (NFA), regular grammars and equivalence with finite automata, properties of regular languages, pumping lemma for regular languages.

Context-free languages and pushdown automata: Context-free grammars (CFG) and languages (CFL), Chomsky and Greibach normal forms, nondeterministic pushdown automata (PDA), ambiguity in CFG, pumping lemma for context-free languages, deterministic pushdown automata.

Context-sensitive languages: Context-sensitive grammars (CSG) and languages, linear bounded automata.

Turing machines: The basic model for Turing machines (TM), Turing- recognizable (recursively enumerable) and Turing-decidable (recursive), variants of Turing machines

Undecidability: Church-Turing thesis, universal Turing machine, reduction between languages and Rice's theorem, undecidable problems about languages.

Books

1. J. E. Hopcroft, R. Motwani and J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Pearson Education India (3rd edition).
2. K. L. P. Mishra, N. Chandrasekaran, Theory of Computer Science: Automata, Languages and Computation, PHI Learning Pvt. Ltd. (3rd edition).
3. D. I. A. Cohen, Introduction to Computer Theory, John Wiley & Sons, 1997.
4. J. C. Martin, Introduction to Languages and the Theory of Computation, Tata McGraw-Hill (3rd Ed.).
5. H. R. Lewis and C. H. Papadimitriou, Elements of the Theory of Computation, Prentice Hall, 1997.
6. Garey, D.S., Johnson, G., Computers and Intractability: A Guide to the Theory of NP- Completeness, Freeman, New York, 1979

Evaluation Policy

EndSem - 25%

MidSem - 25%

Class performance etc. - 10%

Regular assessments - 40%

Why Study Automata Theory and Formal Languages?

- A survey of Stanford grads 5 years out asked which of their courses did they use in their job.
- Basics like Programming took the top spots, of course.
- But among optional courses, Automata Theory stood remarkably high.
 - 3X the score for AI, for example.

Why Finite Automata and Regular Expressions?

- Regular expressions (REs) are used in many systems.
 - E.g., UNIX, Linux, OS X,... $a.^*b$.
 - E.g., Document Type Definitions describe XML tags with a RE format like person (name, addr, child*).
- Finite automata model protocols, electronic circuits.
 - Theory is used in model-checking.

Why Context-Free Grammars?

- Context-free grammars (CFGs) are used to describe the syntax of essentially every modern programming language.
- Every modern compiler uses CFG concepts to parse programs
 - Not to forget their important role in describing natural languages.
- And Document Type Definitions are really CFG's.

Why Turing Machines?

- When developing solutions to real problems, we often confront the limitations of what software can do.
 - Undecidable things - no program can do it 100% of the time with 100% accuracy.
 - Intractable things - there are programs, but no fast programs.
- A course on Automata Theory and Formal Languages gives you the tools.

Other Good Stuff

- We'll learn how to deal formally with discrete systems.
 - **Proofs:** You never really prove a program correct, but you need to be thinking of why a tricky technique really works.
- You'll gain experience with abstract models and constructions.
 - Models layered software architectures.

Course Outline

- Regular Languages and their descriptors:
 - Finite automata, nondeterministic finite automata, regular expressions.
 - Algorithms to decide questions about regular languages, e.g., is it empty?
 - Closure properties of regular languages.

Course Outline - (2)

- Context-free languages and their descriptors:
 - Context-free grammars, pushdown automata.
 - Decision and closure properties.

Course Outline - (3)

- Recursive and recursively enumerable languages.
 - Turing machines, decidability of problems.
 - The limit of what can be computed.
- Intractable problems.
 - Problems that (appear to) require exponential time.
 - NP-completeness and beyond.

Concepts and Notations

- **Alphabet:** A finite, nonempty set of symbols.
Conventionally, we use the symbol Σ for an alphabet.
 - Examples:
 - The set of all ASCII characters, or the set of all printable ASCII characters.
 - $\Sigma_1 = \{ a, b \}$
 - $\Sigma_2 = \{ \text{Spring, Summer, Autumn, Winter} \}$
 - $\Sigma_3 = \{ 0, 1 \}$
- **String:** A finite sequence of zero or more symbols from an alphabet.
 - The empty string: ϵ
 - 01101 is a string from the binary alphabet $\Sigma = \{ 0, 1 \}$

Concepts and Notations

- **Powers of an Alphabet:** If Σ is an alphabet, we denote by Σ^k the set of all strings of length k .
 - Examples: Let $\Sigma = \{a, b, c\}$
 - $\Sigma^0 = \epsilon$
 - $\Sigma^1 = \{a, b, c\}$
 - $\Sigma^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$
 - $\Sigma^3 = \{aaa, aab, aac, aba, abb, abc, aca, acb, \dots\}$

Σ^* = The set of all strings over $\Sigma = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$

Σ^+ = The set of nonempty strings over $\Sigma = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$

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

- Exercise: Given $\Sigma = \{0, 1\}$, compute Σ^+ and Σ^* .

Formal Language

- **Language:** A set of strings over an alphabet.
 - If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a language over Σ .
 - Also known as a **formal language**.
- Examples:
 - The language of all strings consisting of n 0's followed by n 1's for some $n \geq 0$:
$$\{\epsilon, 01, 0011, 000111, \dots\}.$$
 - The set of strings with equal numbers of 0's and 1's
$$\{\epsilon, 01, 10, 0011, 0101, 1001, \dots\}$$
 - The set of binary numbers whose value is a prime
$$\{10, 11, 101, 111, 1011, \dots\}$$
 - The empty language, denoted \emptyset , is a language over any alphabet.

Operations on Languages

- Suppose L_1 and L_2 are languages over some common alphabet.
- Union ($L_1 \cup L_2$): $\underline{\{w | w \in L_1 \vee w \in L_2\}}$
- Concatenation ($L_1 \cdot L_2$): $\{w \cdot z | w \in L_1 \wedge z \in L_2\}$
- The Kleene Closure (L_1^*): $\{\epsilon\} \cup \{w \cdot z | w \in L_1 \wedge z \in L_1^*\}$

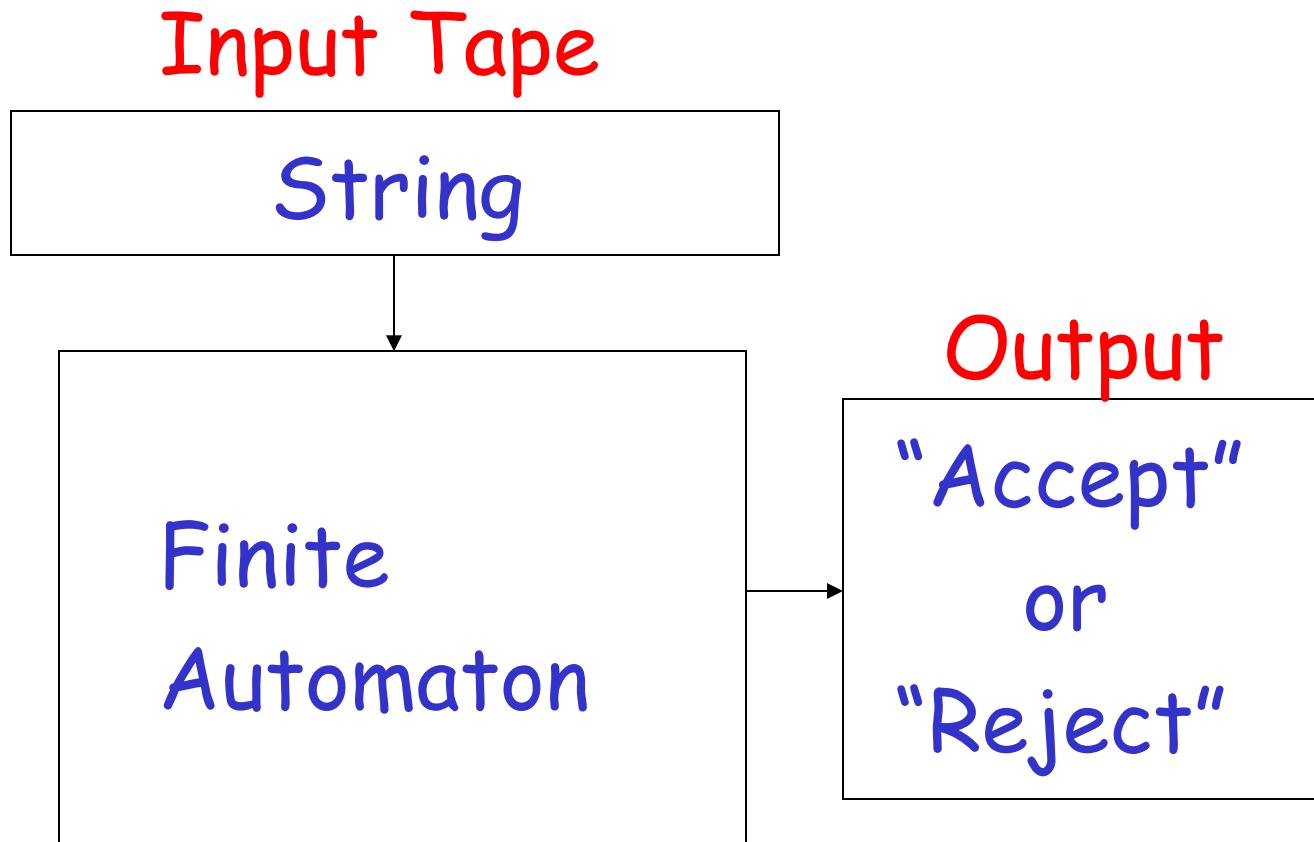
Regular Language

- Regular Languages are the simplest class of formal languages.
- Regular languages can be specified by
 - regular expressions (REs),
 - finite-state automata (FSAs),
 - regular grammars.

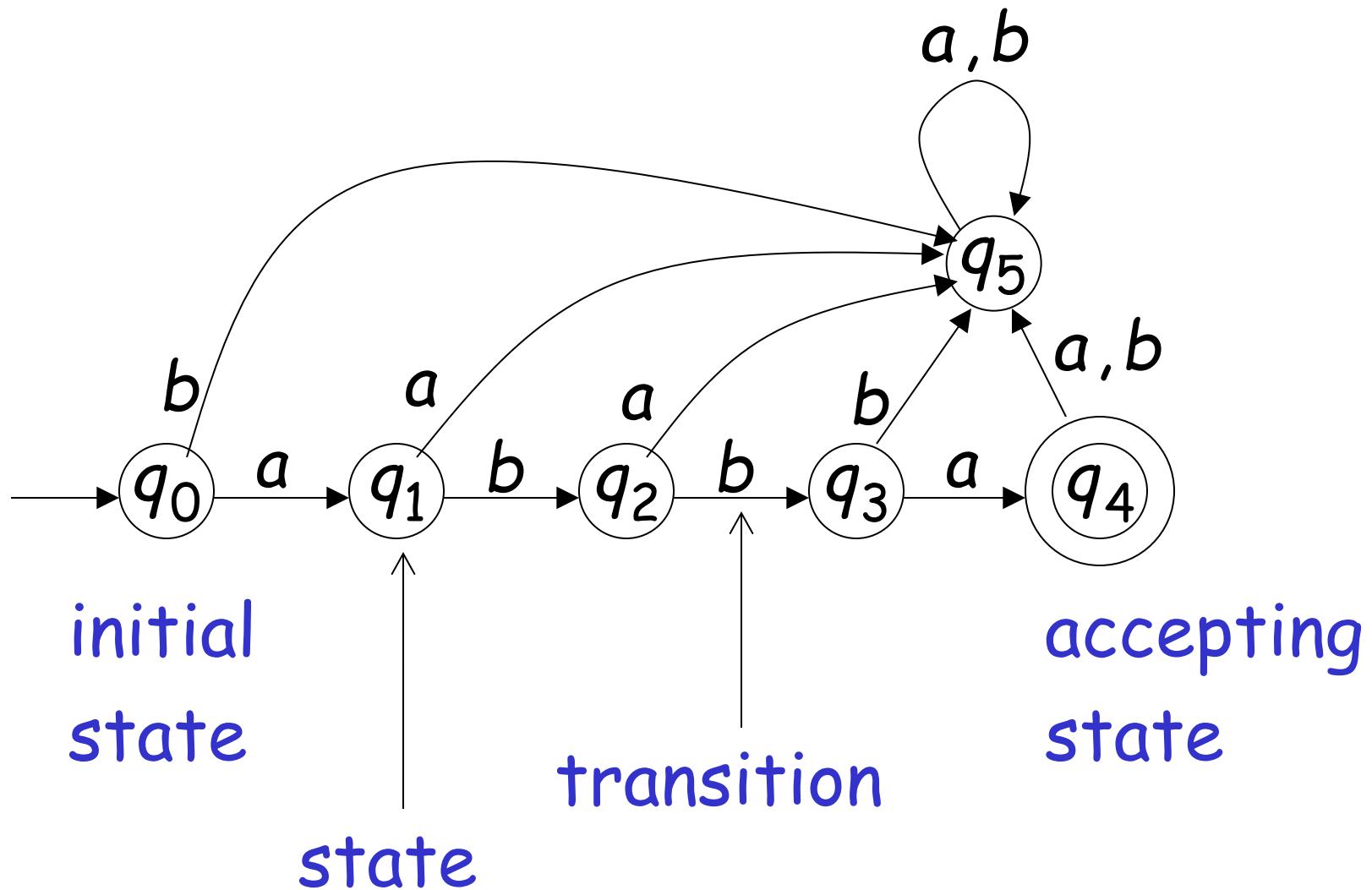
Deterministic Finite Automata

And Regular Languages

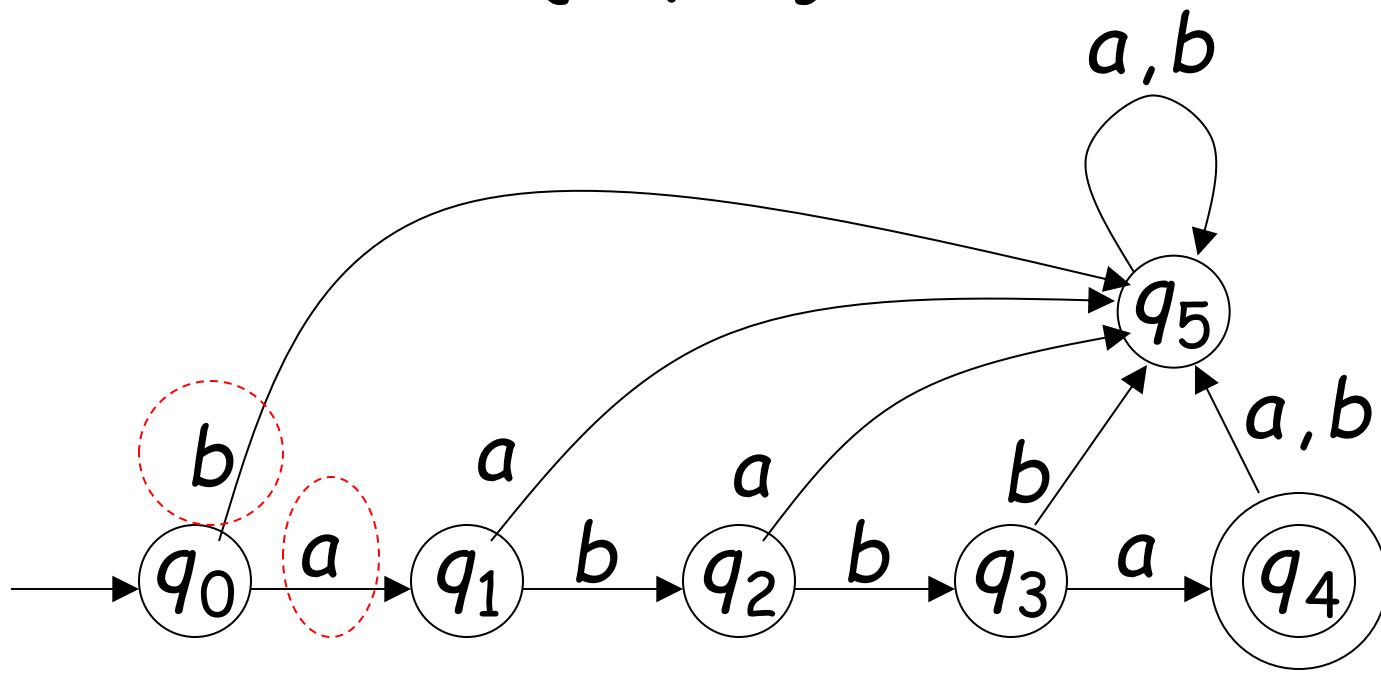
Deterministic Finite Automaton (DFA)



Transition Graph



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



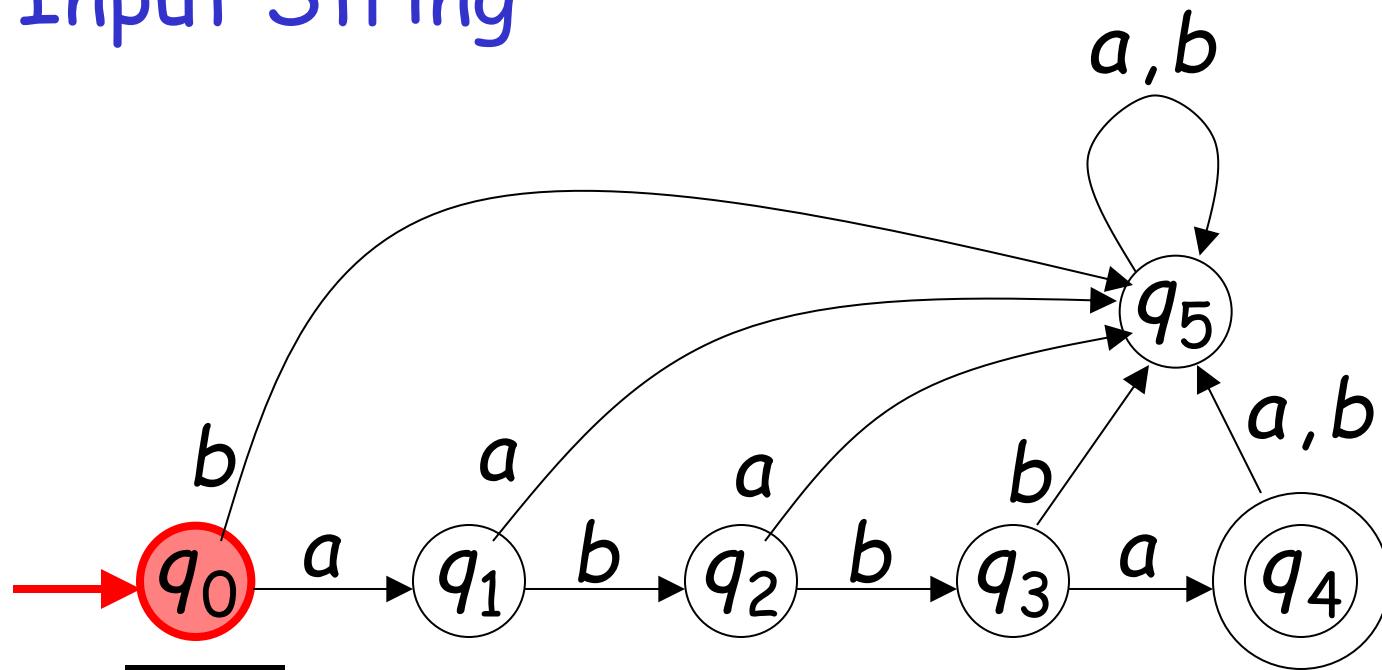
For every state, there is a transition
for every symbol in the alphabet

head

Initial Configuration

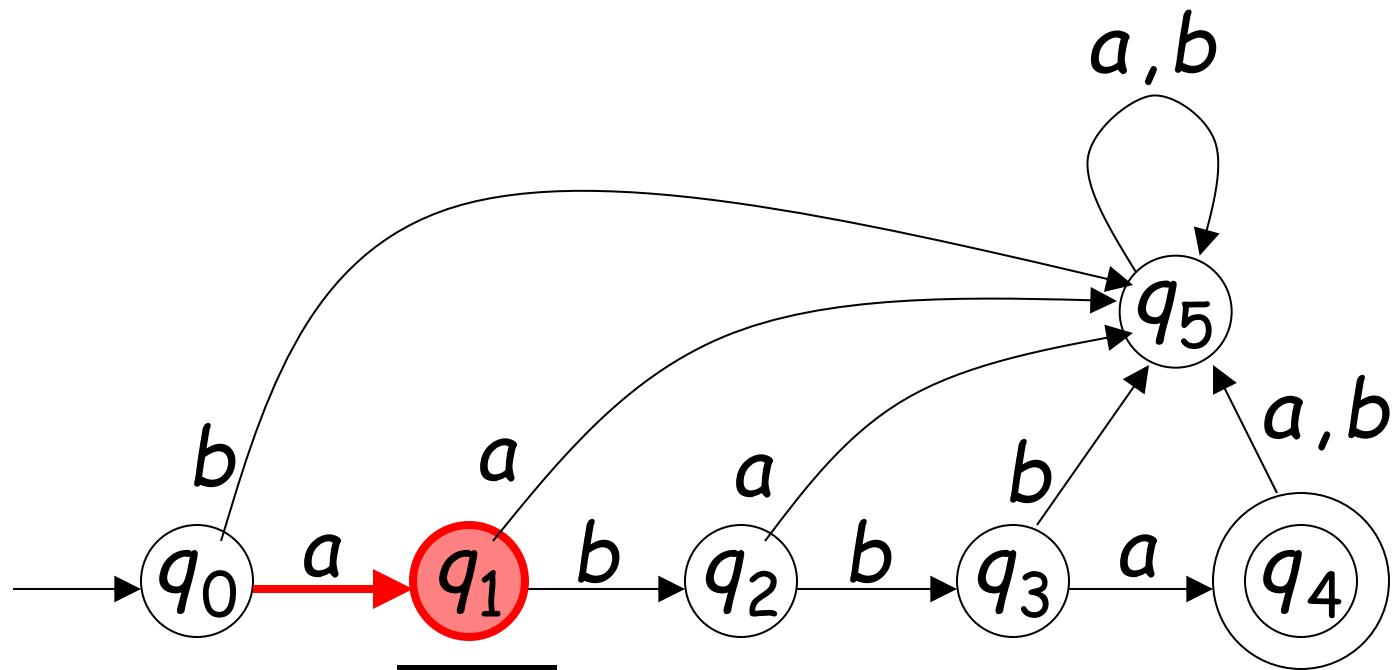
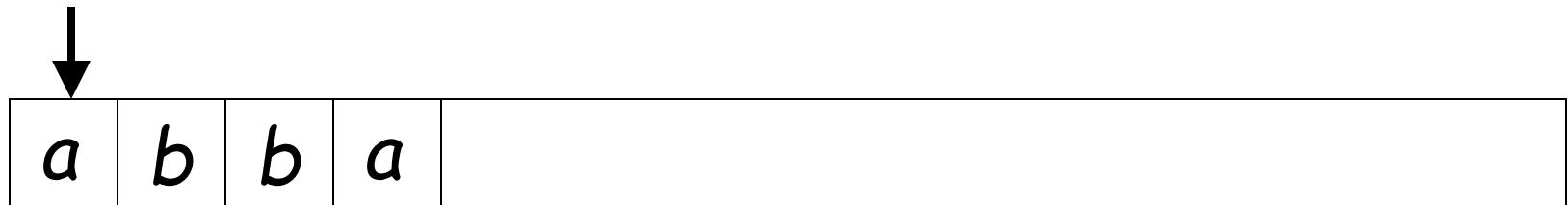


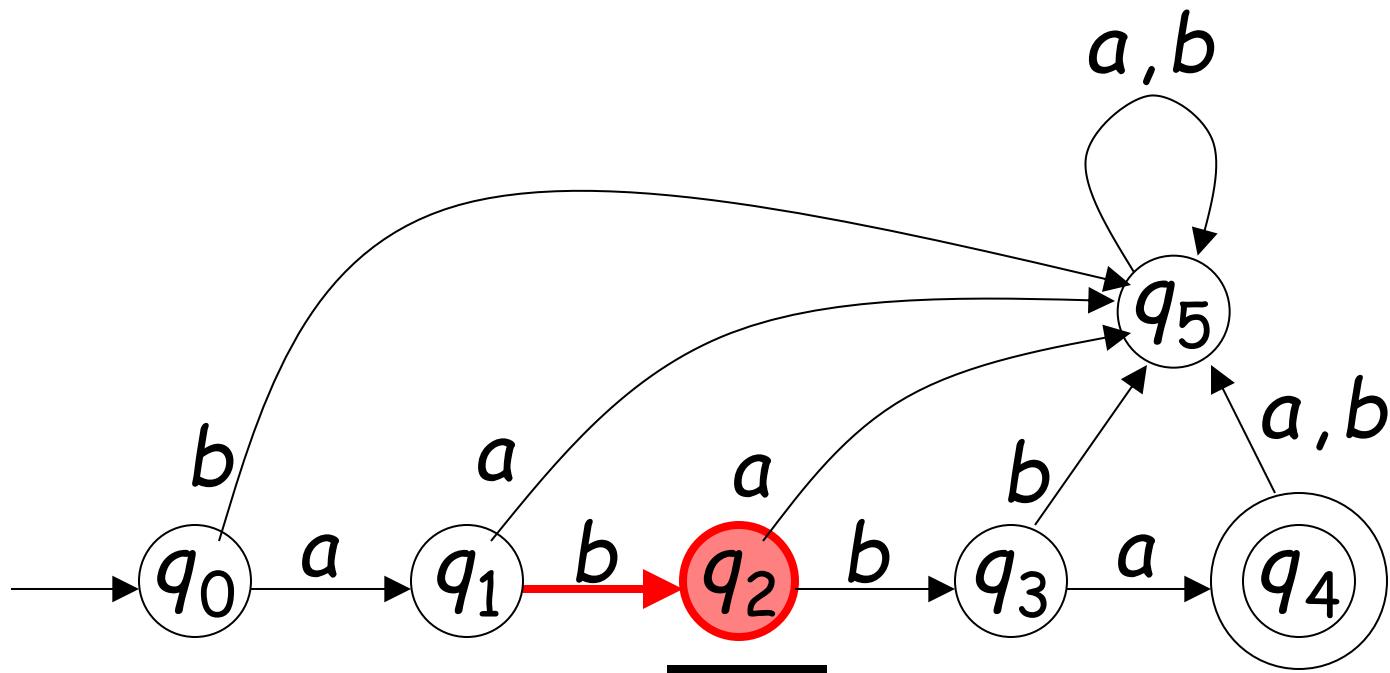
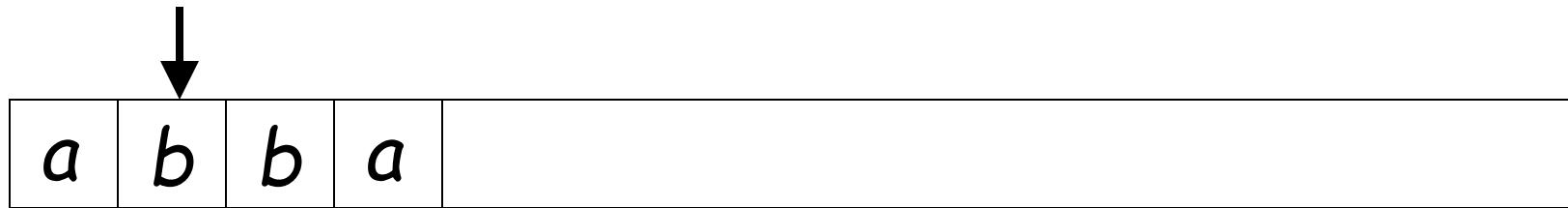
Input String

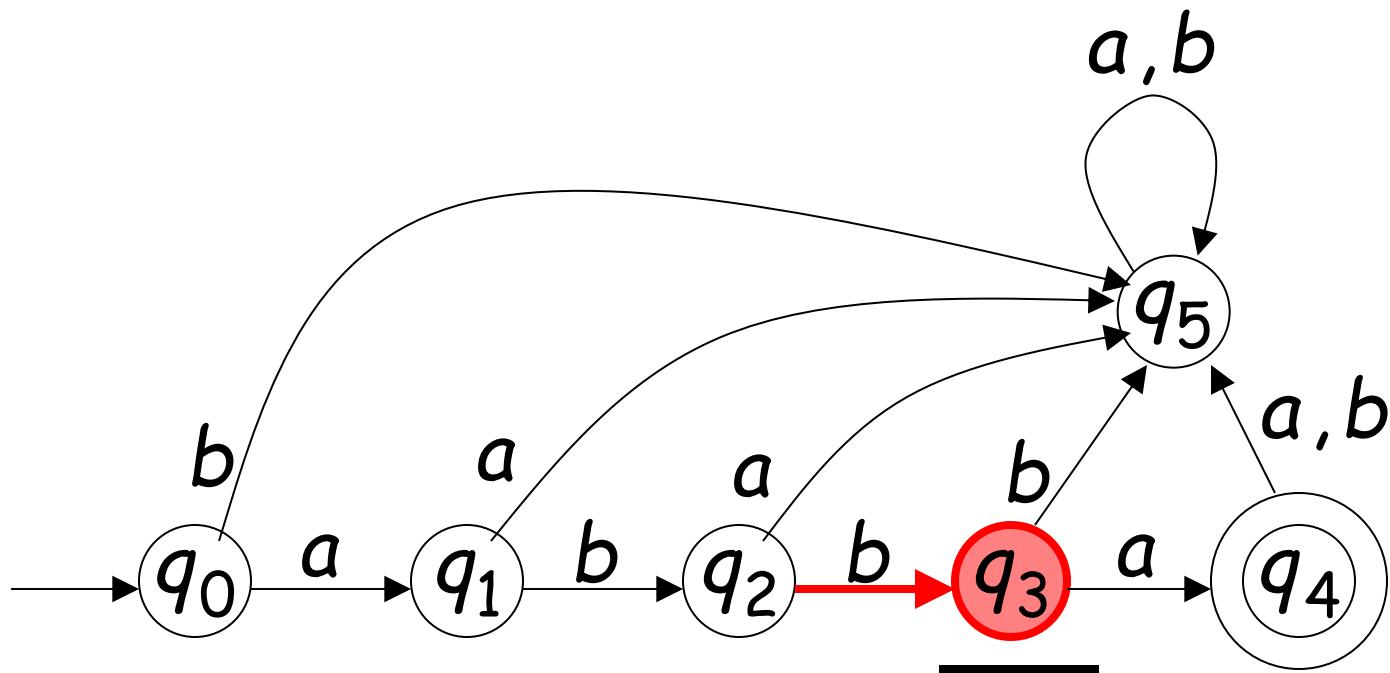
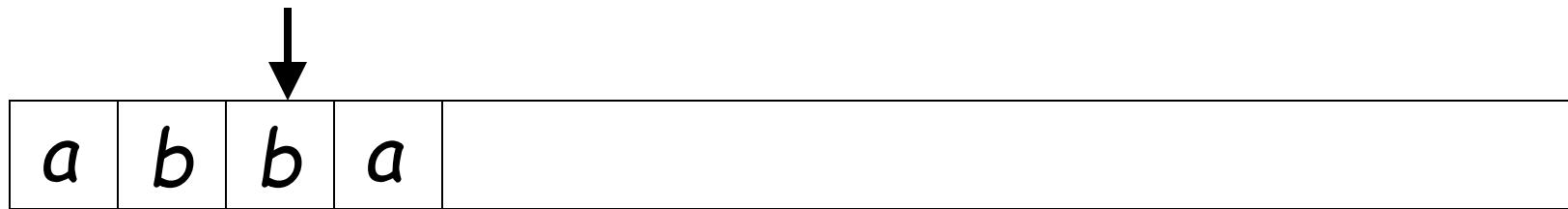


Initial state

Scanning the Input



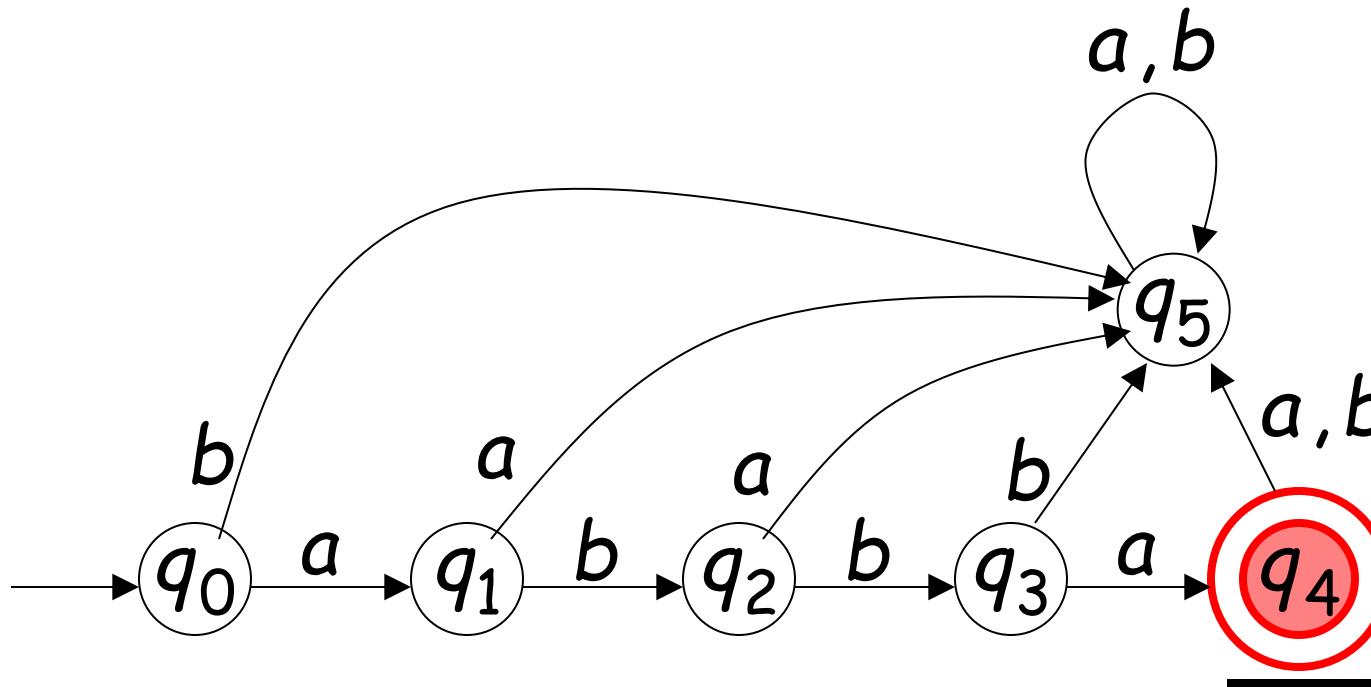




Input finished



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



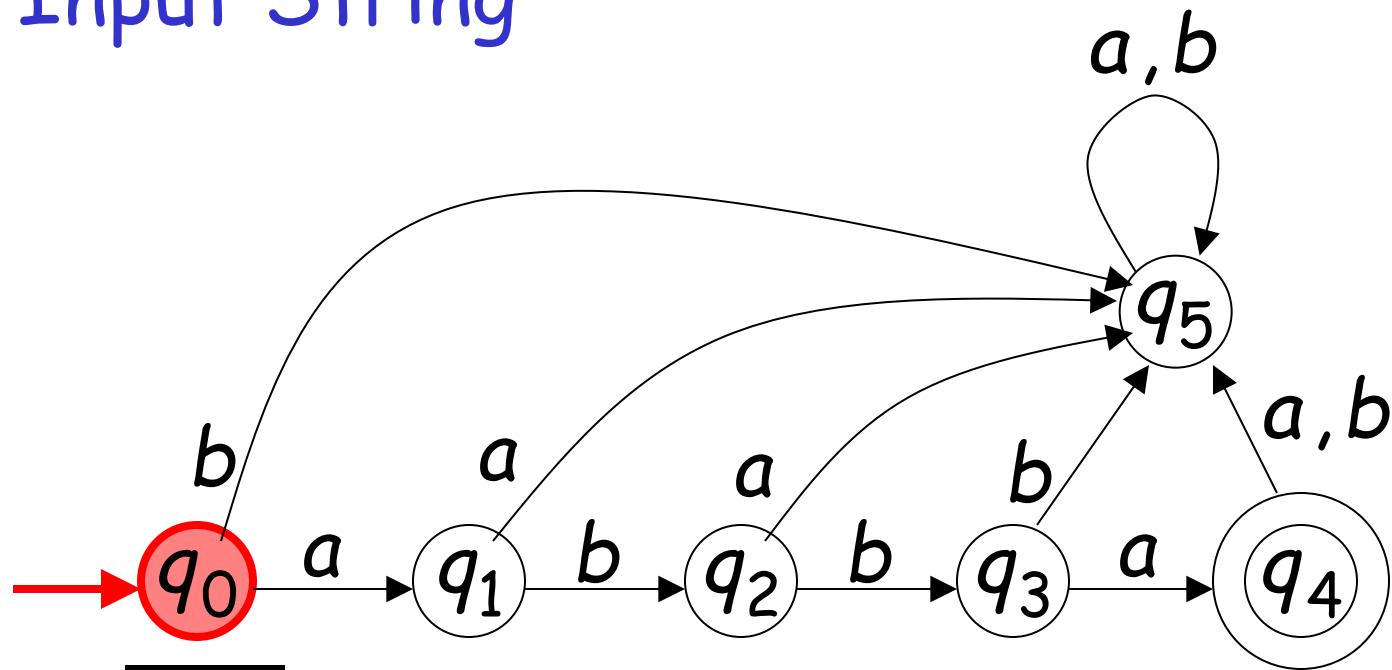
accept

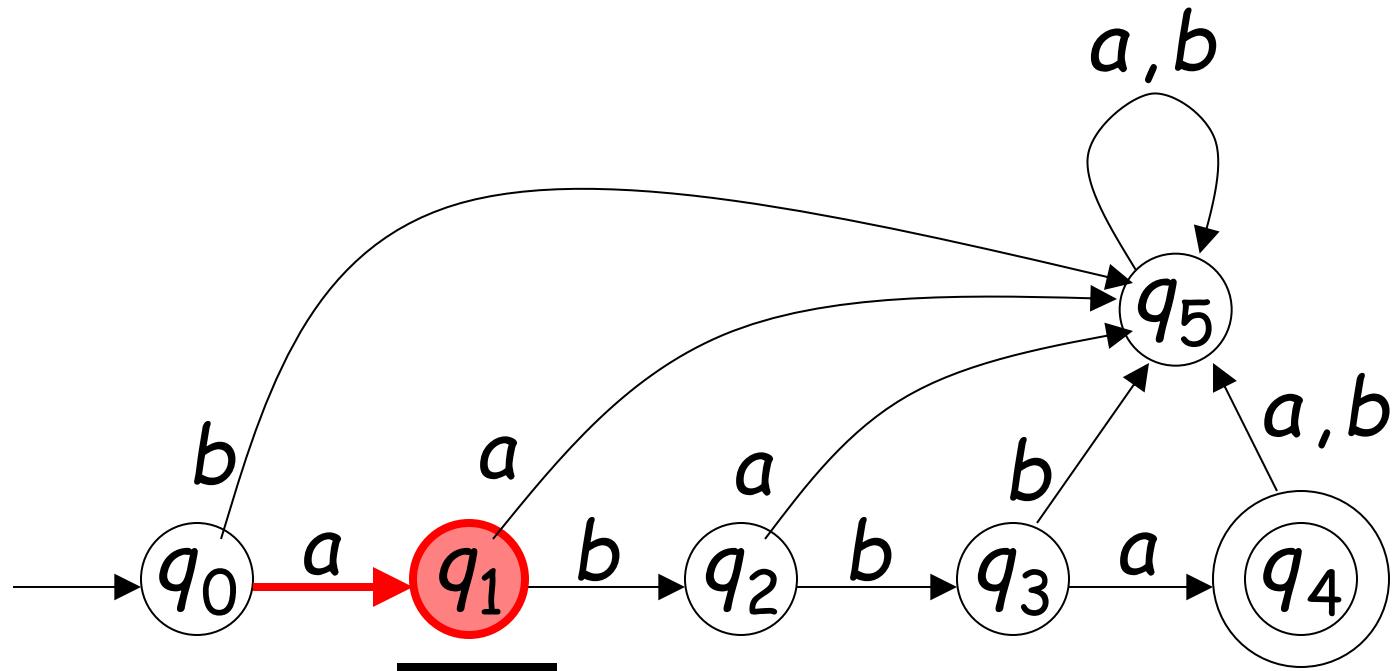
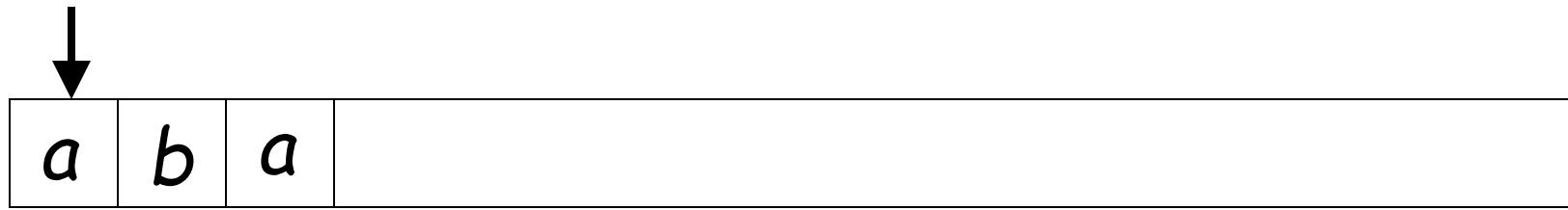
A Rejection Case

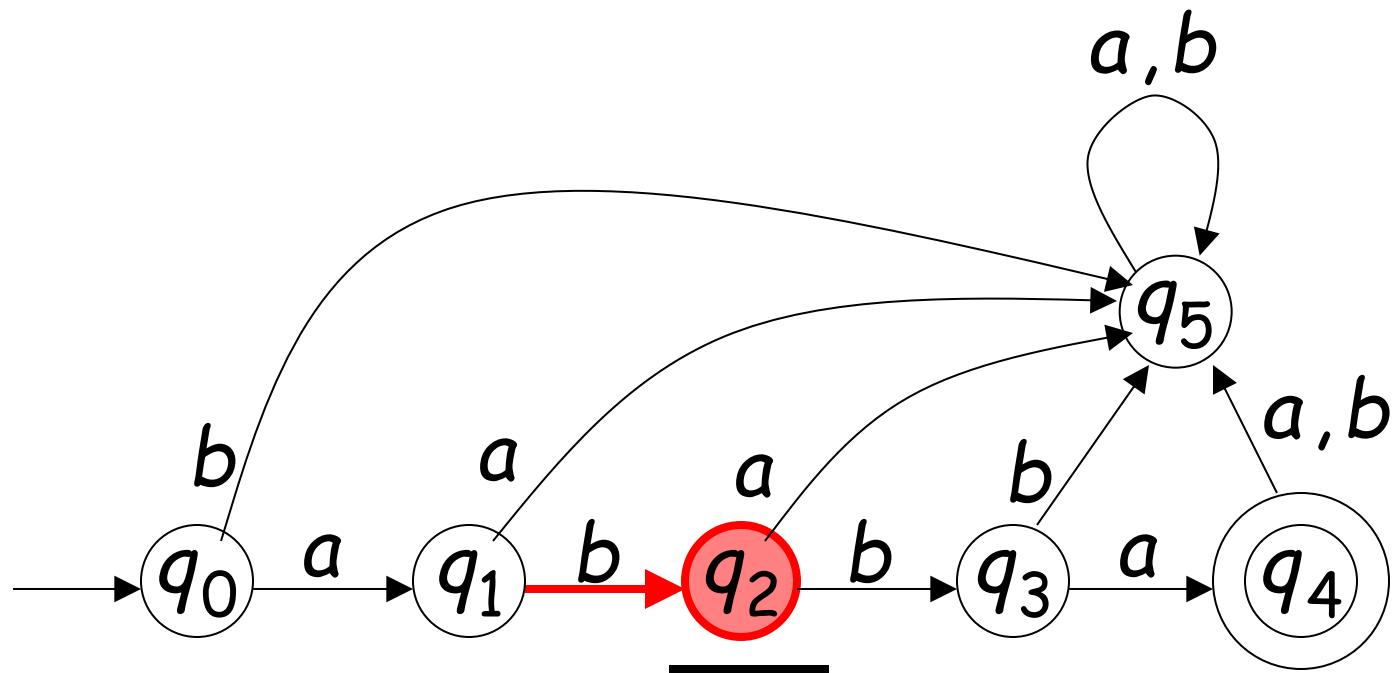
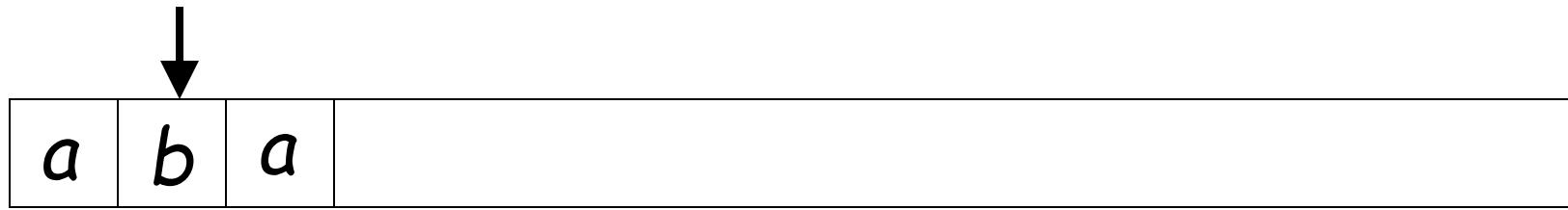


a	b	a	
---	---	---	--

Input String



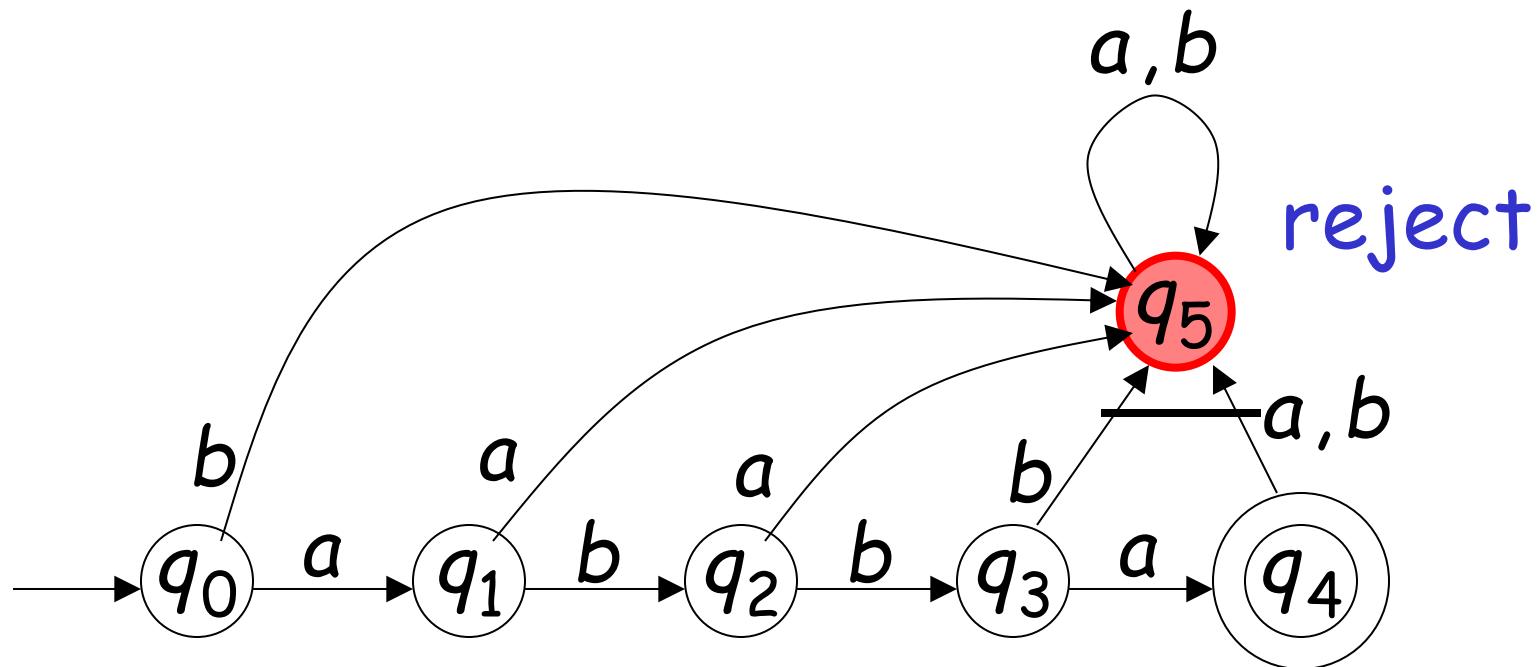




Input finished



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

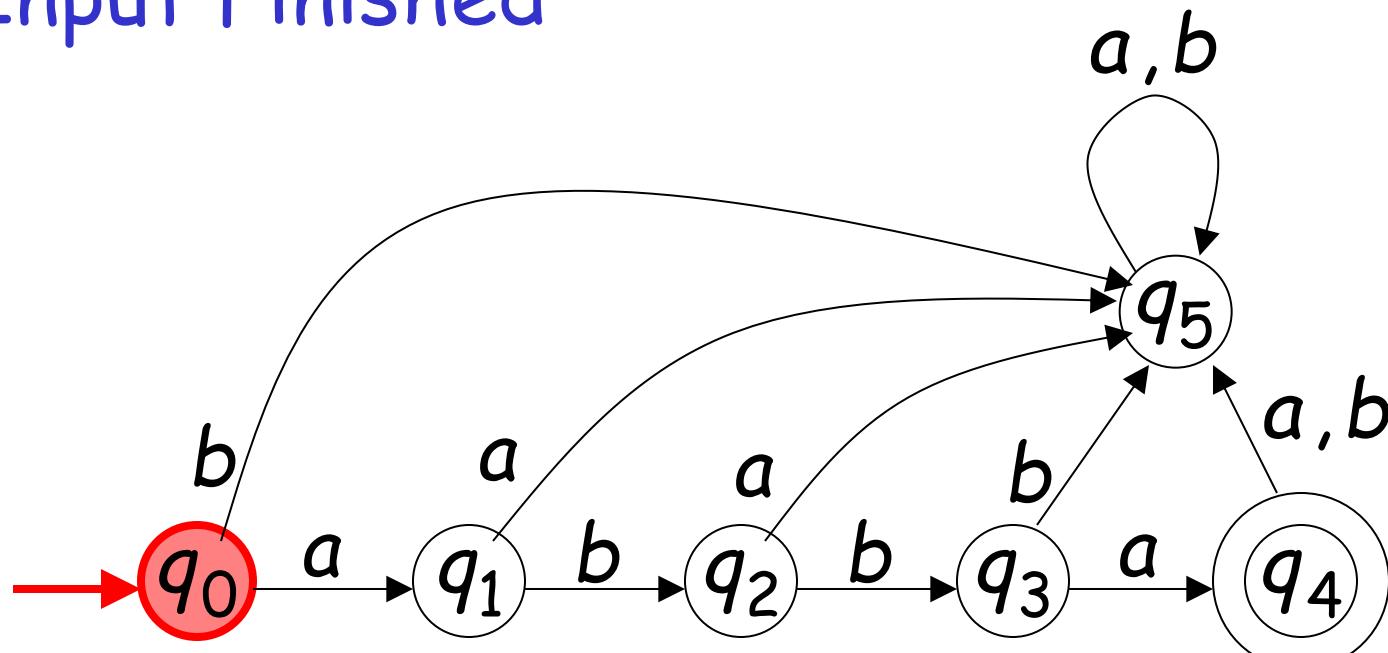


Another Rejection Case

Tape is empty

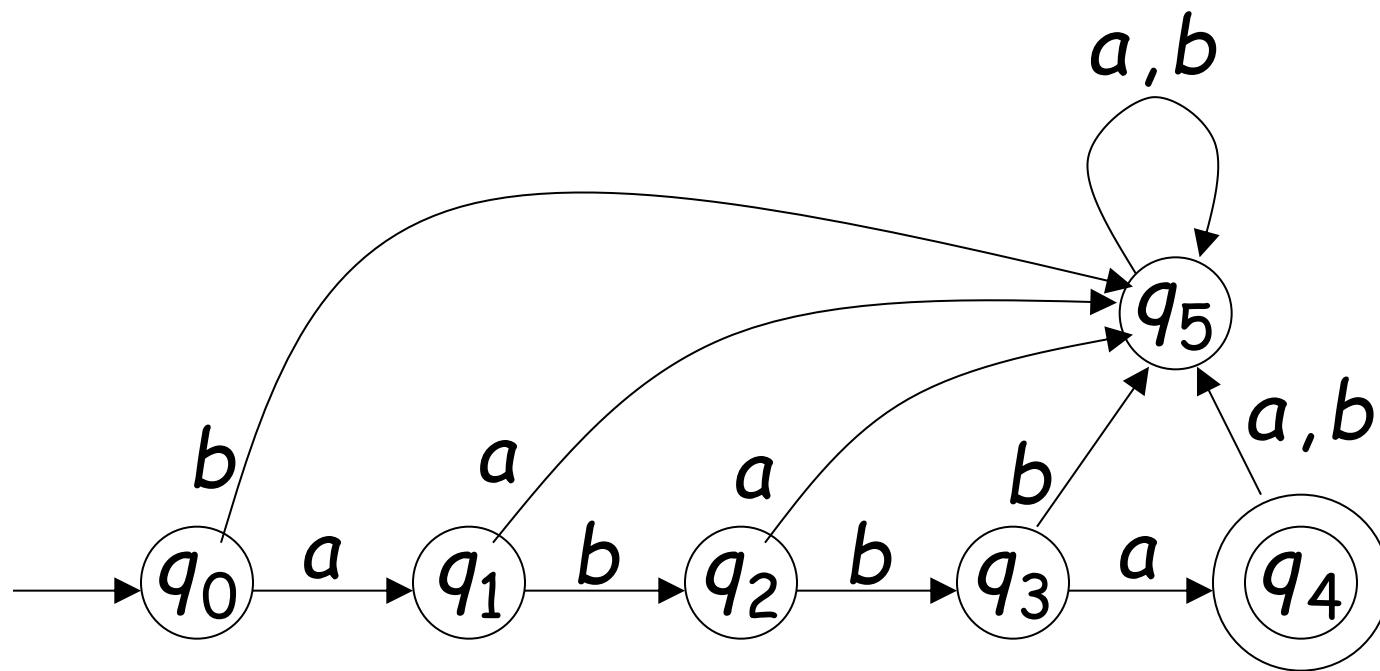
(λ)

Input Finished



reject

Language Accepted: $L = \{abba\}$



To accept a string:

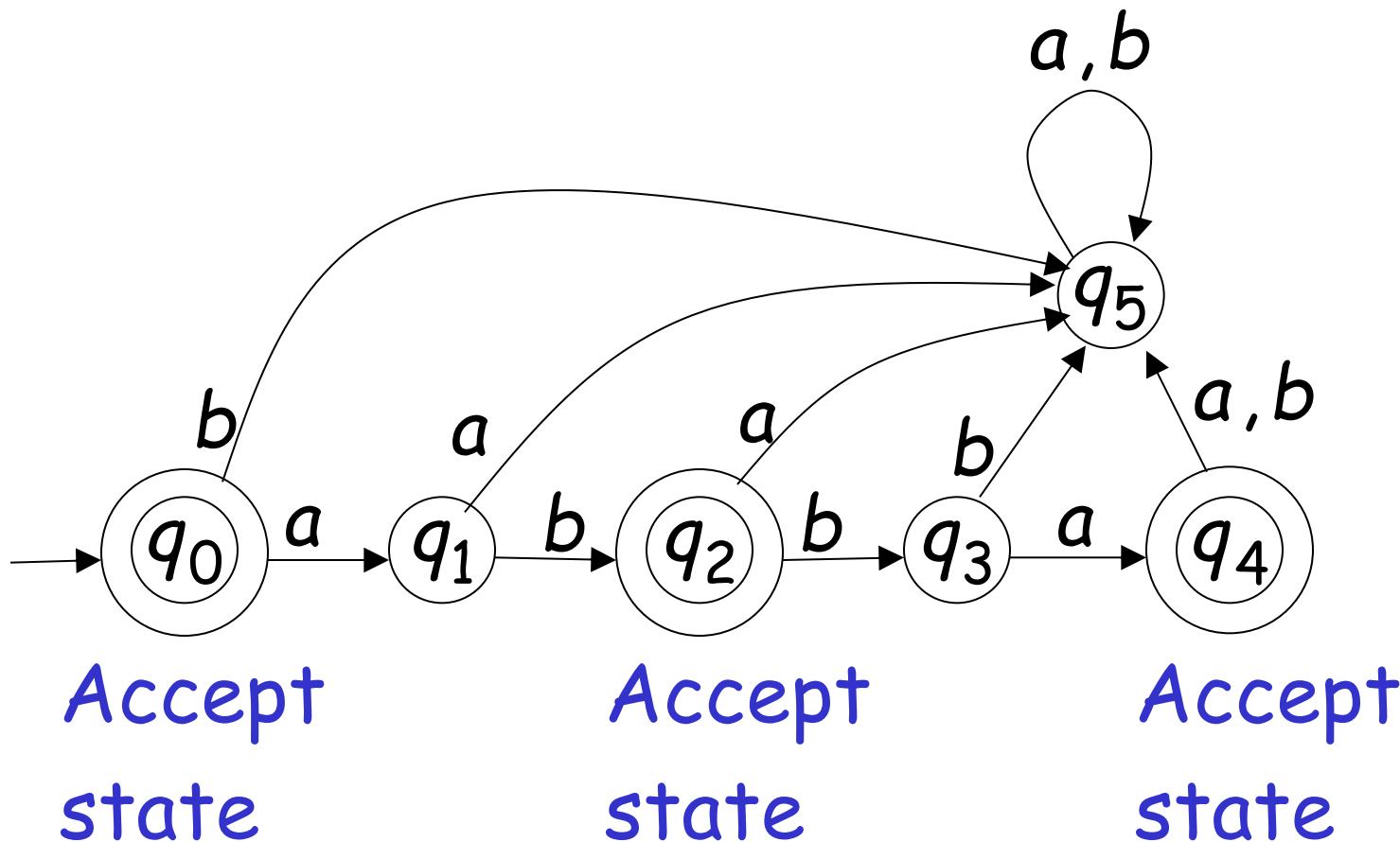
all the input string is scanned
and the last state is accepting

To reject a string:

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

Another Example

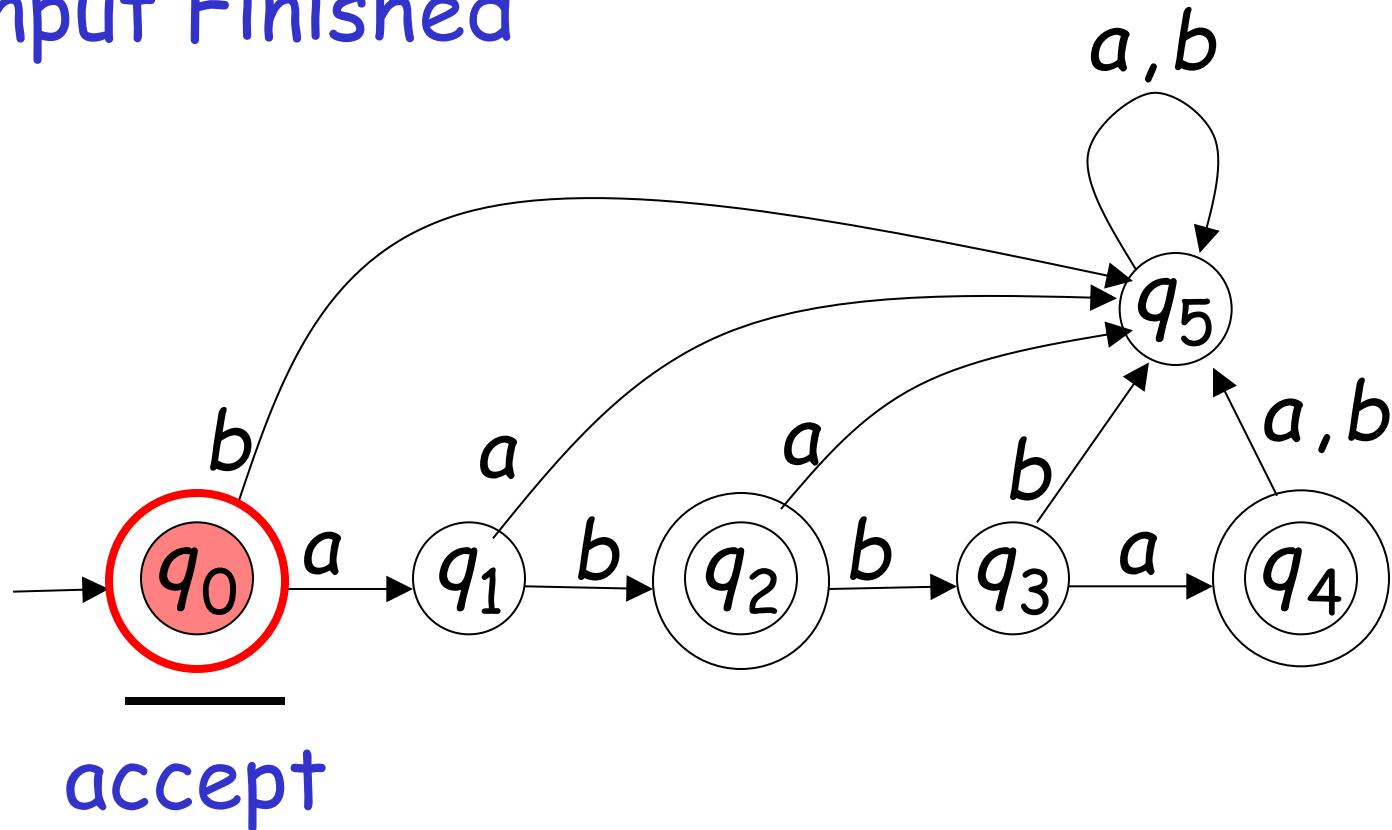
$$L = \{\lambda, ab, abba\}$$



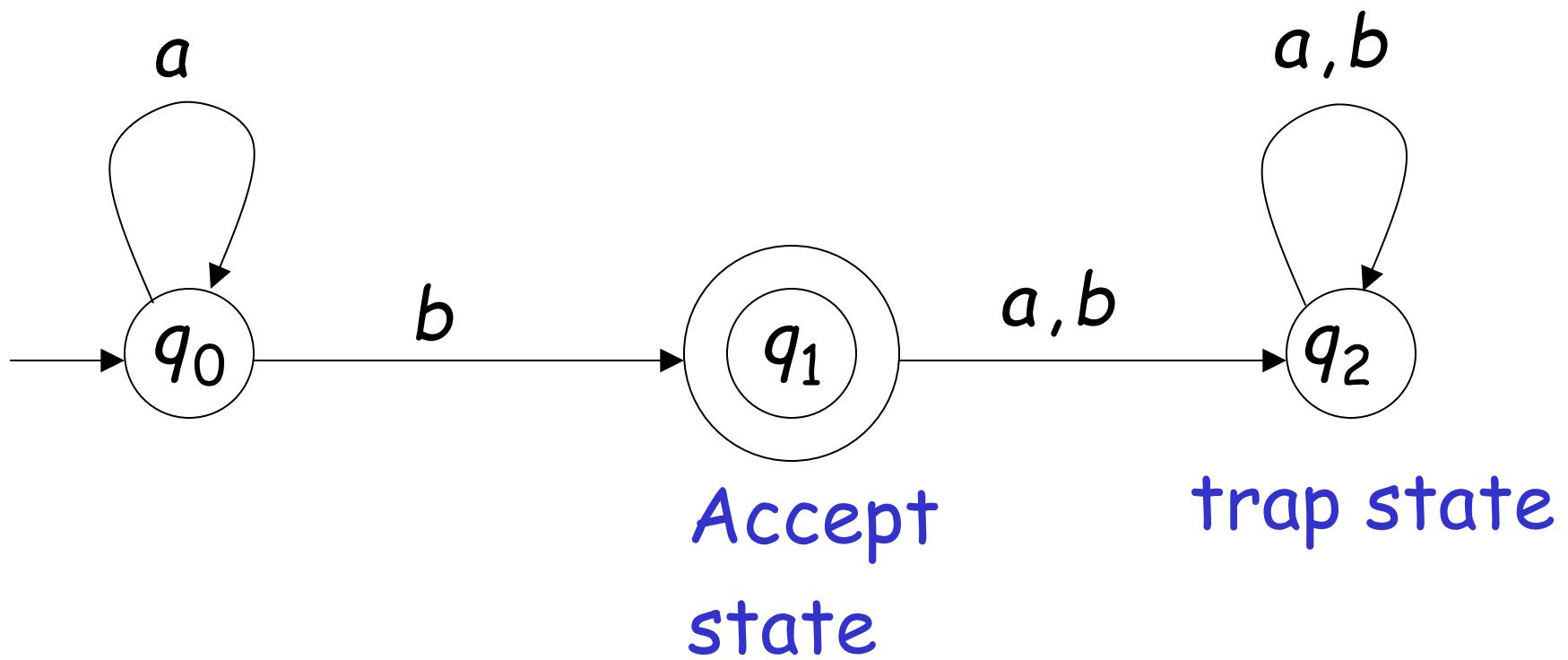
Empty Tape

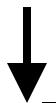
(λ)

Input Finished



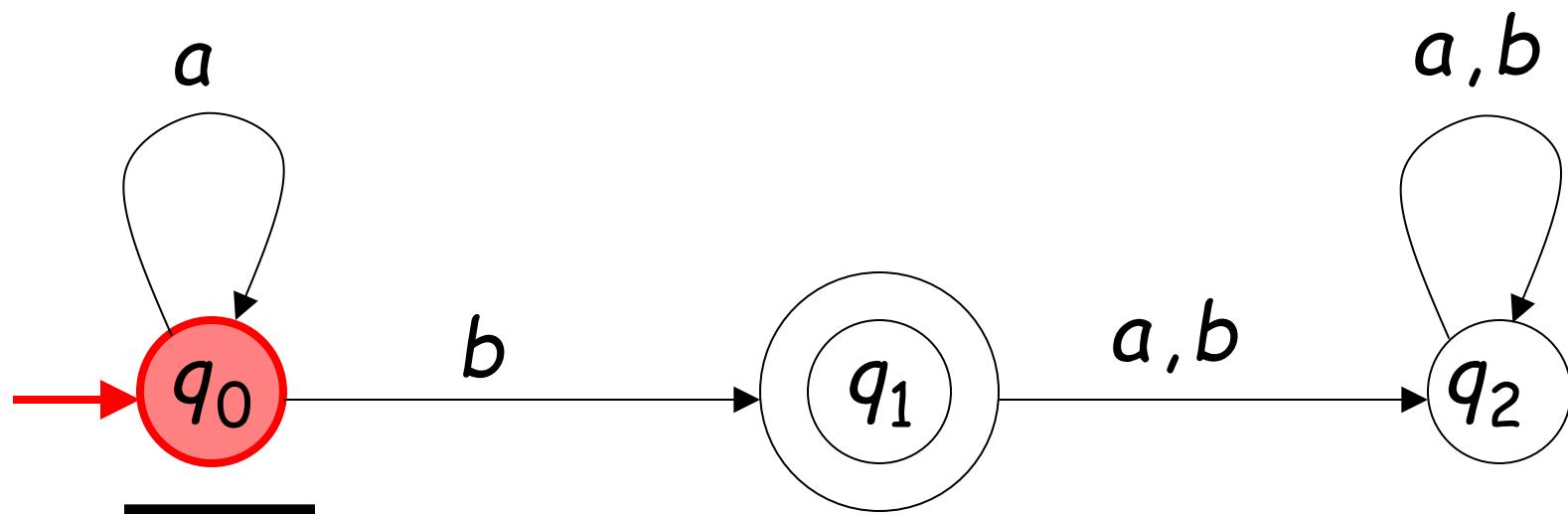
Another Example

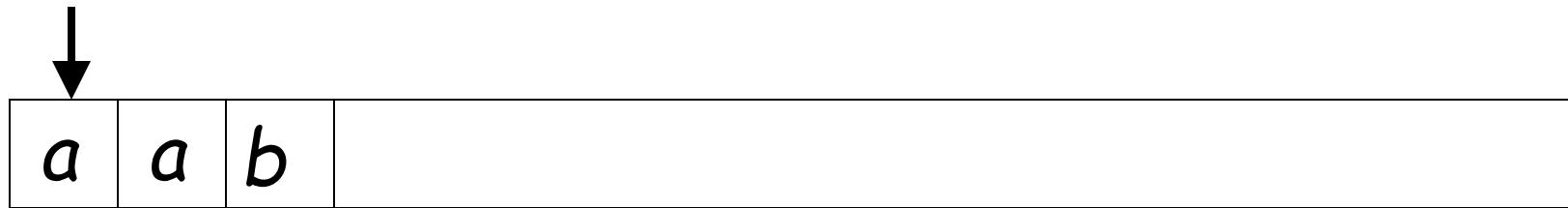


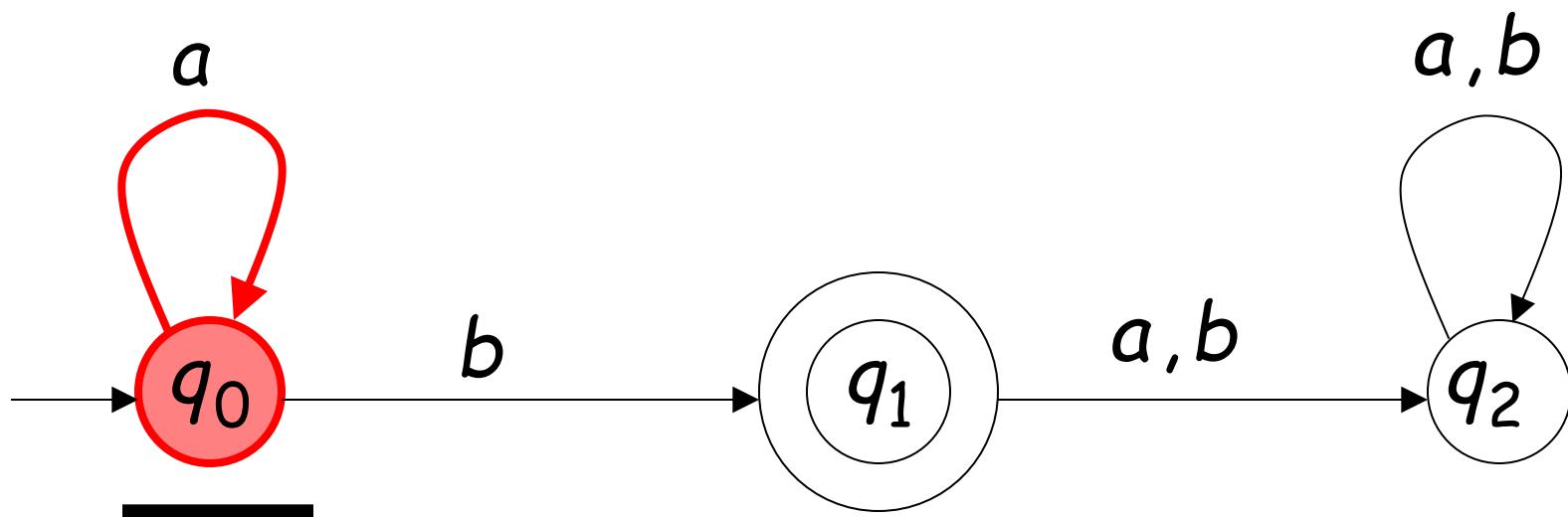
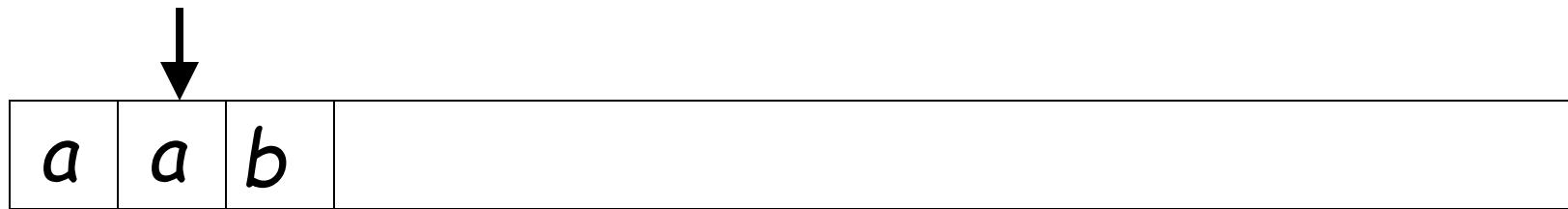


a	a	b	
-----	-----	-----	--

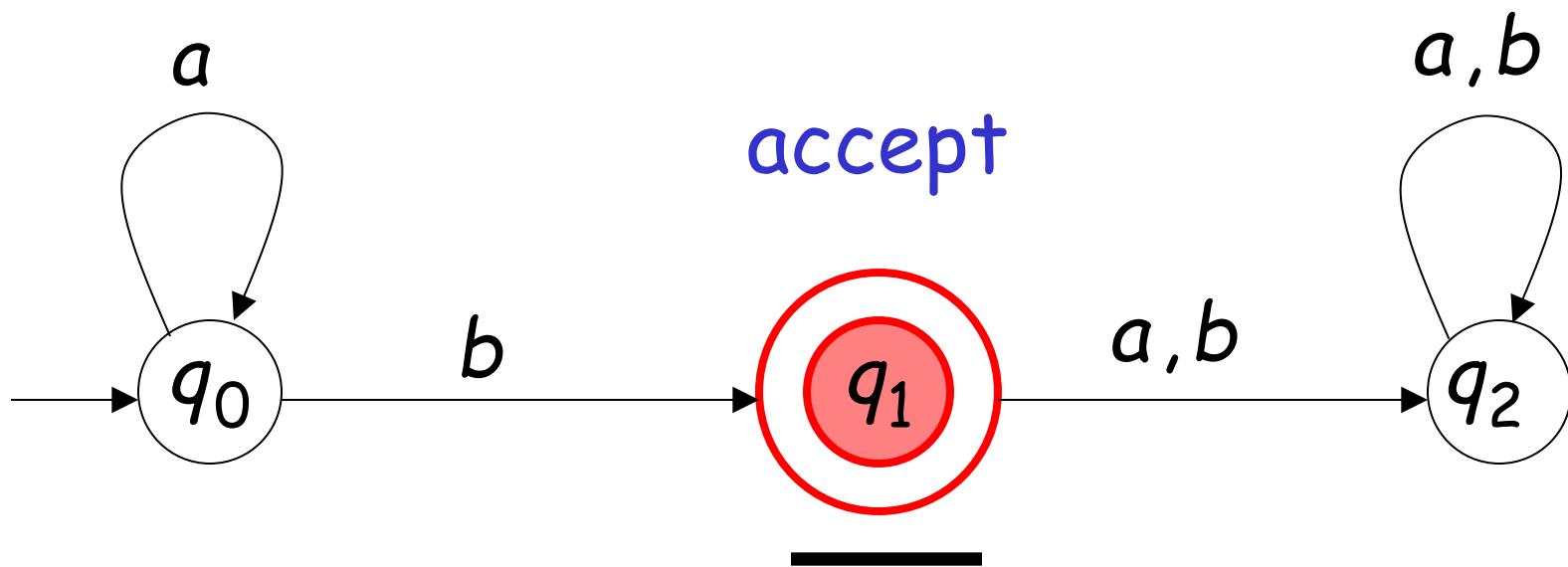
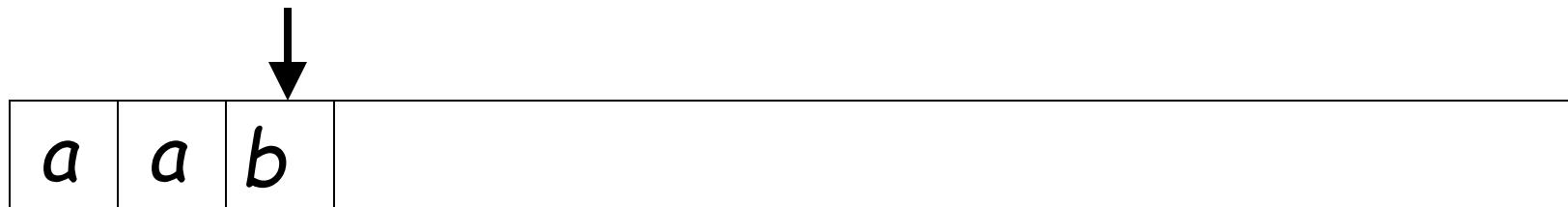
Input String







Input finished

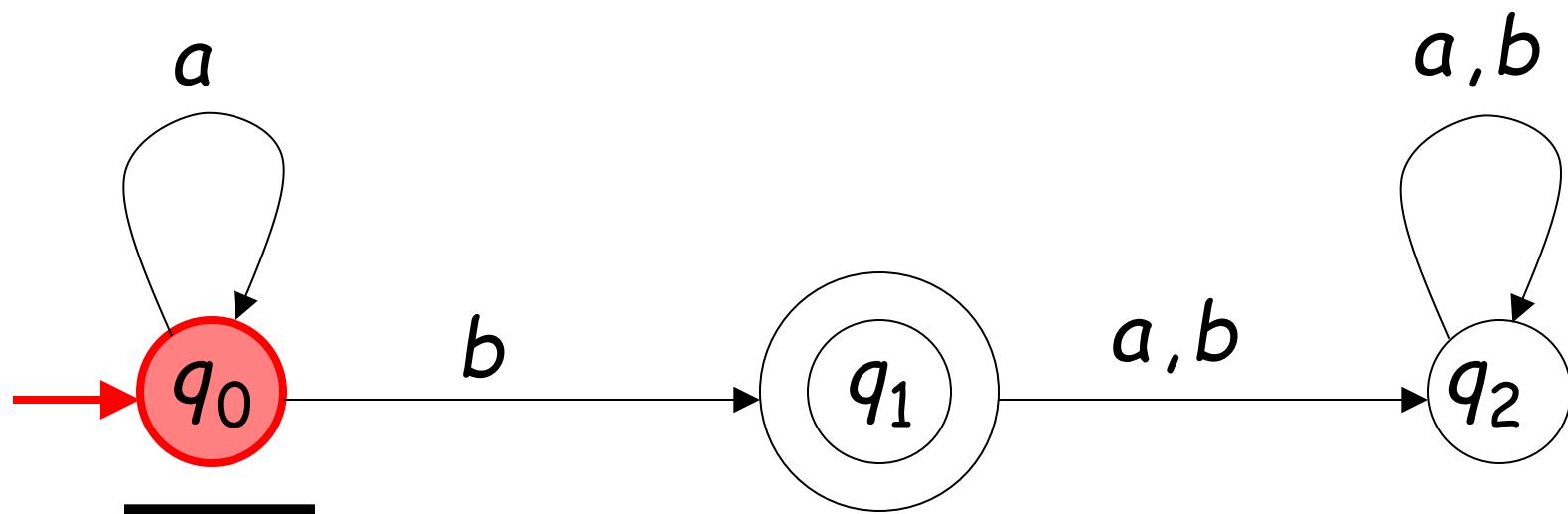


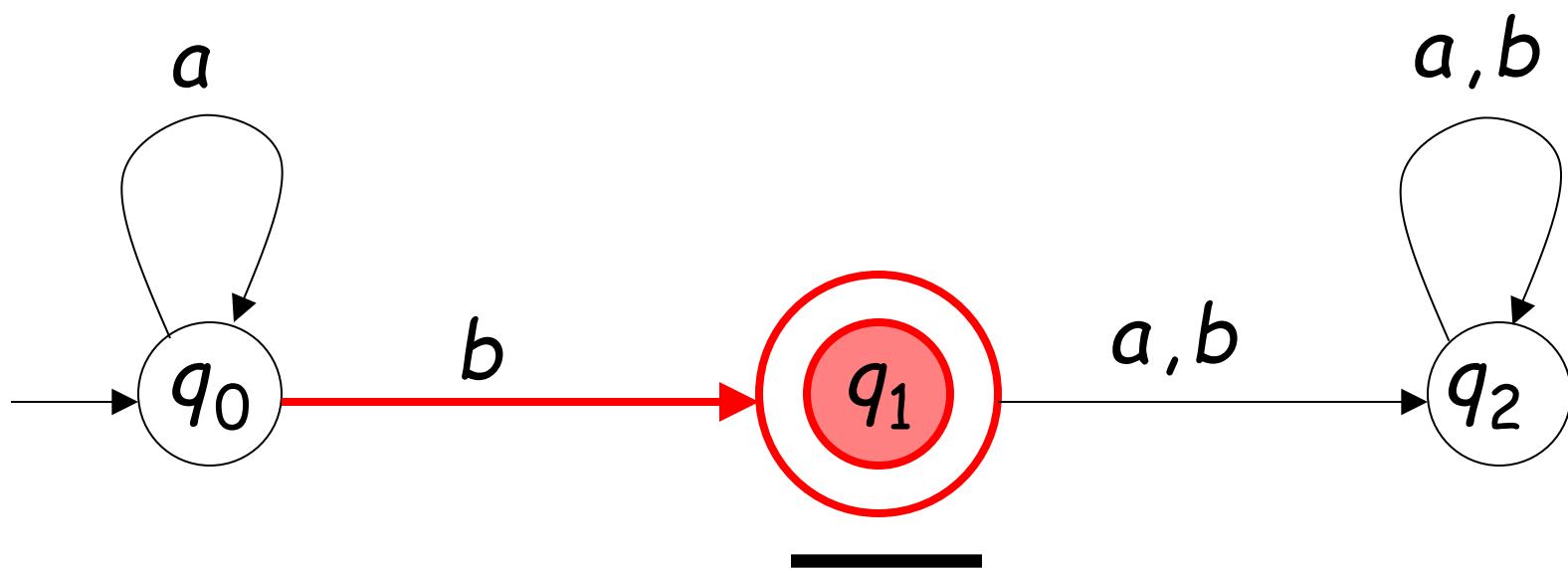
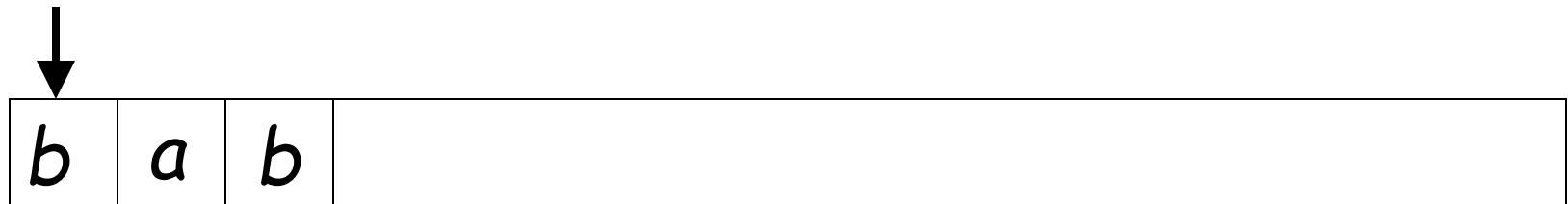
A rejection case

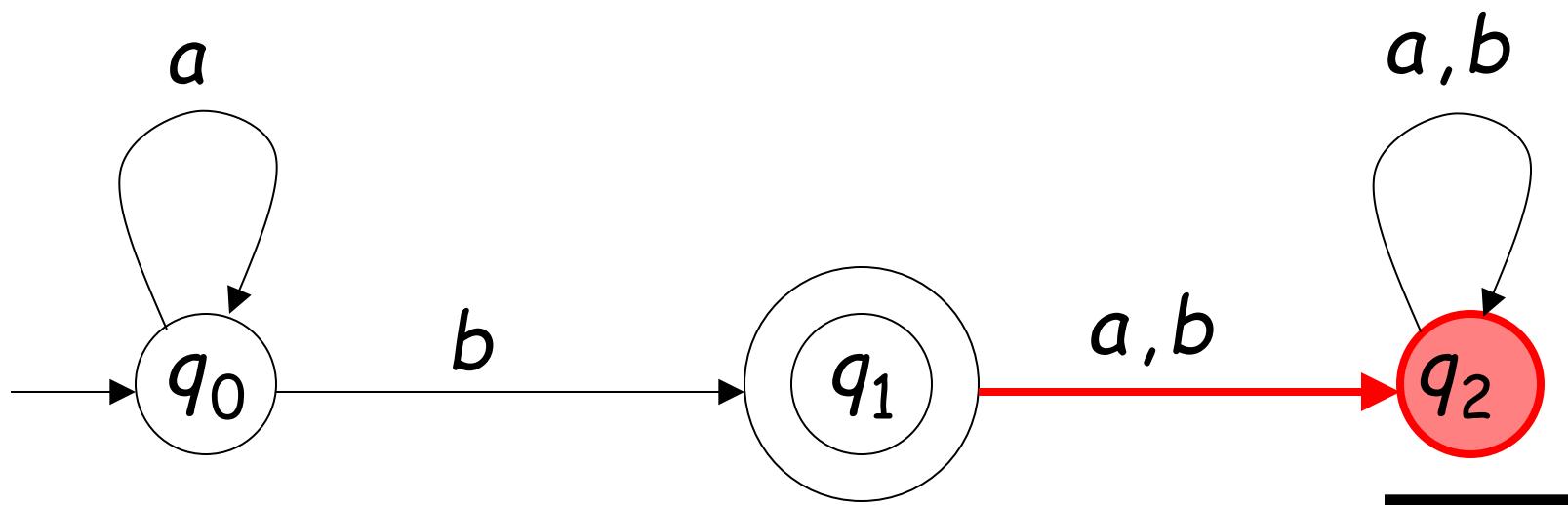
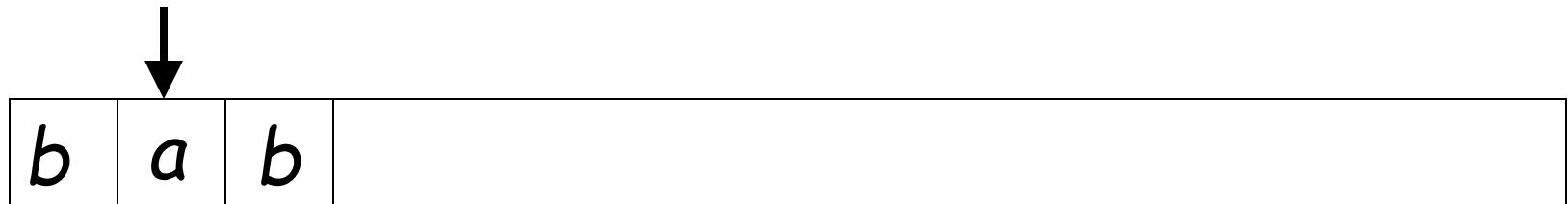


b	a	b	
---	---	---	--

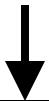
Input String



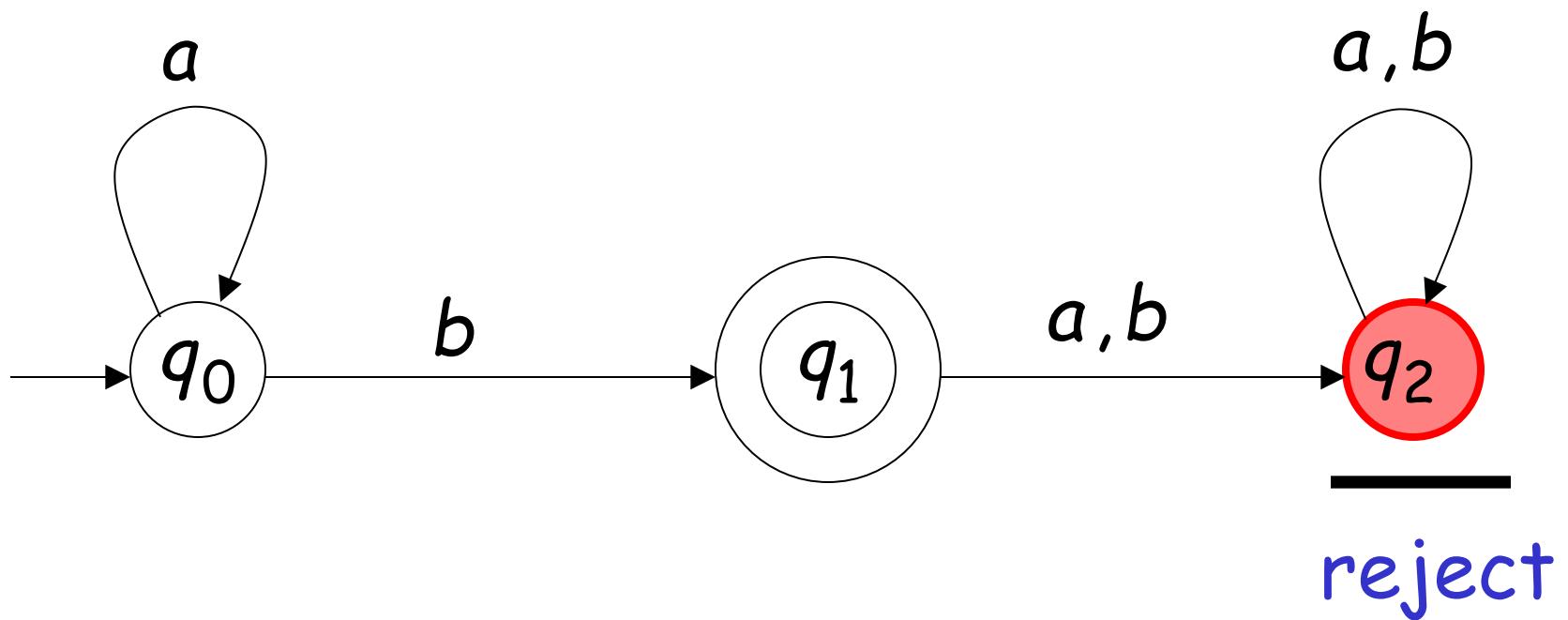




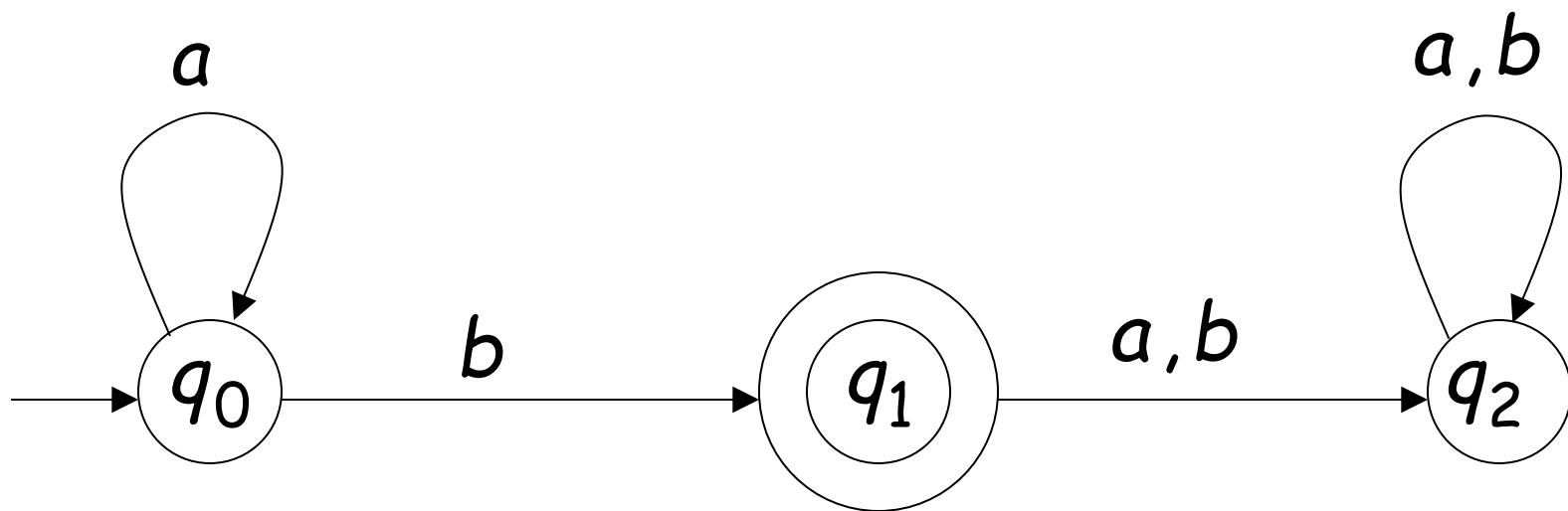
Input finished



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

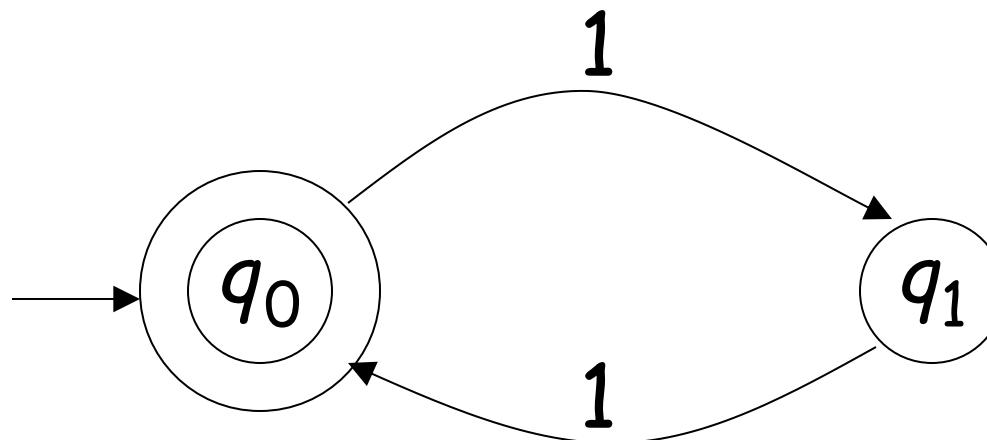


Language Accepted: $L = \{a^n b : n \geq 0\}$



Another Example

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



Language Accepted:

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

Formal Definition

Deterministic Finite Automaton (DFA)

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

Q : set of states

Σ : input alphabet $\lambda \notin \Sigma$

δ : transition function

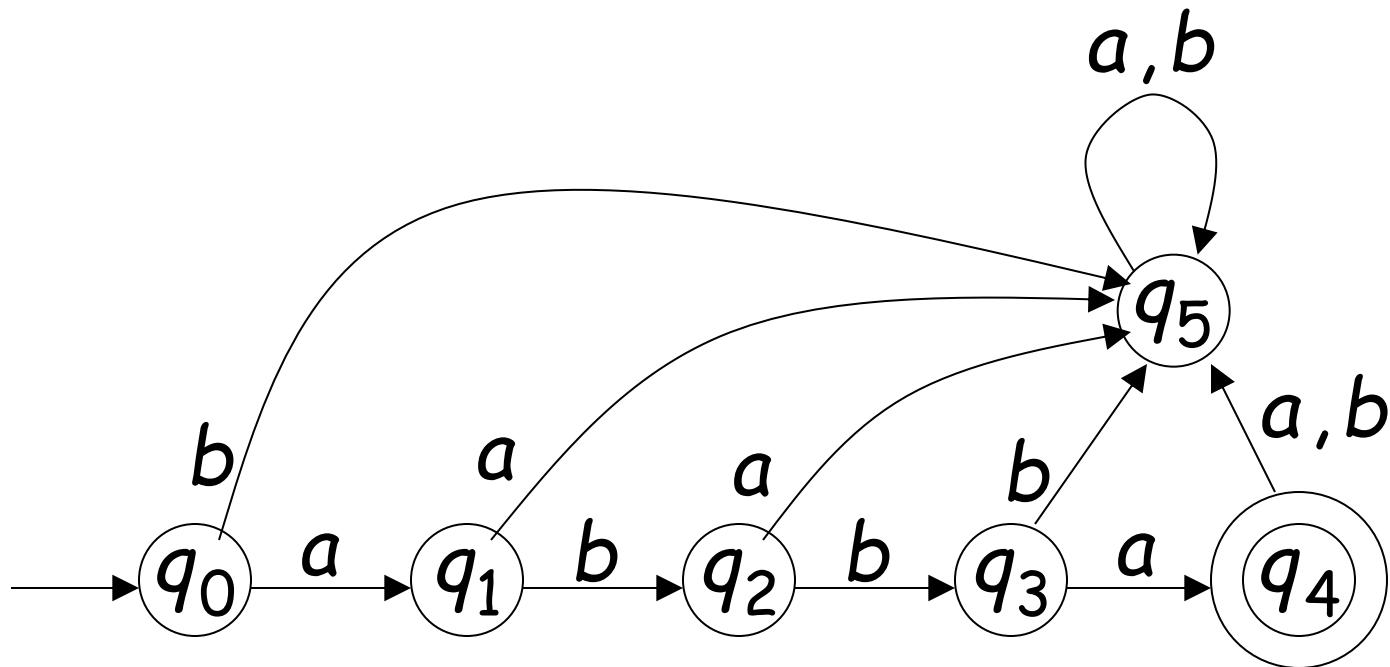
q_0 : initial state

F : set of accepting states

Set of States \mathcal{Q}

Example

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

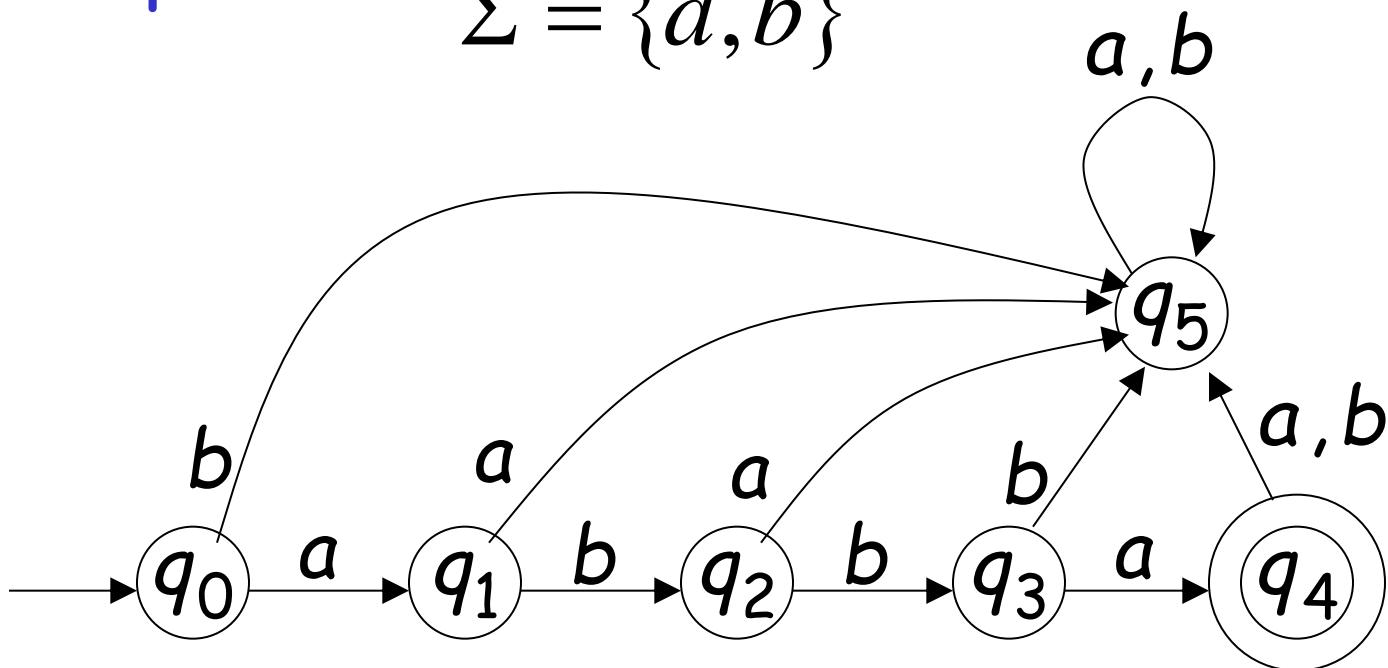


Input Alphabet Σ

$\lambda \notin \Sigma$:the input alphabet never contains λ

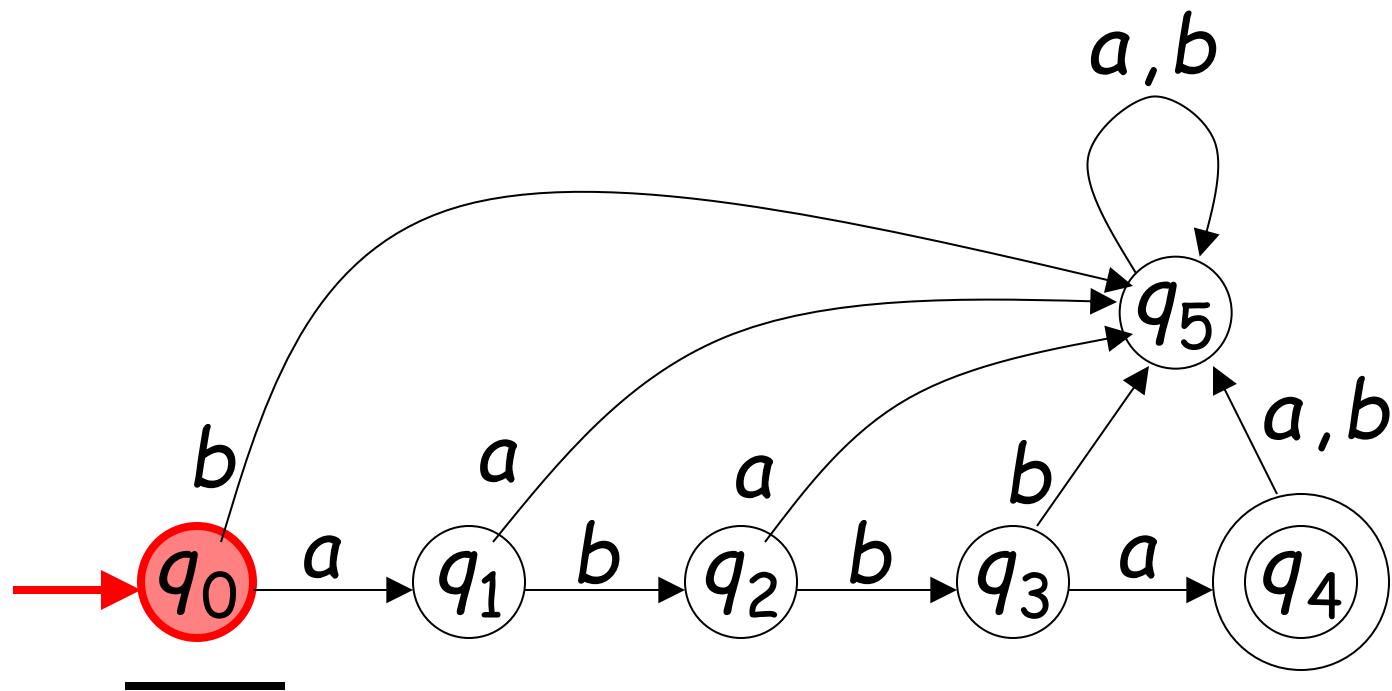
Example

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



Initial State q_0

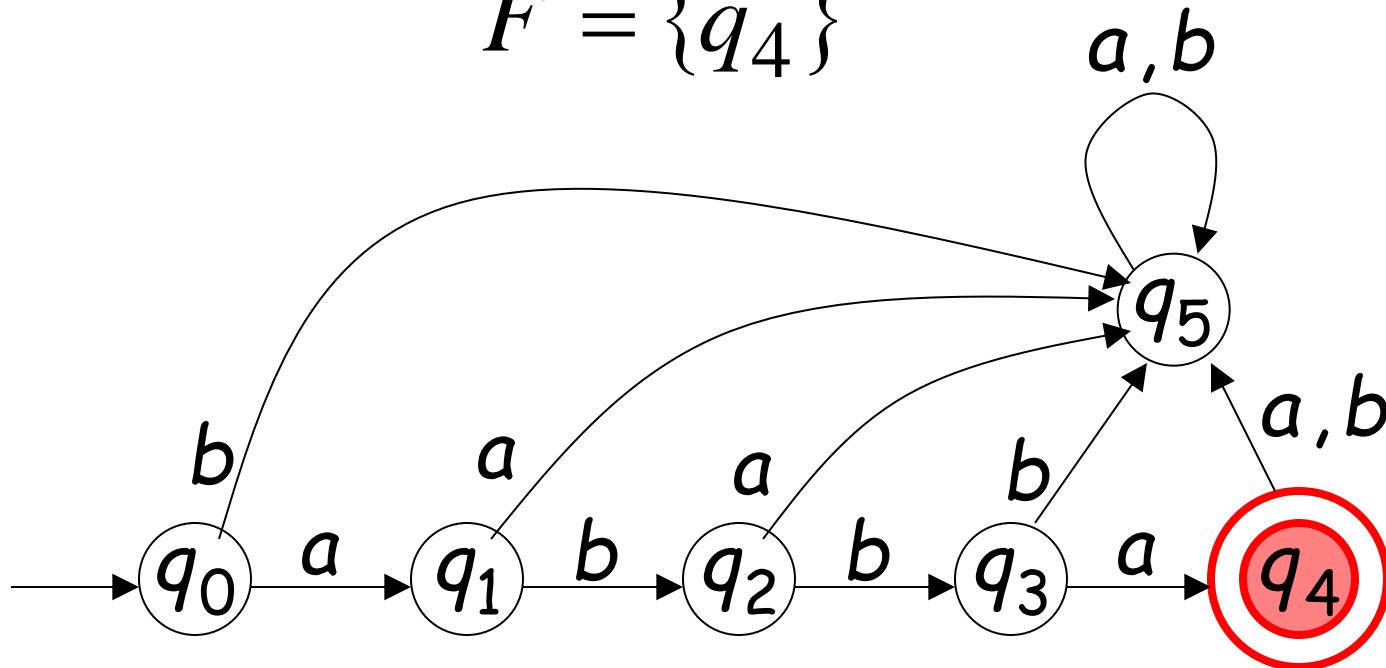
Example



Set of Accepting States $F \subseteq Q$

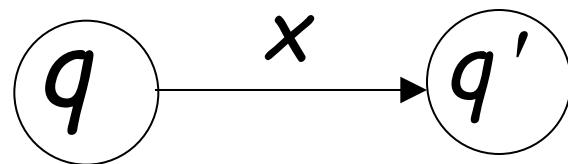
Example

$$F = \{q_4\}$$



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

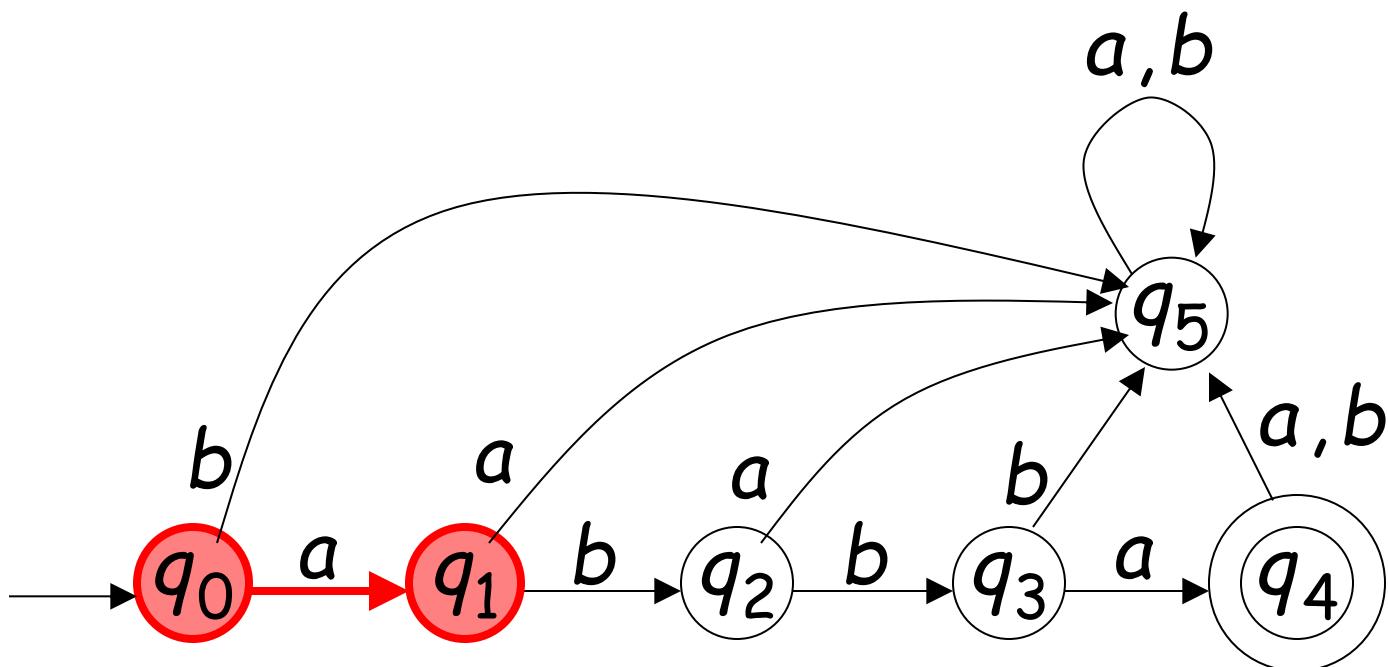
$$\delta(q, x) = q'$$



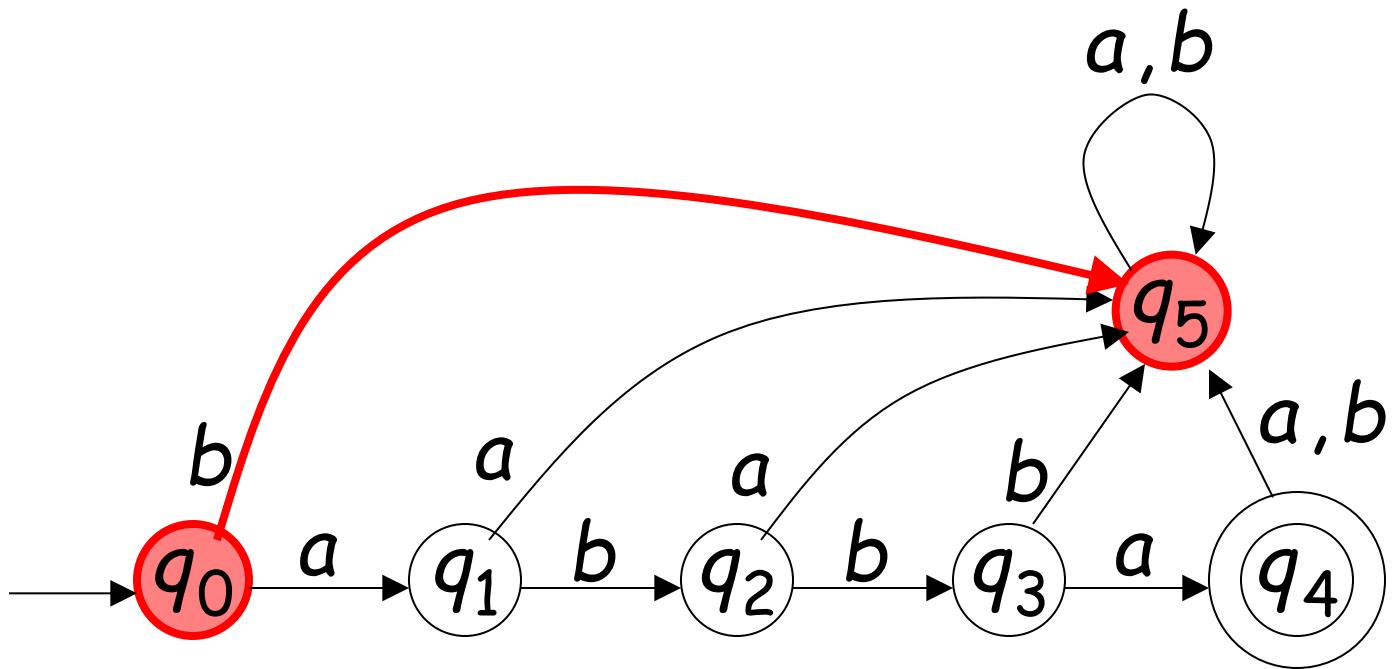
Describes the result of a transition
from state q with symbol x

Example:

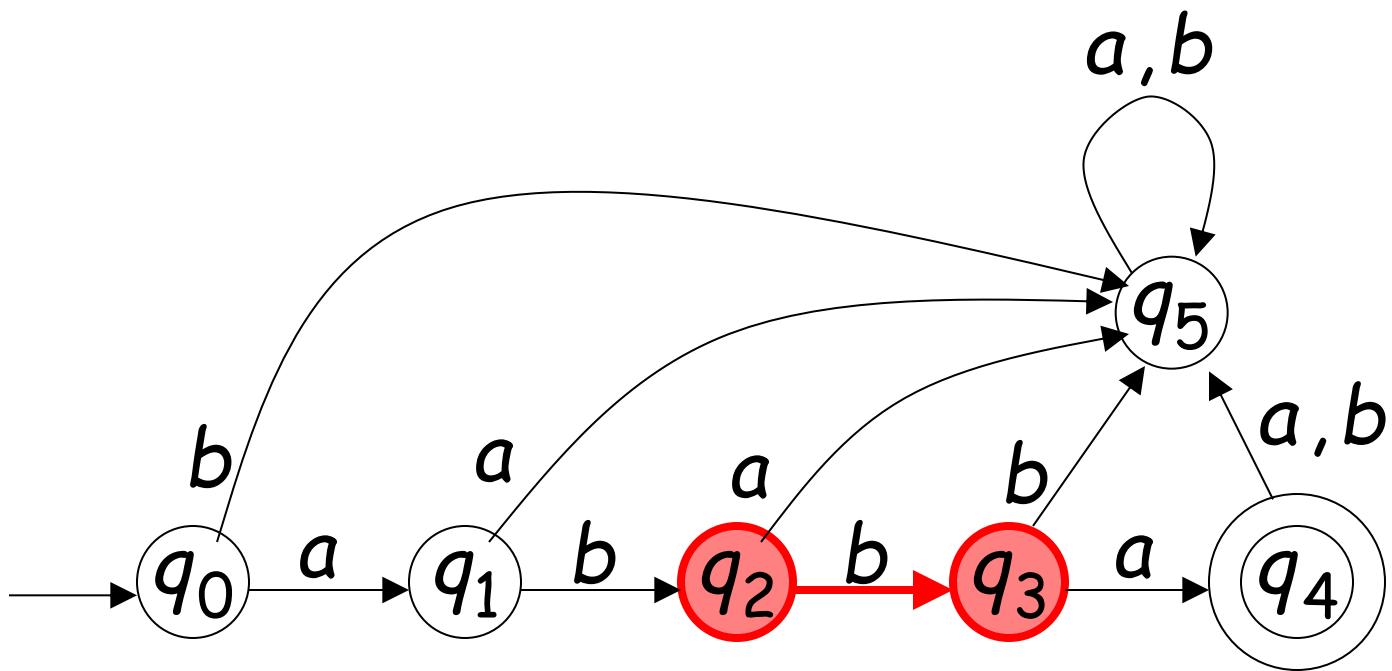
$$\delta(q_0, a) = q_1$$



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



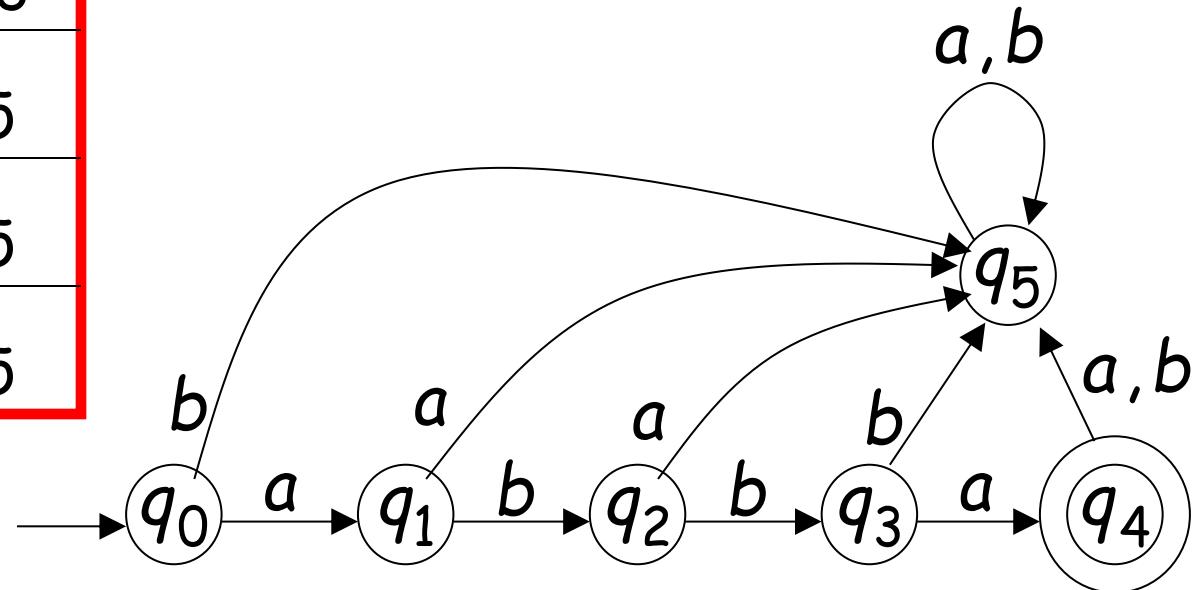
$$\delta(q_2, b) = q_3$$



Transition Table for δ

symbols

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



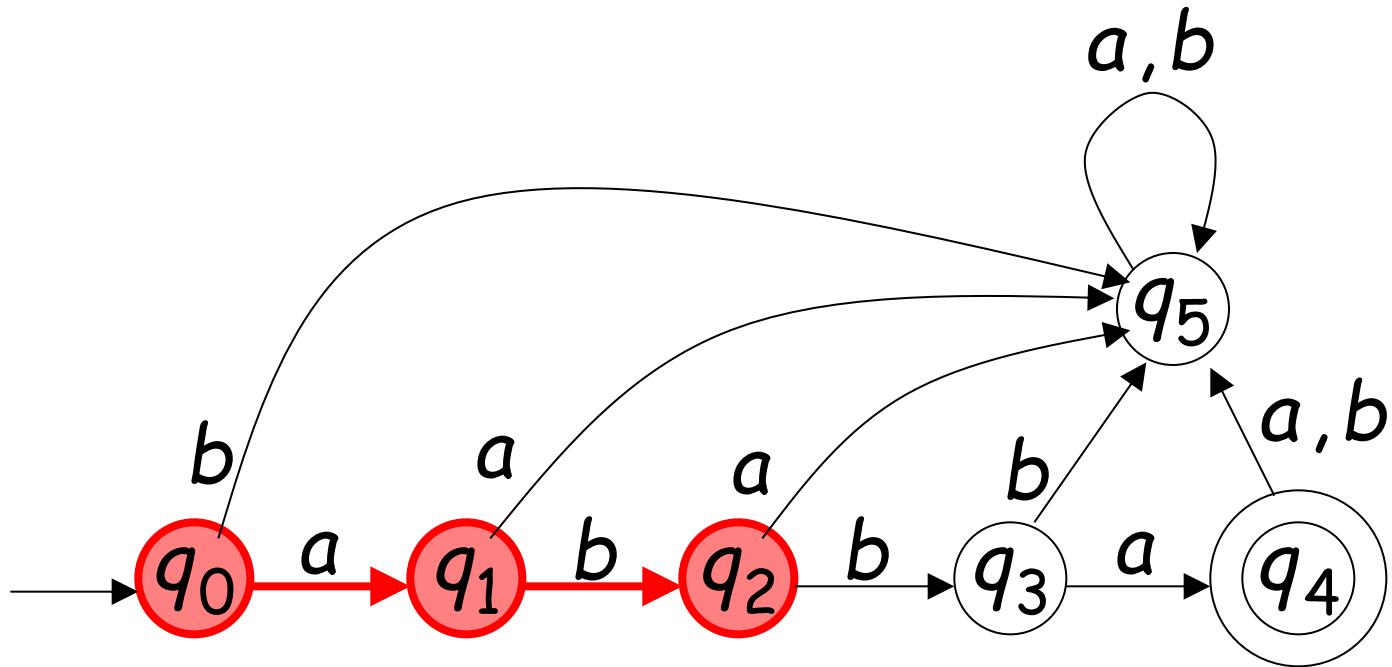
Extended Transition Function

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

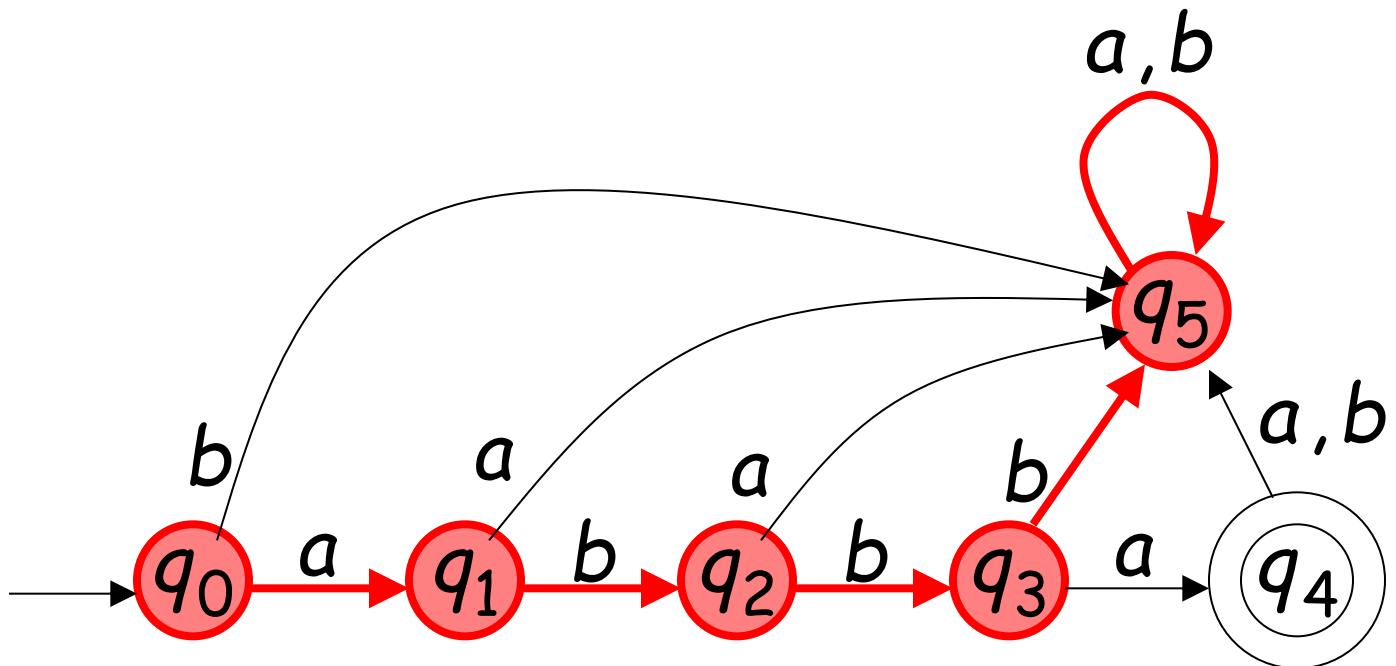
$$\delta^*(q, w) = q'$$

Describes the resulting state
after scanning string w from state q

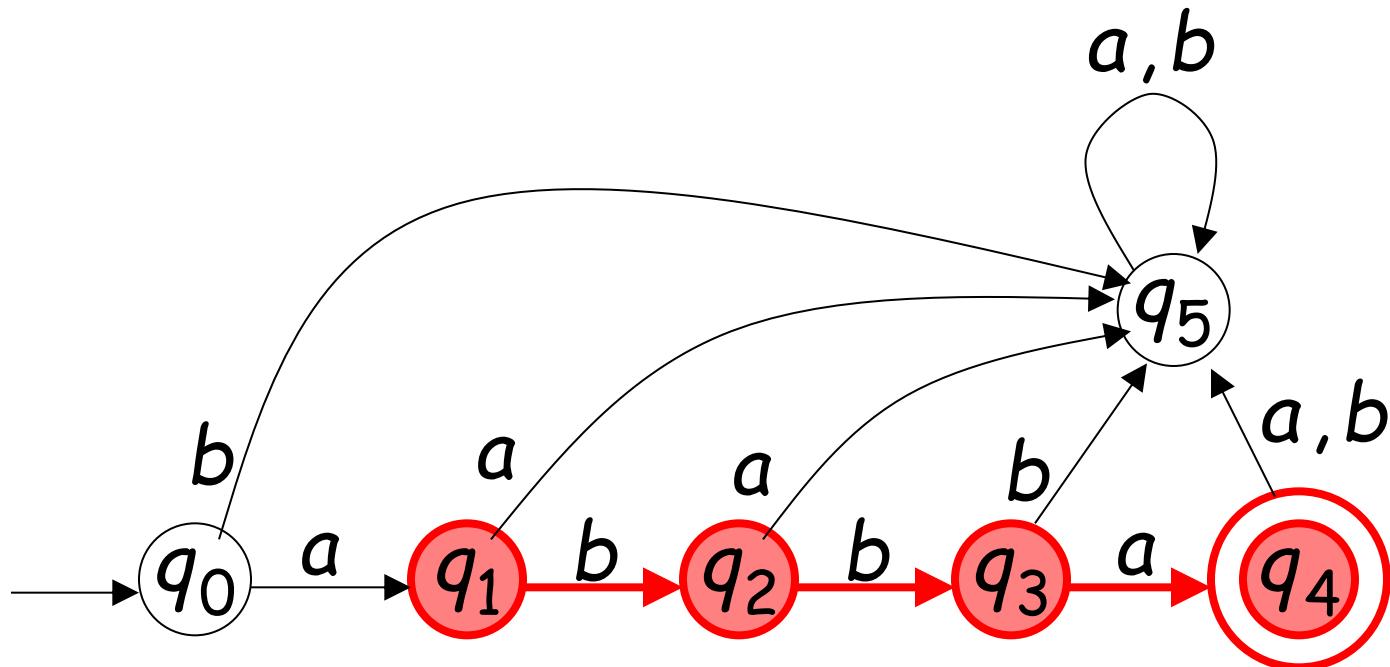
Example: $\delta^*(q_0, ab) = q_2$



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



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



Special case:

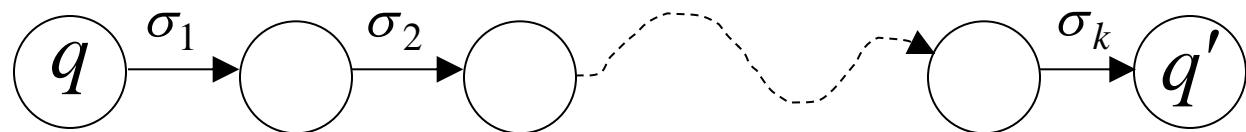
for any state q

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

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

implies that there is a walk of transitions

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



states may be repeated



Language Accepted by DFA

Language of DFA M :

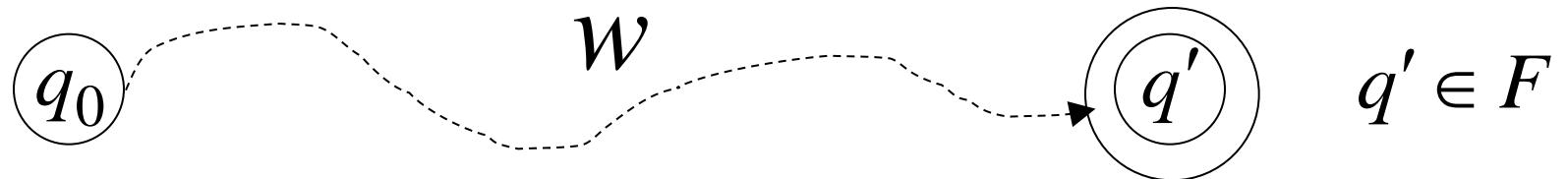
it is denoted as $L(M)$ and contains all the strings accepted by M

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

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

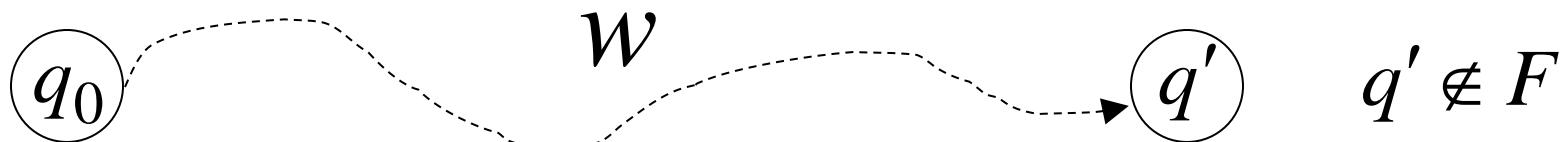
Language accepted by M :

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



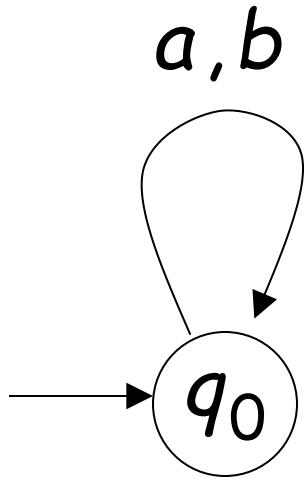
Language rejected by M :

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



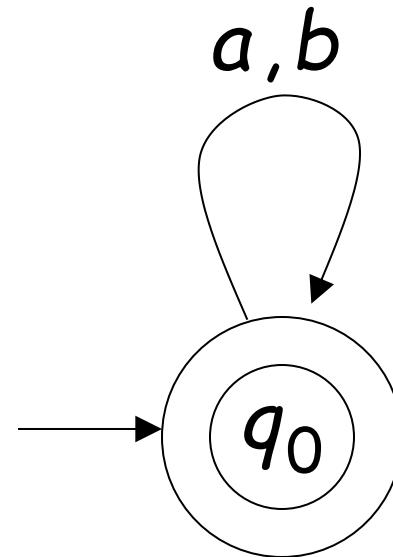
More DFA Examples

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



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

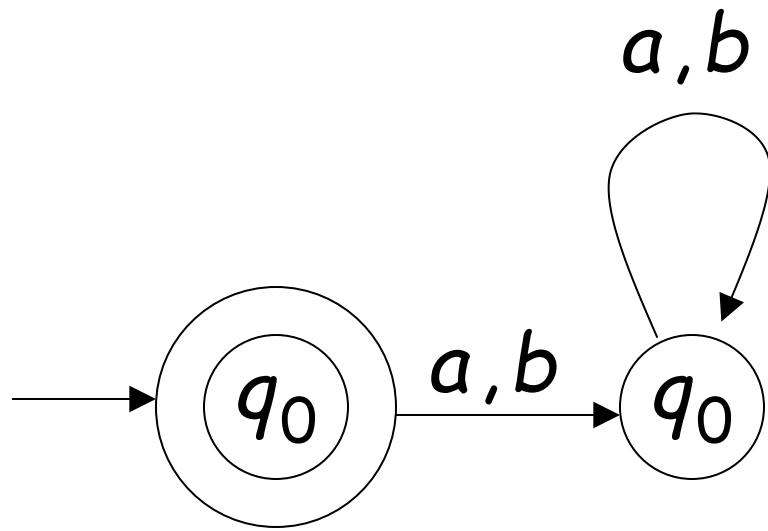
Empty language



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

All strings

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

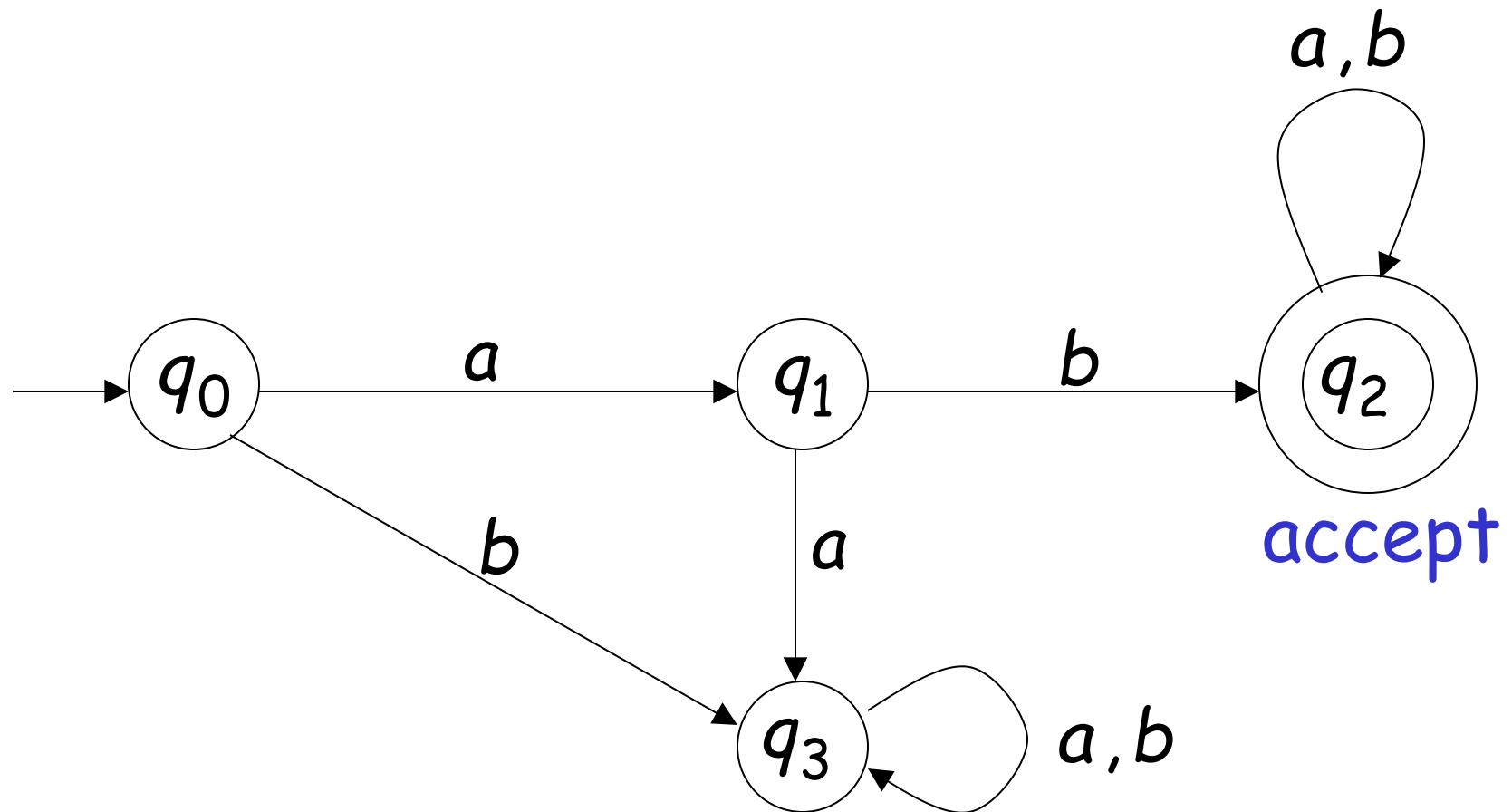


$$L(M) = \{\lambda\}$$

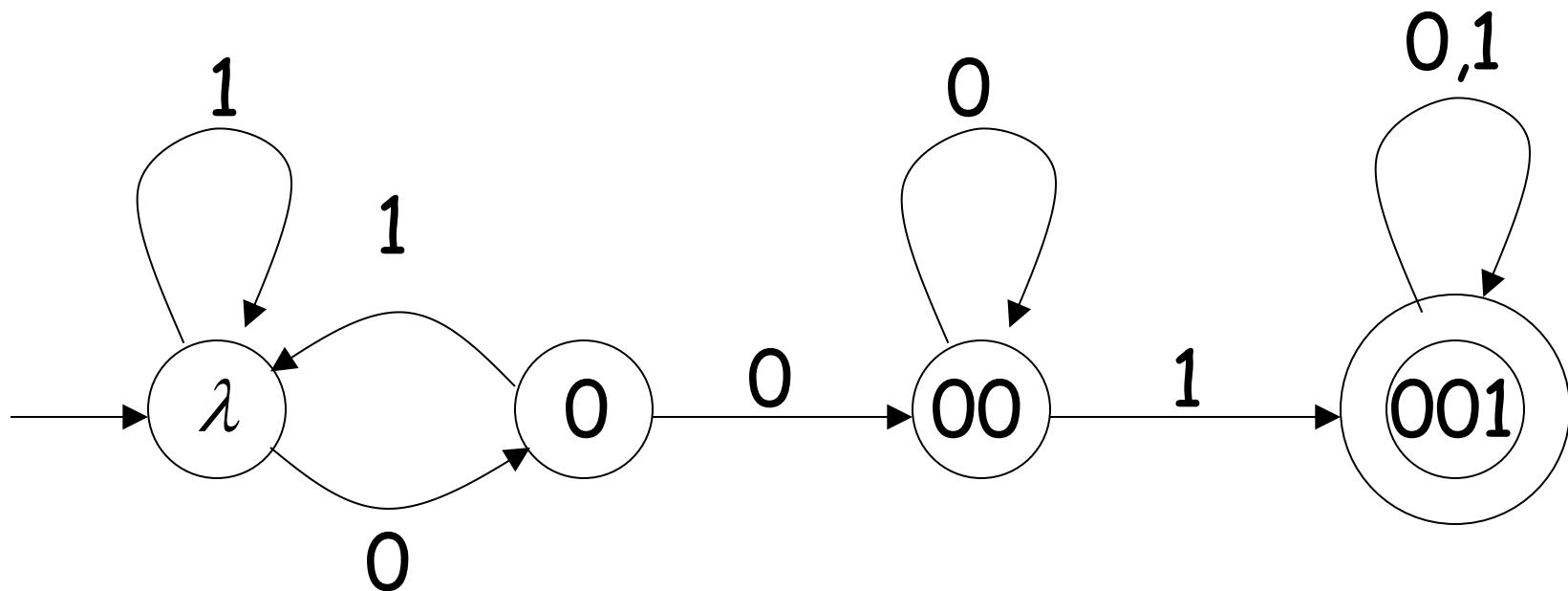
Language of the empty string

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

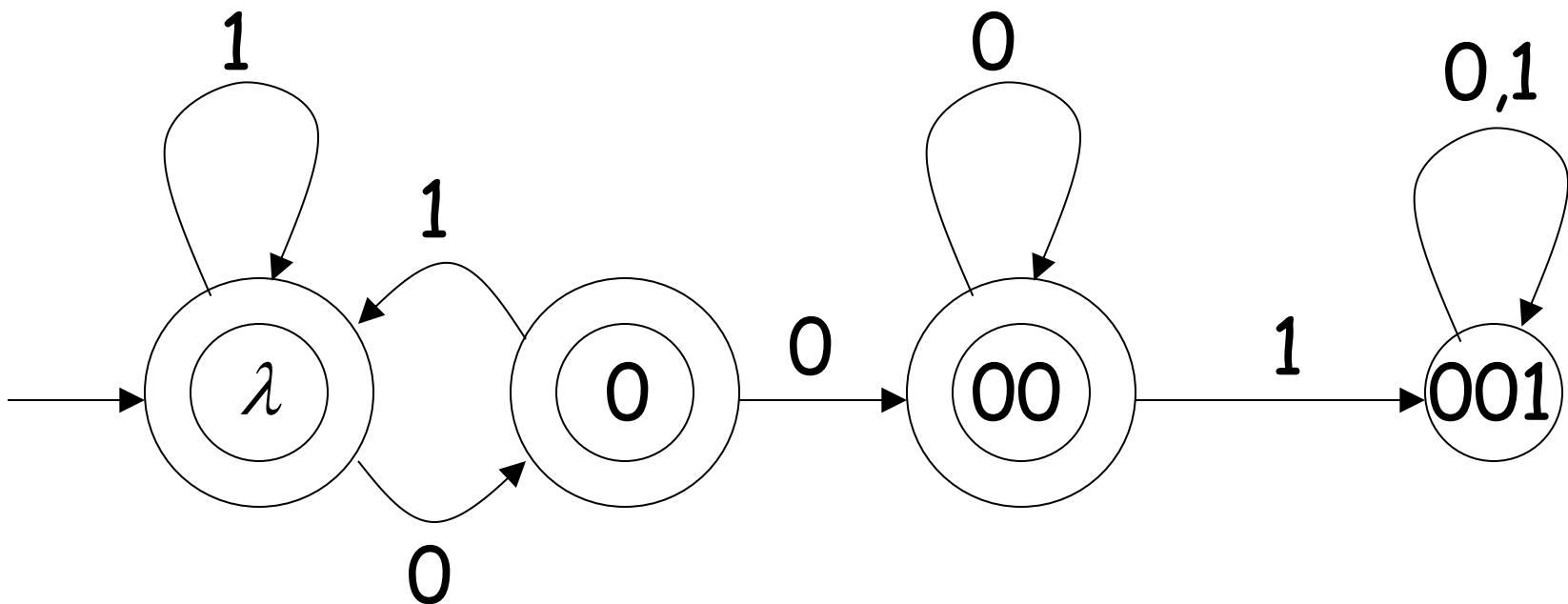
$L(M) = \{ \text{all strings with prefix } ab \}$



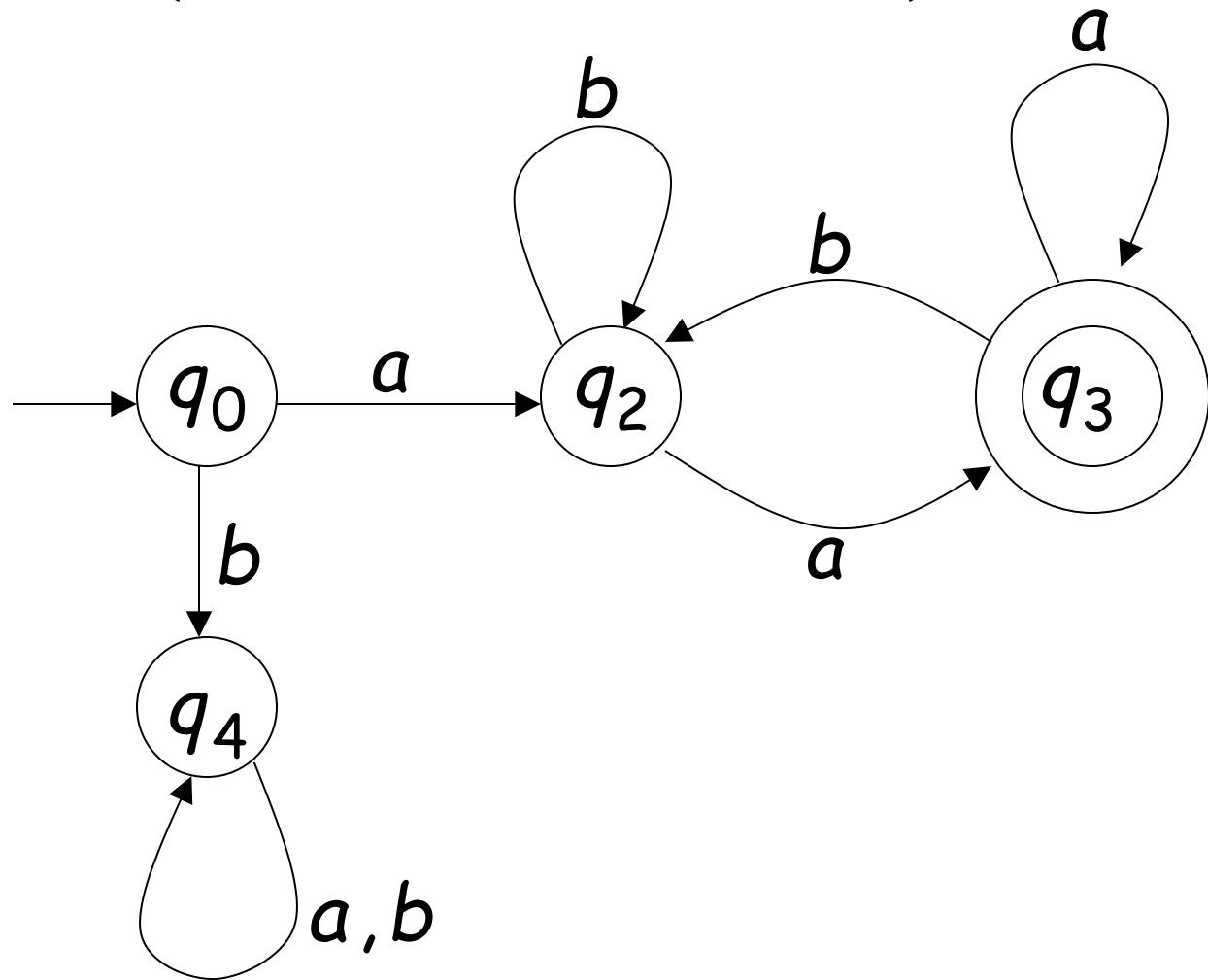
$L(M) = \{ \text{ all binary strings containing substring } 001 \ }$



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



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



Regular Languages

Definition:

A language L is **regular** if there is a DFA M that accepts it ($L(M) = L$)

The languages accepted by all DFAs form the family of **regular languages**

Example regular languages:

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

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

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

{ all binary strings without substring 001 }

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

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

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

There exist languages which are not Regular:

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

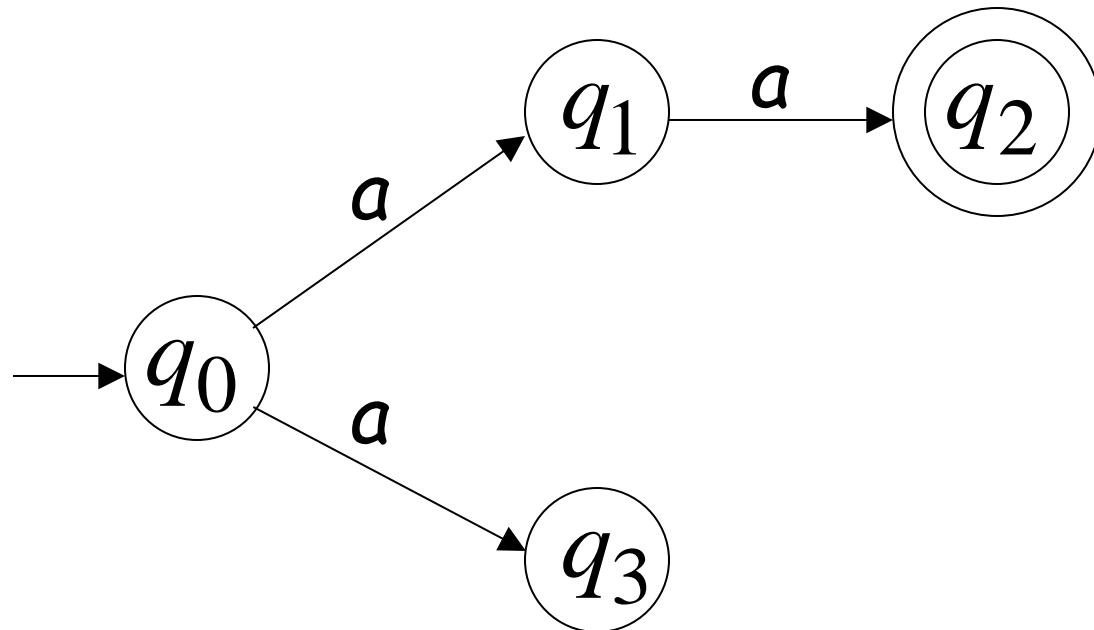
$$\text{ADDITION} = \{x + y = z : x = 1^n, y = 1^m, z = 1^k, \\ n + m = k\}$$

There is no DFA that accepts these languages
(we will prove this in a later class)

Non-Deterministic Finite Automata

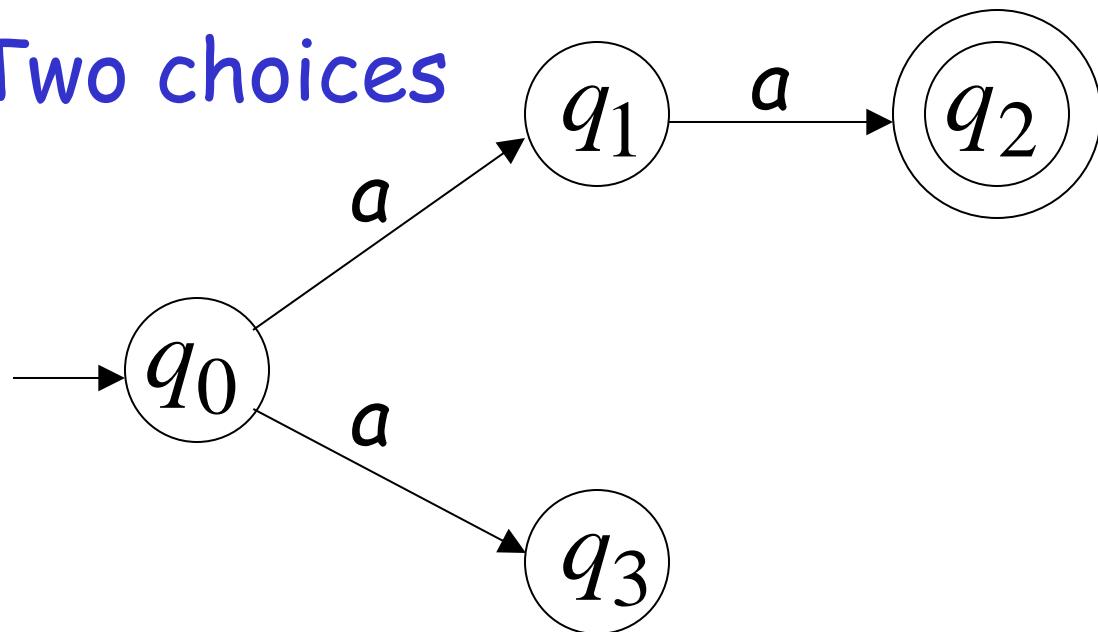
Nondeterministic Finite Automaton (NFA)

Alphabet = $\{a\}$

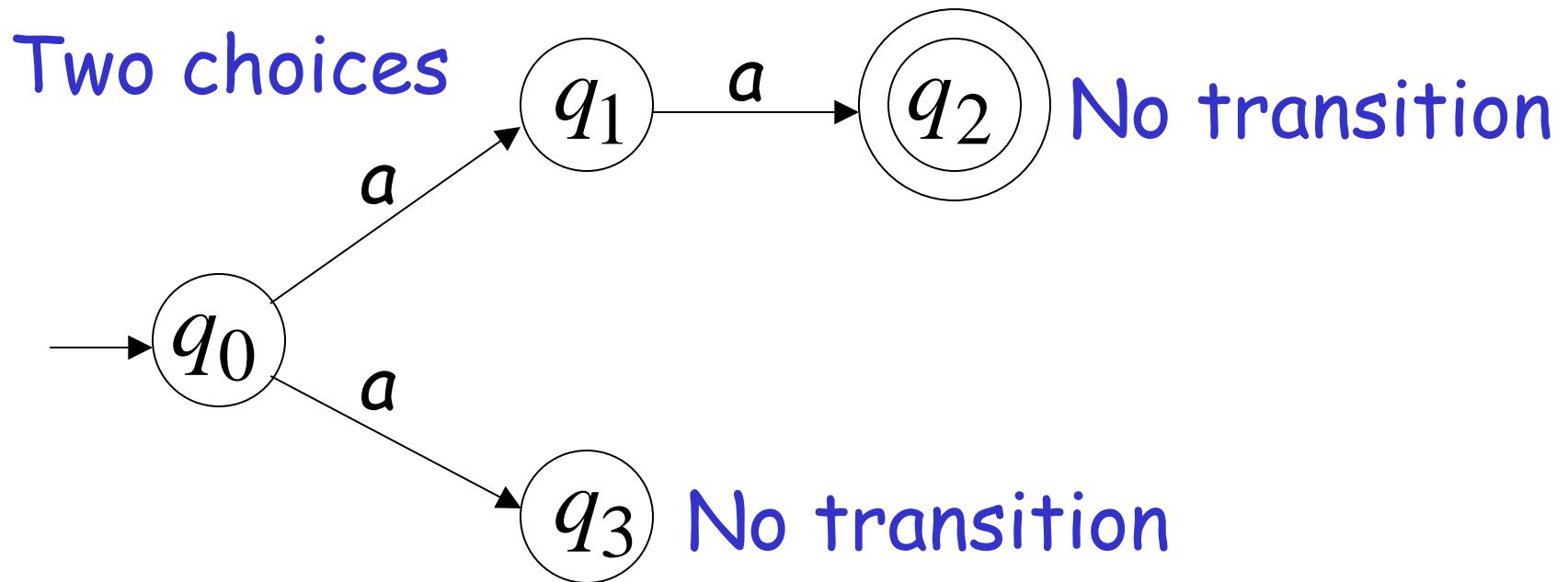


Alphabet = $\{a\}$

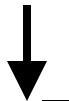
Two choices



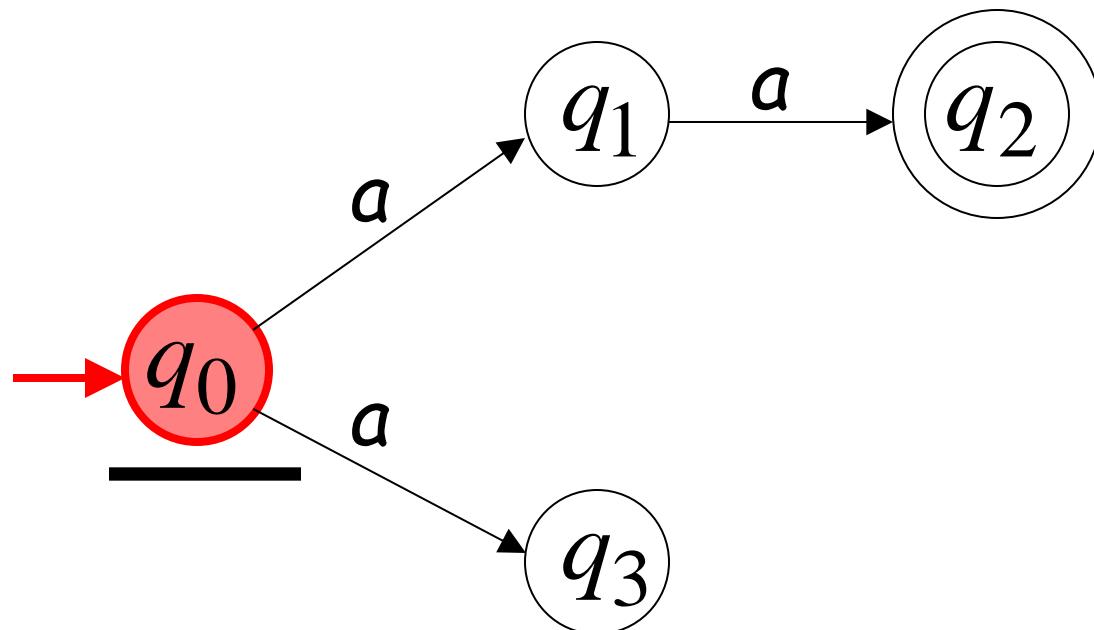
Alphabet = $\{a\}$



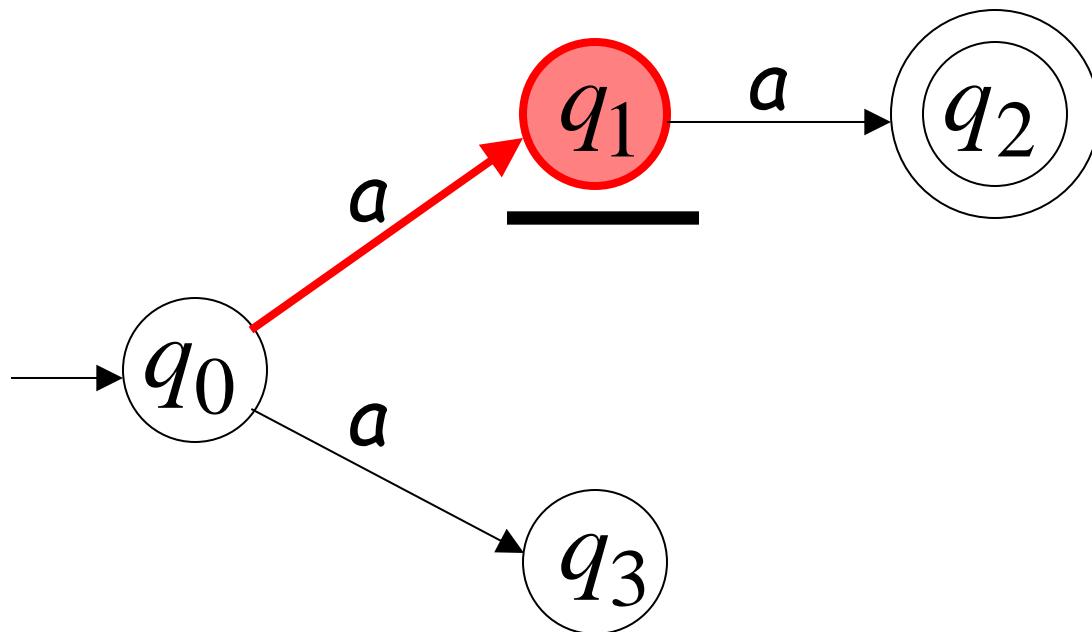
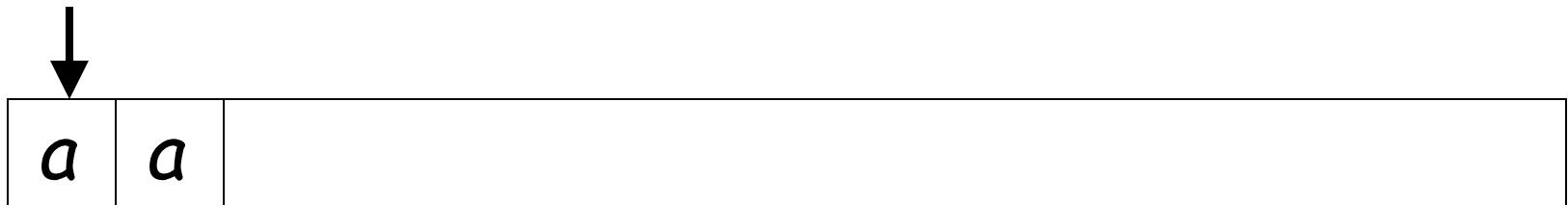
First Choice



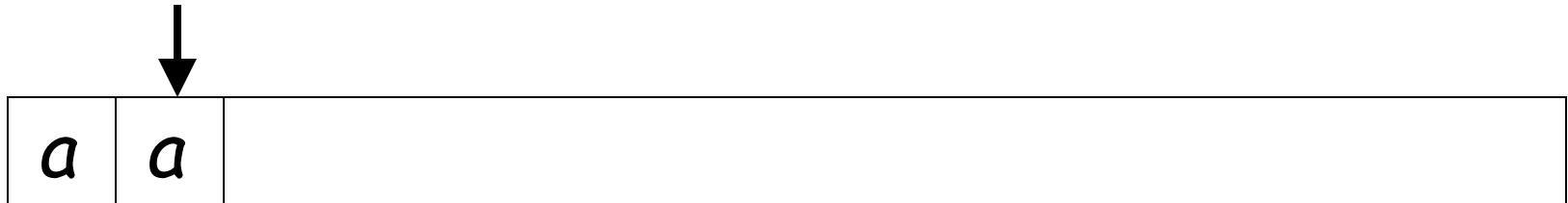
a	a	
-----	-----	--



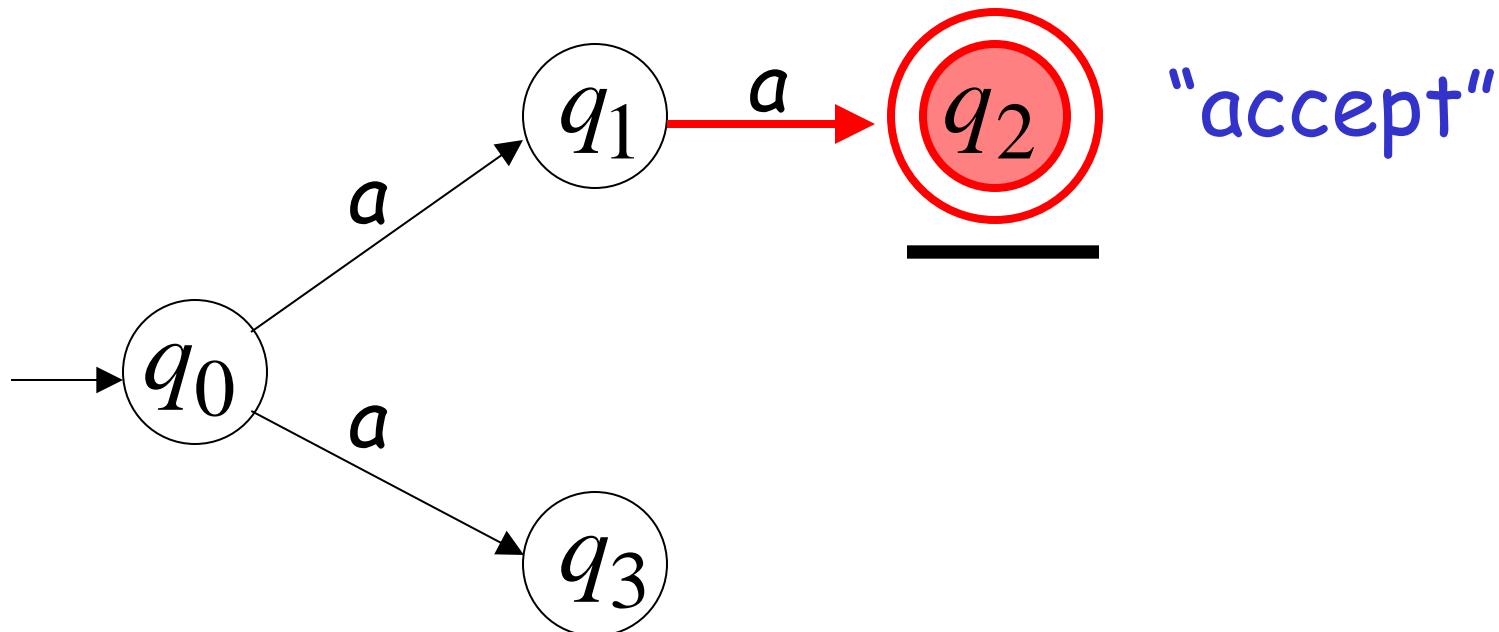
First Choice



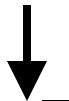
First Choice



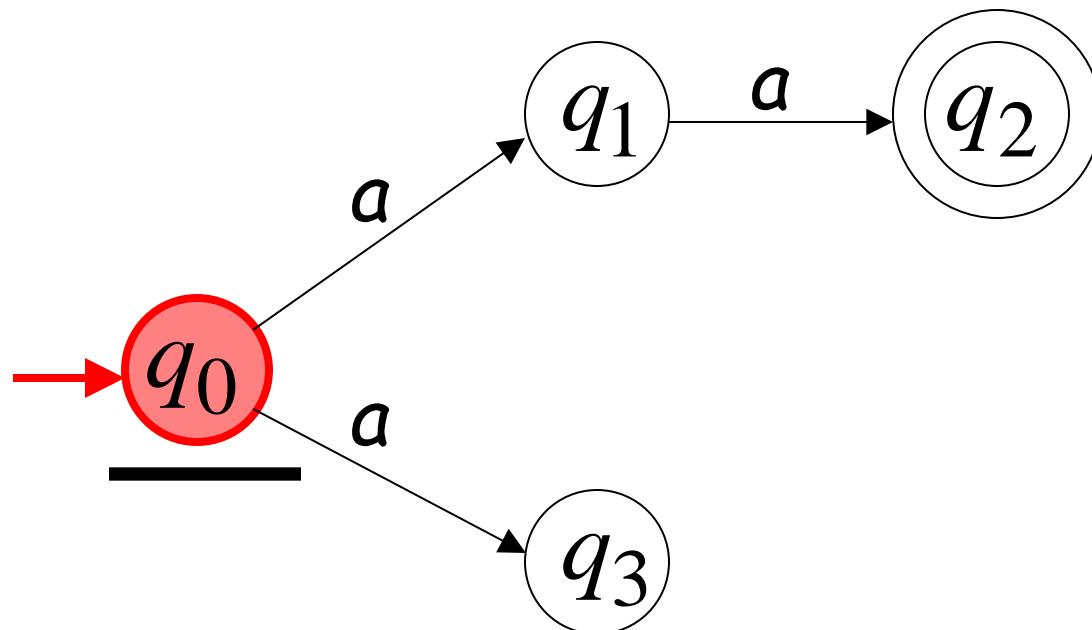
All input is consumed



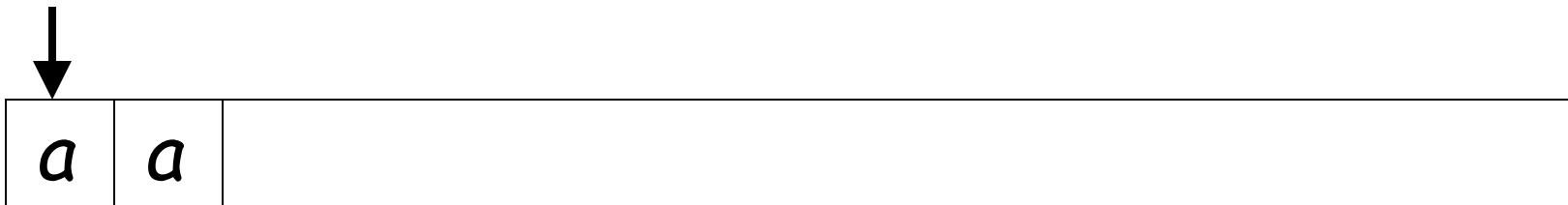
Second Choice



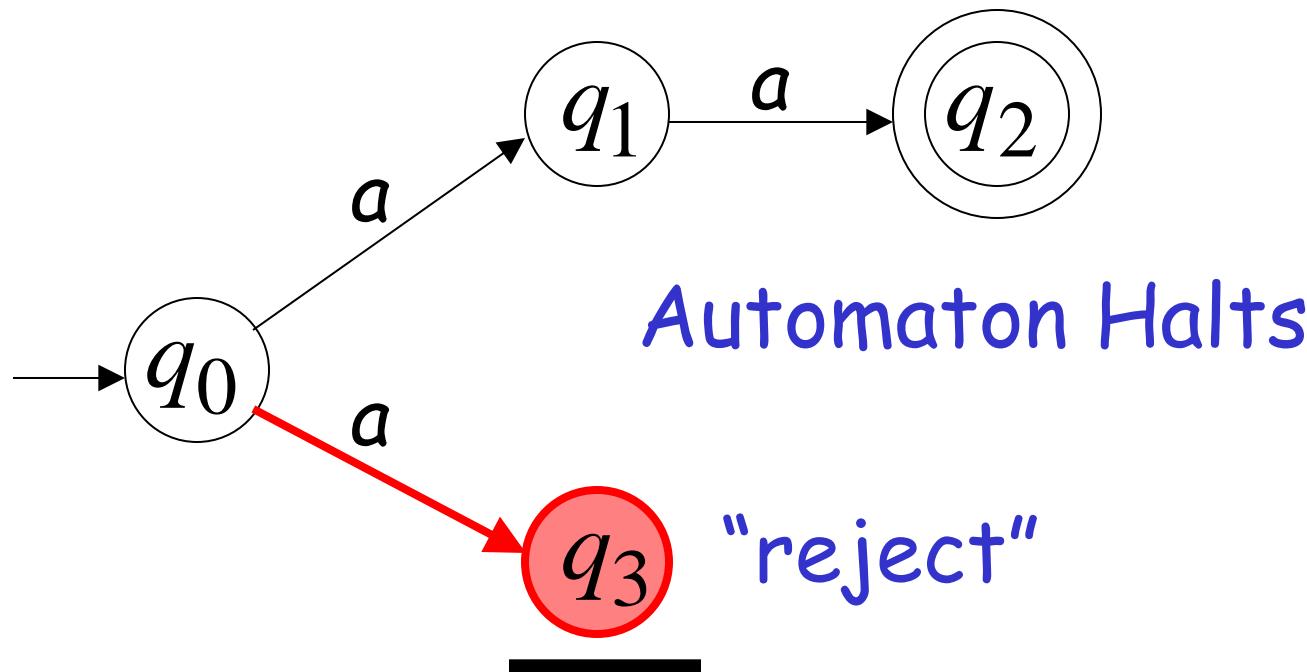
a	a	
-----	-----	--



Second Choice



Input cannot be consumed

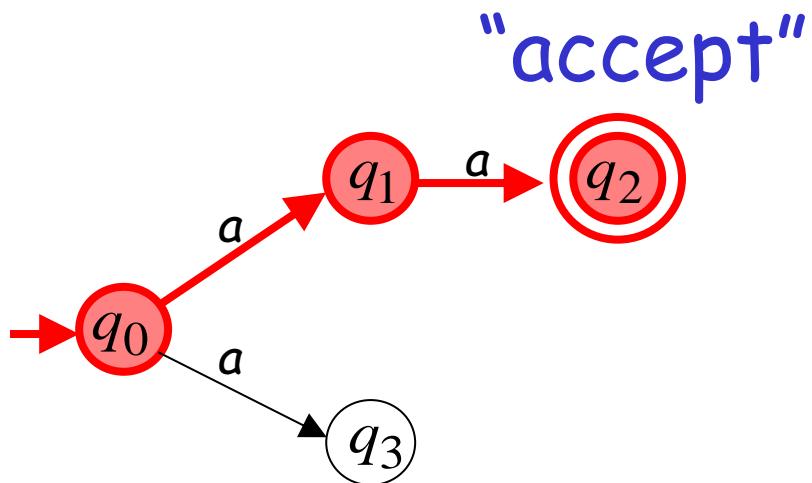


An NFA accepts a string:

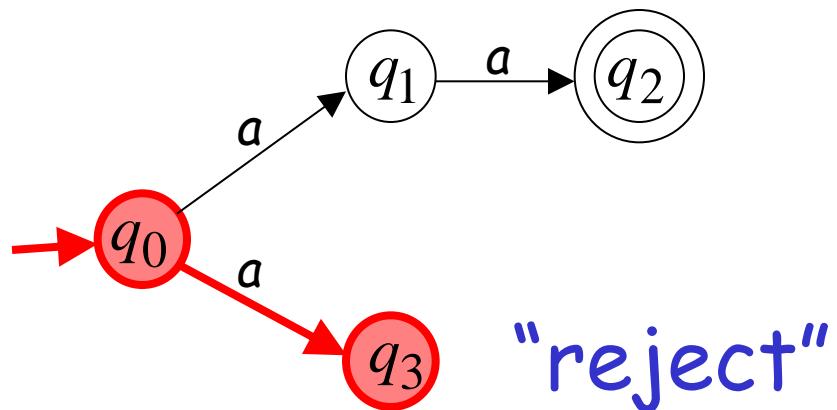
if there is a computation of the NFA
that accepts the string

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

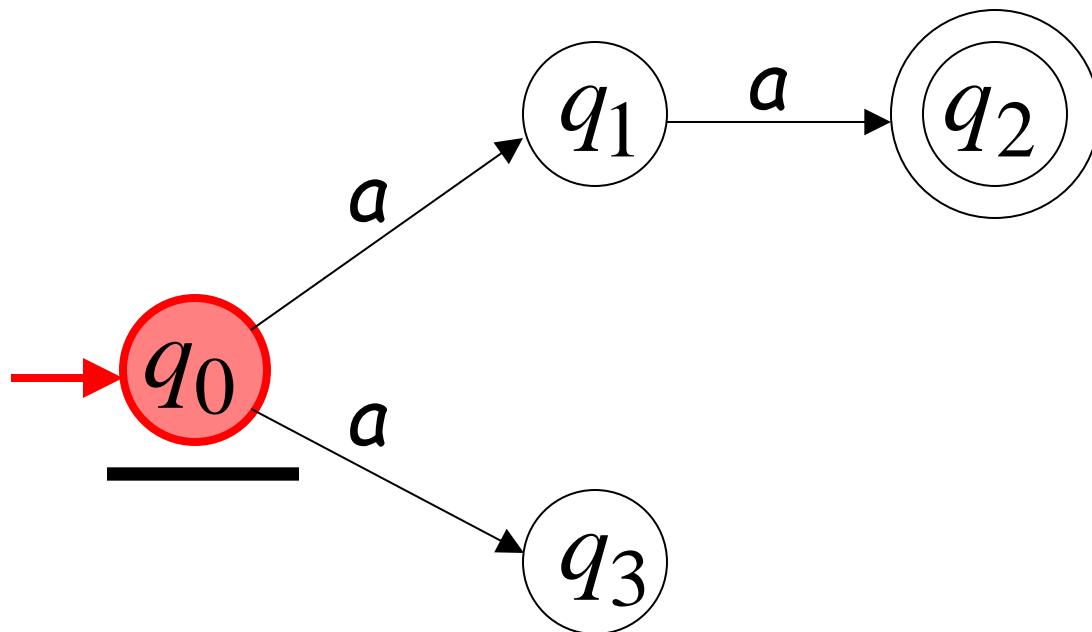


this computation
is ignored

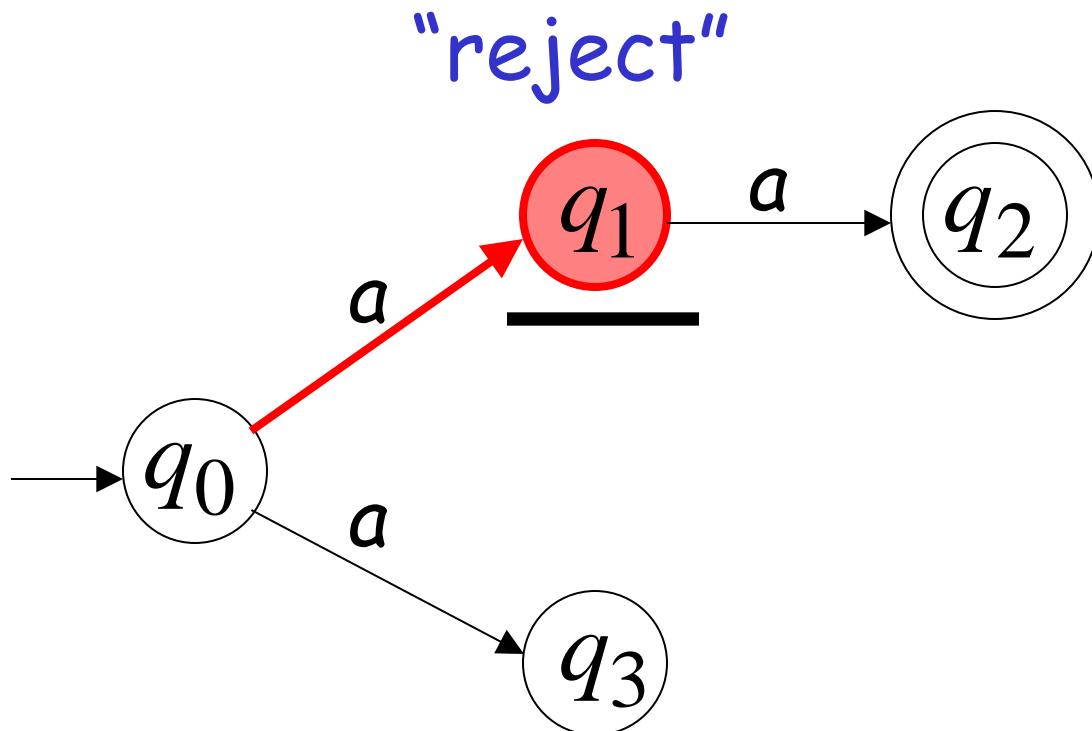
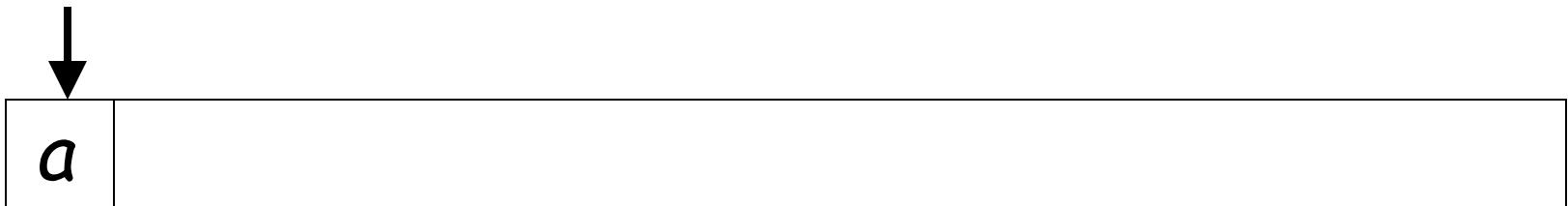
Rejection example



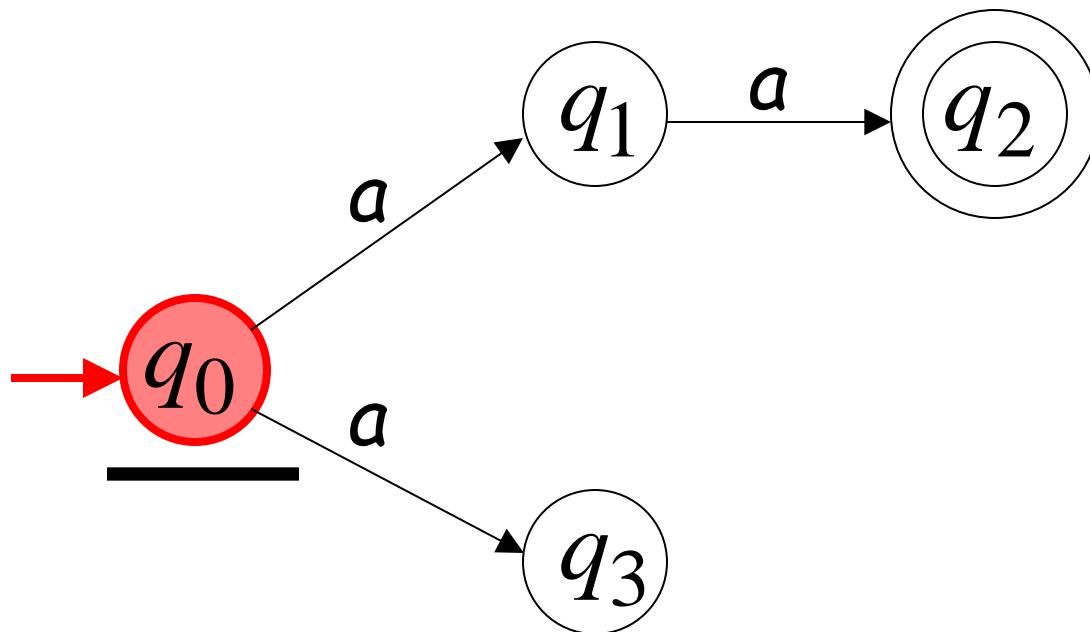
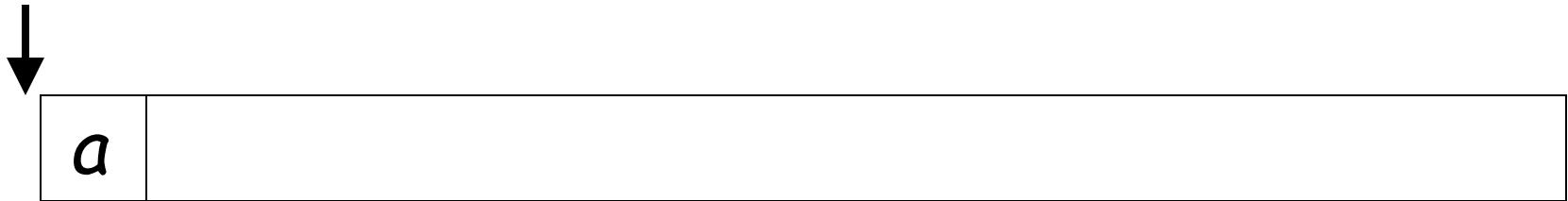
a	
-----	--



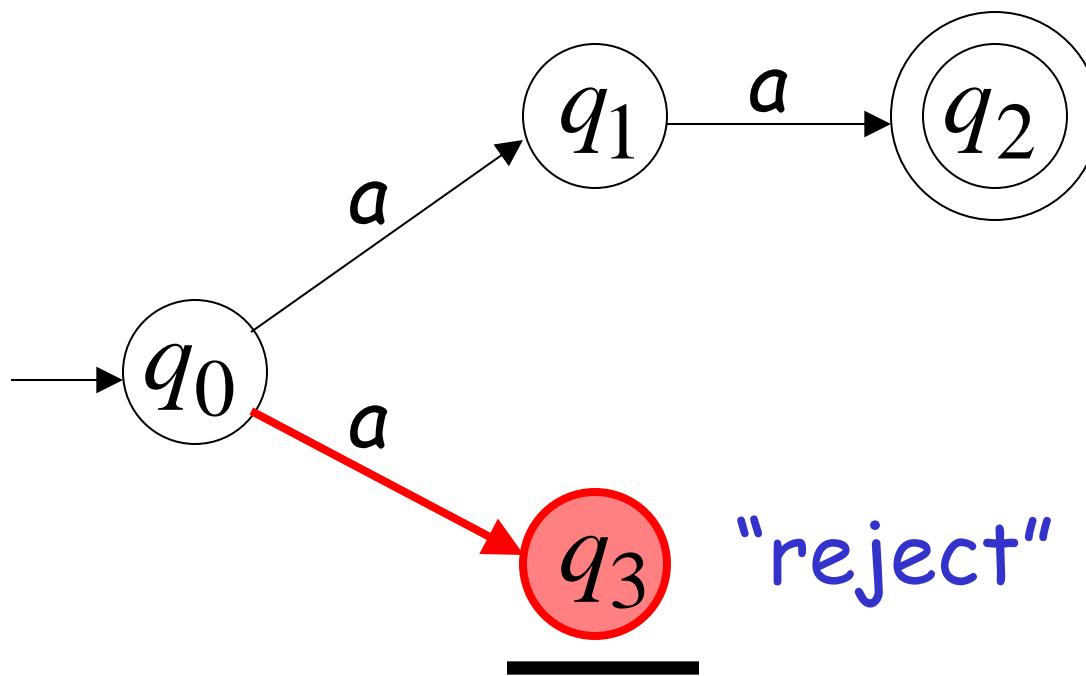
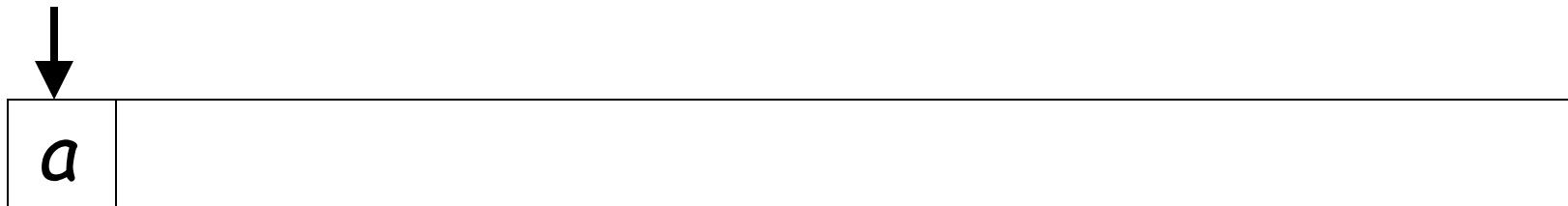
First Choice



Second Choice



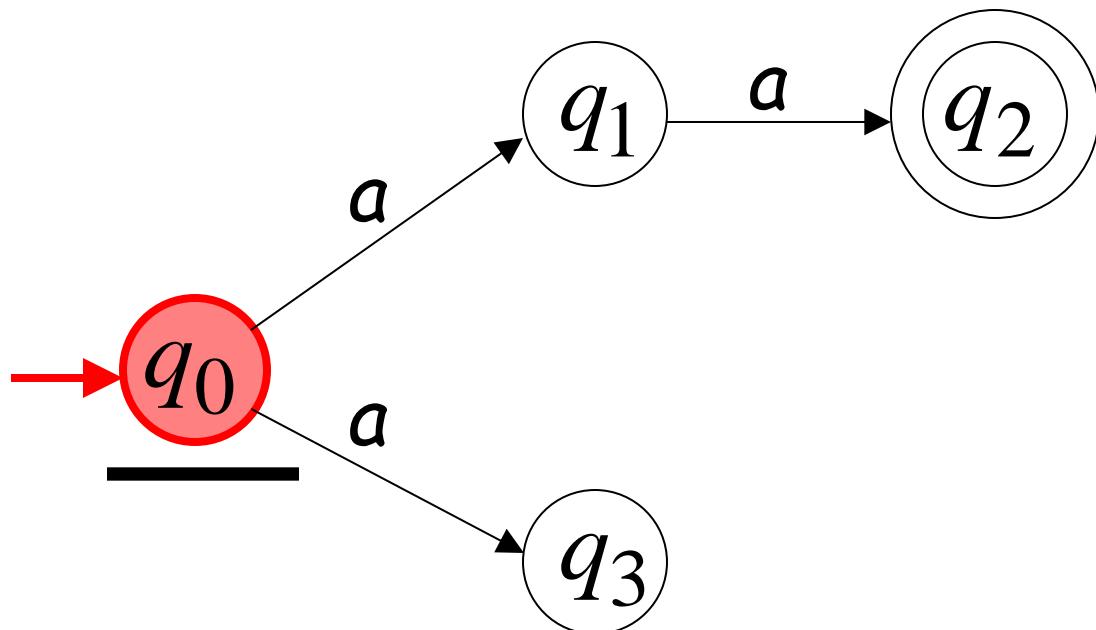
Second Choice



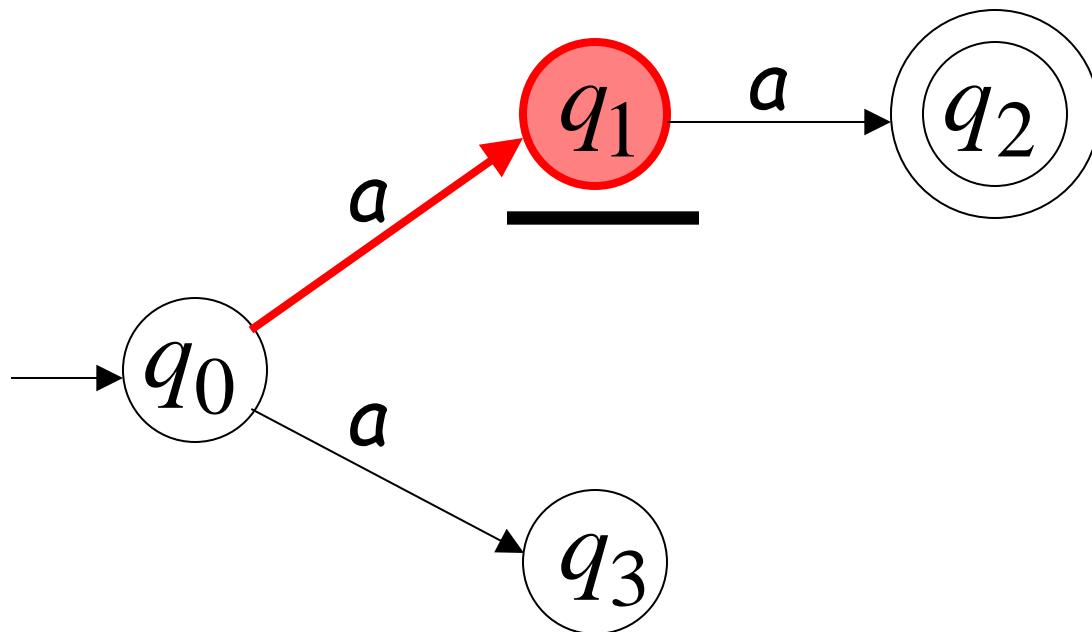
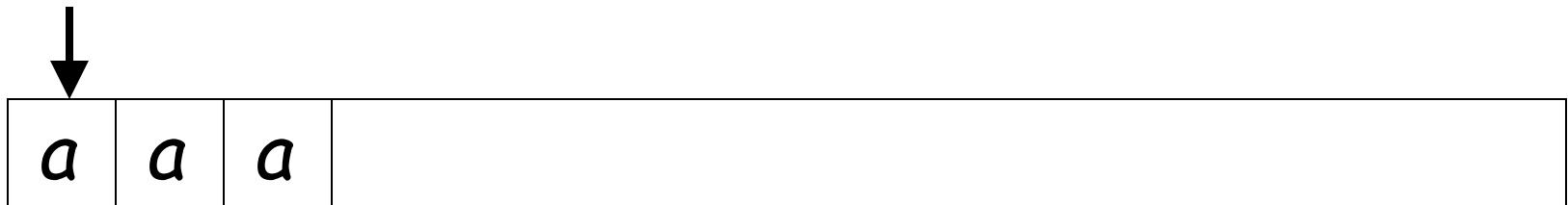
Another Rejection example



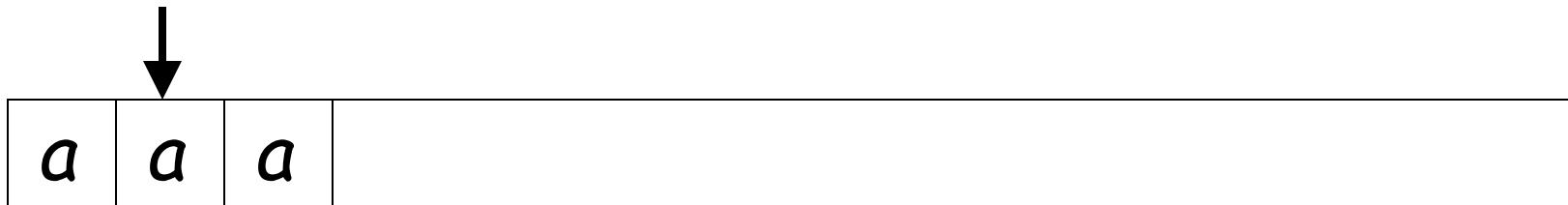
a	a	a	
-----	-----	-----	--



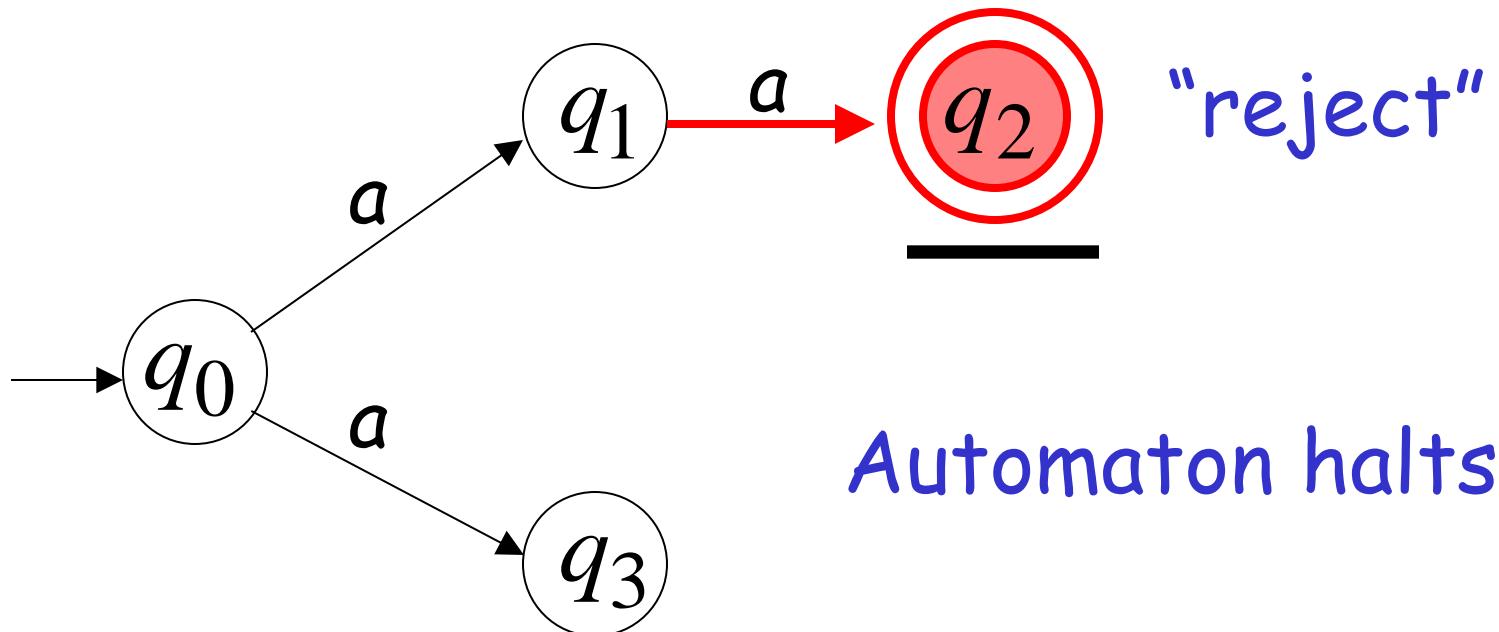
First Choice



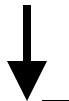
First Choice



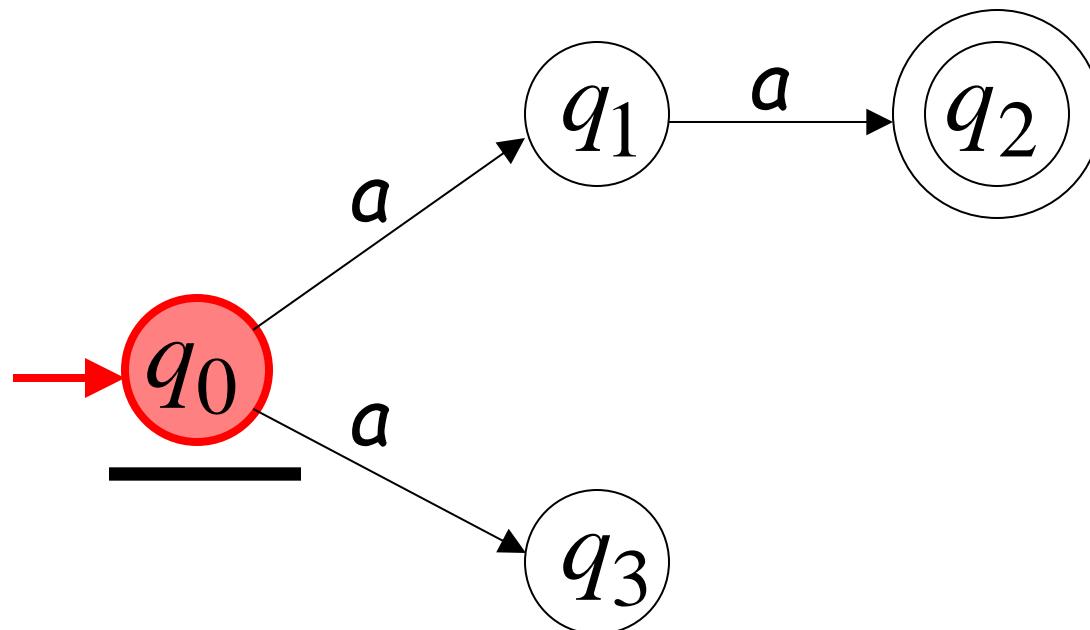
Input cannot be consumed



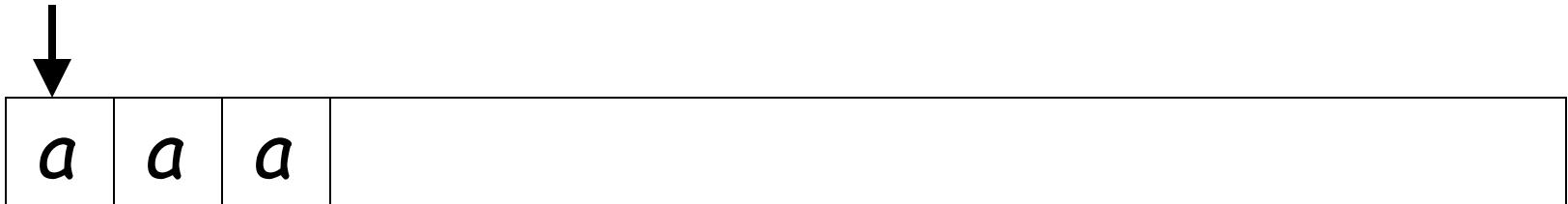
Second Choice



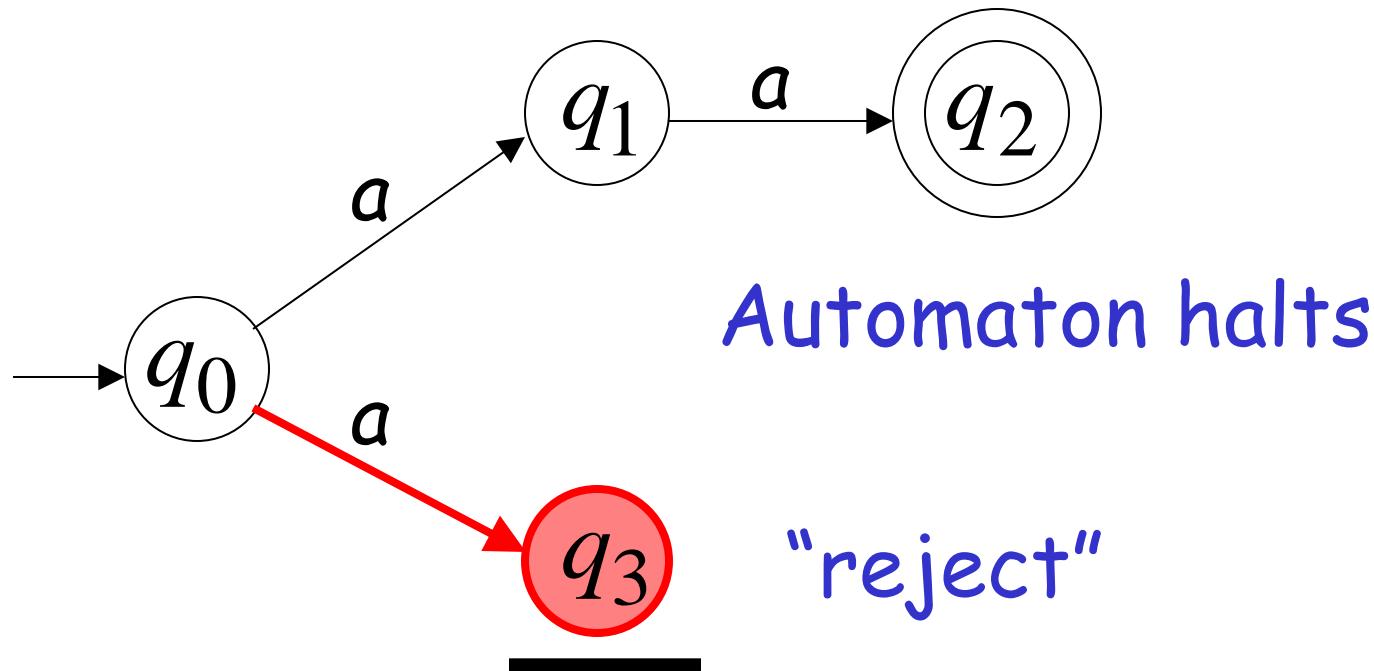
a	a	a	
-----	-----	-----	--



Second Choice



Input cannot be consumed



An NFA rejects a string:

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

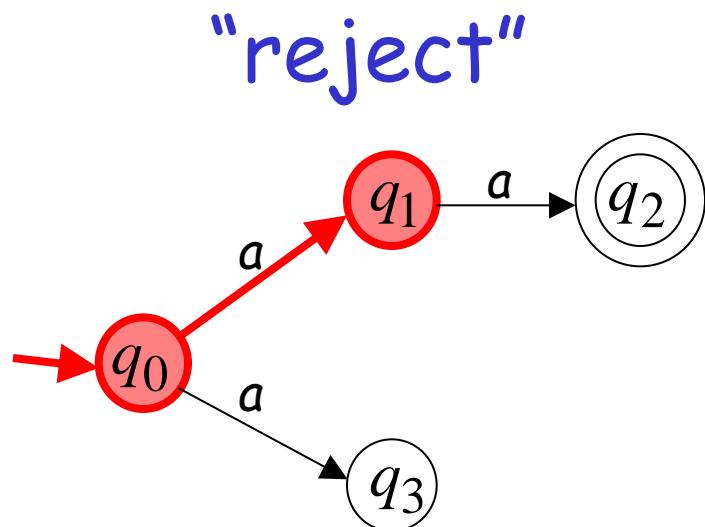
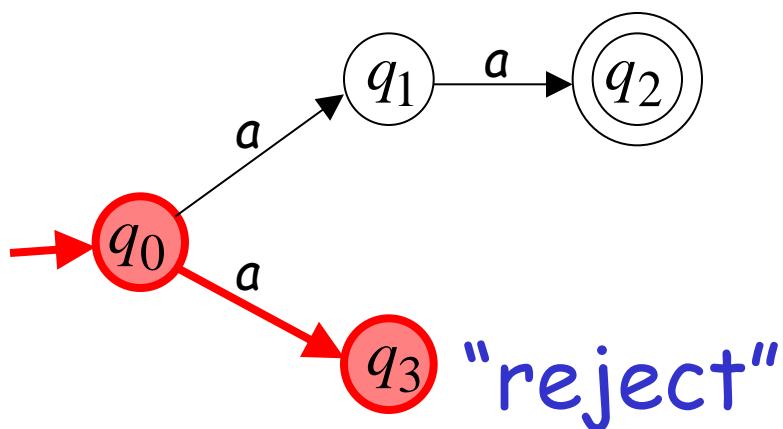
For each computation:

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

OR

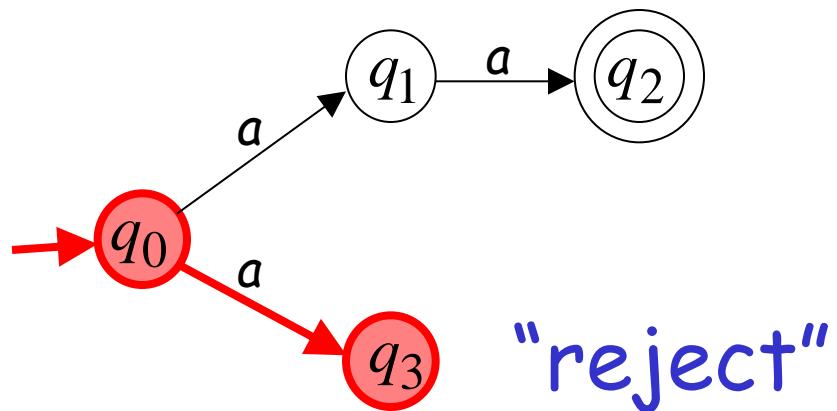
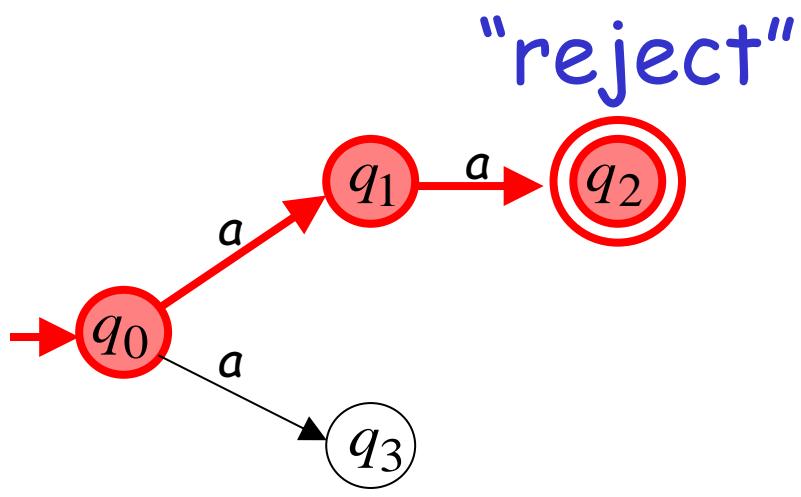
- The input cannot be consumed

a is rejected by the NFA:



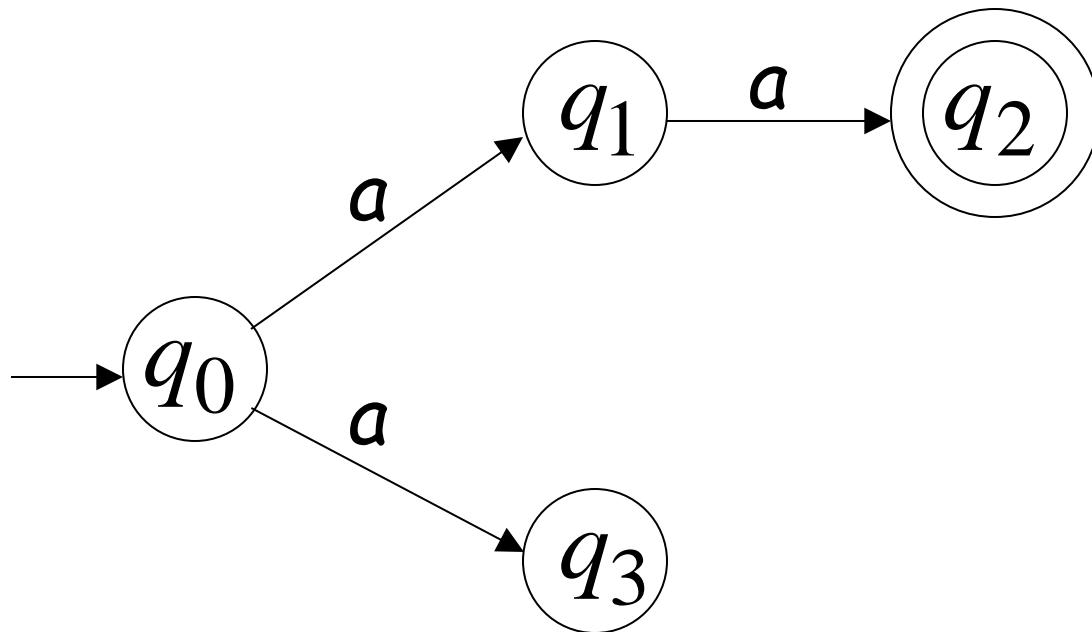
All possible computations lead to rejection

aaa is rejected by the NFA:

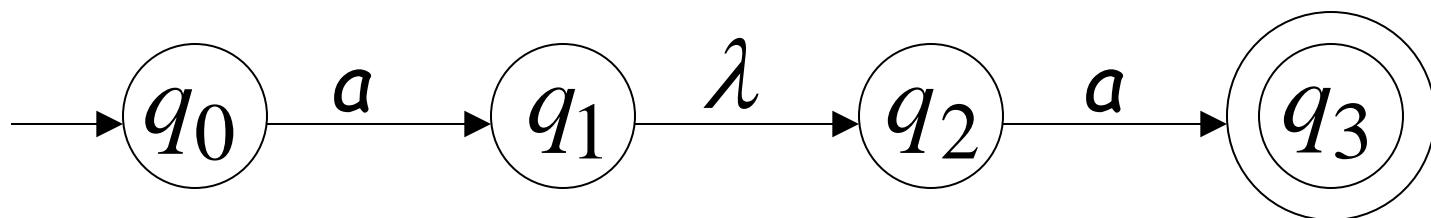


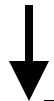
All possible computations lead to rejection

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

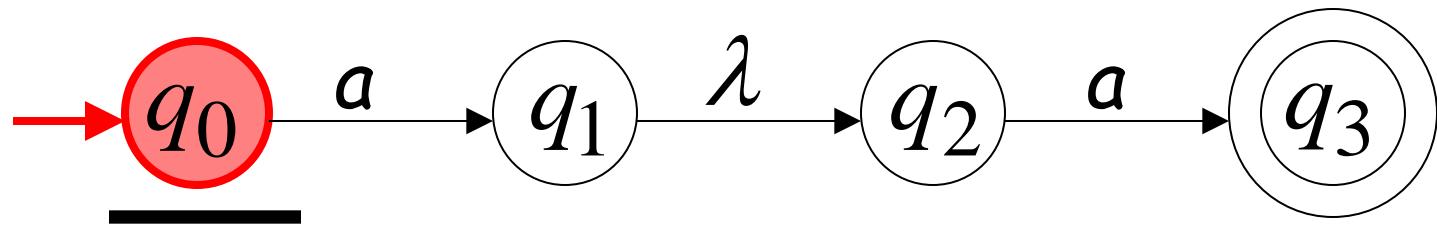


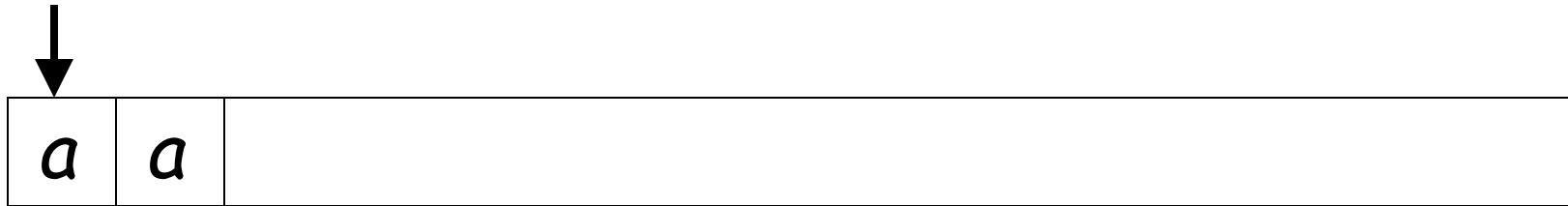
Lambda Transitions



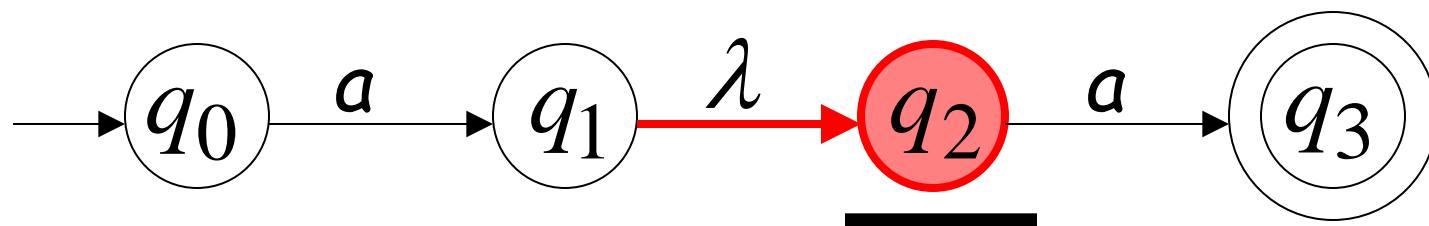
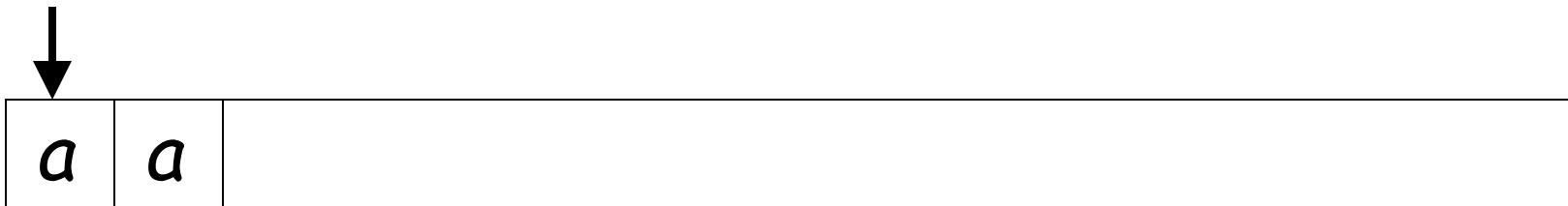


a	a	
-----	-----	--





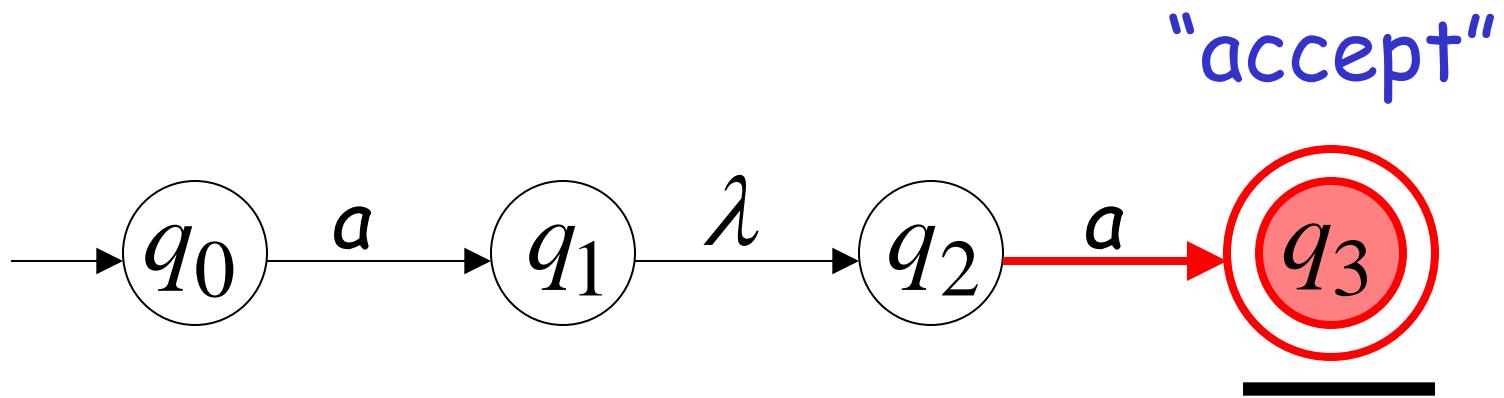
input tape head does not move



all input is consumed



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

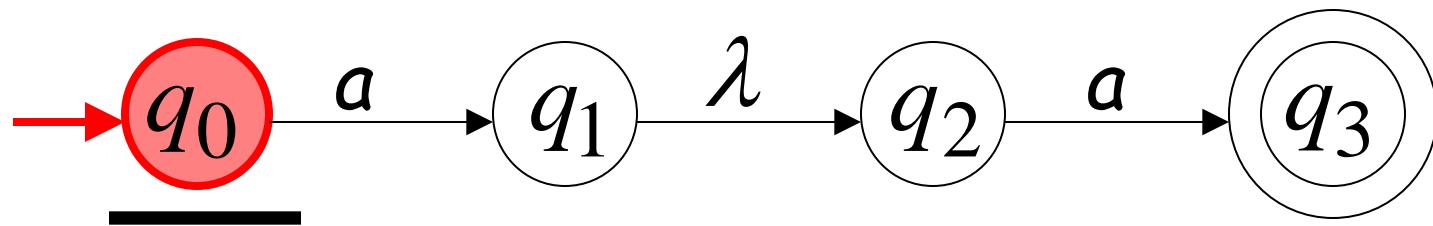


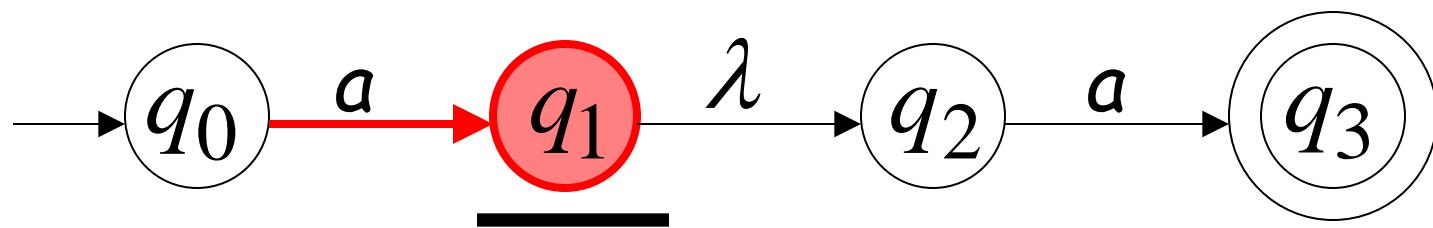
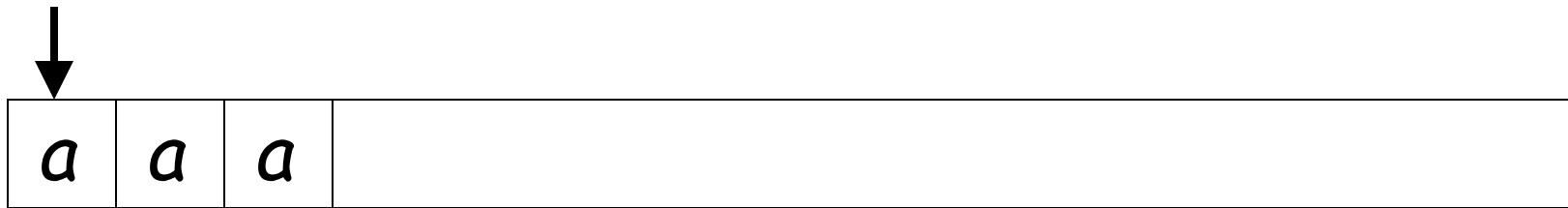
String aa is accepted

Rejection Example

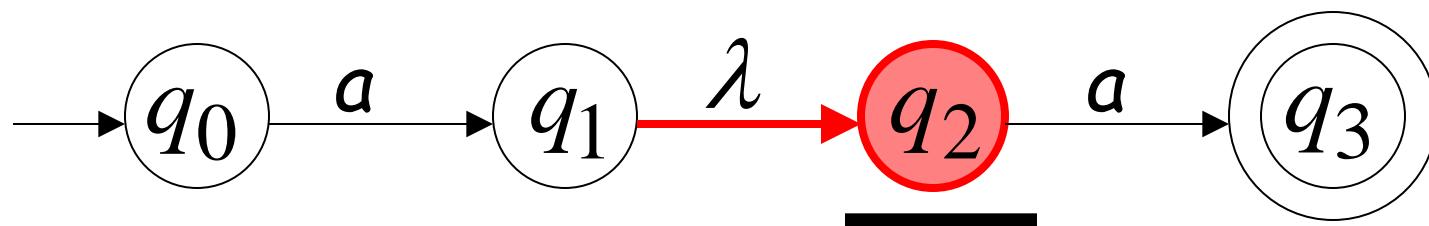


a	a	a	
---	---	---	--





(read head doesn't move)



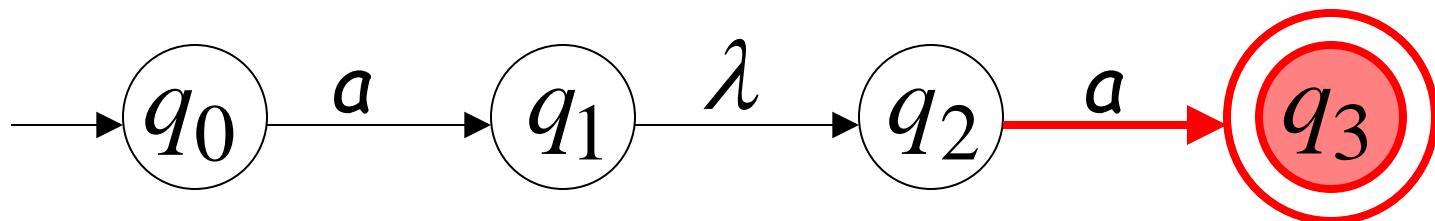
Input cannot be consumed



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

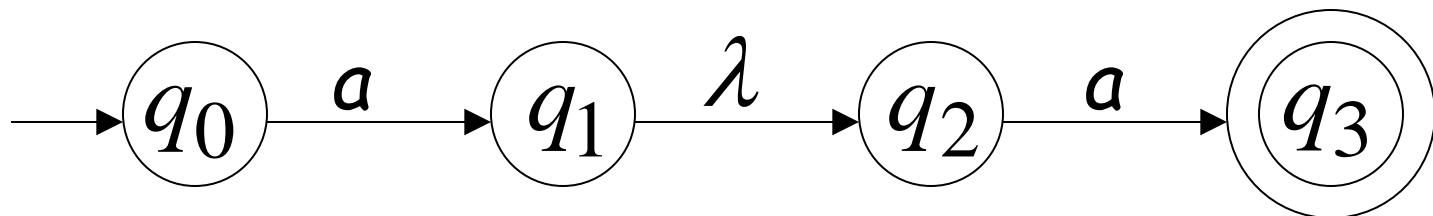
Automaton halts

"reject"

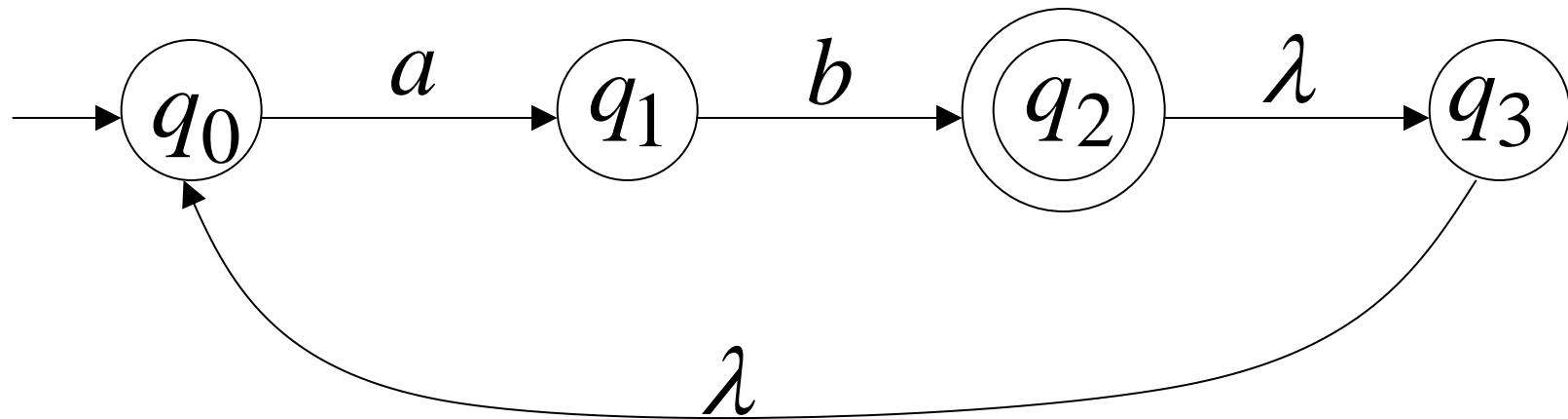


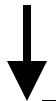
String aaa is rejected

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

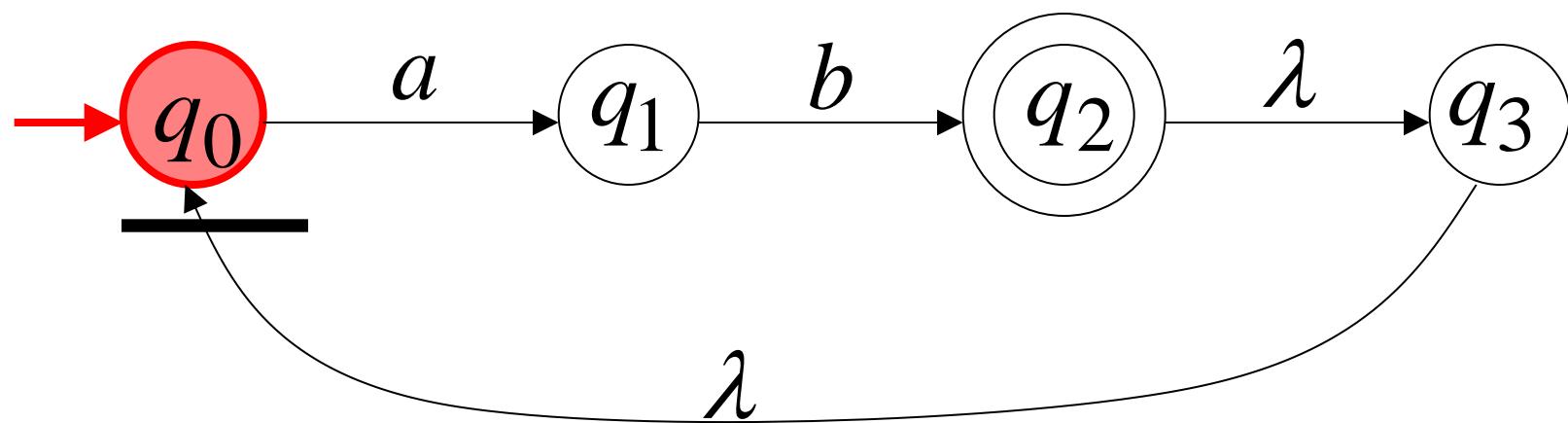


Another NFA Example

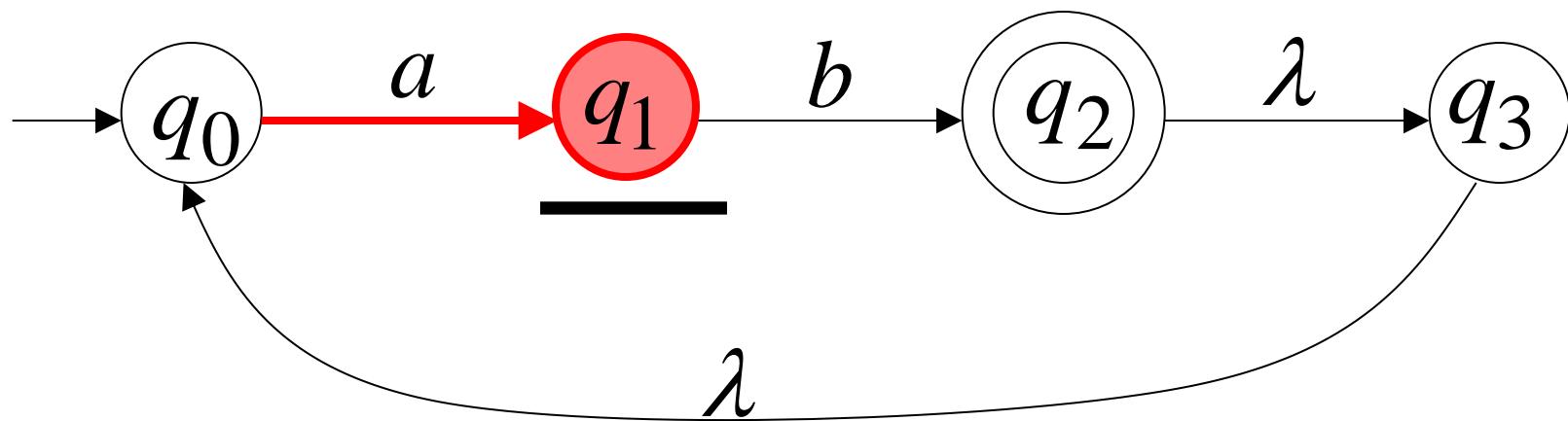




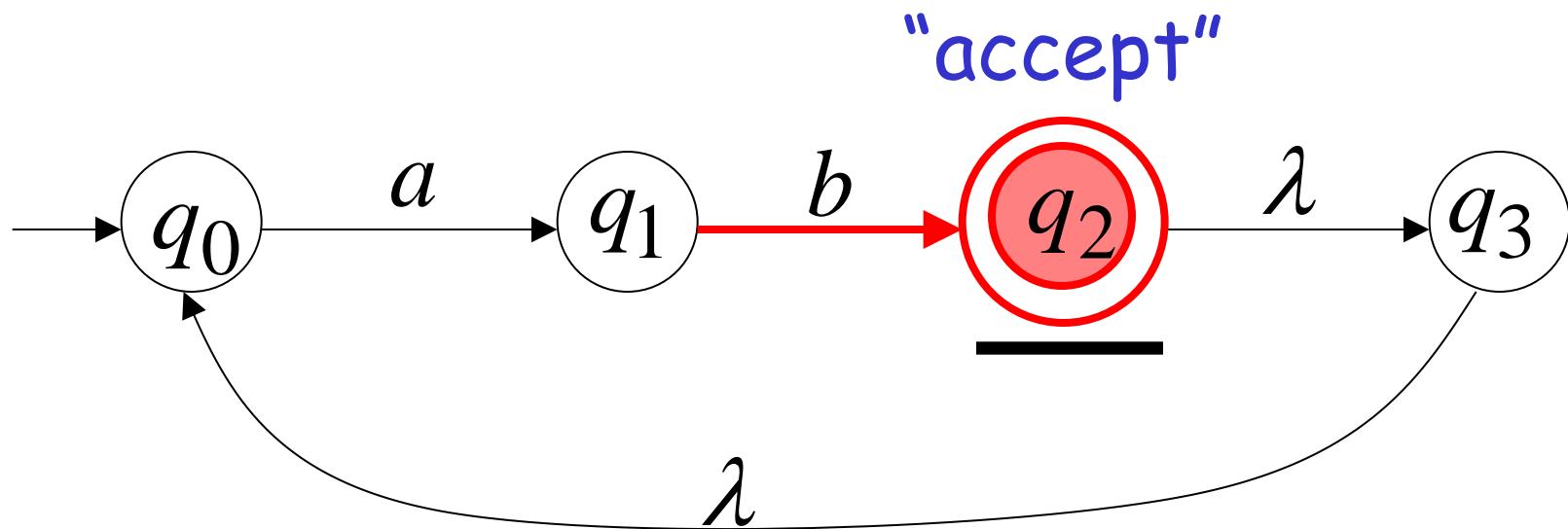
a	b	
-----	-----	--



a	b	



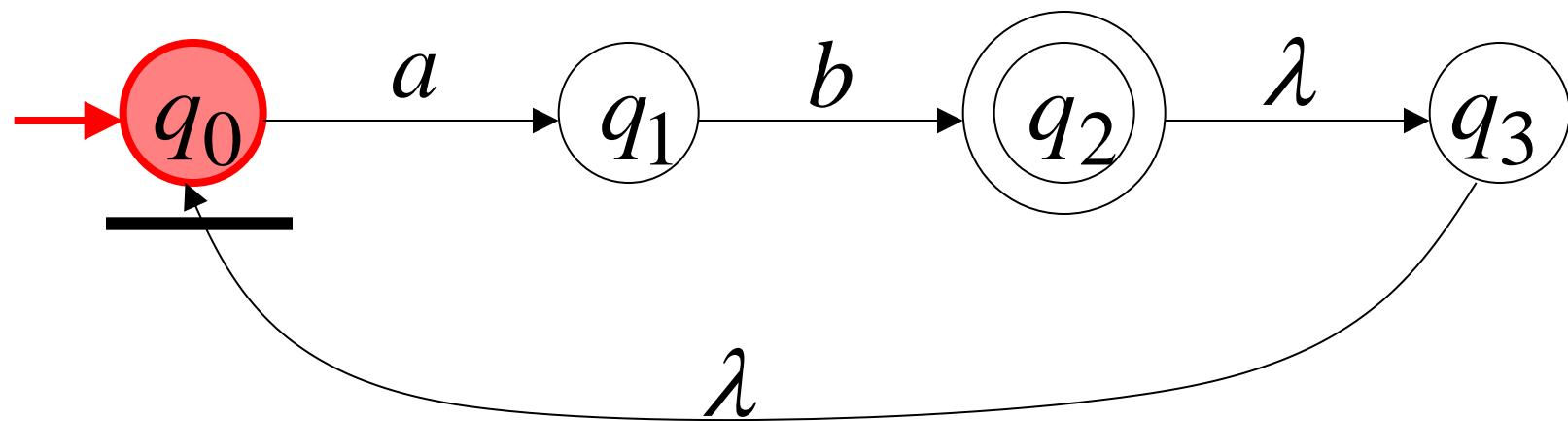
a	b	
-----	-----	--



Another String

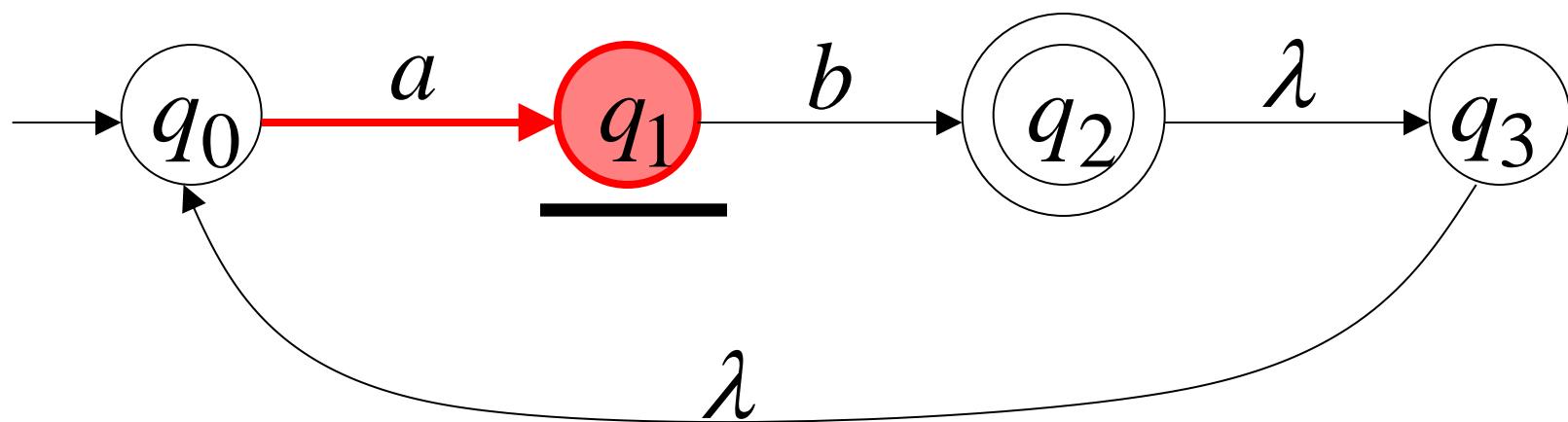


a	b	a	b	
-----	-----	-----	-----	--

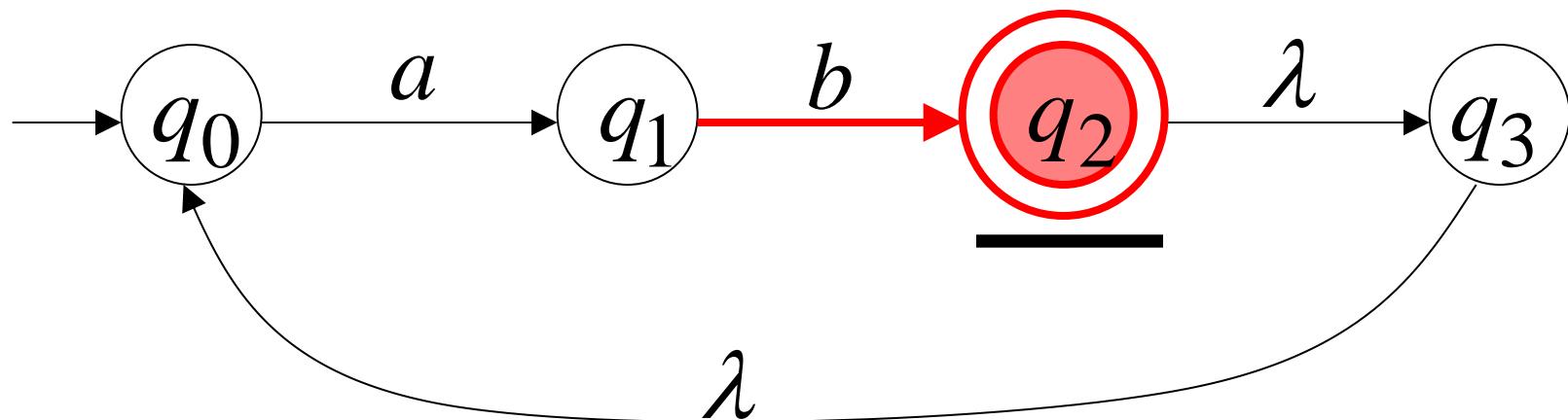


↓

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

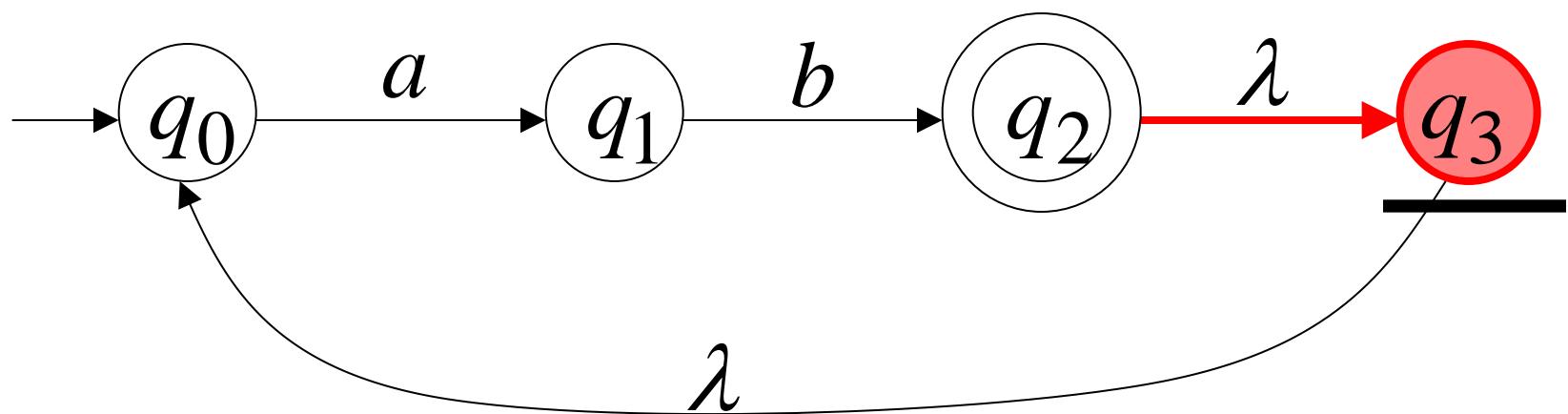


a	b	a	b	

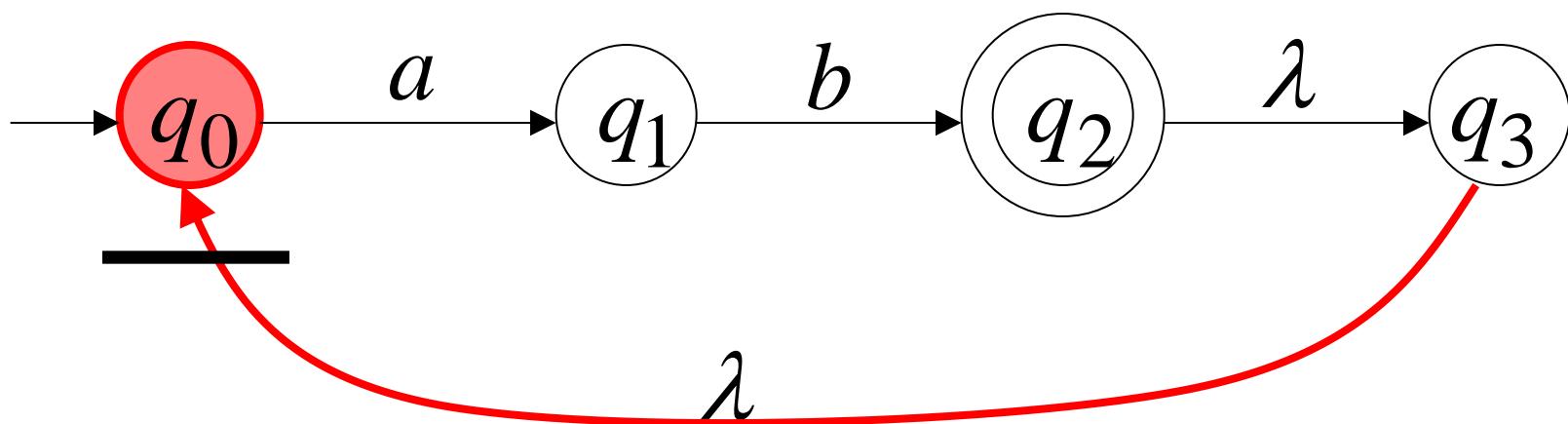


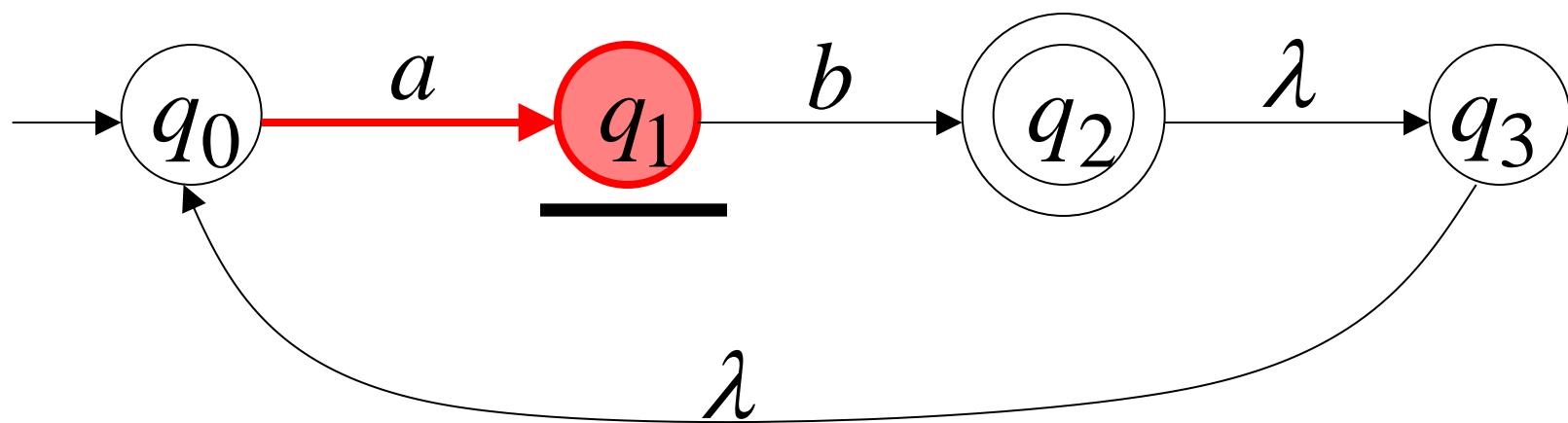
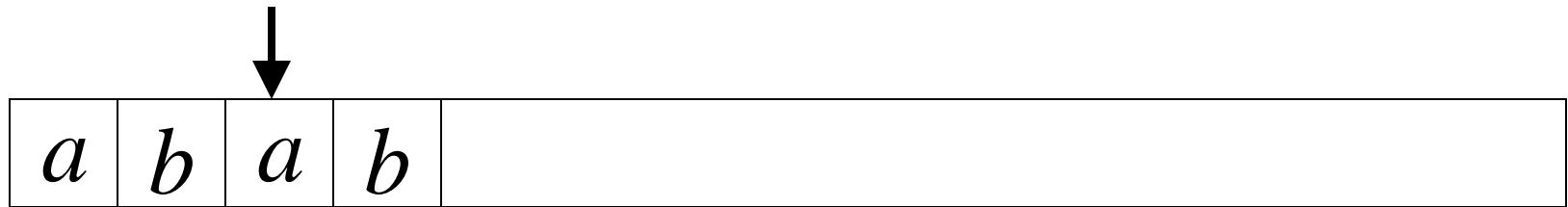
↓

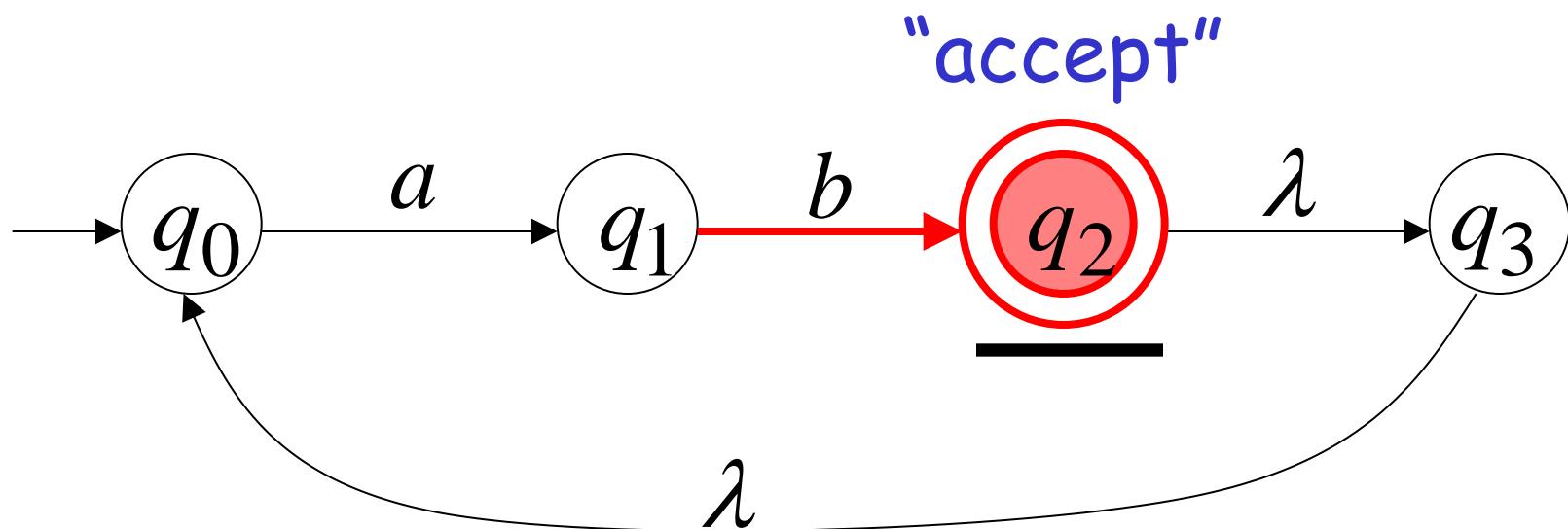
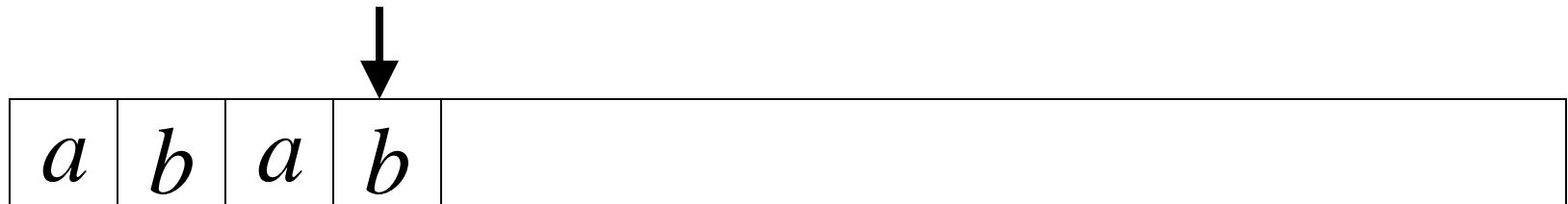
a	b	a	b	
-----	-----	-----	-----	--



a	b	a	b	



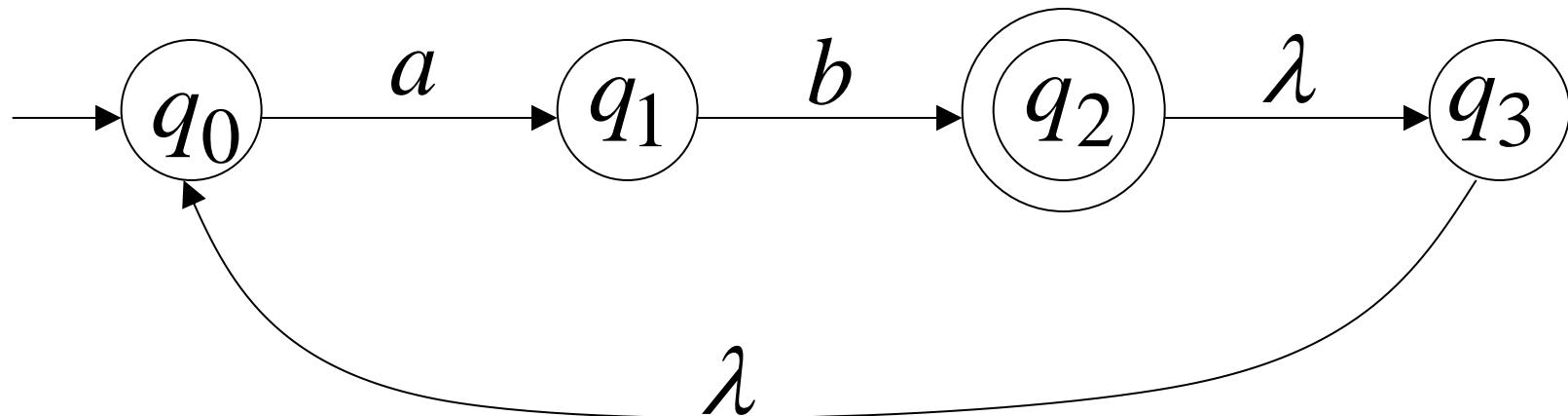




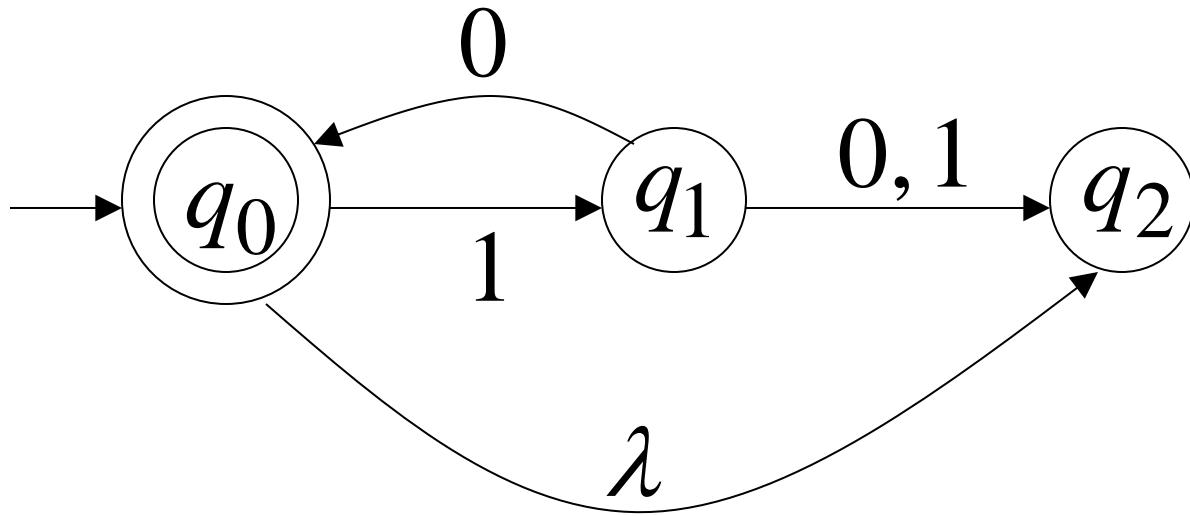
Language accepted

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

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

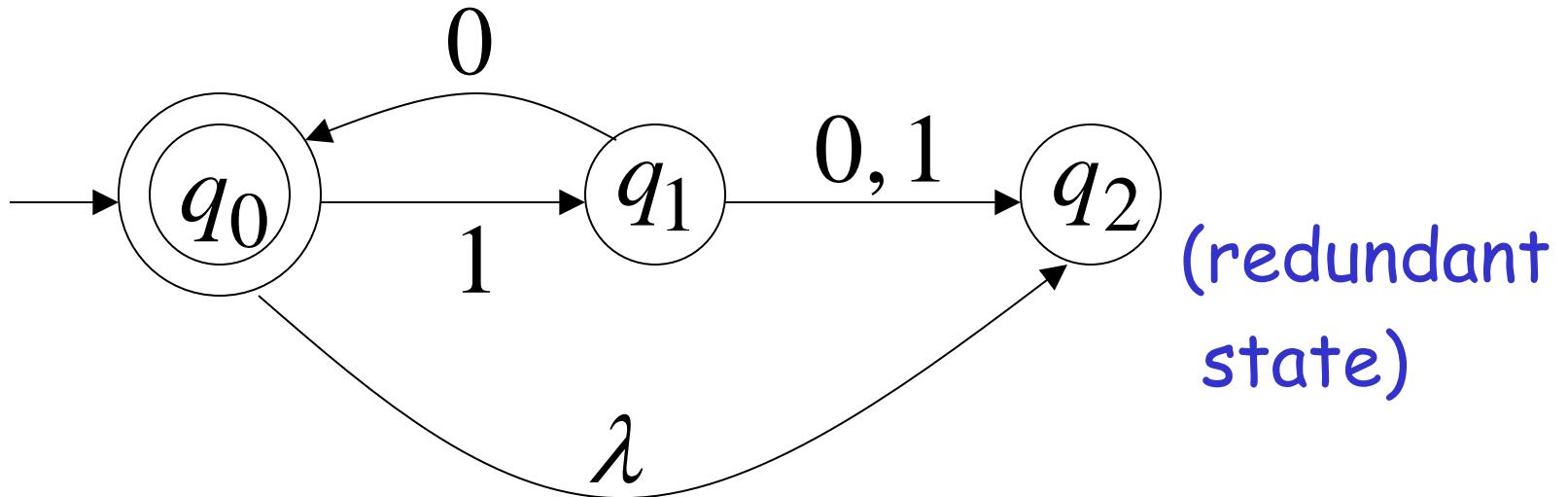


Another NFA Example



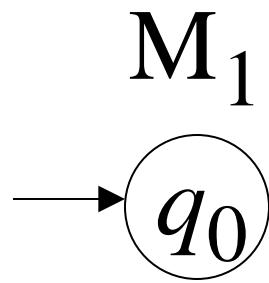
Language accepted

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

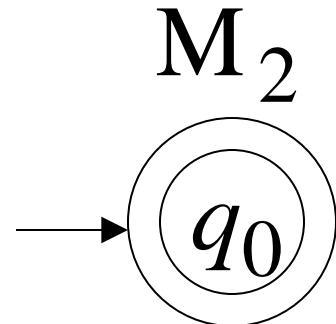


Remarks:

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

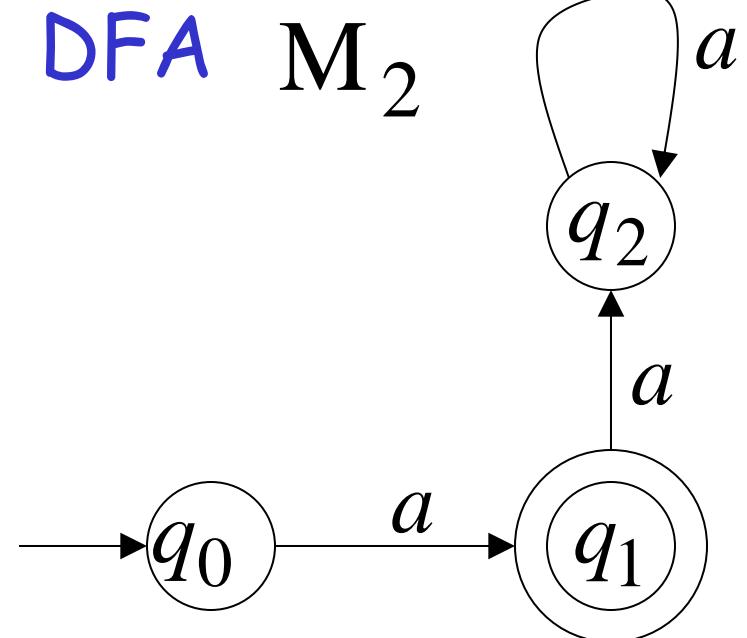
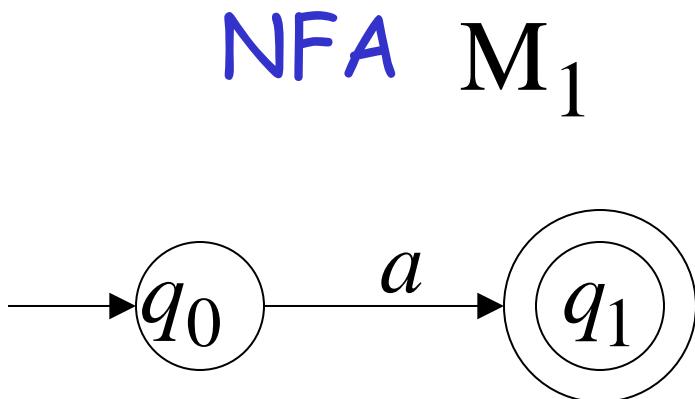


$$L(M_1) = \{ \}$$



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

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



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

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

Formal Definition of NFAs

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

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

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

δ : Transition function

q_0 : Initial state

F : Accepting states

Definition

Given an alphabet Σ , we define

$$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\},$$

which is the set of all strings over Σ of length 0 or 1.

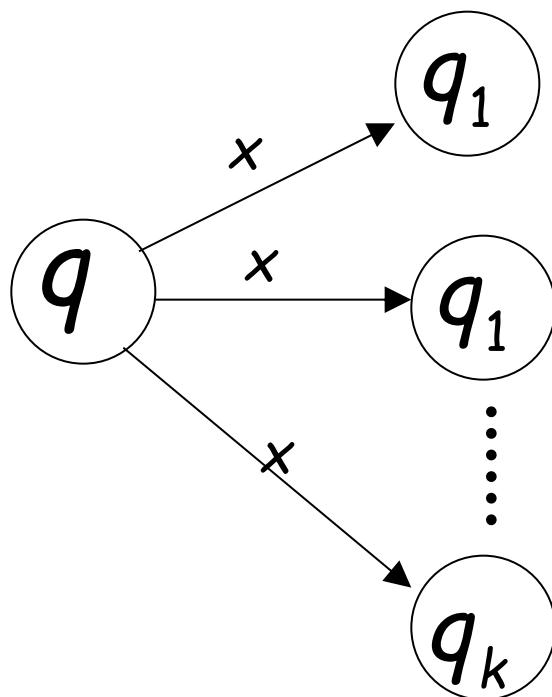
Definition

A *nondeterministic finite automaton with ε -transitions* (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- ▶ Q is a finite set (the set of states),
- ▶ Σ is a finite set (the alphabet),
- ▶ $\delta : Q \times \Sigma_\varepsilon \rightarrow 2^Q$ is the transition function,
- ▶ $q_0 \in Q$ is the start state, and
- ▶ $F \subseteq Q$ is the set of accepting states.

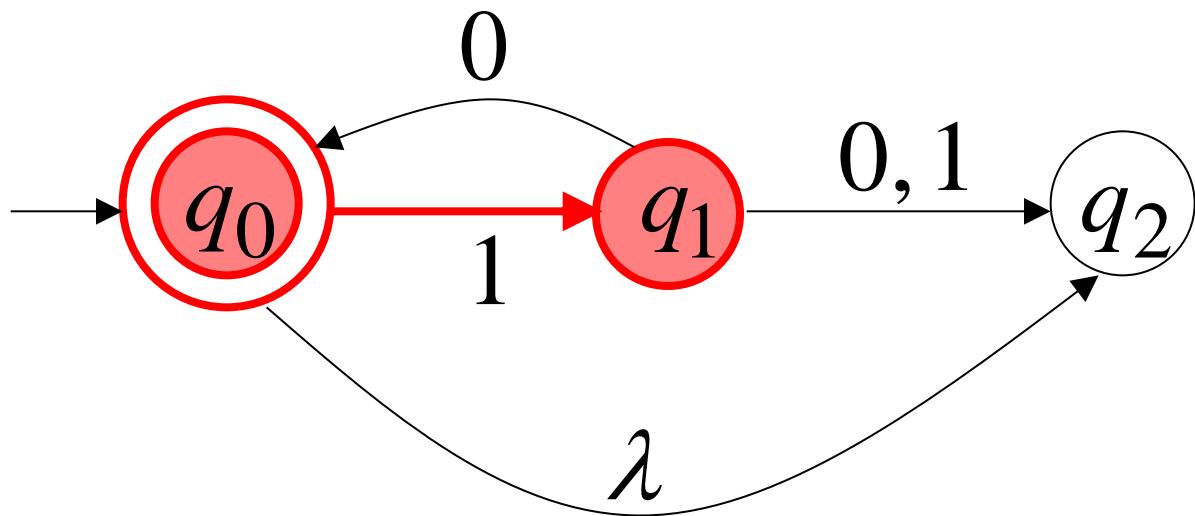
Transition Function δ

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

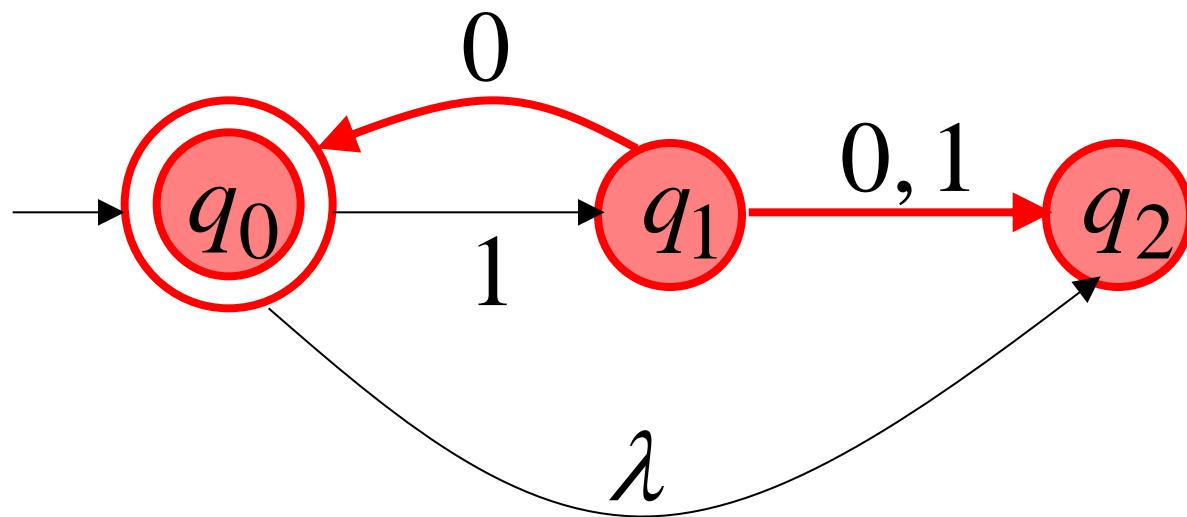


resulting states with
following **one** transition
with symbol x

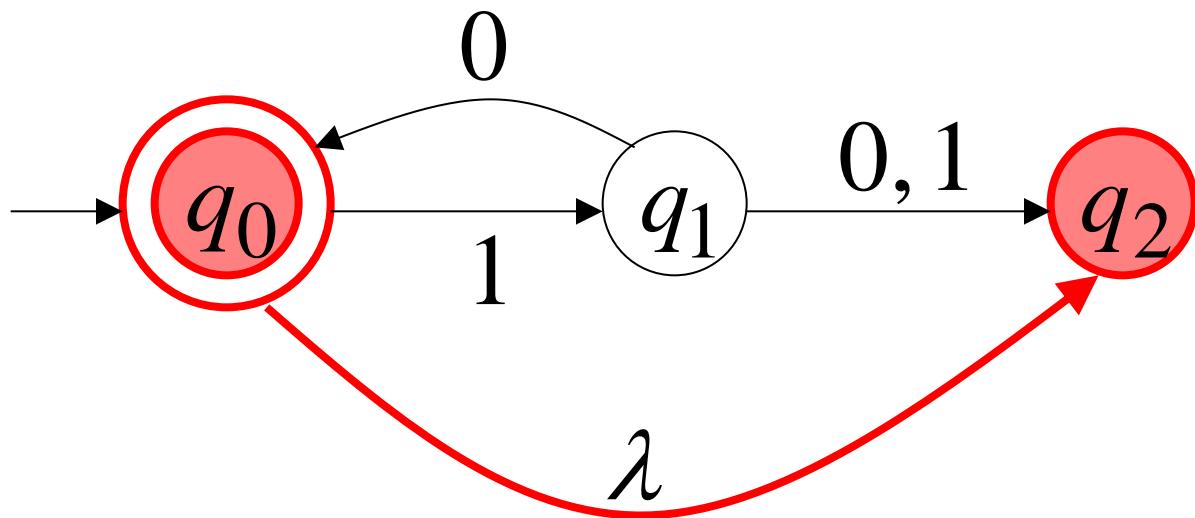
$$\delta(q_0, 1) = \{q_1\}$$



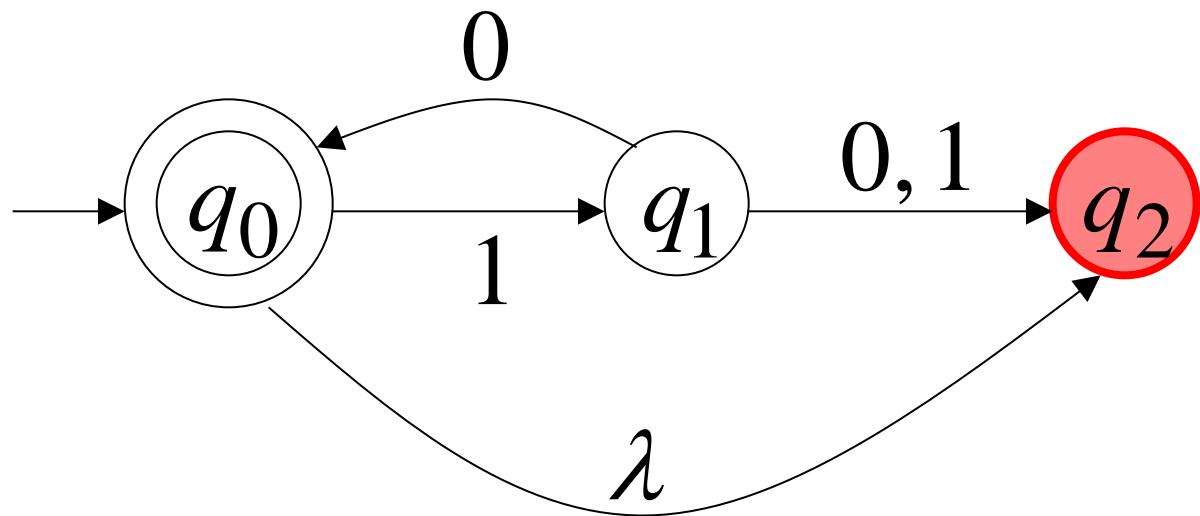
$$\delta(q_1, 0) = \{q_0, q_2\}$$



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



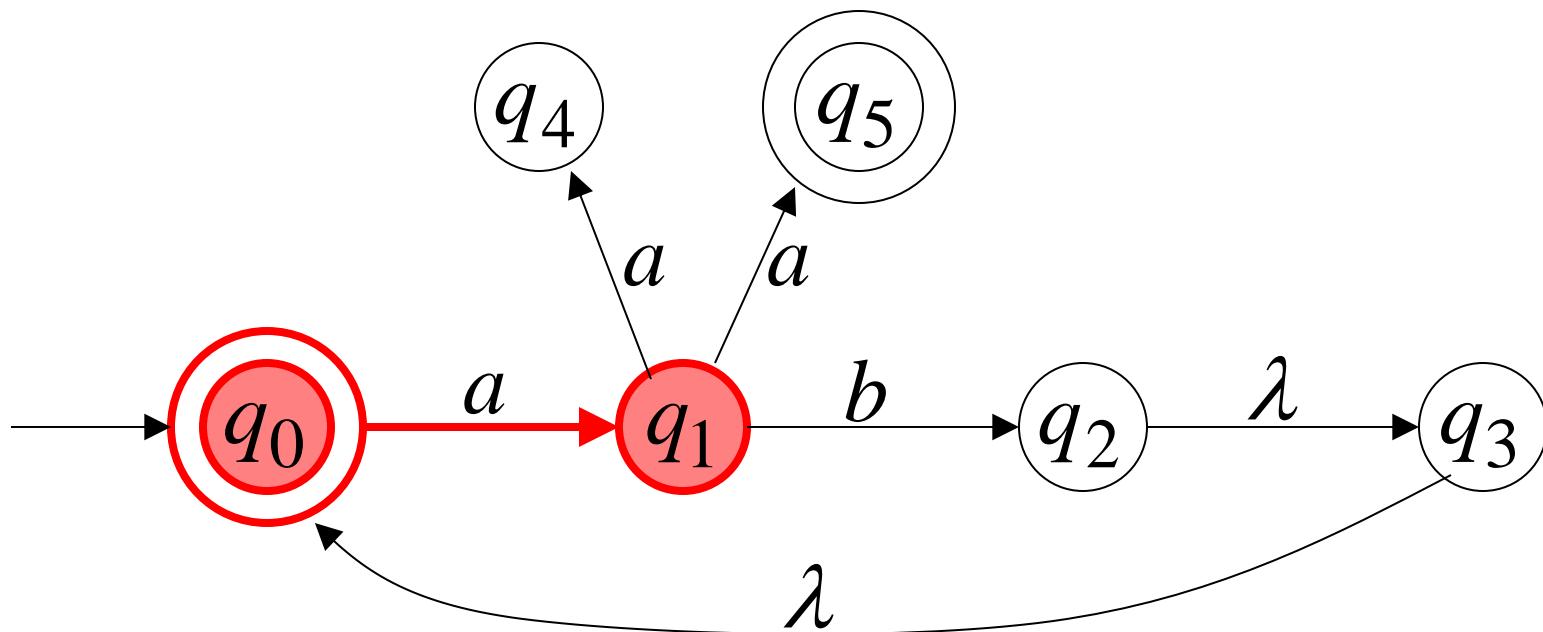
$$\delta(q_2, 1) = \emptyset$$



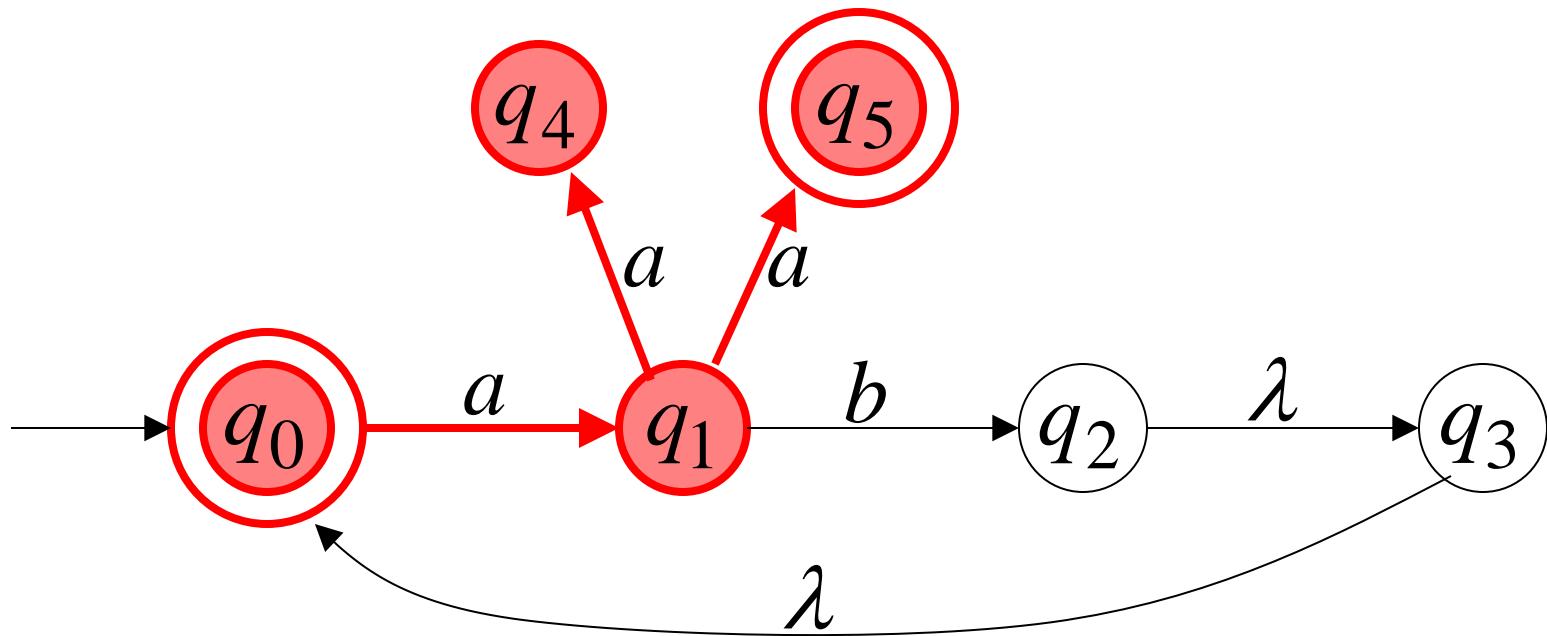
Extended Transition Function δ^*

Same with δ but applied on strings

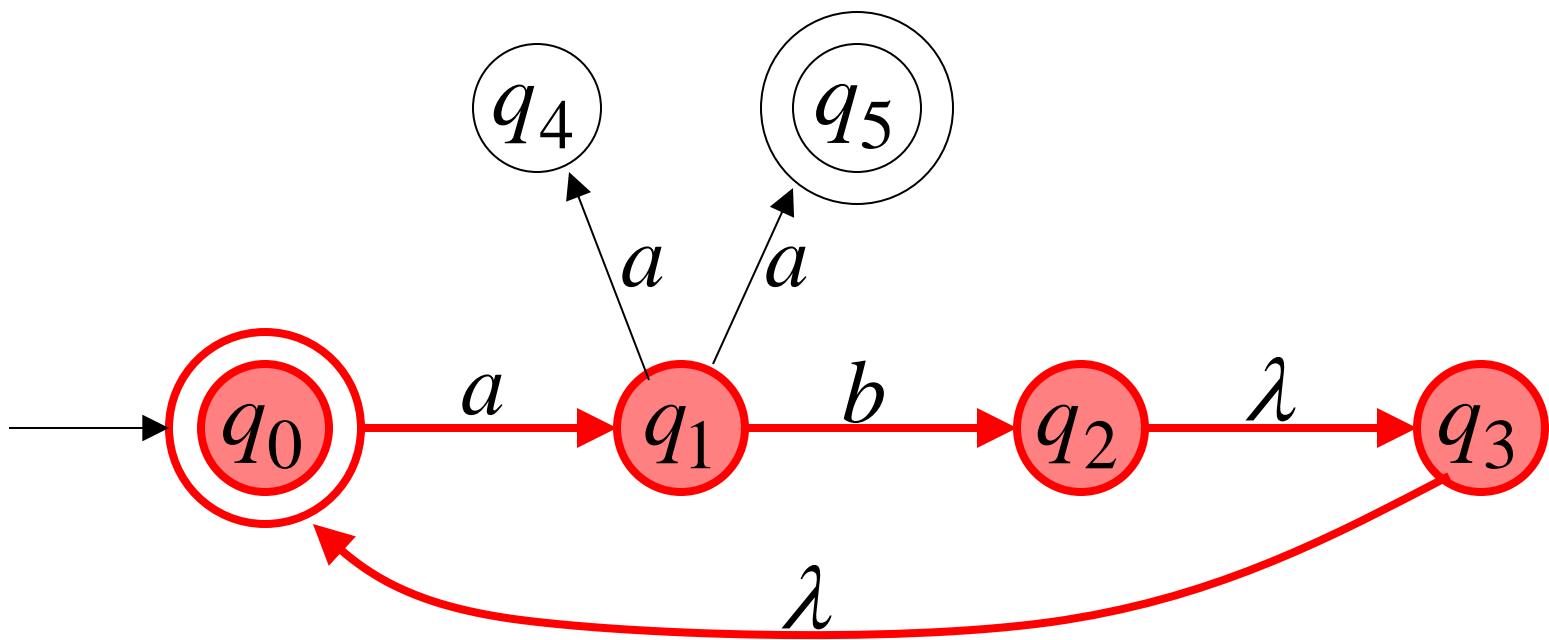
$$\delta^*(q_0, a) = \{q_1\}$$



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



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



Special case:

for any state q

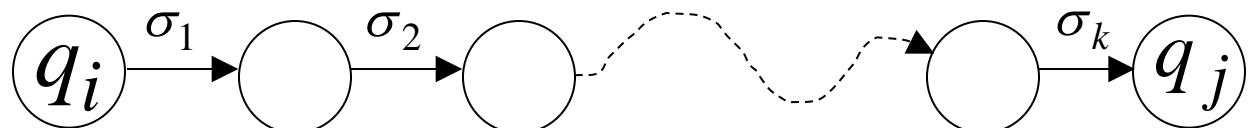
$$q \in \delta^*(q, \lambda)$$

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 M

The language accepted by M is:

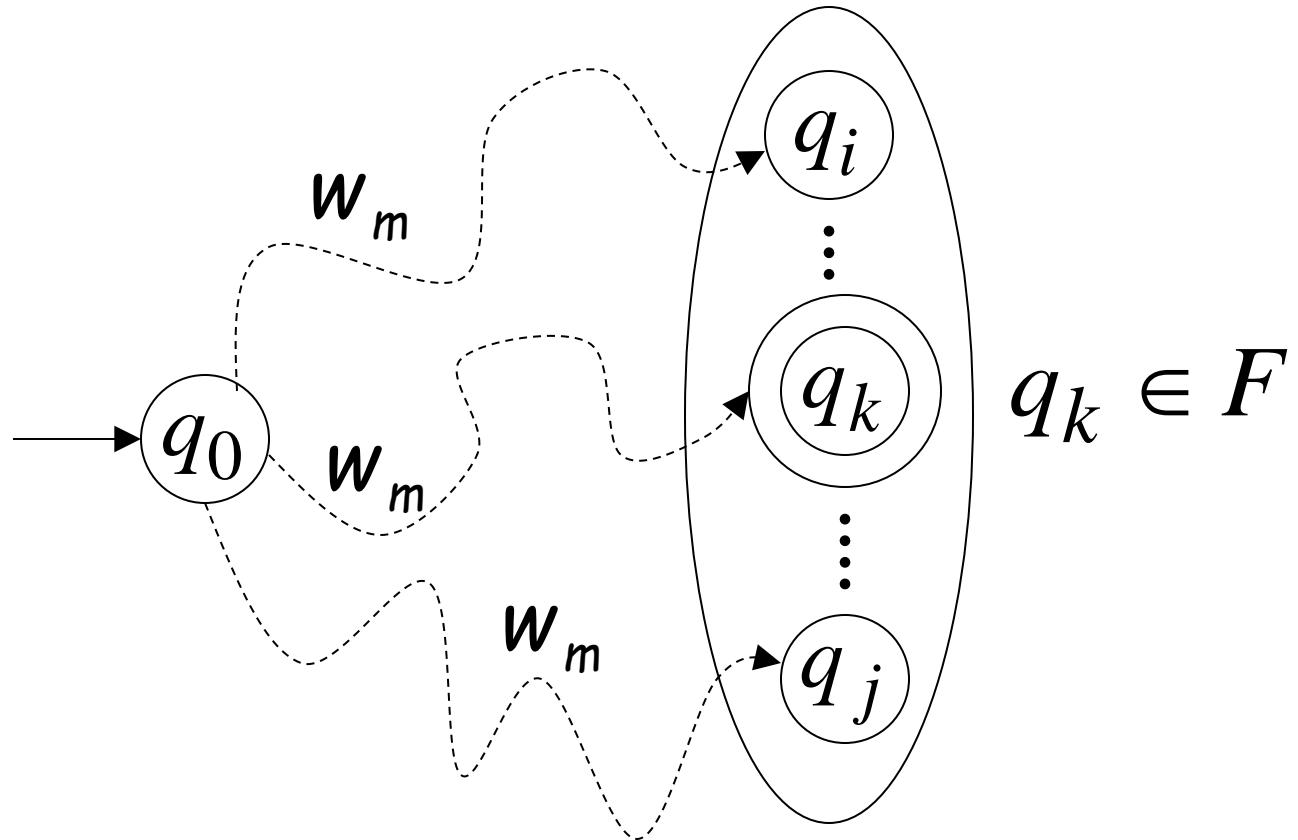
$$L(M) = \{w_1, w_2, \dots, w_n\}$$

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

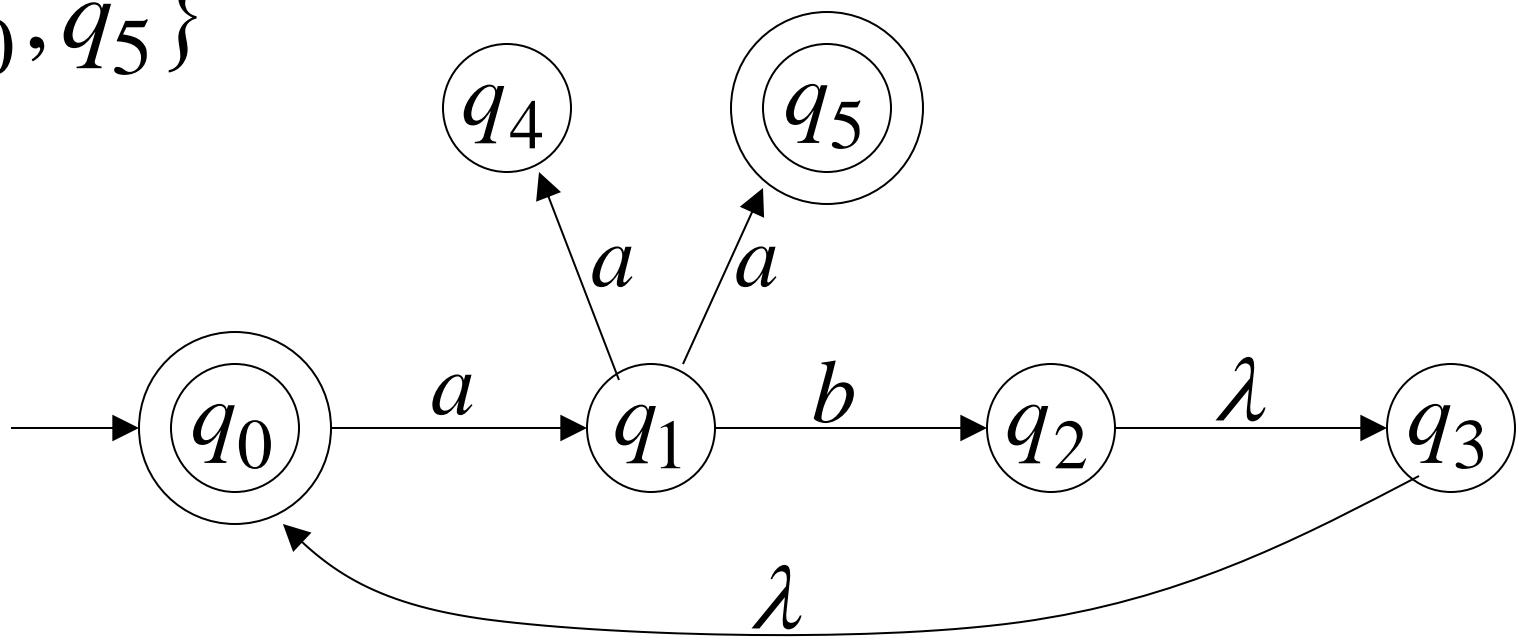
and there is some $q_k \in F$ (accepting state)

$$w_m \in L(M)$$

$$\delta^*(q_0, w_m)$$



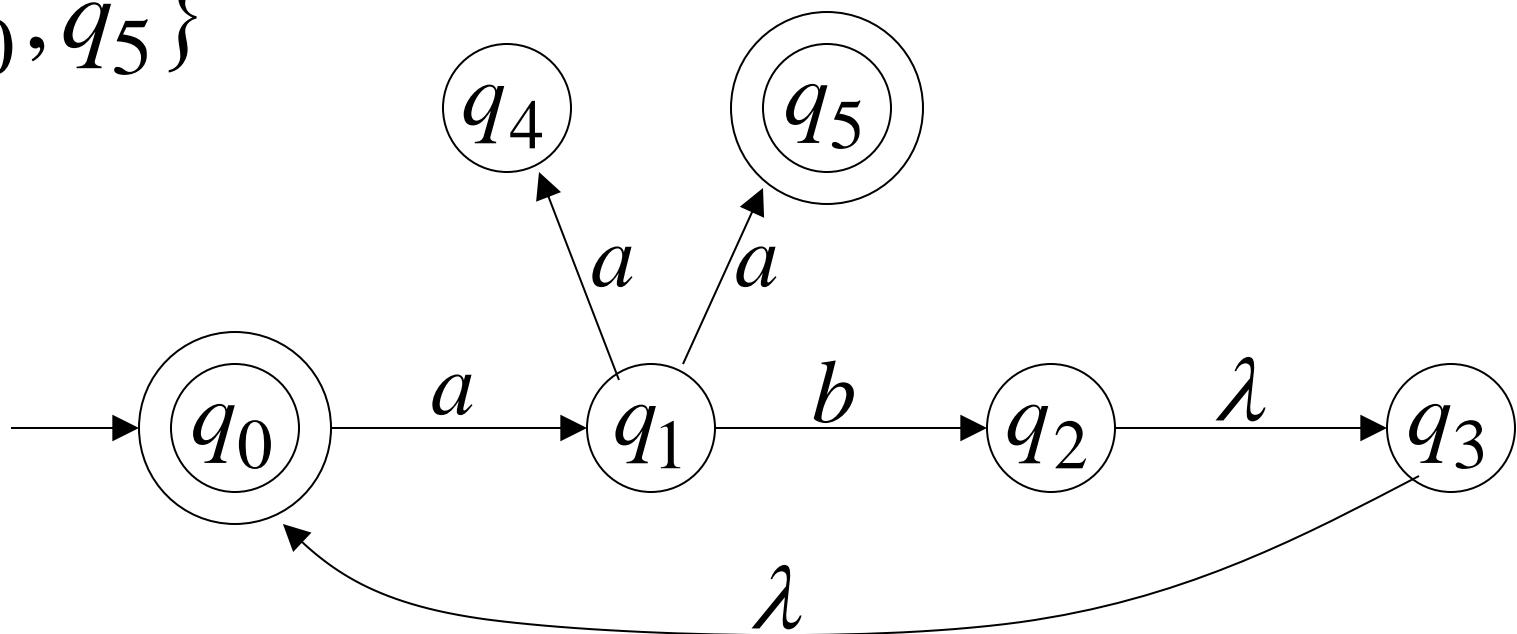
$$F = \{q_0, q_5\}$$



$$\delta^*(q_0, aa) = \{q_4, \underline{q_5}\} \quad \longrightarrow \quad aa \in L(M)$$

$\in F$

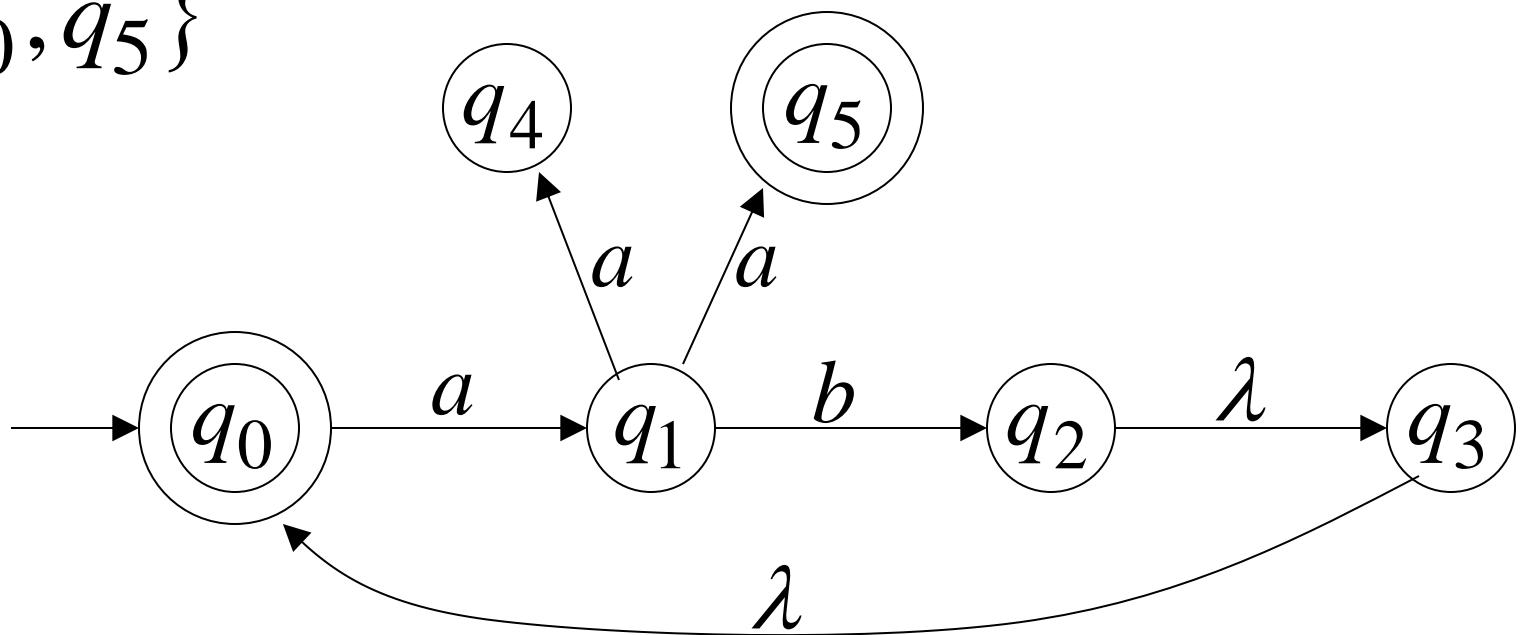
$$F = \{q_0, q_5\}$$



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

$\searrow \in F$

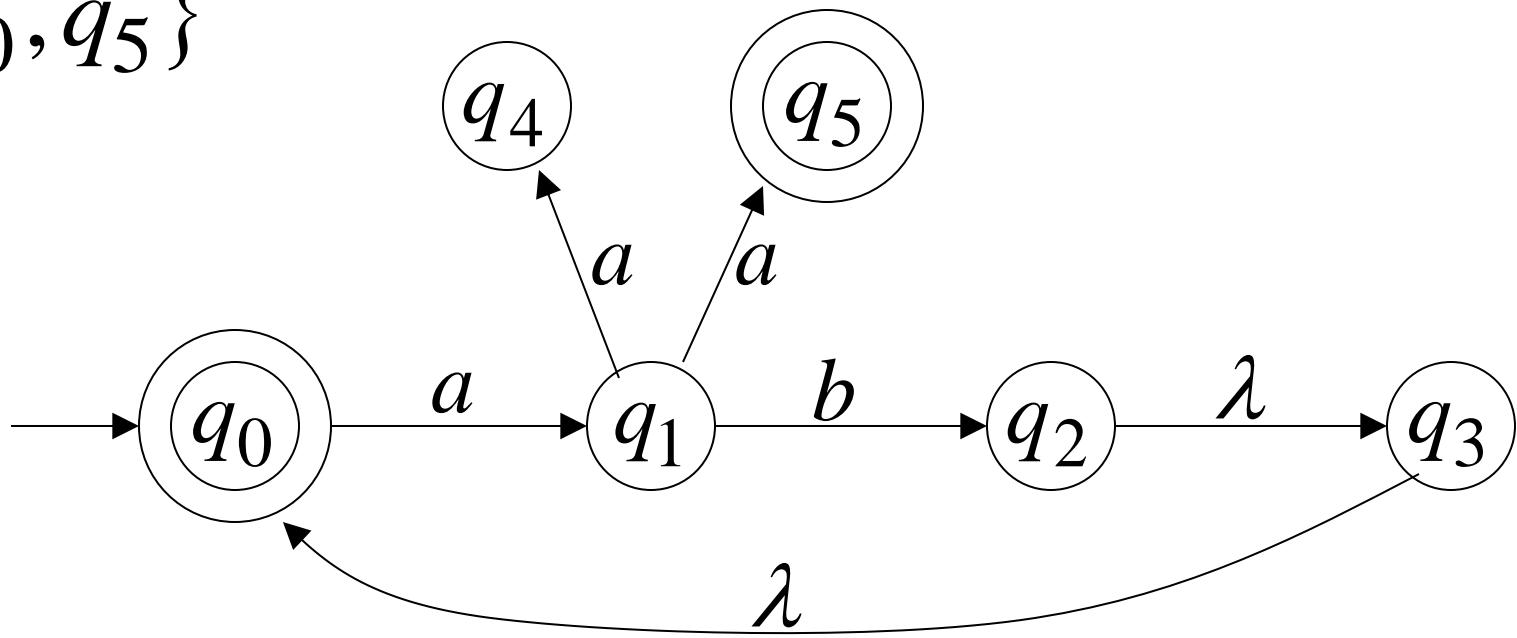
$$F = \{q_0, q_5\}$$



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

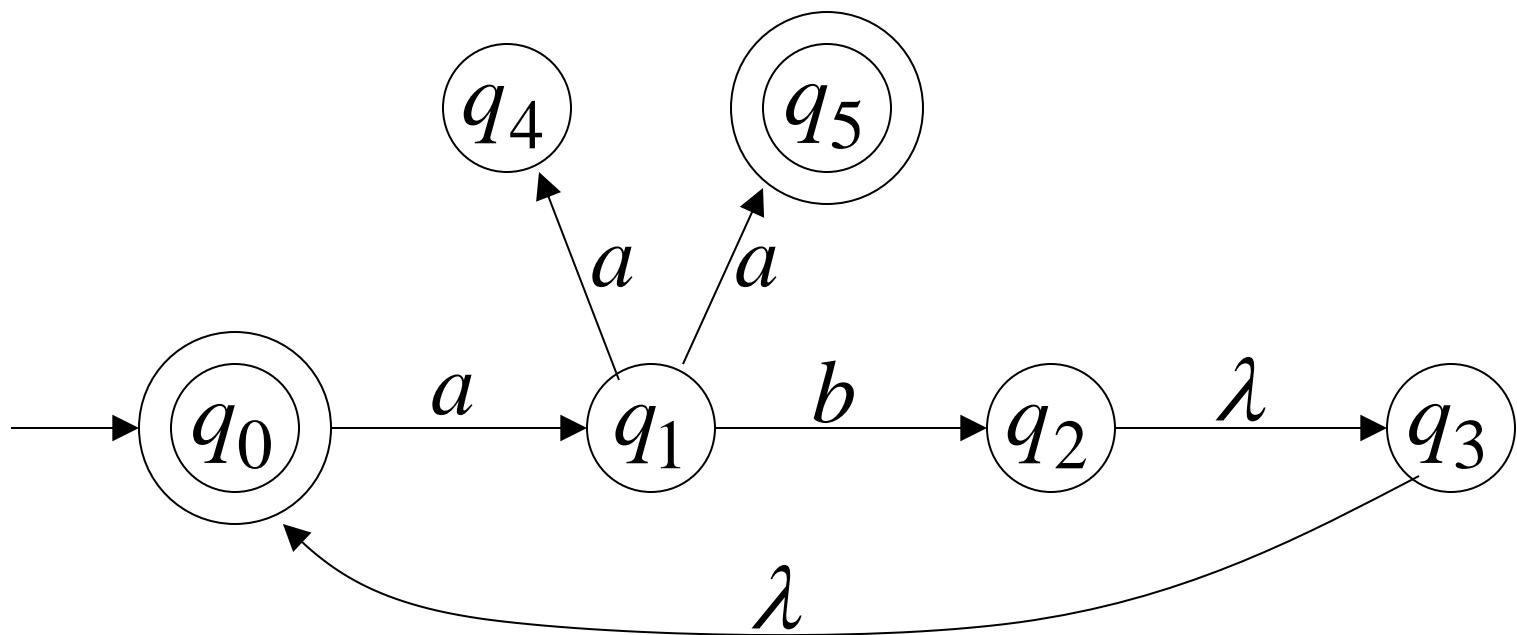
$\searrow \in F$

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



$$\delta^*(q_0, aba) = \{q_1\} \quad \longrightarrow \quad aba \notin L(M)$$

$\not\in F$



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

NFAs accept the Regular
Languages

Equivalence of Machines

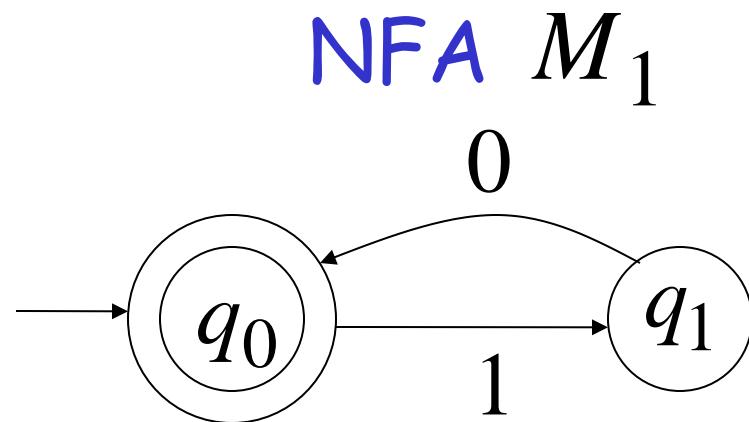
Definition:

Machine M_1 is equivalent to machine M_2

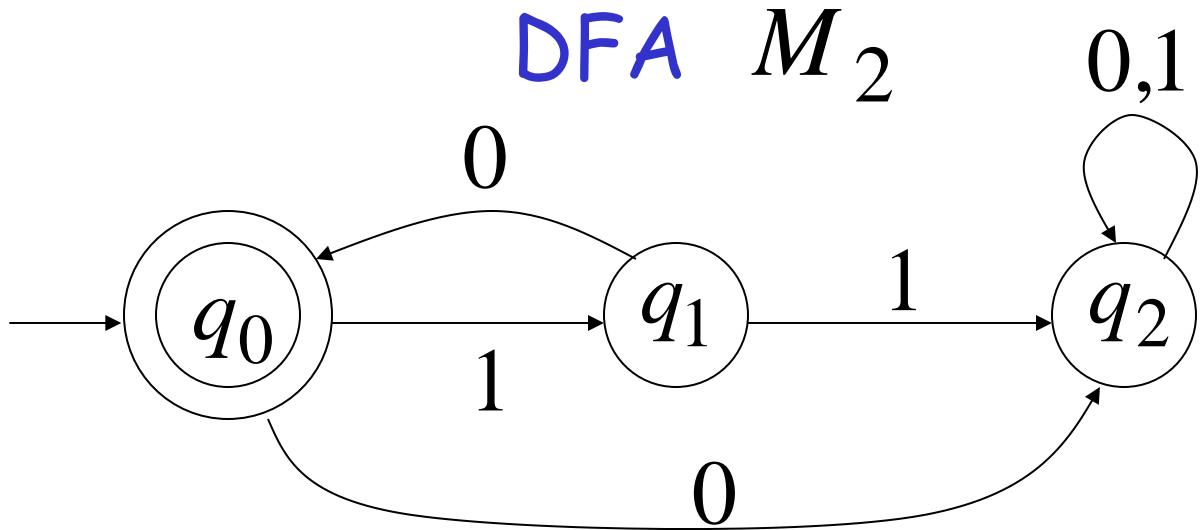
if $L(M_1) = L(M_2)$

Example of equivalent machines

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



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



Theorem:

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

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

Proof: we only need to show

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

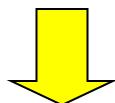
AND

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

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

Every DFA is trivially an NFA

For any DFA D , we build an equivalent NFA N with the same transition diagram, i.e., each $\delta_N(q, a)$ is a singleton for $a \in \Sigma$, and $\delta_N(q, \epsilon) = \emptyset$. More formally, if $D = (Q, \Sigma, \delta_D, q_0, F)$ is a DFA, we define the NFA $N = (Q, \Sigma, \delta_N, q_0, F)$, where

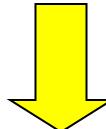
$$\delta_N(q, a) = \begin{cases} \{\delta_D(q, a)\} & \text{if } a \in \Sigma, \\ \emptyset & \text{if } a = \epsilon. \end{cases}$$

It is easy to see that $L(N) = L(D)$, i.e., that N is equivalent to D .

Proof-Step 2

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

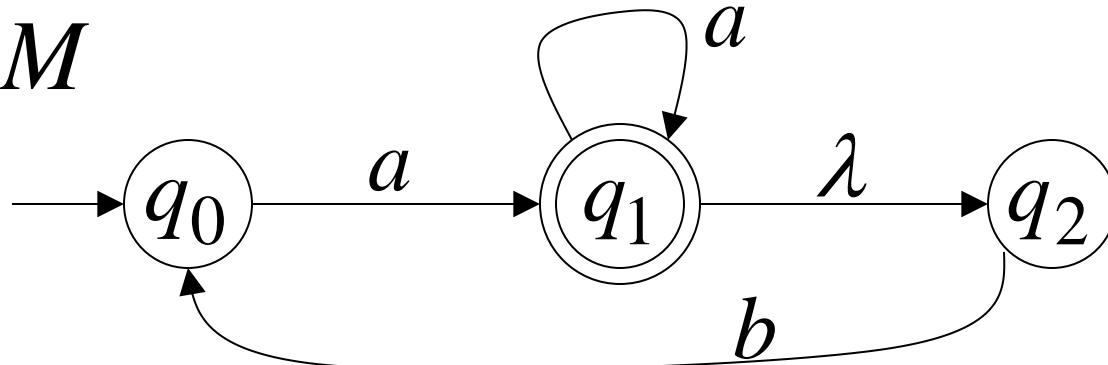
Any NFA can be converted to an equivalent DFA



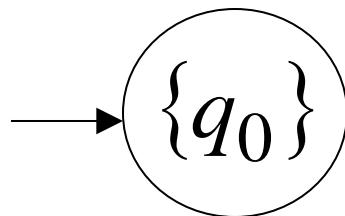
Any language L accepted by an NFA is also accepted by a DFA

Conversion NFA to DFA

NFA M

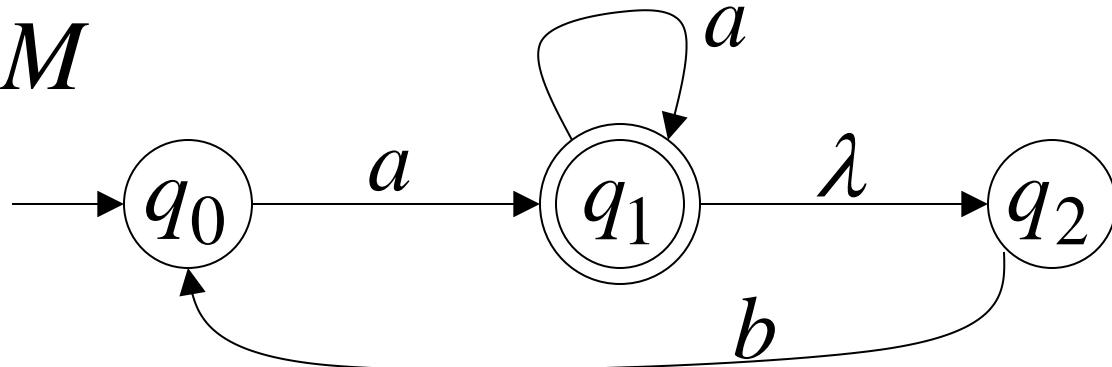


DFA M'

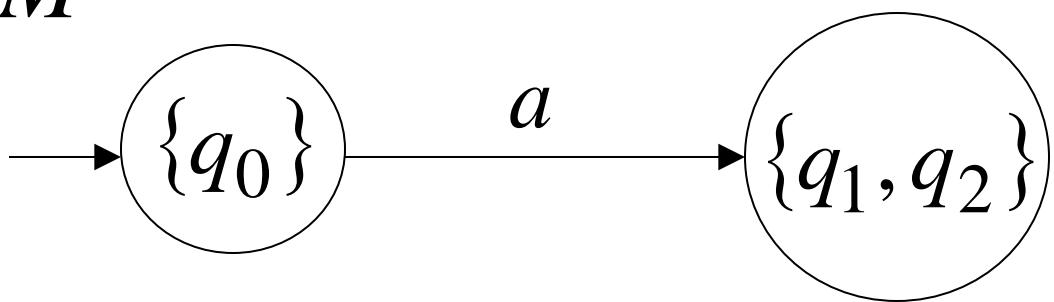


$$\delta^*(q_0, a) = \{q_1, q_2\}$$

NFA M

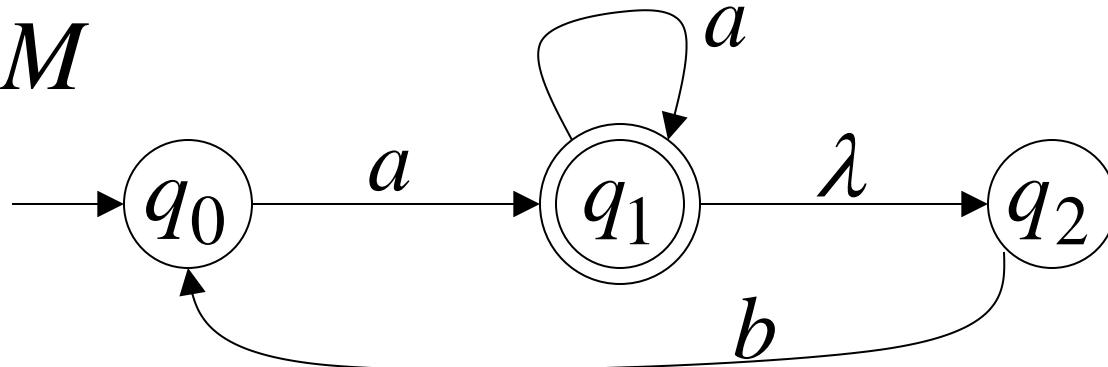


DFA M'

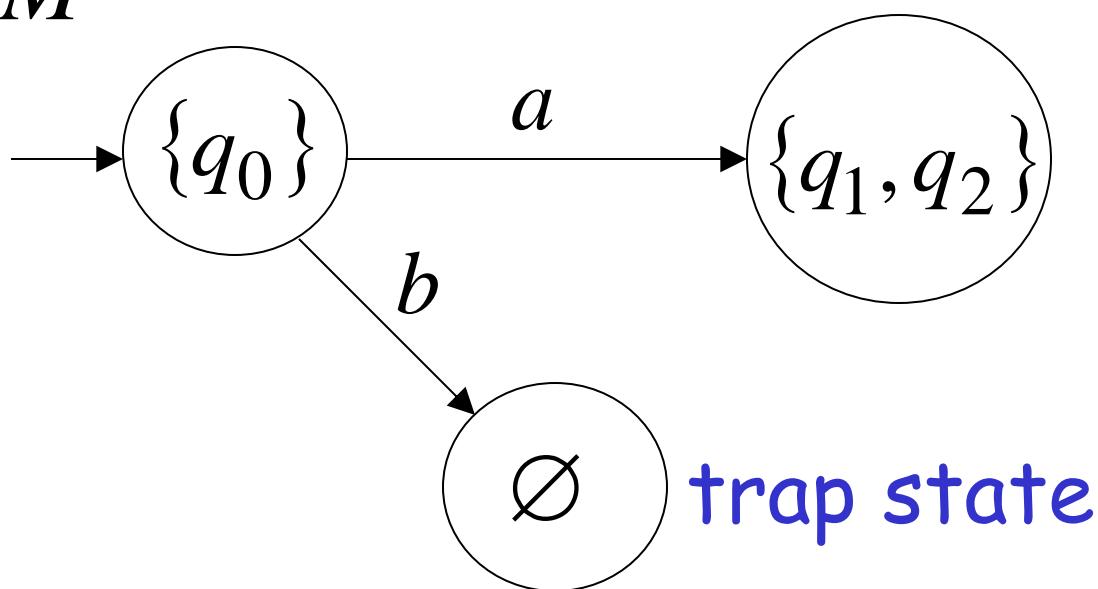


$$\delta^*(q_0, b) = \emptyset \quad \text{empty set}$$

NFA M

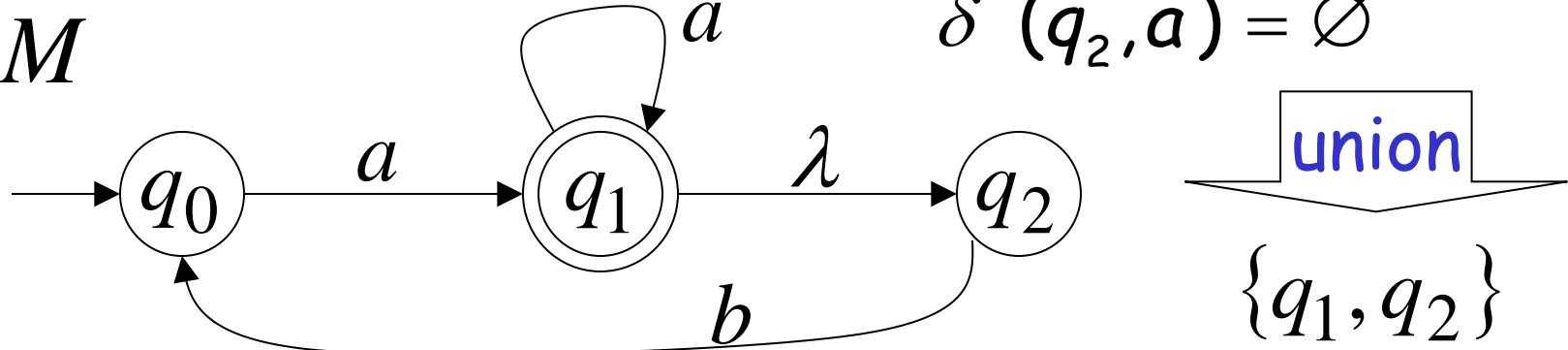


DFA M'

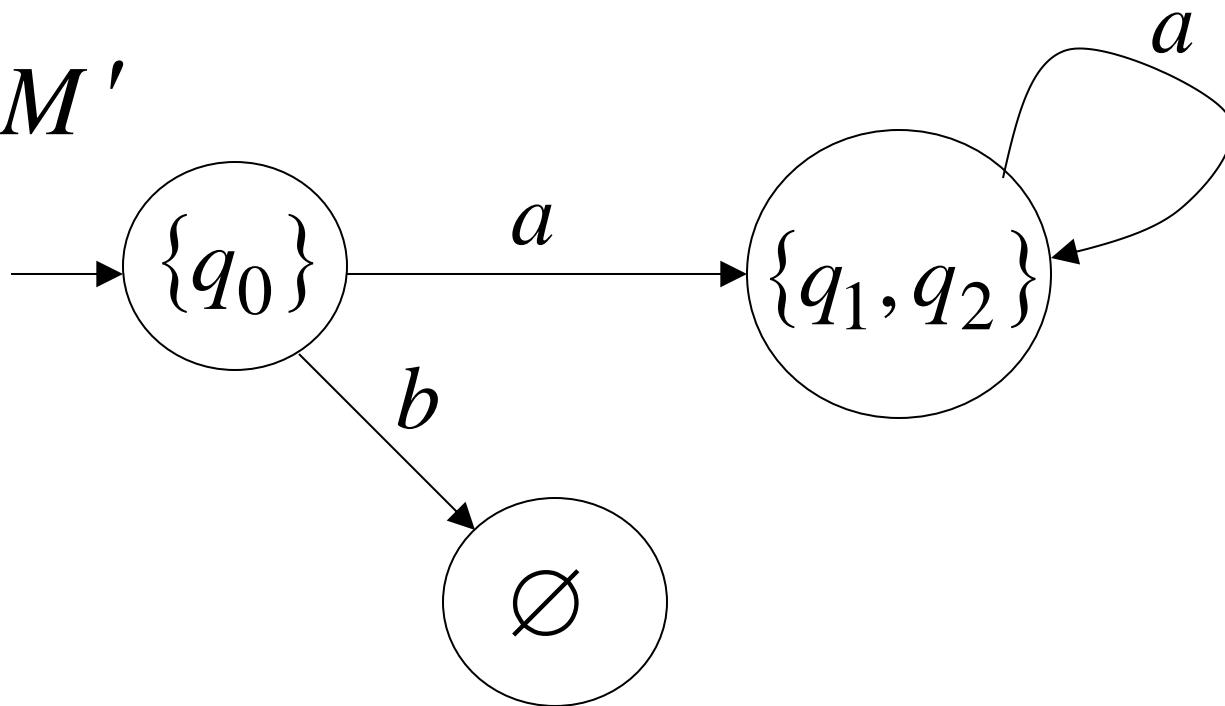


trap state

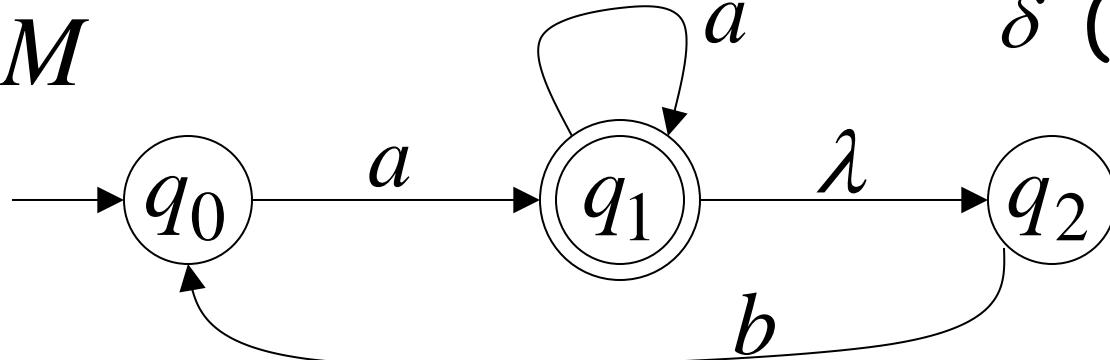
NFA M



DFA M'



NFA M

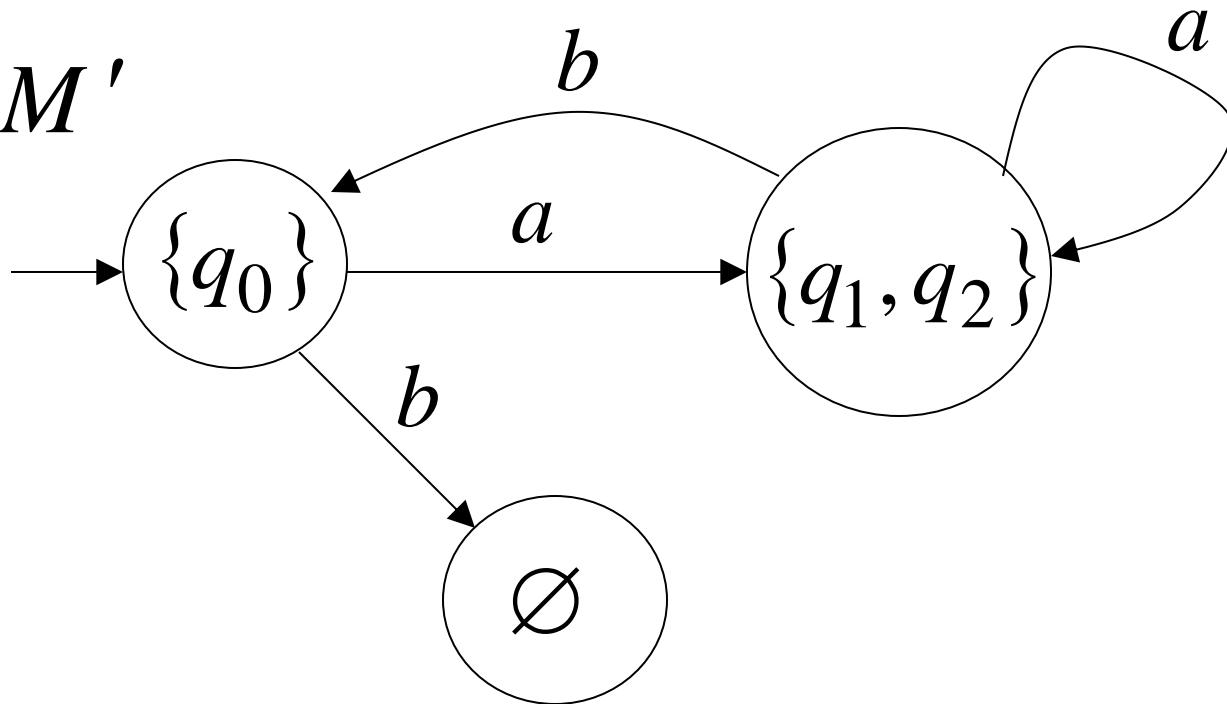


$$\delta^*(q_1, b) = \{q_0\}$$
$$\delta^*(q_2, b) = \{q_0\}$$

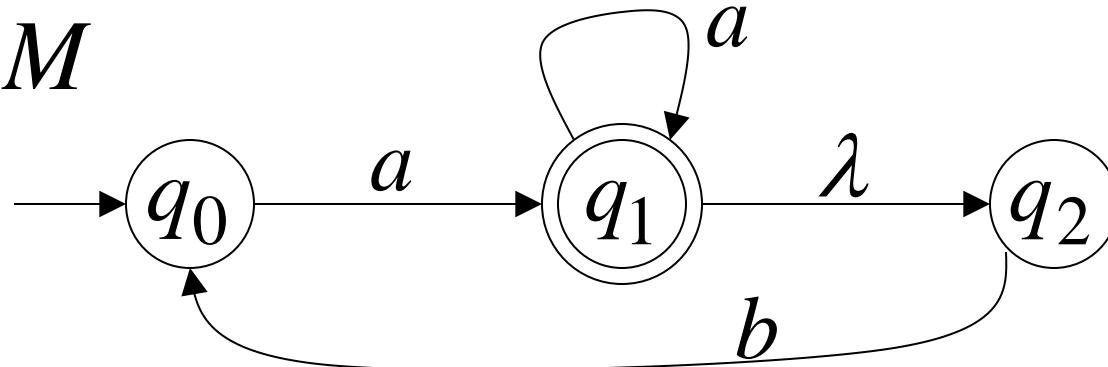
union

$\{q_0\}$

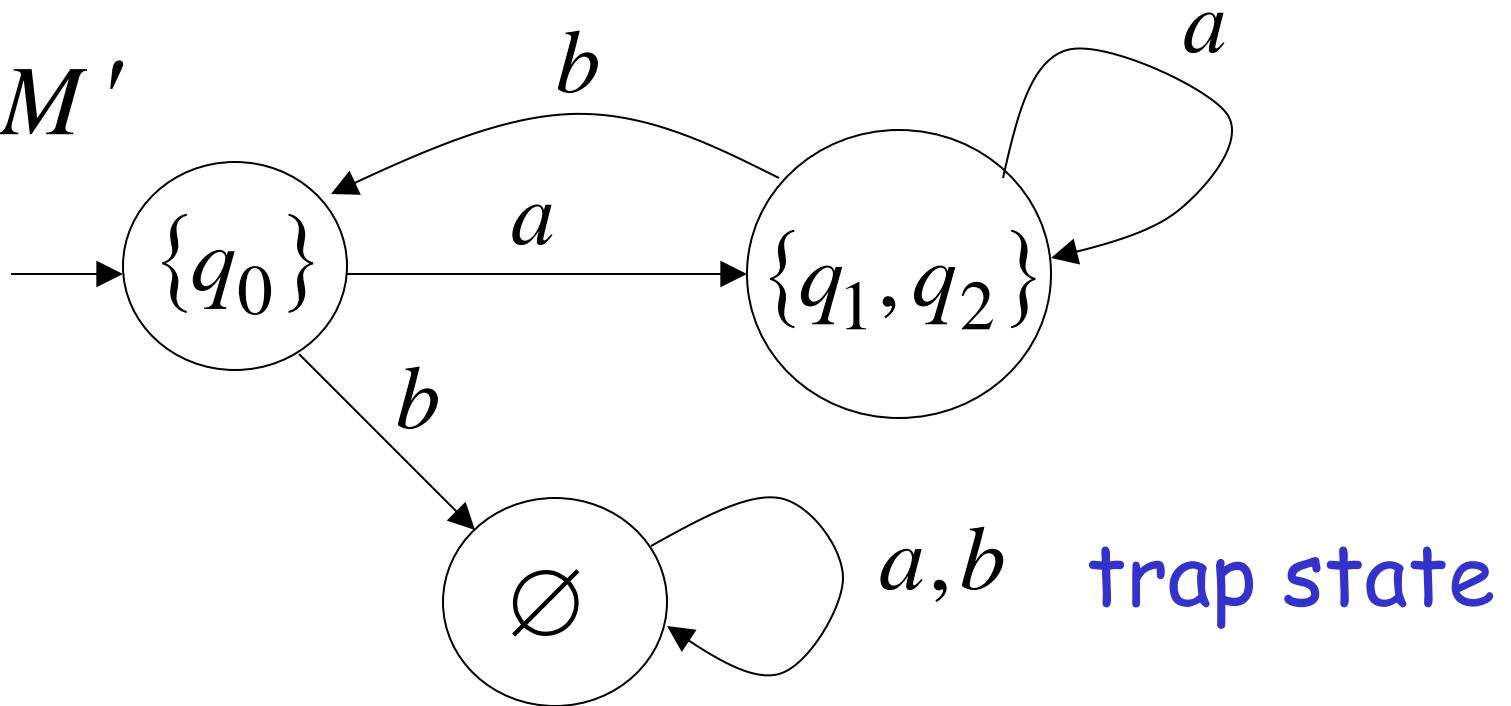
DFA M'



NFA M

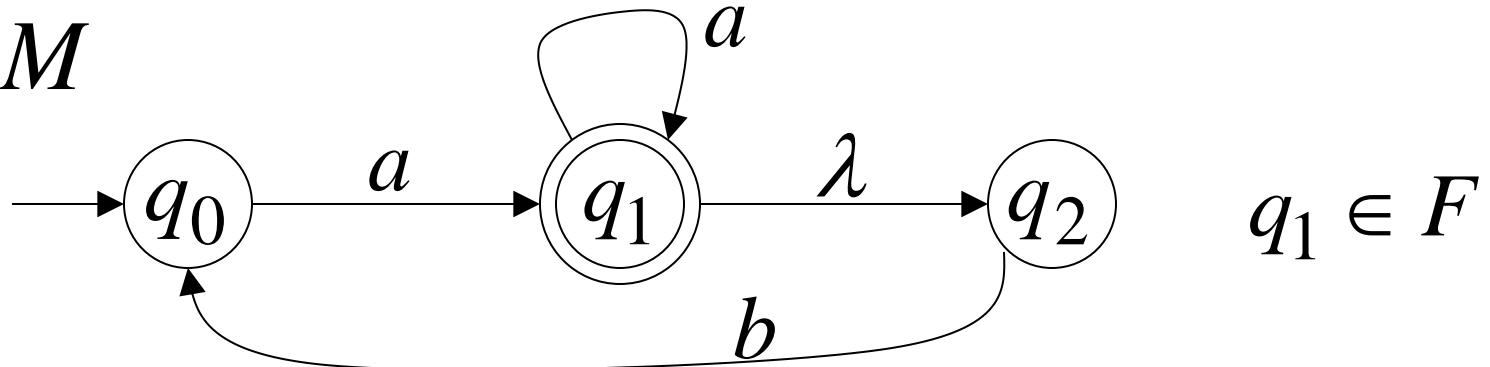


DFA M'



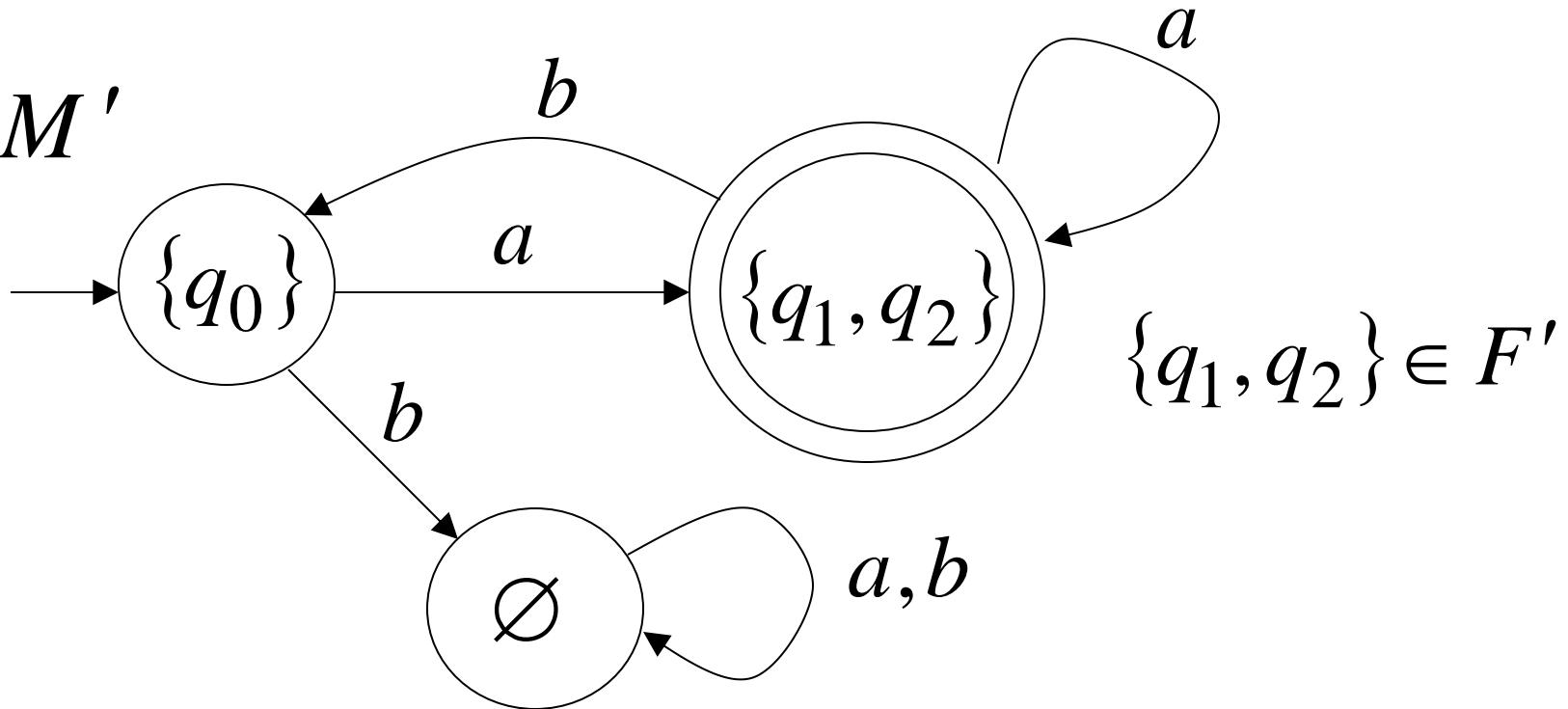
END OF CONSTRUCTION

NFA M



$$q_1 \in F$$

DFA M'



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

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

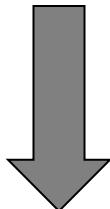
The DFA has states from the power set

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

Conversion Procedure Steps

step

1. Initial state of NFA: q_0

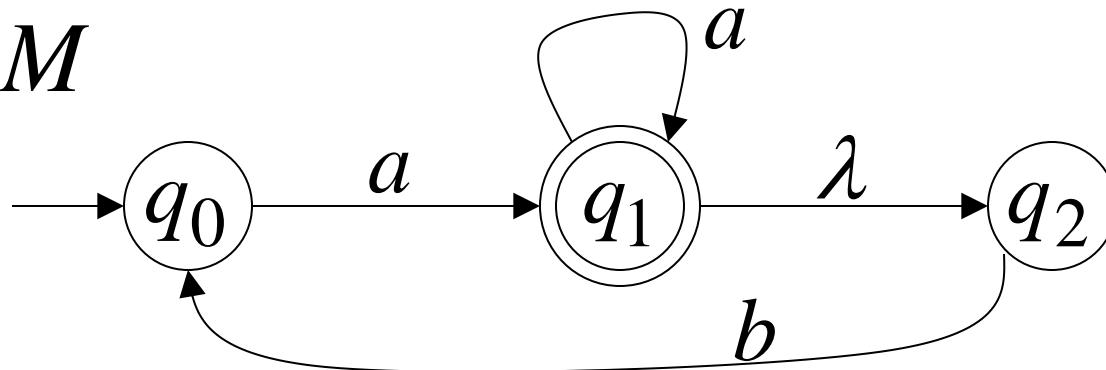


Initial state of DFA (lambda-closure):

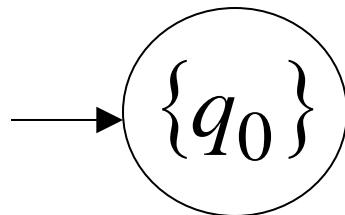
$\{q_0\}$

Example

NFA M



DFA M'



step

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

compute in the NFA

$$\left. \begin{array}{l} \delta^*(q_i, a) \\ \cup \delta^*(q_j, a) \\ \dots \\ \cup \delta^*(q_m, a) \end{array} \right\} = \{q'_k, q'_l, \dots, q'_n\}$$

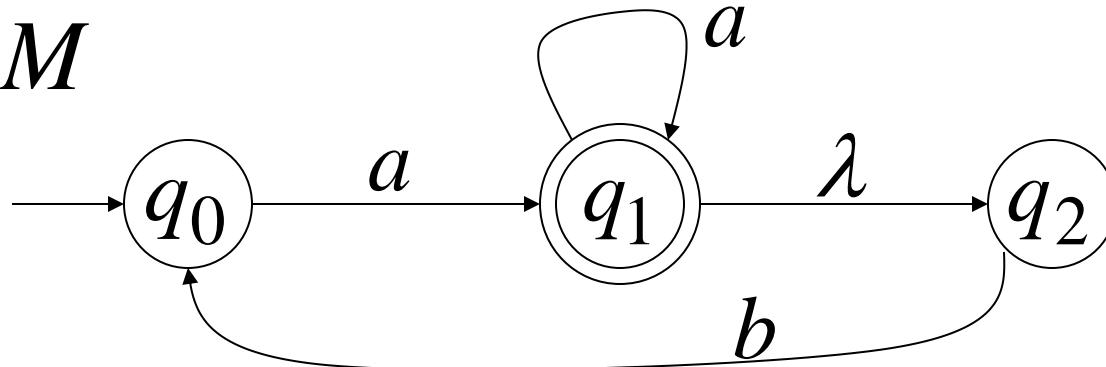
Union

add transition to DFA

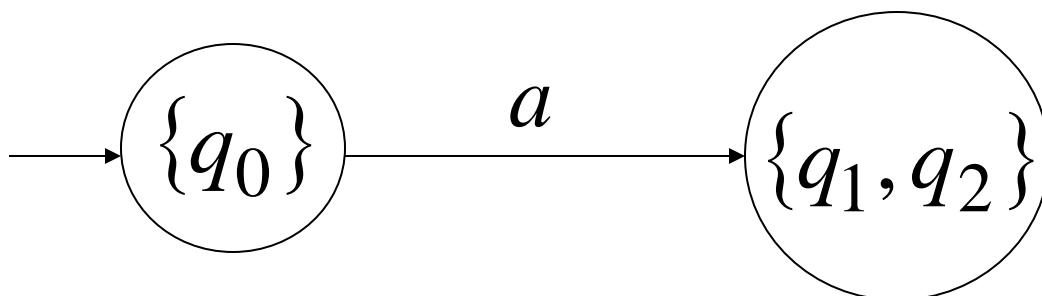
$$\delta(\{q_i, q_j, \dots, q_m\}, a) = \{q'_k, q'_l, \dots, q'_n\}$$

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

NFA M



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

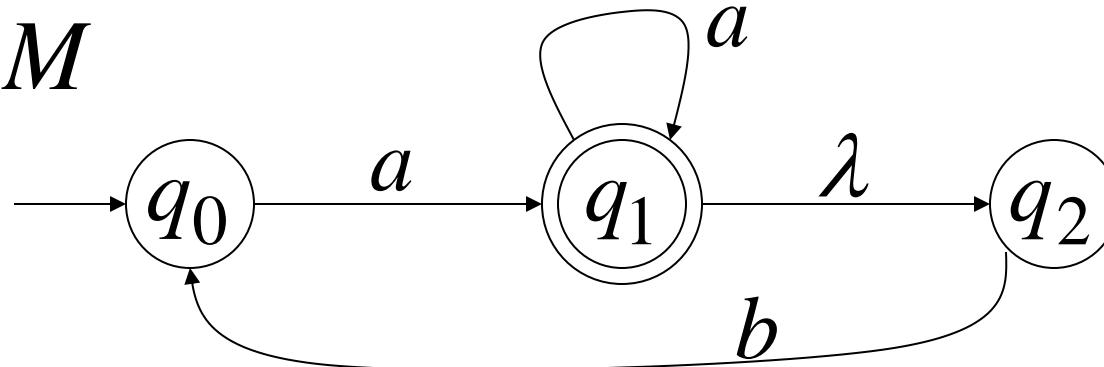


step

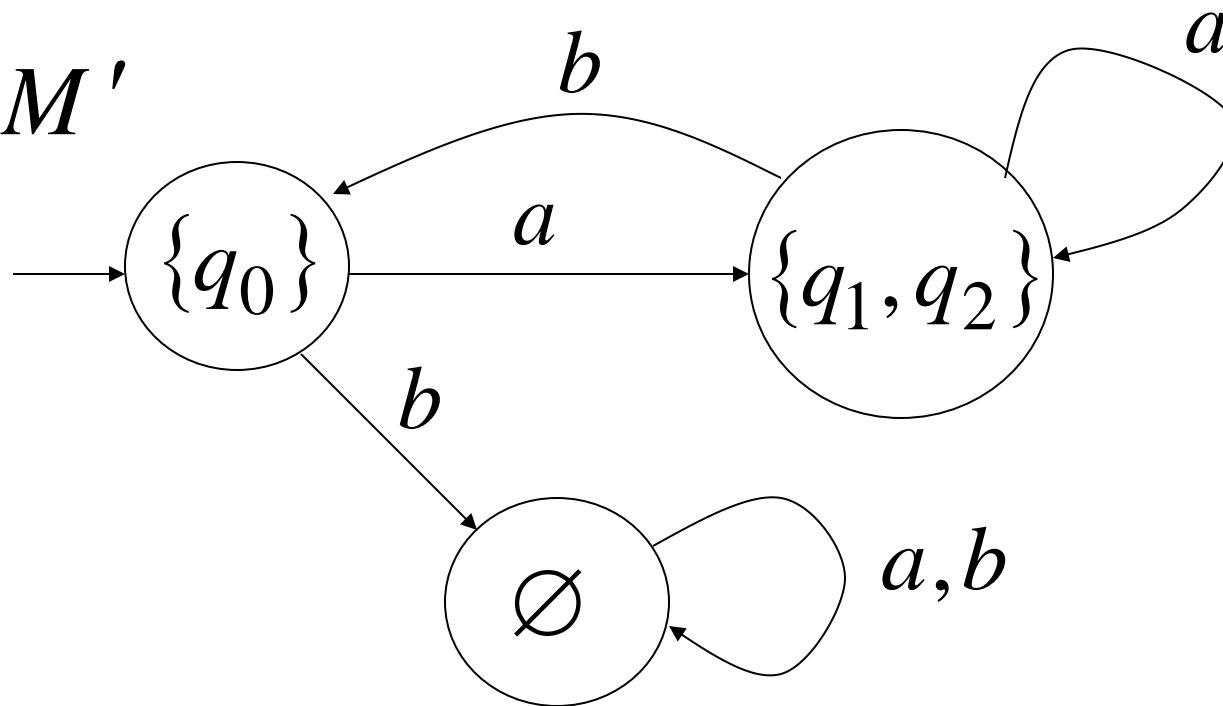
3. Repeat Step 2 for every state in DFA and symbols in alphabet until no more states can be added in the DFA

Example

NFA M



DFA M'



step

4. For any DFA state $\{q_i, q_j, \dots, q_m\}$

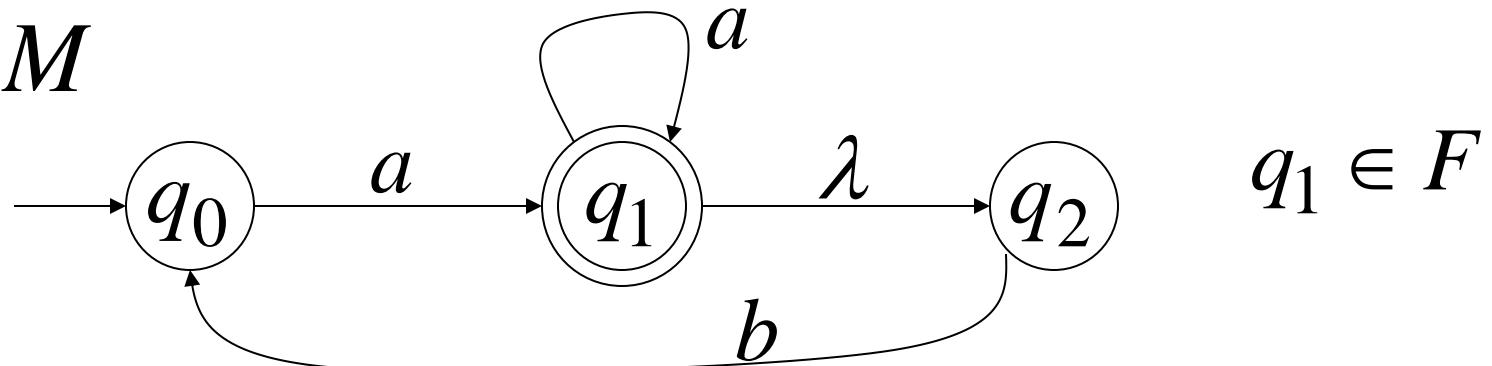
if some q_j is accepting state in NFA

Then, $\{q_i, q_j, \dots, q_m\}$

is accepting state in DFA

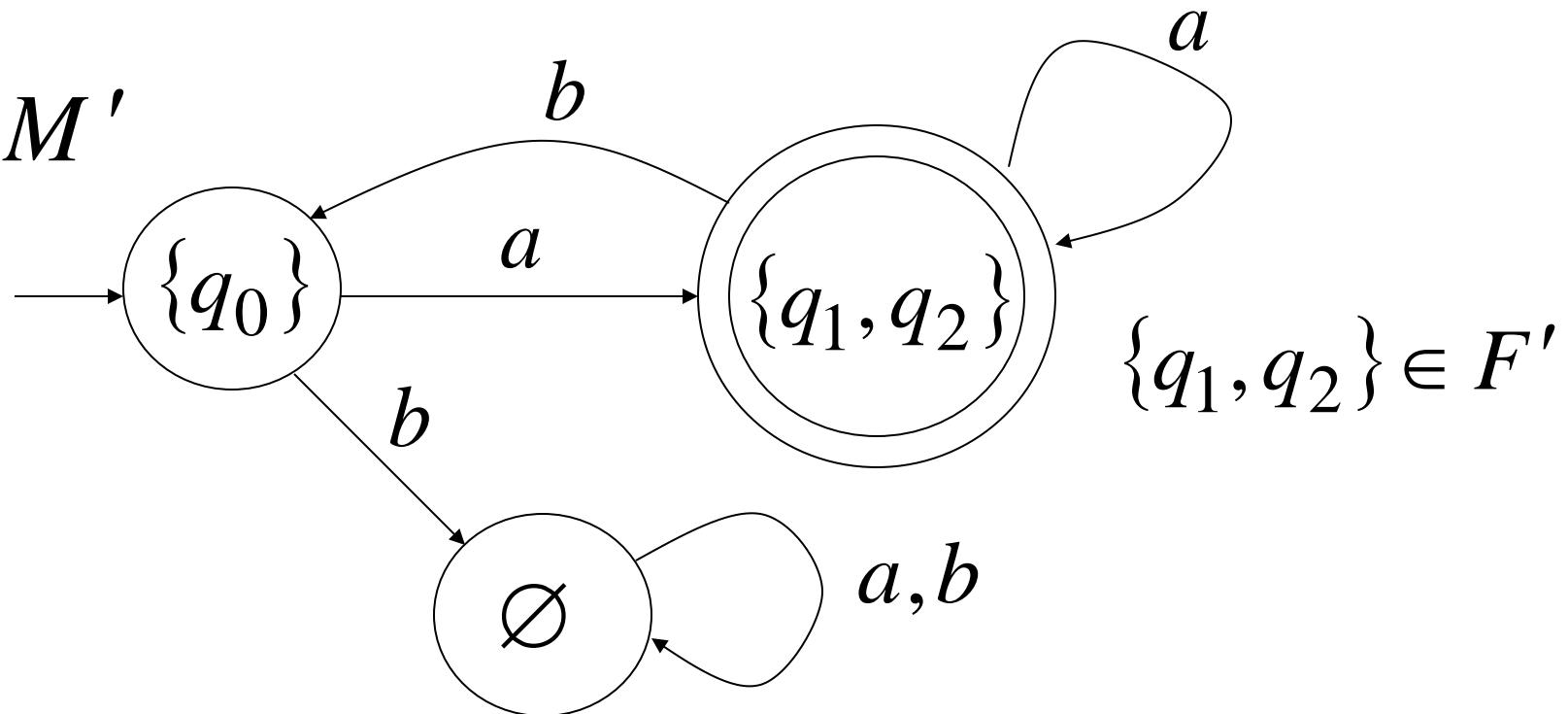
Example

NFA M



$$q_1 \in F$$

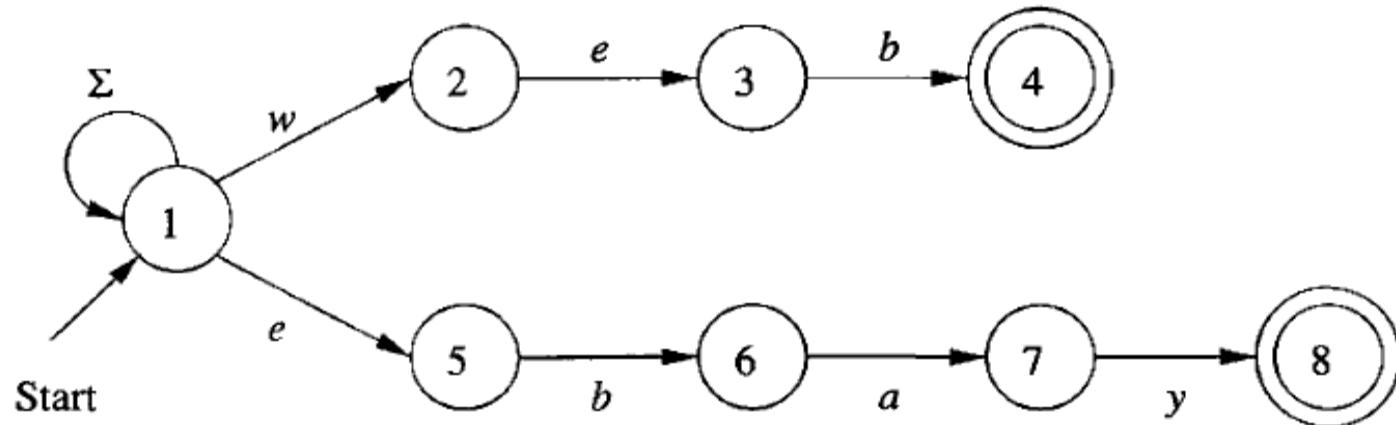
DFA M'



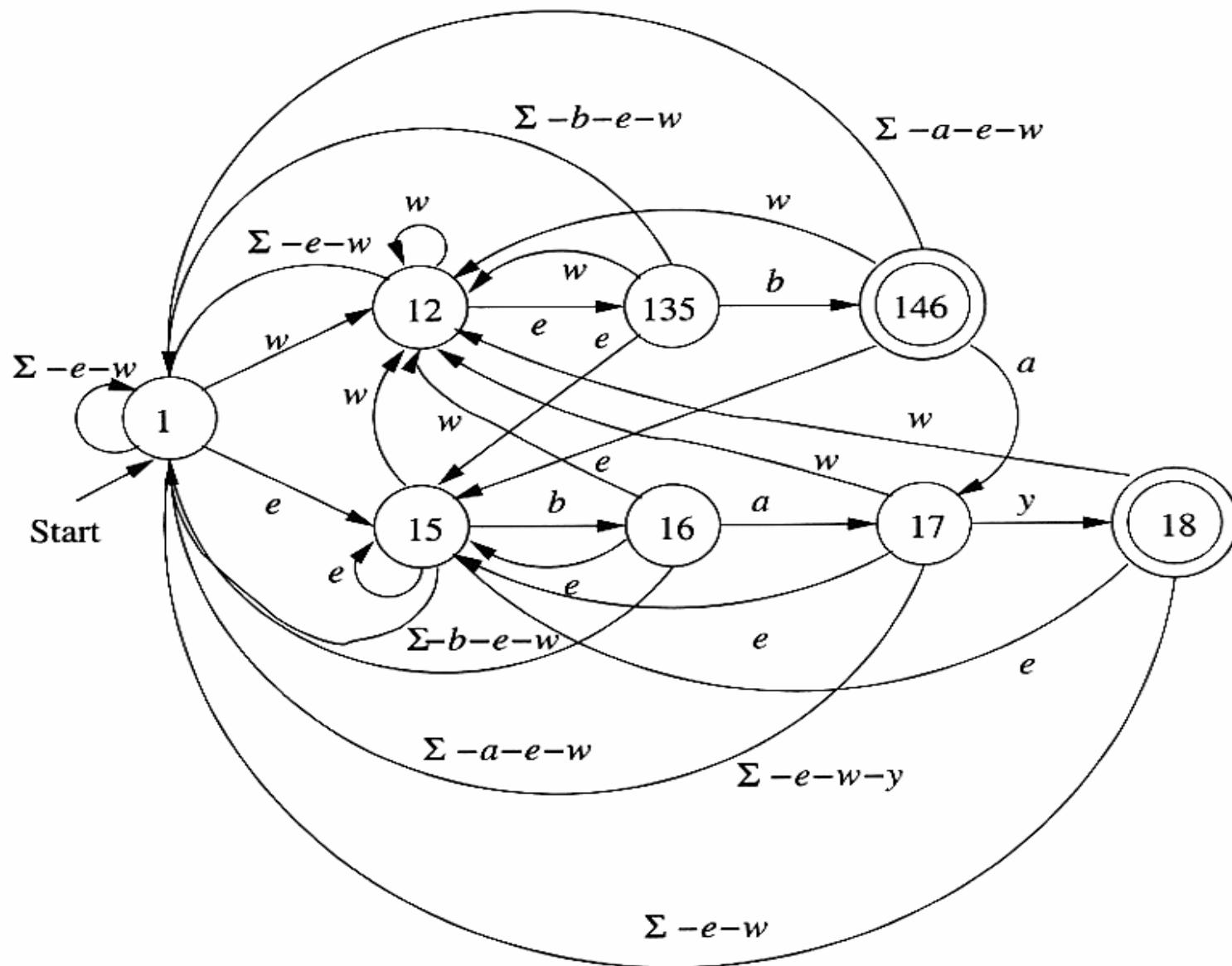
$$\{q_1, q_2\} \in F'$$

Keyword search: example

The following NFA recognizes occurrences of the keywords WEB and EBAY:



Corresponding DFA for keywords search



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

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

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

We only need to prove:

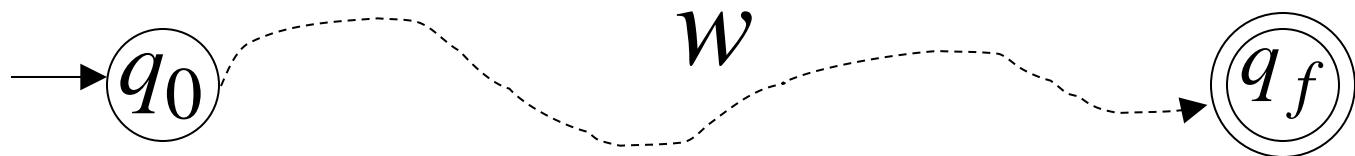
$$w \in L(M)$$



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

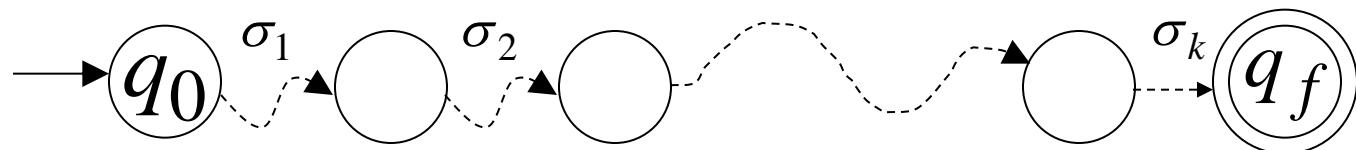
NFA

Consider $w \in L(M)$

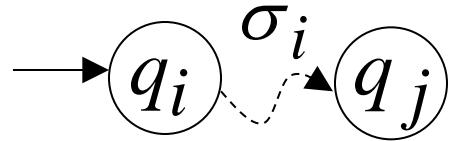


symbols

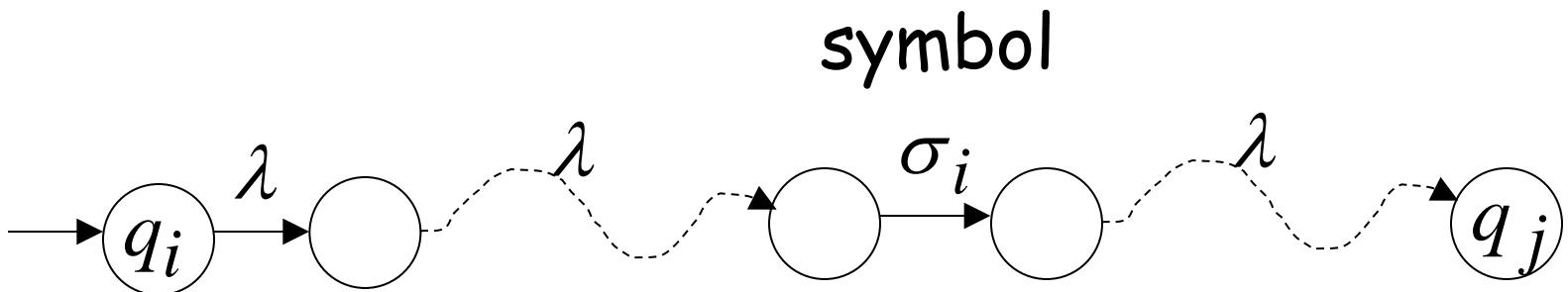
$$w = \sigma_1 \sigma_2 \cdots \sigma_k$$



symbol

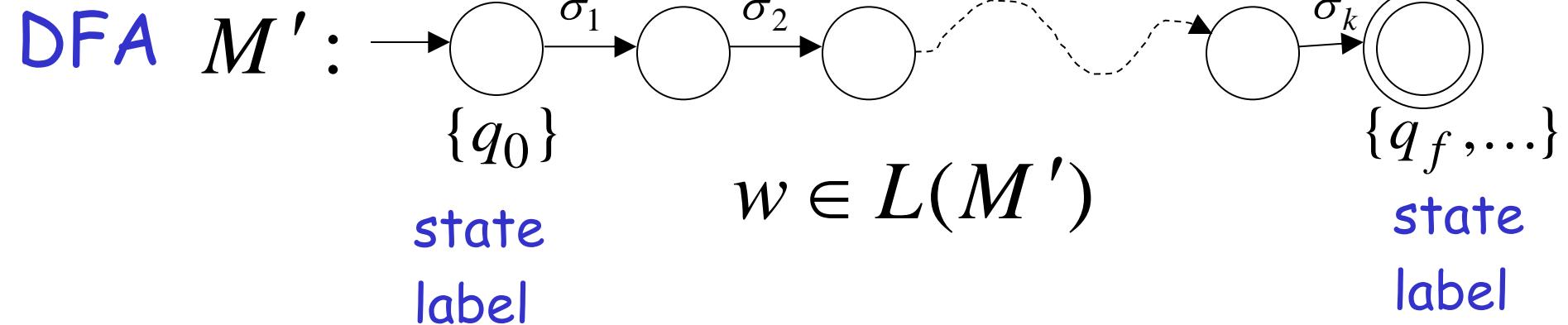
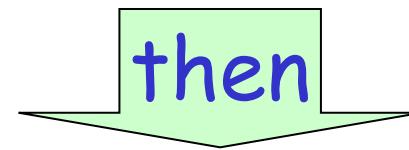
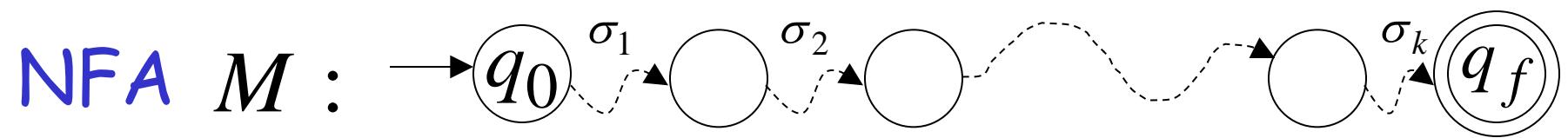


denotes a possible sub-path like



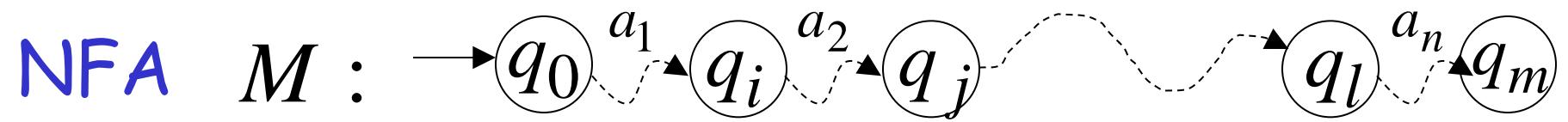
We will show that if $w \in L(M)$

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

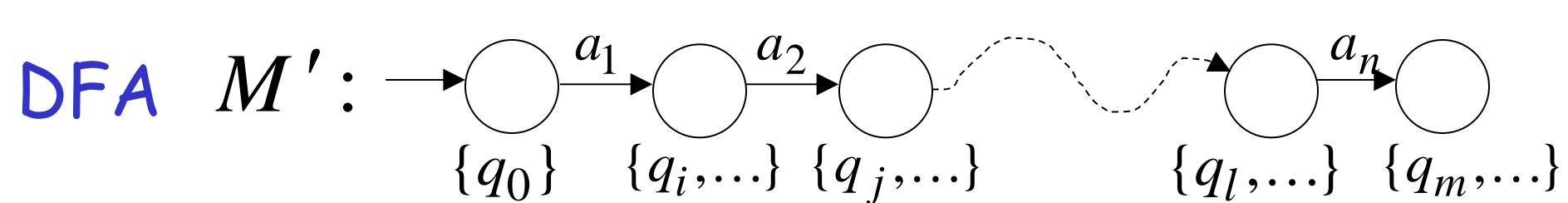


More generally, we will show that if in M :

(arbitrary string) $v = a_1 a_2 \cdots a_n$

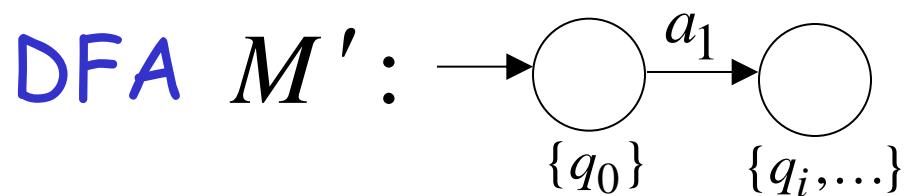
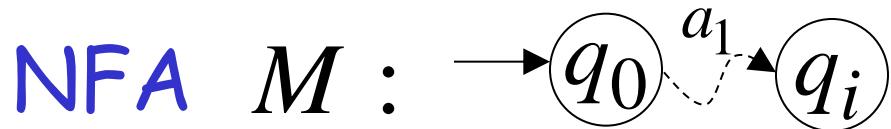


then



Proof by induction on $|v|$

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

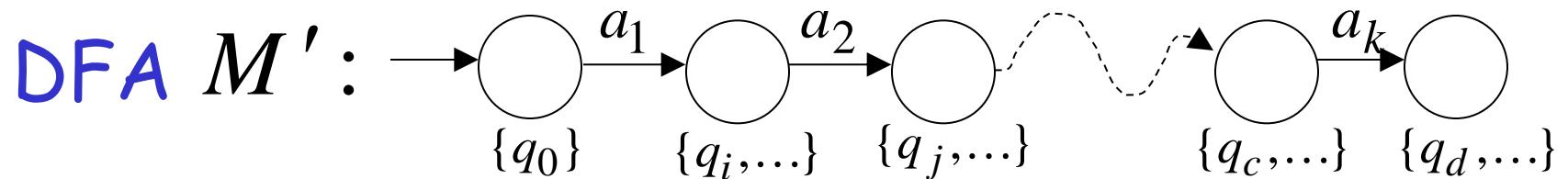
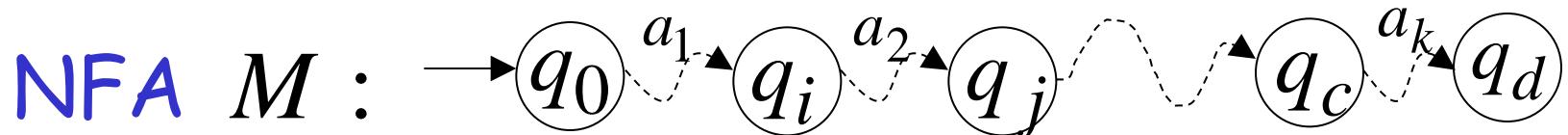


is true by construction of M'

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

$$v = a_1 a_2 \cdots a_k$$

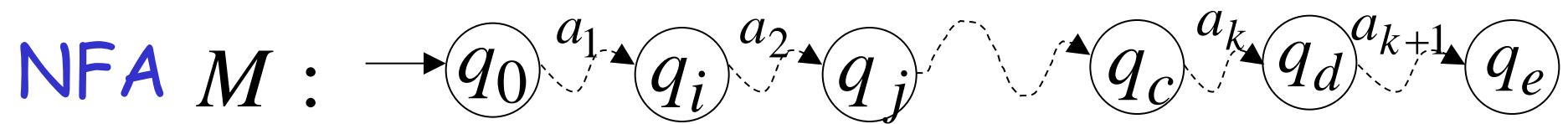
Suppose that the following hold



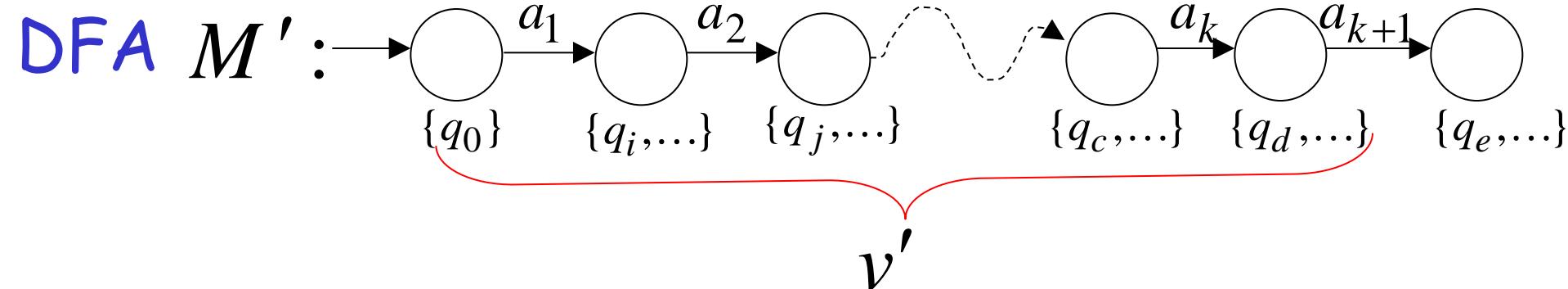
Induction Step: $|v| = k + 1$

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

Then this is true by construction of M'

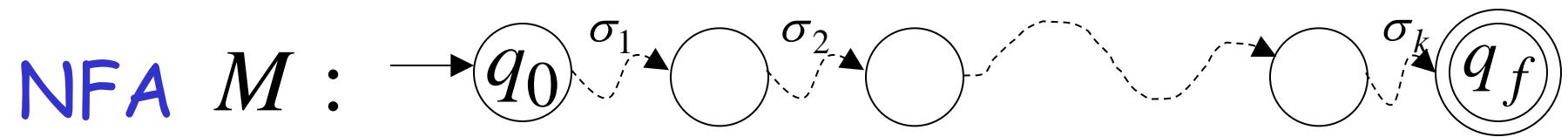


v'

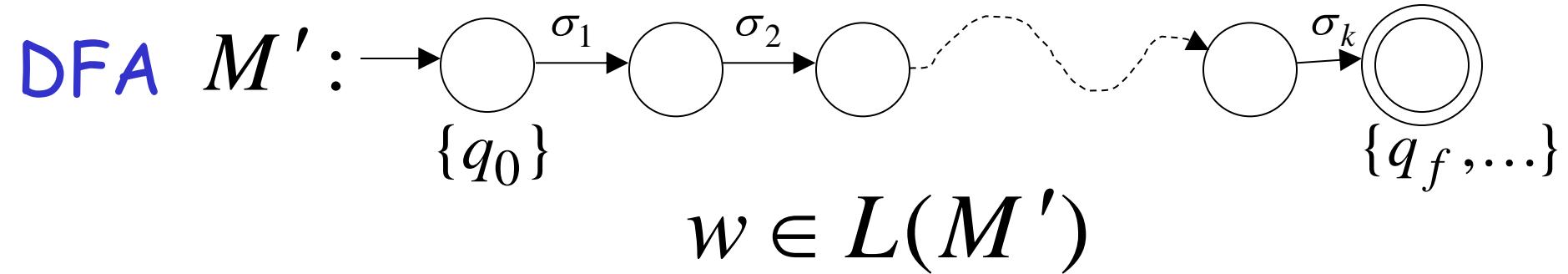


Therefore if $w \in L(M)$

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



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

Clean NFA

The notion of a clean NFA pops up on several occasions.

Definition

Let M be an NFA. We say that M is *clean* iff

- ▶ M has exactly one final state, which is distinct from the start state,
- ▶ M has no transitions into its start state (even self-loops), and
- ▶ M has no transitions out of its final state (even self-loops).

Existence of a Clean NFA

Proposition

For any NFA M , there is an equivalent clean NFA N .

Proof.

If M is not clean, then we can “clean it up” by adding two additional states:

- ▶ a new start state with a single ε -transition to M 's original start state (which is no longer the start state), and
- ▶ a new final state with ε -transitions from all of M 's original final states (which are no longer final states) to the new final state.

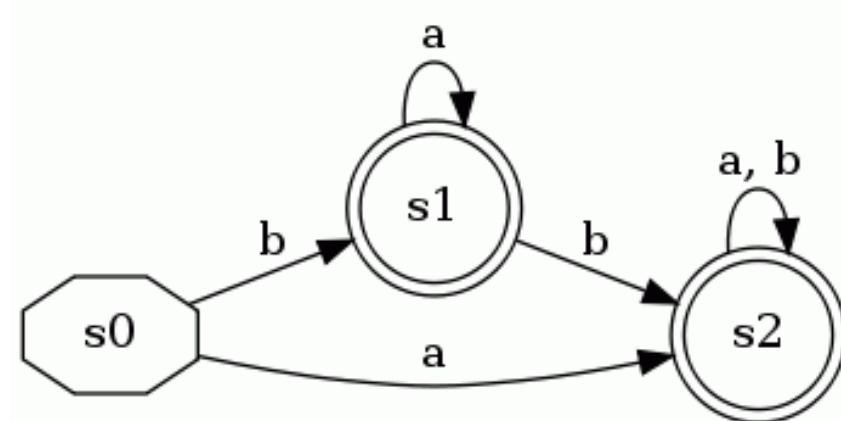
The new NFA is obviously clean, and a simple, informal argument shows that it is equivalent to the original M . □

Q: Give clean NFAs for concatenation and union (series and parallel connections, respectively) as well as for the Kleene closure.



DFA Minimization

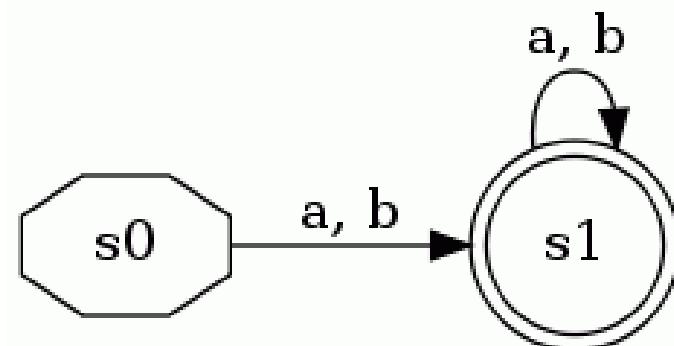
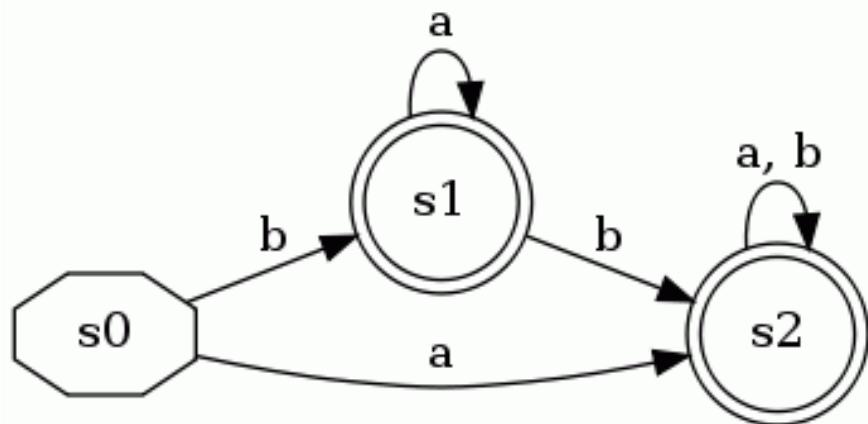
- Some states can be redundant:
 - The following DFA accepts $(a|b)^+$
 - State $s1$ is not necessary





DFA Minimization

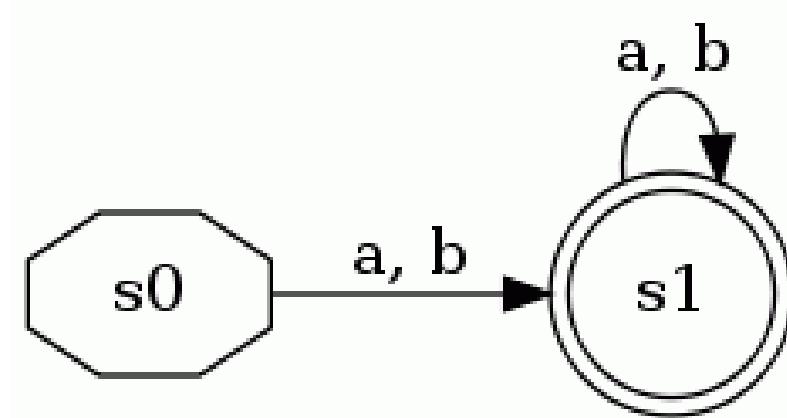
- So these two DFAs are *equivalent*:

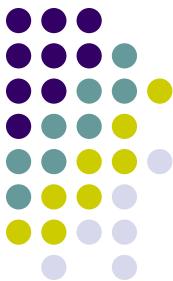




DFA Minimization

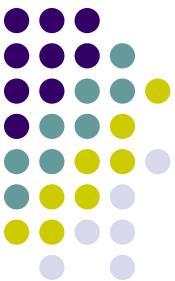
- This is a *state-minimized* (or just *minimized*) DFA
 - Every remaining state is necessary





DFA Minimization

- The task of *DFA minimization*, then, is to automatically transform a given DFA into a state-minimized DFA
 - Several algorithms and variants are known
 - Note that this also in effect can minimize an NFA (since we know algorithm to convert NFA to DFA)



DFA Minimization Algorithm

- Recall that a DFA $M=(Q, \Sigma, \delta, q_0, F)$
- Two states p and q are distinct if
 - $p \in F$ and $q \notin F$ or vice versa, or
 - for some $\alpha \in \Sigma$, $\delta(p, \alpha)$ and $\delta(q, \alpha)$ are distinct
- Using this inductive definition, we can calculate which states are distinct



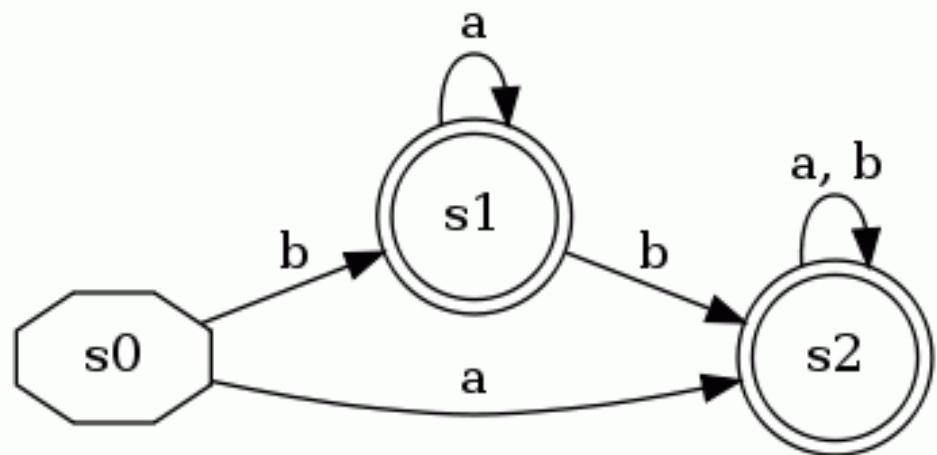
DFA Minimization Algorithm

- Create lower-triangular table DISTINCT, initially blank
- For every pair of states (p, q) :
 - If p is final and q is not, or vice versa
 - $\text{DISTINCT}(p, q) = \epsilon$
- Loop until no change for an iteration:
 - For every pair of states (p, q) and each symbol α
 - If $\text{DISTINCT}(p, q)$ is blank and $\text{DISTINCT}(\delta(p, \alpha), \delta(q, \alpha))$ is not blank
 - $\text{DISTINCT}(p, q) = \alpha$
- Combine all states that are not distinct



Very Simple Example

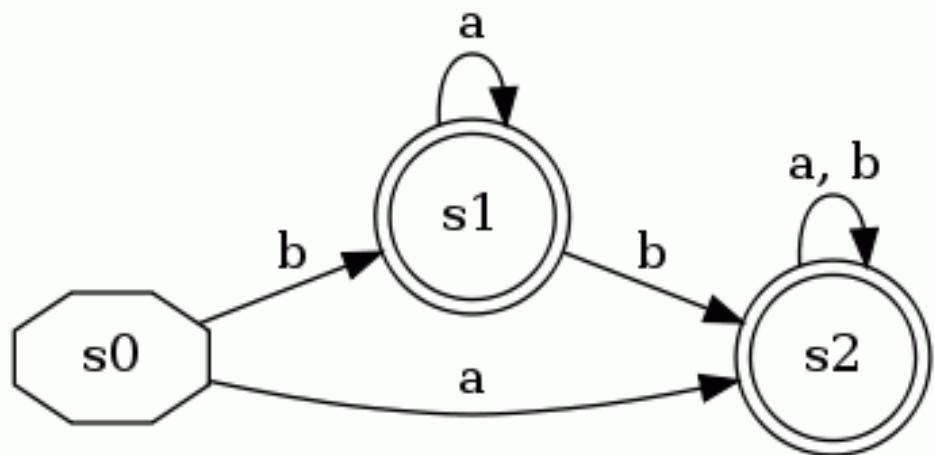
s0			
s1			
s2			
	s0	s1	s2





Very Simple Example

s_0			
s_1	ϵ		
s_2	ϵ		
	s_0	s_1	s_2

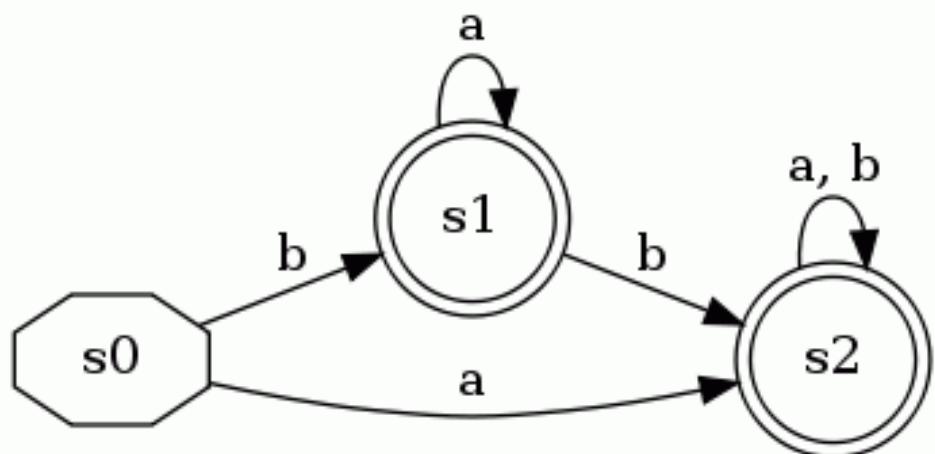


Label pairs with ϵ where one is a final state and the other is not



Very Simple Example

s_0			
s_1	ϵ		
s_2	ϵ		
	s_0	s_1	s_2

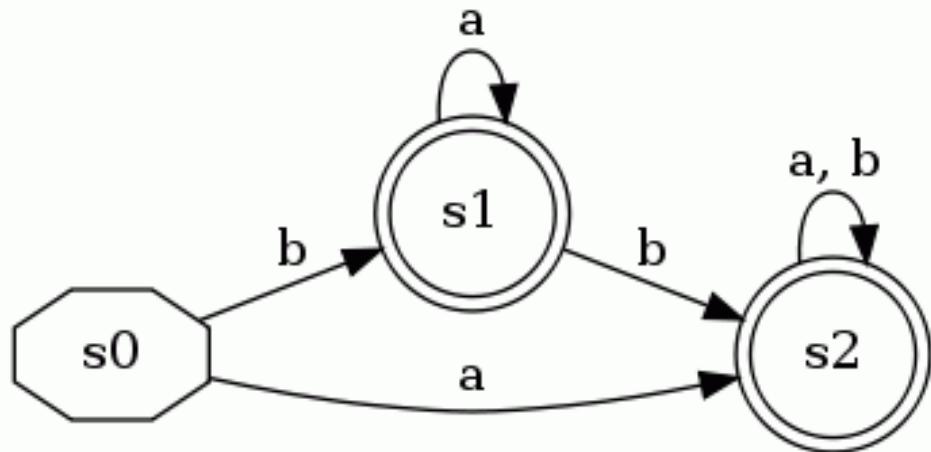


Main loop (no changes occur)



Very Simple Example

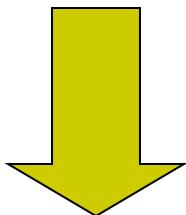
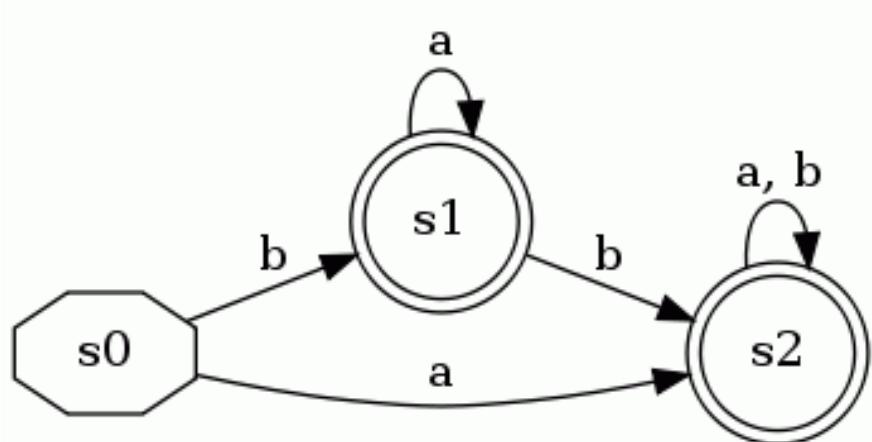
s0			
s1	ϵ		
s2	ϵ		
	s0	s1	s2



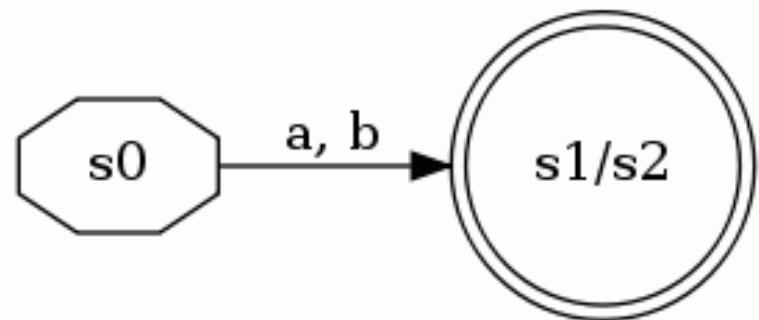
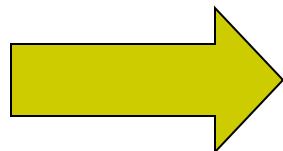
$\text{DISTINCT}(s1, s2)$ is empty, so $s1$ and $s2$ are equivalent states



Very Simple Example

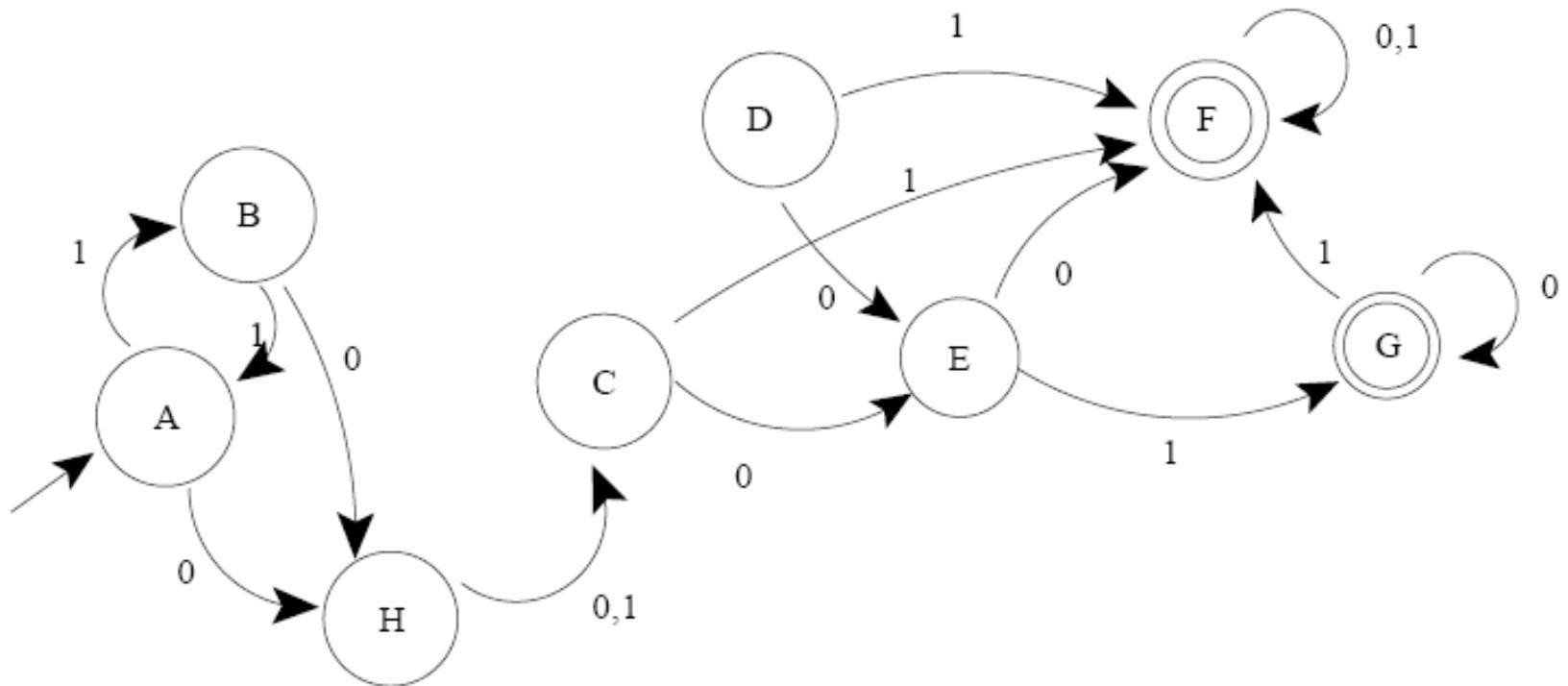


Merge s_1 and s_2





More Complex Example





More Complex Example

- Check for pairs with one state final and one not:

b							
c							
d							
e							
f	ϵ	ϵ	ϵ	ϵ	ϵ		
g	ϵ	ϵ	ϵ	ϵ	ϵ		
h						ϵ	ϵ
	a	b	c	d	e	f	g



More Complex Example

- First iteration of main loop:

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



More Complex Example

- Second iteration of main loop:

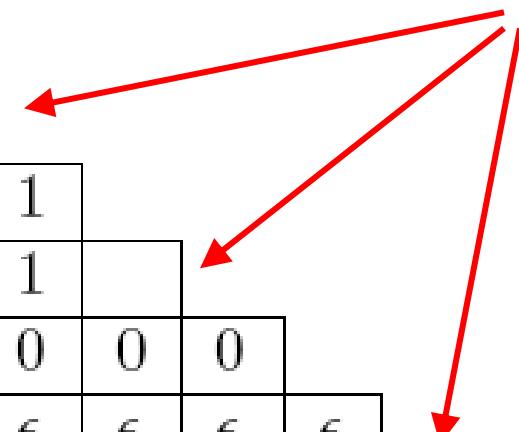
b							
c	1	1					
d	1	1					
e	0	0	0	0			
f	ϵ	ϵ	ϵ	ϵ	ϵ		
g	ϵ	ϵ	ϵ	ϵ	ϵ		
h	1	1	1	1	0	ϵ	ϵ
	a	b	c	d	e	f	g



More Complex Example

- Third iteration makes no changes
 - Blank cells are equivalent pairs of states

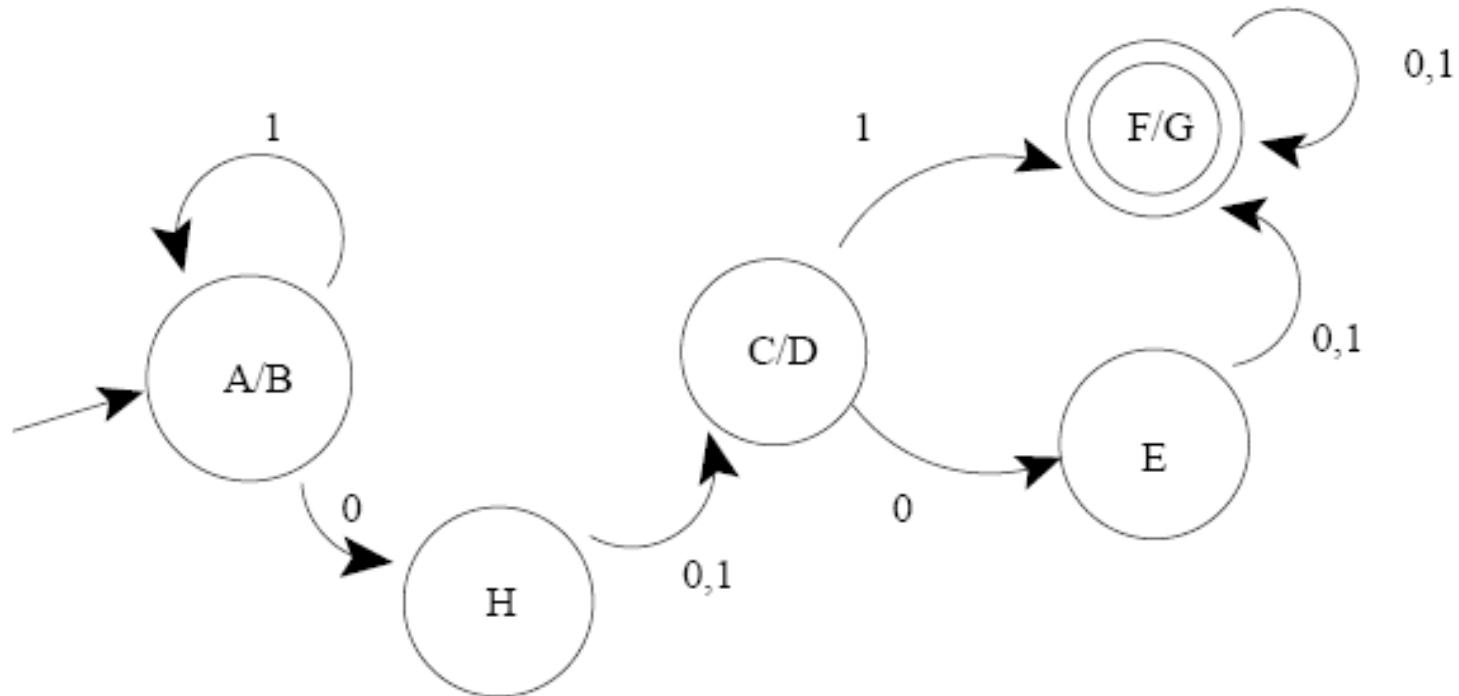
b							
c	1	1					
d	1	1					
e	0	0	0	0			
f	ϵ	ϵ	ϵ	ϵ	ϵ		
g	ϵ	ϵ	ϵ	ϵ	ϵ		
h	1	1	1	1	0	ϵ	ϵ
	a	b	c	d	e	f	g

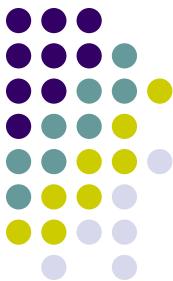




More Complex Example

- Combine equivalent states for minimized DFA:





Conclusion

- DFA Minimization is a fairly understandable process, and is useful in several areas
 - Regular expression matching implementation
 - Very similar algorithm is used for compiler optimization to eliminate duplicate computations
- The algorithm described is $O(kn^2)$
 - John Hopcraft describes another more complex algorithm that is $O(k(n \log n))$

Regular Expressions

Regular Expression (RE)

Regular expression: An algebraic way to describe regular languages.

Many of today's programming languages use regular expressions to match patterns in strings.

E.g., awk, flex, lex, java, javascript, perl, python

Used for searching texts in UNIX (vi, Perl, Emacs, grep), Microsoft Word (version 6 and beyond), and WordPerfect.

Few Web search engines may allow the use of Regular Expressions

Recursive Definition

Primitive regular expressions: $\emptyset, \lambda, \alpha$

Given regular expressions r_1 and r_2

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

Are regular expressions

Examples

A regular expression: $(a + b \cdot c)^* \cdot (c + \emptyset)$

Not a regular expression: $(a + b +)$

Languages of Regular Expressions

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

Example

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

Definition

For primitive regular expressions:

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

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

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

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

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

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

Regular Expressions

Operator Precedence:

Highest: Kleene Closure

Then: Concatenation

Lowest: Union

Example

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

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

Example

Regular expression $r = (aa)^*(bb)^*b$

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

Example

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

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

Example

Regular expression $r = (1 + 01)^* (0 + \lambda)$

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

Regular Expressions

EXAMPLE 2.1 The expression $0(0 + 1)^*1$ represents the set of all strings that begin with a 0 and end with a 1.

EXAMPLE 2.2 The expression $0 + 1 + 0(0 + 1)^*0 + 1(0 + 1)^*1$ represents the set of all nonempty binary strings that begin and end with the same bit. Note the inclusion of the strings 0 and 1 as special cases.

EXAMPLE 2.3 The expressions 0^* , 0^*10^* , and $0^*10^*10^*$ represent the languages consisting of strings that contain no 1, exactly one 1, and exactly two 1's, respectively.

EXAMPLE 2.4 The expressions $(0 + 1)^*1(0 + 1)^*1(0 + 1)^*$, $(0 + 1)^*10^*1(0 + 1)^*$, $0^*10^*1(0 + 1)^*$, and $(0 + 1)^*10^*10^*$ all represent the same set of strings that contain at least two 1's.

Equivalent Regular Expressions

Definition:

Regular expressions r_1 and r_2

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

Example

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

$$r_1 = (1 + 01)^* (0 + \lambda)$$

$$r_2 = (1^* 01 1^*)^* (0 + \lambda) + 1^* (0 + \lambda)$$

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



r_1 and r_2
are equivalent
regular expressions

Regular Expression: The IEEE POSIX standard

Character	Meaning	Examples
[]	alternatives	/[aeiou]/, /m[ae]n/
-	range	/[a-z]/
[^]	not	/[^pbm]/, /[^ox]s/
?	optionality	/Kath?mandu/
*	zero or more	/baa*/
+	one or more	/ba+/
.	any character	/cat.[aeiou]/
^, \$	start, end of line	
\	not special character	\.\?\^\^
	alternate strings	/cat dog/
()	substring	/cit(y ies)/
etc.		

Regular Expressions

Valid Email Addresses

Valid IP Addresses

Valid Dates

Floating Point Numbers

Variables

Integers

Numeric Values

Naming Regular Expressions

Can assign names to regular expressions

Can use the name of a RE in the definition of another RE

Examples:

```
letter      ::= a | b | ... | z
digit       ::= 0 | 1 | ... | 9
alphanum   ::= letter | digit
```

Grammar-like notation for named RE's: a regular grammar

Can reduce named RE's to plain RE by "macro expansion"

- no recursive definitions allowed,
unlike full context-free grammars

Specifying Tokens

Identifiers

```
ident      ::= letter (letter | digit)*
```

Integer constants

```
integer    ::= digit^+
```

```
sign       ::= + | -
```

```
signed_int ::= [sign] integer
```

Real number constants

```
real      ::= signed_int  
           [fraction] [exponent]
```

```
fraction  ::= . digit^+
```

```
exponent ::= (E|e) signed_int
```

RE specification of initial MiniJava lexical structure

```
Program      ::= (Token | Whitespace)*

Token        ::= ID | Integer | ReservedWord |
                  Operator | Delimiter

ID           ::= Letter (Letter | Digit)*

Letter       ::= a | ... | z | A | ... | Z

Digit        ::= 0 | ... | 9

Integer       ::= Digit+

ReservedWord ::= class | public | static |
                  extends | void | int |
                  boolean | if | else |
                  while | return | true | false |
                  this | new | string | main |
                  System.out.println

Operator      ::= + | - | * | / | < | <= | >= | |
                  > | == | != | && | !

Delimiter    ::= ; | . | , | = | |
                  ( | ) | { | } | [ | ] |

Whitespace   ::= <space> | <tab> | <newline>
```

Regular Expressions and Regular Languages

Theorem

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

Theorem (Kleene 1956):

We say that a language $L \subseteq \Sigma^*$ is regular if there exists a regular expression r such that $L = L(r)$. In this case, we also say that r represents the language L .

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

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

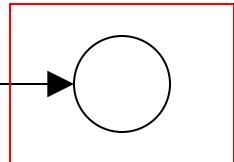
For any regular expression r
the language $L(r)$ is regular

Proof by induction on the size of r

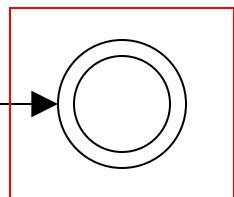
Induction Basis

Primitive Regular Expressions: $\emptyset, \lambda, \alpha$

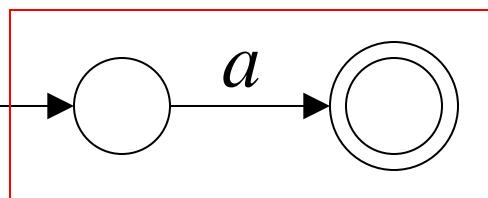
Corresponding
NFAs



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



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



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

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:

$$\left. \begin{array}{l} L(r_1 + r_2) \\ L(r_1 \cdot r_2) \\ L(r_1^*) \\ L((r_1)) \end{array} \right\}$$

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)

End of Proof-Part 1

Are regular languages

Proof - Part 2

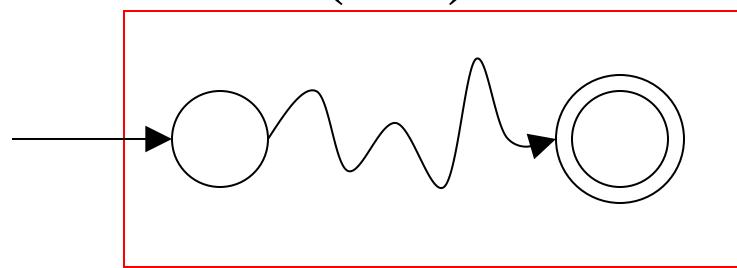
$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Expressions} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

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

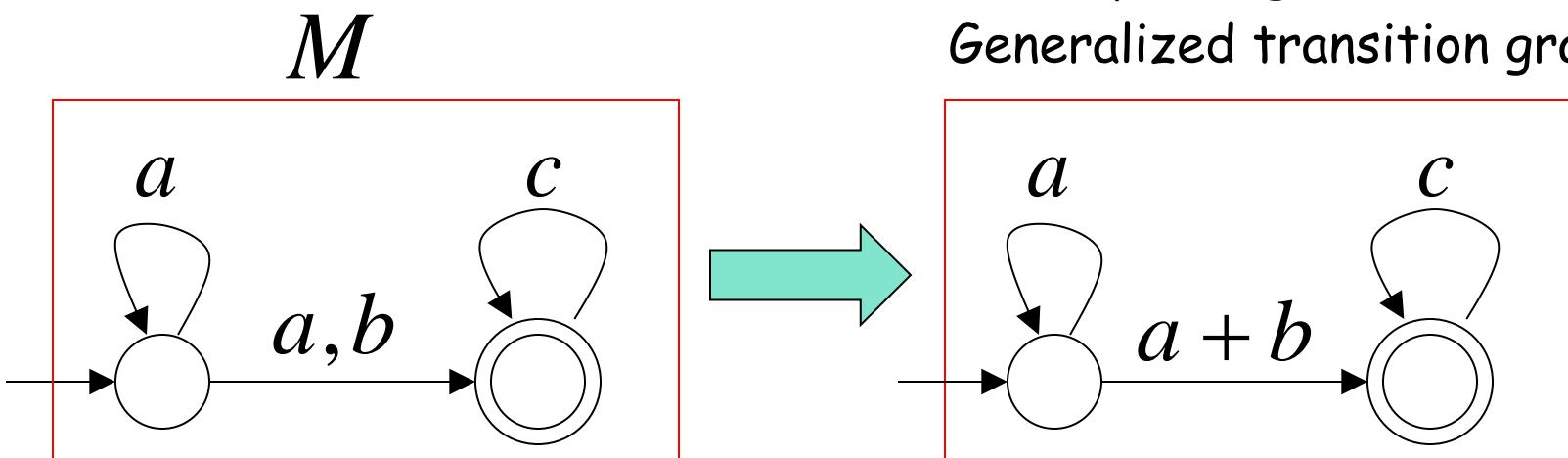
$$L(M) = L$$



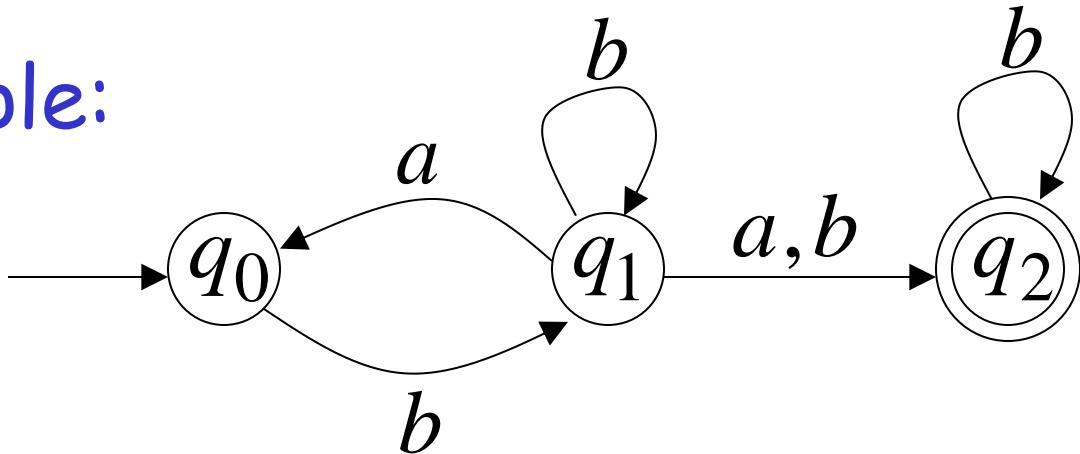
Take it with a single final state

From M construct the equivalent
Generalized Transition Graph
in which transition labels are regular expressions

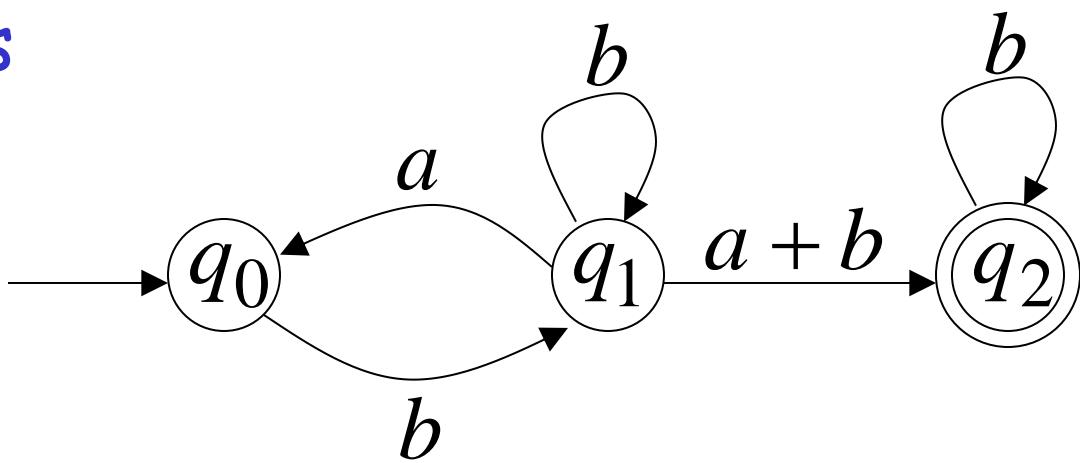
Example:



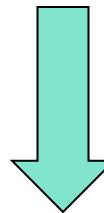
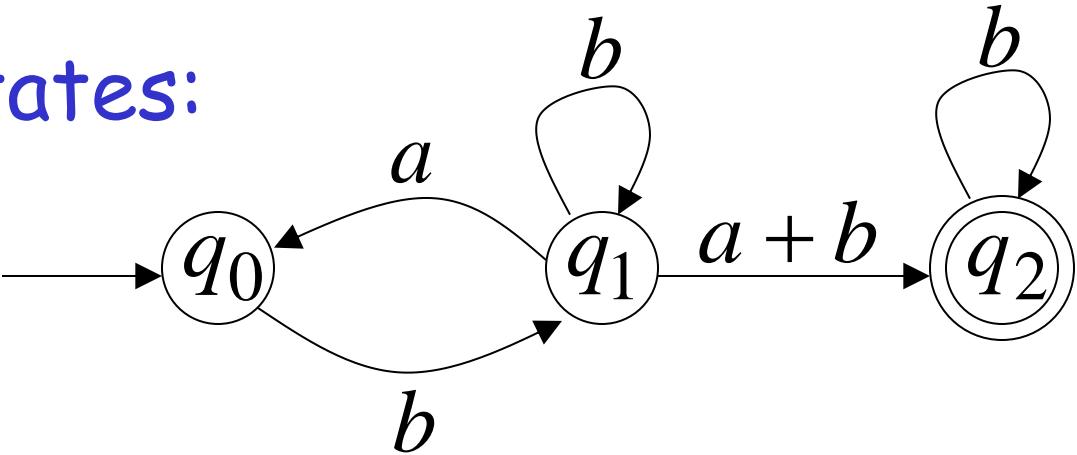
Another Example:



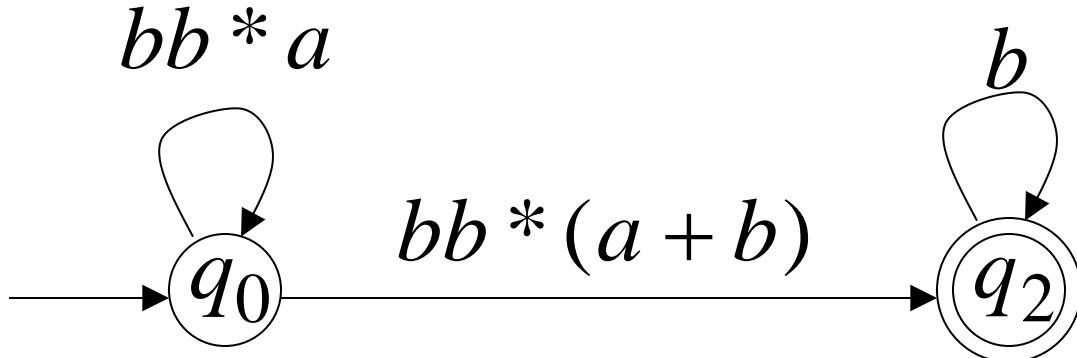
Transition labels
are regular
expressions



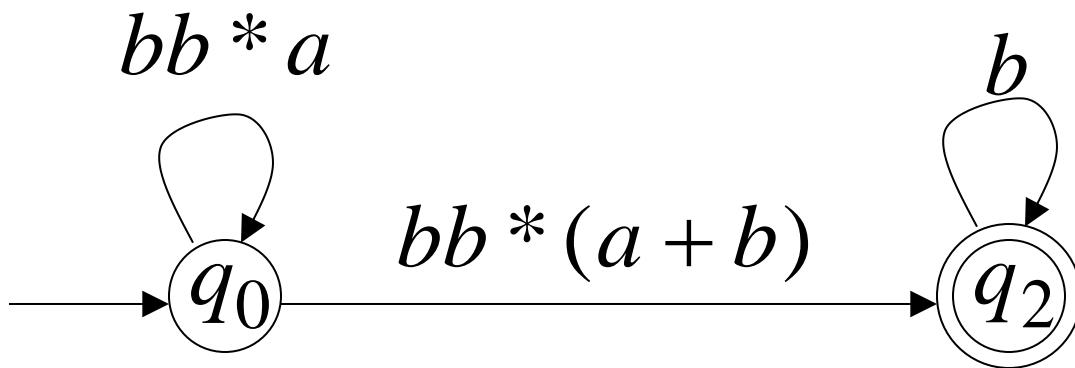
Reducing the states:



Transition labels
are regular
expressions



Resulting Regular Expression:

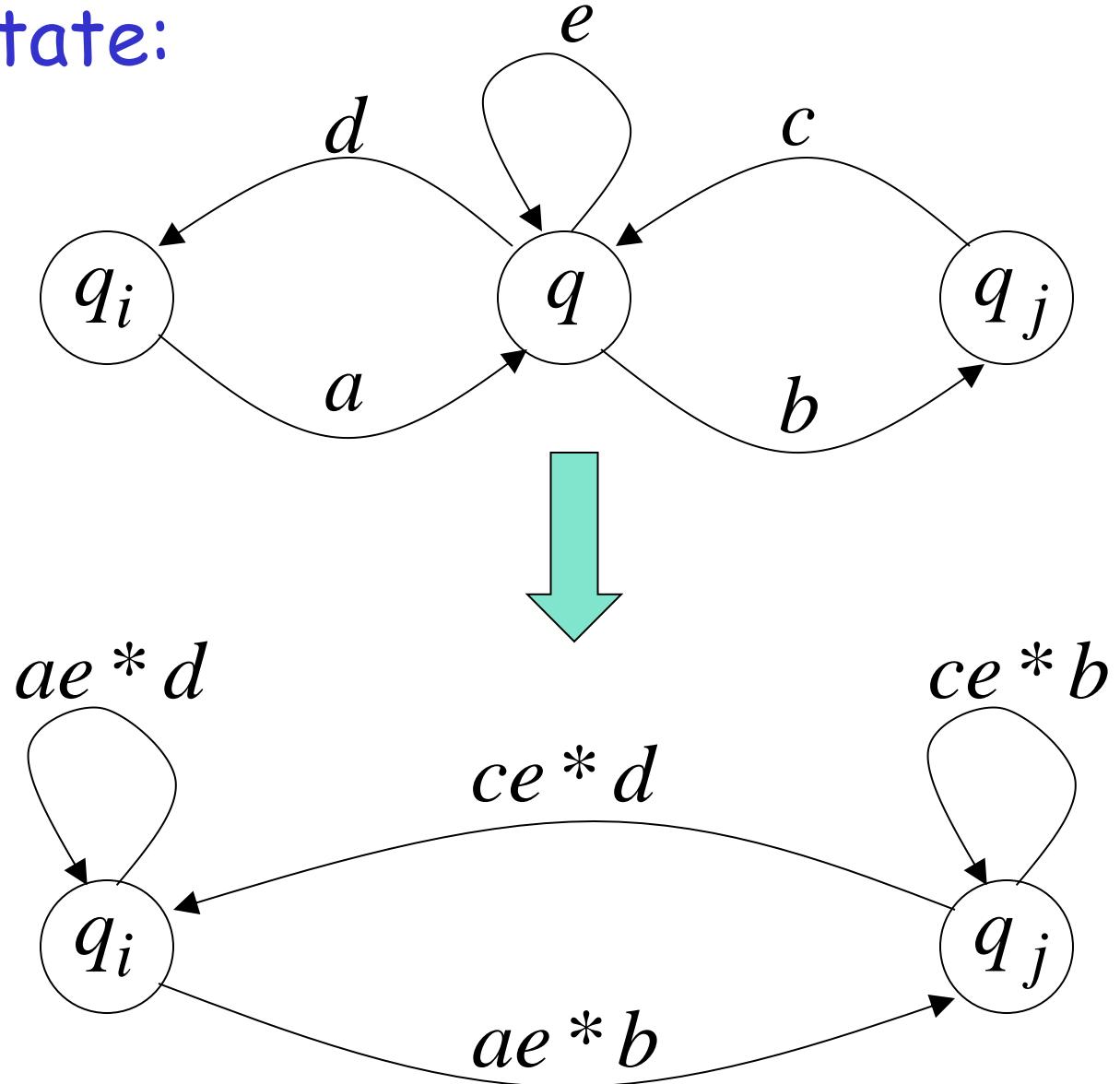


$$r = (bb^*a)^*bb^*(a+b)b^*$$

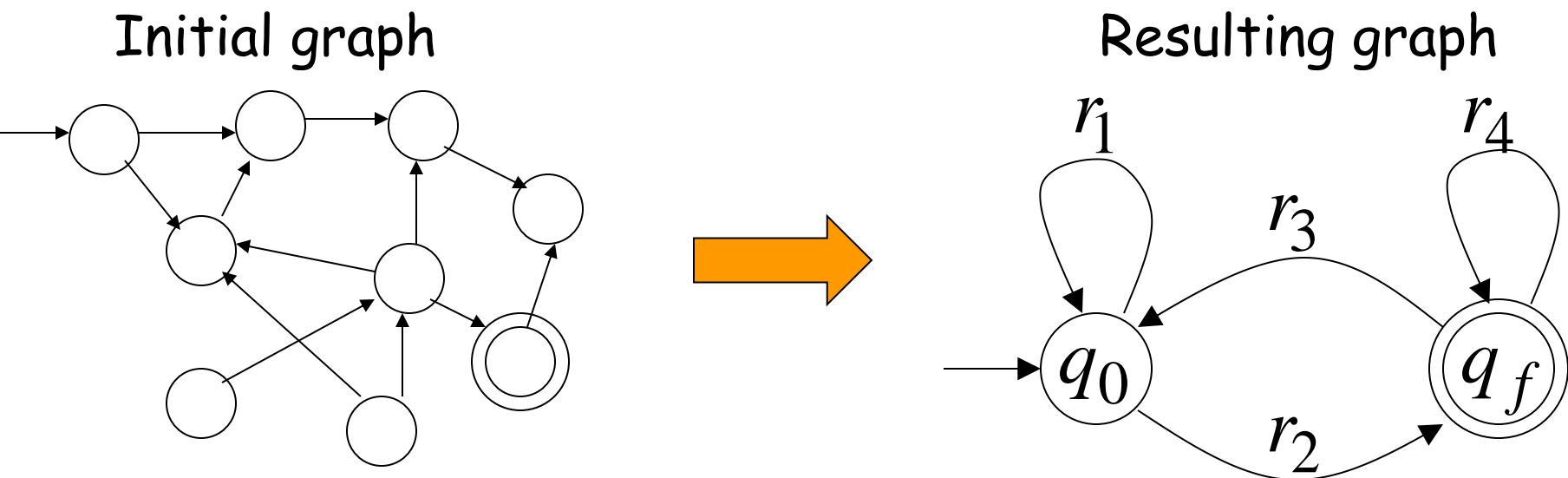
$$L(r) = L(M) = L$$

In General

Removing a state:



By repeating the process until two states are left, the resulting graph is



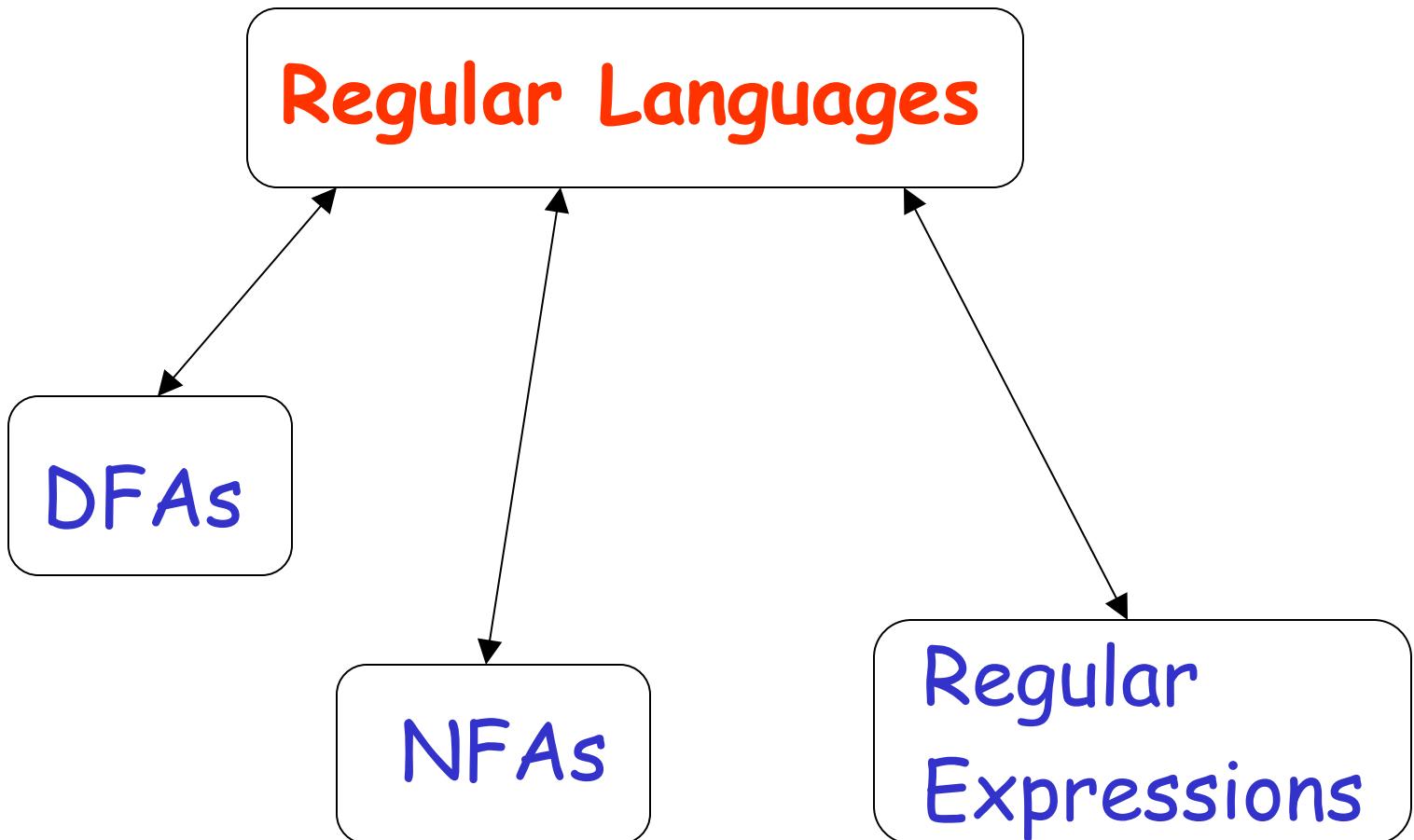
The resulting regular expression:

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

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

End of Proof-Part 2

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)

Formal Grammar

Languages & Grammars

Phrase-Structure Grammars

Types of Phrase-Structure Grammars

Derivation Trees

Backus-Naur Form

Intro to Languages

English grammar tells us if a given combination of words is a valid sentence.

The **syntax** of a sentence concerns its **form** while the **semantics** concerns its **meaning**.

e.g. the mouse wrote a poem

From a **syntax** point of view this is a valid sentence.

From a **semantics** point of view not so fast...perhaps in Disney land

Natural languages (English, French, Portuguese, etc) have very complex rules of syntax and not necessarily well-defined.

Formal Language

Formal language – is specified by well-defined set of rules of syntax

We describe the sentences of a formal language using a grammar.

Two key questions:

- 1 - Is a combination of words a valid sentence in a formal language?
- 2 – How can we generate the valid sentences of a formal language?

Formal languages provide models for both natural languages and programming languages.

Grammars

A formal *grammar* G is any compact, precise mathematical definition of a language L .

- As opposed to just a raw listing of all of the language's legal sentences, or just examples of them.

A grammar implies an algorithm that would generate all legal sentences of the language.

- Often, it takes the form of a set of recursive definitions.

A popular way to specify a grammar recursively is to specify it as a *phrase-structure grammar*.

Grammars (Semi-formal)

Example: A grammar that generates a subset of the English language

$$\langle \textit{sentence} \rangle \rightarrow \langle \textit{noun_phrase} \rangle \; \langle \textit{predicate} \rangle$$
$$\langle \textit{noun_phrase} \rangle \rightarrow \langle \textit{article} \rangle \; \langle \textit{noun} \rangle$$
$$\langle \textit{predicate} \rangle \rightarrow \langle \textit{verb} \rangle$$

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

A derivation of “**the boy sleeps**”:

$$\begin{aligned}\langle \textit{sentence} \rangle &\Rightarrow \langle \textit{noun_phrase} \rangle \langle \textit{predicate} \rangle \\&\Rightarrow \langle \textit{noun_phrase} \rangle \langle \textit{verb} \rangle \\&\Rightarrow \langle \textit{article} \rangle \langle \textit{noun} \rangle \langle \textit{verb} \rangle \\&\Rightarrow \textit{the} \langle \textit{noun} \rangle \langle \textit{verb} \rangle \\&\Rightarrow \textit{the boy} \langle \textit{verb} \rangle \\&\Rightarrow \textit{the boy sleeps}\end{aligned}$$

A derivation of “**a dog runs**”:

$$\begin{aligned}\langle \textit{sentence} \rangle &\Rightarrow \langle \textit{noun_phrase} \rangle \; \langle \textit{predicate} \rangle \\&\Rightarrow \langle \textit{noun_phrase} \rangle \; \langle \textit{verb} \rangle \\&\Rightarrow \langle \textit{article} \rangle \; \langle \textit{noun} \rangle \; \langle \textit{verb} \rangle \\&\Rightarrow a \; \langle \textit{noun} \rangle \; \langle \textit{verb} \rangle \\&\Rightarrow a \; \textit{dog} \; \langle \textit{verb} \rangle \\&\Rightarrow a \; \textit{dog} \; \textit{runs}\end{aligned}$$

Language of the grammar:

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

Notation

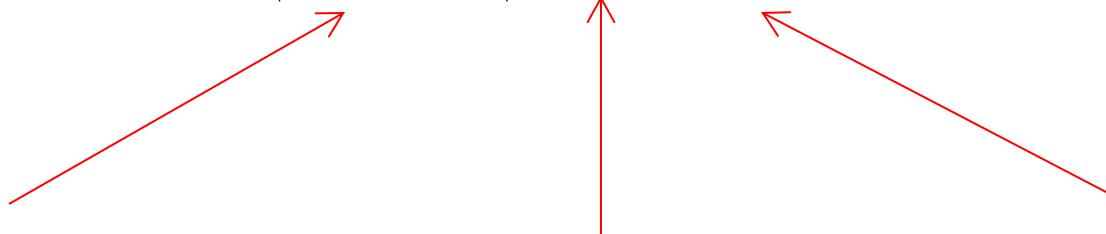
$$\langle \text{noun} \rangle \rightarrow \text{boy}$$
$$\langle \text{noun} \rangle \rightarrow \text{dog}$$

Variable
or
Non-terminal

Symbols of
the vocabulary

Production
rule

Terminal
Symbols of
the vocabulary



Basic Terminology

- ▶ A **vocabulary/alphabet**, V is a finite nonempty set of elements called symbols.
 - Example: $V = \{a, b, c, A, B, C, S\}$
- ▶ A **word/sentence** over V is a string of finite length of elements of V .
 - Example: Aba
- ▶ The **empty/null string**, λ is the string with no symbols.
- ▶ V^* is the set of all words over V .
 - Example: $V^* = \{Aba, BBa, bAA, cab \dots\}$
- ▶ A **language** over V is a subset of V^* .
 - We can give some criteria for a word to be in a language.

Phrase-Structure Grammars

A *phrase-structure grammar* (abbr. PSG)

$G = (V, T, S, P)$ is a 4-tuple, in which:

- V is a vocabulary (set of symbols)
 - The “template vocabulary” of the language.
- $T \subseteq V$ is a set of symbols called *terminals*
 - Actual symbols of the language.
 - Also, $N := V - T$ is a set of special “symbols” called *nonterminals*. (Representing concepts like “noun”)
- $S \in N$ is a special nonterminal, the *start symbol*.
 - in our example the start symbol was “sentence”.
- P is a set of *productions* (to be defined).
 - Rules for substituting one sentence fragment for another
 - Every production rule must contain at least one nonterminal on its left side.

Phrase-structure Grammar

► **EXAMPLE:**

- Let $G = (V, T, S, P)$,
- where $V = \{a, b, A, B, S\}$
- $T = \{a, b\}$,
- S is a start symbol
- $P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, A \rightarrow Bb\}$.

G is a Phrase-Structure Grammar.

What sentences can be generated
with this grammar?

Derivation

Definition

Let $G = (V, T, S, P)$ be a phrase-structure grammar.

Let $w_0 = lz_0r$ (the concatenation of l, z_0 , and r) $w_1 = lz_1r$ be strings over V.

If $z_0 \rightarrow z_1$ is a production of G we say that w_1 is **directly derivable** from w_0 and we write $w_0 \Rightarrow w_1$.

If w_0, w_1, \dots, w_n are strings over V such that $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \dots, w_{n-1} \Rightarrow w_n$, then we say that w_n is derivable from w_0 , and write $w_0 \Rightarrow^* w_n$.

The sequence of steps used to obtain w_n from w_0 is called a **derivation**.

Language

Let $G(V, T, S, P)$ be a phrase-structure grammar. The language generated by G (or the language of G) denoted by $L(G)$, is the set of all strings of terminals that are derivable from the starting state S .

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

Language L(G)

► EXAMPLE:

Let $G = (V, T, S, P)$, where $V = \{a, b, A, S\}$, $T = \{a, b\}$, S is a start symbol and $P = \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\}$.

The language of this grammar is given by $L(G) = \{b, aaa\}$;

1. we can derive aA from using $S \rightarrow aA$, and then derive aaa using $A \rightarrow aa$.
2. We can also derive b using $S \rightarrow b$.

Another example

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

$T = \{a, b\}$

$P = \begin{array}{l} S \rightarrow aSb \\ S \rightarrow \lambda \end{array}$

$V = \{a, b, S\}$

Derivation of sentence : ab

$$S \Rightarrow aSb \Rightarrow ab$$

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

$$S \rightarrow aSb$$

Grammar:

$$S \rightarrow \lambda$$

Derivation of sentence

$$aabb$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$



$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

Other derivations:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabb$$

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

$$\Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$$

So, what's the language of the grammar with the productions?

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

Language of the grammar with the productions:

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

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

PSG Example – English Fragment

We have $G = (V, T, S, P)$, where:

$V = \{\text{(sentence), (noun phrase),}$
 $\quad \text{(verb phrase), (article), (adjective),}$
 $\quad \text{(noun), (verb), (adverb), } a, the, large,$
 $\quad hungry, rabbit, mathematician, eats, hops,$
 $\quad quickly, wildly\}$

$T = \{a, the, large, hungry, rabbit, mathematician,$
 $\quad eats, hops, quickly, wildly\}$

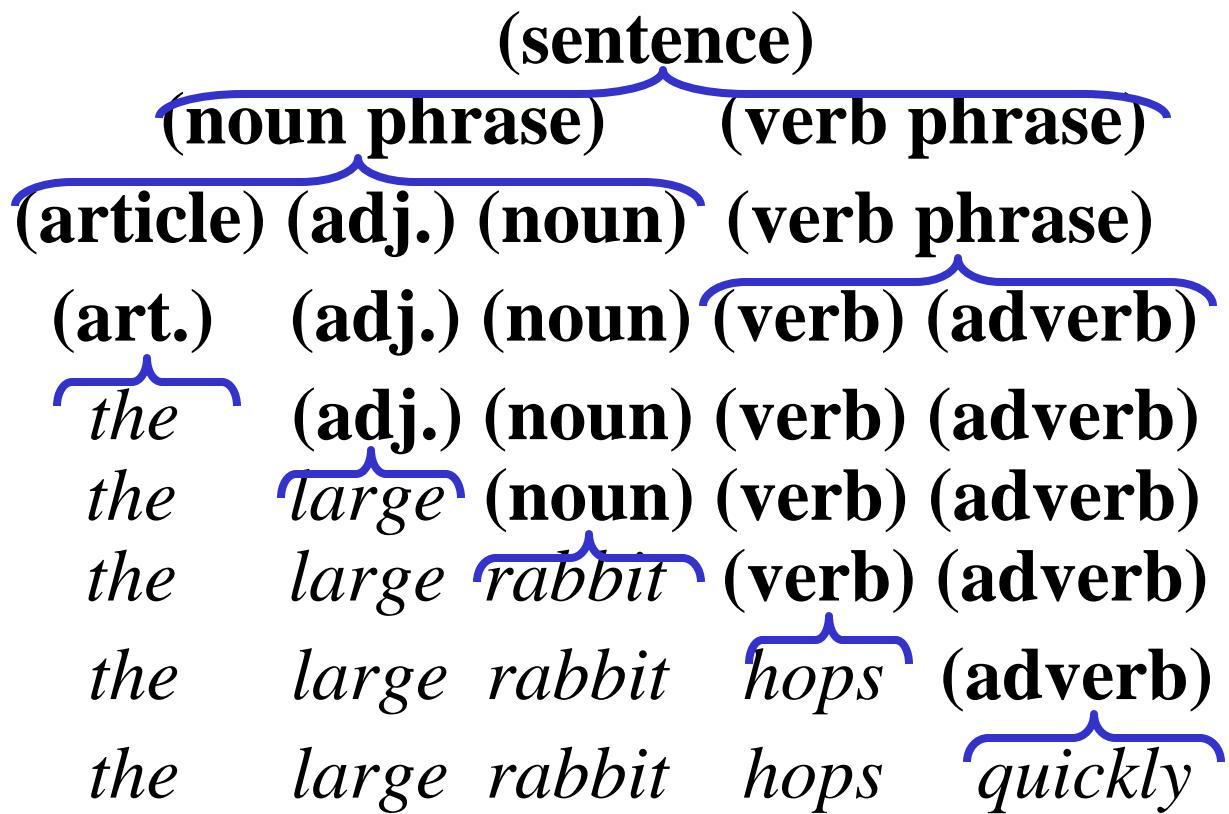
$S = \text{(sentence)}$

$P = \text{(see next slide)}$

Productions for our Language

$P = \{ (\text{sentence}) \rightarrow (\text{noun phrase}) (\text{verb phrase}),$
 $(\text{noun phrase}) \rightarrow (\text{article}) (\text{adjective}) (\text{noun}),$
 $(\text{noun phrase}) \rightarrow (\text{article}) (\text{noun}),$
 $(\text{verb phrase}) \rightarrow (\text{verb}) (\text{adverb}),$
 $(\text{verb phrase}) \rightarrow (\text{verb}),$
 $(\text{article}) \rightarrow a, (\text{article}) \rightarrow the,$
 $(\text{adjective}) \rightarrow large, (\text{adjective}) \rightarrow hungry,$
 $(\text{noun}) \rightarrow rabbit, (\text{noun}) \rightarrow mathematician,$
 $(\text{verb}) \rightarrow eats, (\text{verb}) \rightarrow hops,$
 $(\text{adverb}) \rightarrow quickly, (\text{adverb}) \rightarrow wildly \}$

A Sample Sentence Derivation



On each step, we apply a production to a fragment of the previous sentence template to get a new sentence template. Finally, we end up with a sequence of terminals (real words), **that is, a sentence of our language L .**

Another Example

Let $G = (\{a, b, A, B, S\}, \{a, b\}, S, P)$

$\{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, AB \rightarrow b\}).$

One possible derivation in this grammar is:

$$\begin{aligned} S &\Rightarrow ABa \\ &\Rightarrow Aaba \\ &\Rightarrow BBaba \\ &\Rightarrow Bababa \\ &\Rightarrow abababa. \end{aligned}$$

Defining the PSG Types

Type 0: Phase-structure grammars – no restrictions on the production rules

Type 1: Context-Sensitive PSG:

- All after fragments are either longer than the corresponding before fragments, or empty:

$$\text{if } b \rightarrow a, \text{ then } |b| < |a| \quad \vee \quad a = \lambda .$$

Type 2: Context-Free PSG:

- All before fragments have length 1 and are nonterminals:

$$\text{if } b \rightarrow a, \text{ then } |b| = 1 (b \in N).$$

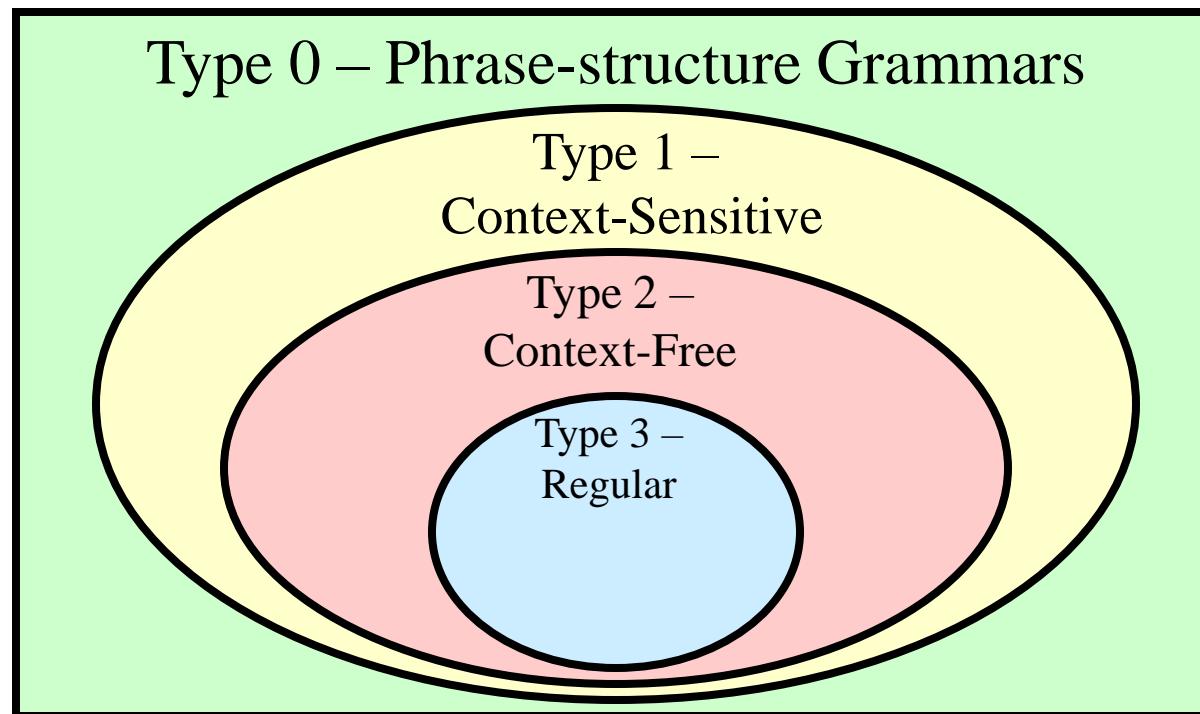
Type 3: Regular PSGs:

- All before fragments have length 1 and nonterminals
- All after fragments are either single terminals, or a pair of a terminal followed by a nonterminal.

$$\text{if } b \rightarrow a, \text{ then } a \in T \quad \vee \quad a \in TN.$$

Types of Grammars - Chomsky hierarchy of languages

Venn Diagram of Grammar Types:



Classifying grammars

Given a grammar, we need to be able to find the smallest class in which it belongs. This can be determined by answering three questions:

Are the left hand sides of all of the productions single non-terminals?

If yes, does each of the productions create at most one non-terminal and is it on the right?

Yes – regular

No – context-free

If not, can any of the rules reduce the length of a string of terminals and non-terminals?

Yes – unrestricted

No – context-sensitive

Definition: Context-Free Grammars

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

Vocabulary Terminal Start
symbols variable

Productions of the form:

$$A \rightarrow x$$

Non-Terminal **String of variables
and terminals**

Derivation Tree of A Context-free Grammar

- ▶ Represents the language using an ordered rooted tree.
- ▶ Root represents the **starting symbol**.
- ▶ Internal vertices represent the **nonterminal symbol** that arise in the production.
- ▶ Leaves represent the **terminal symbols**.
- ▶ If the production $A \rightarrow w$ arise in the derivation, where w is a word, the vertex that represents A has as children vertices that represent each symbol in w , in order from left to right.

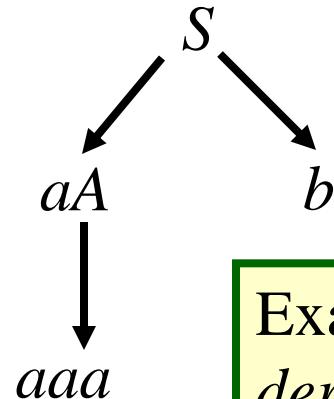
Language Generated by a Grammar

Example: Let $G = (\{S, A, a, b\}, \{a, b\}, S, \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\})$. What is $L(G)$?

Easy: We can just draw a tree of all possible derivations.

- We have: $S \Rightarrow aA \Rightarrow aaa$.
- and $S \Rightarrow b$.

Answer: $L = \{aaa, b\}$.



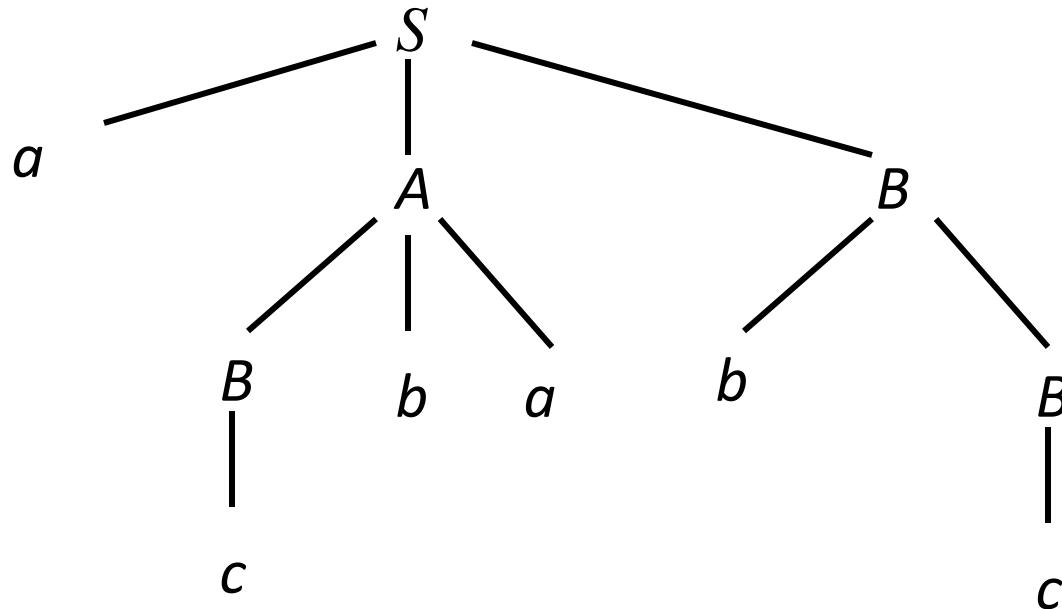
Example of a *derivation tree* or *parse tree* or *sentence diagram*.

Example: Derivation Tree

- Let \overline{G} be a context-free grammar with the productions
 $P = \{S \rightarrow aAB, A \rightarrow Bba, B \rightarrow bB, B \rightarrow c\}$. The word $w = acbabc$ can be derived from S as follows:

$$S \Rightarrow aAB \rightarrow a(Bba)B \Rightarrow acbaB \Rightarrow acba(bB) \Rightarrow acbabc$$

Thus, the derivation tree is given as follows:



Backus-Naur Form

$\langle \text{sentence} \rangle ::= \langle \text{noun phrase} \rangle \langle \text{verb phrase} \rangle$
 $\langle \text{noun phrase} \rangle ::= \langle \text{article} \rangle [\langle \text{adjective} \rangle] \langle \text{noun} \rangle$
 $\langle \text{verb phrase} \rangle ::= \langle \text{verb} \rangle [\langle \text{adverb} \rangle]$
 $\langle \text{article} \rangle ::= \text{a} / \text{the}$
 $\langle \text{adjective} \rangle ::= \text{large} | \text{hungry}$
 $\langle \text{noun} \rangle ::= \text{rabbit} | \text{mathematician}$
 $\langle \text{verb} \rangle ::= \text{eats} / \text{hops}$
 $\langle \text{adverb} \rangle ::= \text{quickly} / \text{wildly}$

Square brackets []
mean “optional”

Vertical bars
mean “alternatives”

Generating Infinite Languages

A simple PSG can easily generate an infinite language.

Example: $S \rightarrow 11S, S \rightarrow 0$ ($T = \{0,1\}$).

The derivations are:

- $S \Rightarrow 0$
- $S \Rightarrow 11S \Rightarrow 110$
- $S \Rightarrow 11S \Rightarrow 1111S \Rightarrow 11110$
- and so on...

$L = \{(11)^*0\}$ – the set of all strings consisting of some number of concatenations of 11 with itself, followed by 0.

Another example

Construct a PSG that generates the language $L = \{0^n 1^n \mid n \in \mathbb{N}\}$.

- **0** and **1** here represent symbols being concatenated n times, not integers being raised to the n th power.

Solution strategy: Each step of the derivation should preserve the invariant that the number of **0**'s = the number of **1**'s in the template so far, and all **0**'s come before all **1**'s.

Solution: $S \rightarrow 0S1, S \rightarrow \lambda$.

Context-Sensitive Languages

The language $\{ a^n b^n c^n \mid n \geq 1 \}$ is context-sensitive but not context free.

A grammar for this language is given by:

$$S \rightarrow aSBC / aBC$$

$$CB \rightarrow BC$$

$$aB \rightarrow ab$$

$$bB \rightarrow bb$$

$$bC \rightarrow bc$$

$$cC \rightarrow cc$$

Terminal
and
non-terminal



A derivation from this grammar is:-

$$\begin{aligned} S &\Rightarrow aSBC \\ &\Rightarrow aaBCBC && \text{(using } S \rightarrow aBC) \\ &\Rightarrow aabCBC && \text{(using } aB \rightarrow ab) \\ &\Rightarrow aabBCC && \text{(using } CB \rightarrow BC) \\ &\Rightarrow aabbCC && \text{(using } bB \rightarrow bb) \\ &\Rightarrow aabbcC && \text{(using } bC \rightarrow bc) \\ &\Rightarrow aabbcc && \text{(using } cC \rightarrow cc) \end{aligned}$$

which derives $a^2b^2c^2$.

Linear Grammars

Grammars with
at most one variable at the right side
of a production

Examples:

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

A Non-Linear Grammar

Grammar G : $S \rightarrow SS$

$S \rightarrow \lambda$

$S \rightarrow aSb$

$S \rightarrow bSa$

$$L(G) = \{w : n_a(w) = n_b(w)\}$$



Number of a in string w

Another Linear Grammar

Grammar $G :$ $S \rightarrow A$

$A \rightarrow aB \mid \lambda$

$B \rightarrow Ab$

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

Right-Linear Grammars

All productions have form: $A \rightarrow xB$

or

$$A \rightarrow x$$



Example: $S \rightarrow abS$

$$S \rightarrow a$$

string of
terminals

Left-Linear Grammars

All productions have form: $A \rightarrow Bx$

or

$$A \rightarrow x$$


Example: $S \rightarrow Aab$

$$A \rightarrow Aab \mid B$$
$$B \rightarrow a$$

string of
terminals

Regular Grammars

Regular Grammars

A regular grammar is any
right-linear or left-linear grammar

Examples:

G_1

$S \rightarrow abS$

$S \rightarrow a$

G_2

$S \rightarrow Aab$

$A \rightarrow Aab \mid B$

$B \rightarrow a$

Observation

Regular grammars generate regular languages

Examples:

G_1

$S \rightarrow abS$

$S \rightarrow a$

$L(G_1) = (ab)^* a$

G_2

$S \rightarrow Aab$

$A \rightarrow Aab \mid B$

$B \rightarrow a$

$L(G_2) = aab(ab)^*$

Regular Grammars
Generate
Regular Languages

Theorem

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

Theorem - Part 1

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

Any regular grammar generates
a regular language

Theorem - Part 2

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Grammars} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Any regular language is generated
by a regular grammar

Proof - Part 1

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

The language $L(G)$ generated by
any regular grammar G is regular

The case of Right-Linear Grammars

Let G be a right-linear grammar

We will prove: $L(G)$ is regular

Proof idea: We will construct NFA M with $L(M) = L(G)$

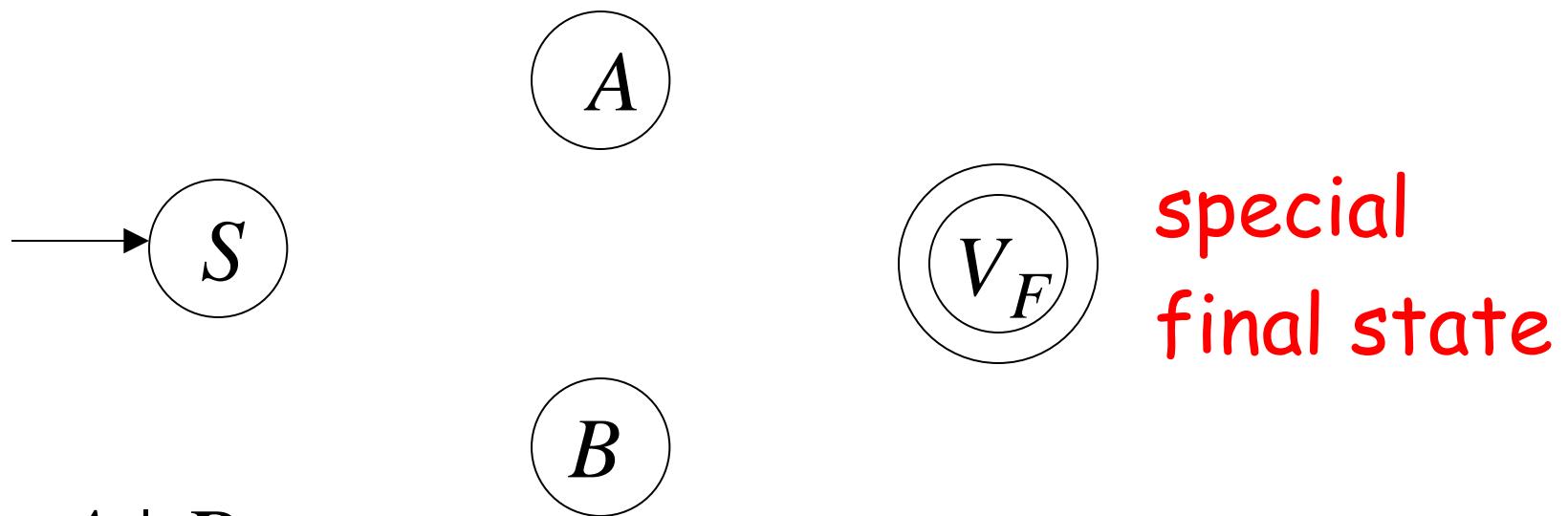
Grammar G is right-linear

Example: $S \rightarrow aA \mid B$

$A \rightarrow aa \mid B$

$B \rightarrow b \mid B \mid a$

Construct NFA M such that
every state is a grammar variable:



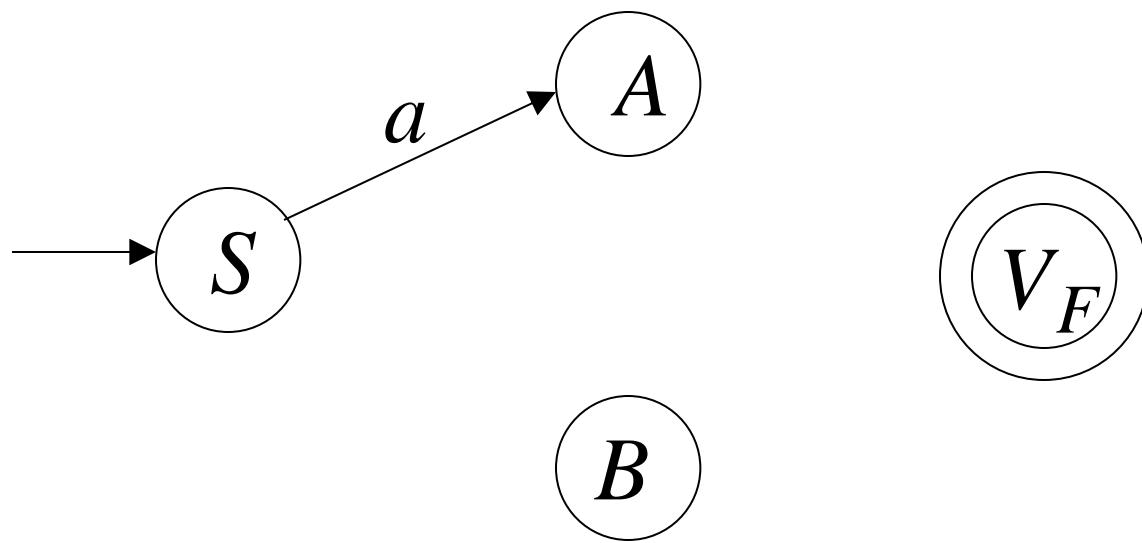
$$S \rightarrow aA \mid B$$

$$A \rightarrow aa \ B$$

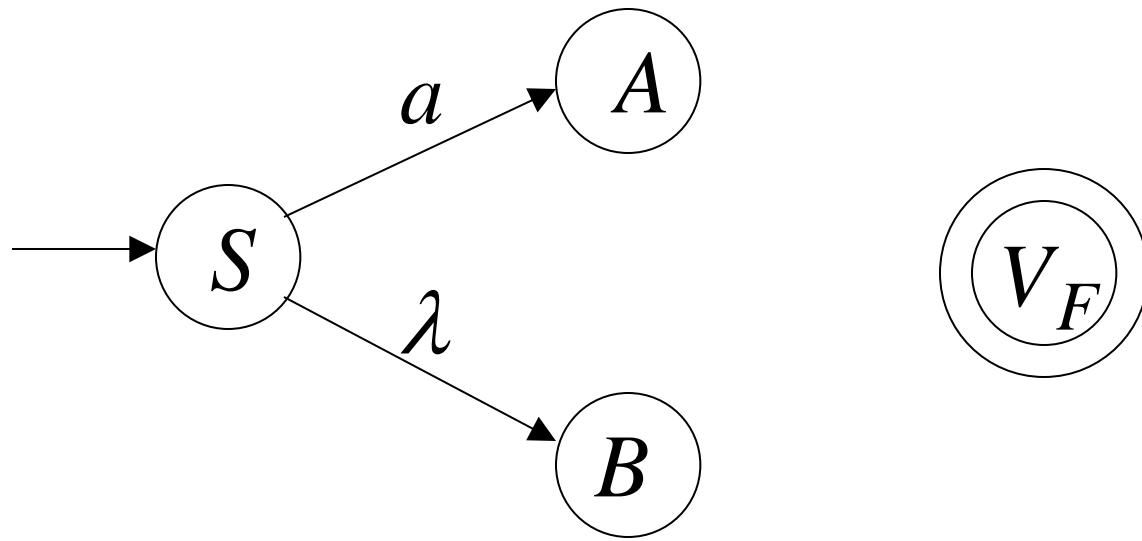
$$B \rightarrow b \ B \mid a$$

special
final state

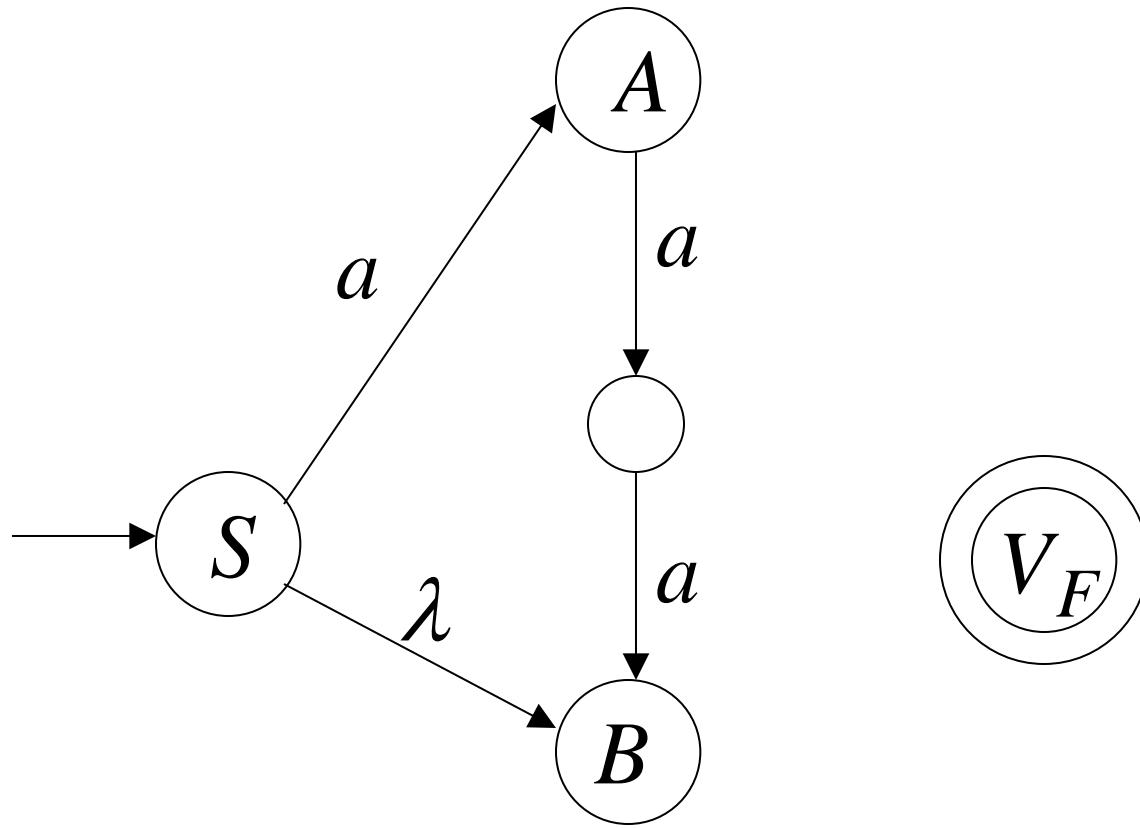
Add edges for each production:



$$S \rightarrow aA$$

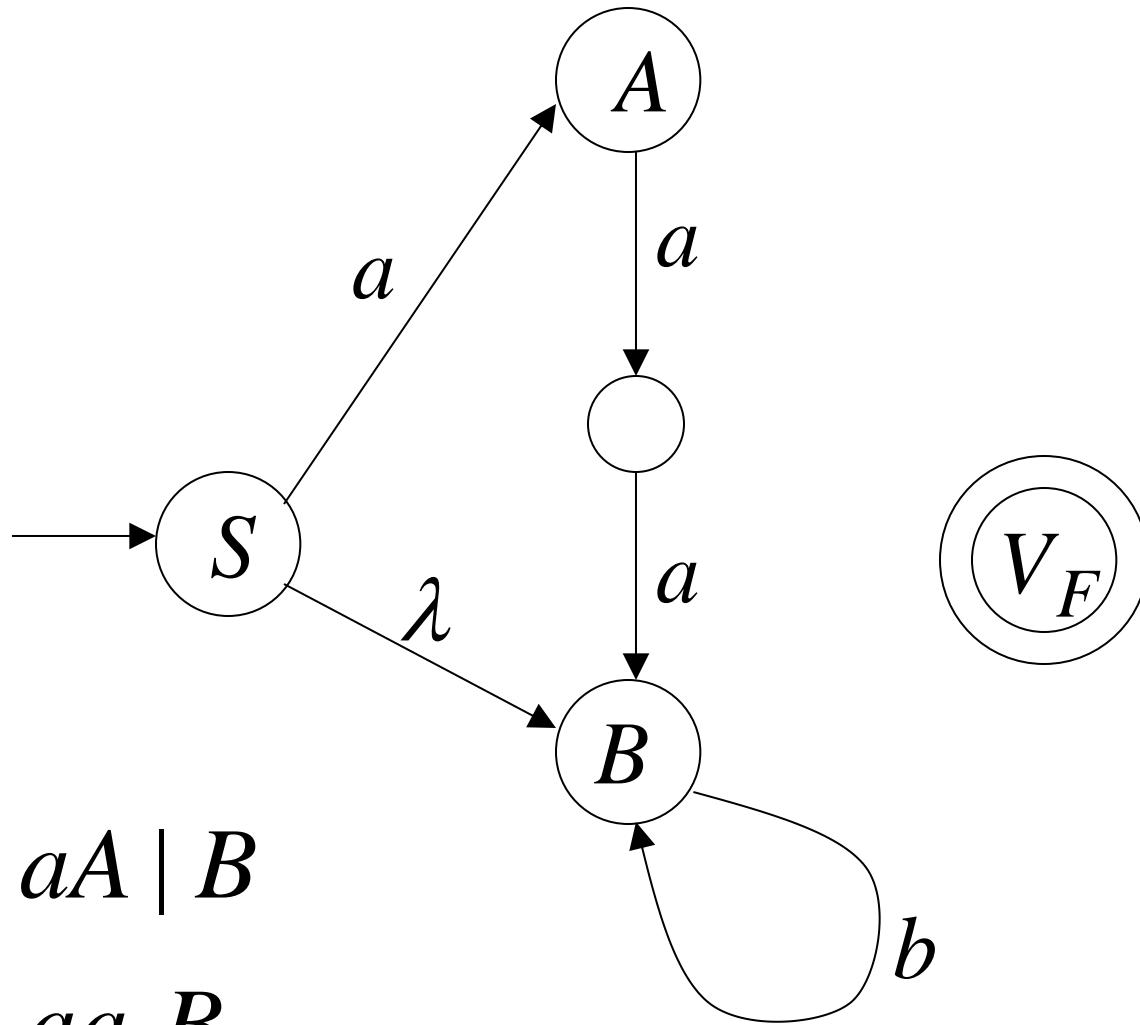


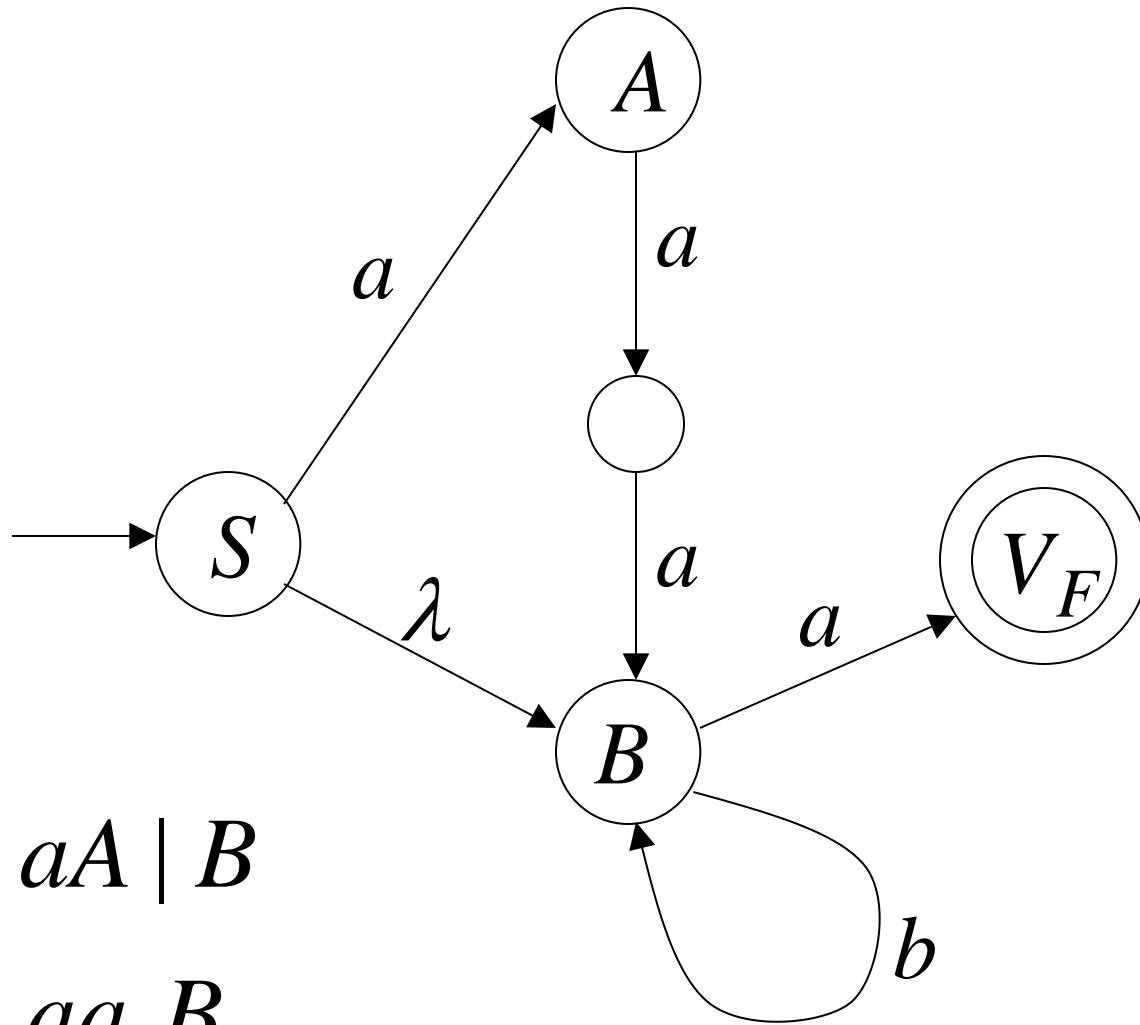
$$S \rightarrow aA \mid B$$

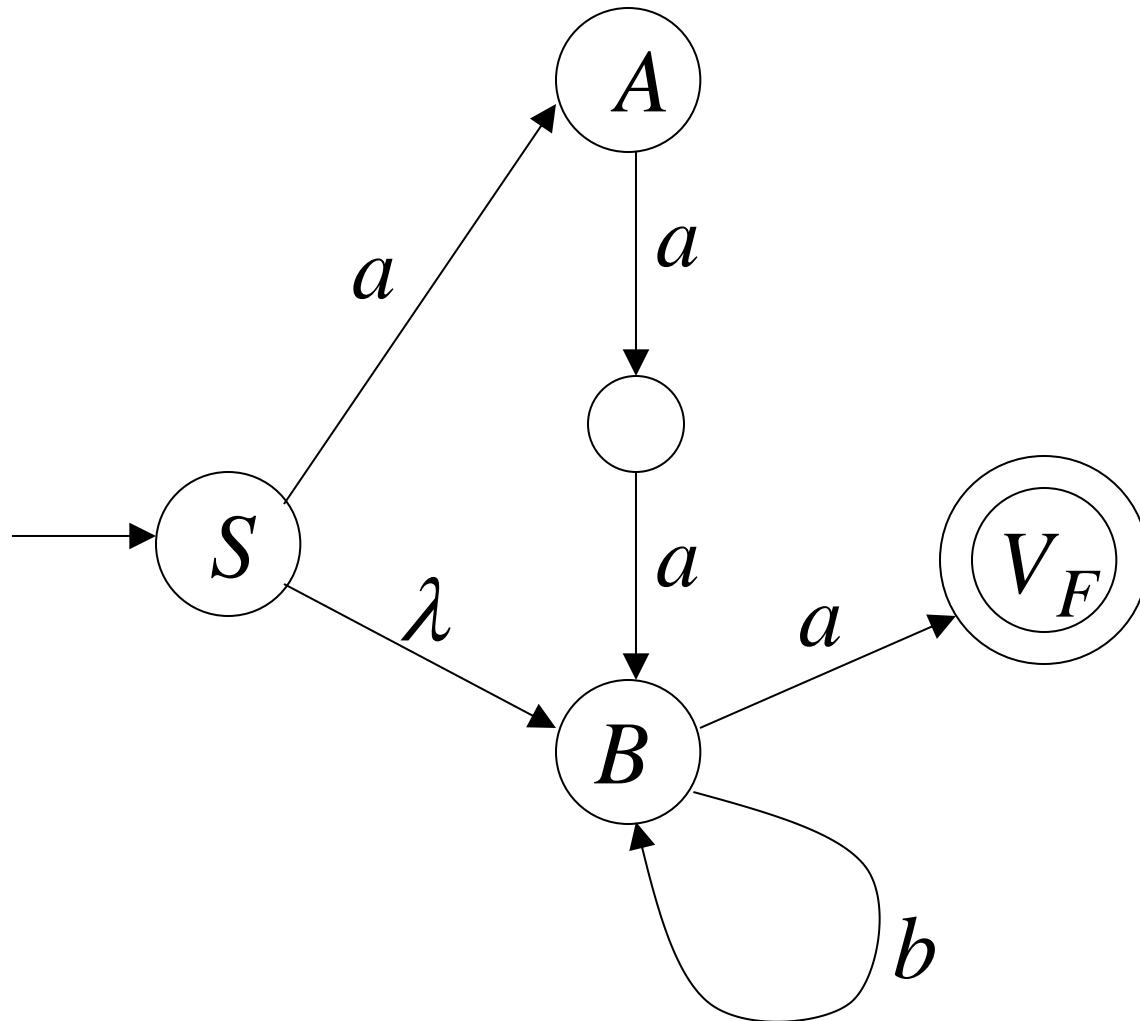


$$S \rightarrow aA \mid B$$

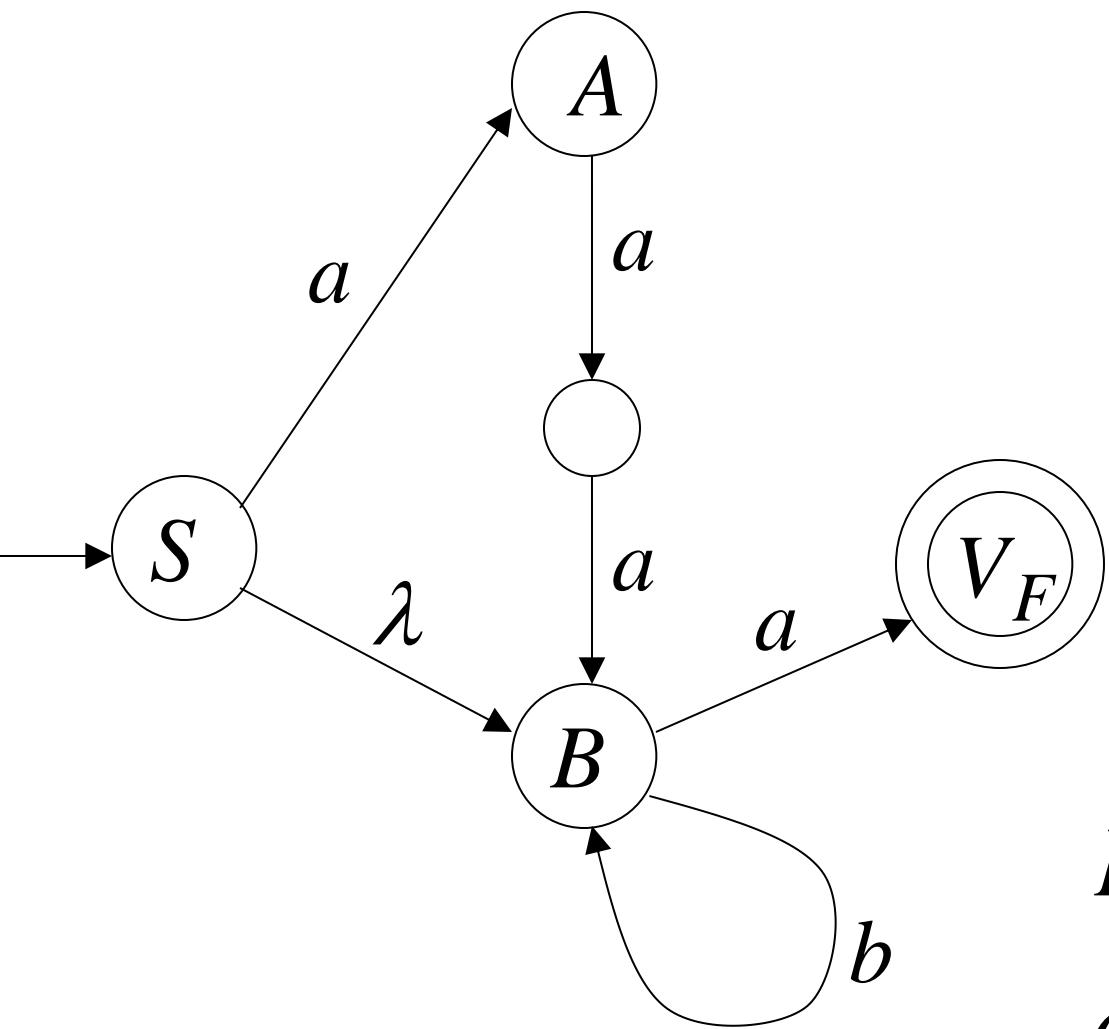
$$A \rightarrow aa \ B$$


$$S \rightarrow aA \mid B$$
$$A \rightarrow aa \ B$$
$$B \rightarrow bB$$


$$S \rightarrow aA \mid B$$
$$A \rightarrow aa \ B$$
$$B \rightarrow bB \mid a$$


$$S \Rightarrow aA \Rightarrow aaaB \Rightarrow aaabB \Rightarrow aaaba$$

NFA M



Grammar

G

$$S \rightarrow aA \mid B$$

$$A \rightarrow aa \ B$$

$$B \rightarrow bB \mid a$$

$$\begin{aligned} L(M) &= L(G) = \\ aaab^*a + b^*a \end{aligned}$$

In General

A right-linear grammar G

has variables: V_0, V_1, V_2, \dots

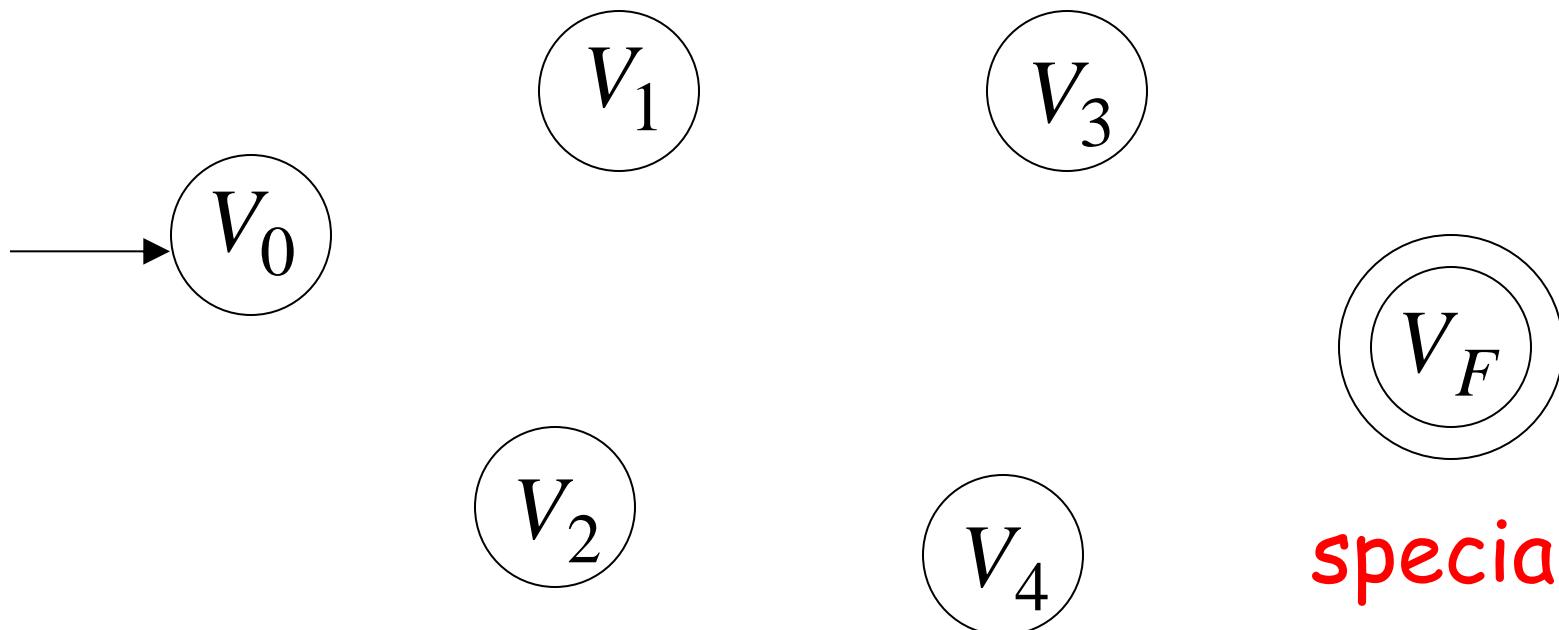
and productions: $V_i \rightarrow a_1 a_2 \cdots a_m V_j$

or

$V_i \rightarrow a_1 a_2 \cdots a_m$

We construct the NFA M such that:

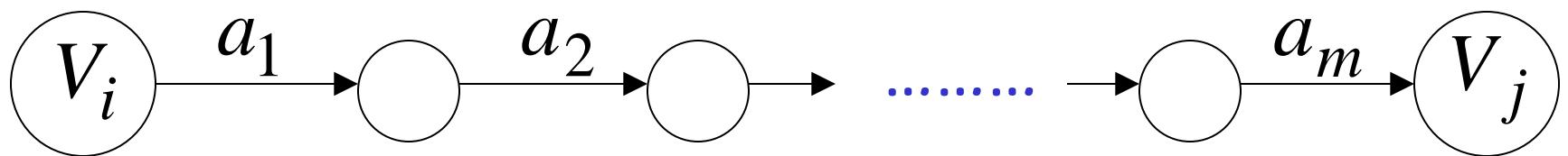
each variable V_i corresponds to a node:



special
final state

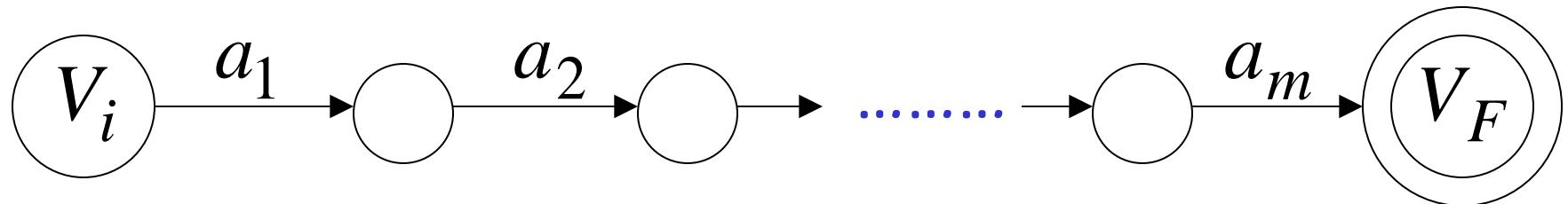
For each production: $V_i \rightarrow a_1 a_2 \cdots a_m V_j$

we add transitions and intermediate nodes

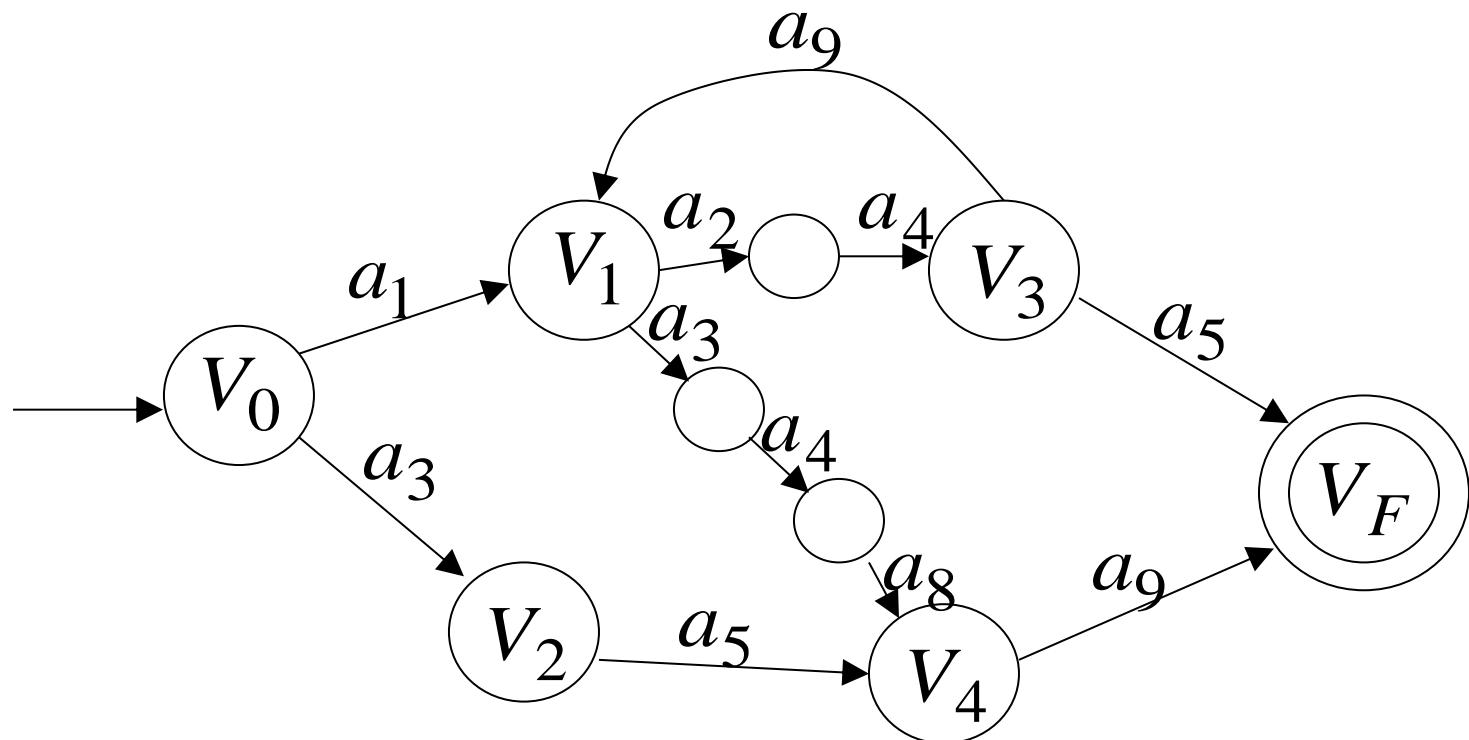


For each production: $V_i \rightarrow a_1 a_2 \cdots a_m$

we add transitions and intermediate nodes



Resulting NFA M looks like this:



It holds that: $L(G) = L(M)$

The case of Left-Linear Grammars

Let G be a left-linear grammar

We will prove: $L(G)$ is regular

Proof idea:

We will construct a right-linear

grammar G' with $L(G) = L(G')^R$

Since G is left-linear grammar
the productions look like:

$$A \rightarrow Ba_1a_2 \cdots a_k$$

$$A \rightarrow a_1a_2 \cdots a_k$$

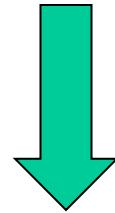
Construct right-linear grammar G'

Left
linear

G

$$A \rightarrow Ba_1a_2 \cdots a_k$$

$$A \rightarrow Bv$$



Right
linear

G'

$$A \rightarrow a_k \cdots a_2 a_1 B$$

$$A \rightarrow v^R B$$

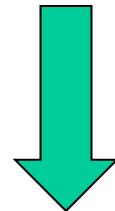
Construct right-linear grammar G'

Left
linear

G

$$A \rightarrow a_1 a_2 \cdots a_k$$

$$A \rightarrow v$$



Right
linear

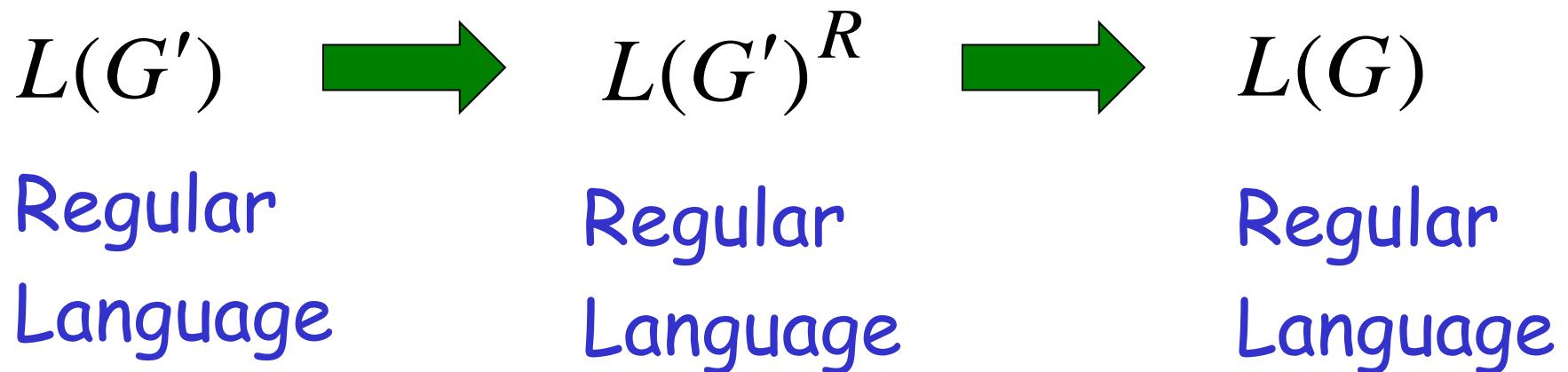
G'

$$A \rightarrow a_k \cdots a_2 a_1$$

$$A \rightarrow v^R$$

It is easy to see that: $L(G) = L(G')^R$

Since G' is right-linear, we have:



Proof - Part 2

$$\left\{ \begin{array}{l} \text{Languages} \\ \text{Generated by} \\ \text{Regular Grammars} \end{array} \right\} \supseteq \left\{ \begin{array}{l} \text{Regular} \\ \text{Languages} \end{array} \right\}$$

Any regular language L is generated
by some regular grammar G

Any regular language L is generated by some regular grammar G

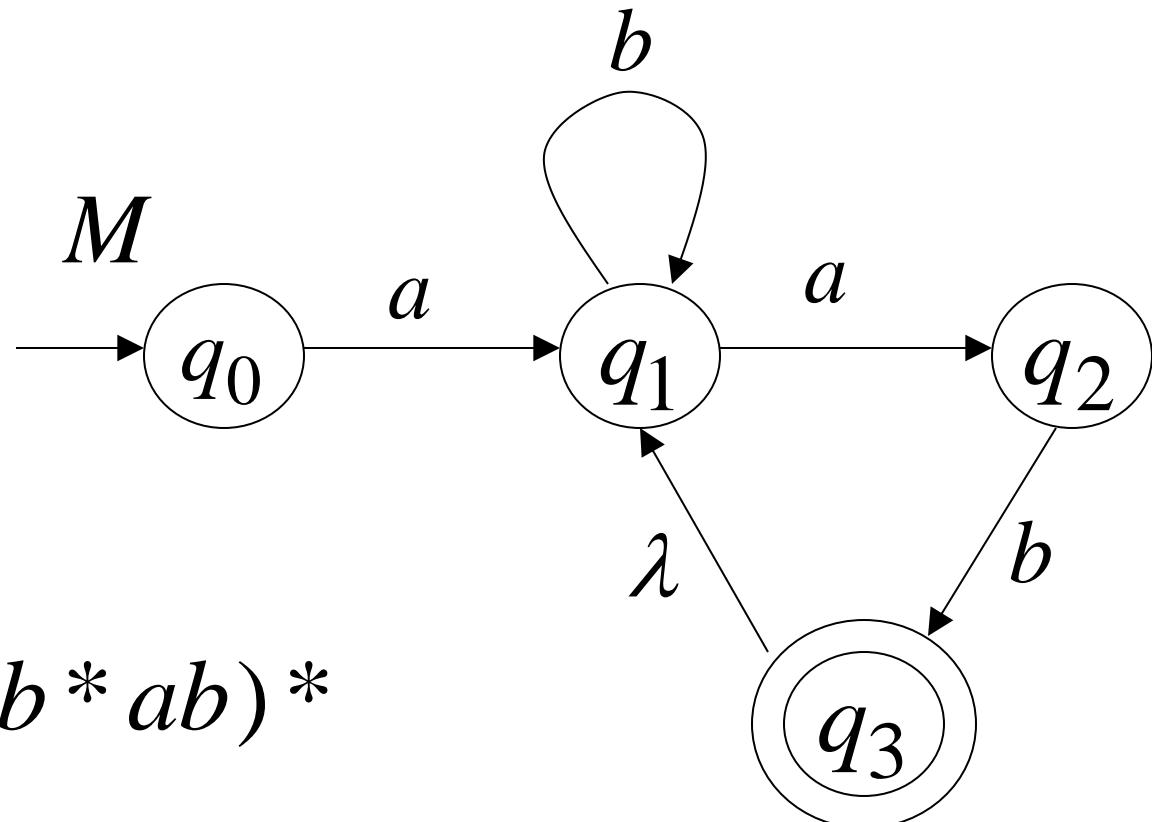
Proof idea:

Let M be the NFA with $L = L(M)$.

Construct from M a regular grammar G such that $L(M) = L(G)$

Since L is regular
there is an NFA M such that $L = L(M)$

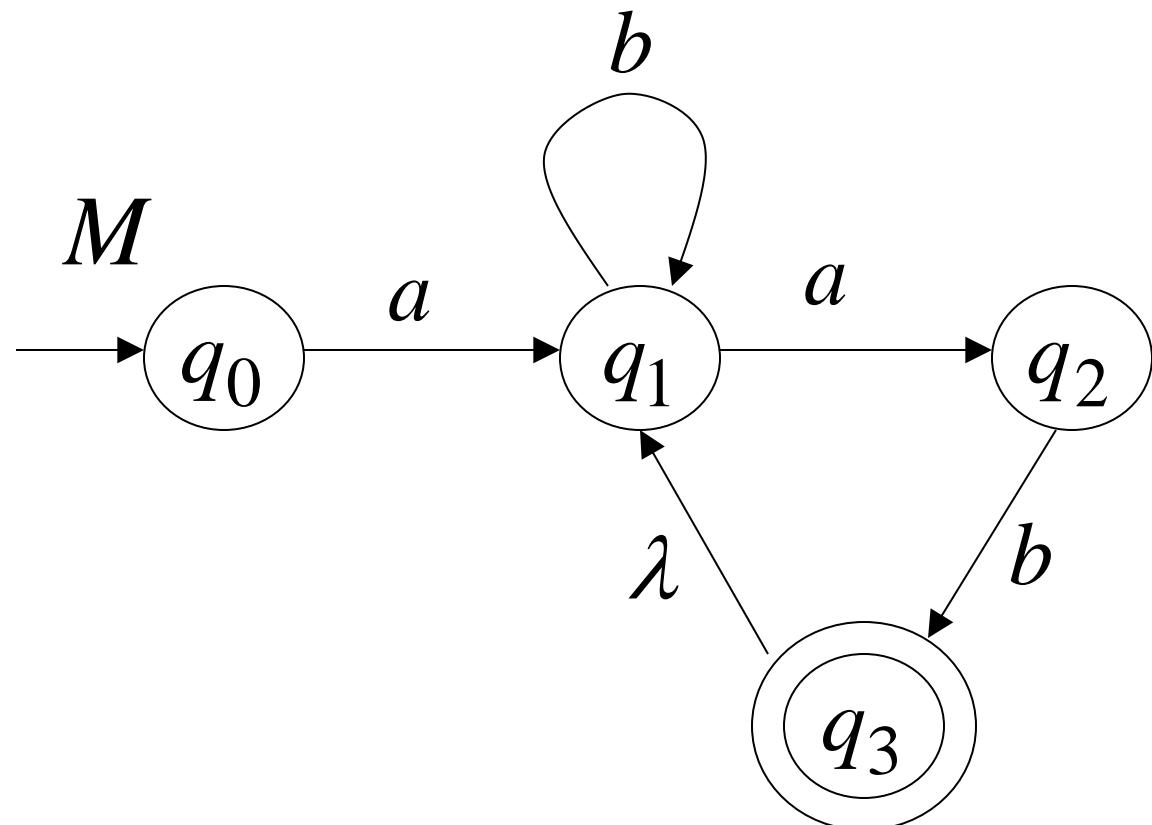
Example:



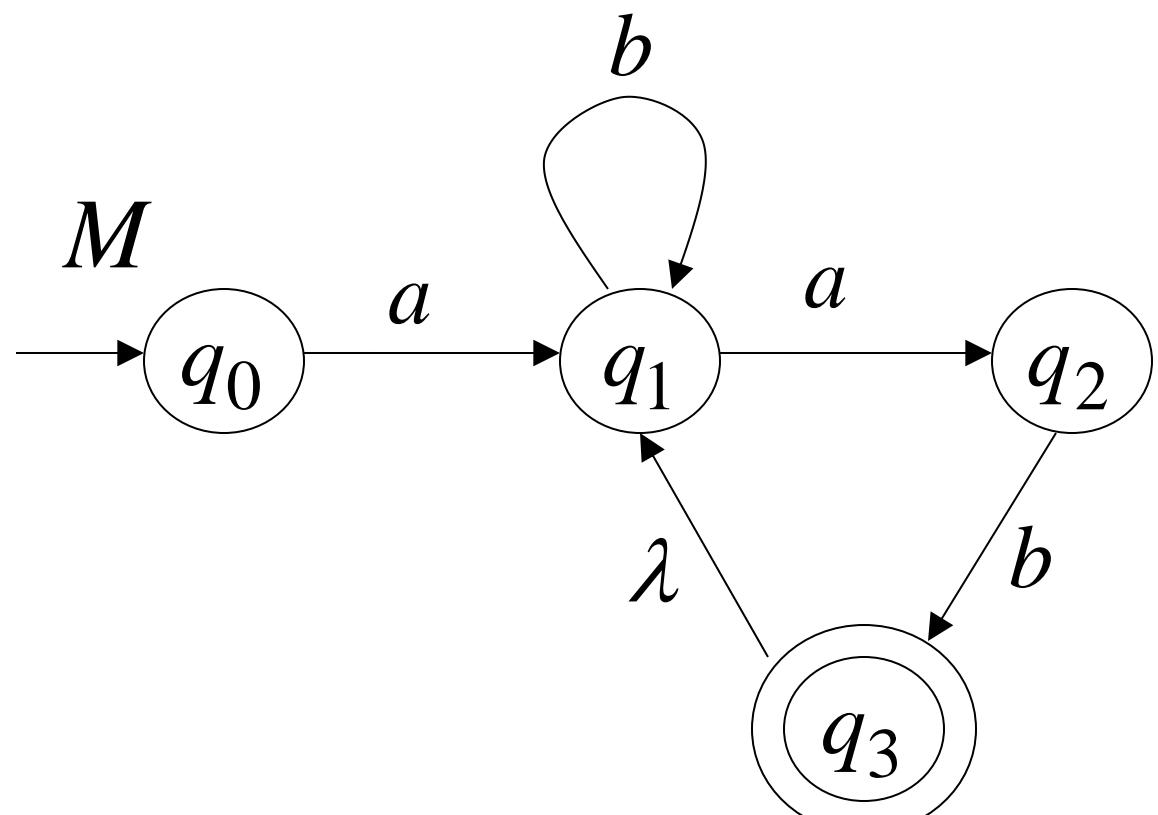
$$L = ab^*ab(b^*ab)^*$$

$$L = L(M)$$

Convert M to a right-linear grammar



$$q_0 \rightarrow aq_1$$

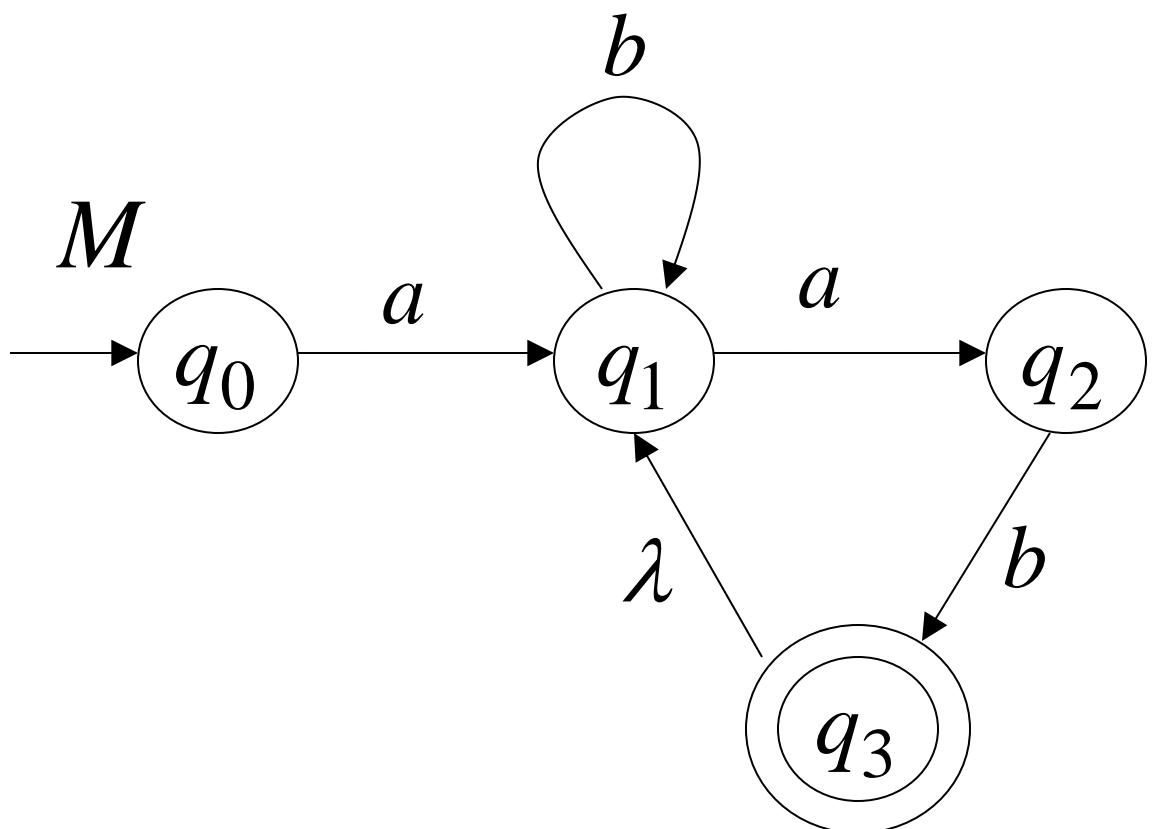


$$q_0 \rightarrow aq_1$$

$$q_1 \rightarrow bq_1$$

$$q_1 \rightarrow aq_2$$

$$\begin{aligned}q_0 &\rightarrow aq_1 \\q_1 &\rightarrow bq_1 \\q_1 &\rightarrow aq_2 \\q_2 &\rightarrow bq_3\end{aligned}$$



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

G

$$q_0 \rightarrow aq_1$$

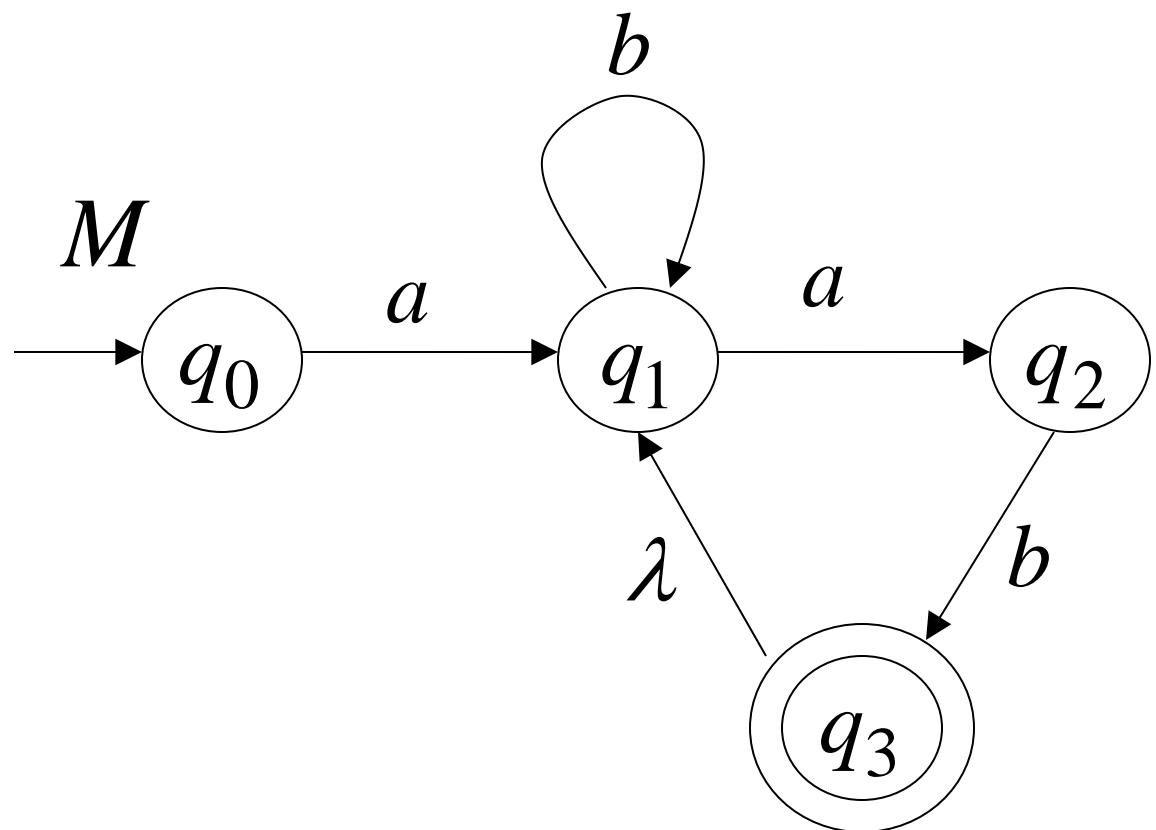
$$q_1 \rightarrow bq_1$$

$$q_1 \rightarrow aq_2$$

$$q_2 \rightarrow bq_3$$

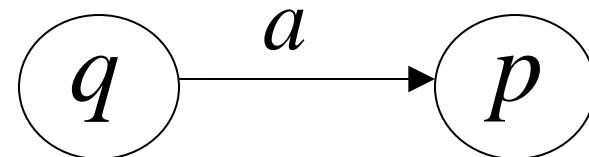
$$q_3 \rightarrow q_1$$

$$q_3 \rightarrow \lambda$$

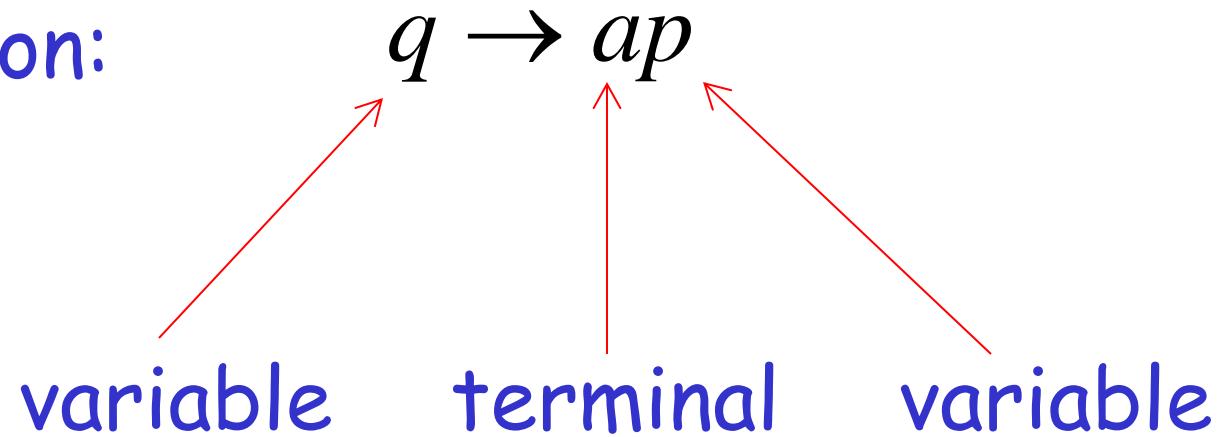


In General

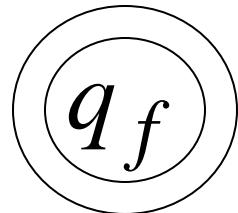
For any transition:



Add production:



For any final state:



Add production:

$$q_f \rightarrow \lambda$$

Since G is right-linear grammar

G is also a regular grammar

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

Lecture Outline

Nonregular Languages

Nonregular Languages

Pumping Lemma

Examples of nonregular languages

Examples

For a fixed integer $n \geq 0$, define a language $L_n = \{0^n 1^n\}$.

Q: Is L_n regular?

Examples

For a fixed integer $n \geq 0$, define a language $L_n = \{0^n 1^n\}$.

Q: Is L_n regular?

Q: Are the following languages regular?

$$L_0 \cup L_1, \quad L_0 \cup L_1 \cup L_2, \quad L_0 \cup L_1 \cup L_2 \cup L_3$$

Examples

For a fixed integer $n \geq 0$, define a language $L_n = \{0^n 1^n\}$.

Q: Is L_n regular?

Q: Are the following languages regular?

$$L_0 \cup L_1, \quad L_0 \cup L_1 \cup L_2, \quad L_0 \cup L_1 \cup L_2 \cup L_3$$

Q: Is the finite union $L_0 \cup L_1 \cup \dots \cup L_m$ regular (for a *fixed* integer $m \geq 0$)?

Examples

For a fixed integer $n \geq 0$, define a language $L_n = \{0^n 1^n\}$.

Q: Is L_n regular?

Q: Are the following languages regular?

$$L_0 \cup L_1, \quad L_0 \cup L_1 \cup L_2, \quad L_0 \cup L_1 \cup L_2 \cup L_3$$

Q: Is the finite union $L_0 \cup L_1 \cup \dots \cup L_m$ regular (for a *fixed* integer $m \geq 0$)?

Q: Is the infinite union of the languages L_n regular?

$$\bigcup_{n \geq 0} = L_0 \cup L_1 \cup \dots = \{0^n 1^n \mid n \geq 0\}.$$

Nonregular Languages

Some languages seem to require an infinite number of states to be recognized by NFA.

$$C = \{w \mid w \text{ has the same number of 0's and 1's}\}$$

$$D = \{w \mid w \text{ has the same number substrings } 01 \text{ and } 10\}$$

Nonregular Languages

Some languages seem to require an infinite number of states to be recognized by NFA.

$$C = \{w \mid w \text{ has the same number of 0's and 1's}\}$$

$$D = \{w \mid w \text{ has the same number substrings } 01 \text{ and } 10\}$$

However, only those that *cannot* be recognized with any finite number of states (recall that F in DFA/NFA stands for “finite”) are nonregular.

Surprisingly, the language D can be recognized by an NFA, i.e., D is regular. The language C is not but that requires a proof.

Proving nonregularness

Q: How one can prove that a language is regular?

Proving nonregularness

Q: How one can prove that a language is regular?

Q: How one can prove that a language is NOT regular?

Proving nonregularness

Q: How one can prove that a language is regular?

Q: How one can prove that a language is NOT regular?

We need to find a property that (i) characterizes regular languages; and (ii) can be more or less easily tested for a given language.

Proving nonregularness

Q: How one can prove that a language is regular?

Q: How one can prove that a language is NOT regular?

We need to find a property that (i) characterizes regular languages; and (ii) can be more or less easily tested for a given language.

Unfortunately, non-existence of an NFA recognizing a given language is not easy to test — we need another property. One such useful property is given by the Pumping Lemma.

Pumping Lemma

Lemma

If A is a regular language, then there exists an integer $p > 0$ (called the pumping length) such that for any string $s \in A$ with $|s| \geq p$, the string s can be divided into three pieces: $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$;
2. $|y| > 0$; and
3. $|xy| \leq p$.

Q: What would be a proof outline?

Pumping Lemma

Lemma

If A is a regular language, then there exists an integer $p > 0$ (called the pumping length) such that for any string $s \in A$ with $|s| \geq p$, the string s can be divided into three pieces: $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$;
2. $|y| > 0$; and
3. $|xy| \leq p$.

Q: What would be a proof outline?

We will prove the pumping lemma by construction of p , x , y , and z for a given element $s \in A$ and a DFA N recognizing A , and show that they satisfy conditions 1, 2, and 3.

Proof of the Pumping Lemma: Construction

For a given regular language A , consider a DFA
 $N = (Q, \Sigma, q_0, \delta, F)$ that recognizes A .

Proof of the Pumping Lemma: Construction

For a given regular language A , consider a DFA
 $N = (Q, \Sigma, q_0, \delta, F)$ that recognizes A .

Let $p = |Q|$, i.e., the number of states in N . For a given string $s \in A$ of length $|s| = m \geq p$, consider a computation (r_0, r_1, \dots, r_m) of N that accepts s , i.e.:

$$r_0 \xrightarrow{s_1} r_1 \xrightarrow{s_2} r_2 \xrightarrow{s_3} \cdots \xrightarrow{s_{m-1}} r_{m-1} \xrightarrow{s_m} r_m$$

where $s_1 s_2 \dots s_m = s$, $r_0 = q_0$, and $r_m \in F$.

Proof of the Pumping Lemma: Construction

For a given regular language A , consider a DFA
 $N = (Q, \Sigma, q_0, \delta, F)$ that recognizes A .

Let $p = |Q|$, i.e., the number of states in N . For a given string $s \in A$ of length $|s| = m \geq p$, consider a computation (r_0, r_1, \dots, r_m) of N that accepts s , i.e.:

$$r_0 \xrightarrow{s_1} r_1 \xrightarrow{s_2} r_2 \xrightarrow{s_3} \cdots \xrightarrow{s_{m-1}} r_{m-1} \xrightarrow{s_m} r_m$$

where $s_1 s_2 \dots s_m = s$, $r_0 = q_0$, and $r_m \in F$.

By the *pigeonhole principle*, at least two states among r_0, r_1, \dots, r_p must be equal, i.e., $r_i = r_j$ for some i, j such that $0 \leq i < j \leq p$.

Proof of the Pumping Lemma: Construction

For a given regular language A , consider a DFA
 $N = (Q, \Sigma, q_0, \delta, F)$ that recognizes A .

Let $p = |Q|$, i.e., the number of states in N . For a given string $s \in A$ of length $|s| = m \geq p$, consider a computation (r_0, r_1, \dots, r_m) of N that accepts s , i.e.:

$$r_0 \xrightarrow{s_1} r_1 \xrightarrow{s_2} r_2 \xrightarrow{s_3} \cdots \xrightarrow{s_{m-1}} r_{m-1} \xrightarrow{s_m} r_m$$

where $s_1 s_2 \dots s_m = s$, $r_0 = q_0$, and $r_m \in F$.

By the *pigeonhole principle*, at least two states among r_0, r_1, \dots, r_p must be equal, i.e., $r_i = r_j$ for some i, j such that $0 \leq i < j \leq p$.

We define

$$x = s_1 s_2 \dots s_i, \quad y = s_{i+1} s_{i+2} \dots s_j, \quad z = s_{j+1} s_{j+2} \dots s_m.$$

Proof of the Pumping Lemma: Verification

We need to show that the defined p , x , y , and z satisfy the statement of the Lemma.

First off, we have by constructions:

$$xyz = s_1 s_2 \dots s_i s_{i+1} s_{i+2} \dots s_j s_{j+1} s_{j+2} \dots s_m = s.$$

Proof of the Pumping Lemma: Verification

We need to show that the defined p , x , y , and z satisfy the statement of the Lemma.

First off, we have by construction:

$$xyz = s_1 s_2 \dots s_i s_{i+1} s_{i+2} \dots s_j s_{j+1} s_{j+2} \dots s_m = s.$$

We need to show that

1. for each $i \geq 0$, $xy^i z \in A$

Q: Something is wrong with this statement, isn't it?

Proof of the Pumping Lemma: Verification

We need to show that the defined p , x , y , and z satisfy the statement of the Lemma.

First off, we have by construction:

$$xyz = s_1 s_2 \dots s_i s_{i+1} s_{i+2} \dots s_j s_{j+1} s_{j+2} \dots s_m = s.$$

We need to show that

1. for each $i \geq 0$, $xy^i z \in A$

Q: Something is wrong with this statement, isn't it?

To avoid confusion with i defined in our proof, let us use different symbol in the condition 1:

1. for each $k \geq 0$, $xy^k z \in A$

Q: How to prove that this condition holds?

Proof of the Pumping Lemma: Verification (cont'd)

From a computation for $s = xyz$ on N , we will construct a computation for xy^kz on N for any integer $k \geq 0$.

Here is the computation for s where the equal states r_i and r_j denoted by R :

$$r_0 \xrightarrow{s_1} r_1 \dots r_{i-1} \xrightarrow{s_i} (R \xrightarrow{s_{i+1}} r_{i+1} \dots r_{j-1} \xrightarrow{s_j}) R \xrightarrow{s_{j+1}} q_{j+1} \dots r_{m-1} \xrightarrow{s_m} r_m$$

Note that the part in the parentheses can be repeated k times in a row, using the same transitions multiple times. For example, for $k = 2$, we will have in the middle

$$\dots r_{i-1} \xrightarrow{s_i} (R \xrightarrow{s_{i+1}} r_{i+1} \dots r_{j-1} \xrightarrow{s_j})(R \xrightarrow{s_{i+1}} r_{i+1} \dots r_{j-1} \xrightarrow{s_j}) R \xrightarrow{s_{j+1}} q_{j+1} \dots$$

This way we can construct a computation for xy^kz that have the same starting and ending states as in the computation for s , and using the same transitions but possibly multiple times.

The actual proof is done by induction on k .

Proof of the Pumping Lemma: Verification (cont'd)

Now let us prove the second condition:

2. $|y| > 0$;

Recall that $y = s_{i+1}s_{i+2}\dots s_j$ for some i, j such that $0 \leq i < j \leq p$.

Proof of the Pumping Lemma: Verification (cont'd)

Now let us prove the second condition:

2. $|y| > 0$;

Recall that $y = s_{i+1}s_{i+2}\dots s_j$ for some i, j such that $0 \leq i < j \leq p$.

Finally, we need to prove the third condition:

3. $|xy| \leq p$.

Recall that $x = s_1s_2\dots s_i$ so that $xy = s_1s_2\dots s_is_{i+1}s_{i+2}\dots s_j$.

Proving Nonregularity with Pumping Lemma

The following corollary of Pumping Lemma is often used for proving that a given language is nonregular.

Corollary

A language A is nonregular if for every integer $p > 0$, there exists a string $s \in A$ with $|s| \geq p$ that cannot be divided into three pieces: $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$;
2. $|y| > 0$; and
3. $|xy| \leq p$.

Examples of nonregular languages

Prove that the following languages are not regular:

$$B = \{0^n 1^n \mid n \geq 0\}$$

Examples of nonregular languages

Prove that the following languages are not regular:

$$B = \{0^n 1^n \mid n \geq 0\}$$

$$F = \{ww \mid w \in \{0,1\}^*\}$$

Examples of nonregular languages

Prove that the following languages are not regular:

$$B = \{0^n 1^n \mid n \geq 0\}$$

$$F = \{ww \mid w \in \{0,1\}^*\}$$

$$D = \{1^{n^2} \mid n \geq 0\}$$

Examples of nonregular languages

Prove that the following languages are not regular:

$$B = \{0^n 1^n \mid n \geq 0\}$$

$$F = \{ww \mid w \in \{0,1\}^*\}$$

$$D = \{1^{n^2} \mid n \geq 0\}$$

$$E = \{0^i 1^j \mid i > j\}$$

More Applications

of

the Pumping Lemma

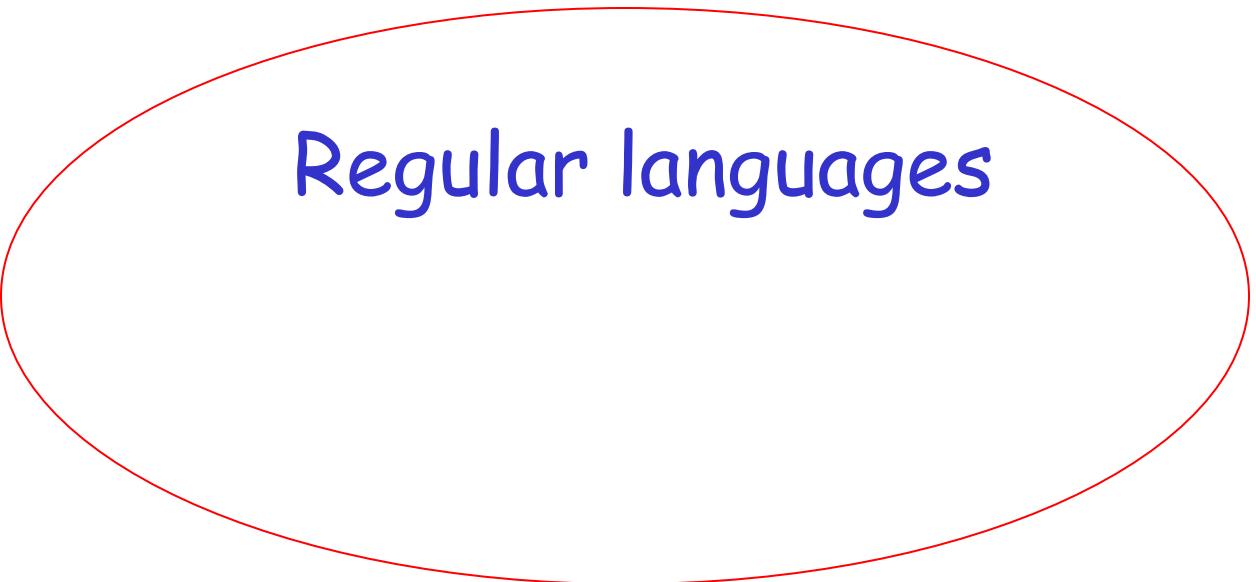
The Pumping Lemma:

- 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

$$L = \{vv^R : v \in \Sigma^*\}$$

Regular languages



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: $w \in L$

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: $|x| \leq m$, $|y| \geq 1$

$$w = xyz = \overbrace{a \dots aa \dots a \dots ab \dots bb \dots ba \dots a}^m \underbrace{\qquad\qquad\qquad}_{x} \underbrace{\qquad\qquad\qquad}_{y} \underbrace{\qquad\qquad\qquad}_{z}$$

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: $x \ y^2 \ z \in L$

$$xy^2z = \underbrace{a \dots aa \dots aa \dots a}_{x} \underbrace{\dots}_{y} \underbrace{\dots}_{y} \underbrace{ab \dots bb \dots ba \dots a}_{z} \underbrace{\dots}_{m+k} \underbrace{\dots}_{m} \underbrace{\dots}_{m} \underbrace{\dots}_{m} \in L$$

Thus: $a^{m+k} b^m b^m a^m \in L$

$$a^{m+k} b^m b^m a^m \in L \quad k \geq 1$$

BUT: $L = \{vv^R : v \in \Sigma^*\}$



$$a^{m+k} b^m b^m a^m \notin L$$

CONTRADICTION!!!

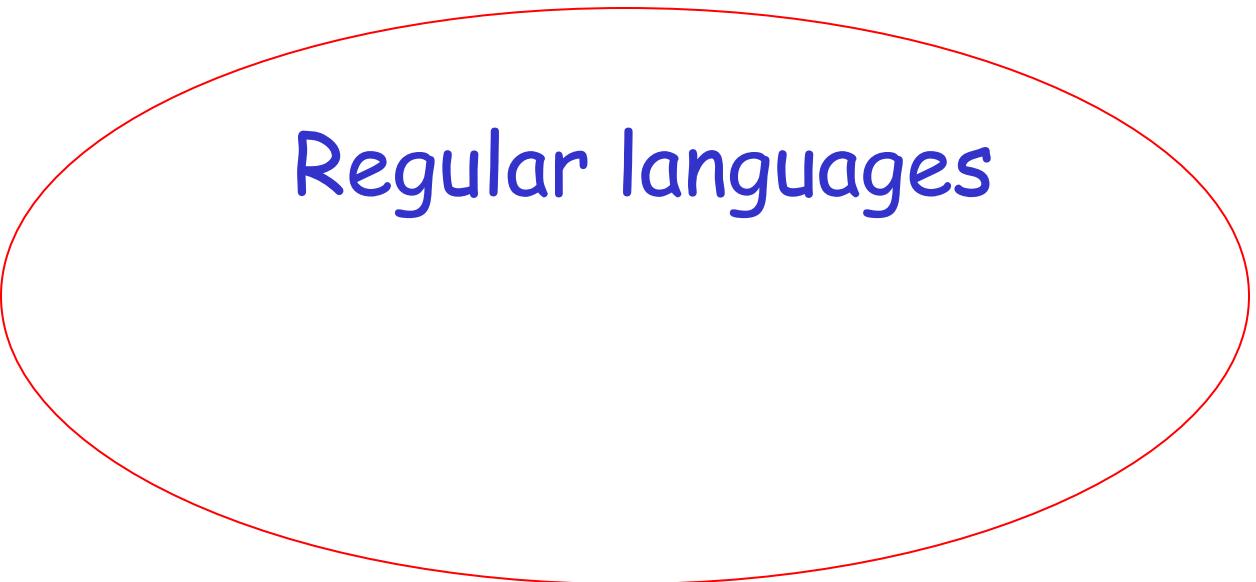
Therefore: Our assumption that L is a regular language is not true

Conclusion: L is not a regular language

END OF PROOF

Non-regular languages

$$L = \{a^n b^l c^{n+l} : n, l \geq 0\}$$



Regular languages

Theorem: The language

$$L = \{a^n b^l c^{n+l} : n, l \geq 0\}$$

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

$$L = \{a^n b^l c^{n+l} : n, l \geq 0\}$$

Let m be the critical length of L

Pick a string w such that: $w \in L$ and

length $|w| \geq m$

We pick $w = a^m b^m c^{2m}$

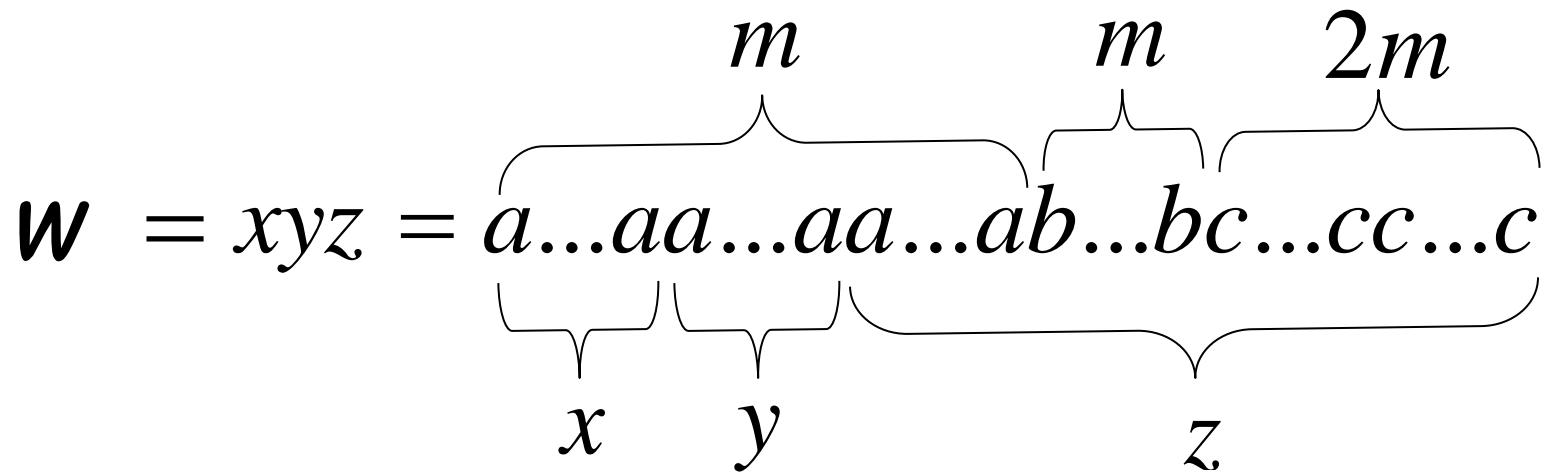
From the Pumping Lemma:

We can write $w = a^m b^m c^{2m} = xyz$

With lengths $|xy| \leq m$, $|y| \geq 1$

$$w = xyz = \underbrace{a \dots aa \dots aa}_{x} \dots \underbrace{ab \dots bc \dots cc \dots c}_{y} \dots \underbrace{c}_{z}$$

m m $2m$



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: $xz \in L$

$$xz = a \dots aa \dots ab \dots bc \dots cc \dots c \in L$$

$m-k$ m $2m$

x z

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!!!

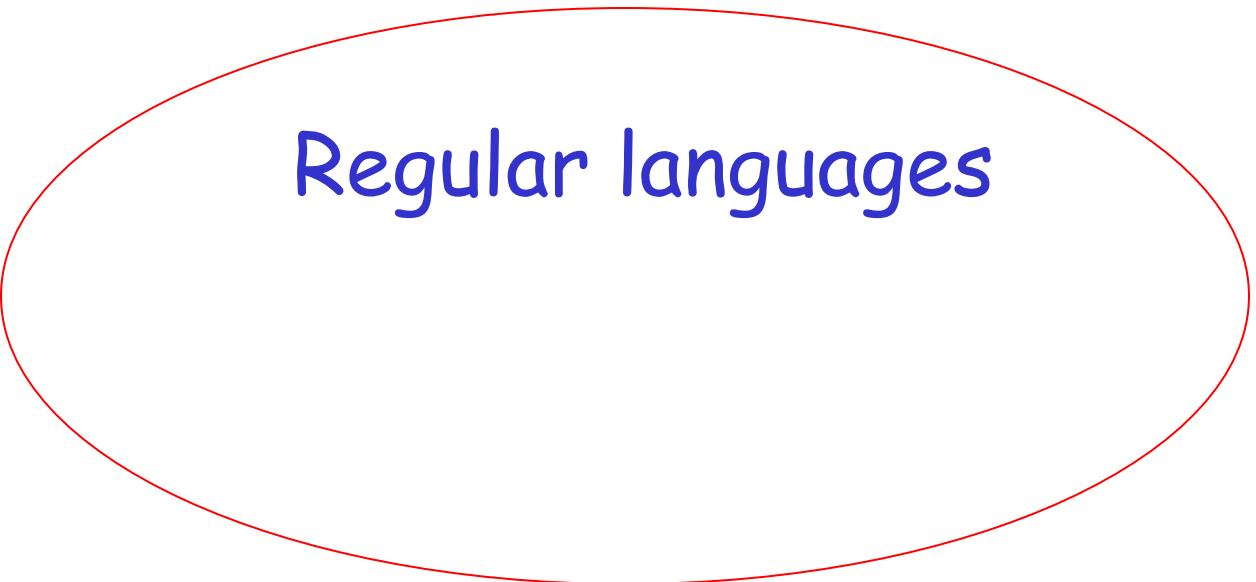
Therefore: Our assumption that L
is a regular language is not true

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 $|x| \leq m$, $|y| \geq 1$

$$w = xyz = a^{m!} = \overbrace{a \dots a}^m \underbrace{a \dots a}_{x} \underbrace{a \dots a}_{y} \underbrace{a \dots a}_{z} \overbrace{a \dots a}^{m!-m}$$

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: $x \ y^2 \ z \in L$

$$xy^2z = \overbrace{a \dots aa \dots aa \dots aa \dots aa \dots aa \dots a}^{m+k} \in L$$

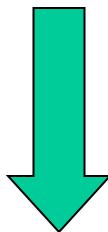
$m + k$
 $m! - m$

$x \quad y \quad y \quad z$

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:

$$m!+k \leq m!+m \quad \text{for } m > 1$$

$$\leq m!+m!$$

$$< m!m + m!$$

$$= m!(m+1)$$

$$= (m+1)!$$



$$m!+k < (m+1)!$$



$$m!+k \neq p! \quad \text{for any } p$$

$$a^{m!+k} \in L \quad 1 \leq k \leq m$$

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

Context-Free Languages

Context-Free Languages

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

$\{ww^R\}$

Regular Languages

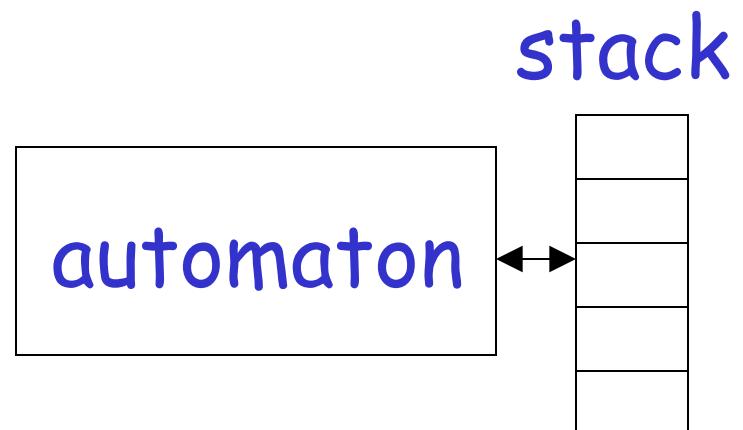
$a^* b^*$

$(a + b)^*$

Context-Free Languages

Context-Free
Grammars

Pushdown
Automata



Context-Free Grammars

Grammars

Grammars express languages

Example: the English language grammar

$$\langle \textit{sentence} \rangle \rightarrow \langle \textit{noun_phrase} \rangle \langle \textit{predicate} \rangle$$
$$\langle \textit{noun_phrase} \rangle \rightarrow \langle \textit{article} \rangle \langle \textit{noun} \rangle$$
$$\langle \textit{predicate} \rangle \rightarrow \langle \textit{verb} \rangle$$

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

Derivation of string “the dog walks”:

$\langle sentence \rangle \Rightarrow \langle noun_phrase \rangle \langle predicate \rangle$
 $\Rightarrow \langle noun_phrase \rangle \langle verb \rangle$
 $\Rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle$
 $\Rightarrow the \langle noun \rangle \langle verb \rangle$
 $\Rightarrow the \ dog \langle verb \rangle$
 $\Rightarrow the \ dog \ sleeps$

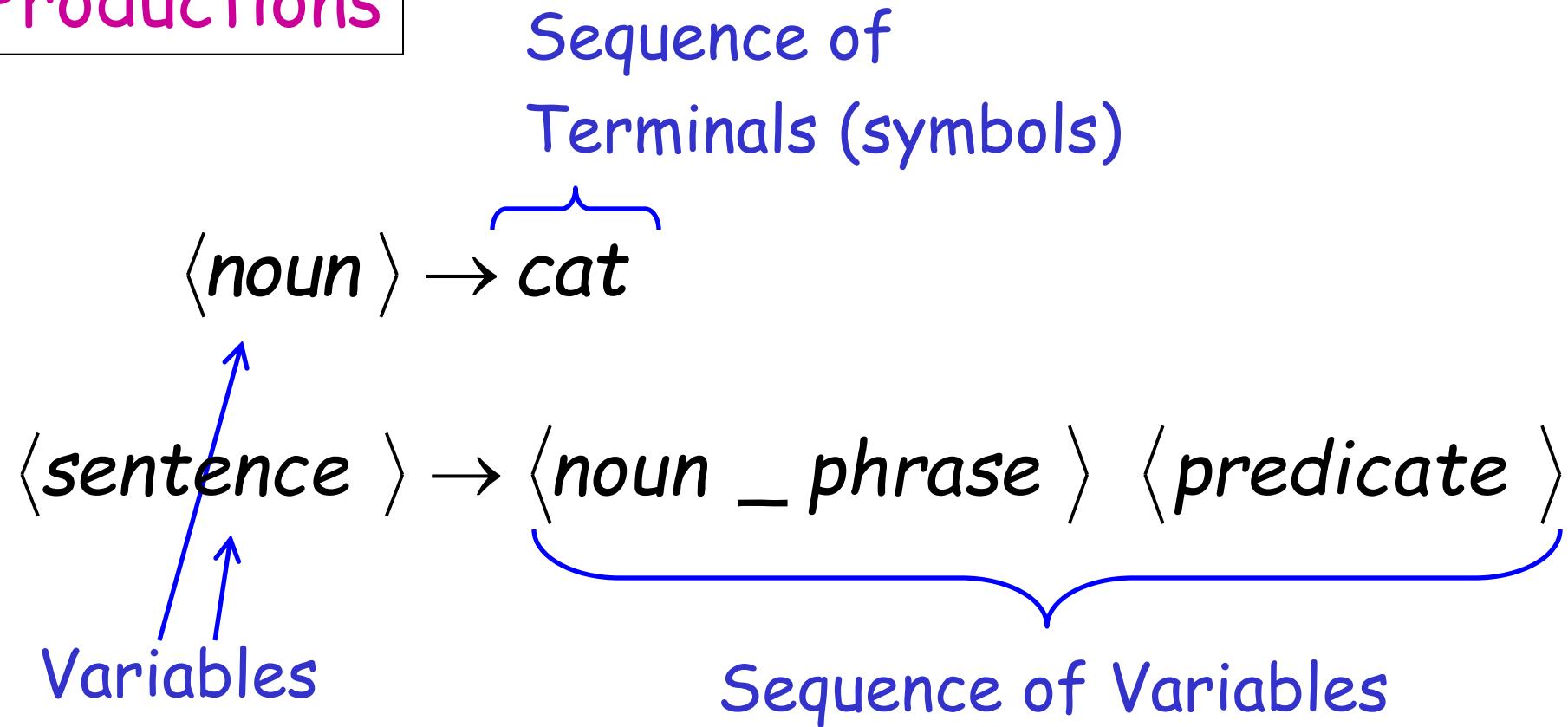
Derivation of string “a cat runs”:

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

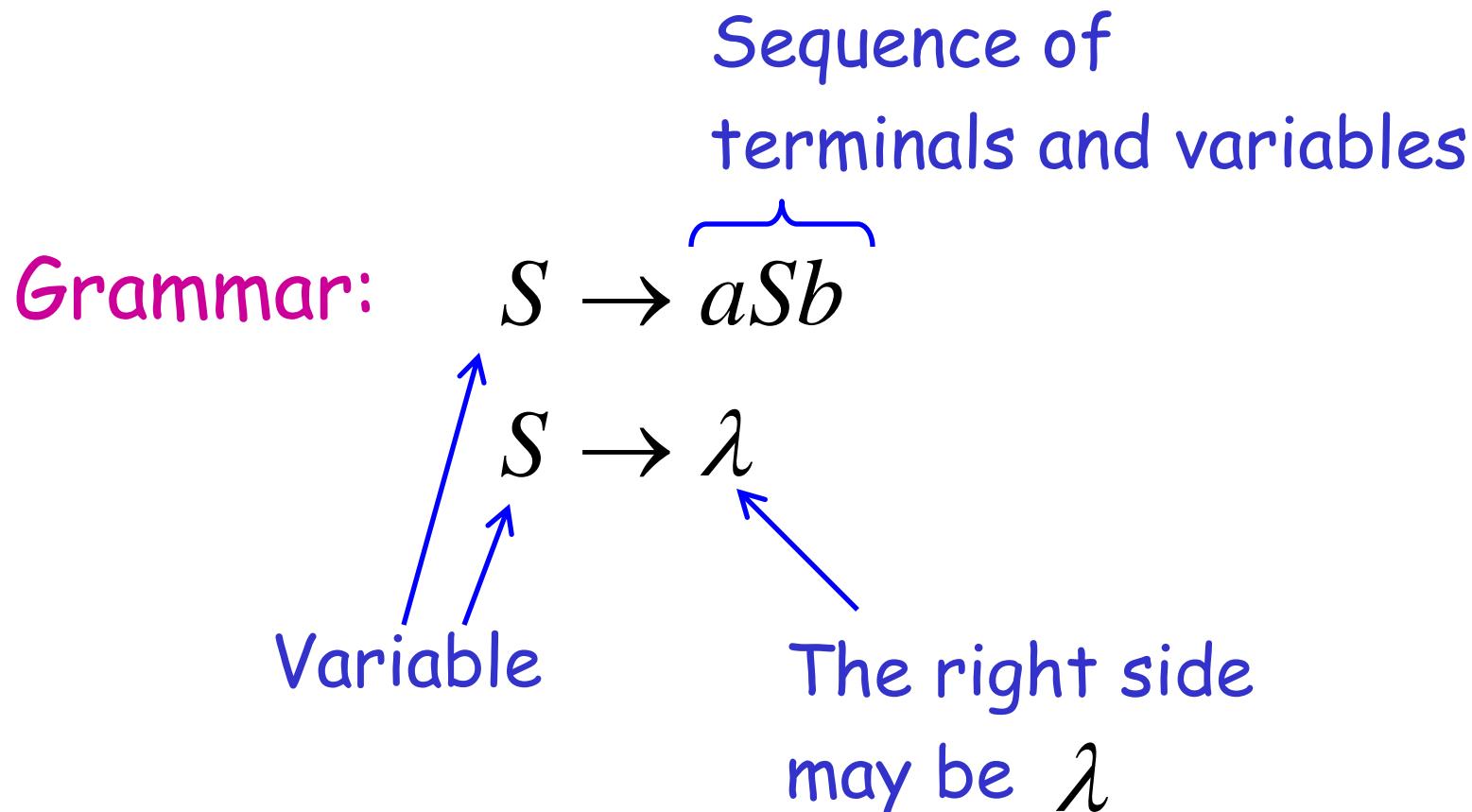
Language of the grammar:

$L = \{$ "a cat runs",
"a cat sleeps",
"the cat runs",
"the cat sleeps",
"a dog runs",
"a dog sleeps",
"the dog runs",
"the dog sleeps" $\}$

Productions



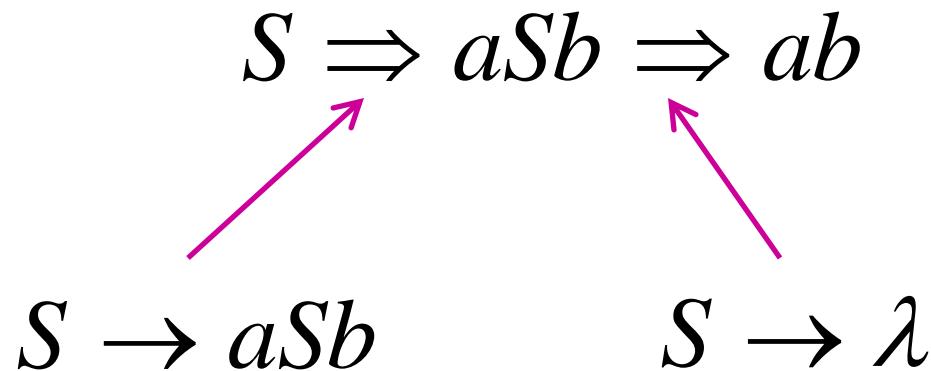
Another Example



Grammar: $S \rightarrow aSb$

$S \rightarrow \lambda$

Derivation of string ab :



Grammar: $S \rightarrow aSb$

$S \rightarrow \lambda$

Derivation of string $aabb$:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$



$S \rightarrow aSb$

$S \rightarrow \lambda$

Grammar: $S \rightarrow aSb$

$S \rightarrow \lambda$

Other derivations:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabb$

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

$\Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$

Grammar: $S \rightarrow aSb$

$S \rightarrow \lambda$

Language of the grammar:

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

A Convenient Notation

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

for zero or more derivation steps

Instead of:

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

*

In general we write: $w_1 \Rightarrow w_n$

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

in zero or more derivation steps

*

Trivially: $w \Rightarrow w$

Example Grammar

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

Possible Derivations

$$S \xrightarrow{*} \lambda$$

$$S \xrightarrow{*} ab$$

$$S \xrightarrow{*} aaabbb$$

$$S \xrightarrow{*} aaSbb \xrightarrow{*} aaaaSbbbb b$$

Another convenient notation:

$$\begin{array}{c} S \rightarrow aSb \\ S \rightarrow \lambda \end{array} \quad \longrightarrow \quad S \rightarrow aSb \mid \lambda$$

$$\begin{array}{c} \langle \text{article} \rangle \rightarrow a \\ \langle \text{article} \rangle \rightarrow \text{the} \end{array} \quad \longrightarrow \quad \langle \text{article} \rangle \rightarrow a \mid \text{the}$$

Formal Definitions

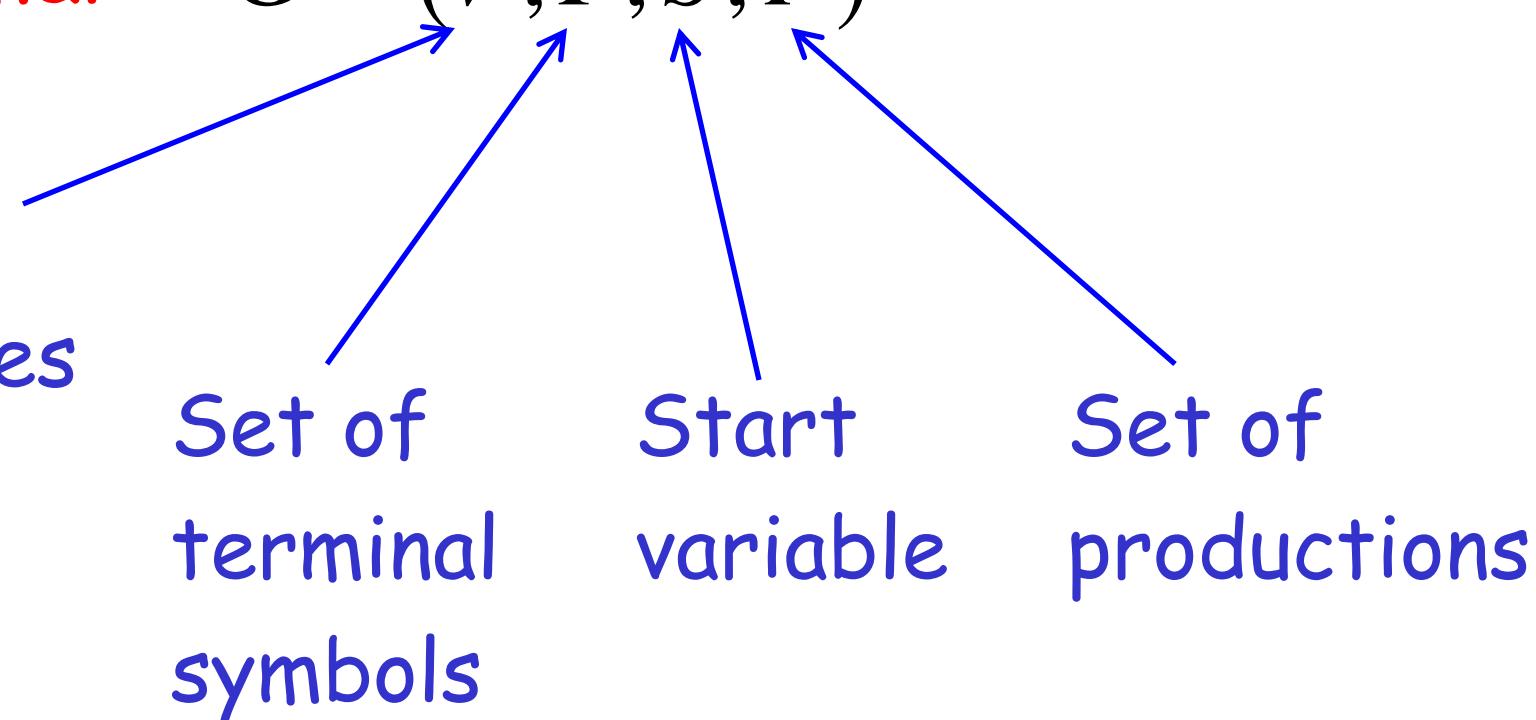
Grammar: $G = (V, T, S, P)$

Set of
variables

Set of
terminal
symbols

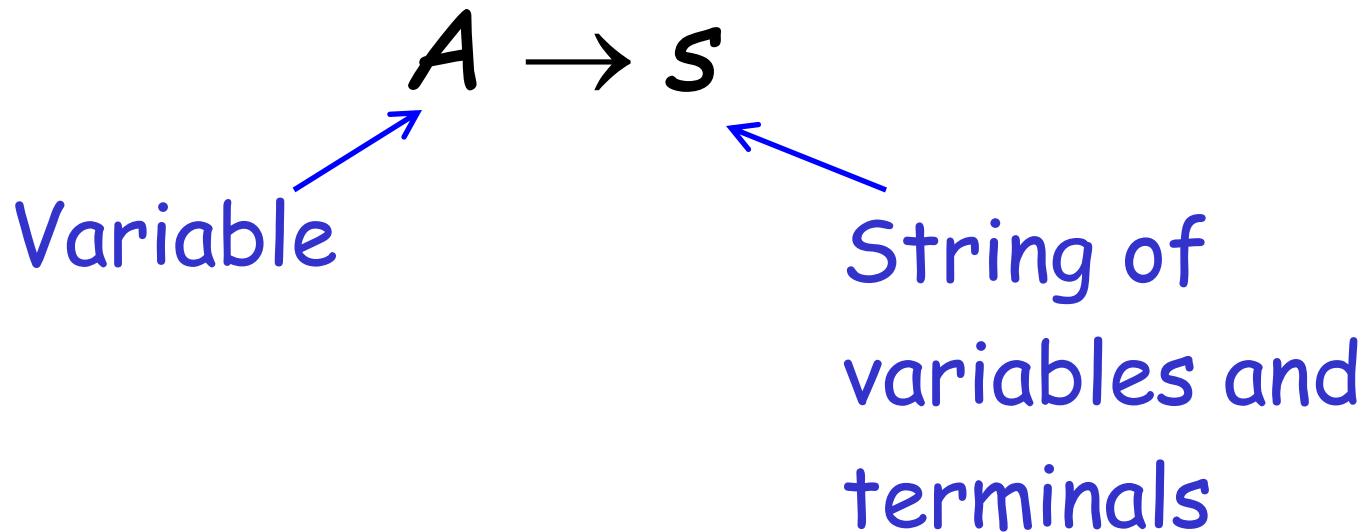
Start
variable

Set of
productions



Context-Free Grammar: $G = (V, T, S, P)$

All productions in P are of the form



Example of Context-Free Grammar

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

productions

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

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

$V = \{S\}$
variables

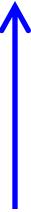
$T = \{a, b\}$
terminals

start variable

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 λ

Example:

context-free grammar G :

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

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

Since, there is derivation

$$S \xrightarrow{*} a^n b^n \quad \text{for any } n \geq 0$$

Context-Free Language:

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 :

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

generates $L(G) = L$

Another Example

Context-free grammar G :

$$S \rightarrow aSa \mid bSb \mid \lambda$$

Example derivations:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaba$$

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

Palindromes of even length

Another Example

Context-free grammar G :

$$S \rightarrow aSb \mid SS \mid \lambda$$

Example derivations:

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow ab$$

$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$$

$$L(G) = \{w : n_a(w) = n_b(w), \text{ and } n_a(v) \geq n_b(v)$$

Describes
matched

parentheses: $() ((()))) (())$ $a = (,$ $b =)$

Derivation Order and Derivation Trees

Derivation Order

Consider the following example grammar
with 5 productions:

$$1. S \rightarrow AB \quad 2. A \rightarrow aaA \quad 4. B \rightarrow Bb$$

$$3. A \rightarrow \lambda \quad 5. B \rightarrow \lambda$$

- | | | |
|----------------------------|------------------------|----------------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow Bb$ |
| 3. $A \rightarrow \lambda$ | | 5. $B \rightarrow \lambda$ |

Leftmost derivation order of string aab :

$$\begin{array}{ccccccccc}
 & 1 & & 2 & & 3 & & 4 & & 5 \\
 S \Rightarrow & AB \Rightarrow & aaAB \Rightarrow & aaB \Rightarrow & aaBb \Rightarrow & aab
 \end{array}$$

At each step, we substitute the leftmost variable

- | | | |
|----------------------------|------------------------|----------------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow Bb$ |
| 3. $A \rightarrow \lambda$ | | 5. $B \rightarrow \lambda$ |

Rightmost derivation order of string aab :

$$S \xrightarrow{1} AB \xrightarrow{4} ABb \xrightarrow{5} Ab \xrightarrow{2} aaAb \xrightarrow{3} aab$$

At each step, we substitute the rightmost variable

- | | | |
|-----------------------|----------------------------|----------------------------|
| 1. $S \rightarrow AB$ | 2. $A \rightarrow aaA$ | 4. $B \rightarrow Bb$ |
| | 3. $A \rightarrow \lambda$ | 5. $B \rightarrow \lambda$ |

Leftmost derivation of aab :

$$\begin{array}{ccccc}
 1 & & 2 & & 3 & & 4 & & 5 \\
 S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aab
 \end{array}$$

Rightmost derivation of aab :

$$\begin{array}{ccccc}
 1 & & 4 & & 5 & & 2 & & 3 \\
 S \Rightarrow AB \Rightarrow ABb \Rightarrow Ab \Rightarrow aaAb \Rightarrow aab
 \end{array}$$

Derivation Trees

Consider the same example grammar:

$$S \rightarrow AB \quad A \rightarrow aaA \mid \lambda \quad B \rightarrow Bb \mid \lambda$$

And a derivation of aab :

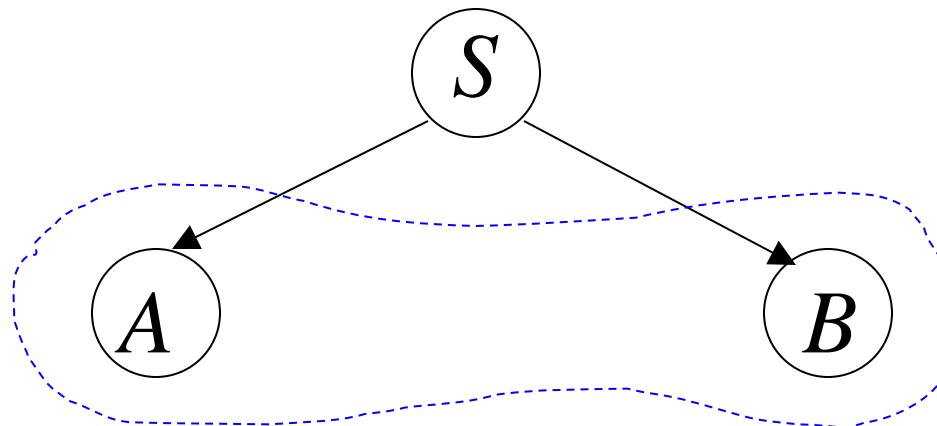
$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB$$



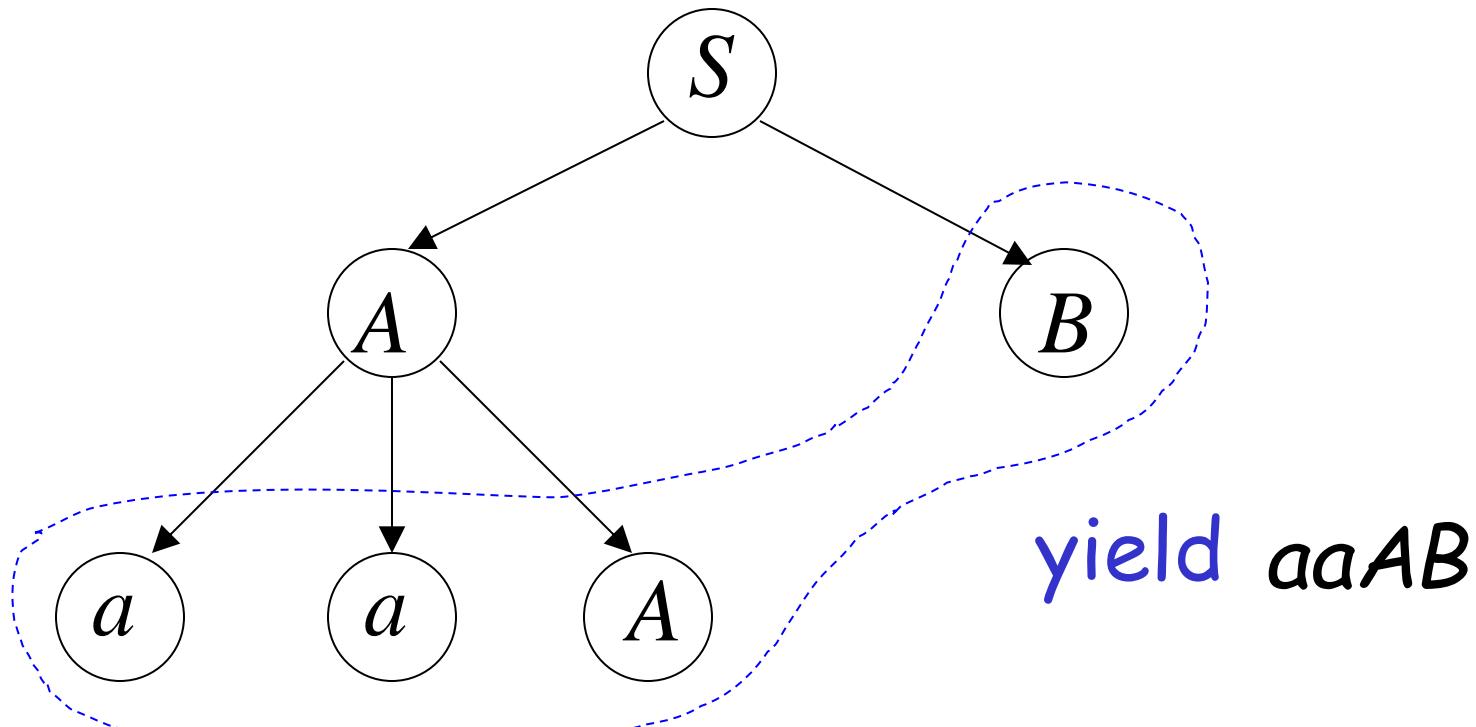
yield *AB*

$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB$$

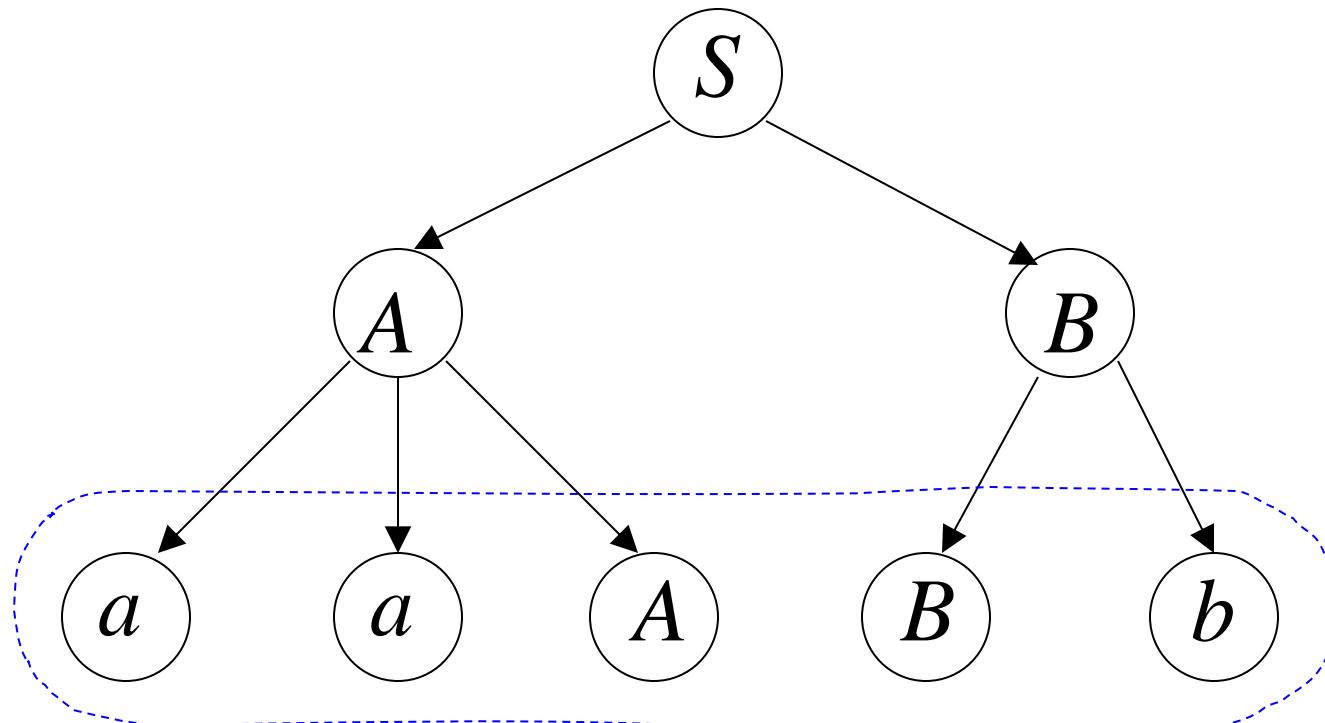


$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb$$



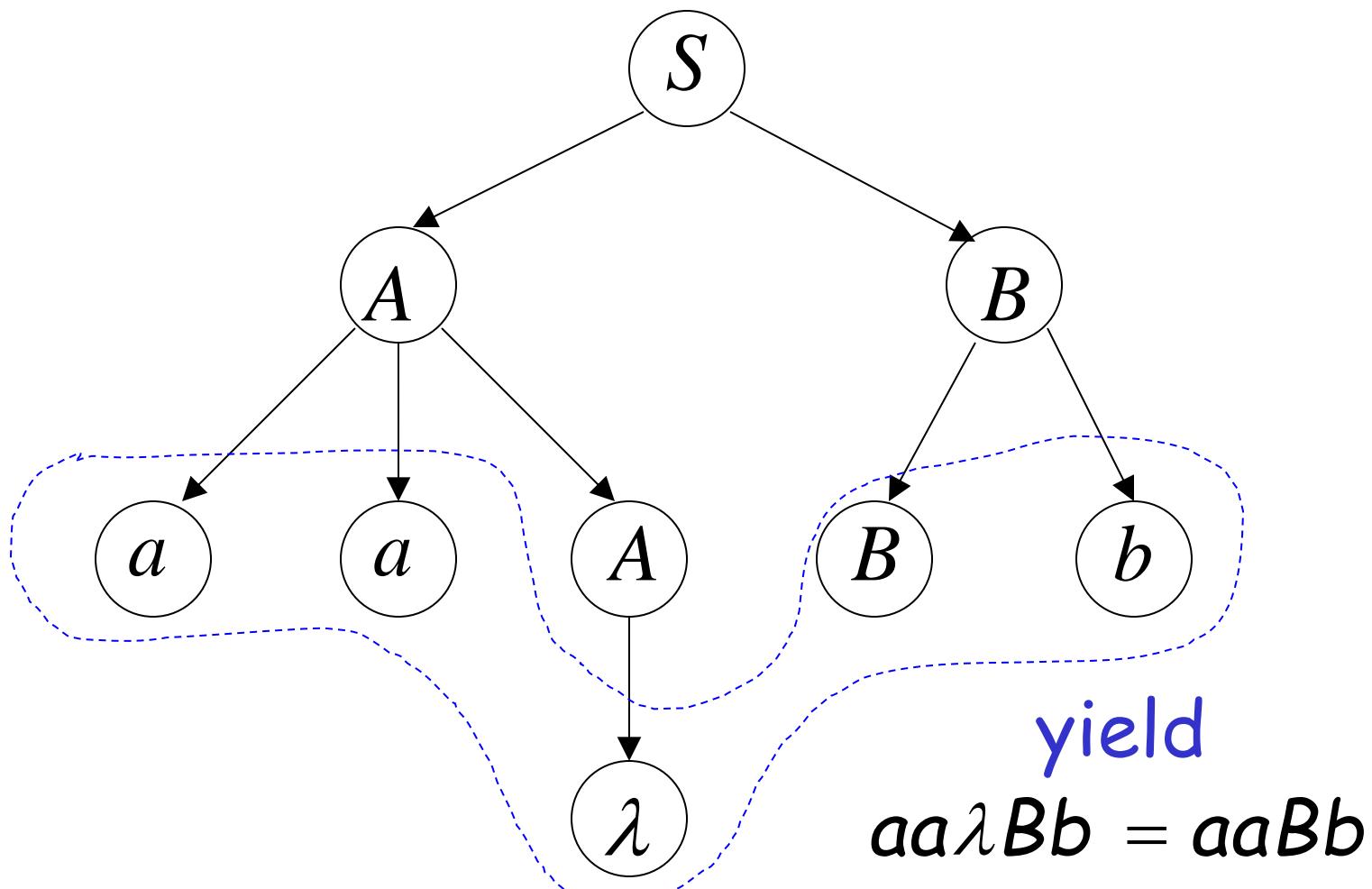
yield *aaABb*

$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb$$



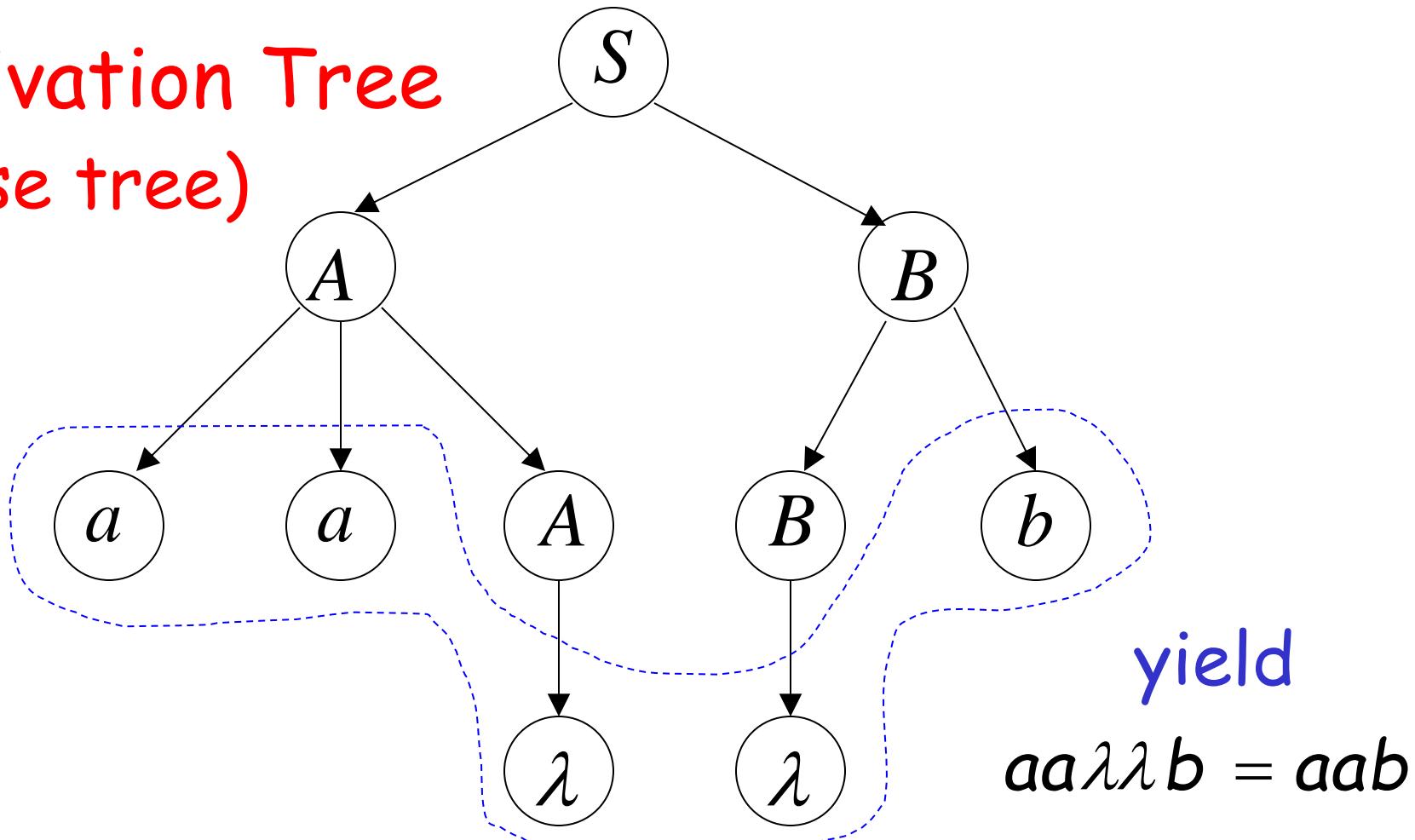
$$S \rightarrow AB$$

$$A \rightarrow aaA \mid \lambda$$

$$B \rightarrow Bb \mid \lambda$$

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$$

Derivation Tree
(parse tree)



Sometimes, derivation order doesn't matter

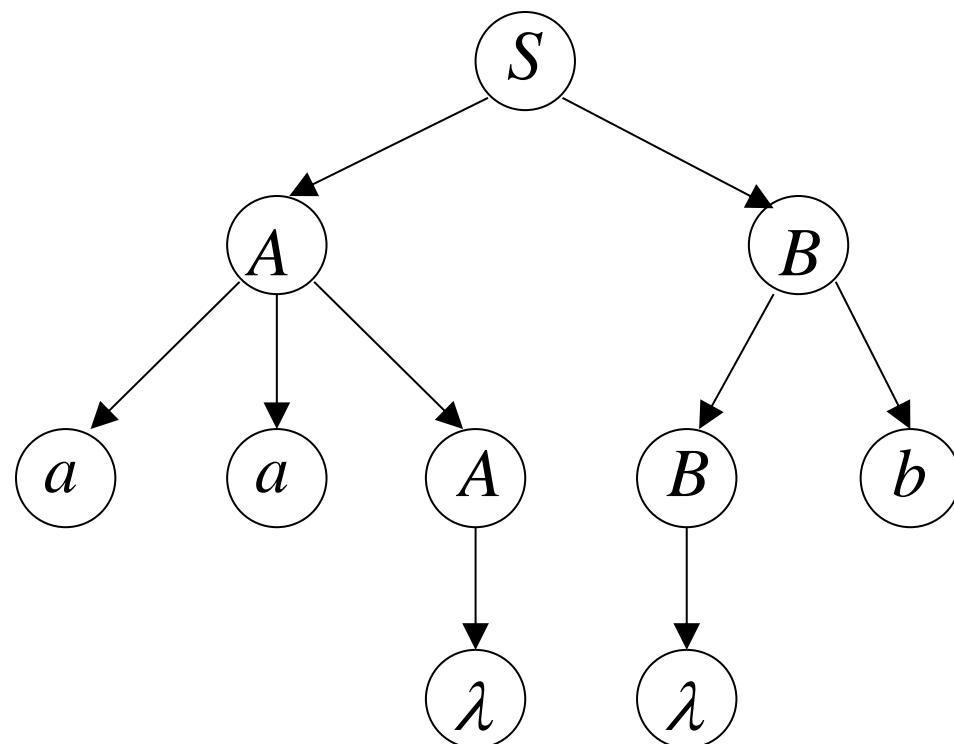
Leftmost derivation:

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aab$$

Rightmost derivation:

$$S \Rightarrow AB \Rightarrow ABb \Rightarrow Ab \Rightarrow aaAb \Rightarrow aab$$

Give same
derivation tree



Ambiguity

Grammar for mathematical expressions

$$E \rightarrow E + E \quad | \quad E * E \quad | \quad (E) \quad | \quad a$$

Example strings:

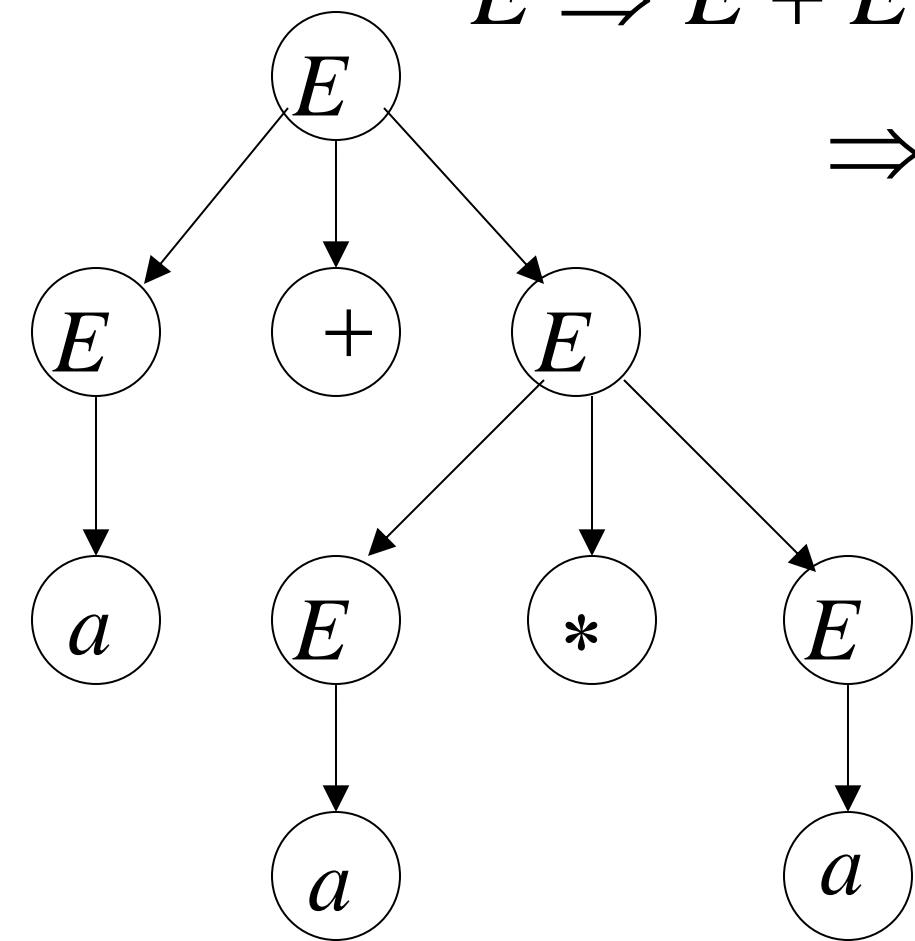
$$(a + a) * a + (a + a * (a + a))$$



Denotes any number

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \\ &\Rightarrow a + a * E \Rightarrow a + a * a \end{aligned}$$

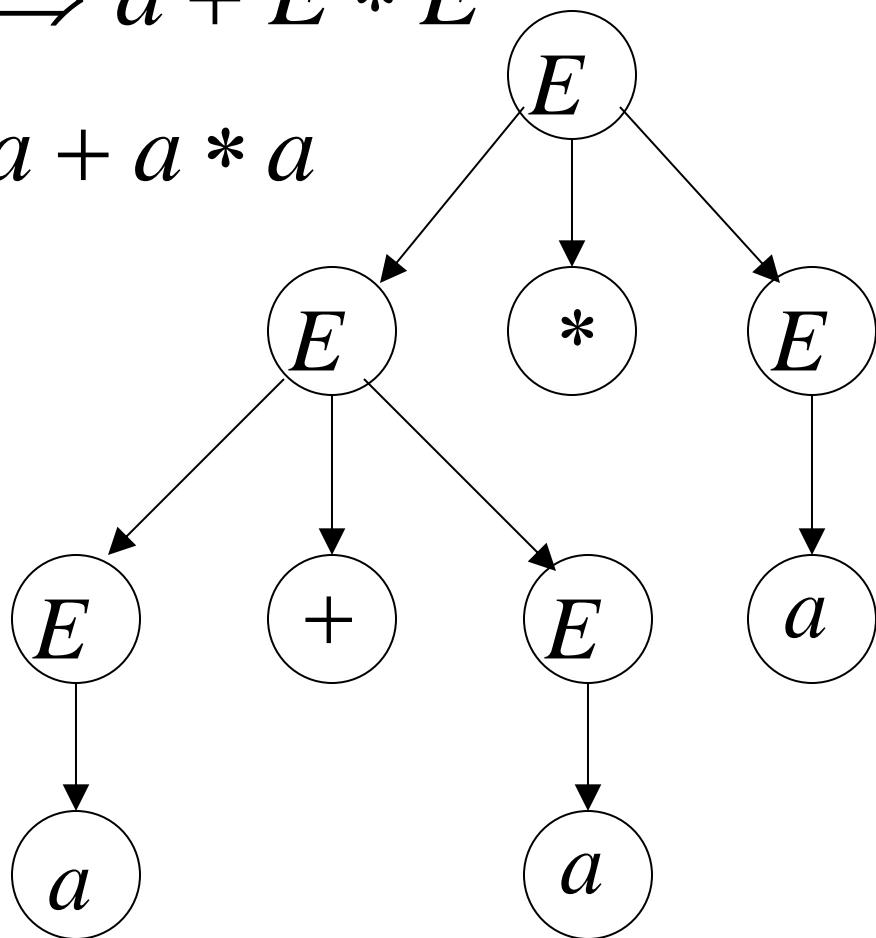


A leftmost derivation
for $a + a * a$

$$E \rightarrow E + E \quad | \quad E * E \quad | \quad (E) \quad | \quad a$$

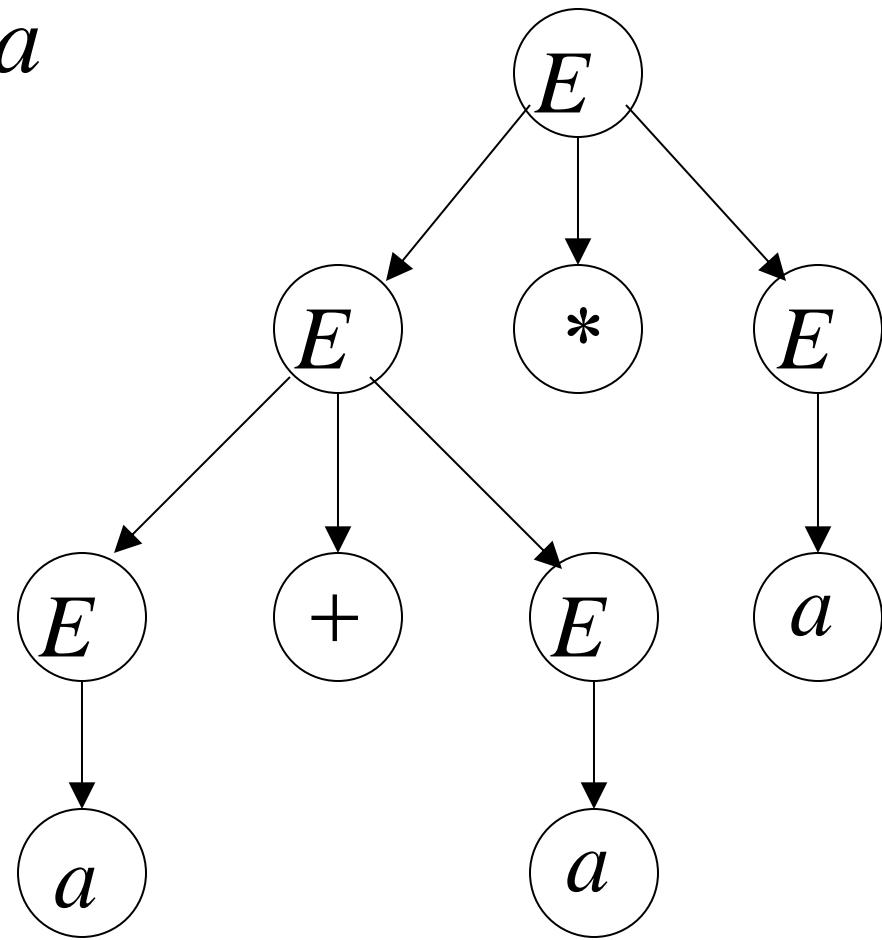
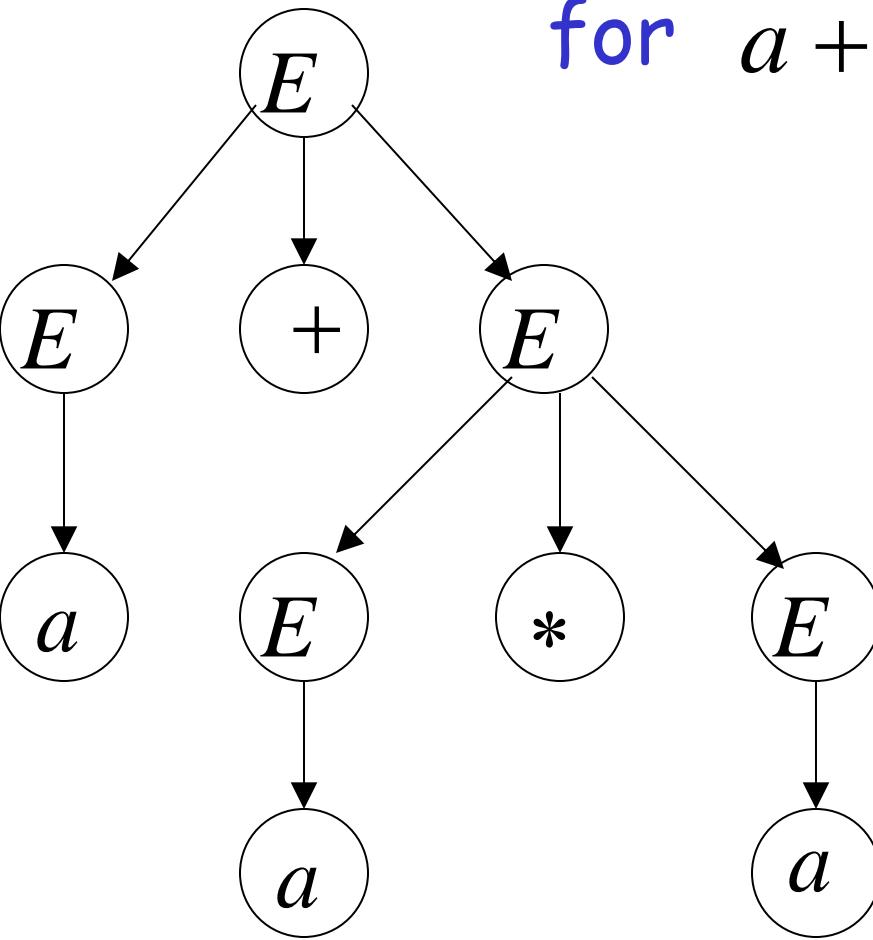
$$\begin{aligned} E &\Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \\ &\Rightarrow a + a * E \Rightarrow a + a * a \end{aligned}$$

Another
leftmost derivation
for $a + a * a$



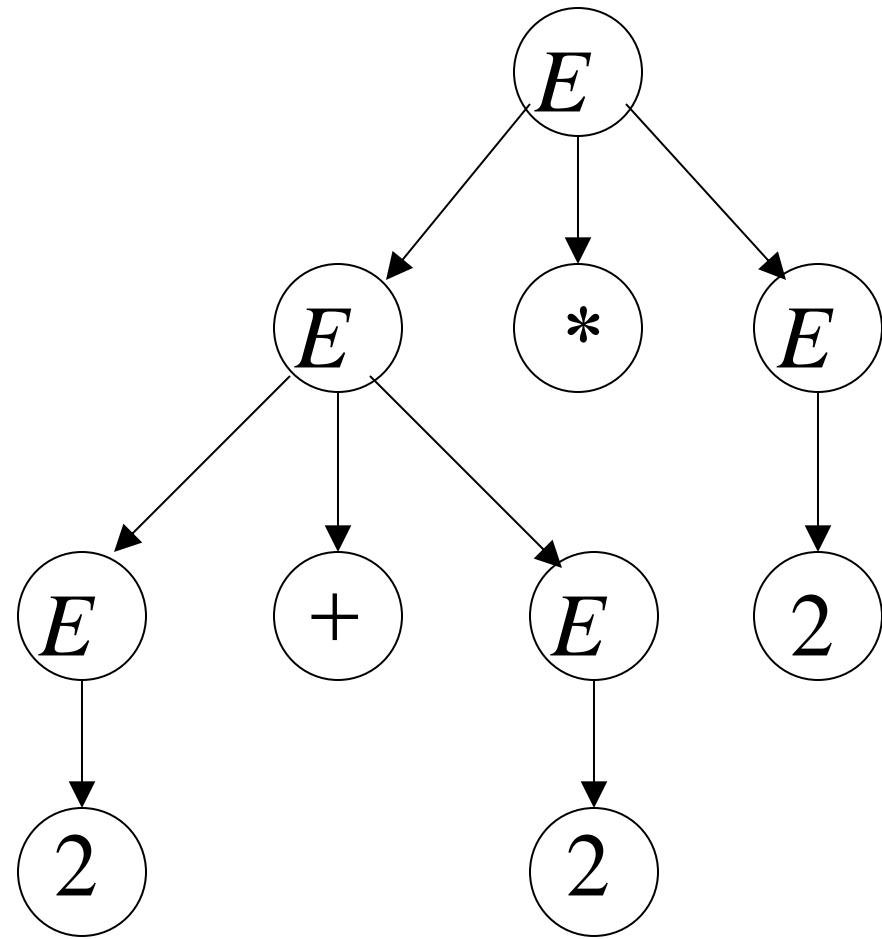
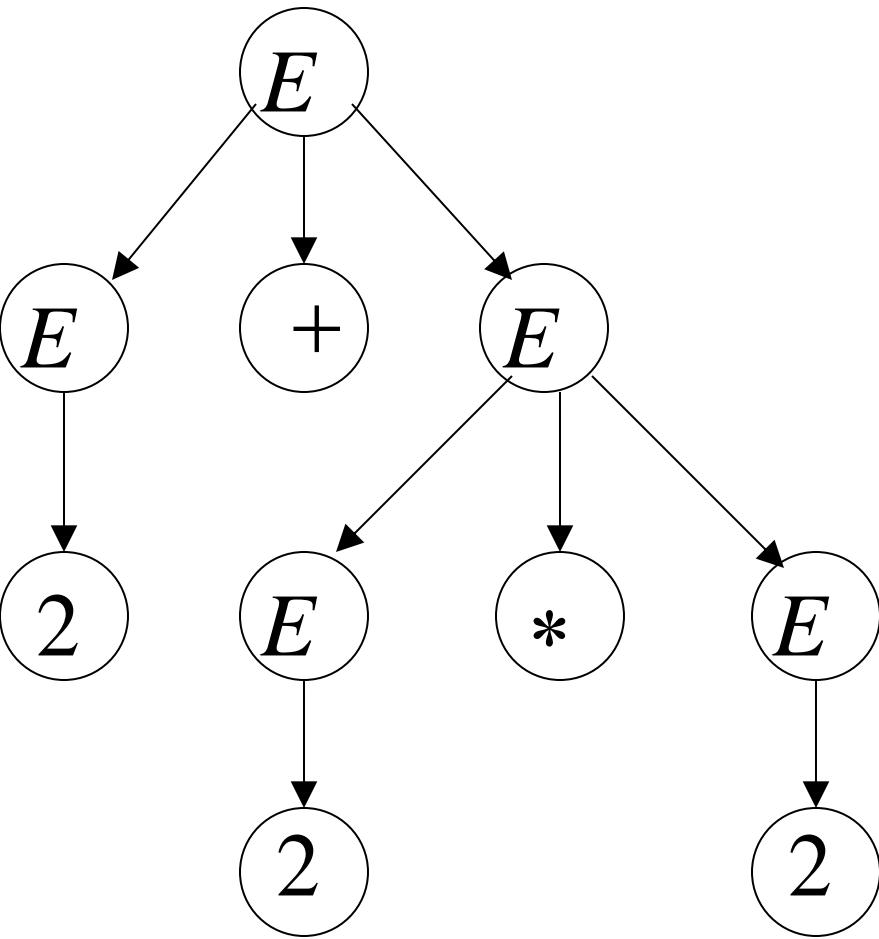
$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

Two derivation trees
for $a + a * a$



take $a = 2$

$$a + a * a = 2 + 2 * 2$$



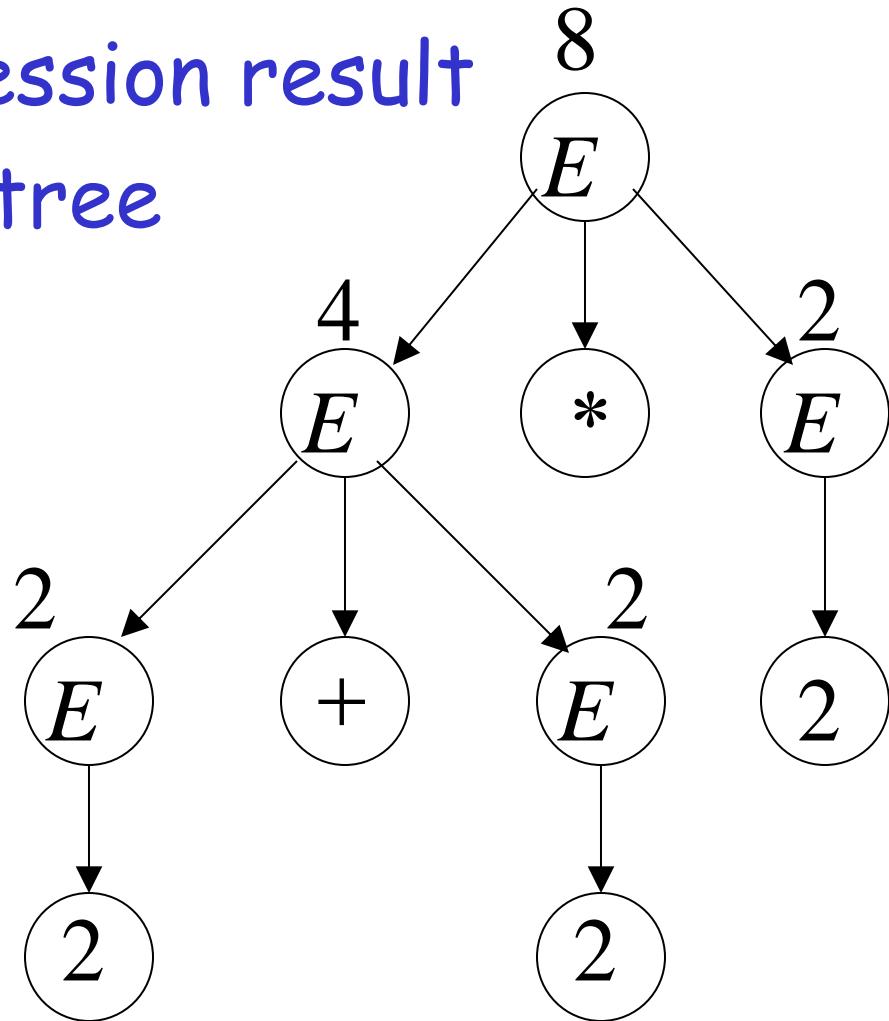
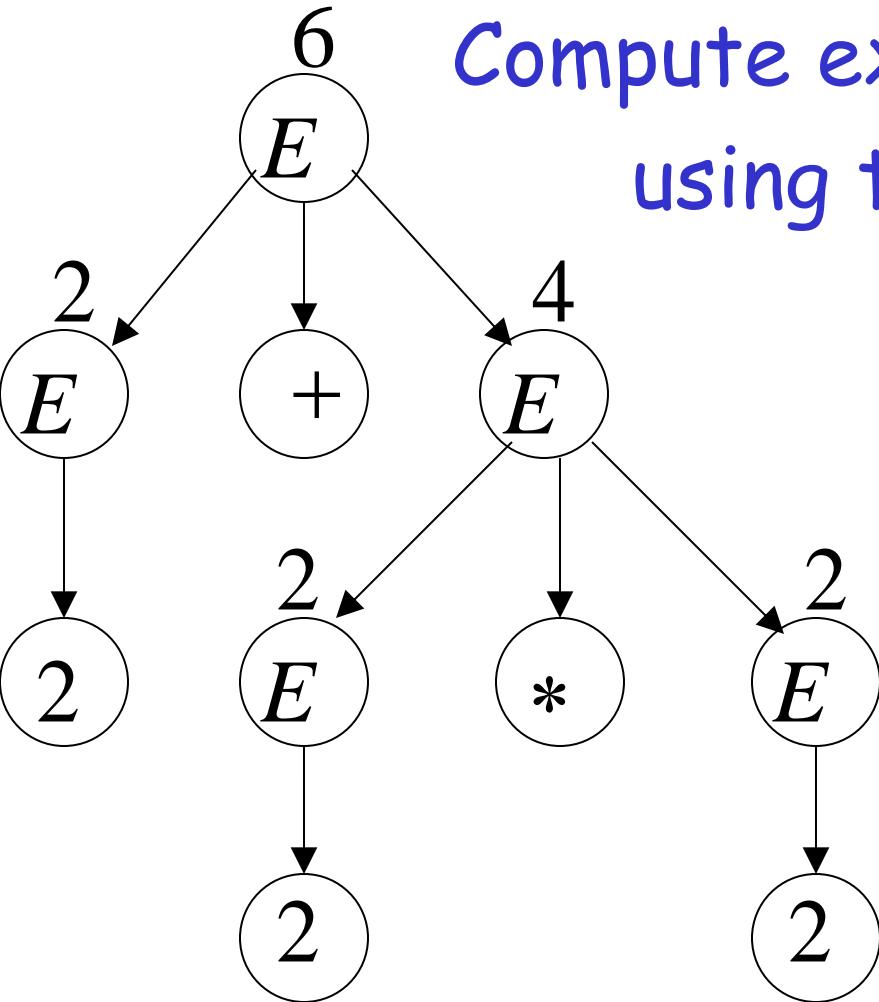
Good Tree

$$2 + 2 * 2 = 6$$

Bad Tree

$$2 + 2 * 2 = 8$$

Compute expression result
using the tree



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:

A context-free grammar G is ambiguous if there is a string $w \in L(G)$ which has:

two different derivation trees

or

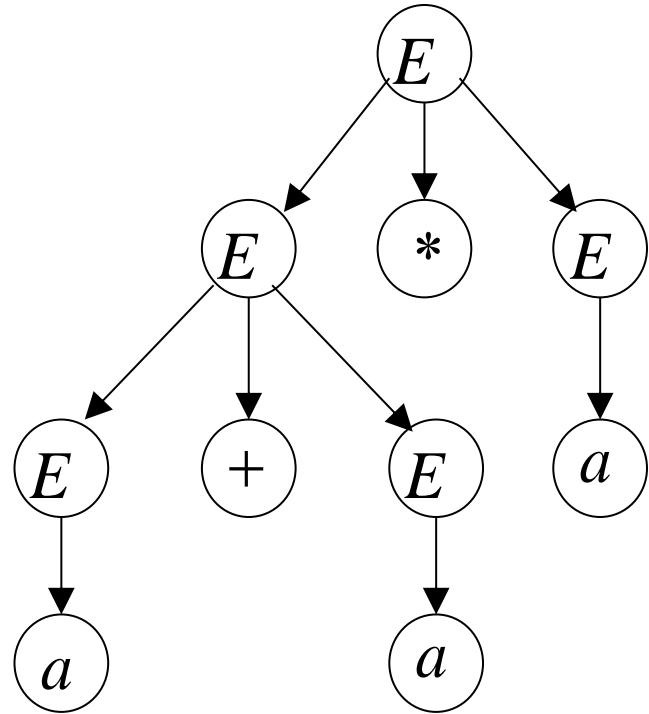
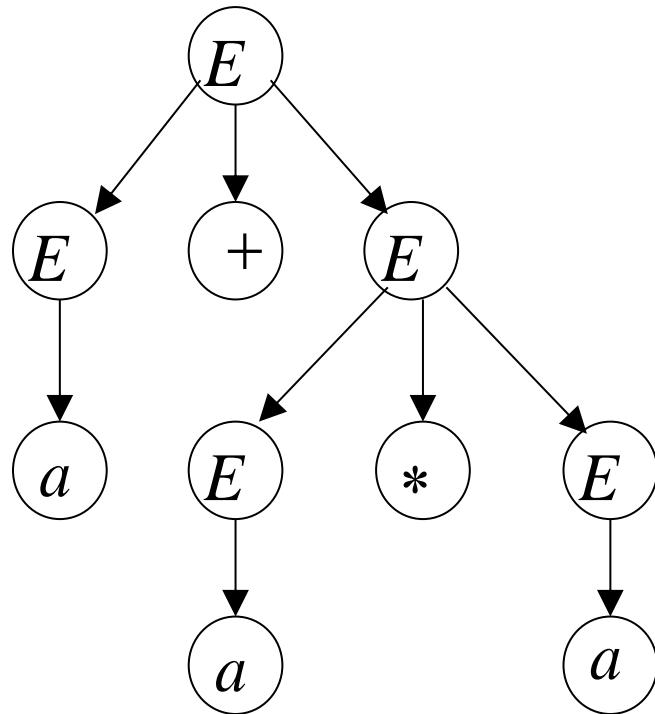
two leftmost derivations

(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



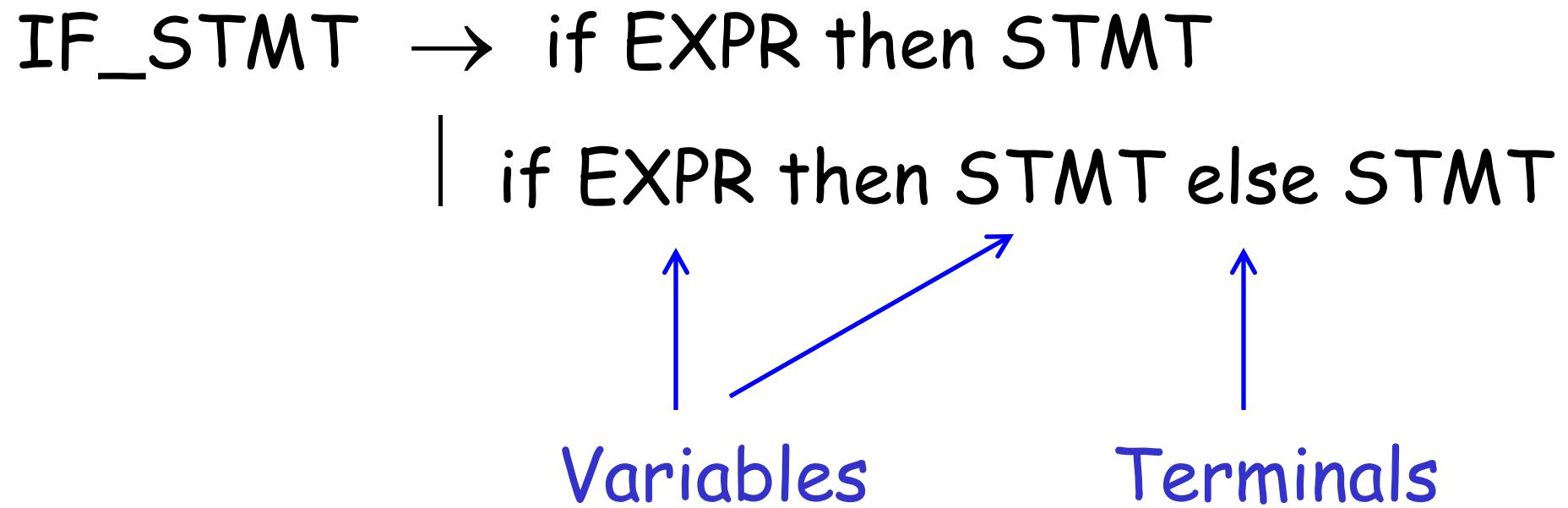
$$E \rightarrow E + E \quad | \quad E * E \quad | \quad (E) \quad | \quad a$$

this grammar is ambiguous also because
string $a + a * a$ has two leftmost derivations

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \\ &\qquad\qquad\qquad\Rightarrow a + a * E \Rightarrow a + a * a \end{aligned}$$

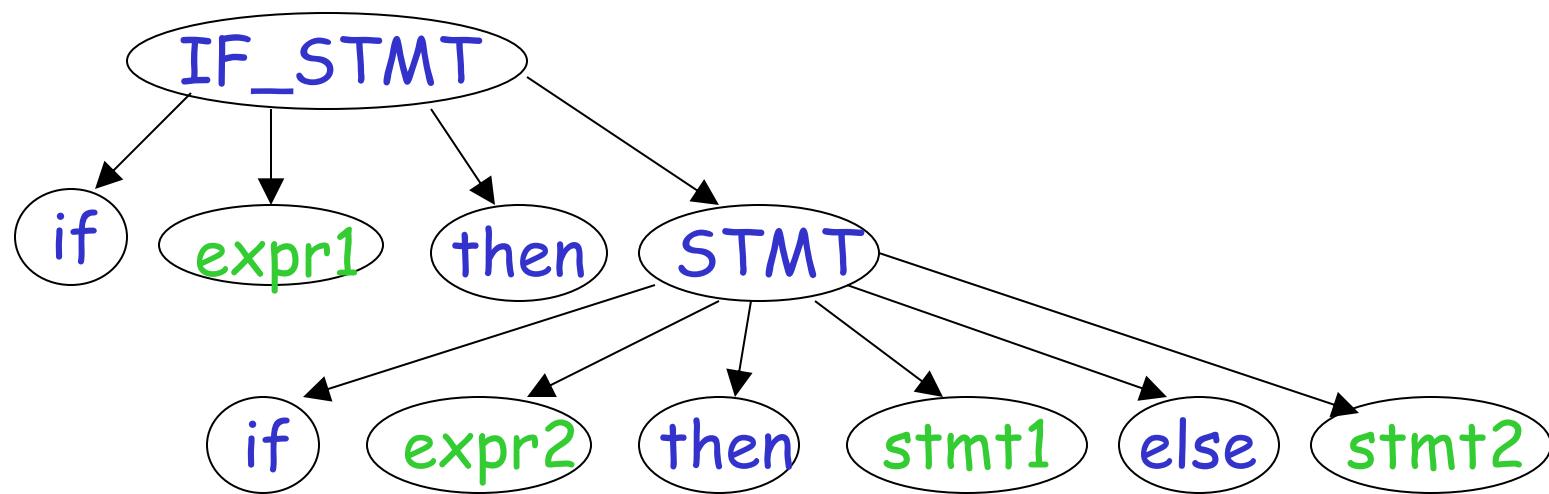
$$\begin{aligned} E &\Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \\ &\qquad\qquad\qquad\Rightarrow a + a * E \Rightarrow a + a * a \end{aligned}$$

Another ambiguous grammar:

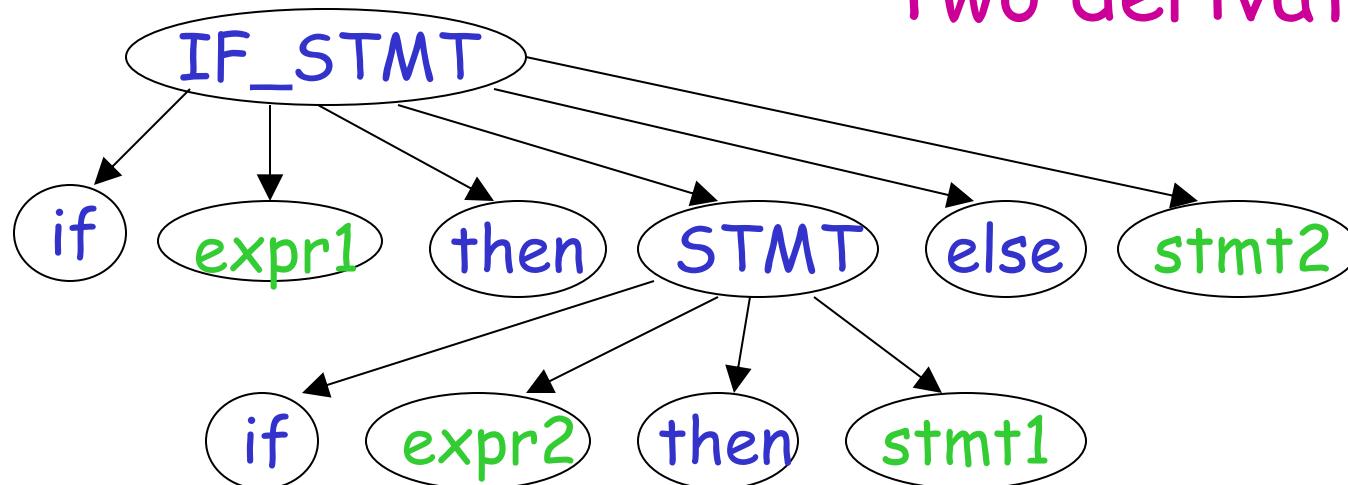


Very common piece of grammar
in programming languages

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

```
 $E \rightarrow E + E$ 
 $E \rightarrow E * E$ 
 $E \rightarrow (E)$ 
 $E \rightarrow a$ 
```

Equivalent Non-Ambiguous Grammar

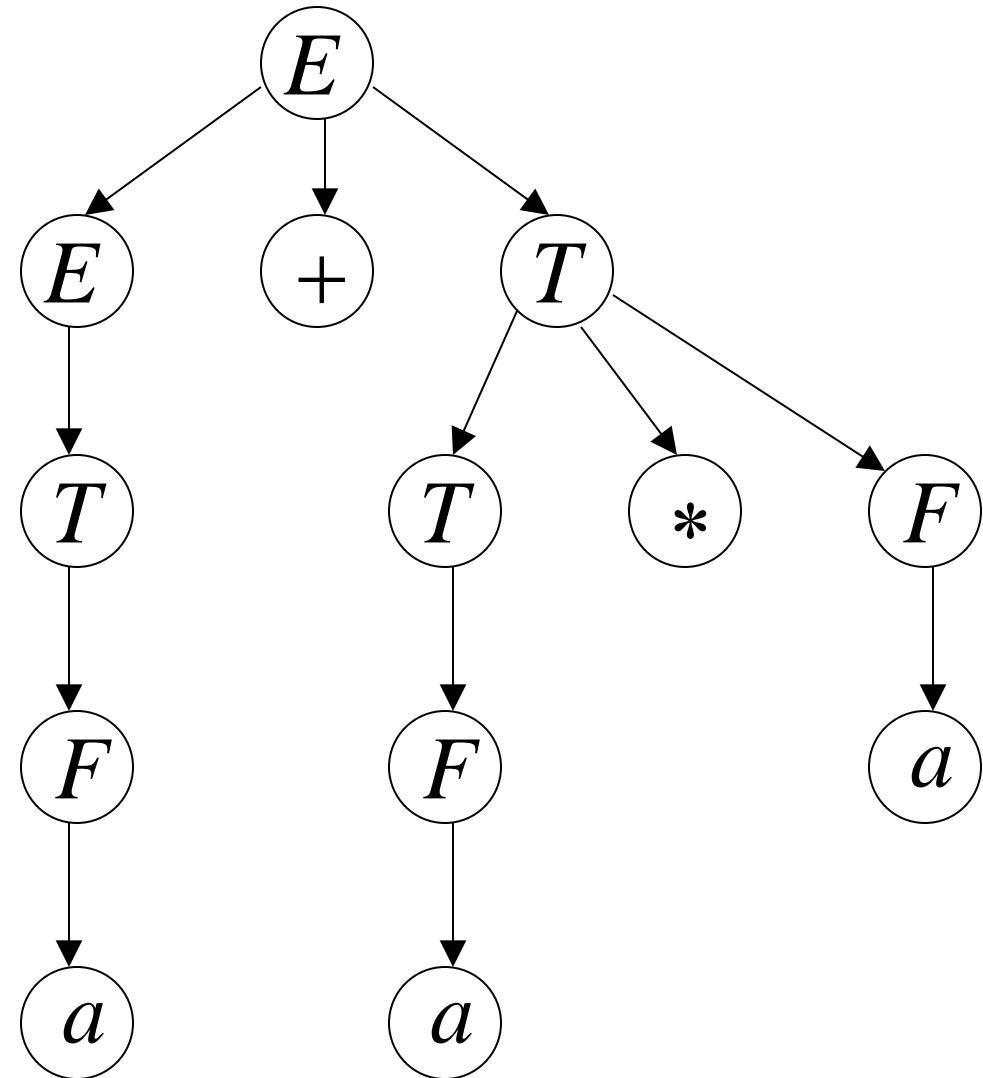
```
 $E \rightarrow E + T \mid T$ 
 $T \rightarrow T * F \mid F$ 
 $F \rightarrow (E) \mid a$ 
```

generates the same language

$$\begin{aligned}
 E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F \\
 &\Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a
 \end{aligned}$$

$$\begin{array}{l}
 E \rightarrow E + T \mid T \\
 T \rightarrow T * F \mid F \\
 F \rightarrow (E) \mid a
 \end{array}$$

Unique
derivation tree
for $a + a * a$



An un-successful example:

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$$

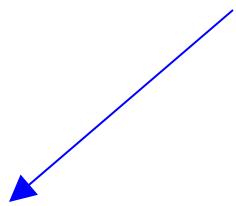
$$n, m \geq 0$$

L is inherently ambiguous:

every grammar that generates this language is ambiguous

Example (ambiguous) grammar for L :

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$$



$$S \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow S_1 c \mid A$$

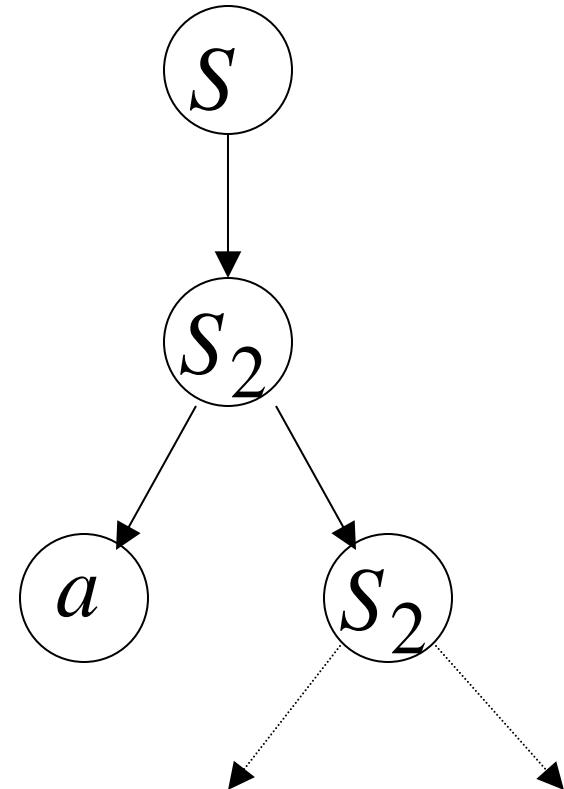
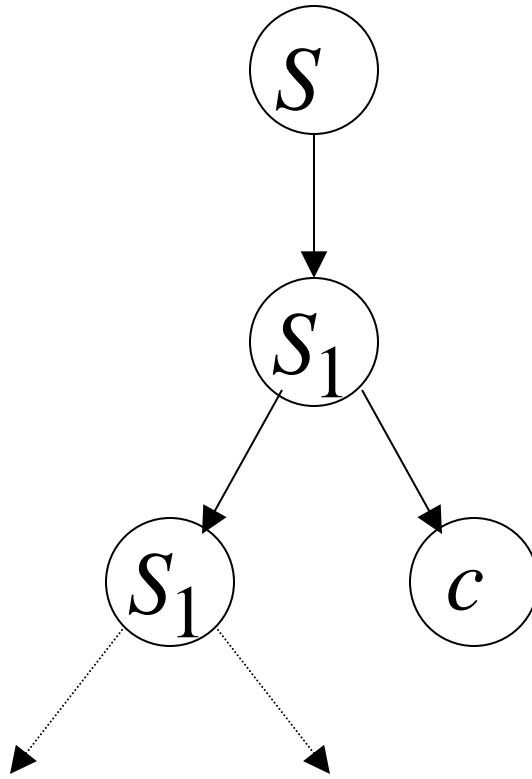
$$S_2 \rightarrow aS_2 \mid B$$

$$A \rightarrow aAb \mid \lambda$$

$$B \rightarrow bBc \mid \lambda$$

The string $a^n b^n c^n \in L$
has always two different derivation trees
(for any grammar)

For example



Simplifications of Context-Free Grammars

A Substitution Rule

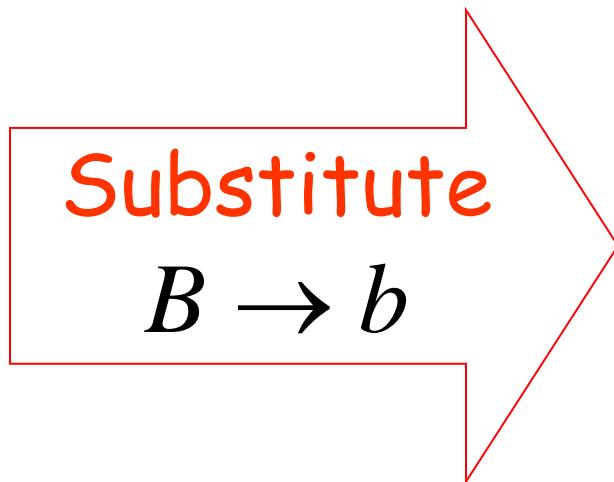
$$S \rightarrow aB$$

$$A \rightarrow aaA$$

$$A \rightarrow abBc$$

$$B \rightarrow aA$$

$$B \rightarrow b$$



Equivalent
grammar

$$S \rightarrow aB \mid ab$$

$$A \rightarrow aaA$$

$$A \rightarrow abBc \mid abbc$$

$$B \rightarrow aA$$

$$S \rightarrow aB \mid ab$$

$$A \rightarrow aaA$$

$$A \rightarrow abBc \mid abbc$$

$$B \rightarrow aA$$

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

λ – production : $X \rightarrow \lambda$

Nullable Variable: $Y \Rightarrow \dots \Rightarrow \lambda$

Example: $S \rightarrow aMb$

$M \rightarrow aMb$

$M \rightarrow \lambda$

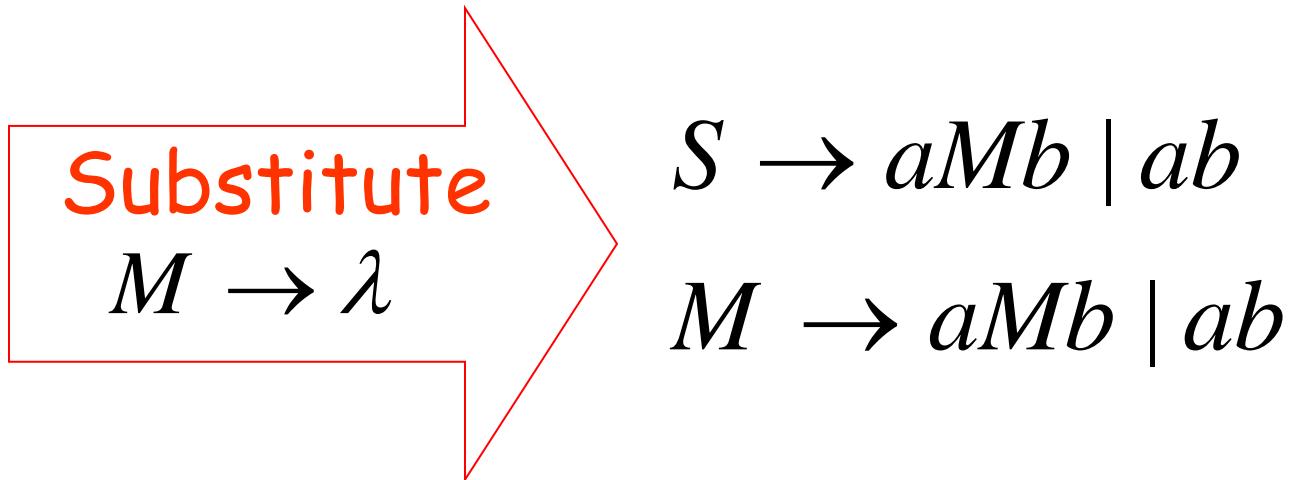
Nullable variable

λ – production



Removing λ – production s

$S \rightarrow aMb$
 $M \rightarrow aMb$
 ~~$M \rightarrow \lambda$~~



After we remove all the λ – production s
all the nullable variables disappear
(except for the start variable)

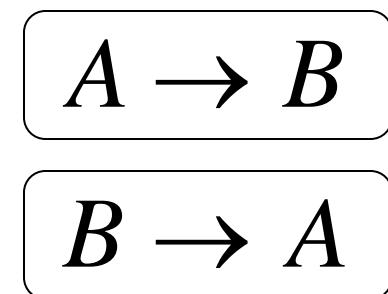
Unit-Productions

Unit Production: $X \rightarrow Y$

(a single variable in both sides)

Example: $S \rightarrow aA$

$A \rightarrow a$



$B \rightarrow bb$

Removal of unit productions:

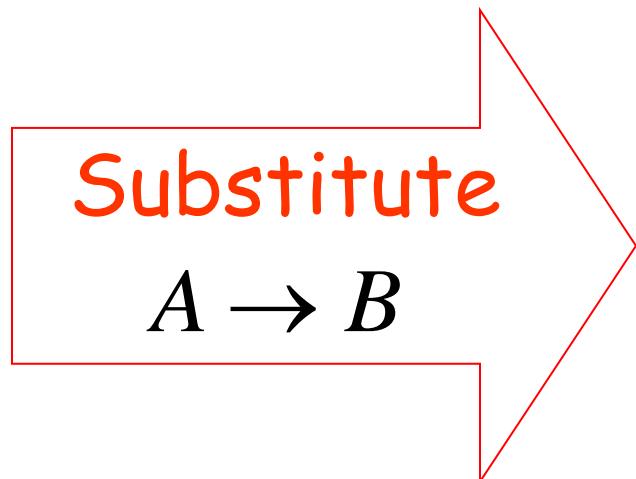
$$S \rightarrow aA$$

$$A \rightarrow a$$

~~$$A \rightarrow B$$~~

$$B \rightarrow A$$

$$B \rightarrow bb$$



$$S \rightarrow aA \mid aB$$

$$A \rightarrow a$$

$$B \rightarrow A \mid B$$

$$B \rightarrow bb$$

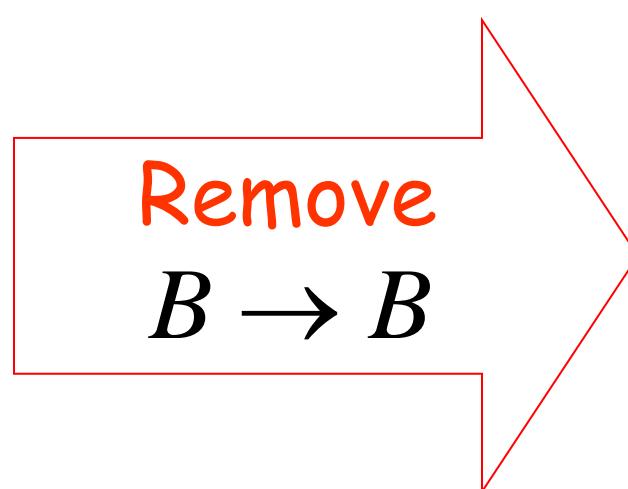
Unit productions of form $X \rightarrow X$
can be removed immediately

$$S \rightarrow aA \mid aB$$

$$A \rightarrow a$$

$$B \rightarrow A \mid \cancel{B}$$

$$B \rightarrow bb$$



$$S \rightarrow aA \mid aB$$

$$A \rightarrow a$$

$$B \rightarrow A$$

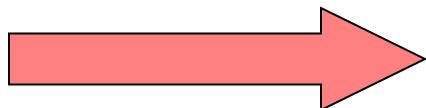
$$B \rightarrow bb$$

$$S \rightarrow aA \mid aB$$
$$A \rightarrow a$$
~~$$B \rightarrow A$$~~
$$B \rightarrow bb$$

Substitute

$$B \rightarrow A$$
$$S \rightarrow aA \mid aB \mid aA$$
$$A \rightarrow a$$
$$B \rightarrow bb$$

Remove repeated productions

$$S \rightarrow aA \mid aB \mid \cancel{aA}$$
$$A \rightarrow a$$
$$B \rightarrow bb$$


Final grammar

$$S \rightarrow aA \mid aB$$
$$A \rightarrow a$$
$$B \rightarrow bb$$

Useless Productions

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

$$S \rightarrow A$$

$$A \rightarrow aA$$

Useless Production

Some derivations never terminate...

$$S \Rightarrow A \Rightarrow aA \Rightarrow aaA \Rightarrow \dots \Rightarrow aa\dots aA \Rightarrow \dots$$

Another grammar:

$$S \rightarrow A$$

$$A \rightarrow aA$$

$$A \rightarrow \lambda$$

$$B \rightarrow bA$$

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

$$S \rightarrow aSb$$

$$S \rightarrow \lambda$$

Variables

useless

useless

useless

$$S \rightarrow A$$

$$A \rightarrow aA$$

$$B \rightarrow C$$

$$C \rightarrow D$$

Productions

useless

useless

useless

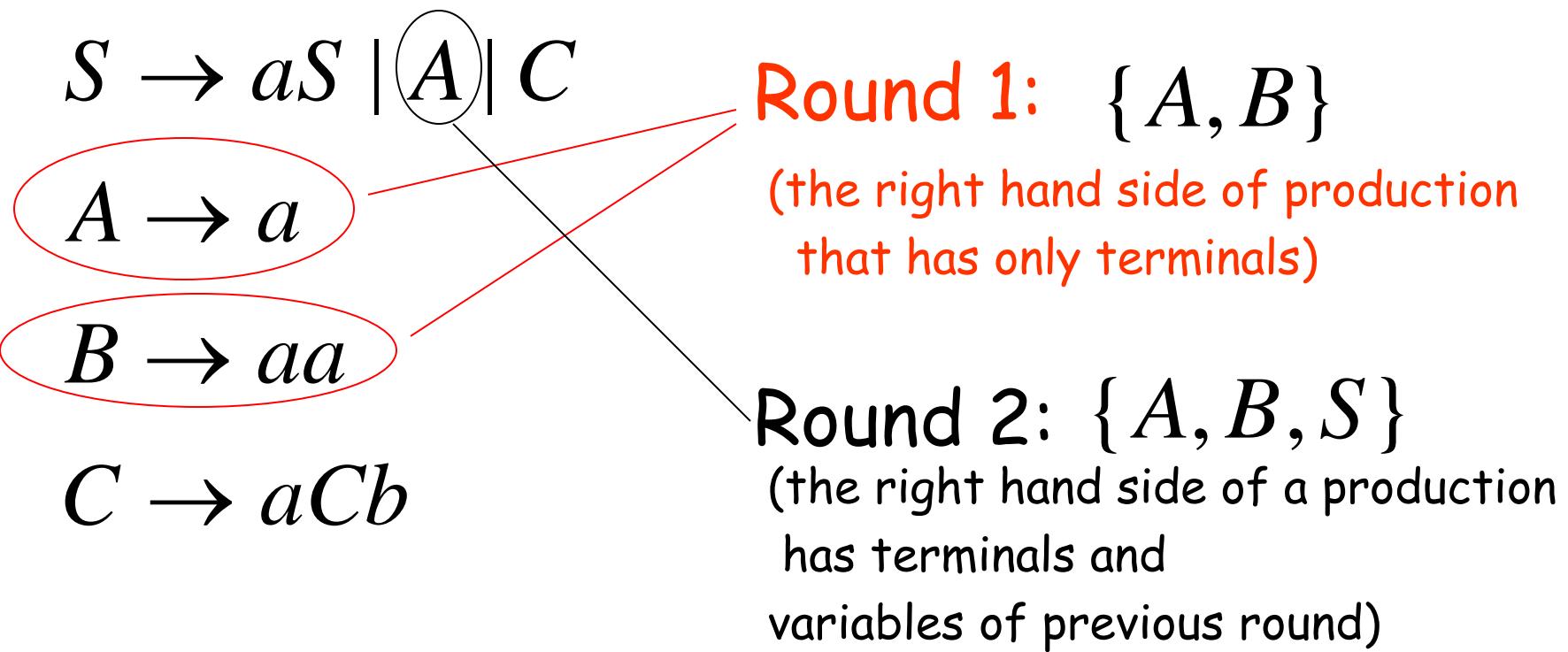
useless

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 λ (possible useful variables)



This process can be generalized

Then, remove productions that use variables other than $\{A, B, S\}$

$$S \rightarrow aS \mid A \mid \cancel{C}$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

~~$$C \rightarrow aCb$$~~



$$S \rightarrow aS \mid A$$

$$A \rightarrow a$$

$$B \rightarrow aa$$

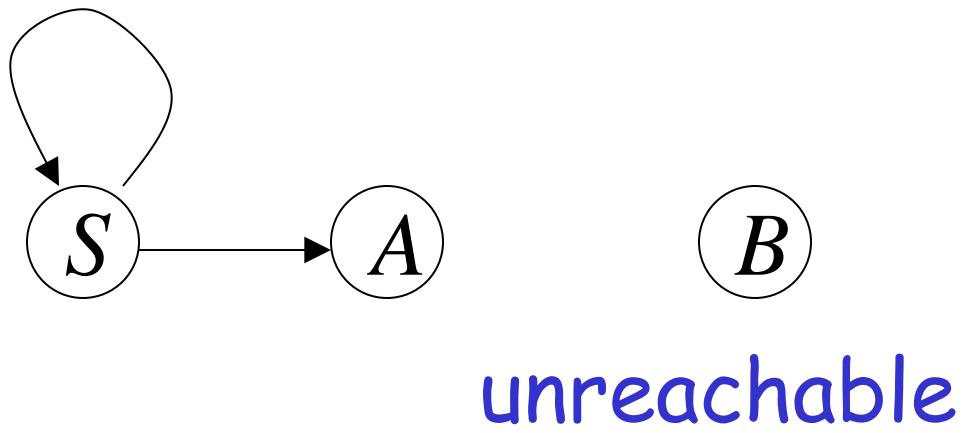
Second: Find all variables
reachable from S

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

Step 2: Remove Unit-Productions

Step 3: Remove Useless Variables

This sequence guarantees that unwanted variables and productions are removed

Normal Forms for Context-free Grammars

Chomsky Normal Form

Each production has form:

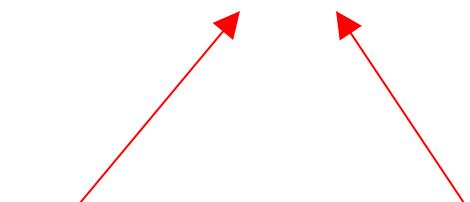
$$A \rightarrow BC$$

variable

or

$$A \rightarrow a$$

terminal



Examples:

$$S \rightarrow AS$$
$$S \rightarrow a$$
$$A \rightarrow SA$$
$$A \rightarrow b$$

Chomsky
Normal Form

$$S \rightarrow AS$$
$$S \rightarrow AAS$$
$$A \rightarrow SA$$
$$A \rightarrow aa$$

Not Chomsky
Normal Form

Conversion to Chomsky Normal Form

Example:

$$S \rightarrow ABa$$

$$A \rightarrow aab$$

$$B \rightarrow Ac$$

Not Chomsky
Normal Form

We will convert it to Chomsky Normal Form

Introduce new variables for the terminals:

$$T_a, T_b, T_c$$

$$S \rightarrow ABT_a$$

$$S \rightarrow ABa$$

$$A \rightarrow T_a T_a T_b$$

$$A \rightarrow aab$$



$$B \rightarrow AT_c$$

$$B \rightarrow Ac$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$

Introduce new intermediate variable V_1
to break first production:

$$S \rightarrow ABT_a$$

$$A \rightarrow T_a T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$



$$S \rightarrow AV_1$$

$$V_1 \rightarrow BT_a$$

$$A \rightarrow T_a T_a T_b$$

$$B \rightarrow AT_c$$

$$T_a \rightarrow a$$

$$T_b \rightarrow b$$

$$T_c \rightarrow c$$

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 λ)
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}

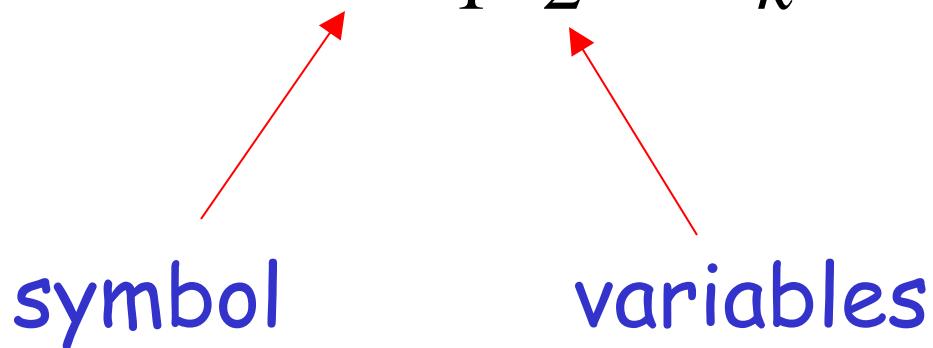
Observations

- 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:

$$A \rightarrow a V_1 V_2 \cdots V_k \quad k \geq 0$$



Examples:

$$S \rightarrow cAB$$

$$A \rightarrow aA \mid bB \mid b$$

$$B \rightarrow b$$

Greinbach
Normal Form

$$S \rightarrow abSb$$

$$S \rightarrow aa$$

Not Greinbach
Normal Form

Conversion to Greinbach Normal Form:

$$S \rightarrow abSb$$
$$S \rightarrow aa$$

$$S \rightarrow aT_b S T_b$$
$$S \rightarrow aT_a$$
$$T_a \rightarrow a$$
$$T_b \rightarrow b$$

Greinbach
Normal Form

Observations

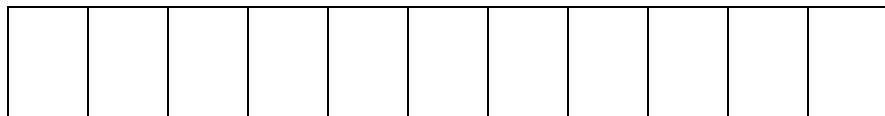
- Greinbach normal forms are very good for parsing strings (better than Chomsky Normal Forms)
- However, it is difficult to find the Greinbach normal of a grammar

Pushdown Automata

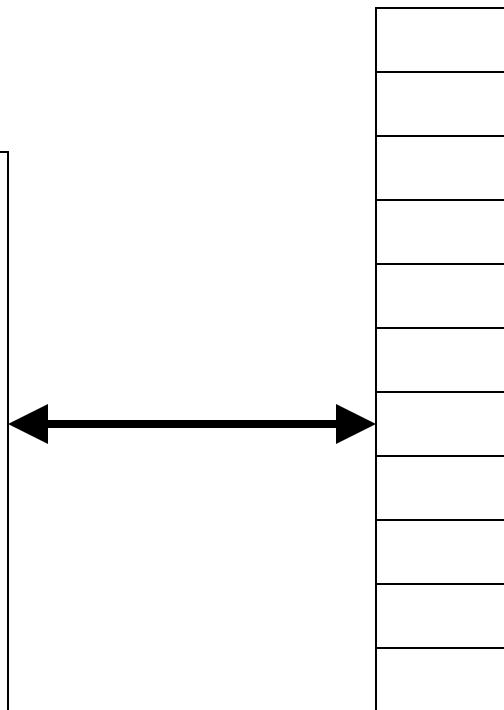
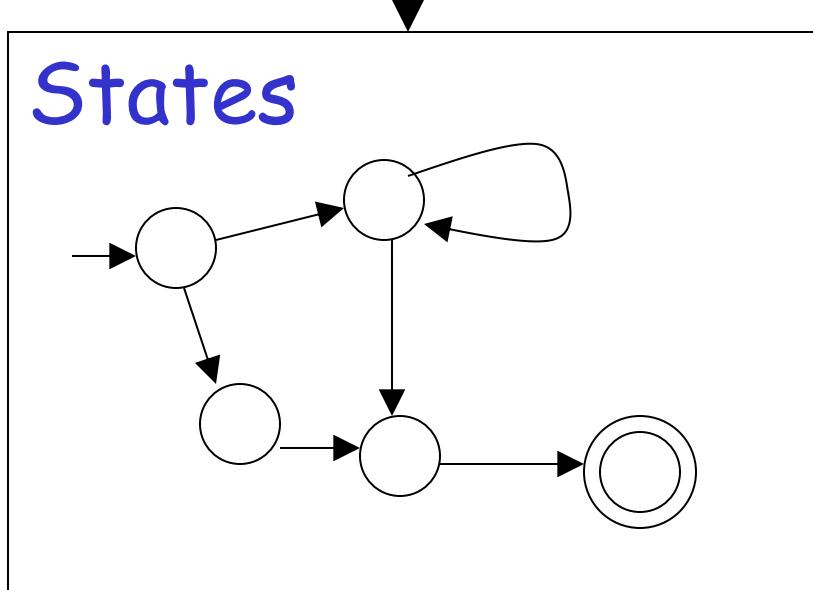
PDAs

Pushdown Automaton -- PDA

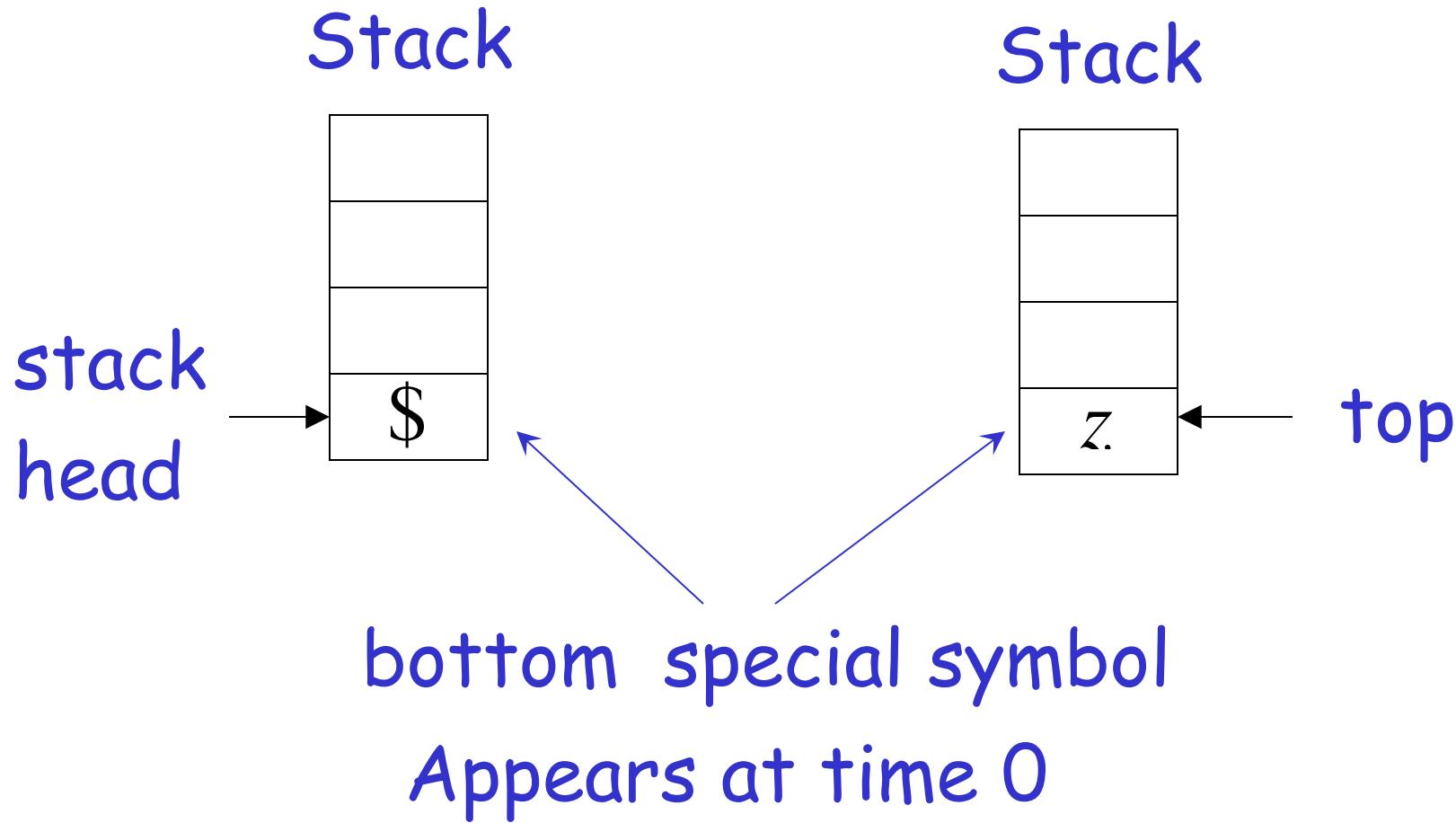
Input String



Stack



Initial Stack Symbol

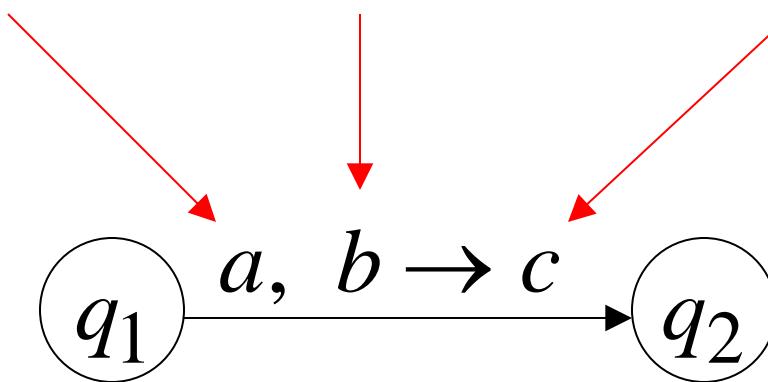


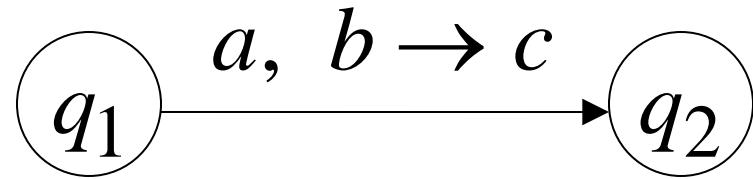
The States

Input
symbol

Pop
symbol

Push
symbol





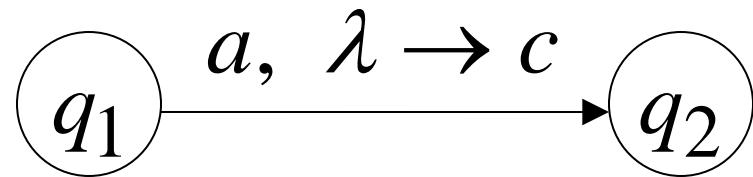
stack

b
h
e
$\$$

top

Replace

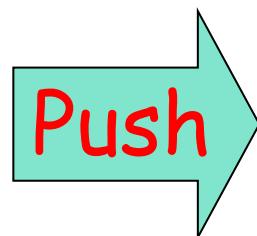
c
h
e
$\$$



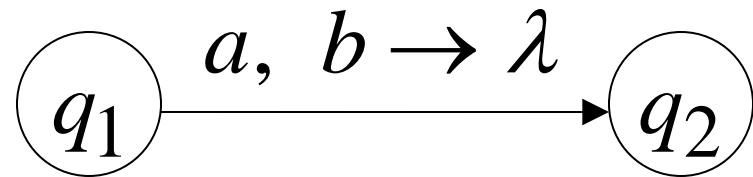
stack

b
h
e
\$

top



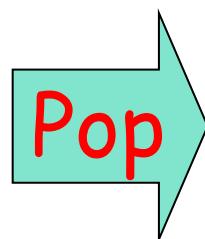
c
b
h
e
\$



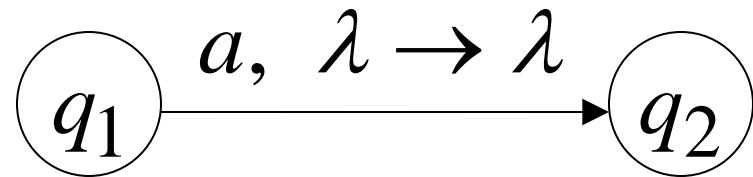
stack

b
h
e
$\$$

top



h
e
$\$$



stack

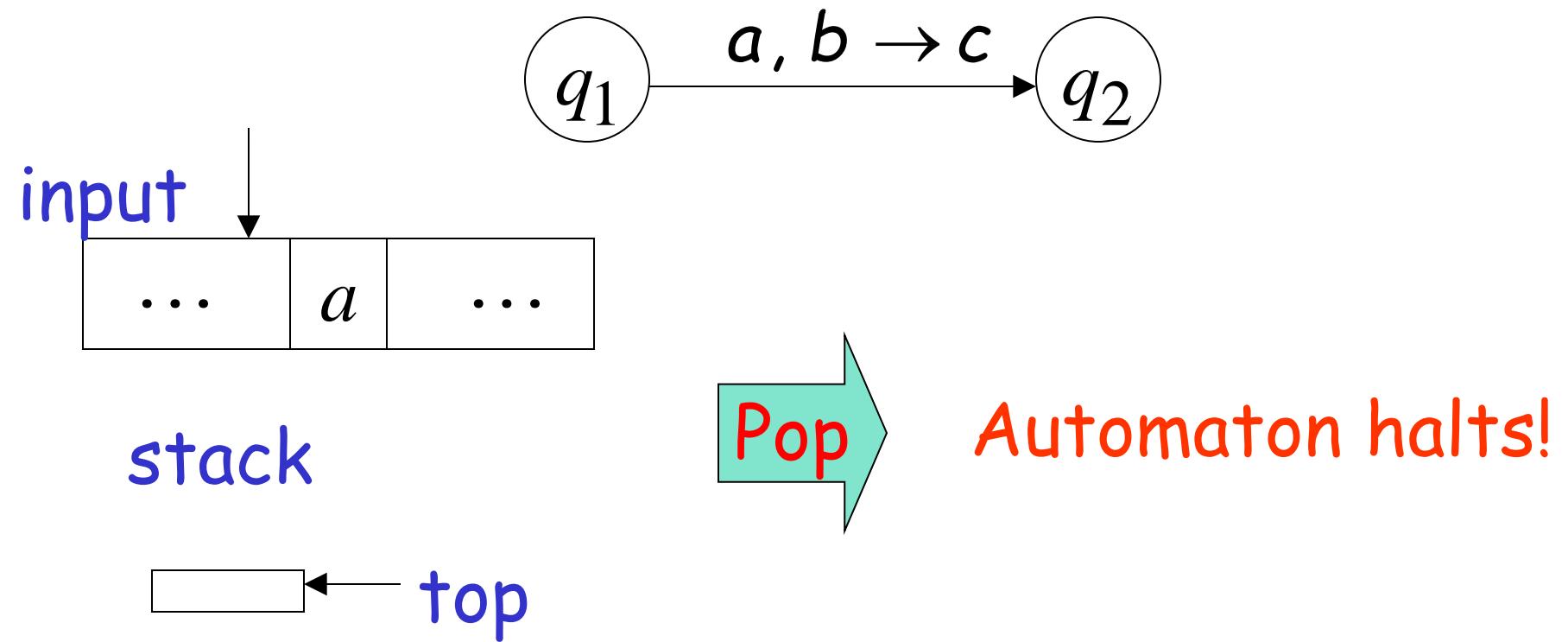
b
h
e
\$

top



b
h
e
\$

Pop from Empty Stack

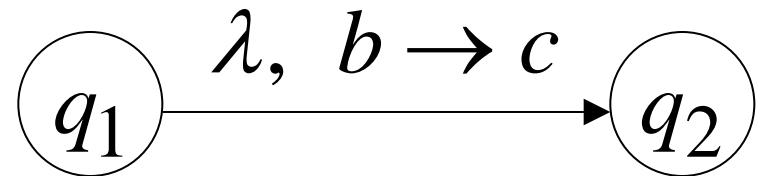
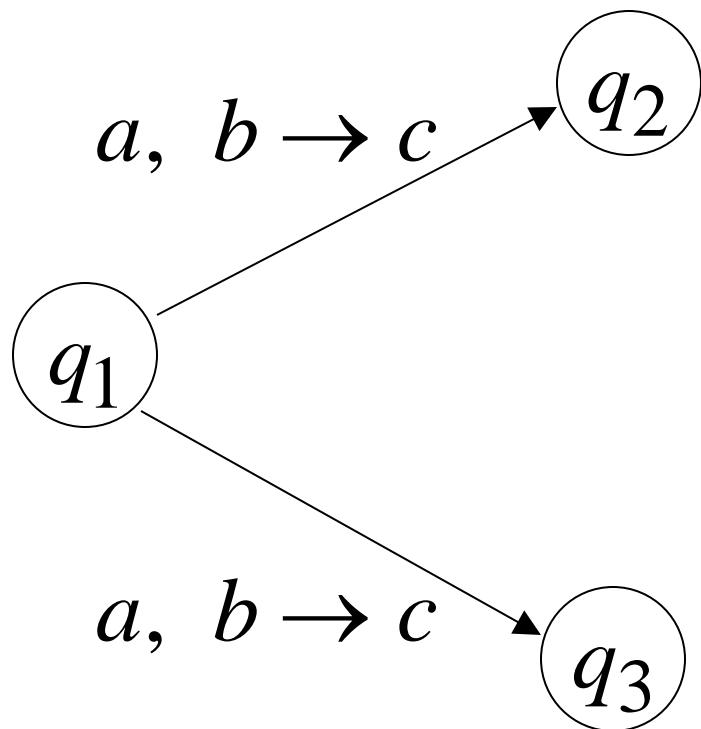


If the automaton attempts to pop from empty stack then it halts and rejects input

Non-Determinism

PDAs are non-deterministic

Allowed non-deterministic transitions

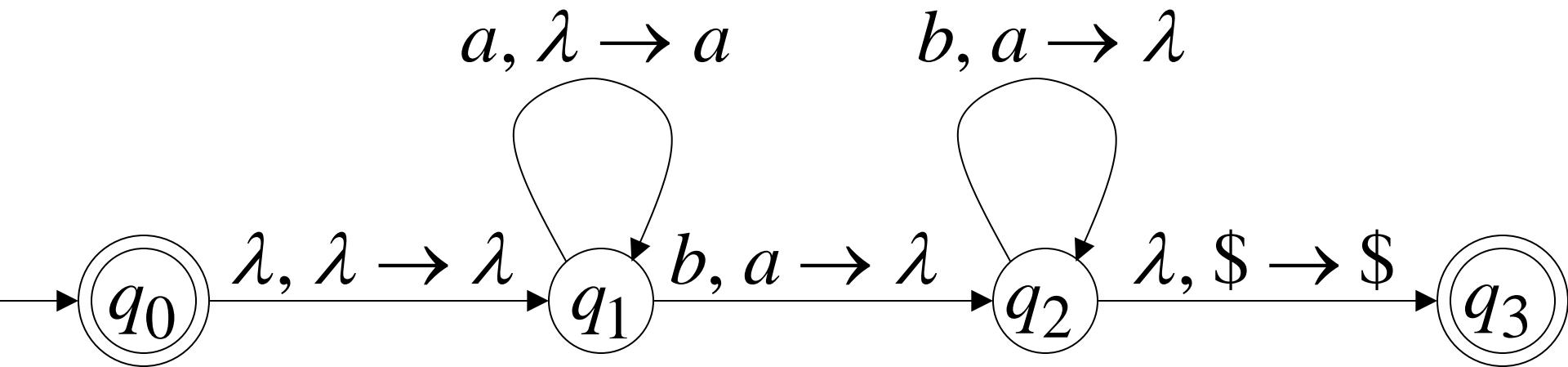


λ – transition

Example PDA

PDA M :

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



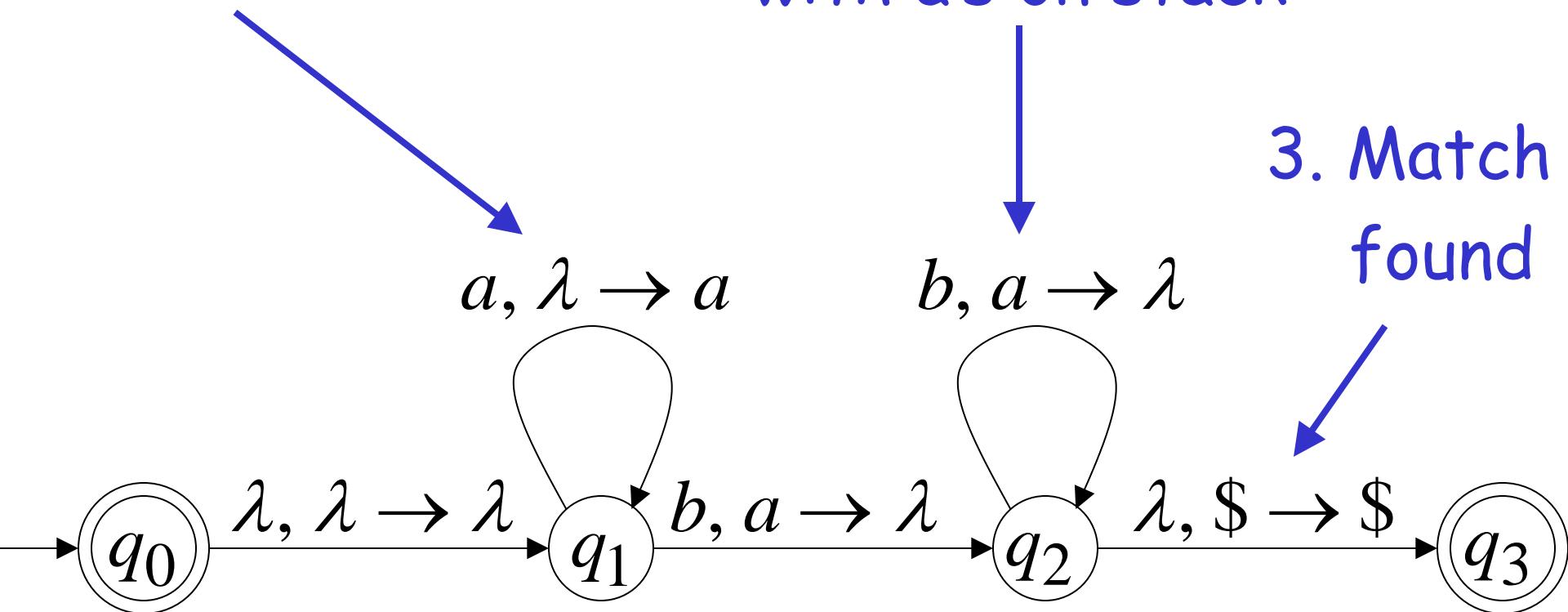
$$L(M) = \{a^n b^n : n \geq 0\}$$

Basic Idea:

1. Push the a's
on the stack

2. Match the b's on input
with a's on stack

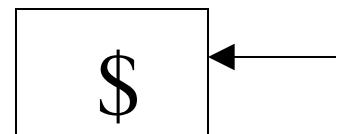
3. Match
found



Execution Example: Time 0

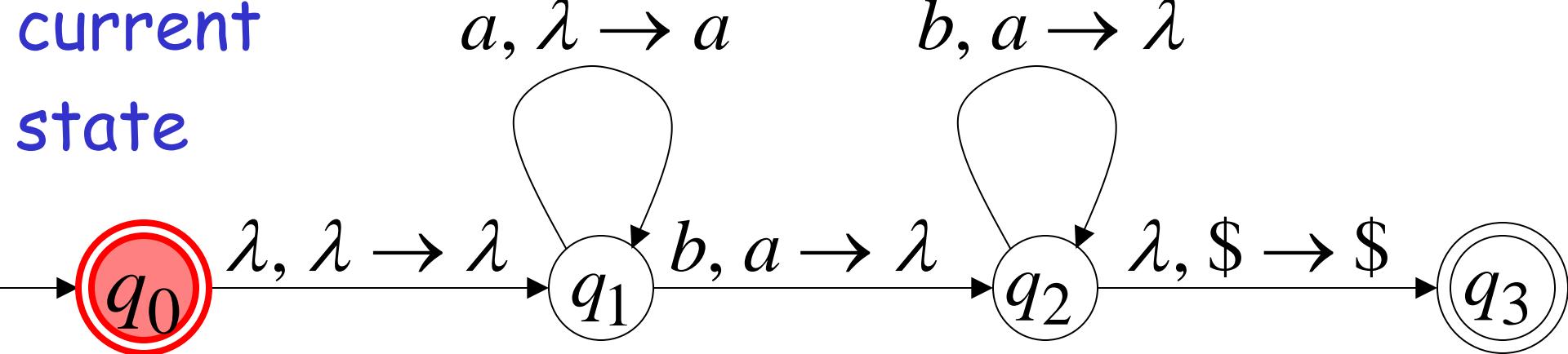
Input

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



Stack

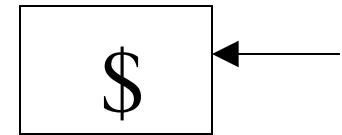
current
state



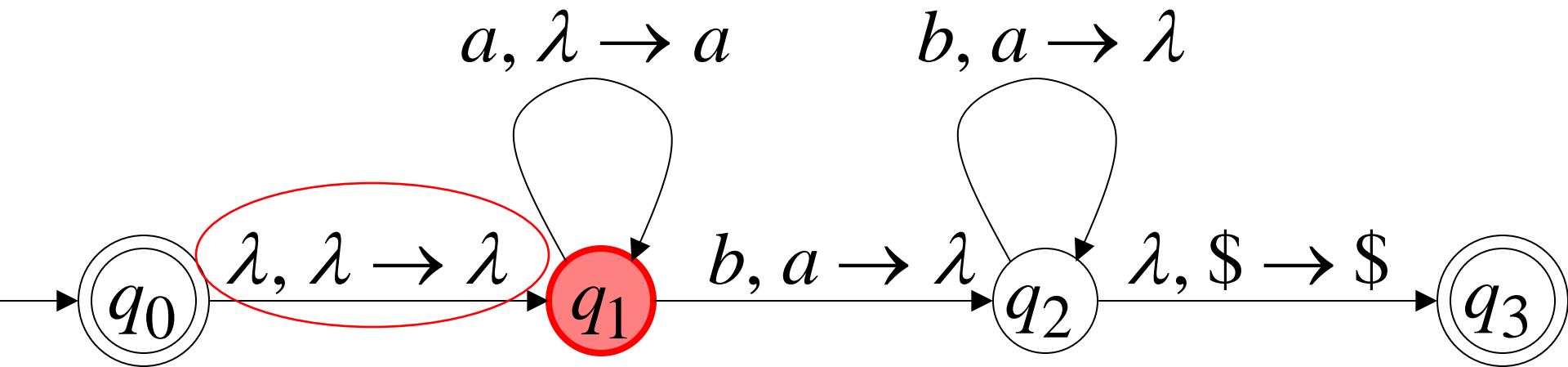
Time 1

Input

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



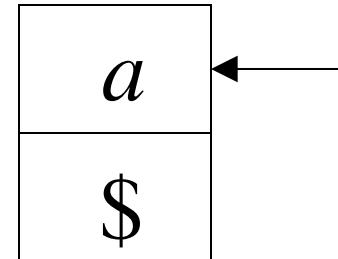
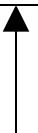
Stack



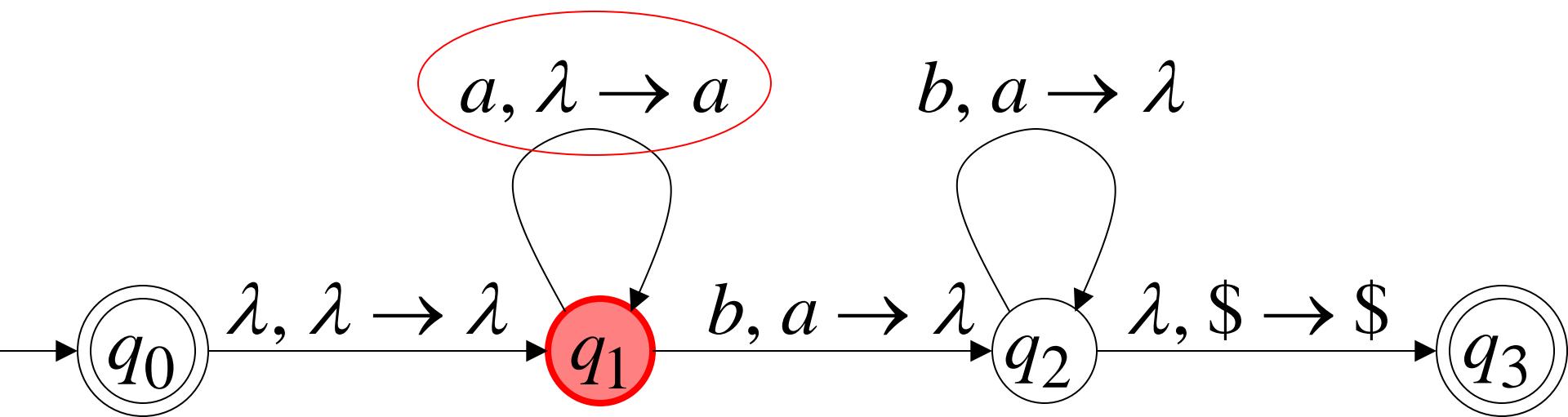
Time 2

Input

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



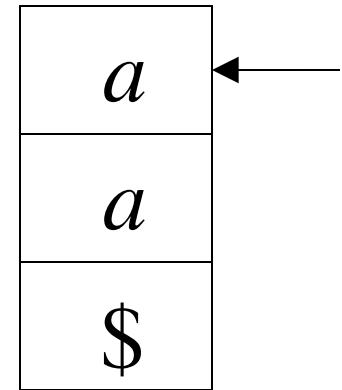
Stack



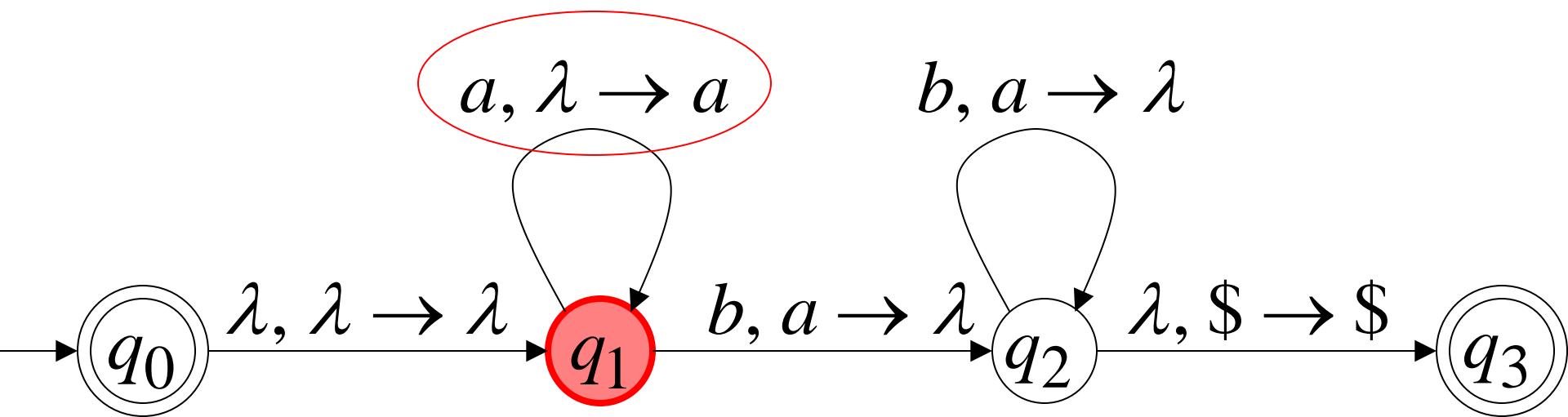
Time 3

Input

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



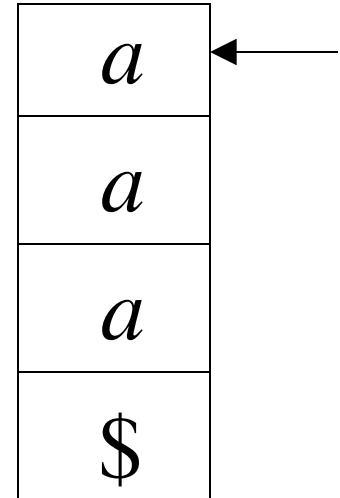
Stack



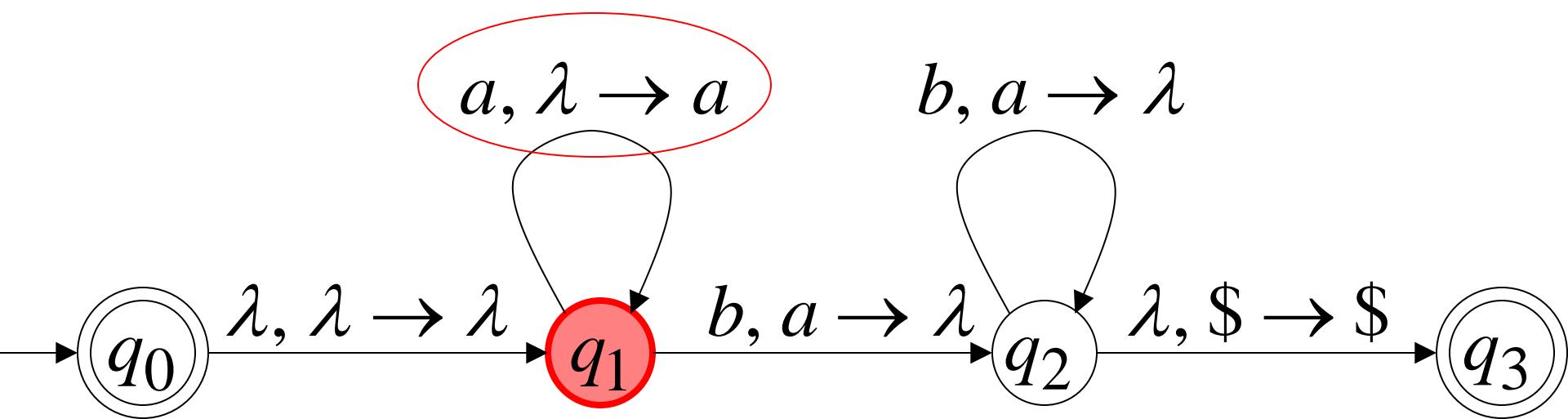
Time 4

Input

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



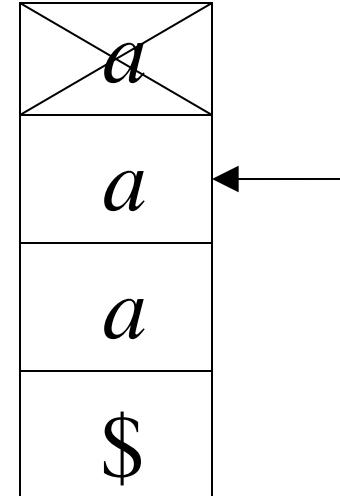
Stack



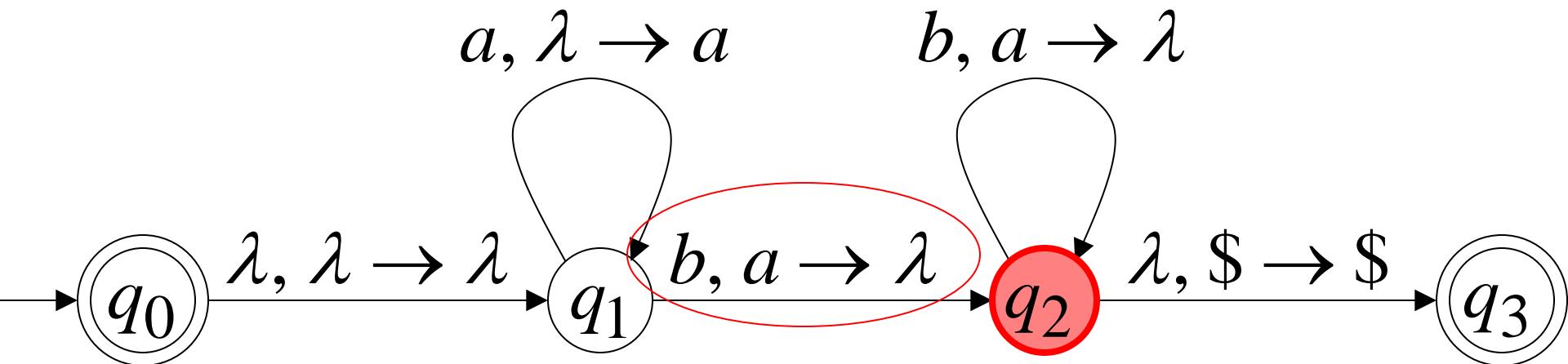
Time 5

Input

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



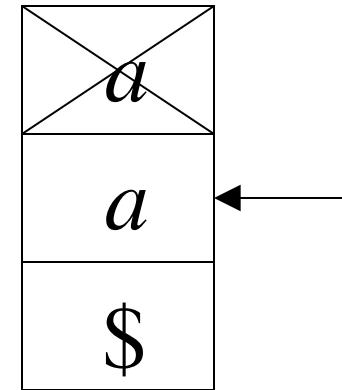
Stack



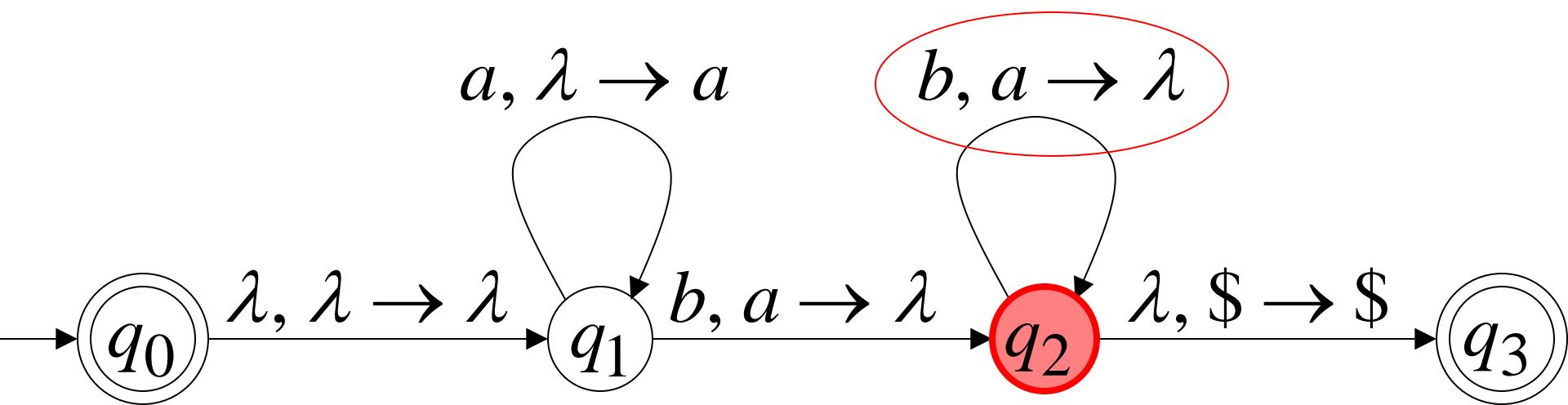
Time 6

Input

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



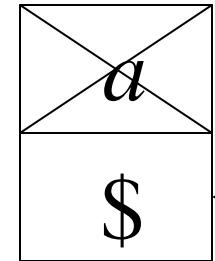
Stack



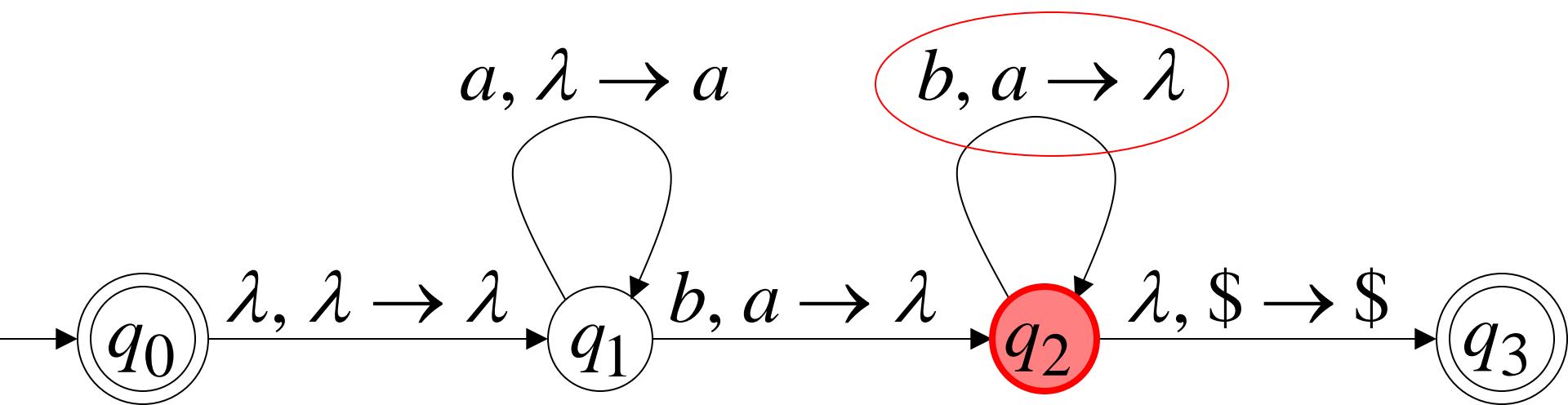
Time 7

Input

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



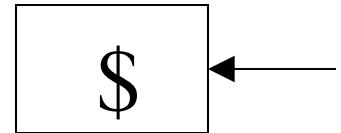
Stack



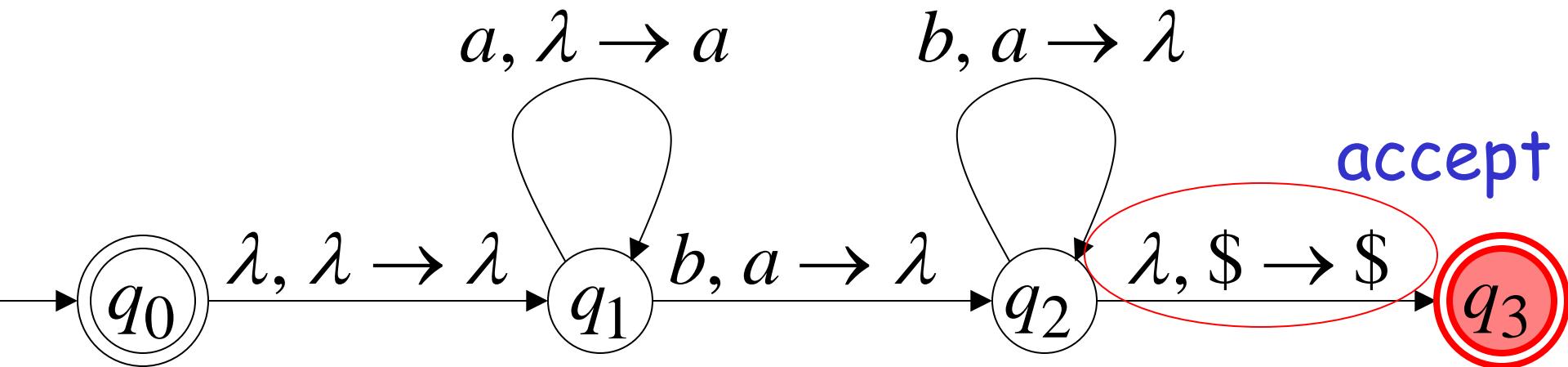
Time 8

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

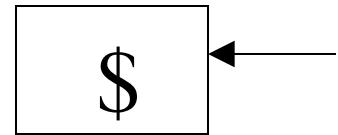
The last state is an accepting state

we do not care about the stack contents
at the end of the accepting computation

Rejection Example: Time 0

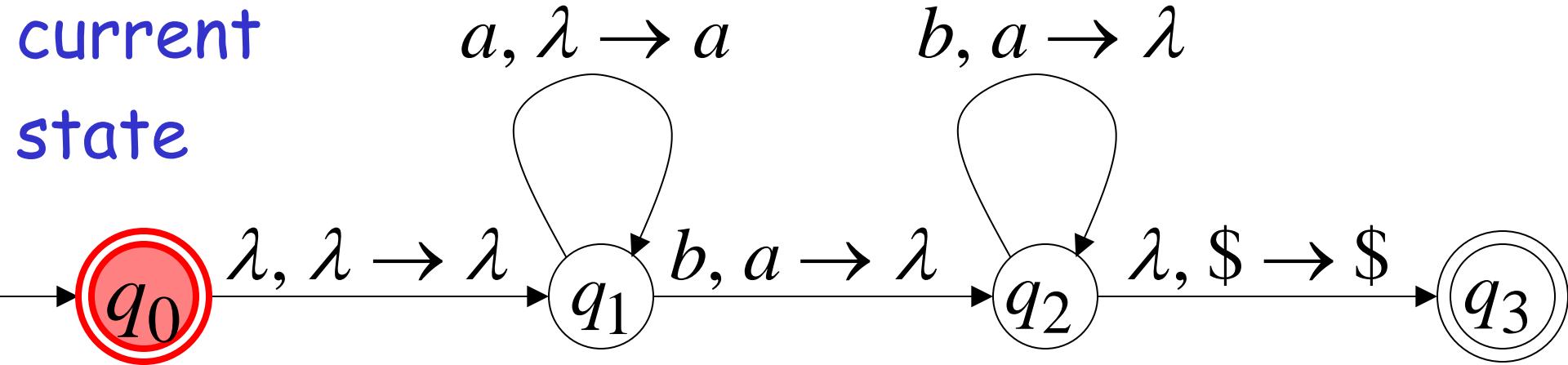
Input

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



Stack

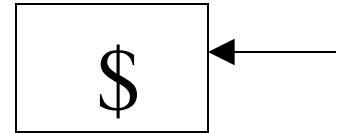
current
state



Rejection Example: Time 1

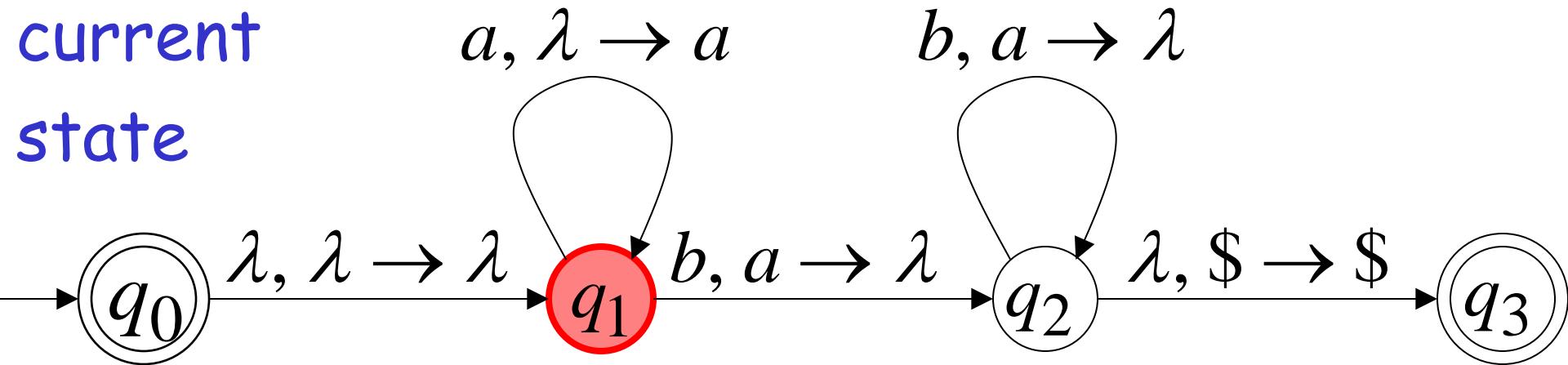
Input

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



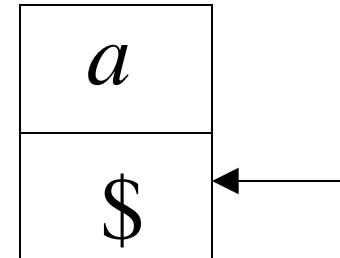
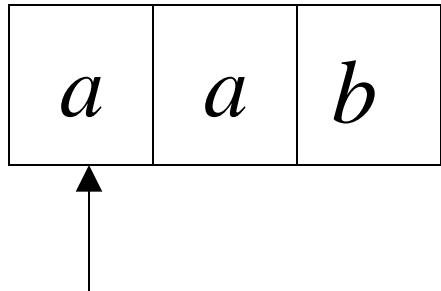
Stack

current
state



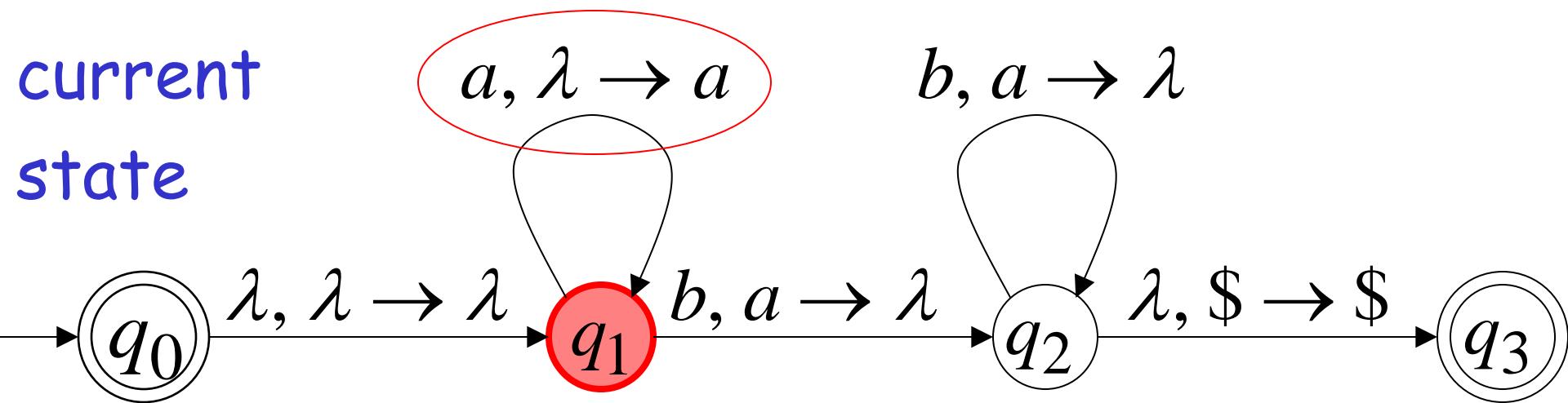
Rejection Example: Time 2

Input



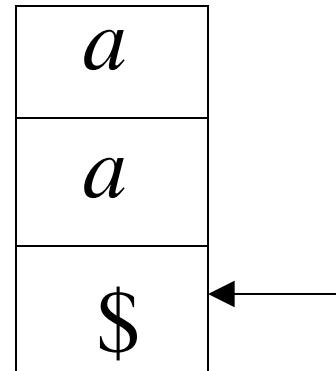
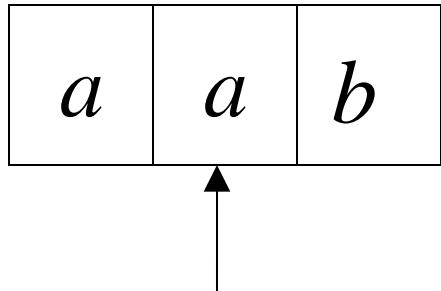
Stack

current
state



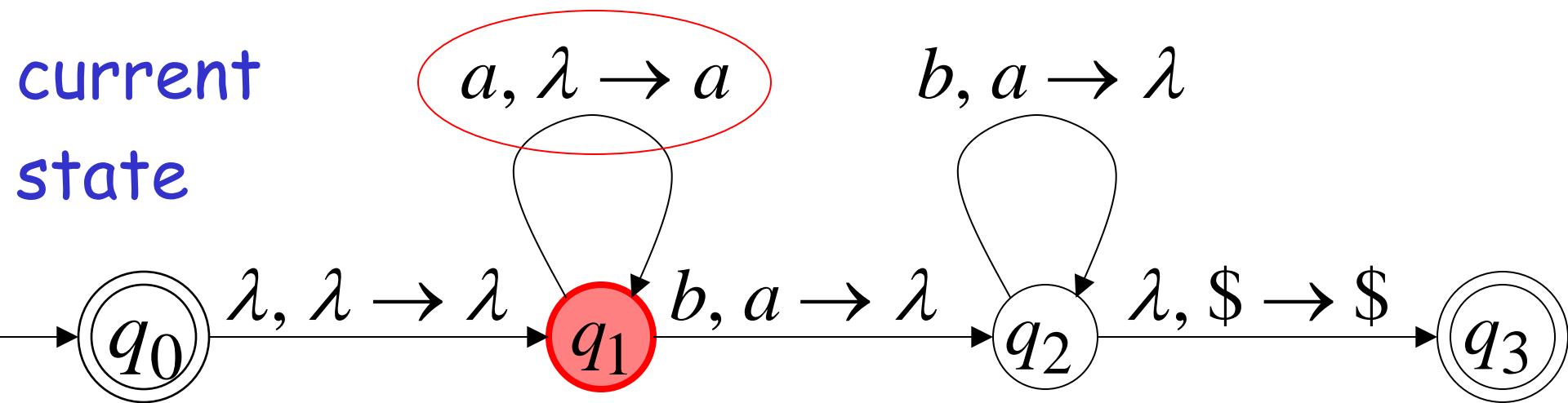
Rejection Example: Time 3

Input



Stack

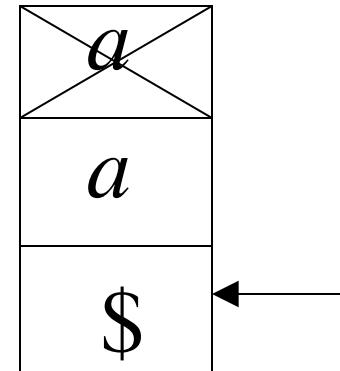
current
state



Rejection Example: Time 4

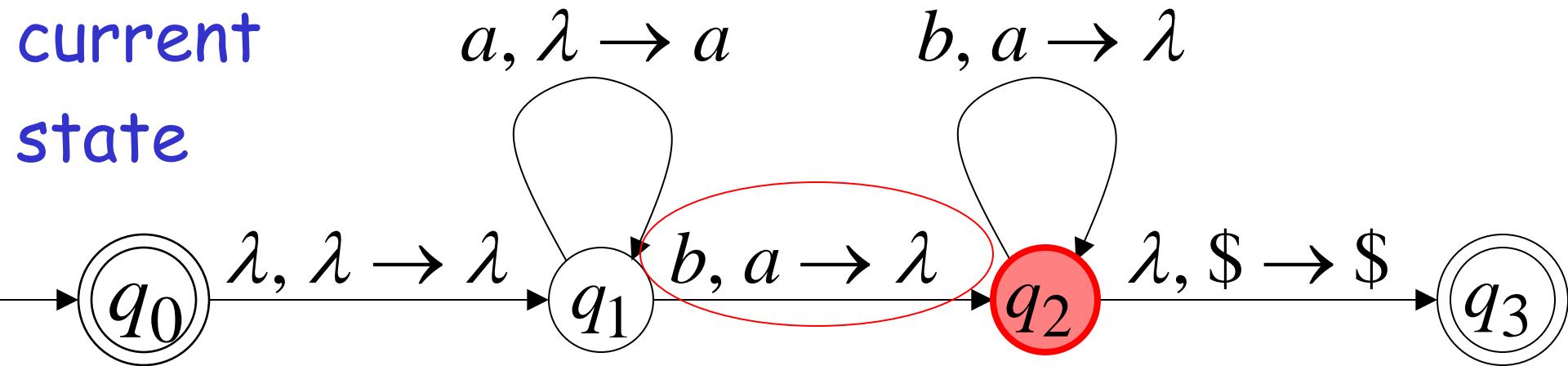
Input

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



Stack

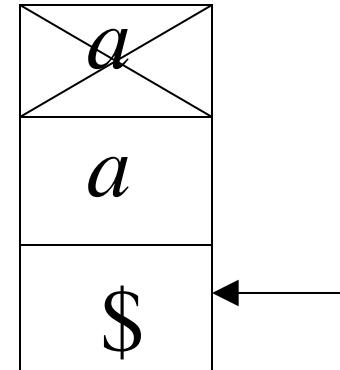
current
state



Rejection Example: Time 4

Input

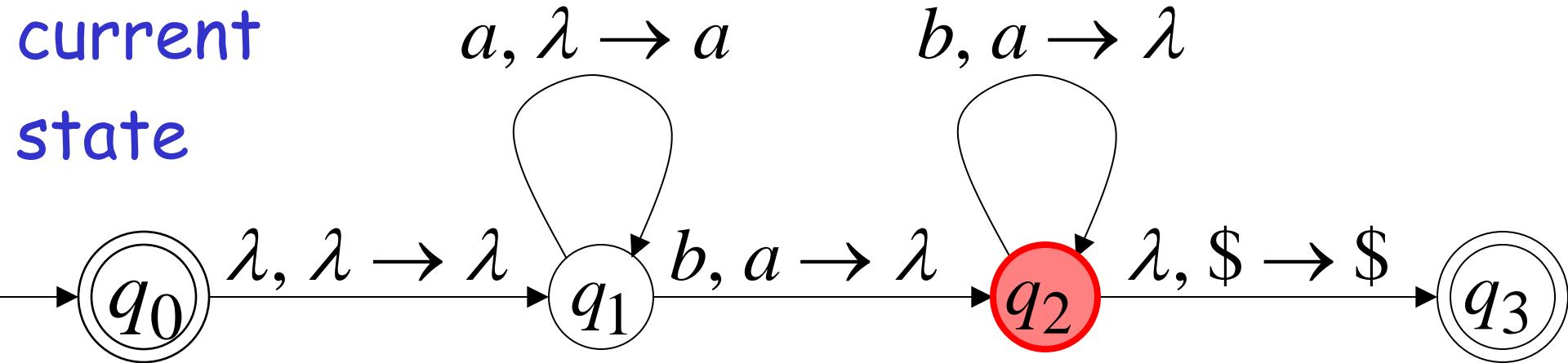
a	a	b
-----	-----	-----



Stack

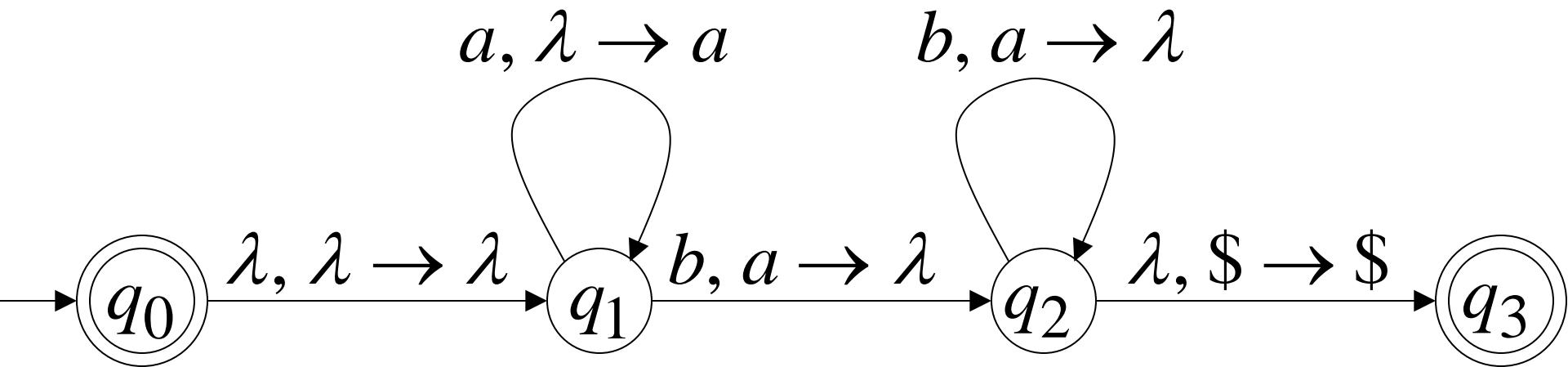
reject

current
state



There is no accepting computation for aab

The string aab is rejected by the PDA



Another PDA example

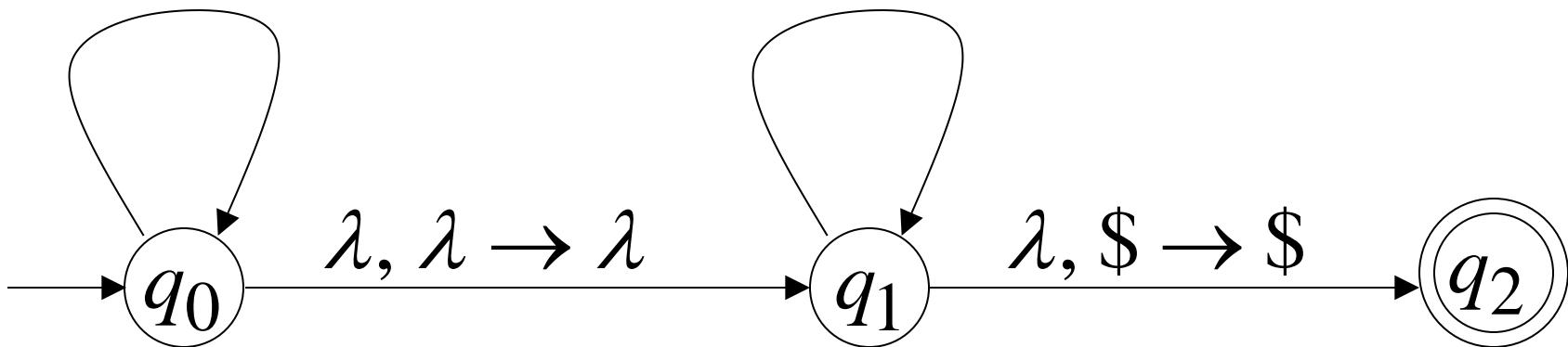
PDA M : $L(M) = \{vv^R : v \in \{a,b\}^*\}$

$$a, \lambda \rightarrow a$$

$$b, \lambda \rightarrow b$$

$$a, a \rightarrow \lambda$$

$$b, b \rightarrow \lambda$$



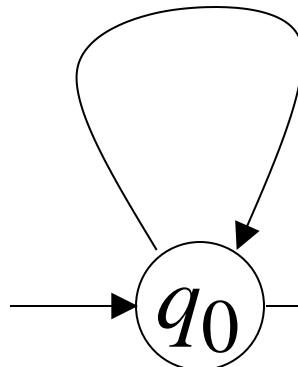
Basic Idea:

$$L(M) = \{vv^R : v \in \{a,b\}^*\}$$

1. Push v
on stack



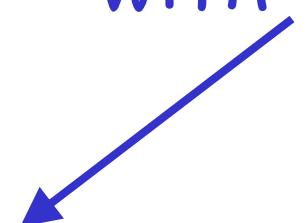
$$\begin{array}{l} a, \lambda \rightarrow a \\ b, \lambda \rightarrow b \end{array}$$



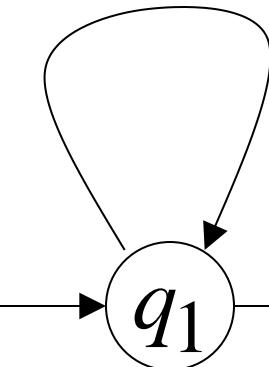
2. Guess
middle
of input



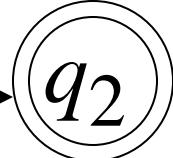
3. Match v^R on input
with v on stack



4. Match
found



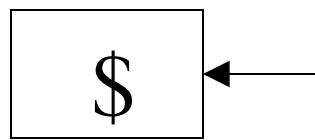
$$\lambda, \$ \rightarrow \$$$



Execution Example: Time 0

Input

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

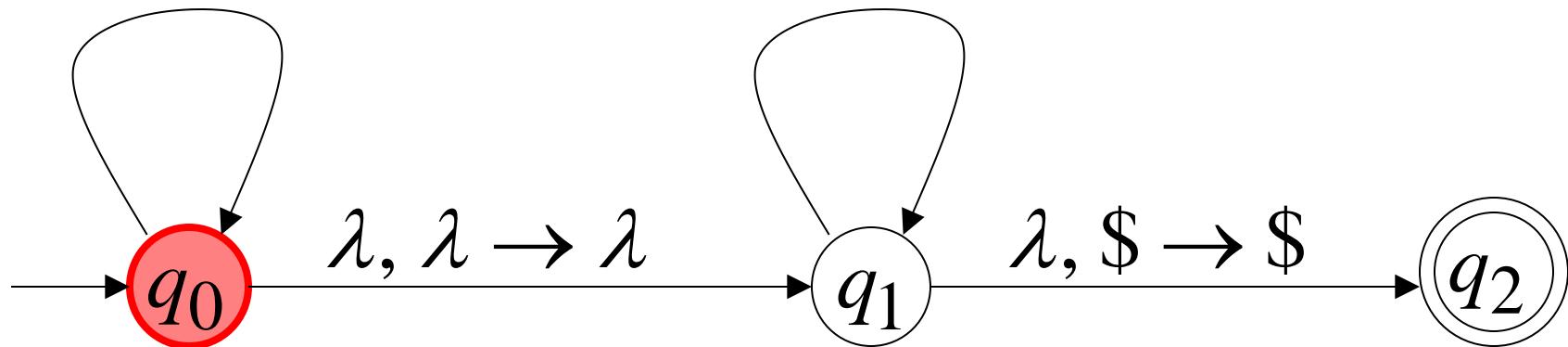


$a, \lambda \rightarrow a$

$a, a \rightarrow \lambda$

$b, \lambda \rightarrow b$

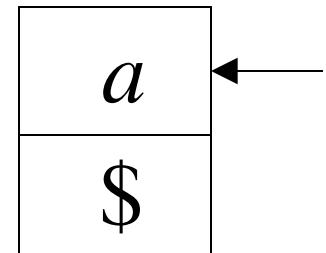
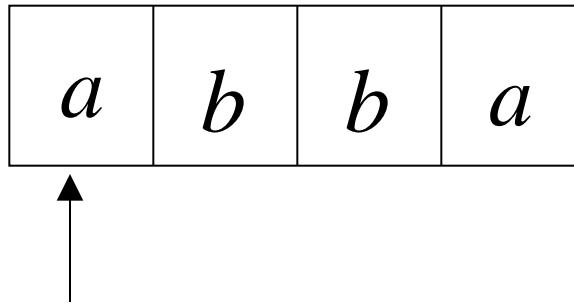
$b, b \rightarrow \lambda$



Stack

Time 1

Input



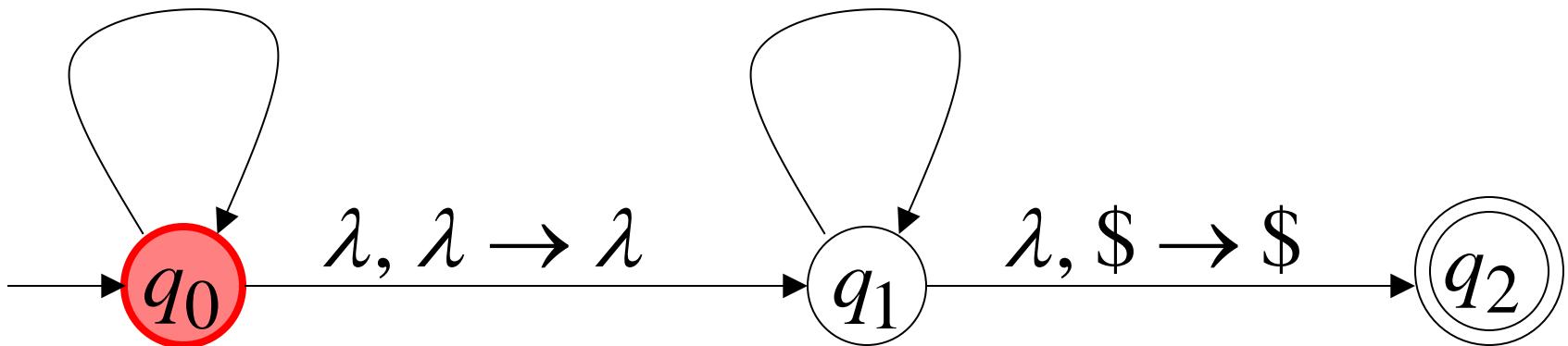
Stack

$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

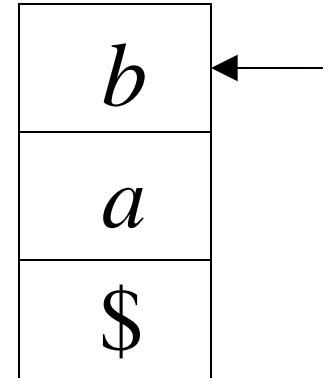
$$b, b \rightarrow \lambda$$



Time 2

Input

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



Stack

$$a, \lambda \rightarrow a$$

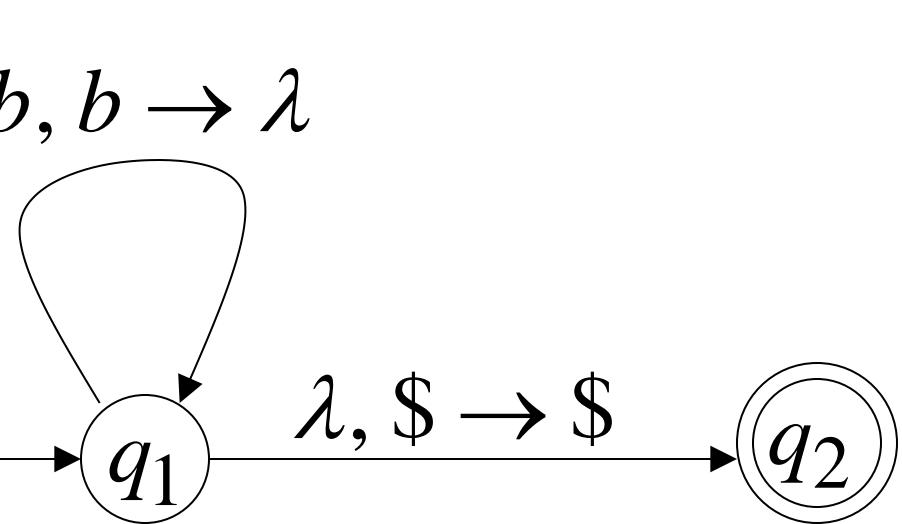
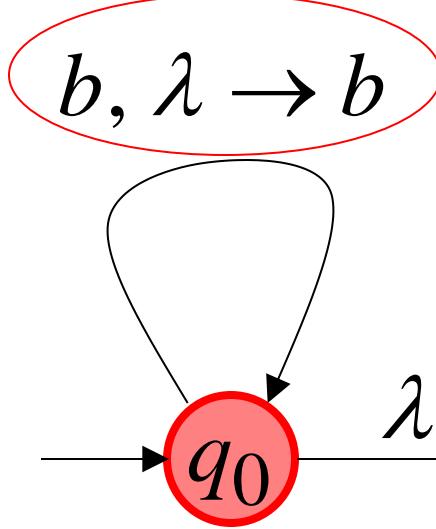
$$b, \lambda \rightarrow b$$

$$a, a \rightarrow \lambda$$

$$b, b \rightarrow \lambda$$

$$\lambda, \lambda \rightarrow \lambda$$

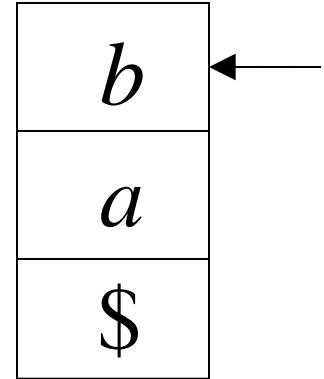
$$\lambda, \$ \rightarrow \$$$



Time 3

Input

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



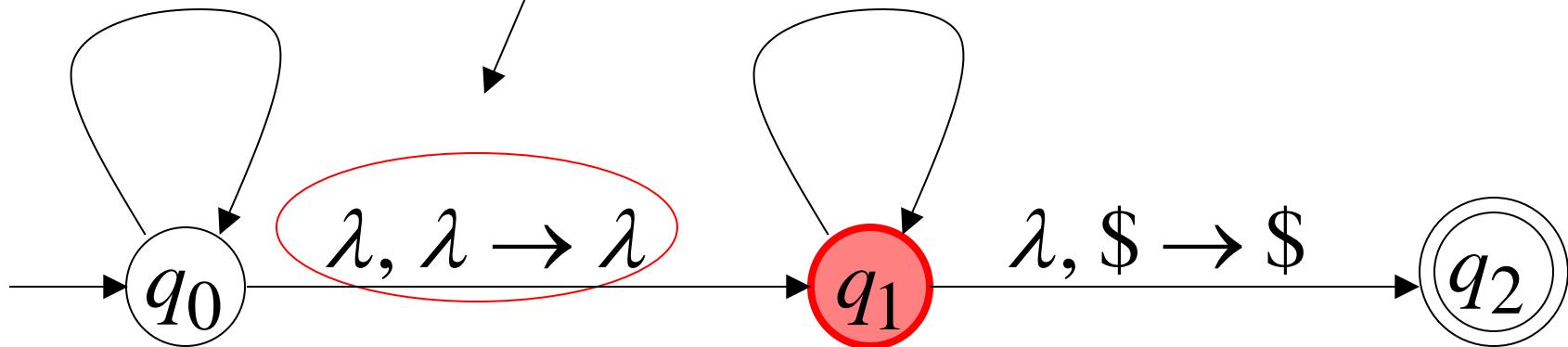
Guess the middle
of string

$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

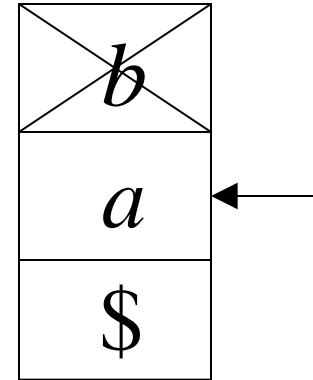
$$b, b \rightarrow \lambda$$



Time 4

Input

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



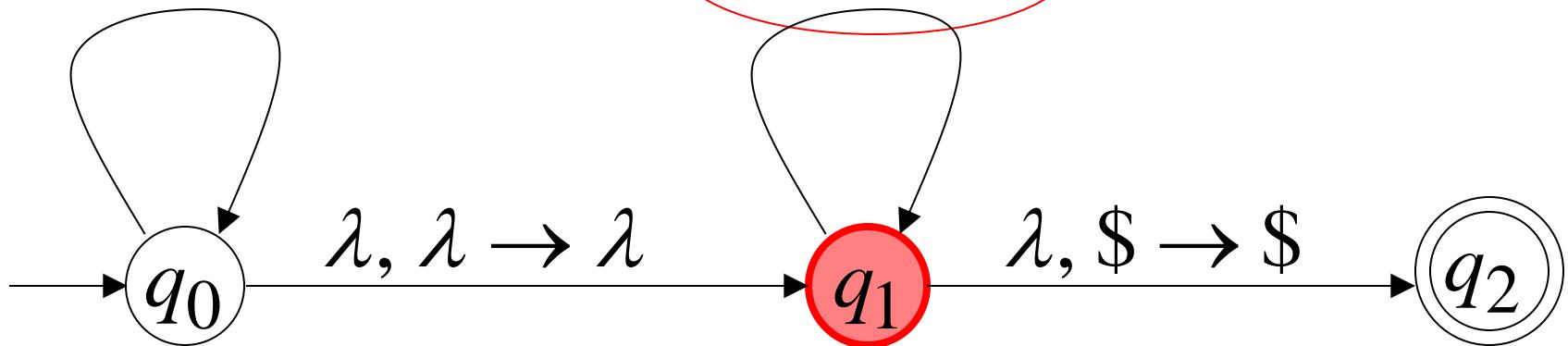
Stack

$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

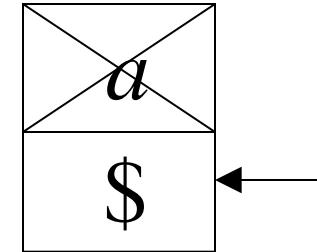
$$b, b \rightarrow \lambda$$



Time 5

Input

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



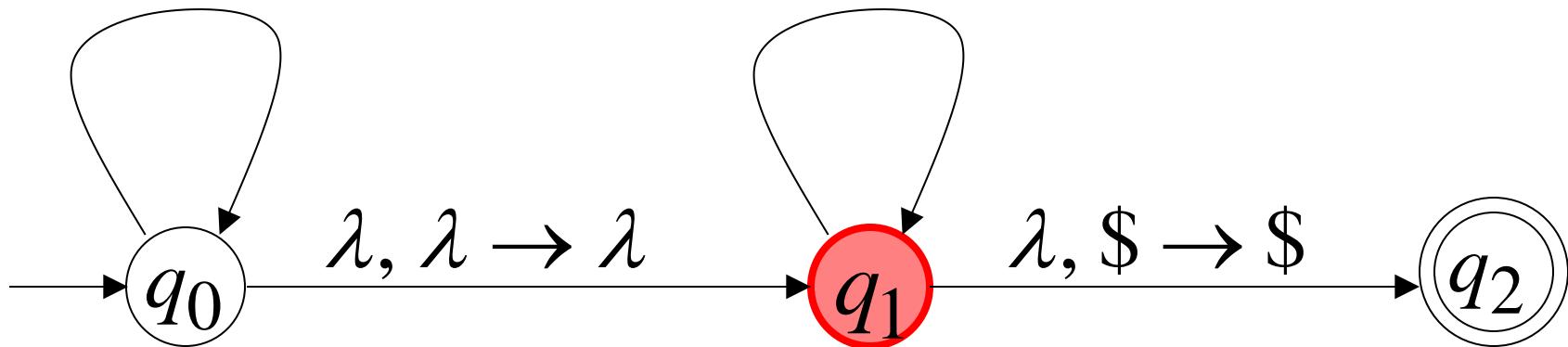
Stack

$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

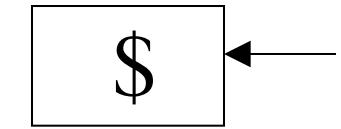
$$b, b \rightarrow \lambda$$



Time 6

Input

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

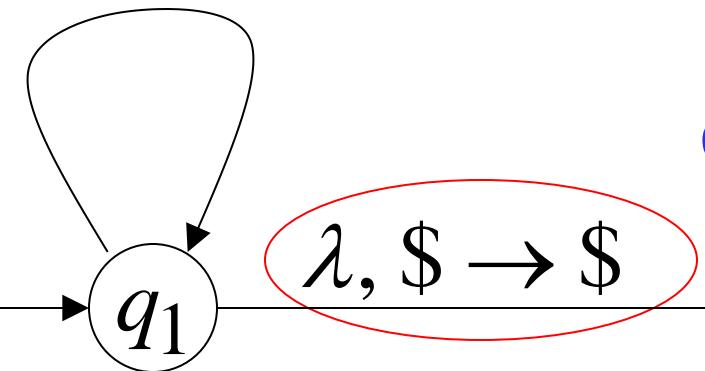
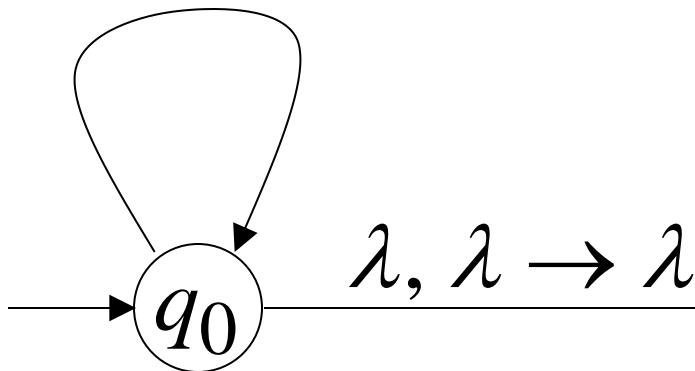


$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

$$b, b \rightarrow \lambda$$



Stack

accept

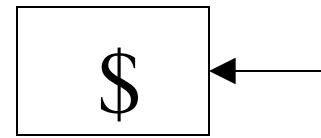


Rejection Example:

Time 0

Input

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

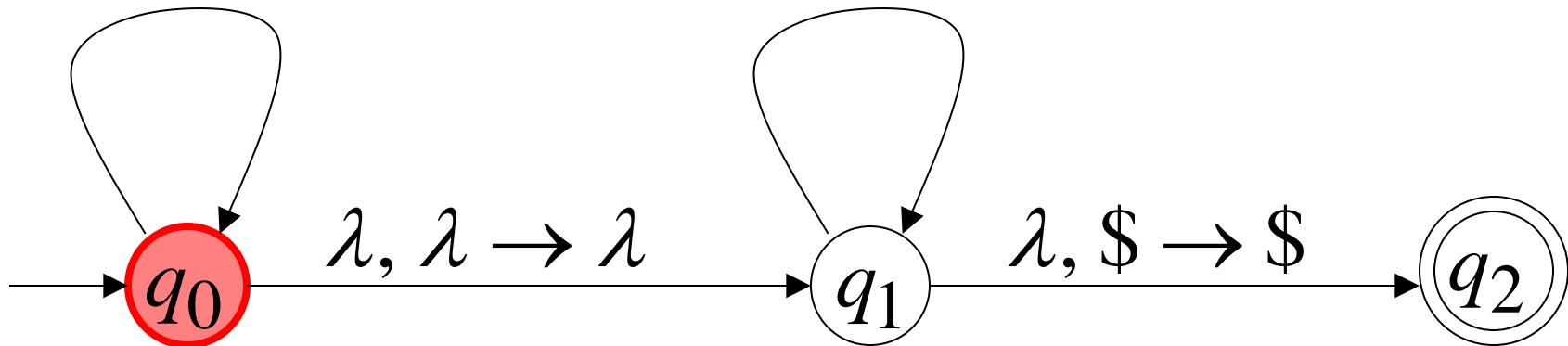


$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

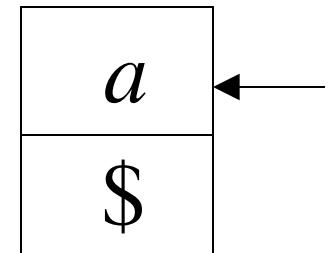
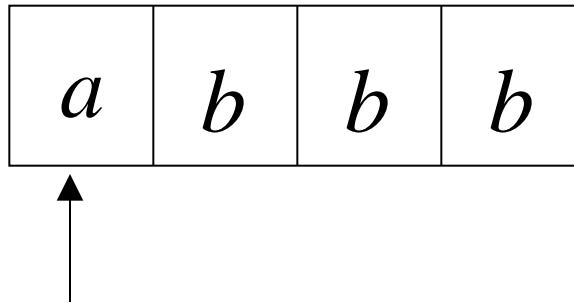
$$b, b \rightarrow \lambda$$



Stack

Time 1

Input



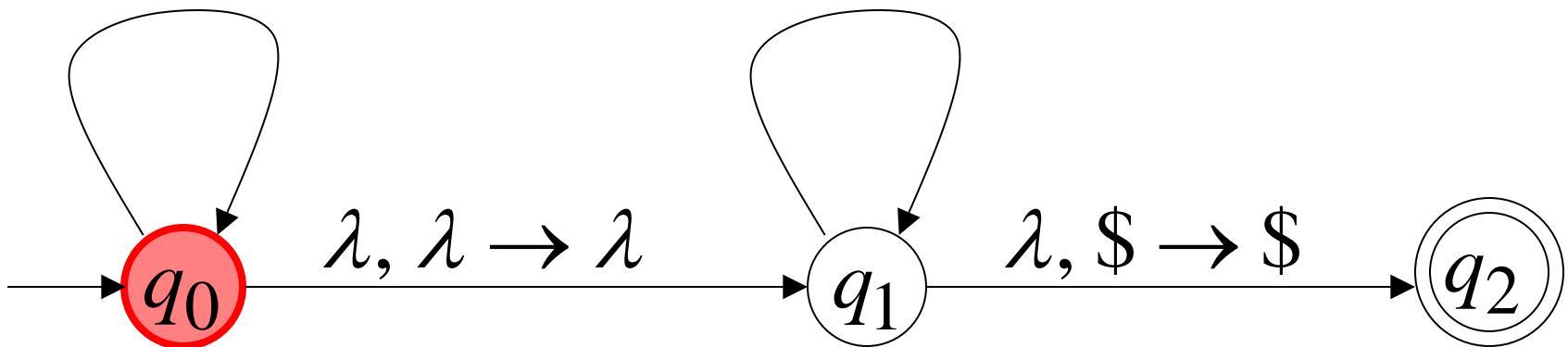
Stack

$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

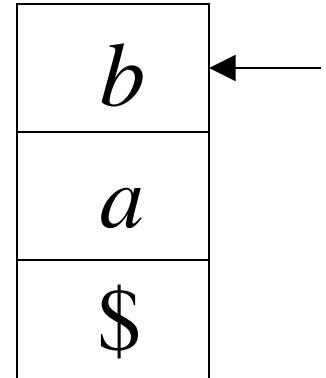
$$b, b \rightarrow \lambda$$



Time 2

Input

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



Stack

$$a, \lambda \rightarrow a$$

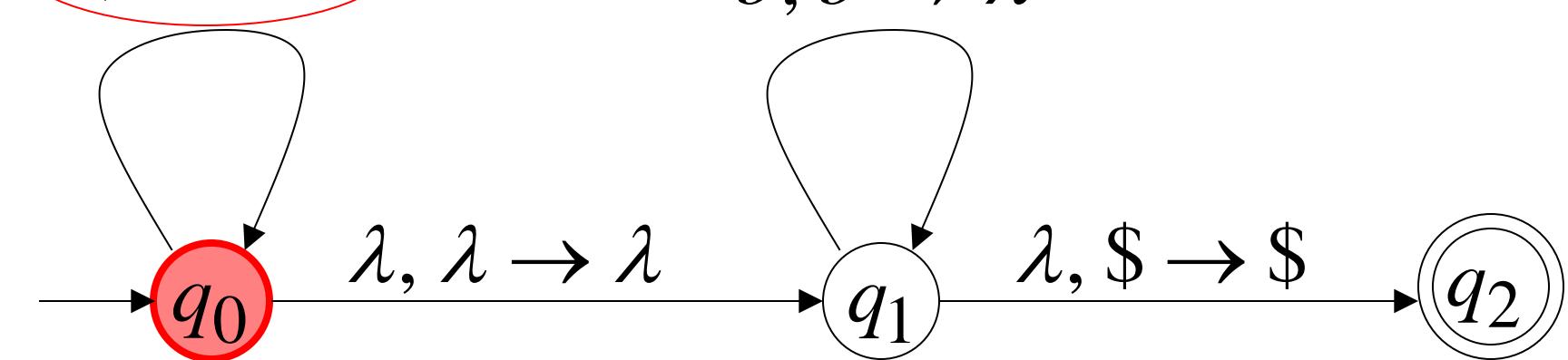
$$b, \lambda \rightarrow b$$

$$a, a \rightarrow \lambda$$

$$b, b \rightarrow \lambda$$

$$\lambda, \lambda \rightarrow \lambda$$

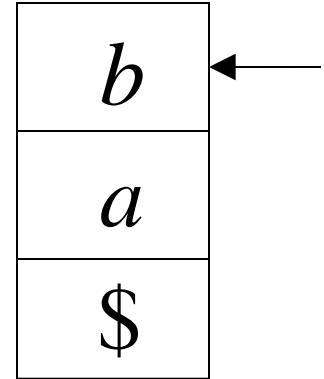
$$\lambda, \$ \rightarrow \$$$



Time 3

Input

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



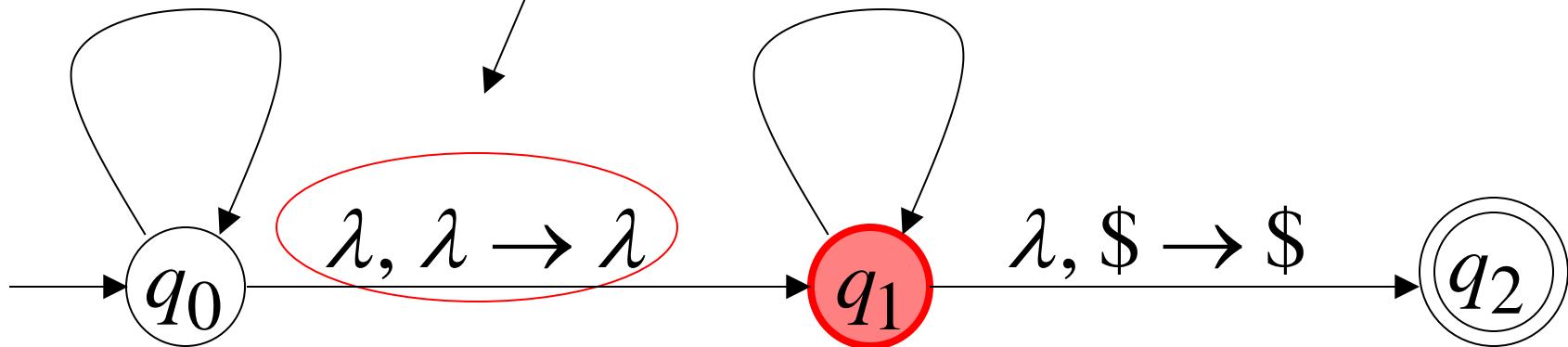
Guess the middle
of string

$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

$$b, b \rightarrow \lambda$$

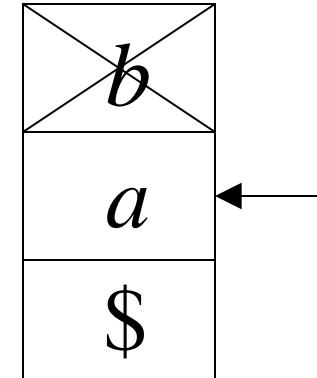


Stack

Time 4

Input

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



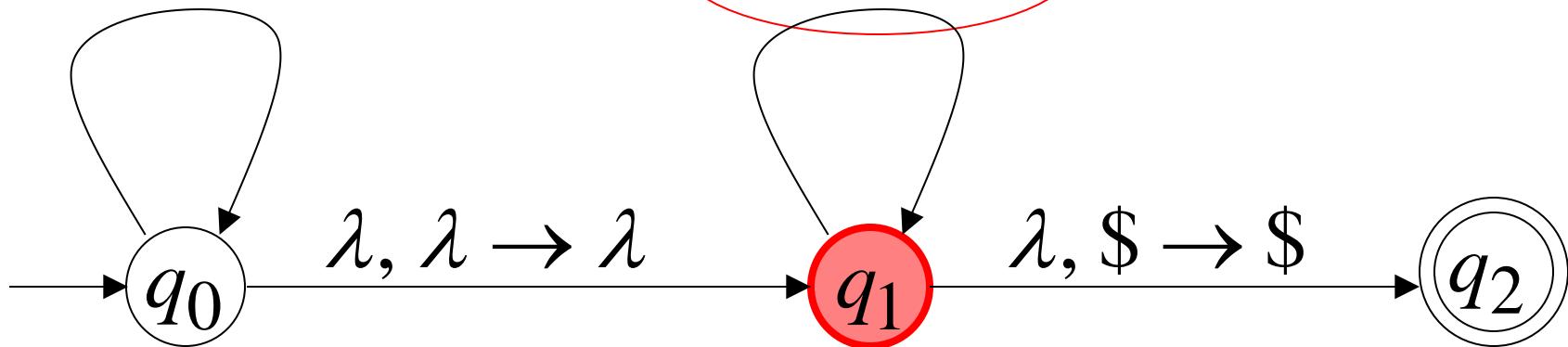
Stack

$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

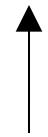
$$b, b \rightarrow \lambda$$



Time 5

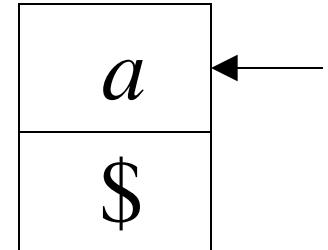
Input

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



There is no possible transition.

Input is not
consumed

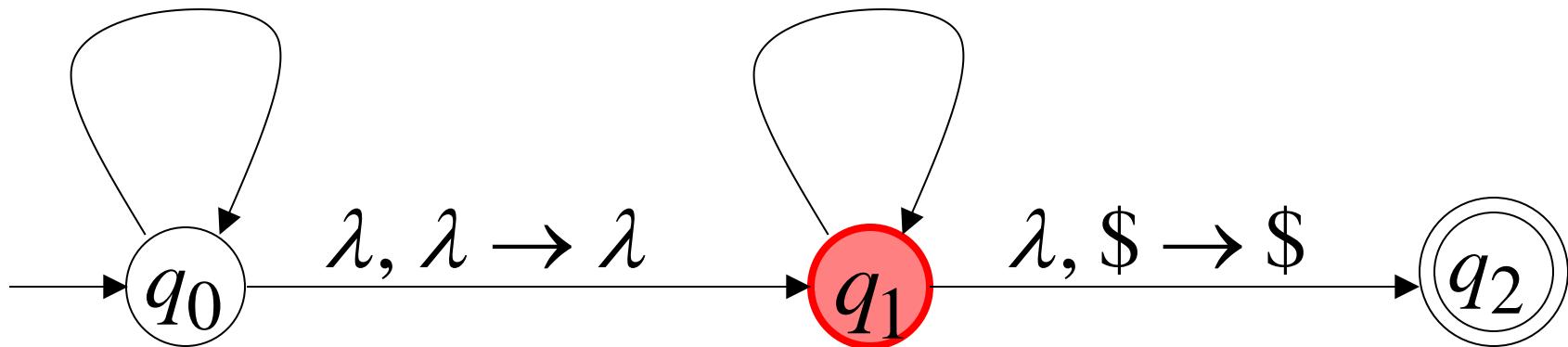


$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

$$b, b \rightarrow \lambda$$



Another computation on same string:

Input

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

Time 0

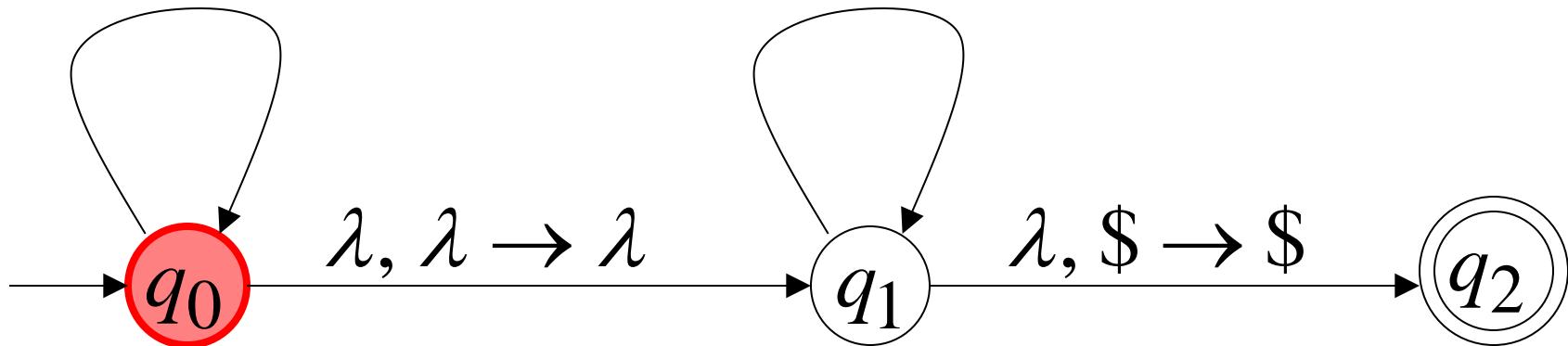


$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

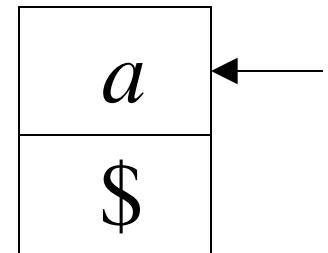
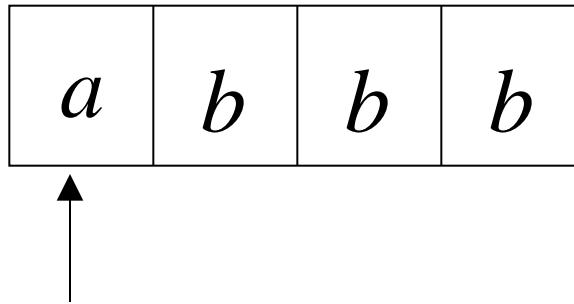
$$b, b \rightarrow \lambda$$



Stack

Time 1

Input



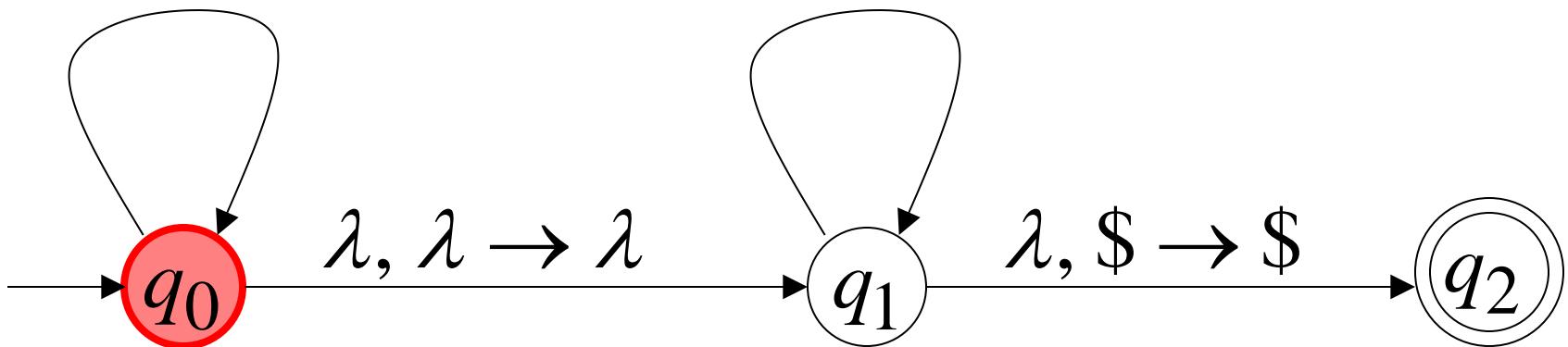
Stack

$$a, \lambda \rightarrow a$$

$$b, \lambda \rightarrow b$$

$$a, a \rightarrow \lambda$$

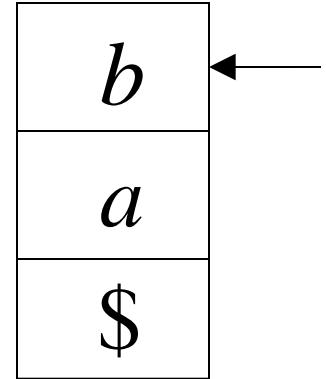
$$b, b \rightarrow \lambda$$



Time 2

Input

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



Stack

$$a, \lambda \rightarrow a$$

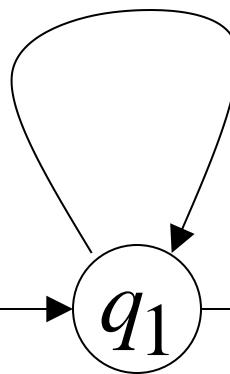
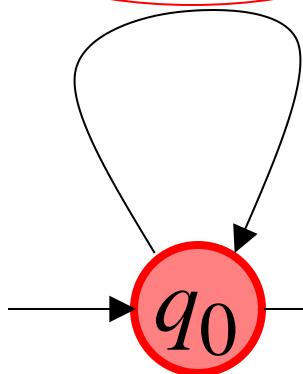
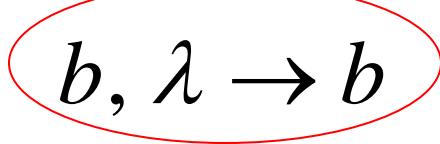
$$b, \lambda \rightarrow b$$

$$\lambda, \lambda \rightarrow \lambda$$

$$a, a \rightarrow \lambda$$

$$b, b \rightarrow \lambda$$

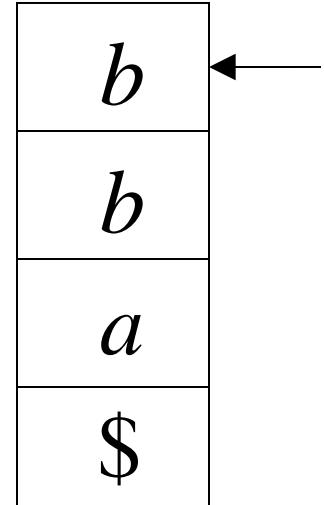
$$\lambda, \$ \rightarrow \$$$



Input

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

Time 3



Stack

$$a, \lambda \rightarrow a$$

$$b, \lambda \rightarrow b$$

$$a, a \rightarrow \lambda$$

$$b, b \rightarrow \lambda$$

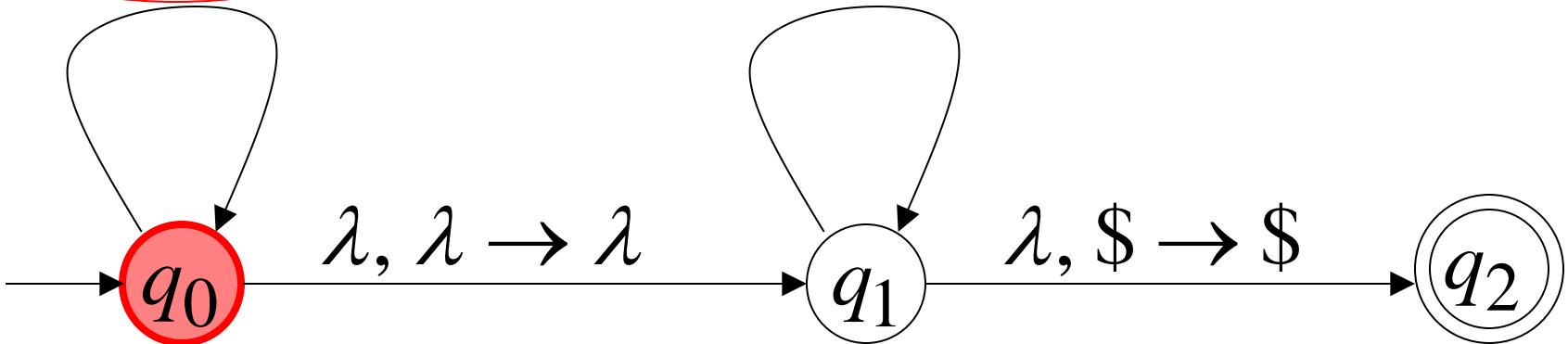
q_0

$$\lambda, \lambda \rightarrow \lambda$$

q_1

$$\lambda, \$ \rightarrow \$$$

q_2



Input

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

Time 4

b
b
b
a
\$

Stack

$$a, \lambda \rightarrow a$$

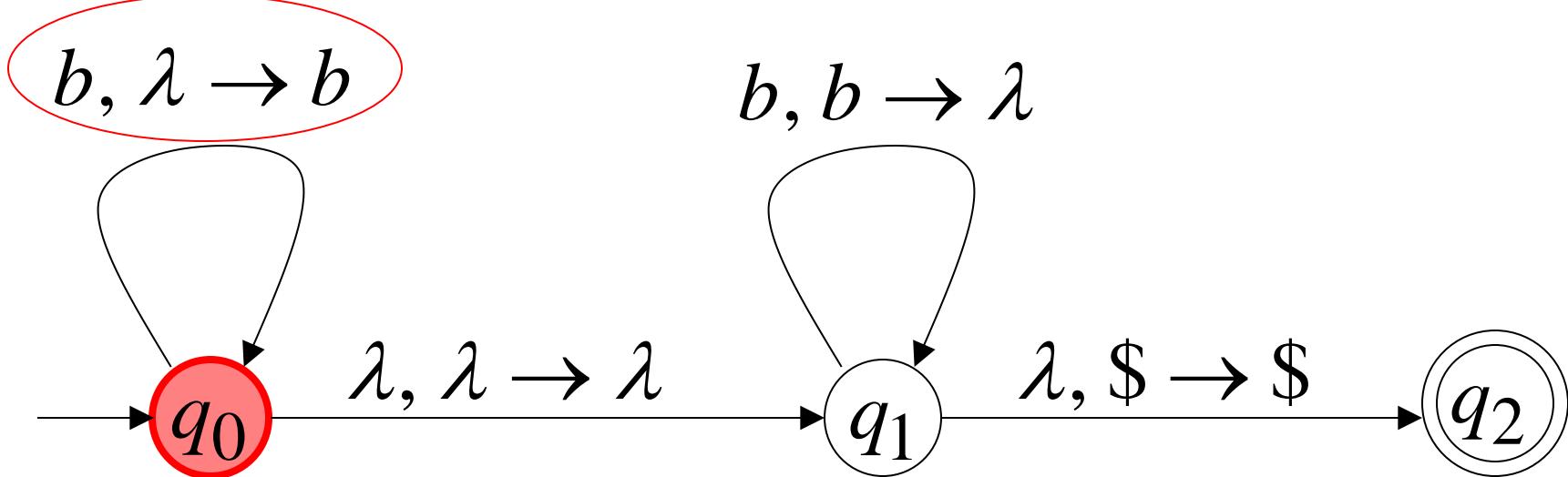
$$b, \lambda \rightarrow b$$

$$a, a \rightarrow \lambda$$

$$b, b \rightarrow \lambda$$

$$\lambda, \lambda \rightarrow \lambda$$

$$\lambda, \$ \rightarrow \$$$



Input

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

Time 5

No accept state
is reached

b
b
b
a
\$

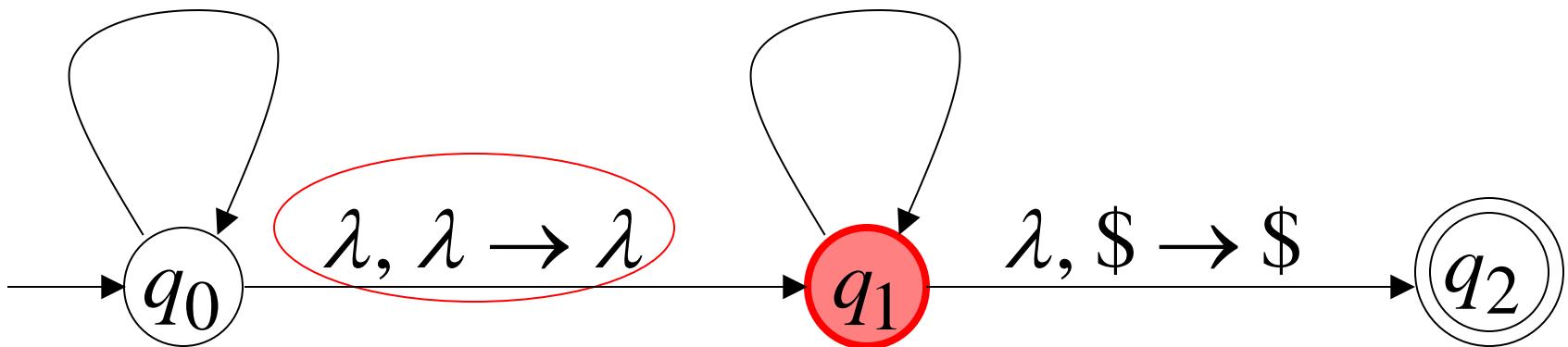
Stack

$$a, \lambda \rightarrow a$$

$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

$$b, b \rightarrow \lambda$$



There is no computation
that accepts string $abbb$

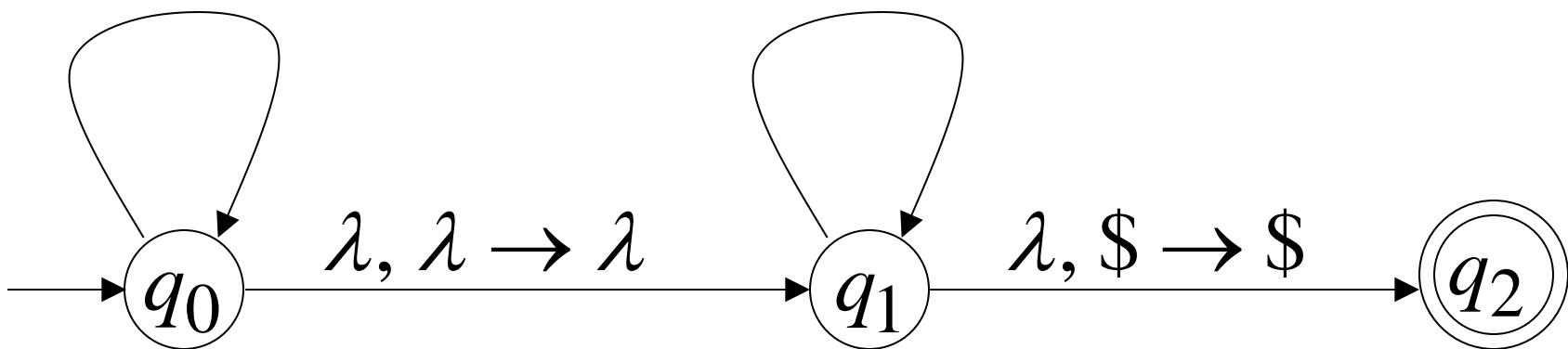
$$abbb \notin L(M)$$

$$a, \lambda \rightarrow a$$

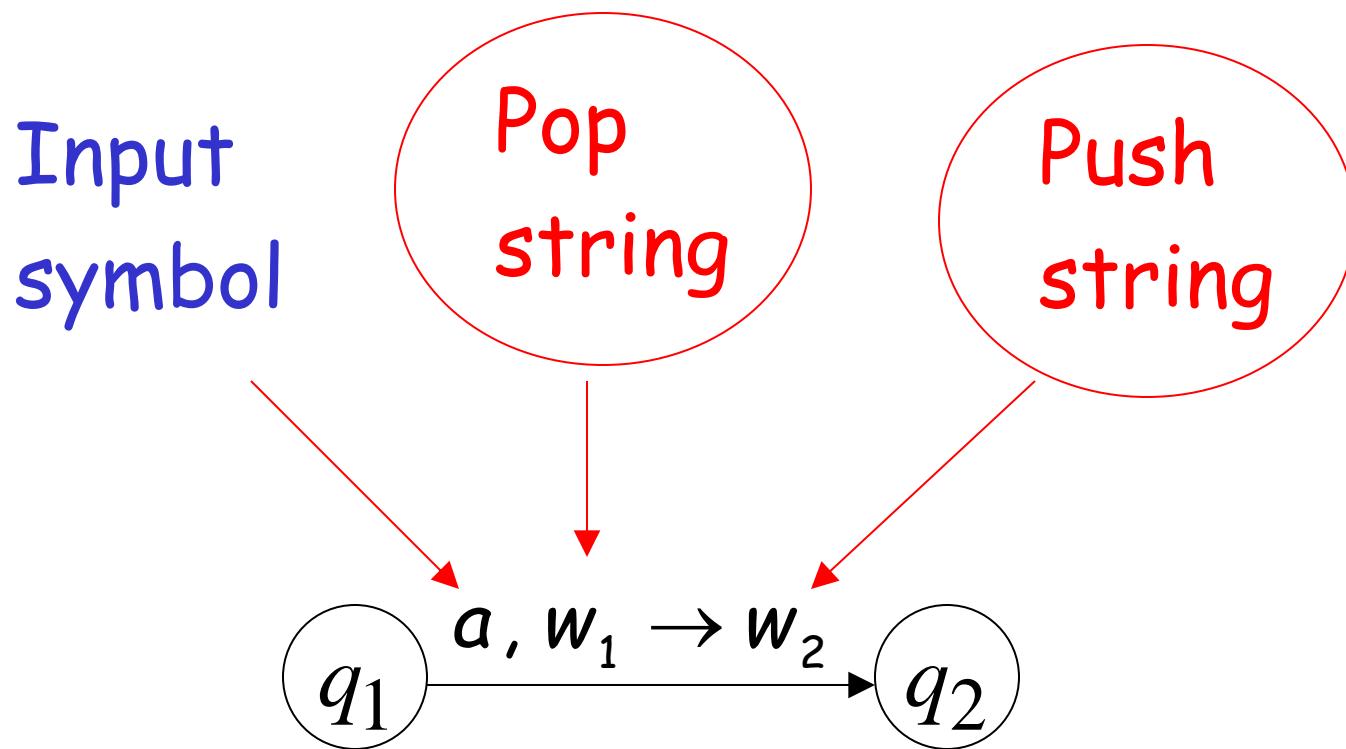
$$a, a \rightarrow \lambda$$

$$b, \lambda \rightarrow b$$

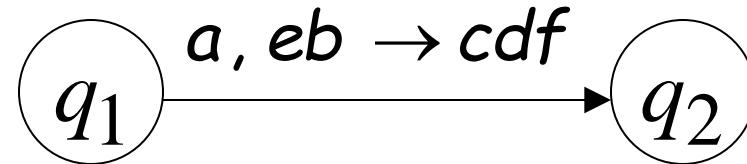
$$b, b \rightarrow \lambda$$



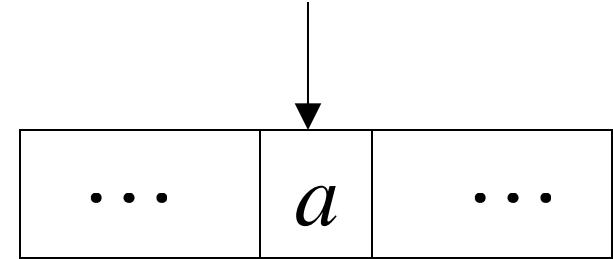
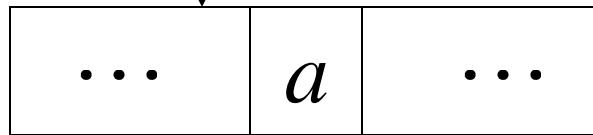
Pushing & Popping Strings



Example:



input



stack

pop
string

<i>e</i>
<i>b</i>
<i>h</i>
<i>e</i>
<i>\$</i>

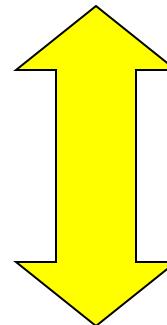
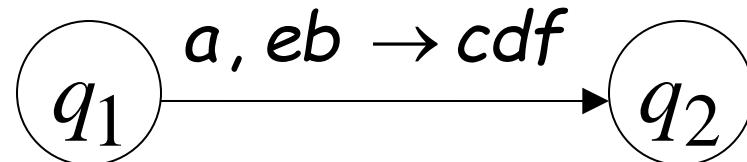
top

Replace

top

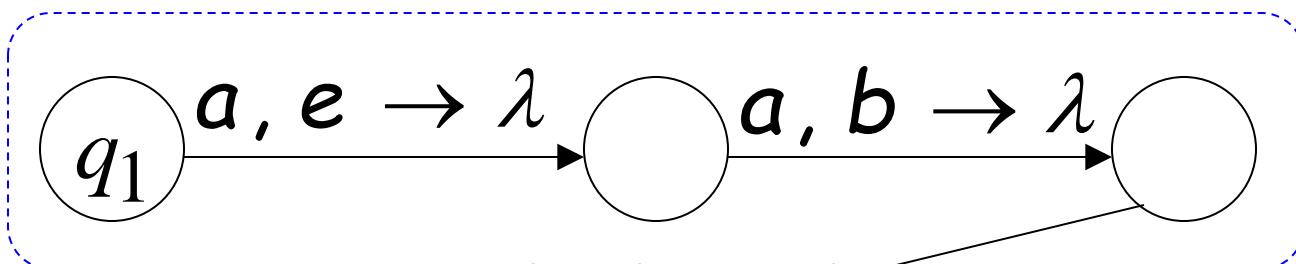
<i>c</i>
<i>d</i>
<i>f</i>
<i>h</i>
<i>e</i>
<i>\$</i>

push
string

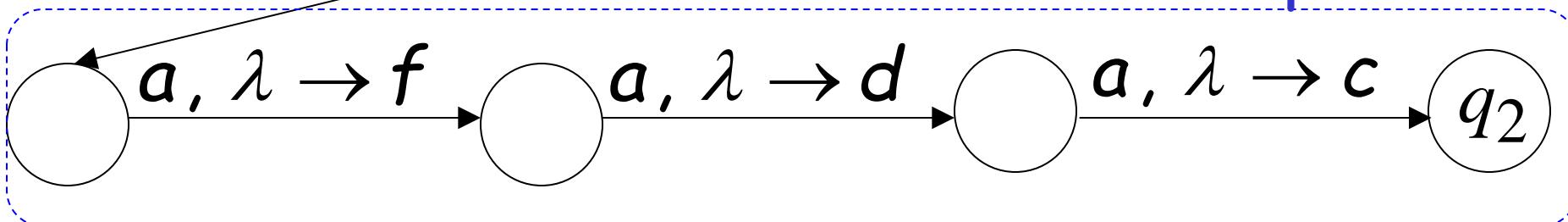


Equivalent
transitions

pop



push



Another PDA example

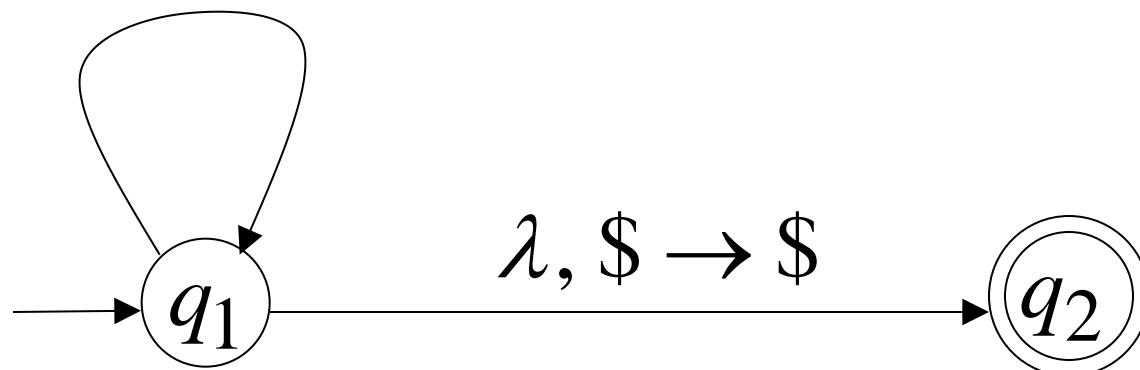
$$L(M) = \{ w \in \{a, b\}^*: n_a(w) = n_b(w) \}$$

PDA M

$$a, \$ \rightarrow 0\$ \quad b, \$ \rightarrow 1\$$$

$$a, 0 \rightarrow 00 \quad b, 1 \rightarrow 11$$

$$a, 1 \rightarrow \lambda \quad b, 0 \rightarrow \lambda$$



Execution Example:

Time 0

Input

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



$a, \$ \rightarrow 0\$$

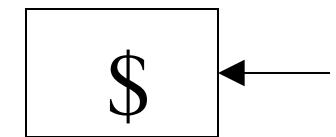
$b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$

$b, 1 \rightarrow 11$

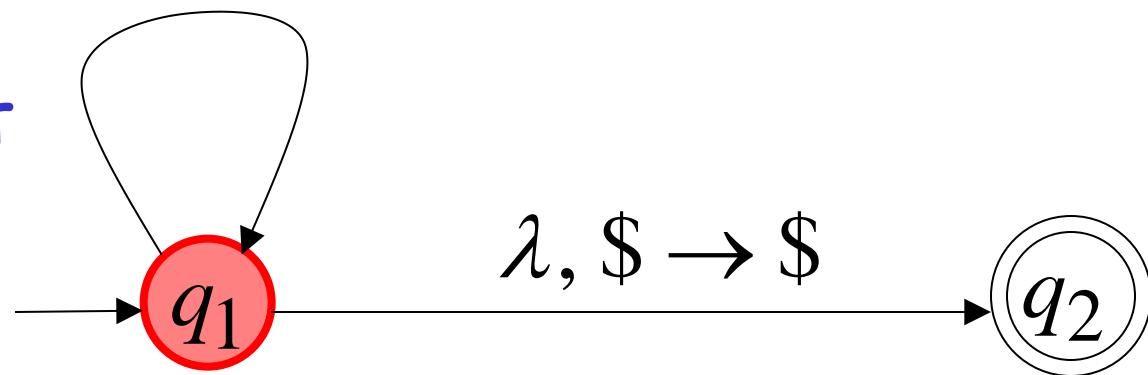
$a, 1 \rightarrow \lambda$

$b, 0 \rightarrow \lambda$



Stack

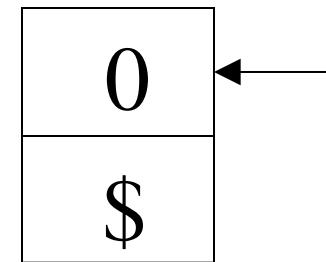
current
state



Time 1

Input

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



Stack

$a, \$ \rightarrow 0\$$

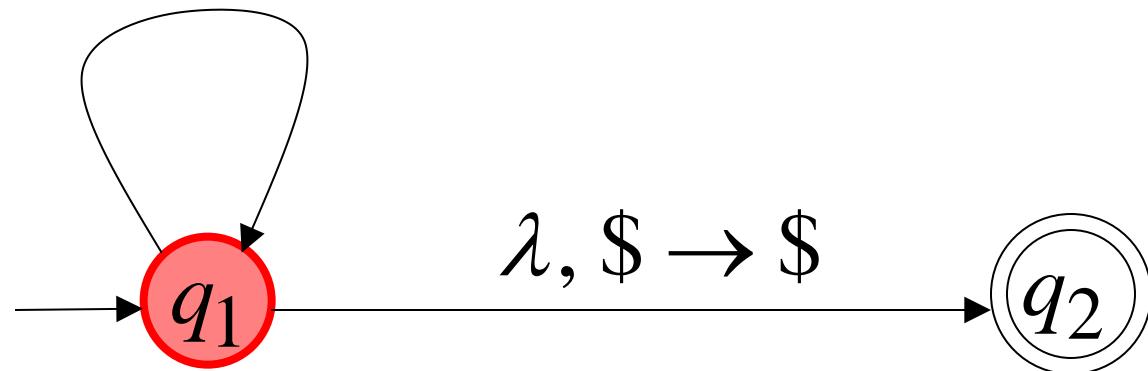
$b, \$ \rightarrow 1\$$

$a, 0 \rightarrow 00$

$b, 1 \rightarrow 11$

$a, 1 \rightarrow \lambda$

$b, 0 \rightarrow \lambda$



Time 3

Input

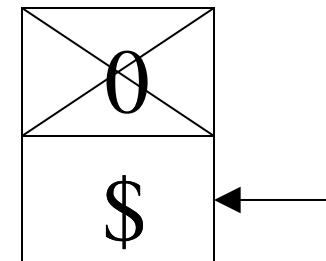
a	b	b	b	a	a
-----	-----	-----	-----	-----	-----



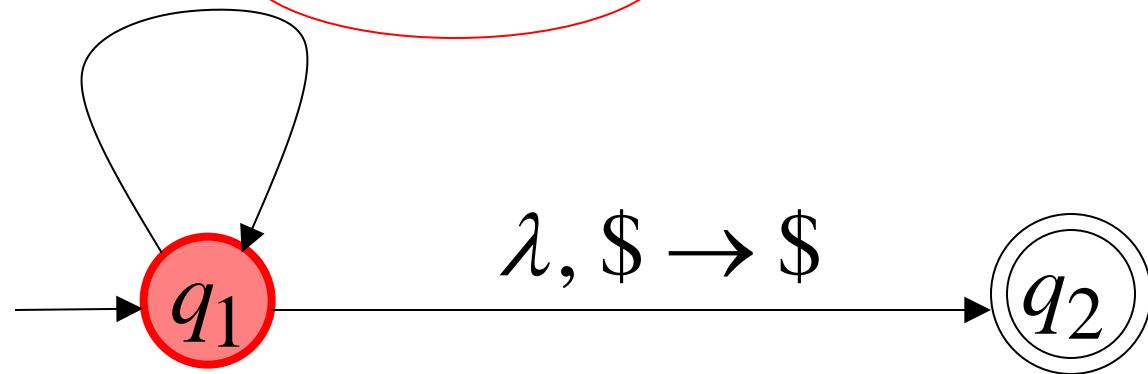
$$a, \$ \rightarrow 0\$ \quad b, \$ \rightarrow 1\$$$

$$a, 0 \rightarrow 00 \quad b, 1 \rightarrow 11$$

$$a, 1 \rightarrow \lambda \quad b, 0 \rightarrow \lambda$$



Stack



Time 4

Input

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



$a, \$ \rightarrow 0\$$

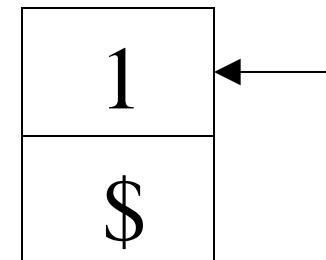
$a, 0 \rightarrow 00$

$a, 1 \rightarrow \lambda$

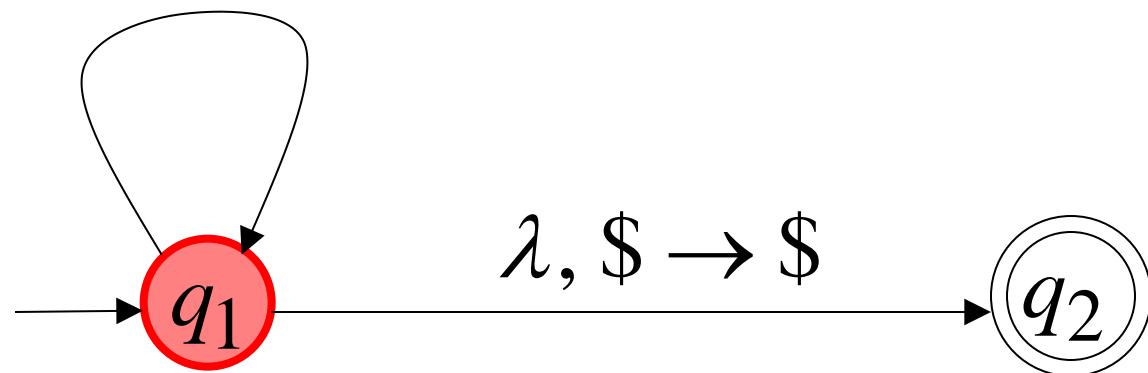
$b, \$ \rightarrow 1\$$

$b, 1 \rightarrow 11$

$b, 0 \rightarrow \lambda$



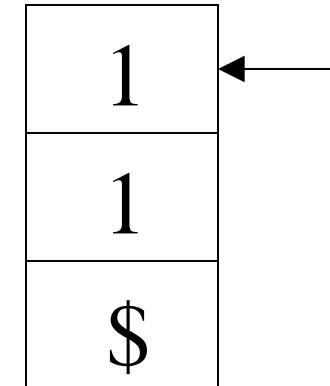
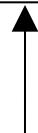
Stack



Time 5

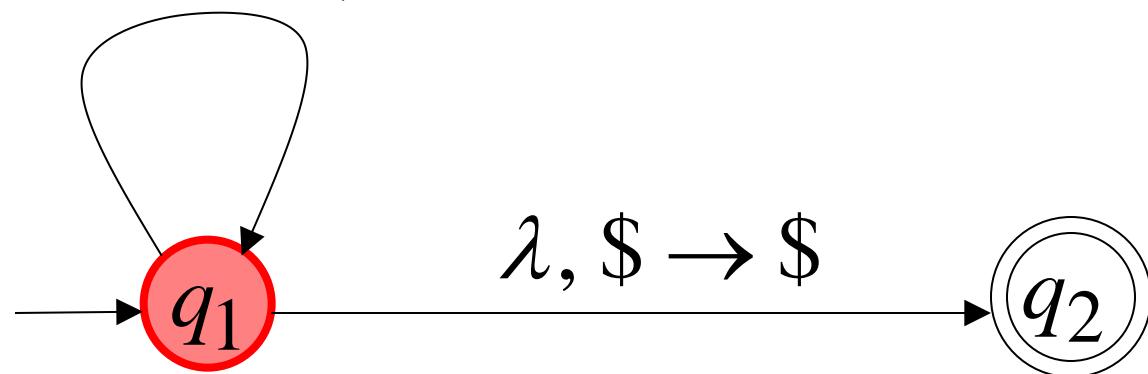
Input

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



Stack

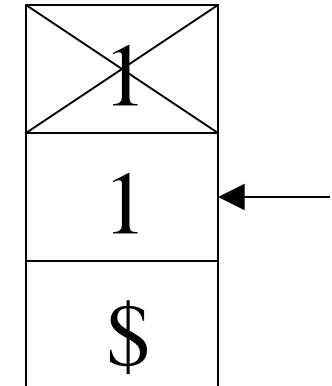
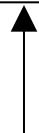
$$\begin{array}{ll} a, \$ \rightarrow 0\$ & b, \$ \rightarrow 1\$ \\ a, 0 \rightarrow 00 & \text{b, 1} \rightarrow 11 \\ a, 1 \rightarrow \lambda & b, 0 \rightarrow \lambda \end{array}$$



Time 6

Input

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

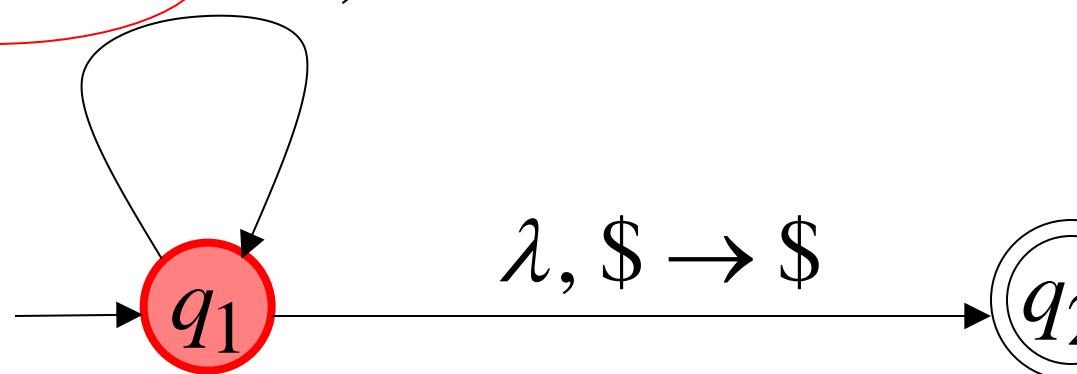


Stack

$$a, \$ \rightarrow 0\$ \quad b, \$ \rightarrow 1\$$$

$$a, 0 \rightarrow 00 \quad b, 1 \rightarrow 11$$

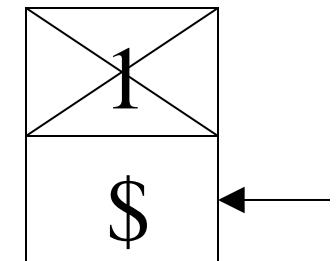
$$a, 1 \rightarrow \lambda \quad b, 0 \rightarrow \lambda$$



Time 7

Input

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

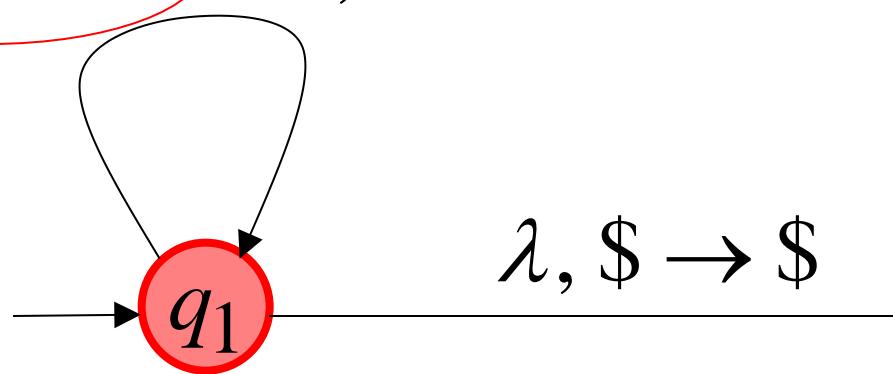


Stack

$$a, \$ \rightarrow 0\$ \quad b, \$ \rightarrow 1\$$$

$$a, 0 \rightarrow 00 \quad b, 1 \rightarrow 11$$

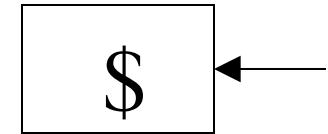
$$a, 1 \rightarrow \lambda \quad b, 0 \rightarrow \lambda$$



Time 8

Input

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

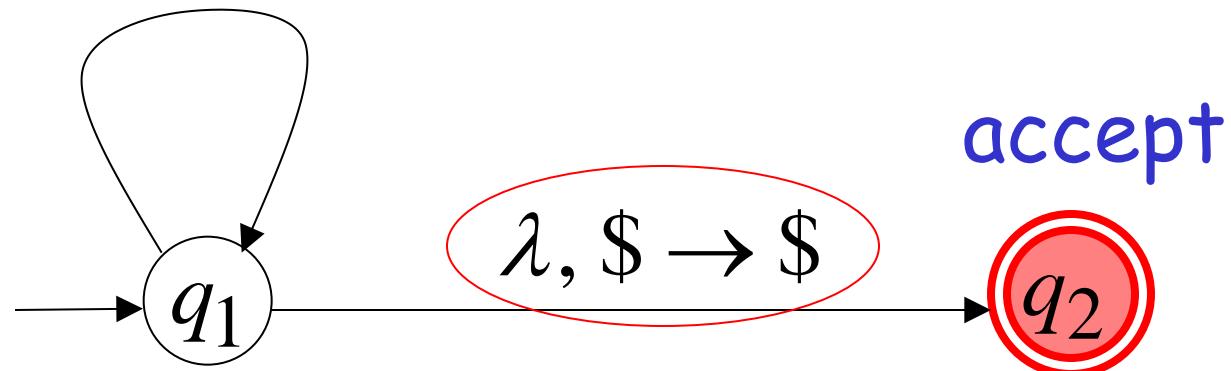


Stack

$$a, \$ \rightarrow 0\$ \quad b, \$ \rightarrow 1\$$$

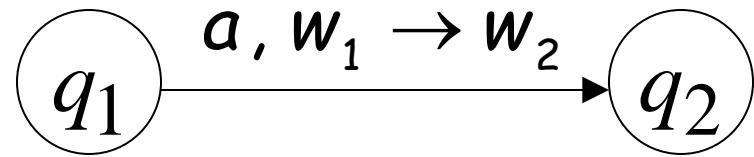
$$a, 0 \rightarrow 00 \quad b, 1 \rightarrow 11$$

$$a, 1 \rightarrow \lambda \quad b, 0 \rightarrow \lambda$$



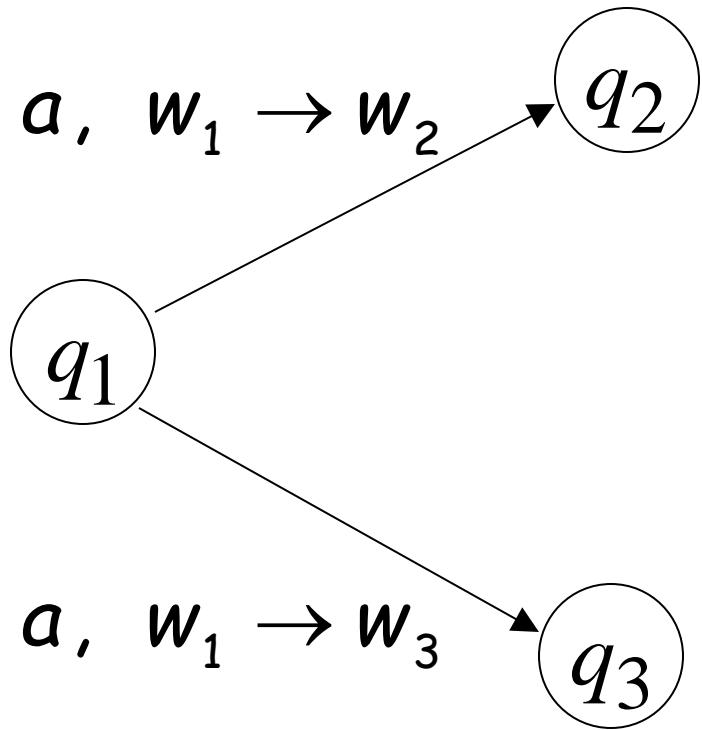
accept

Formalities for PDAs



Transition function:

$$\delta(q_1, a, w_1) = \{(q_2, w_2)\}$$

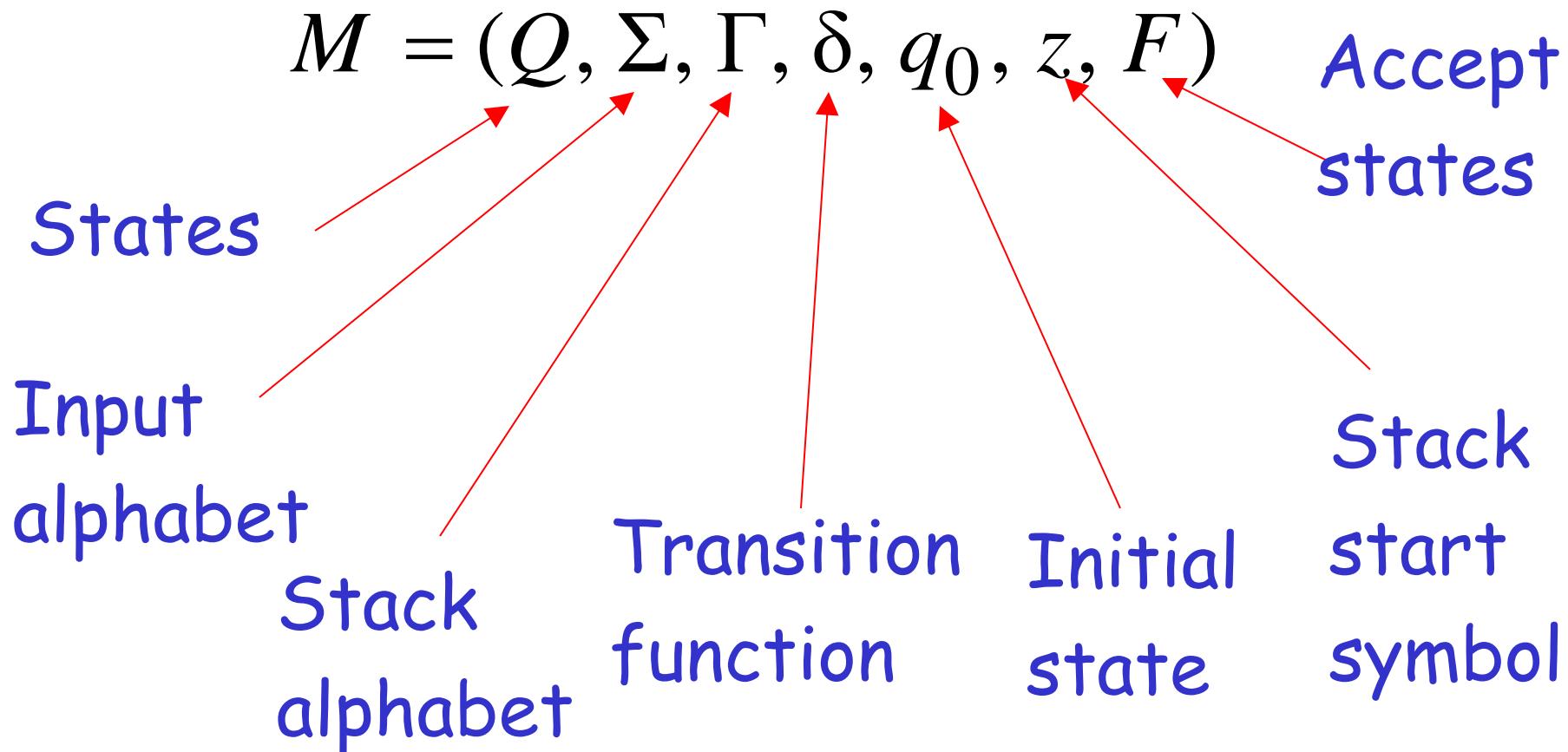


Transition function:

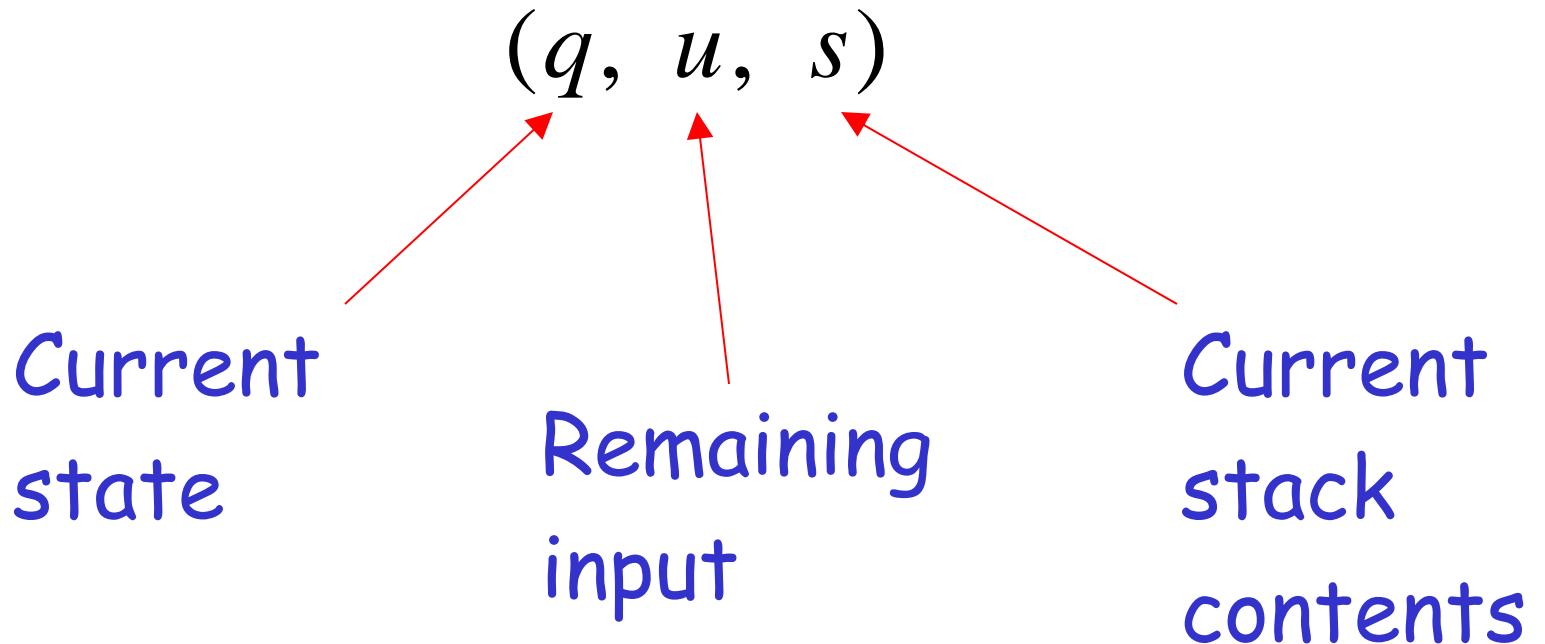
$$\delta(q_1, a, w_1) = \{(q_2, w_2), (q_3, w_3)\}$$

Formal Definition

Pushdown Automaton (PDA)



Instantaneous Description



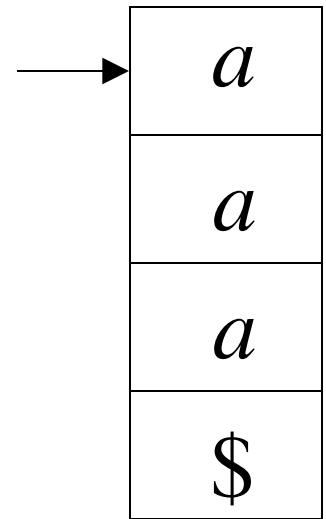
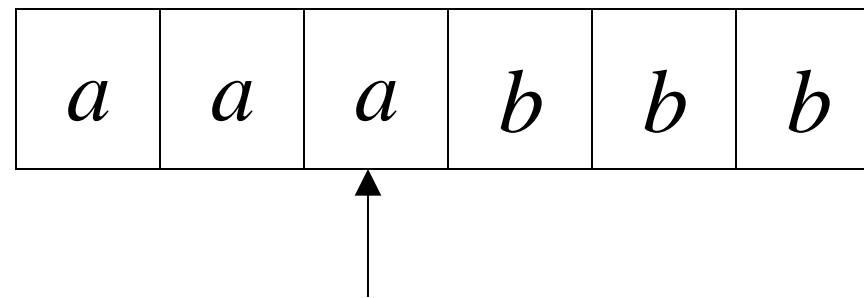
Example:

Instantaneous Description

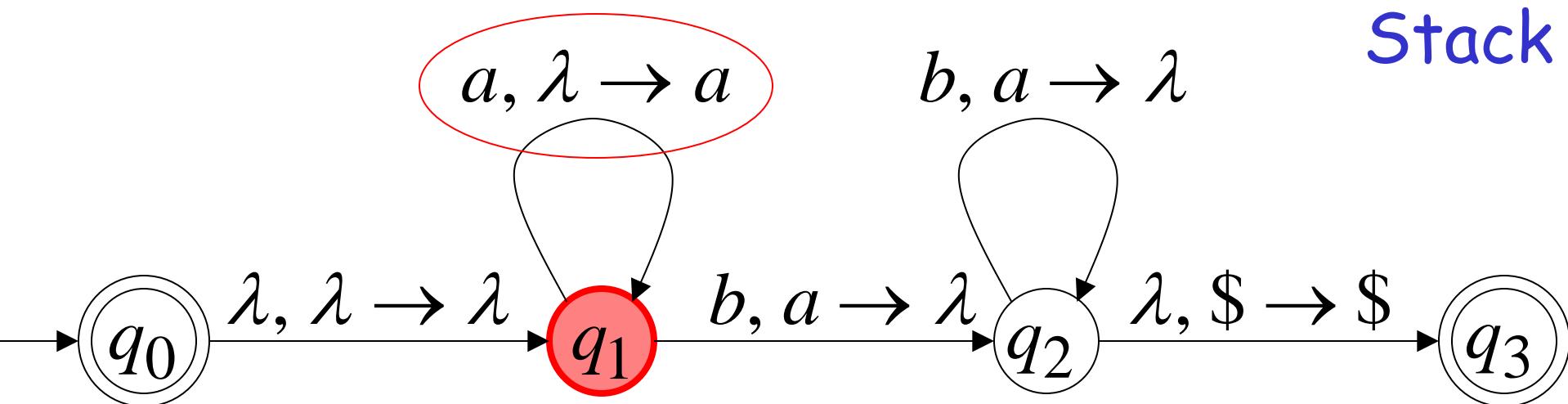
$(q_1, bbb, aaa\$)$

Time 4:

Input



Stack



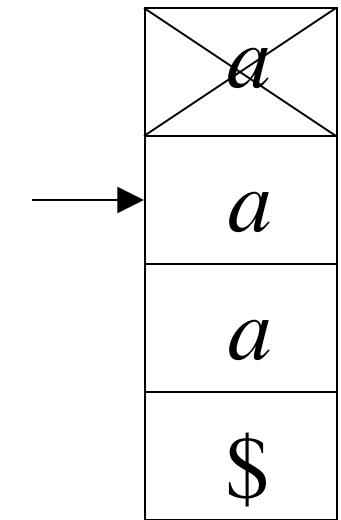
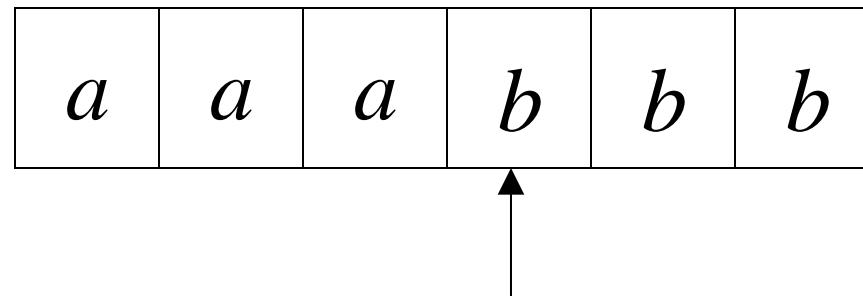
Example:

Instantaneous Description

$(q_2, bb, aa\$)$

Time 5:

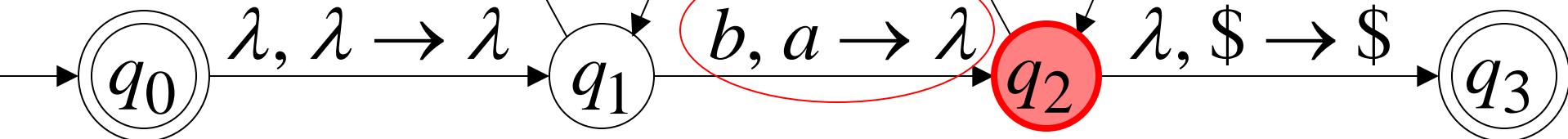
Input



Stack

$a, \lambda \rightarrow a$

$b, a \rightarrow \lambda$



We write:

$$(q_1, bbb, aaa\$) \succ (q_2, bb, aa\$)$$

Time 4

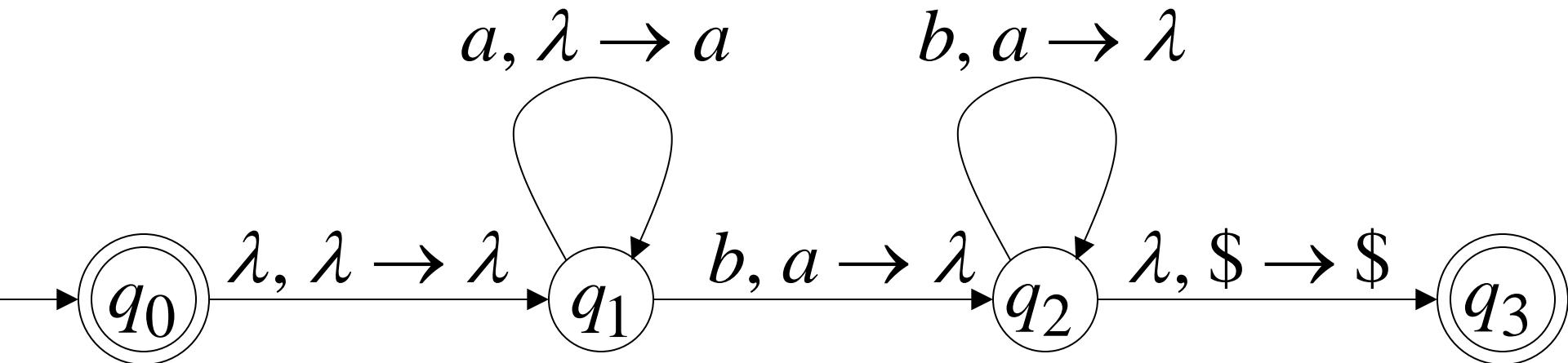
Time 5

A computation:

$(q_0, aaabbb, \$) \succ (q_1, aaabbb, \$) \succ$

$(q_1, aabbb, a\$) \succ (q_1, abbb, aa\$) \succ (q_1, bbb, aaa\$) \succ$

$(q_2, bb, aa\$) \succ (q_2, b, a\$) \succ (q_2, \lambda, \$) \succ (q_3, \lambda, \$)$



$$(q_0, aaabbb, \$) \succ (q_1, aaabbb, \$) \succ$$
$$(q_1, aabbb, a\$) \succ (q_1, abbb, aa\$) \succ (q_1, bbb, aaa\$) \succ$$
$$(q_2, bb, aa\$) \succ (q_2, b, a\$) \succ (q_2, \lambda, \$) \succ (q_3, \lambda, \$)$$

For convenience we write:

$$(q_0, aaabbb, \$) \stackrel{*}{\succ} (q_3, \lambda, \$)$$

Language of PDA

Language $L(M)$ accepted by PDA M :

$$L(M) = \{w : (q_0, w, z) \xrightarrow{*} (q_f, \lambda, s)\}$$

Initial state

Accept state

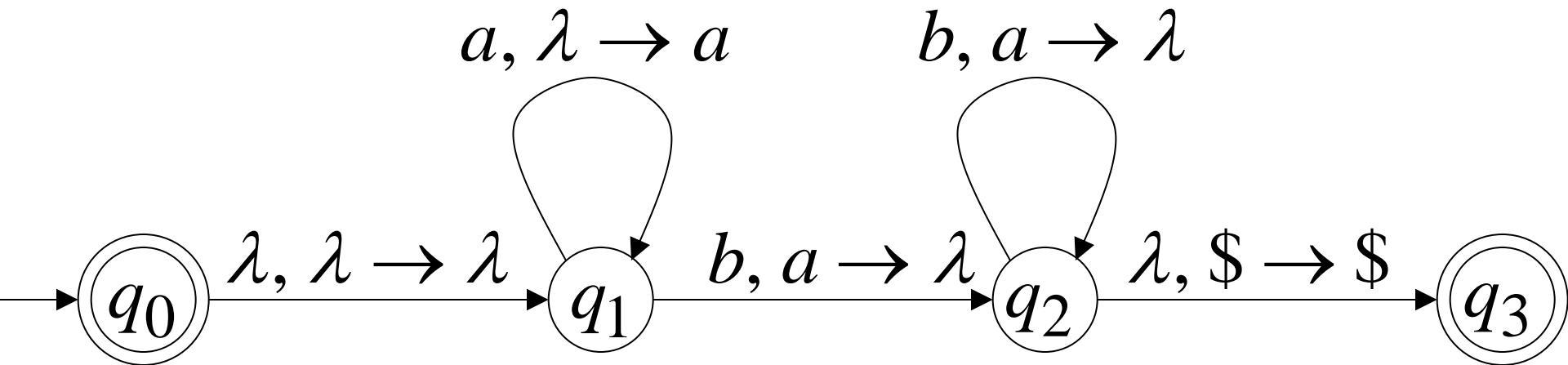
Example:

$$(q_0, aaabbbb, \$) \xrightarrow{*} (q_3, \lambda, \$)$$

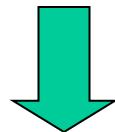


$$aaabbbb \in L(M)$$

PDA M :

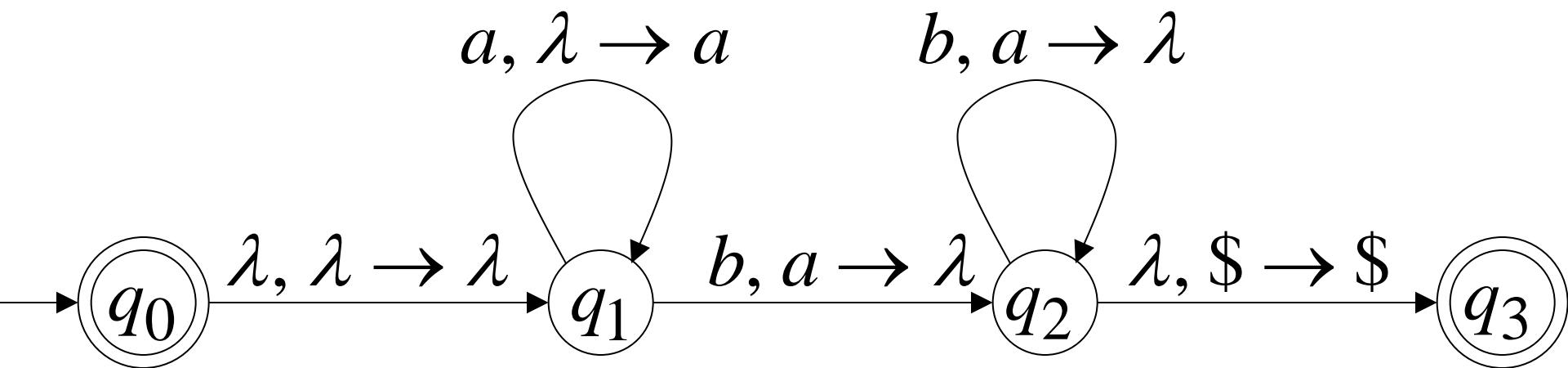


$$(q_0, a^n b^n, \$) \xrightarrow{*} (q_3, \lambda, \$)$$



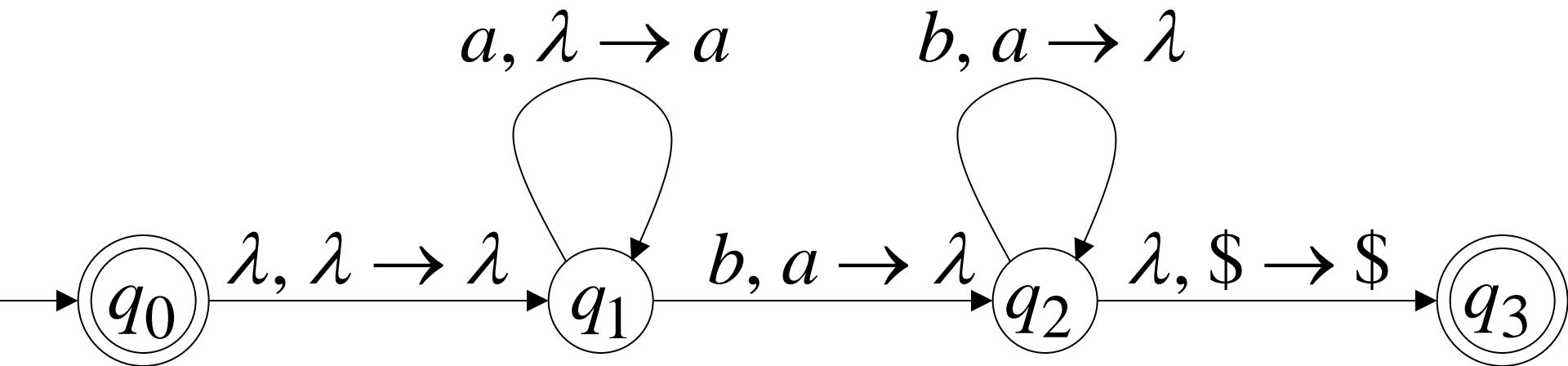
$$a^n b^n \in L(M)$$

PDA M :



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

PDA M :



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:

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

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

$$A \rightarrow w$$

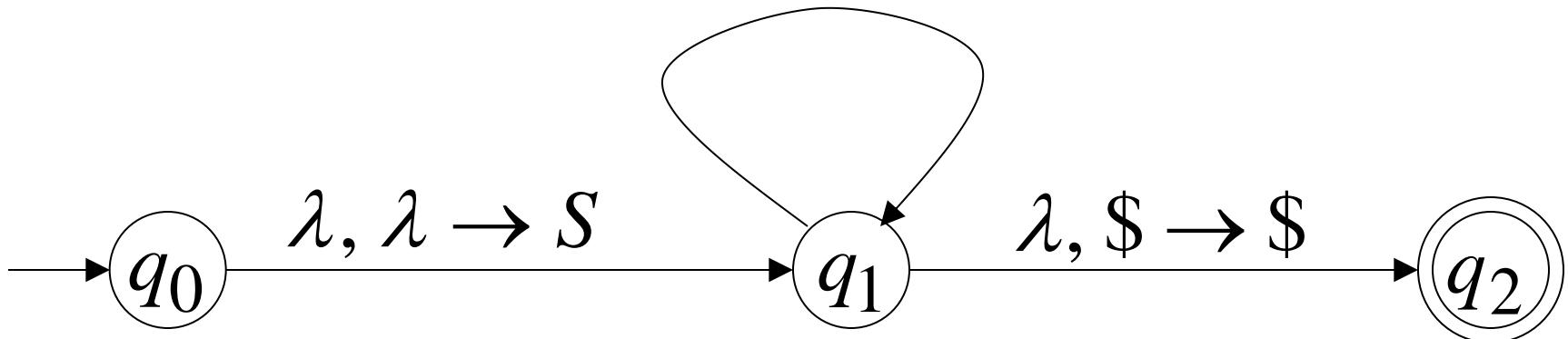
$$\lambda, A \rightarrow w$$

For each terminal in G

$$a$$

$$a, a \rightarrow \lambda$$

Add transitions



Grammar

$$S \rightarrow aSTb$$

$$S \rightarrow b$$

$$T \rightarrow Ta$$

$$T \rightarrow \lambda$$

Example

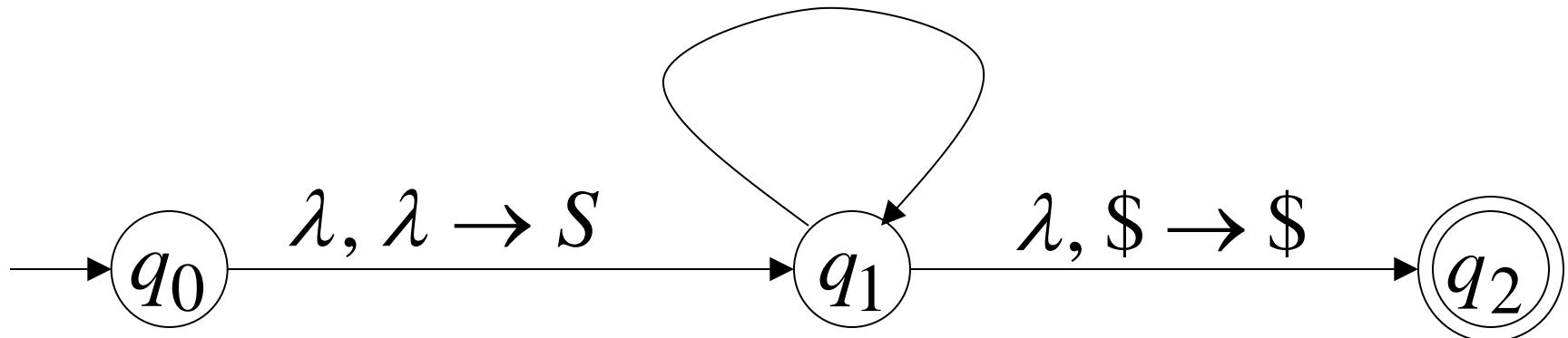
PDA

$$\lambda, S \rightarrow aSTb$$

$$\lambda, S \rightarrow b$$

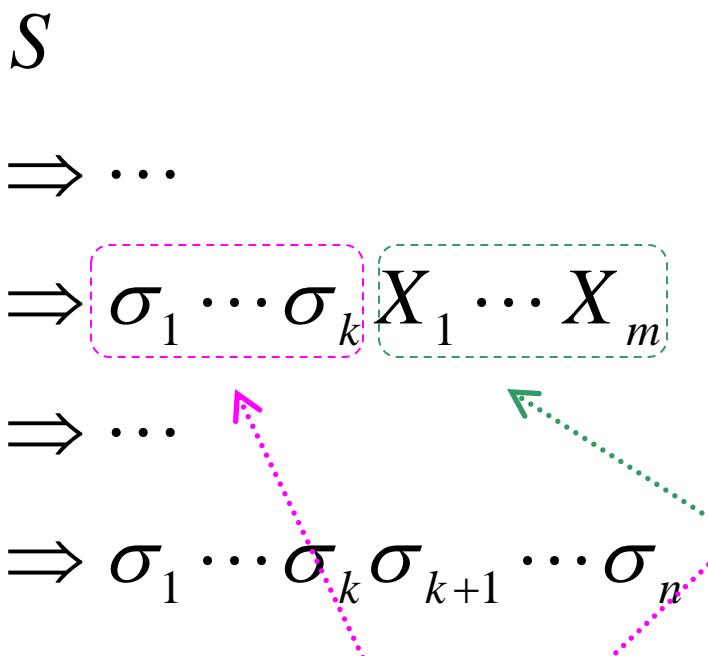
$$\lambda, T \rightarrow Ta \qquad a, a \rightarrow \lambda$$

$$\lambda, T \rightarrow \lambda \qquad b, b \rightarrow \lambda$$

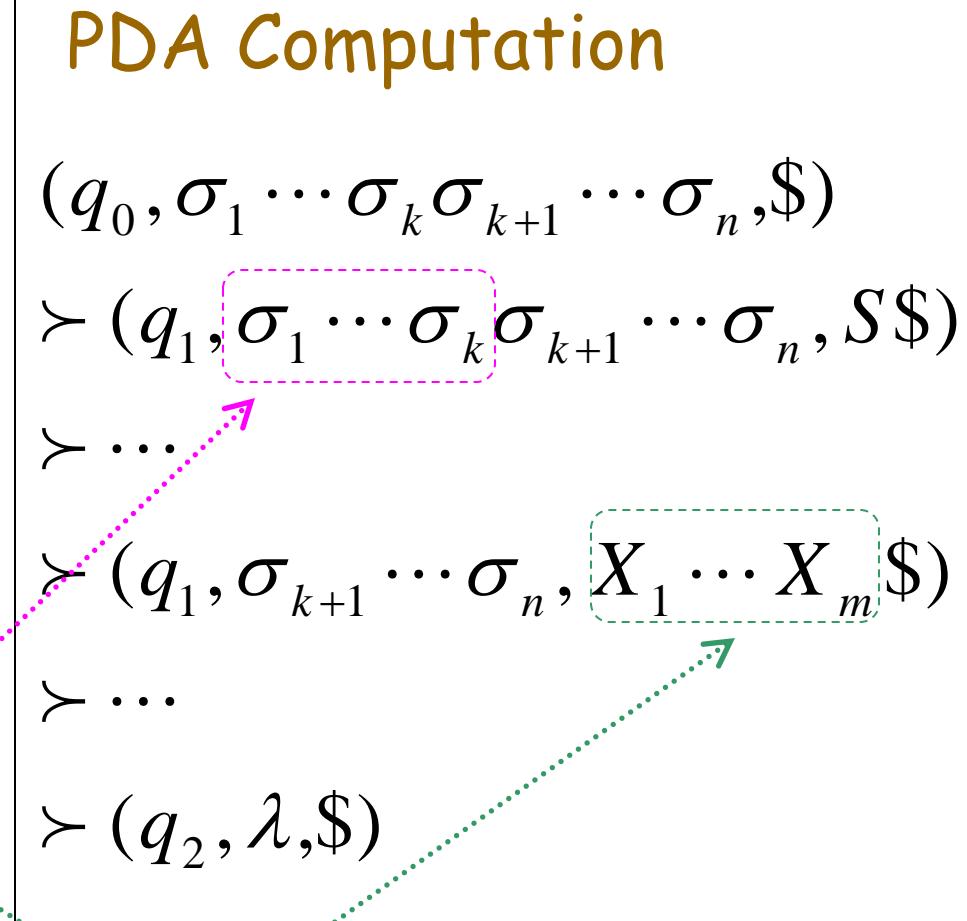


PDA simulates leftmost derivations

Grammar Leftmost Derivation



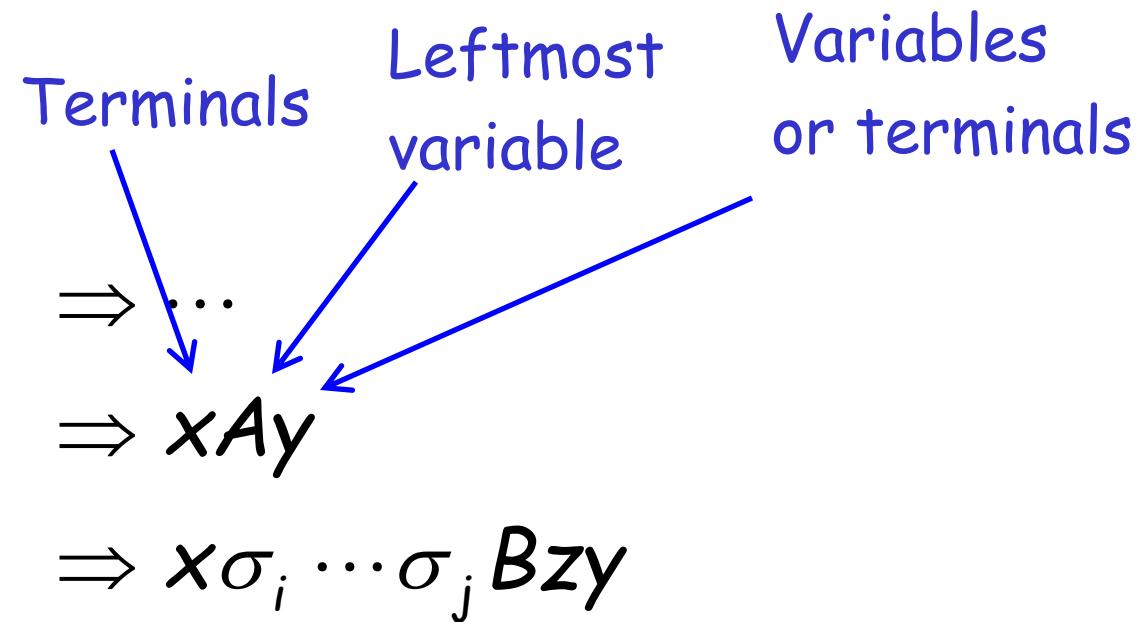
Scanned
symbols



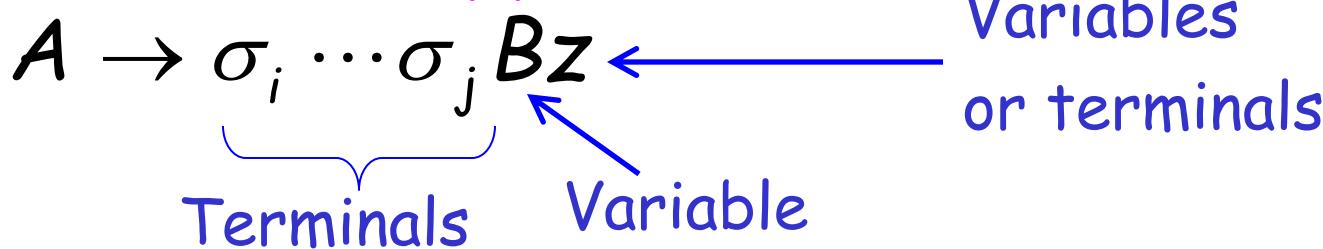
Stack
contents

Grammar

Leftmost Derivation



Production applied



Grammar

Leftmost Derivation

$\Rightarrow \dots$

$\Rightarrow xAy$

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

PDA Computation

$\succ \dots$

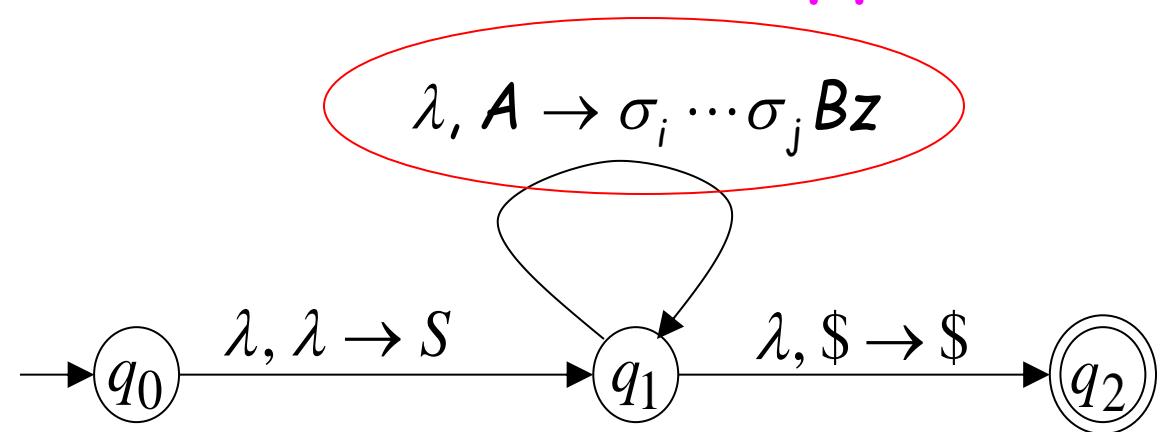
$\succ (q_1, \sigma_i \cdots \sigma_n, Ay \$)$

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

Production applied

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

Transition applied



Grammar

Leftmost Derivation

$\Rightarrow \dots$

$\Rightarrow xAy$

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

$\succ \dots$

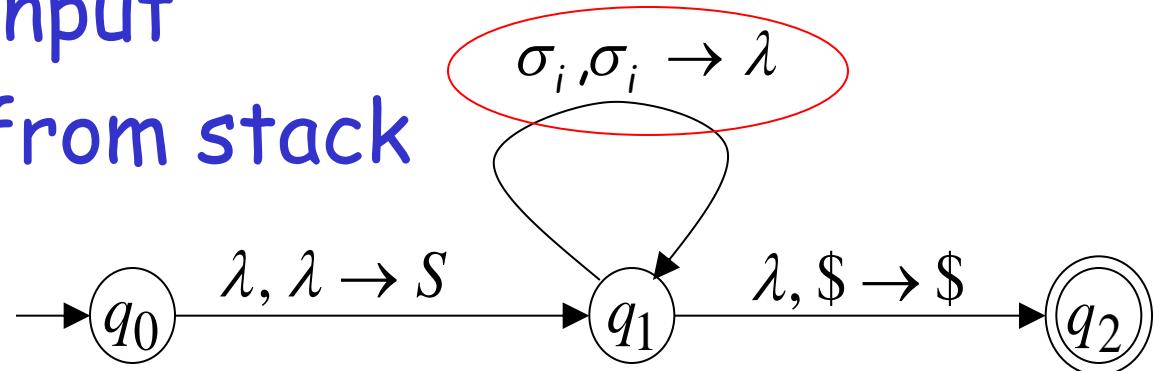
$\succ (q_1, \sigma_i \dots \sigma_n, Ay \$)$

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

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

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

$\vdash \dots$

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

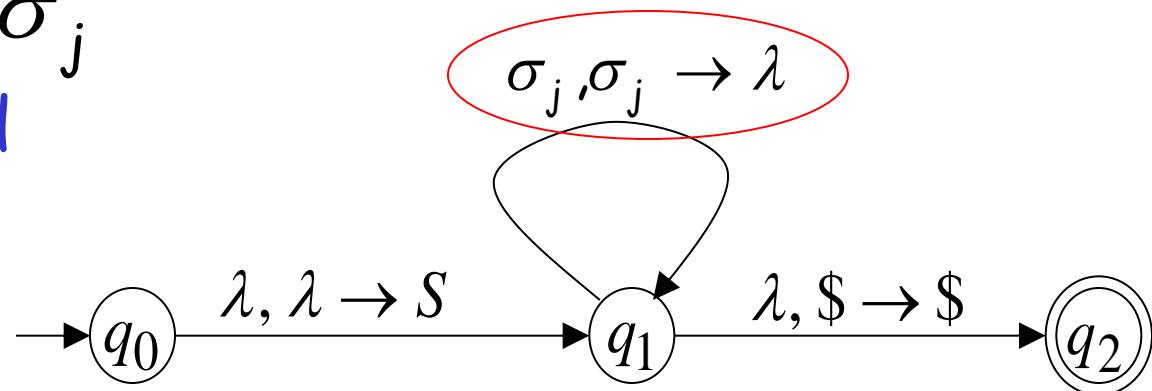
$\vdash (q_1, \sigma_i \dots \sigma_n, \sigma_i \dots \sigma_j Bzy \$)$

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

$\vdash \dots$

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

Last Transition applied



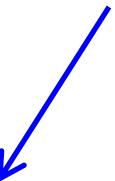
The process repeats with the next leftmost variable

$\Rightarrow \dots$

$\Rightarrow xAy$

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

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



$\succ \dots$

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

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

$\succ \dots$

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

Production applied

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

And so on.....

Example:

Input

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

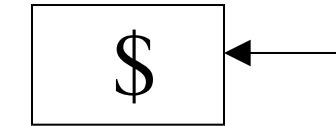


Time 0

$$\lambda, S \rightarrow aSTb$$

$$\lambda, S \rightarrow b$$

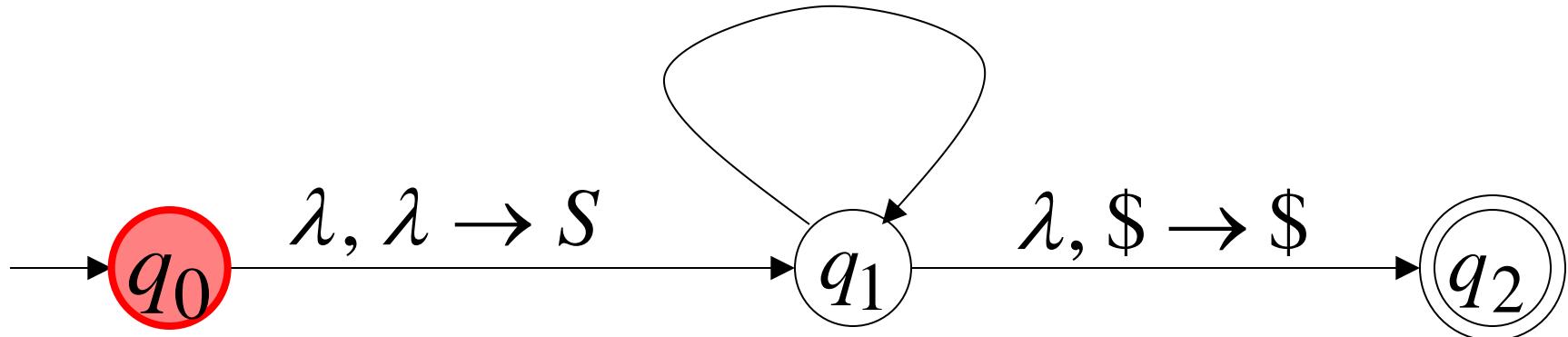
\$



Stack

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

$$\lambda, T \rightarrow \lambda \quad b, b \rightarrow \lambda$$



Derivation: S

Input

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



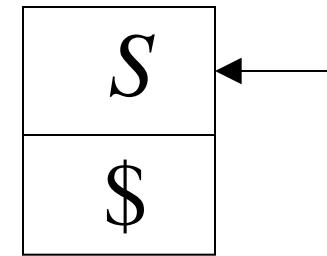
Time 1

$$\lambda, S \rightarrow aSTb$$

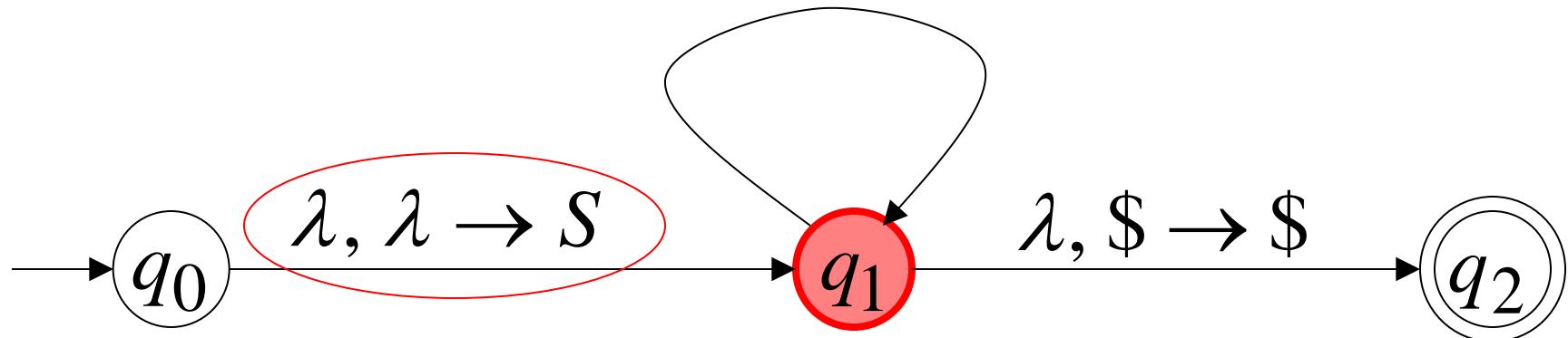
$$\lambda, S \rightarrow b$$

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

$$\lambda, T \rightarrow \lambda \quad b, b \rightarrow \lambda$$



Stack



Derivation: $S \Rightarrow aSTb$

Input

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

Time 2

$$\lambda, S \rightarrow aSTb$$

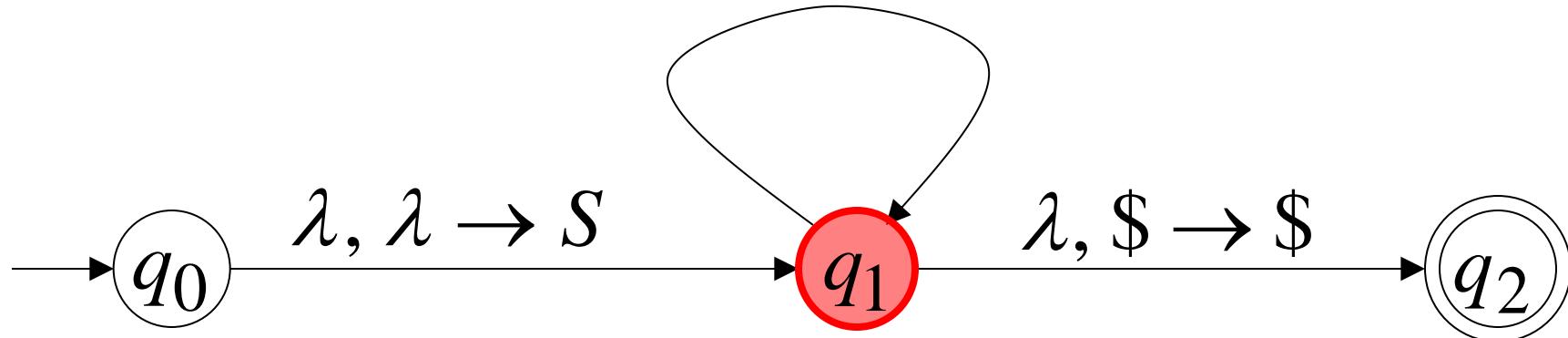
$$\lambda, S \rightarrow b$$

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

$$\lambda, T \rightarrow \lambda \quad b, b \rightarrow \lambda$$

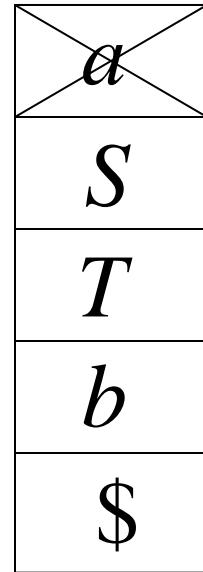
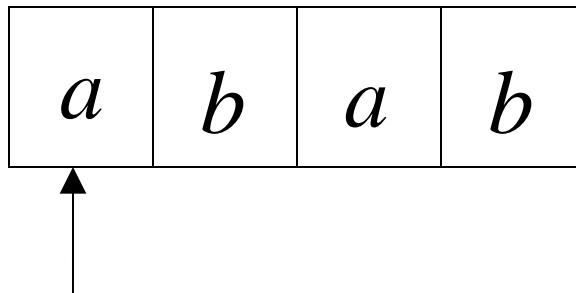
a
S
T
b
\$

Stack



Derivation: $S \Rightarrow aSTb$

Input



Time 3

$$\lambda, S \rightarrow aSTb$$

$$\lambda, S \rightarrow b$$

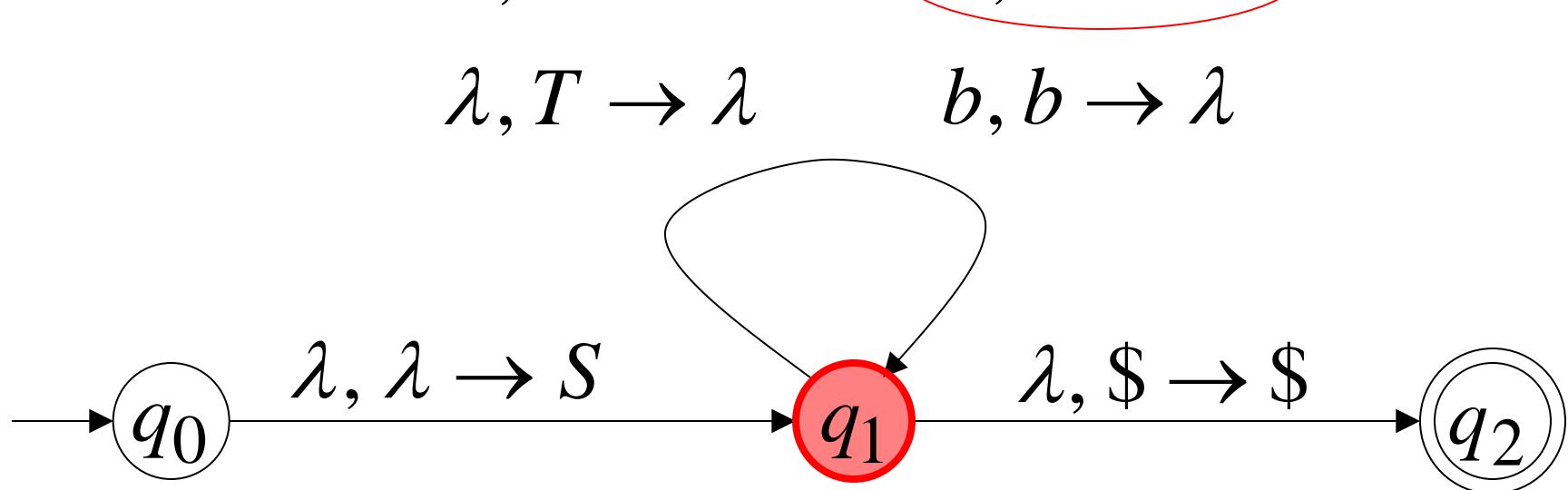
$$\lambda, T \rightarrow Ta$$

$$a, a \rightarrow \lambda$$

$$\lambda, T \rightarrow \lambda$$

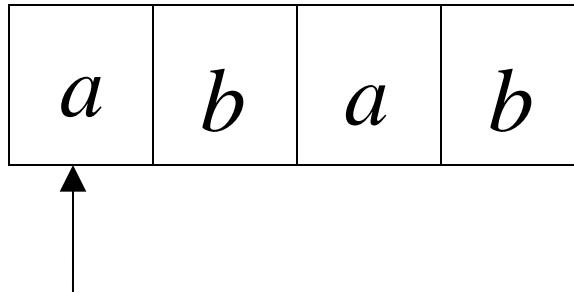
$$b, b \rightarrow \lambda$$

Stack



Derivation: $S \Rightarrow aSTb \Rightarrow abTb$

Input



Time 4

$$\lambda, S \rightarrow aSTb$$

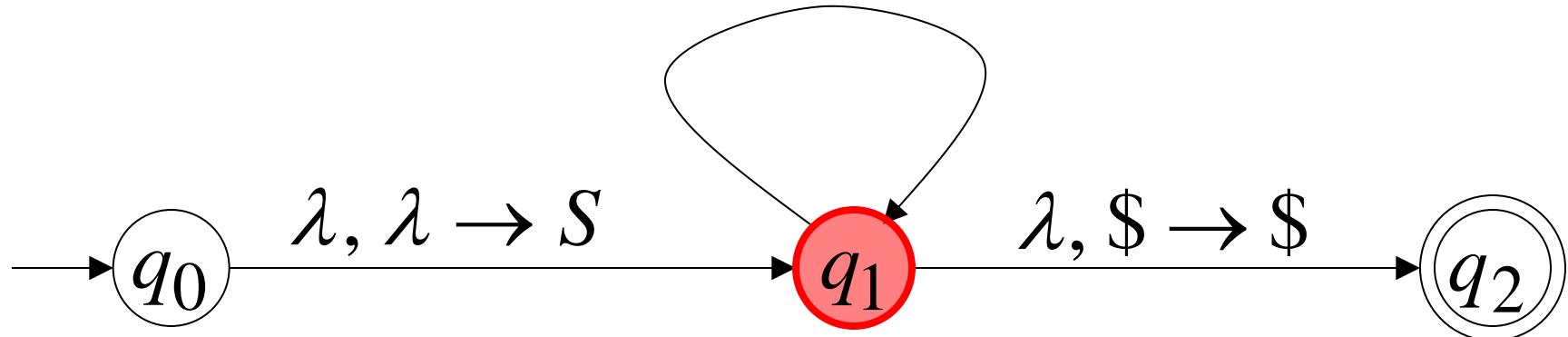
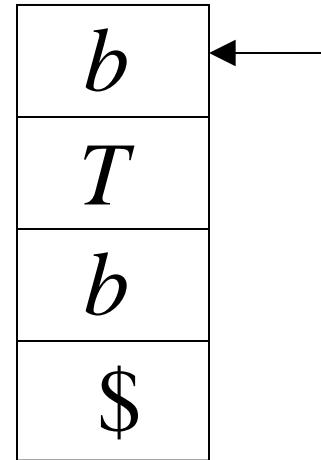
$$\lambda, S \rightarrow b$$

$$\lambda, T \rightarrow Ta$$

$$a, a \rightarrow \lambda$$

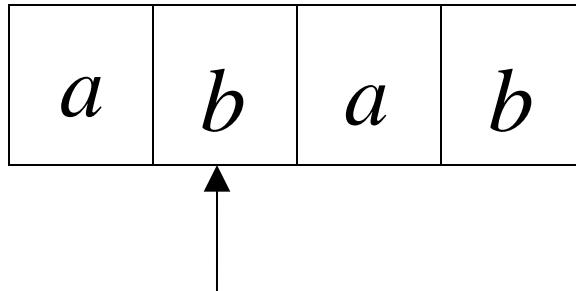
$$\lambda, T \rightarrow \lambda$$

$$b, b \rightarrow \lambda$$



Derivation: $S \Rightarrow aSTb \Rightarrow abTb$

Input



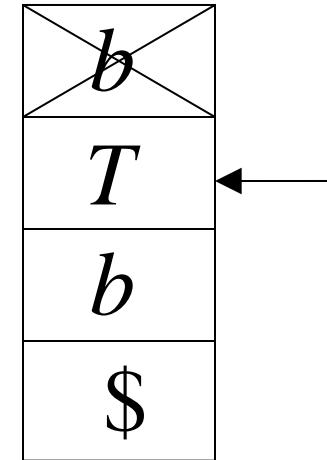
Time 5

$$\lambda, S \rightarrow aSTb$$

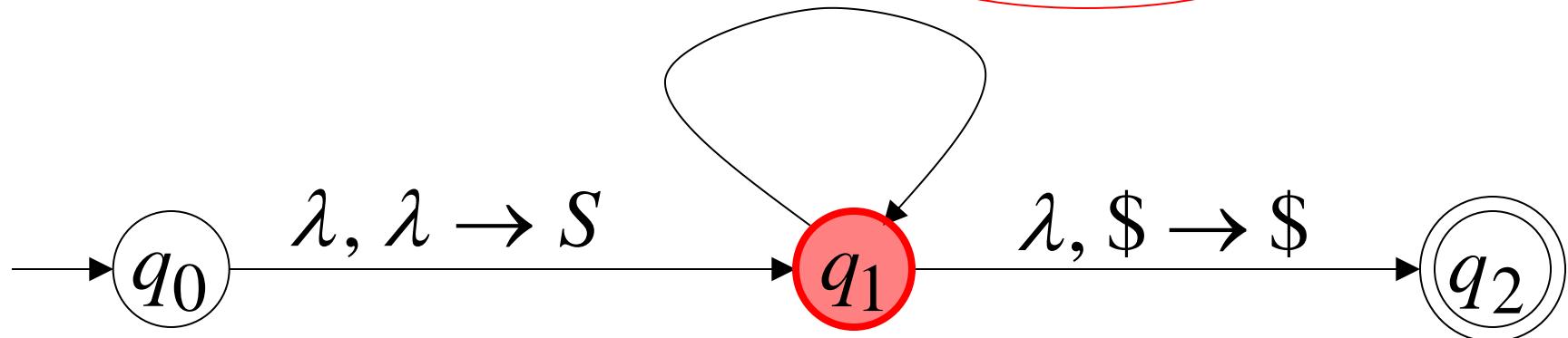
$$\lambda, S \rightarrow b$$

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

$$\lambda, T \rightarrow \lambda \quad b, b \rightarrow \lambda$$

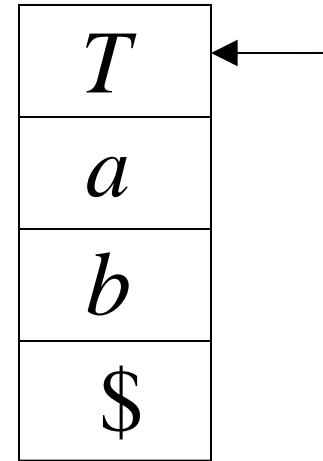
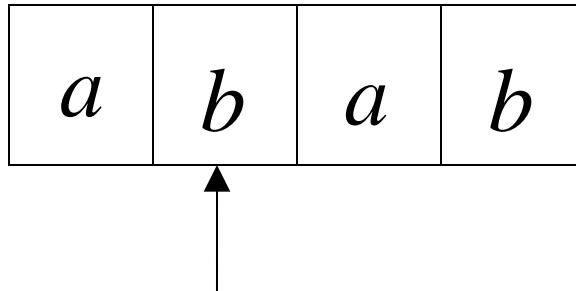


Stack



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

Input



Time 6

$$\lambda, S \rightarrow aSTb$$

$$\lambda, S \rightarrow b$$

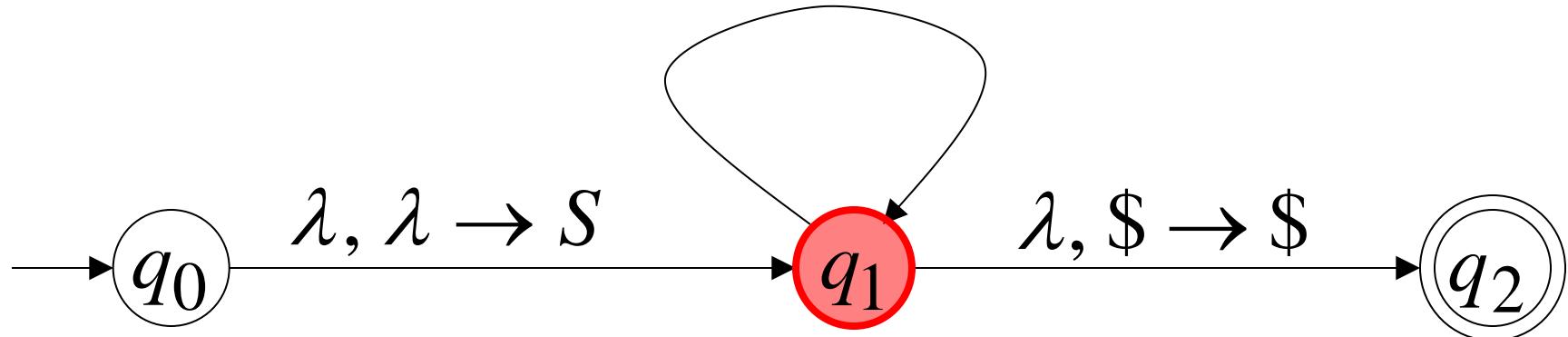
$$\lambda, T \rightarrow Ta$$

$$a, a \rightarrow \lambda$$

$$\lambda, T \rightarrow \lambda$$

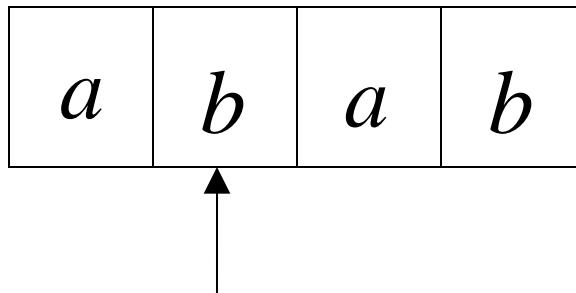
$$b, b \rightarrow \lambda$$

Stack



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

Input



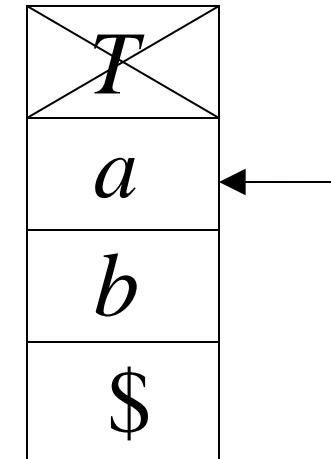
Time 7

$$\lambda, S \rightarrow aSTb$$

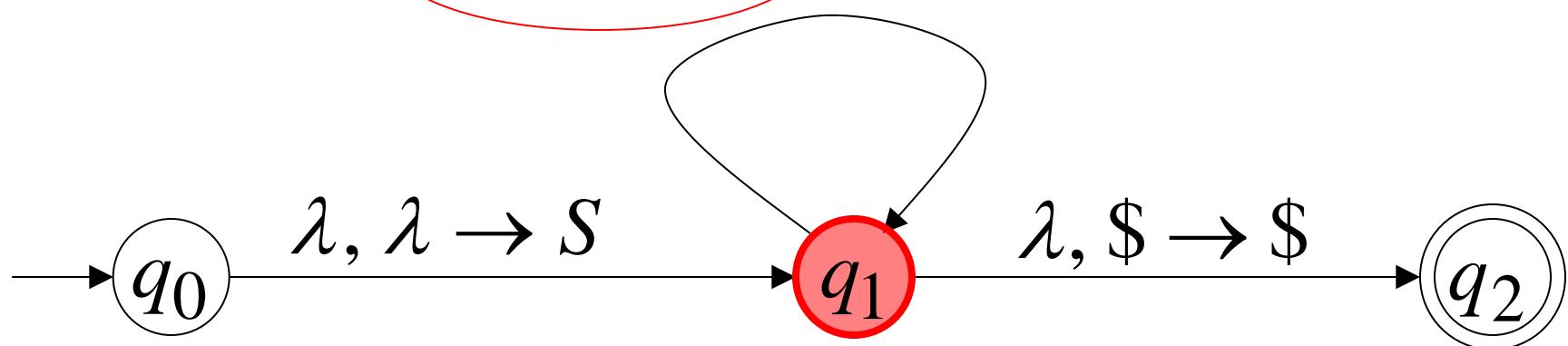
$$\lambda, S \rightarrow b$$

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

$$\lambda, T \rightarrow \lambda \quad b, b \rightarrow \lambda$$

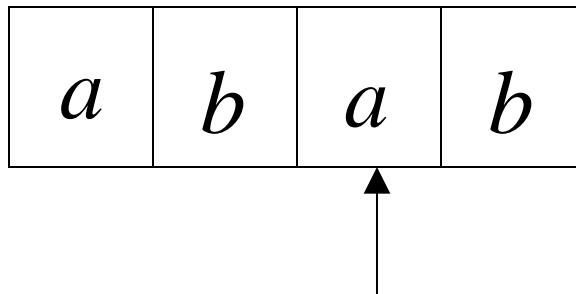


Stack



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

Input



Time 8

$$\lambda, S \rightarrow aSTb$$

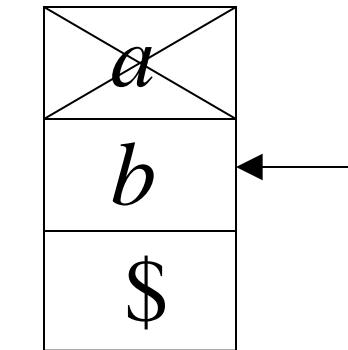
$$\lambda, S \rightarrow b$$

$$\lambda, T \rightarrow Ta$$

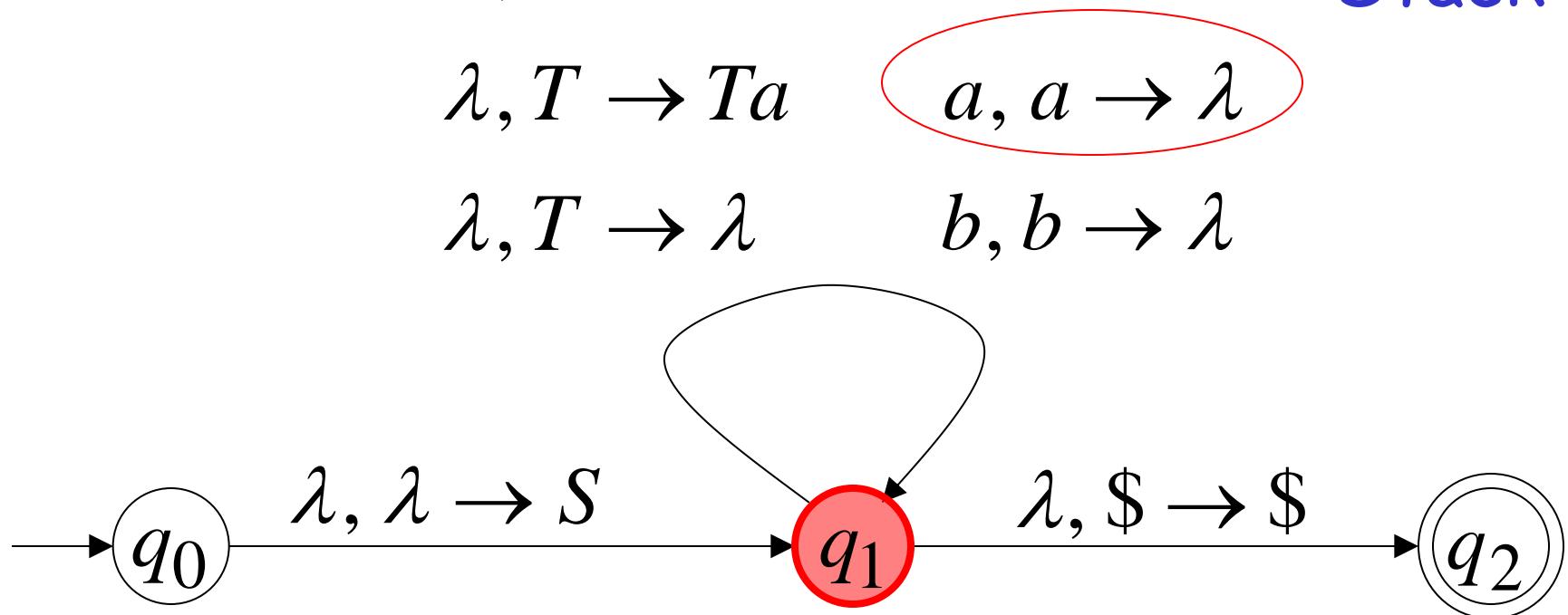
$$a, a \rightarrow \lambda$$

$$\lambda, T \rightarrow \lambda$$

$$b, b \rightarrow \lambda$$



Stack



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

Input

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



Time 9

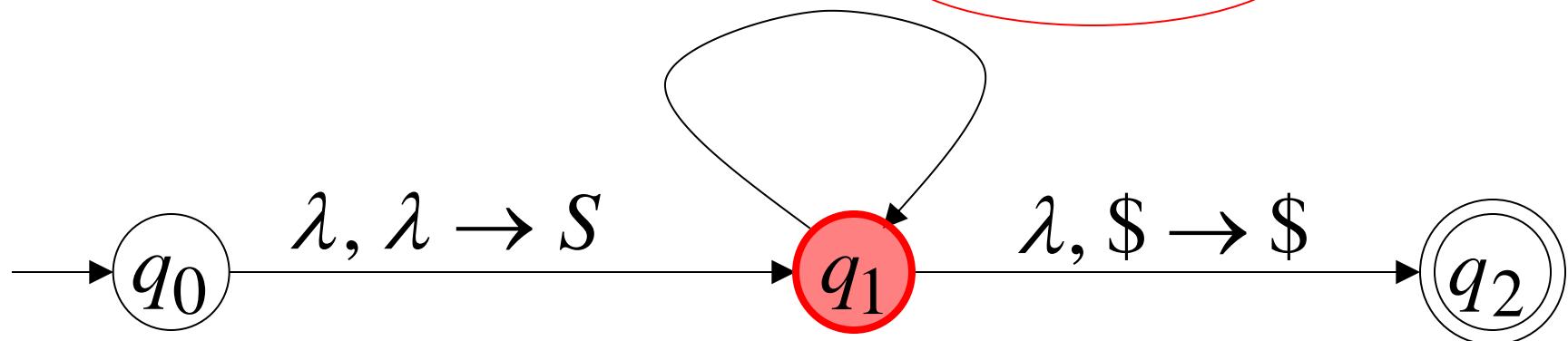
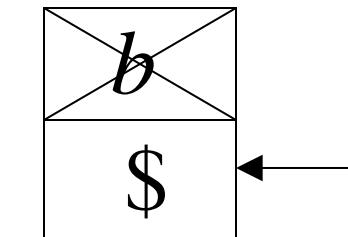
$$\lambda, S \rightarrow aSTb$$

$$\lambda, S \rightarrow b$$

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

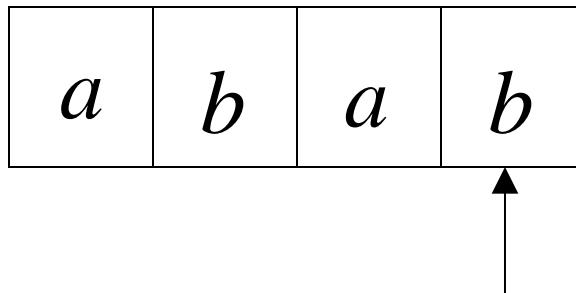
$$\lambda, T \rightarrow \lambda$$

$$b, b \rightarrow \lambda$$



Derivation: $S \Rightarrow aSTb \Rightarrow abTb \Rightarrow ab\cancel{T}ab \Rightarrow abab$

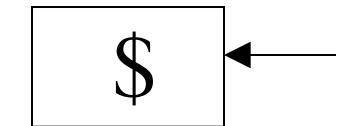
Input



Time 10

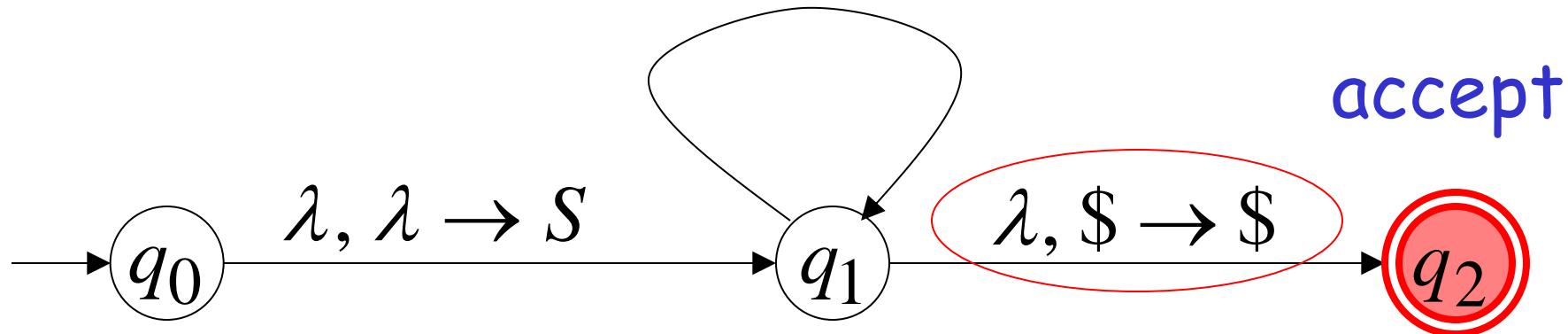
$$\lambda, S \rightarrow aSTb$$

$$\lambda, S \rightarrow b$$



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

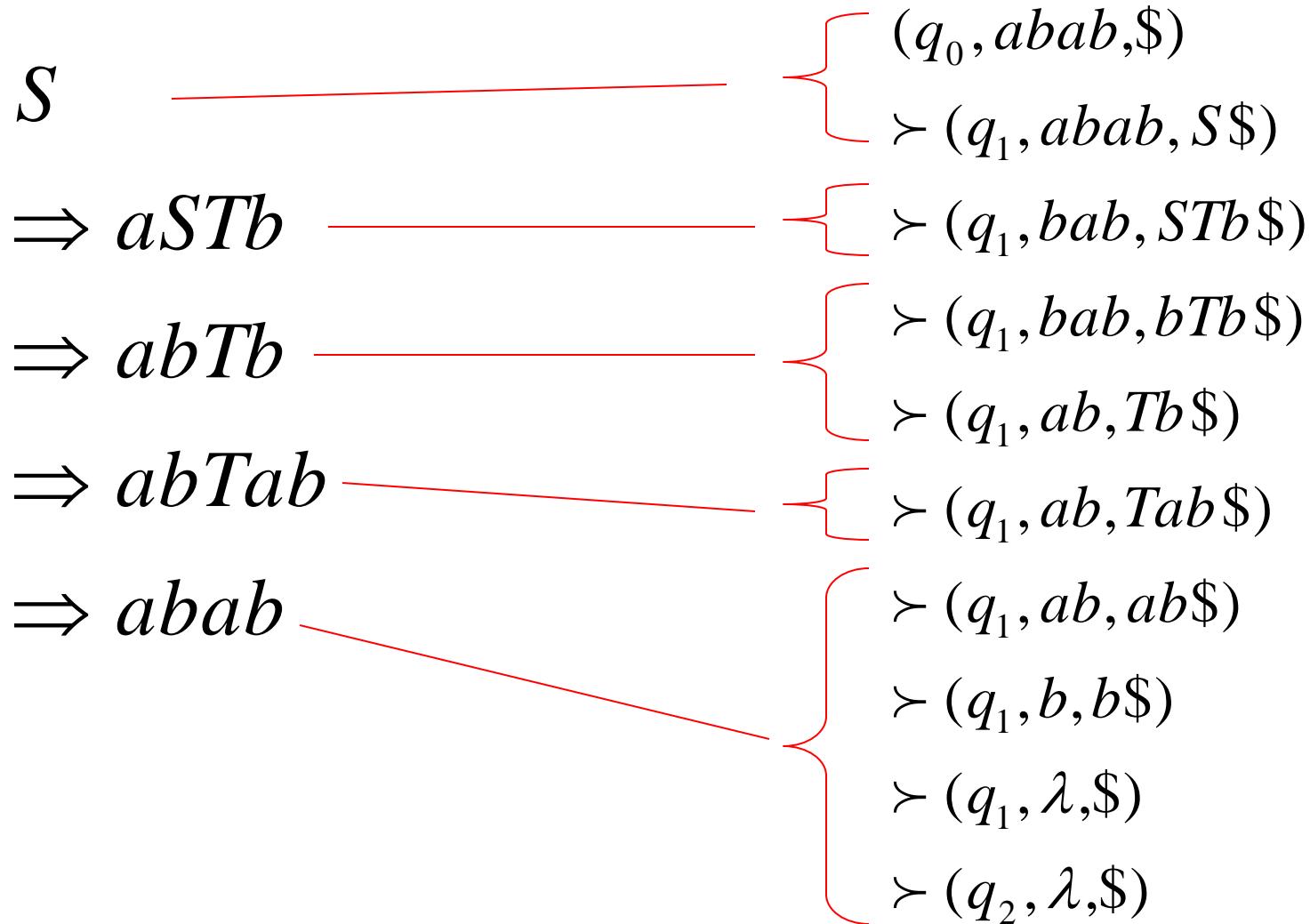
$$\lambda, T \rightarrow \lambda \quad b, b \rightarrow \lambda$$



Grammar

Leftmost Derivation

PDA Computation

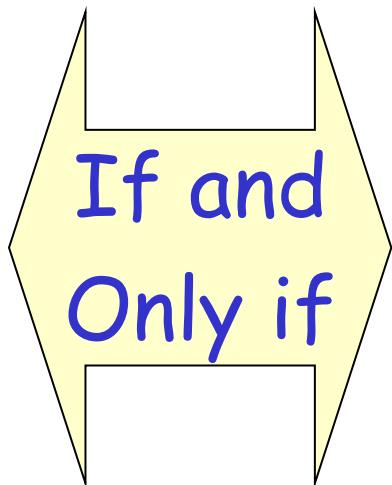


In general, it can be shown that:

Grammar G

generates
string w

$S \xrightarrow{*} w$



PDA M
accepts w

$(q_0, w, \$) \succ (q_2, \lambda, \$)$

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:

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

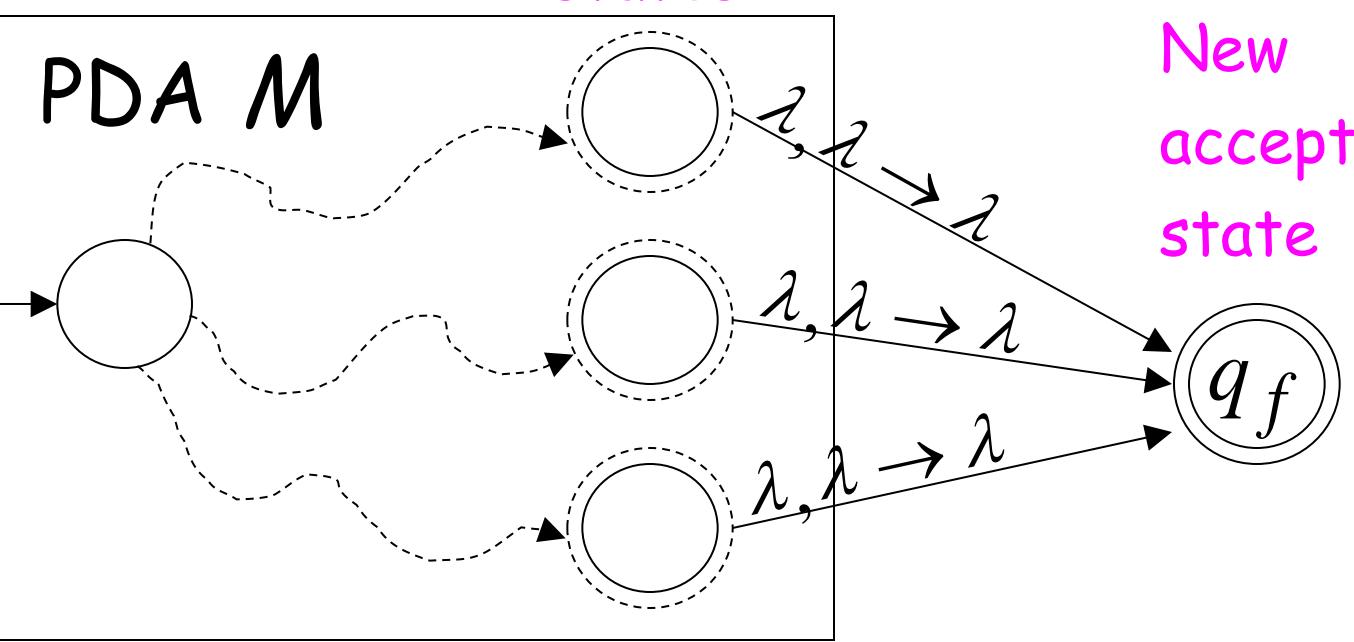
First modify PDA M so that:

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

1. The PDA has a single accept state

PDA M_1

Old
accept
states



2. Use new initial stack symbol

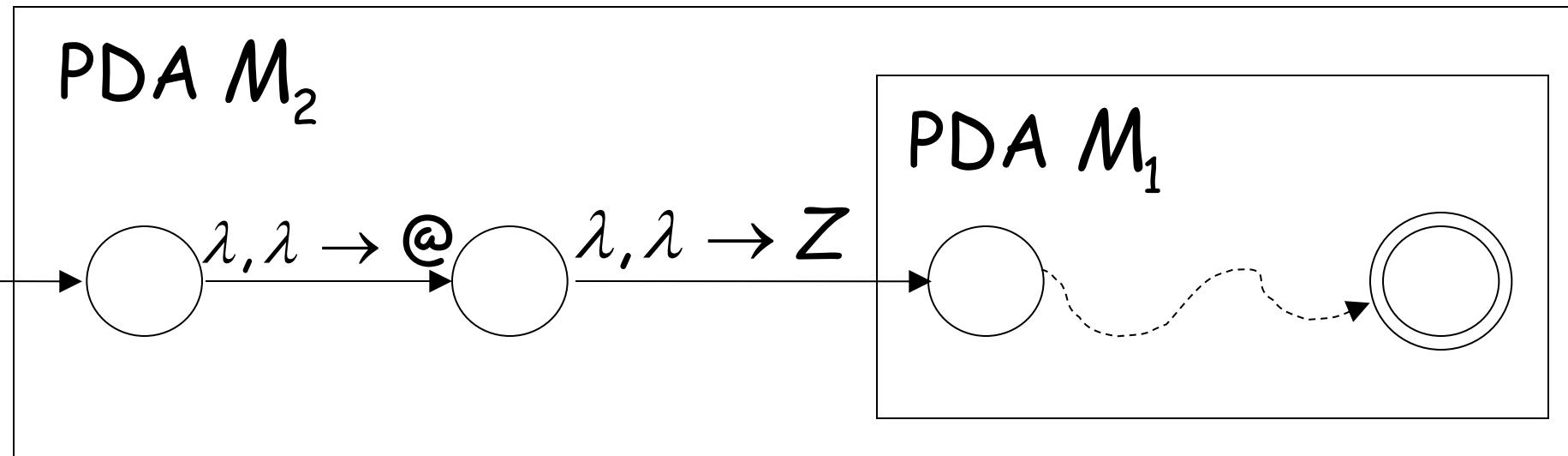
Top of stack

Z
@
#

old initial stack symbol

auxiliary stack symbol

new initial stack symbol



M_1 still thinks that Z is the initial stack

3. On acceptance the stack contains only stack symbol

PDA M_3

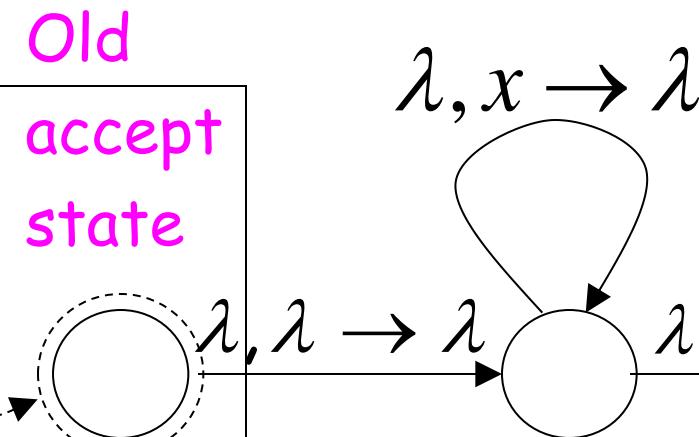
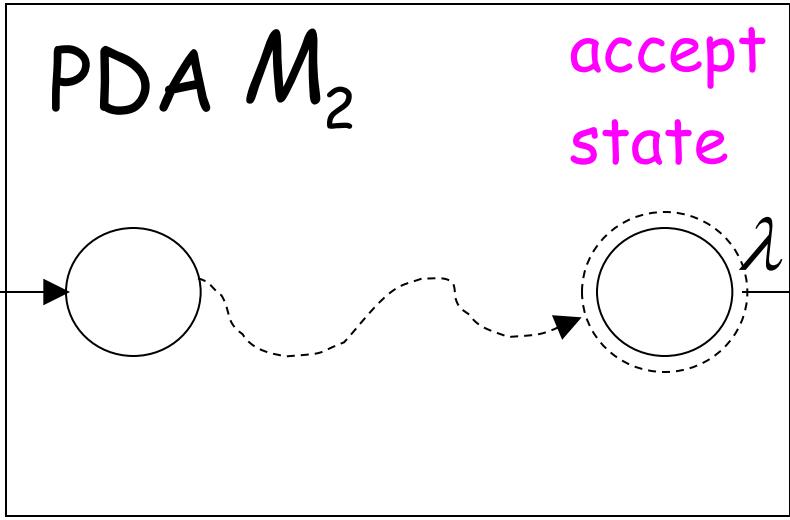
Empty stack

$$\forall x \in \Gamma - \{@, \#\}$$

PDA M_2

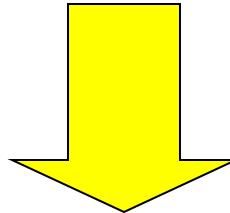
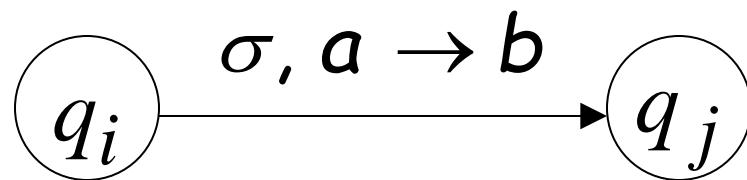
Old
accept
state

New
accept
state

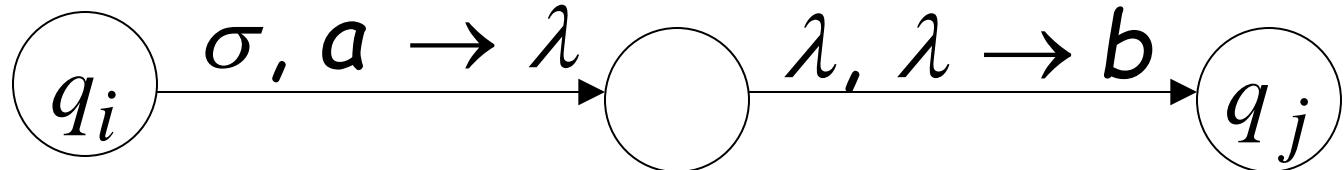


4. Each transition either pushes a symbol
or pops a symbol but not both together

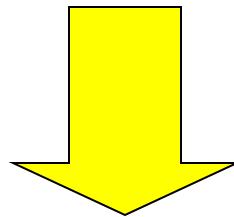
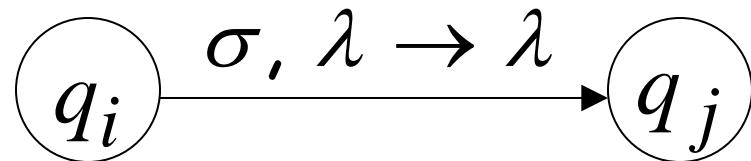
PDA M_3



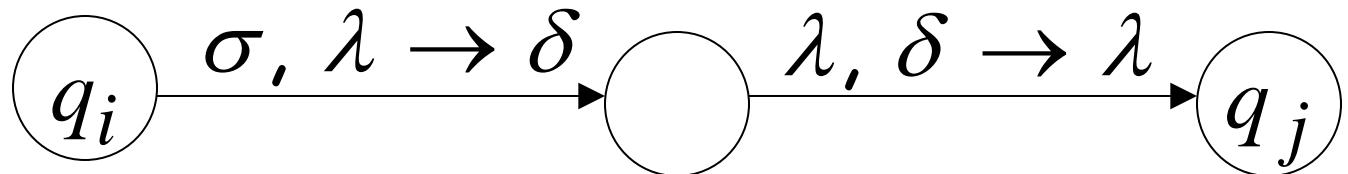
PDA M_4



PDA M_3



PDA M_4

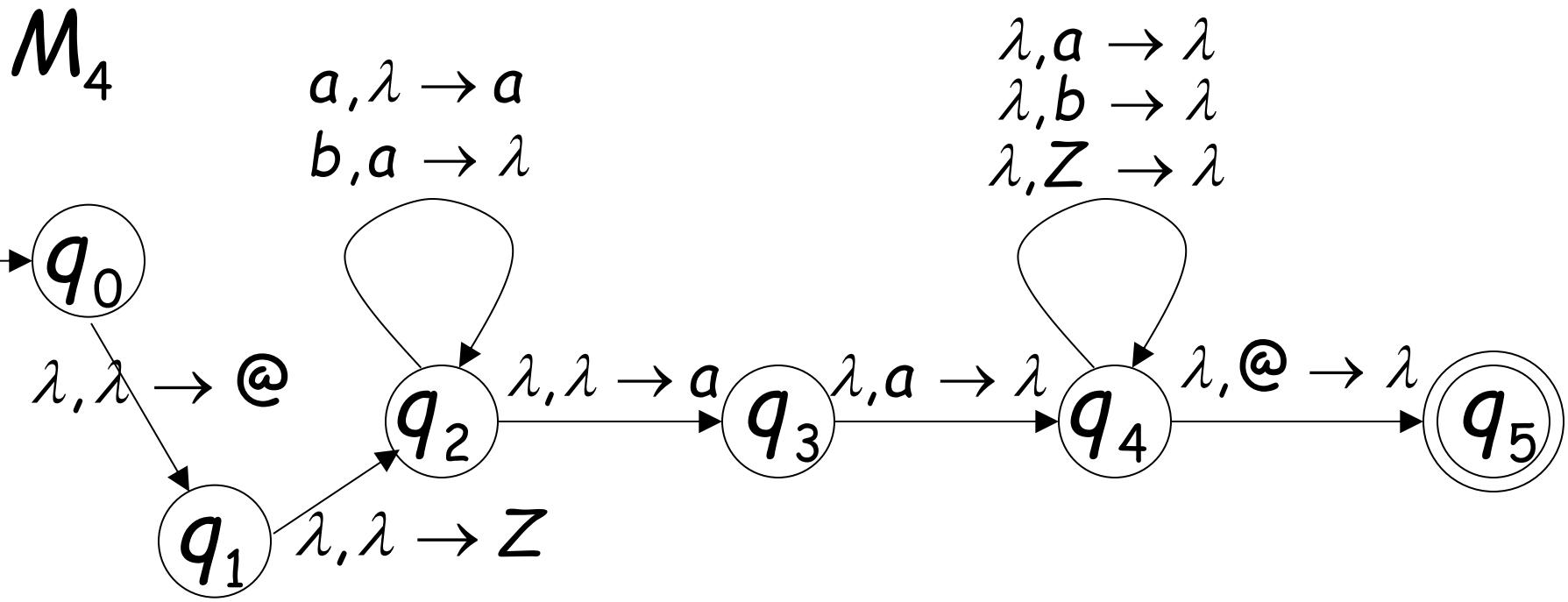
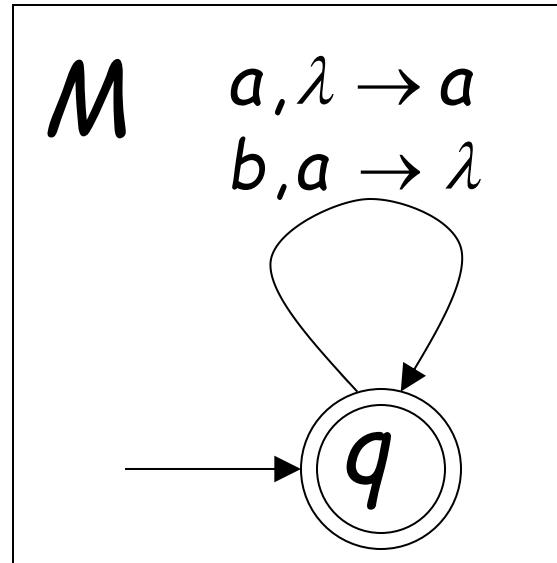


Where δ is a symbol of the stack alphabet

PDA M_4 is the final modified PDA

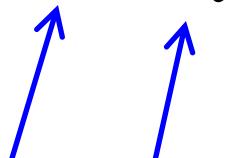
Note that the new initial stack symbol # is never used in any transition

Example:



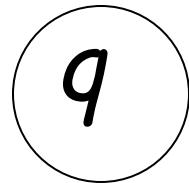
Grammar Construction

Variables: A_{q_i, q_j}
States of PDA



PDA

Kind 1: for each state

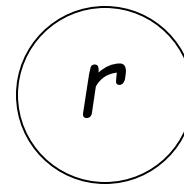
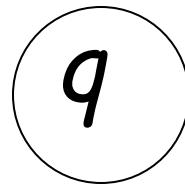
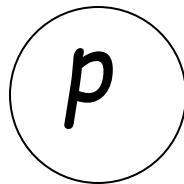


Grammar

$$A_{qq} \rightarrow \lambda$$

PDA

Kind 2: for every three states

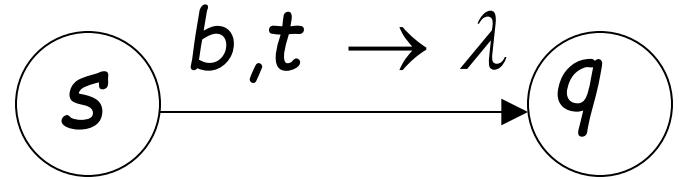
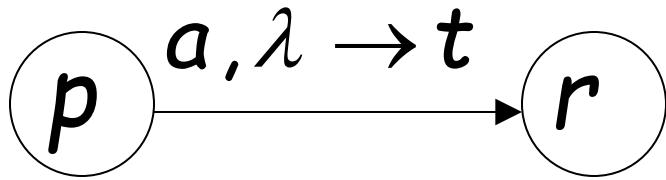


Grammar

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

PDA

Kind 3: for every pair of such transitions

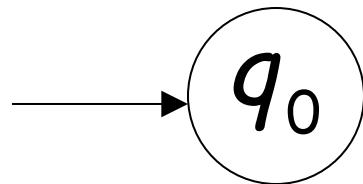


Grammar

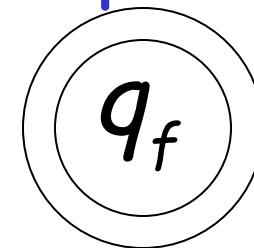
$$A_{pq} \rightarrow a A_{rs} b$$

PDA

Initial state



Accept state



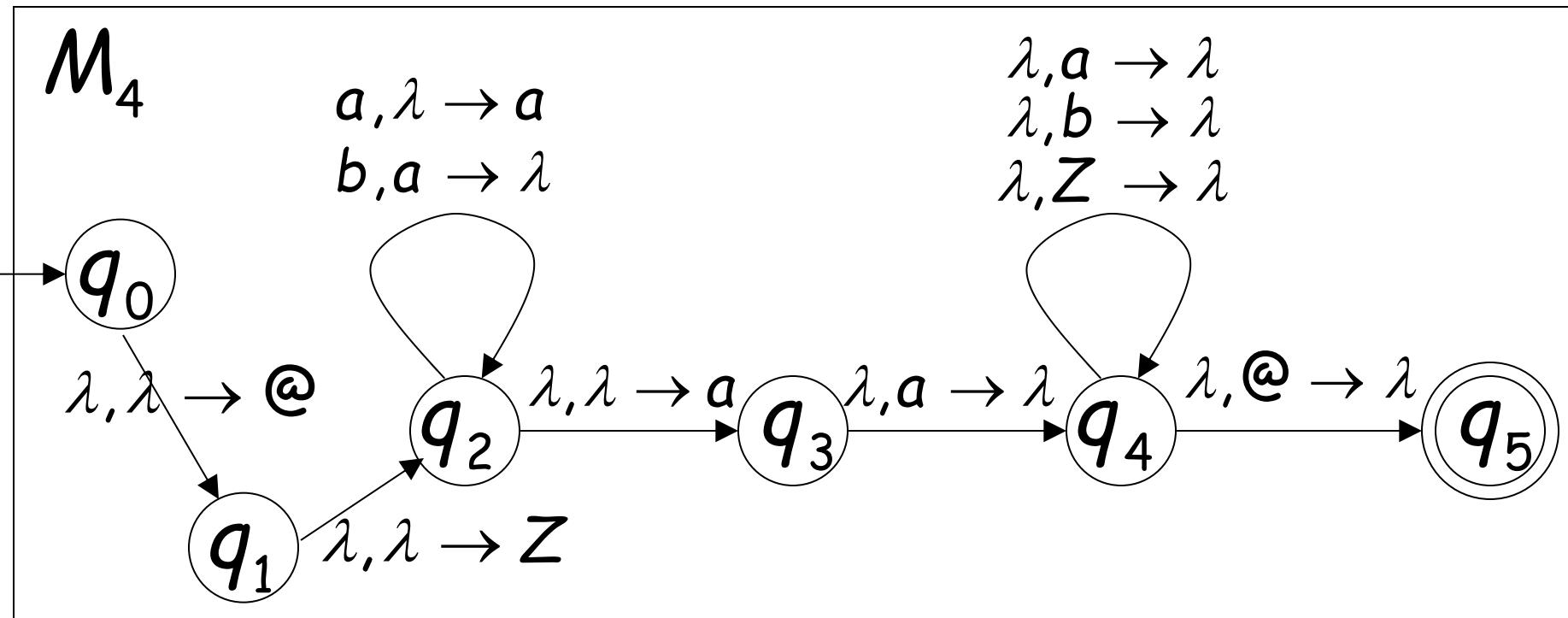
Grammar

Start variable

$A_{q_0 q_f}$

Example:

PDA



Grammar

Kind 1: from single states

$$A_{q_0q_0} \rightarrow \lambda$$

$$A_{q_1q_1} \rightarrow \lambda$$

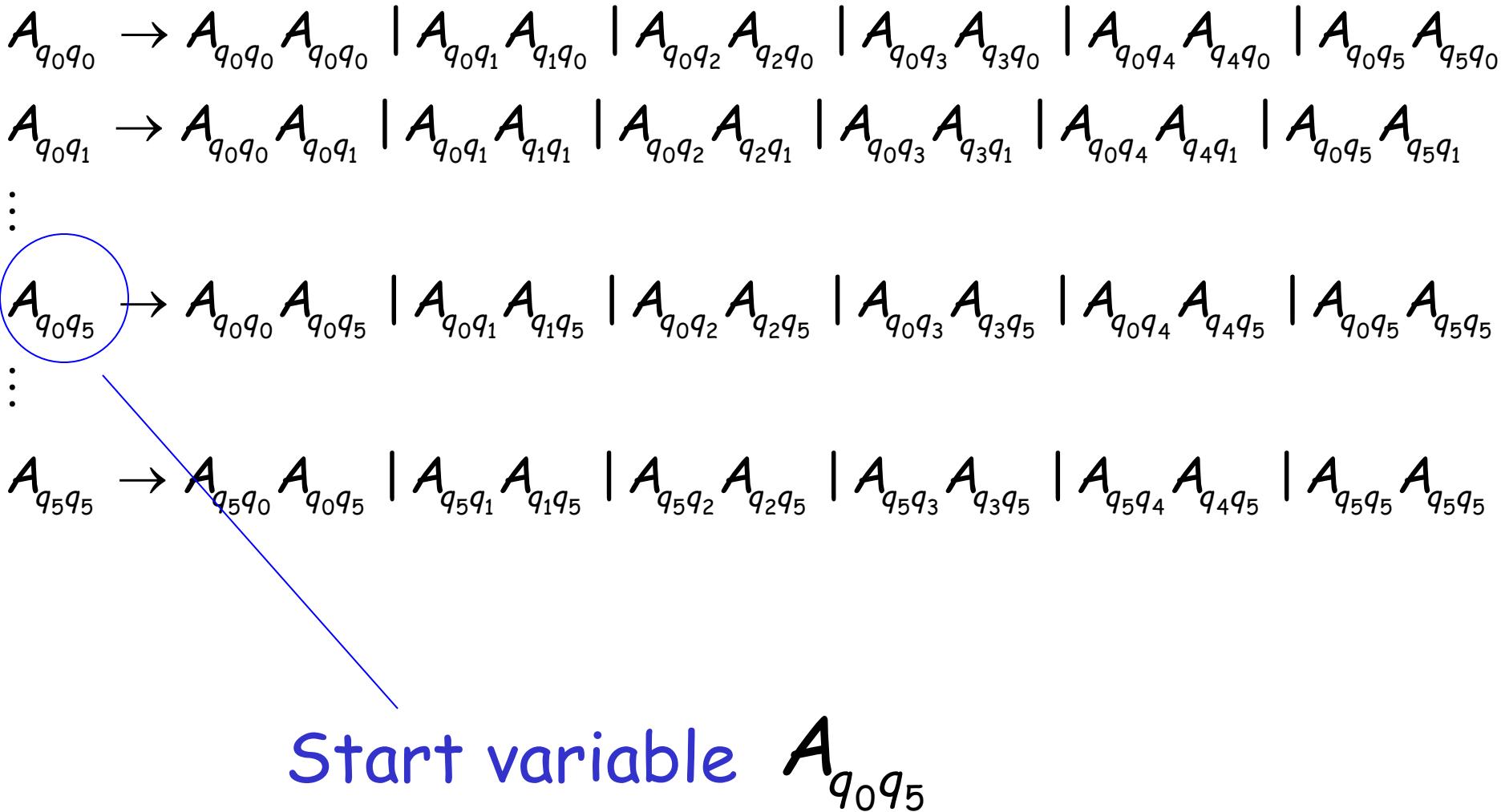
$$A_{q_2q_2} \rightarrow \lambda$$

$$A_{q_3q_3} \rightarrow \lambda$$

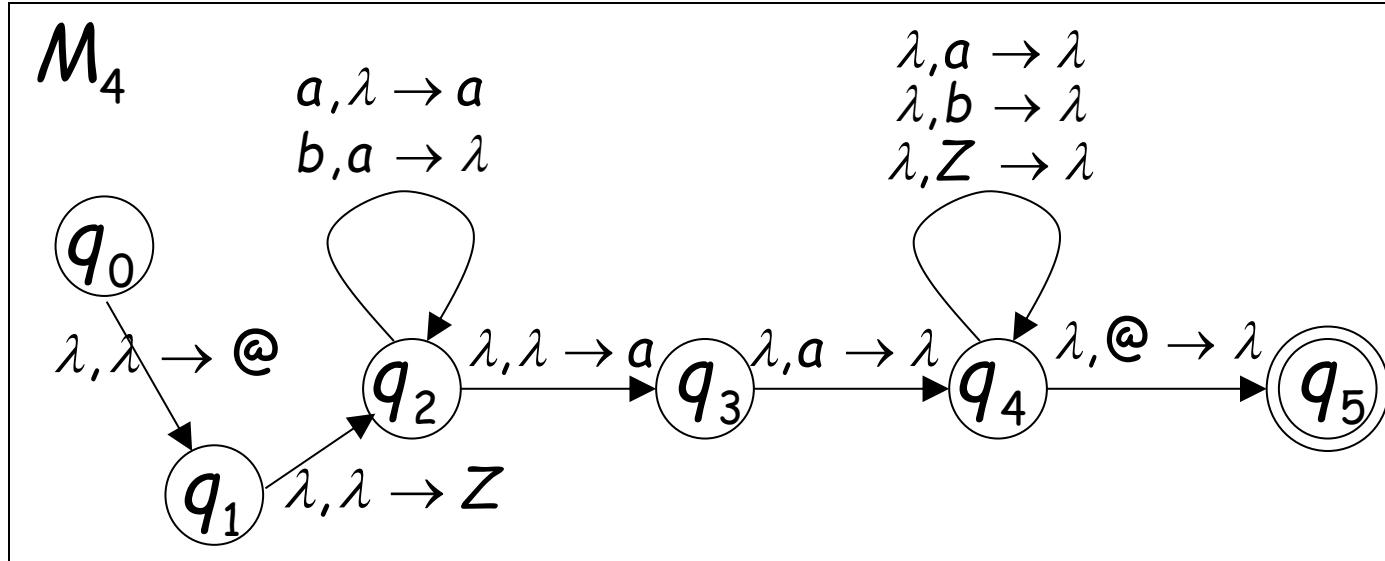
$$A_{q_4q_4} \rightarrow \lambda$$

$$A_{q_5q_5} \rightarrow \lambda$$

Kind 2: from triplets of states



Kind 3: from pairs of transitions



$$A_{q_0q_5} \rightarrow A_{q_1q_4}$$

$$A_{q_2q_4} \rightarrow aA_{q_2q_4}$$

$$A_{q_2q_2} \rightarrow A_{q_3q_2} b$$

$$A_{q_1q_4} \rightarrow A_{q_2q_4}$$

$$A_{q_2q_2} \rightarrow aA_{q_2q_2} b$$

$$A_{q_2q_4} \rightarrow A_{q_3q_3}$$

$$A_{q_2q_4} \rightarrow aA_{q_2q_3}$$

$$A_{q_2q_4} \rightarrow A_{q_3q_4}$$

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 :

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

with input string w

We will actually show that if G has derivation:

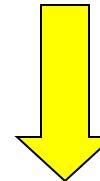
$$A_{pq} \xrightarrow{*} W$$

Then there is a computation in M :

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

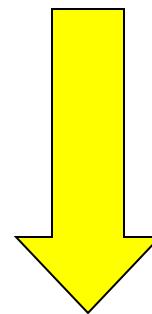
Therefore:

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



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

Since there is no transition
with the # symbol



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

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:

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

Proof Intuition:

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

Type 2

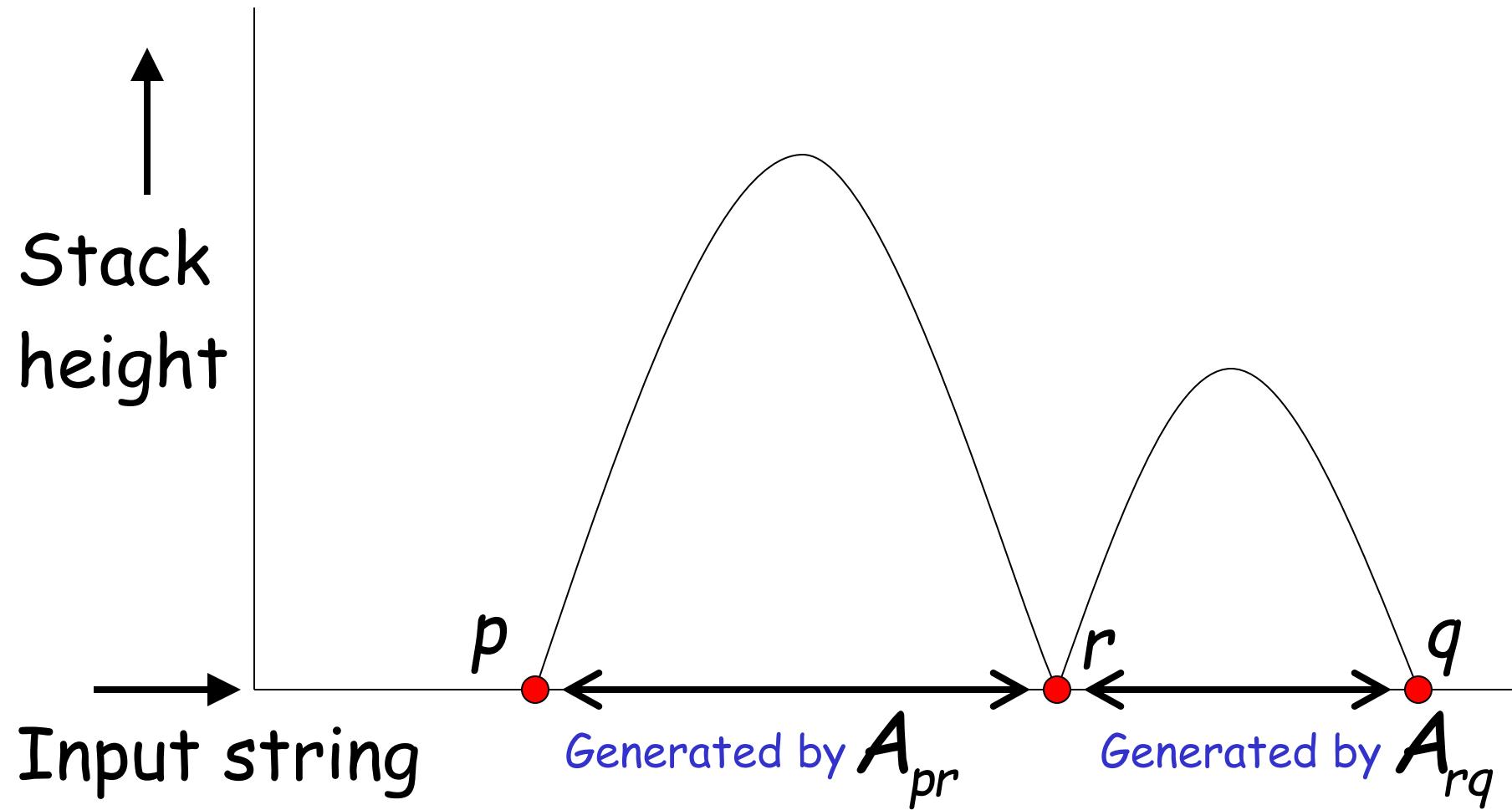
Case 1: $A_{pq} \Rightarrow A_{pr} A_{rq} \Rightarrow \cdots \Rightarrow W$

Type 3

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

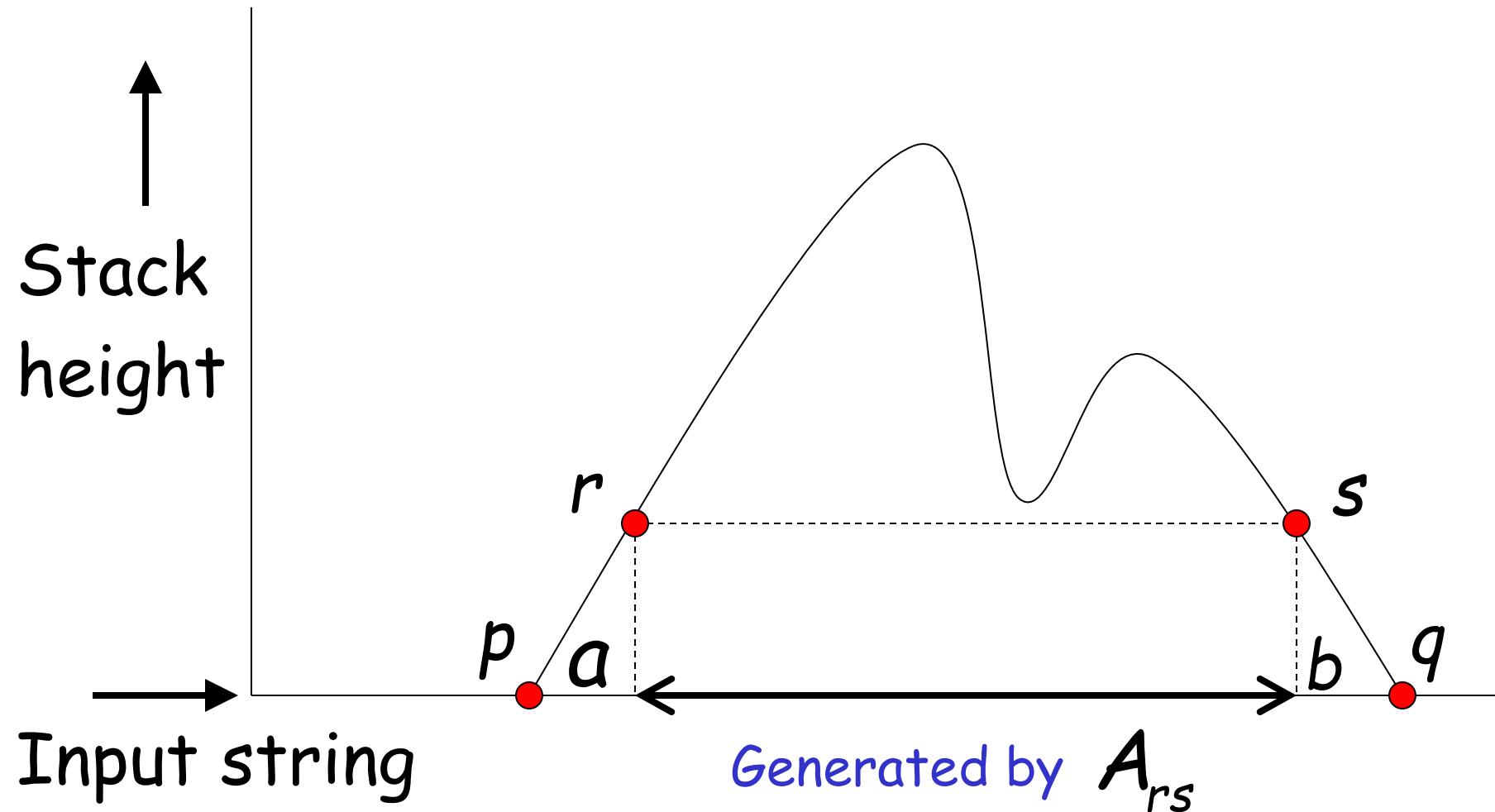
Type 2

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:

$$A_{pp} \rightarrow \lambda$$

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

This computation of PDA trivially exists:

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

Induction Hypothesis:

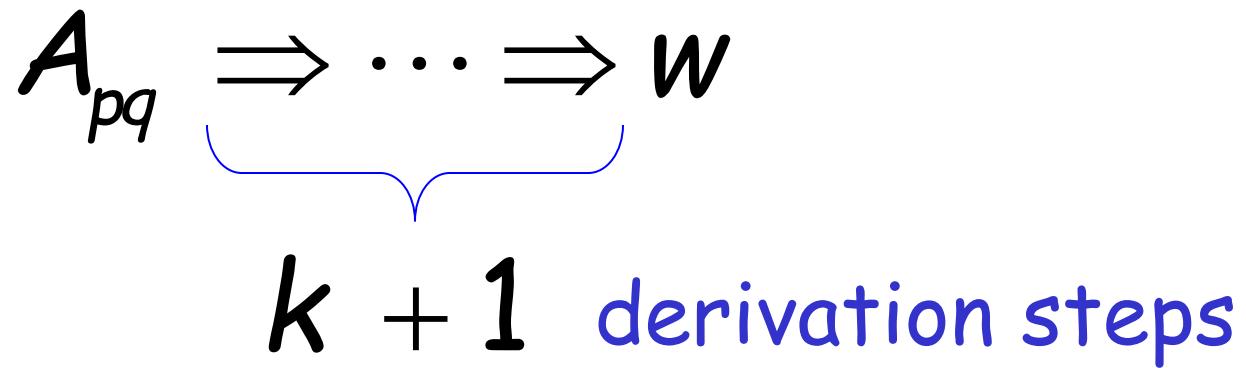
$$A_{pq} \Rightarrow \cdots \Rightarrow W$$

k derivation steps

suppose it holds:

$$(p, w, \lambda) \stackrel{*}{\succ} (q, \lambda, \lambda)$$

Induction Step:



We have to show:

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

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

$k + 1$ 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 $w = yz$

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

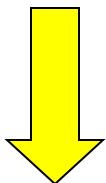
At most k steps

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

At most k steps

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

At most k steps

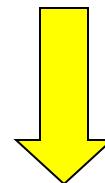


From induction
hypothesis, in PDA:

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

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

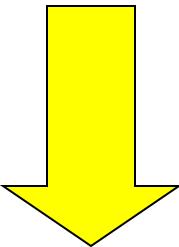
At most k steps



From induction
hypothesis, in PDA:

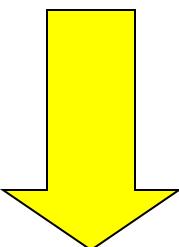
$$(r, z, \lambda) \xrightarrow{*} (q, \lambda, \lambda)$$

$$(p, y, \lambda) \stackrel{*}{\succ} (r, \lambda, \lambda) \quad (r, z, \lambda) \stackrel{*}{\succ} (q, \lambda, \lambda)$$



$$(p, yz, \lambda) \stackrel{*}{\succ} (r, z, \lambda) \stackrel{*}{\succ} (q, \lambda, \lambda)$$

since $w = yz$



$$(p, w, \lambda) \stackrel{*}{\succ} (q, \lambda, \lambda)$$

Type 3

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

$k + 1$ steps

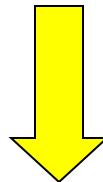
We can write $w = ayb$

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

At most k steps

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

At most k steps

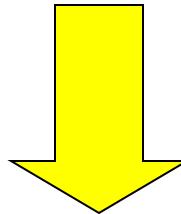


From induction hypothesis,
the PDA has computation:

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

Type 3

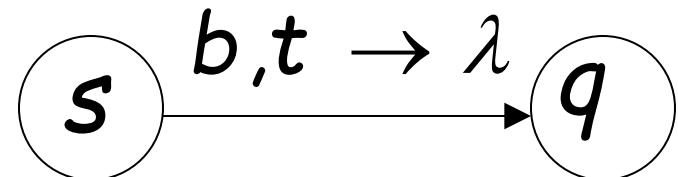
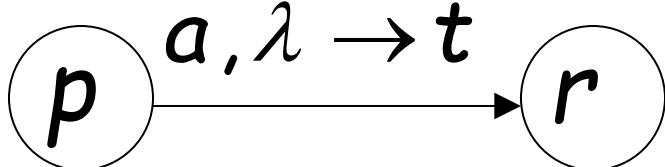
$$A_{pq} \Rightarrow a A_{rs} b \Rightarrow \dots \Rightarrow w$$

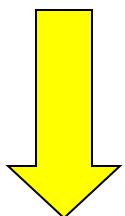
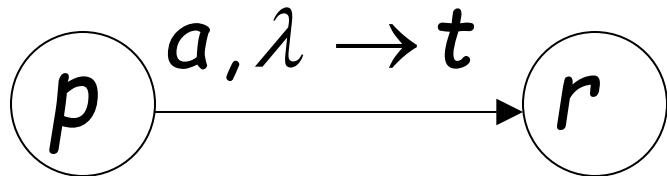
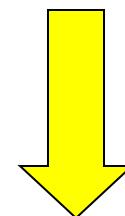
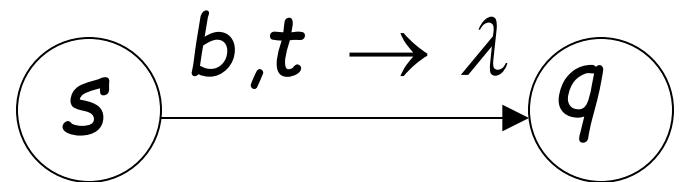


Grammar contains production

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

And PDA Contains transitions




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

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

We know

$$(r, y, \lambda) \overset{*}{\succ} (s, \lambda, \lambda) \quad \longrightarrow \quad (r, yb, t) \overset{*}{\succ} (s, b, t)$$

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

We also know

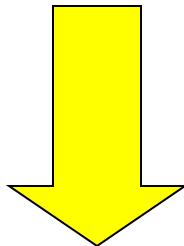
$$(s, b, t) \succ (q, \lambda, \lambda)$$

Therefore:

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

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

since $w = ayb$



$$(p, w, \lambda) \stackrel{*}{\succ} (q, \lambda, \lambda)$$

END OF PROOF

So far we have shown:

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

With a similar proof we can show

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

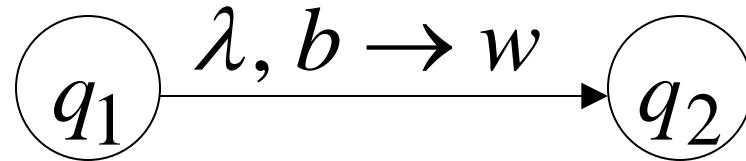
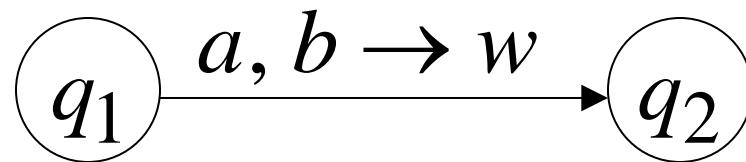
Therefore: $L(G) = L(M)$

DPDA

Deterministic PDA

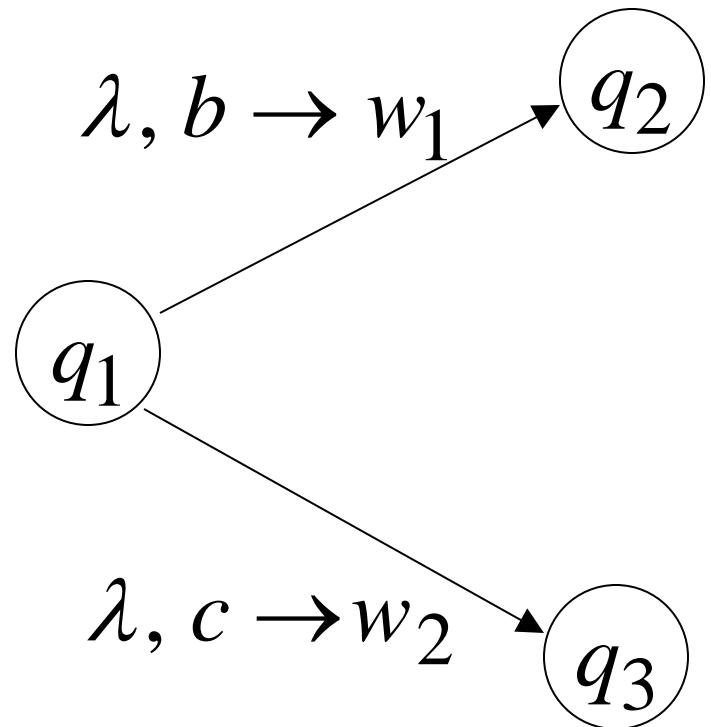
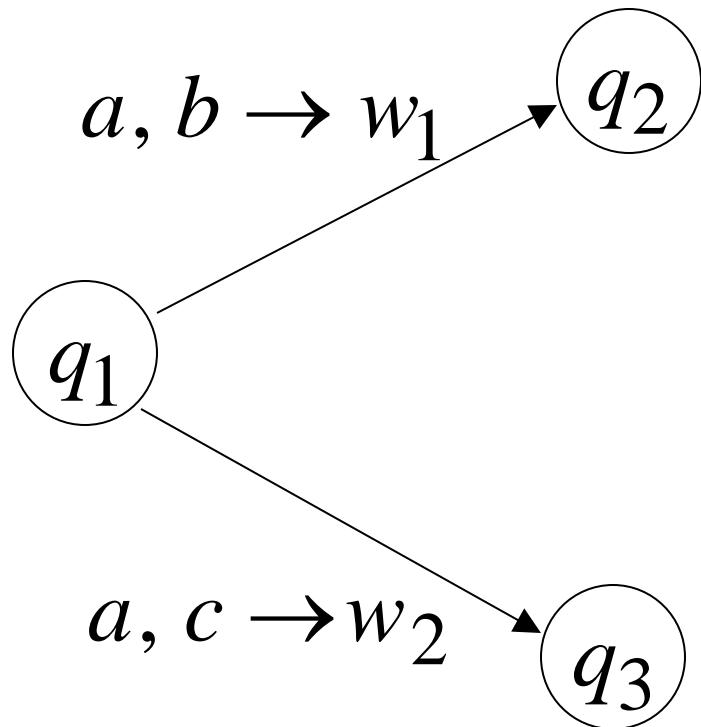
Deterministic PDA: DPDA

Allowed transitions:



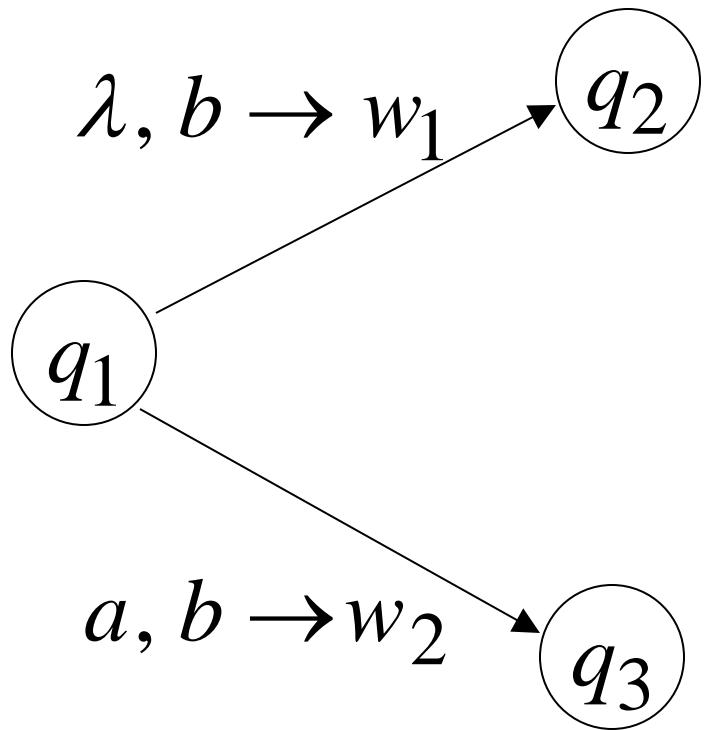
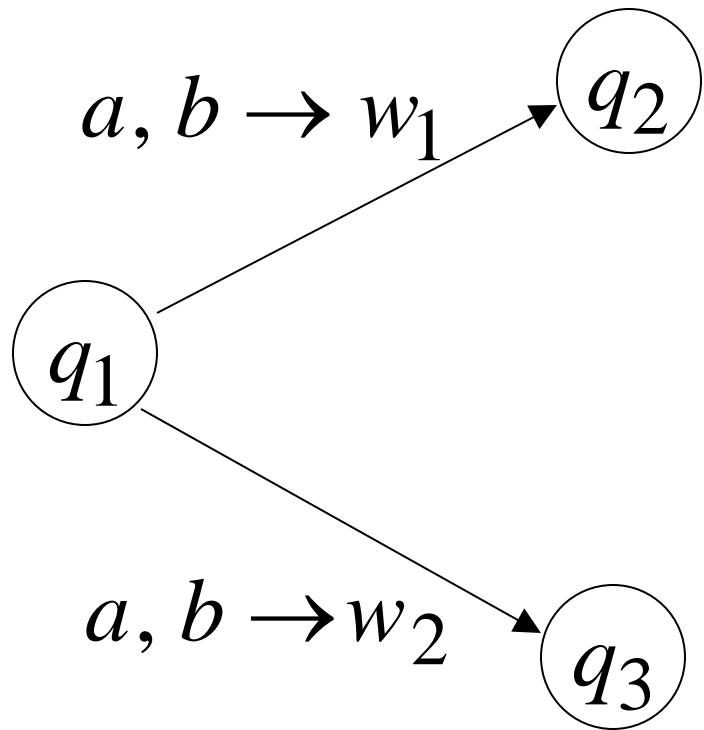
(deterministic choices)

Allowed transitions:



(deterministic choices)

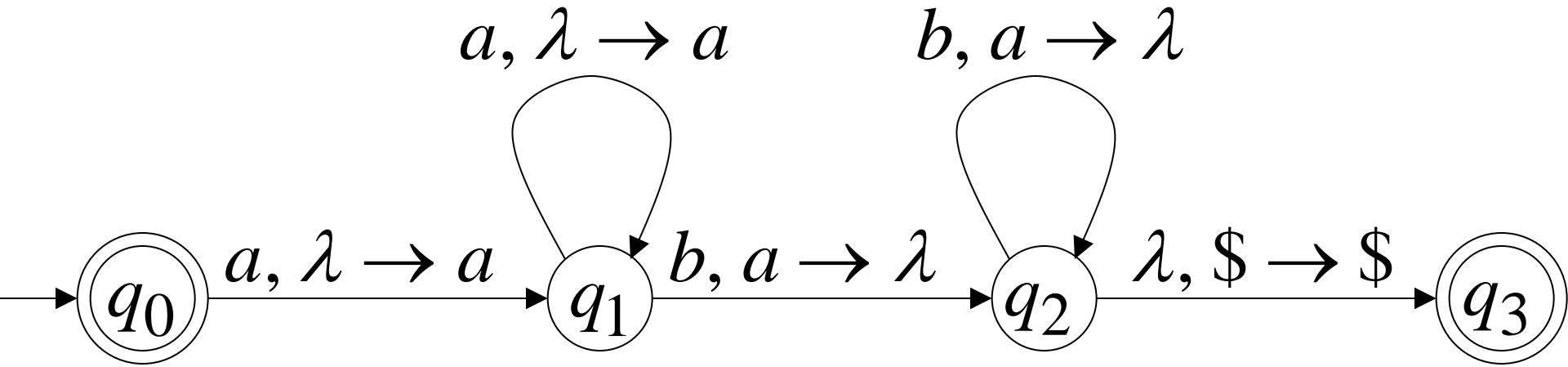
Not allowed:



(non deterministic choices)

DPDA example

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



Definition:

A language L is **deterministic context-free** if there exists some DPDA that accepts it

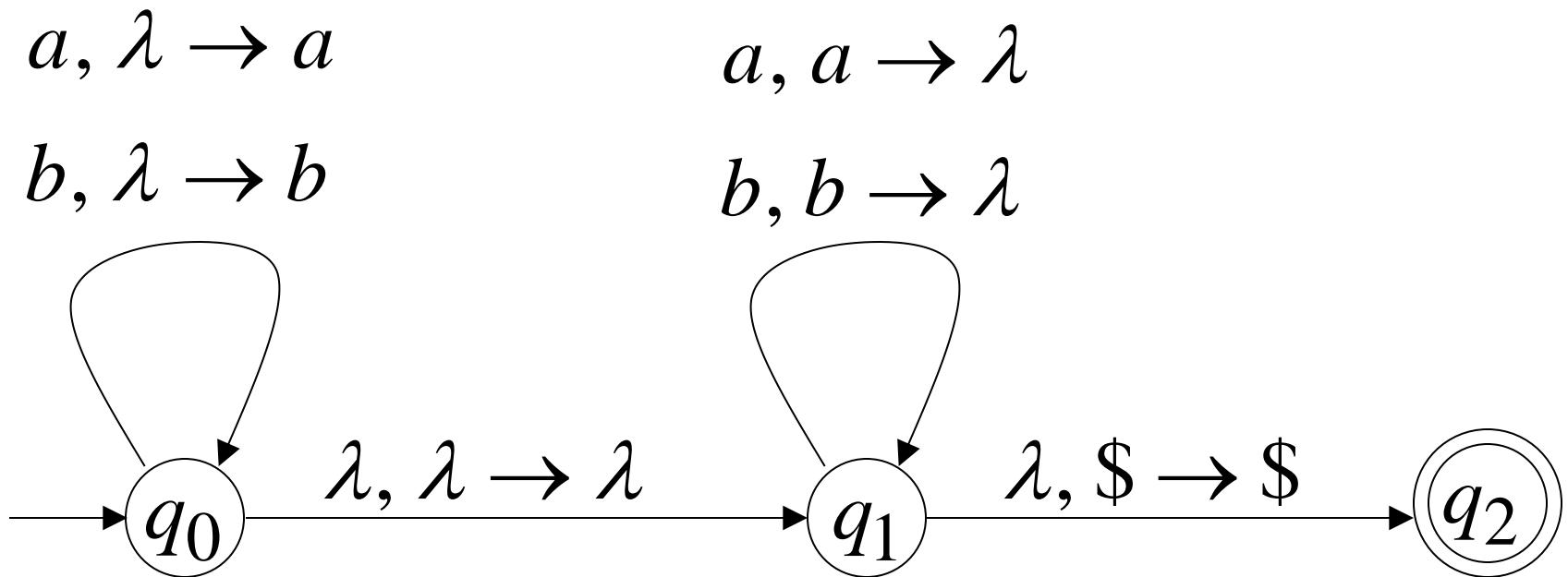
Example:

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

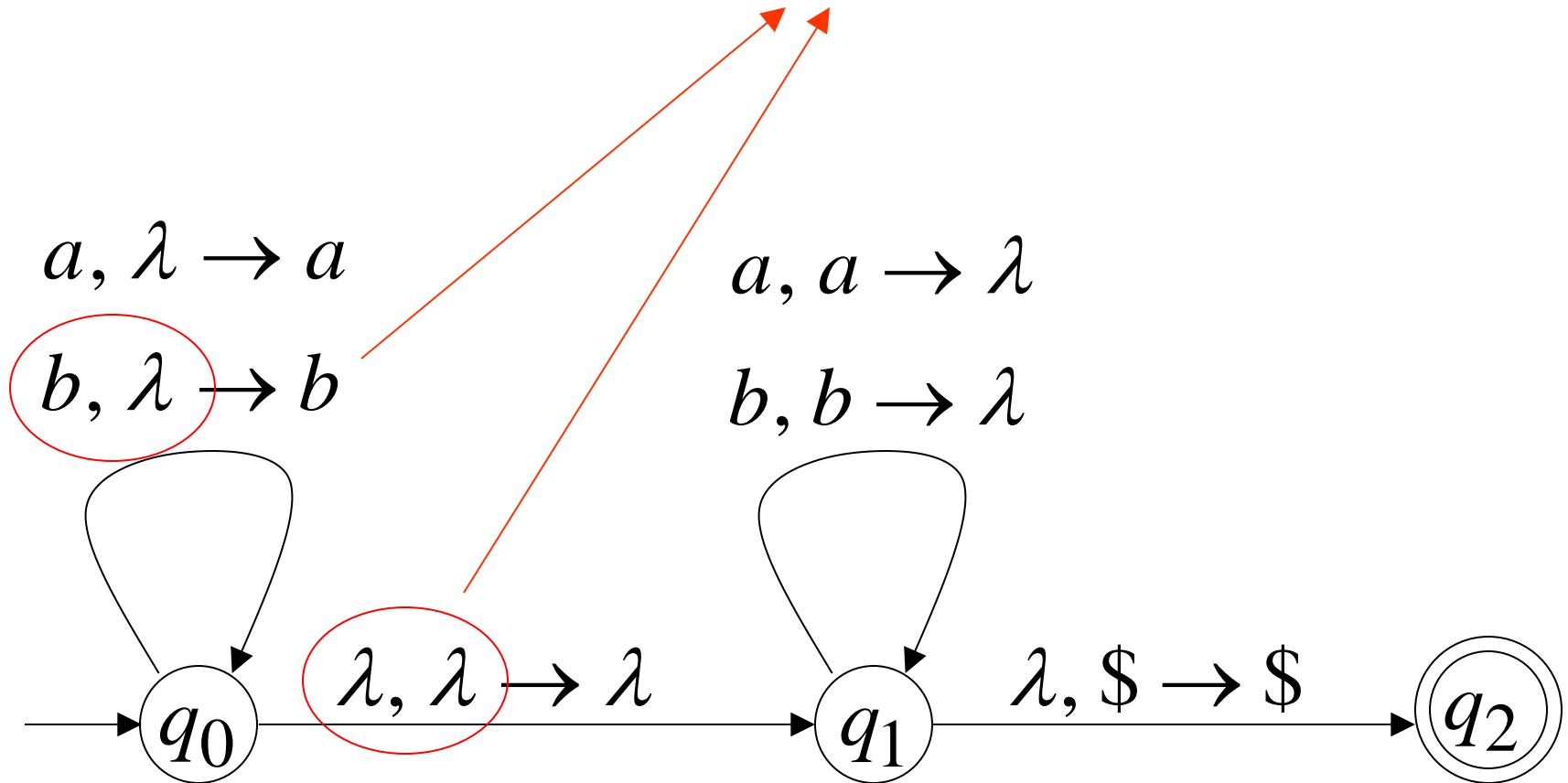
is **deterministic context-free**

Example of Non-DPDA (PDA)

$$L(M) = \{vv^R : v \in \{a,b\}^*\}$$



Not allowed in DPDA

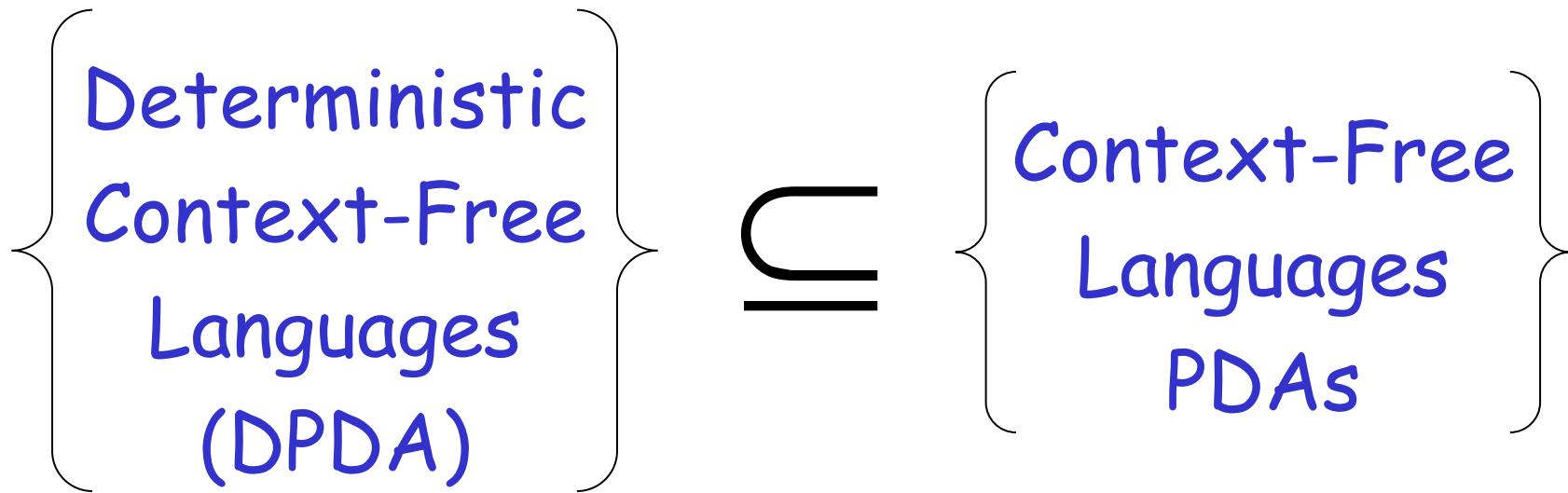


PDAs

Have More Power than

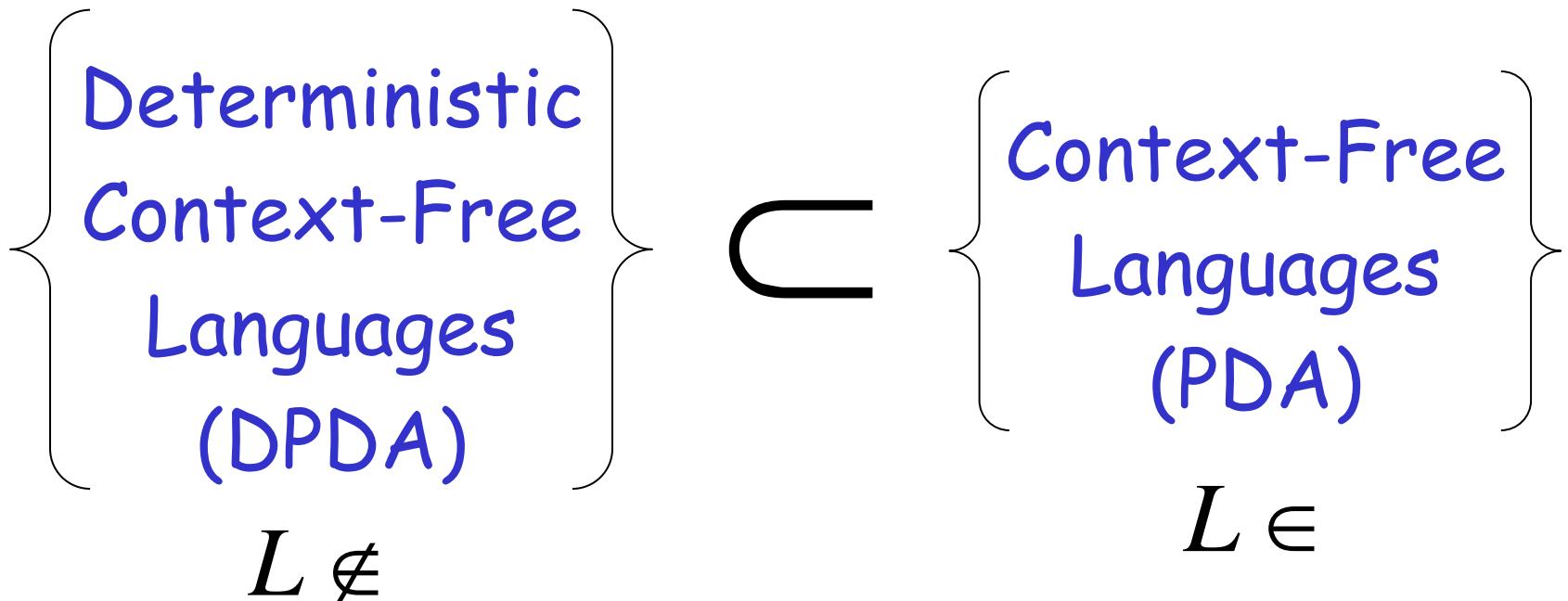
DPDAs

It holds that:



Since every DPDA is also a PDA

We will actually show:



We will show that there exists
a context-free language L which is not
accepted by any DPDA

The language is:

$$L = \{a^n b^n\} \cup \{a^n b^{2n}\} \quad n \geq 0$$

We will show:

- L is context-free
- L is **not** deterministic context-free

$$L = \{a^n b^n\} \cup \{a^n b^{2n}\}$$

Language L is context-free

Context-free grammar for L :

$$S \rightarrow S_1 \mid S_2 \quad \{a^n b^n\} \cup \{a^n b^{2n}\}$$

$$S_1 \rightarrow aS_1b \mid \lambda \quad \{a^n b^n\}$$

$$S_2 \rightarrow aS_2bb \mid \lambda \quad \{a^n b^{2n}\}$$

Theorem:

The language $L = \{a^n b^n\} \cup \{a^n b^{2n}\}$

is **not** deterministic context-free

(there is **no** DPDA that accepts L)

Proof: Assume for contradiction that

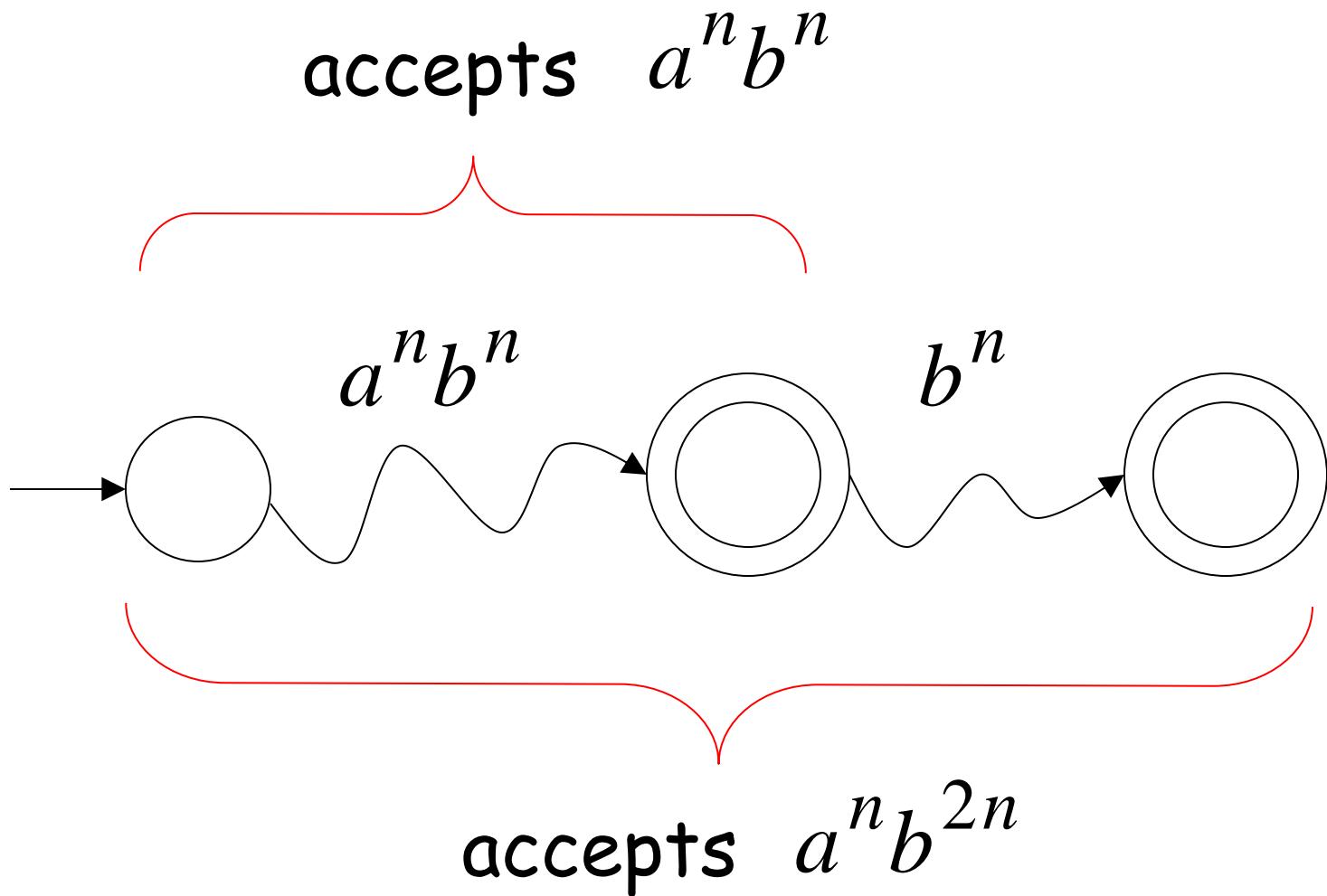
$$L = \{a^n b^n\} \cup \{a^n b^{2n}\}$$

is deterministic context free

Therefore:

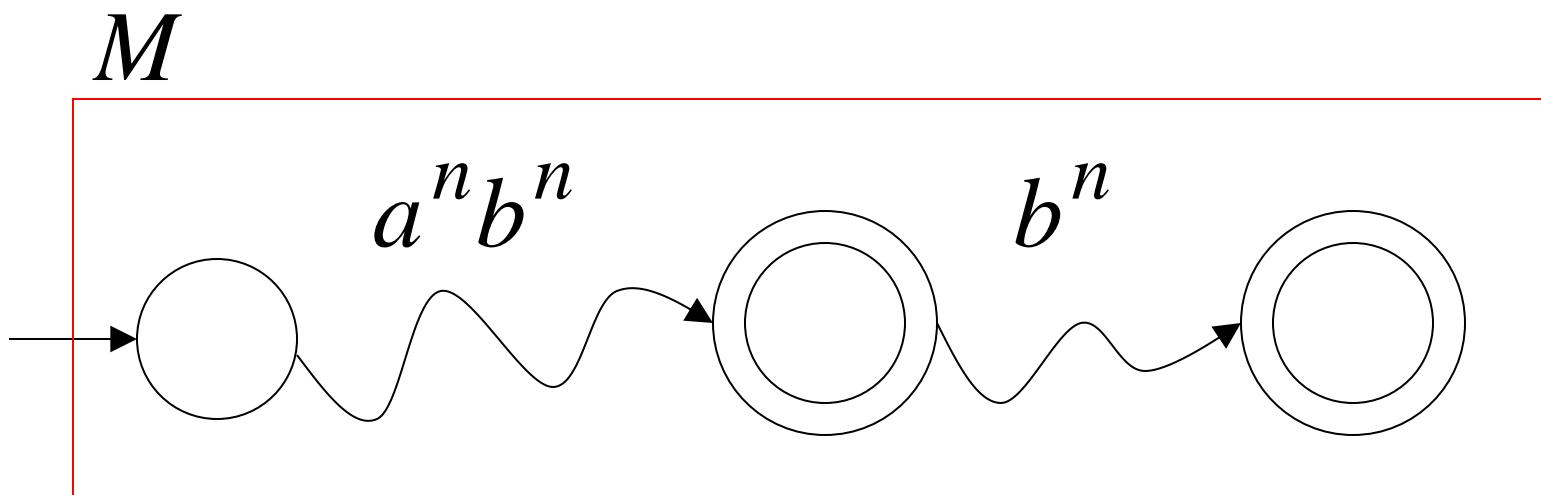
there is a DPDA M that accepts L

DPDA M with $L(M) = \{a^n b^n\} \cup \{a^n b^{2n}\}$

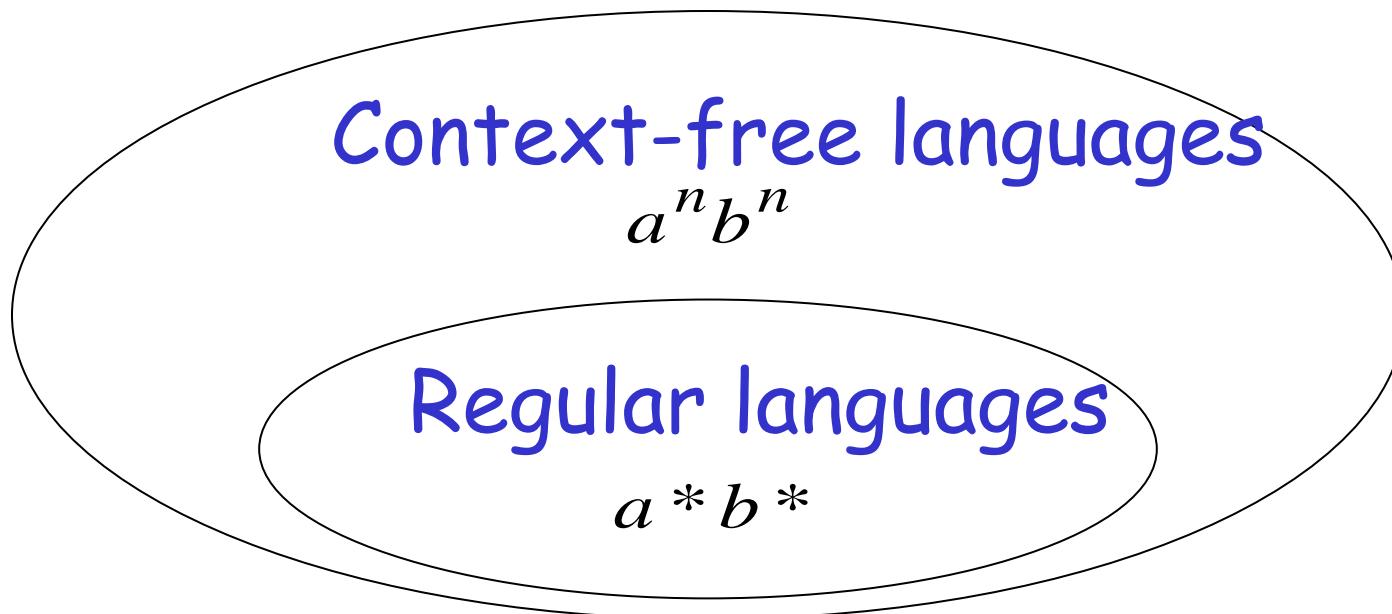


DPDA M with $L(M) = \{a^n b^n\} \cup \{a^n b^{2n}\}$

Such a path exists due to determinism



Fact 1: The language $\{a^n b^n c^n\}$
is not context-free



(we will prove this at a later class using
pumping lemma for context-free languages)

Fact 2: The language $L \cup \{a^n b^n c^n\}$ is not context-free

$$(L = \{a^n b^n\} \cup \{a^n b^{2n}\})$$

(we can prove this using pumping lemma for context-free languages)

We will construct a PDA that accepts:

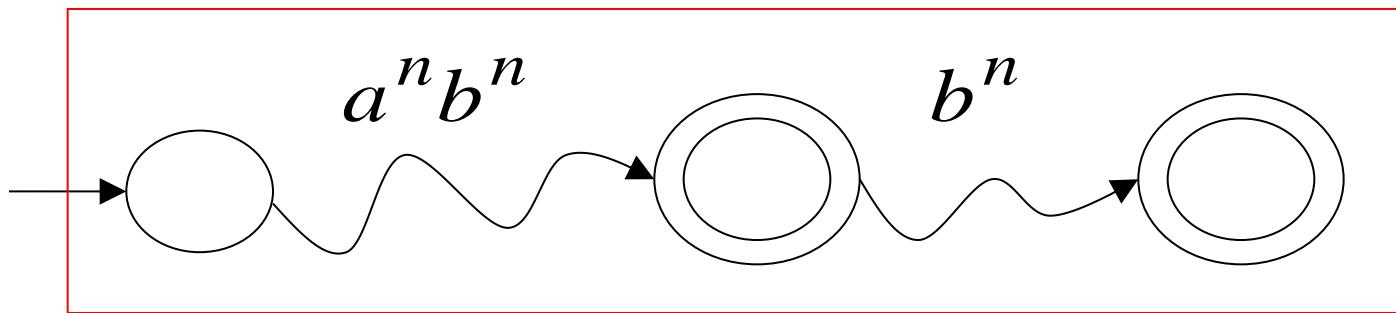
$$L \cup \{a^n b^n c^n\}$$

$$(L = \{a^n b^n\} \cup \{a^n b^{2n}\})$$

which is a contradiction!

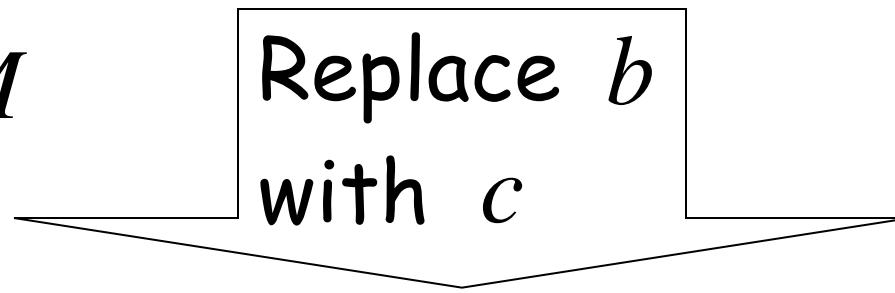
DPDA M

$$L(M) = \{a^n b^n\} \cup \{a^n b^{2n}\}$$



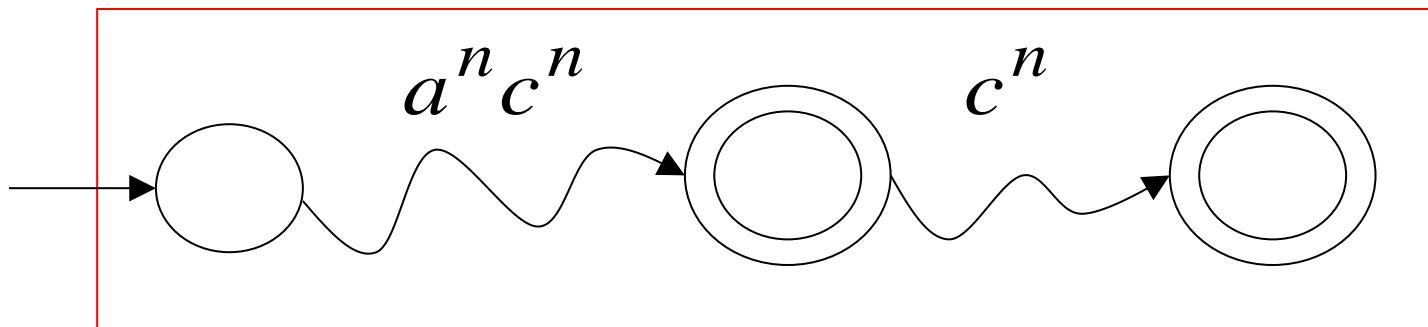
Modify M

Replace b
with c



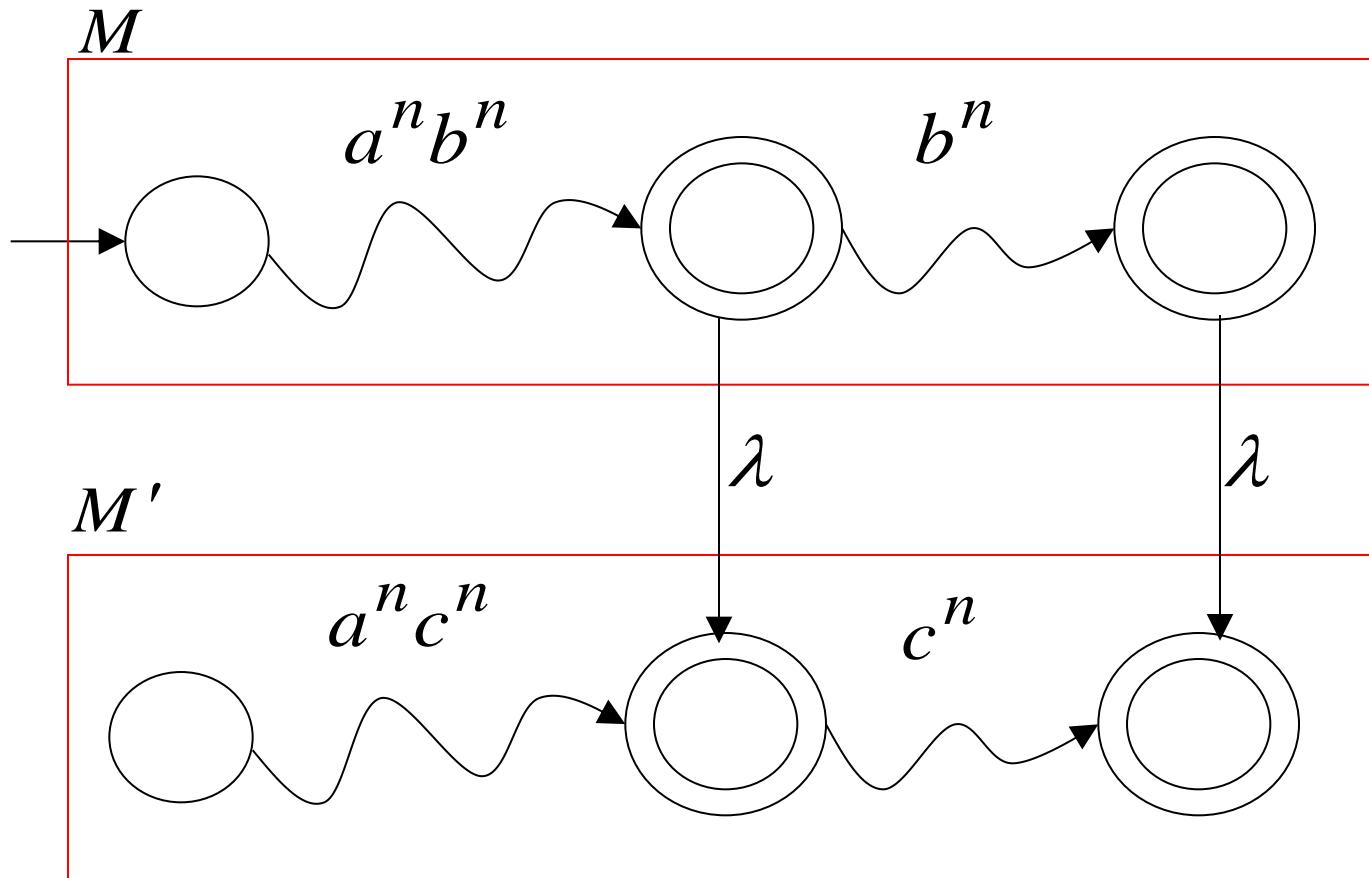
DPDA M'

$$L(M') = \{a^n c^n\} \cup \{a^n c^{2n}\}$$



A PDA that accepts $L \cup \{a^n b^n c^n\}$

Connect the final states of M
with the final states of M'



Since $L \cup \{a^n b^n c^n\}$ is accepted by a PDA
it is context-free

Contradiction!

(since $L \cup \{a^n b^n c^n\}$ is not context-free)

Therefore:

$$L = \{a^n b^n\} \cup \{a^n b^{2n}\}$$

Is not deterministic context free

There is no DPDA that accepts it

End of Proof

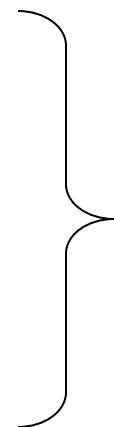
Properties of Context-Free languages

Union

Context-free languages
are closed under: **Union**

L_1 is context free

L_2 is context free



$L_1 \cup L_2$

is context-free

Example

Language

$$L_1 = \{a^n b^n\}$$

Grammar

$$S_1 \rightarrow aS_1b \mid \lambda$$

$$L_2 = \{ww^R\}$$

$$S_2 \rightarrow aS_2a \mid bS_2b \mid \lambda$$

Union

$$L = \{a^n b^n\} \cup \{ww^R\}$$

$$S \rightarrow S_1 \mid S_2$$

In general:

For context-free languages L_1, L_2
with context-free grammars G_1, G_2
and start variables S_1, S_2

The grammar of the **union** $L_1 \cup L_2$
has new start variable S
and additional production $S \rightarrow S_1 \mid S_2$

Concatenation

Context-free languages
are closed under:

Concatenation

L_1 is context free

L_2 is context free



$L_1 L_2$

is context-free

Example

Language

$$L_1 = \{a^n b^n\}$$

Grammar

$$S_1 \rightarrow aS_1b \mid \lambda$$

$$L_2 = \{ww^R\}$$

$$S_2 \rightarrow aS_2a \mid bS_2b \mid \lambda$$

Concatenation

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

$$S \rightarrow S_1 S_2$$

In general:

For context-free languages L_1, L_2
with context-free grammars G_1, G_2
and start variables S_1, S_2

The grammar of the **concatenation** L_1L_2
has new start variable S
and additional production $S \rightarrow S_1S_2$

Star Operation

Context-free languages

are closed under:

Star-operation

L is context free



L^* is context-free

Example

Language

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

Grammar

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

Star Operation

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

$$S_1 \rightarrow SS_1 \mid \lambda$$

In general:

For context-free language L
with context-free grammar G
and start variable S

The grammar of the **star operation** L^*
has new start variable S_1
and additional production $S_1 \rightarrow SS_1 \mid \lambda$

Negative Properties of Context-Free Languages

Intersection

Context-free languages
are not closed under:

intersection

L_1 is context free

L_2 is context free



$$L_1 \cap L_2$$

not necessarily
context-free

Example

$$L_1 = \{a^n b^n c^m\}$$

Context-free:

$$S \rightarrow AC$$

$$A \rightarrow aAb \mid \lambda$$

$$C \rightarrow cC \mid \lambda$$

$$L_2 = \{a^n b^m c^m\}$$

Context-free:

$$S \rightarrow AB$$

$$A \rightarrow aA \mid \lambda$$

$$B \rightarrow bBc \mid \lambda$$

Intersection

$$L_1 \cap L_2 = \{a^n b^n c^n\} \quad \text{NOT context-free}$$

Complement

Context-free languages
are not closed under:

complement

L is context free



\bar{L}

not necessarily
context-free

Example

$$L_1 = \{a^n b^n c^m\}$$

$$L_2 = \{a^n b^m c^m\}$$

Context-free:

$$S \rightarrow AC$$

$$A \rightarrow aAb \mid \lambda$$

$$C \rightarrow cC \mid \lambda$$

Context-free:

$$S \rightarrow AB$$

$$A \rightarrow aA \mid \lambda$$

$$B \rightarrow bBc \mid \lambda$$

Complement

$$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2 = \{a^n b^n c^n\}$$

NOT context-free

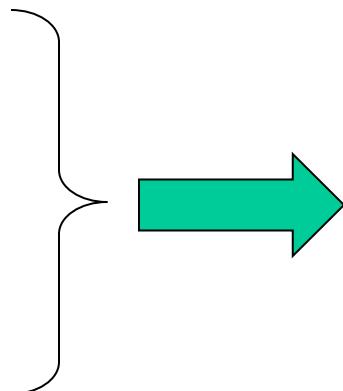
Intersection
of
Context-free languages
and
Regular Languages

The intersection of
a context-free language and
a regular language

is a context-free language

L_1 context free

L_2 regular



$$L_1 \cap L_2$$

context-free

Machine M_1

NPDA for L_1
context-free

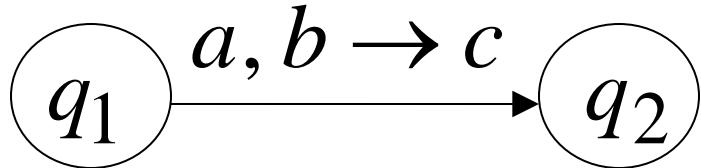
Machine M_2

DFA for L_2
regular

Construct a new NPDA machine M
that accepts $L_1 \cap L_2$

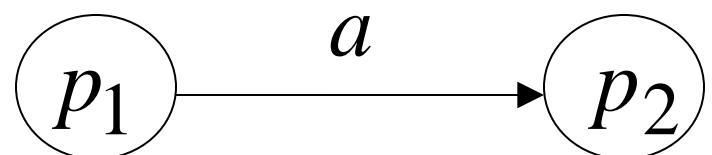
M simulates in parallel M_1 and M_2

NPDA M_1

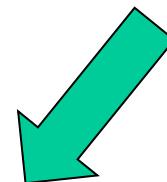
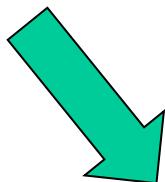


transition

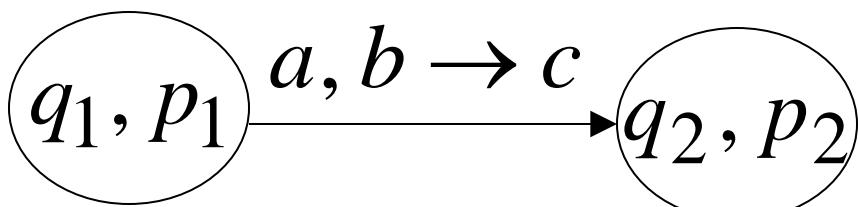
DFA M_2



transition

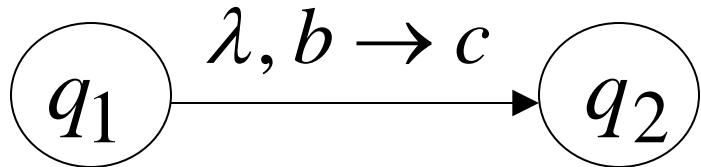


NPDA M



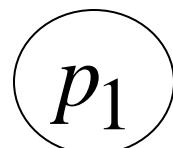
transition

NPDA M_1

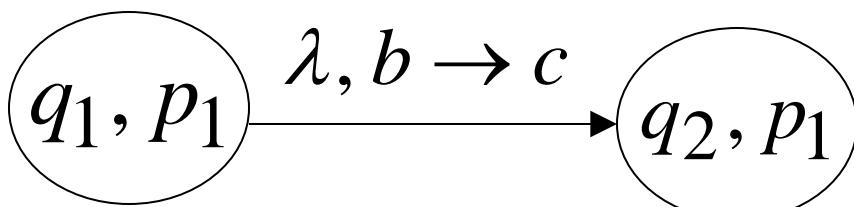


transition

DFA M_2

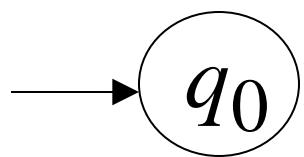


NPDA M



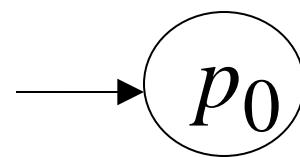
transition

NPDA M_1



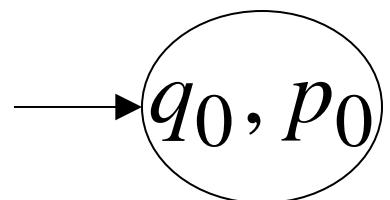
initial state

DFA M_2



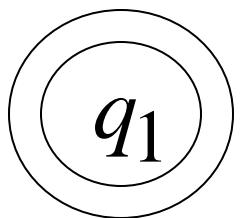
initial state

NPDA M



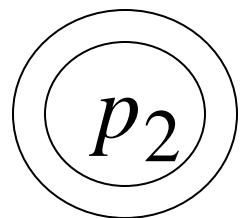
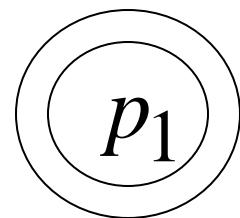
Initial state

NPDA M_1

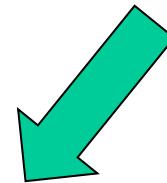
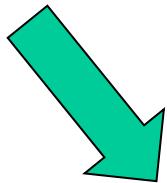


final state

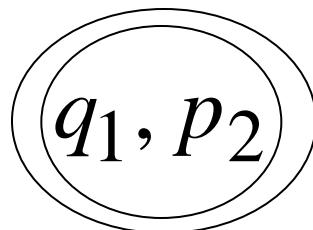
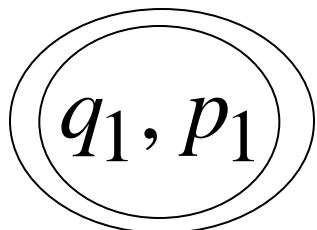
DFA M_2



final states



NPDA M



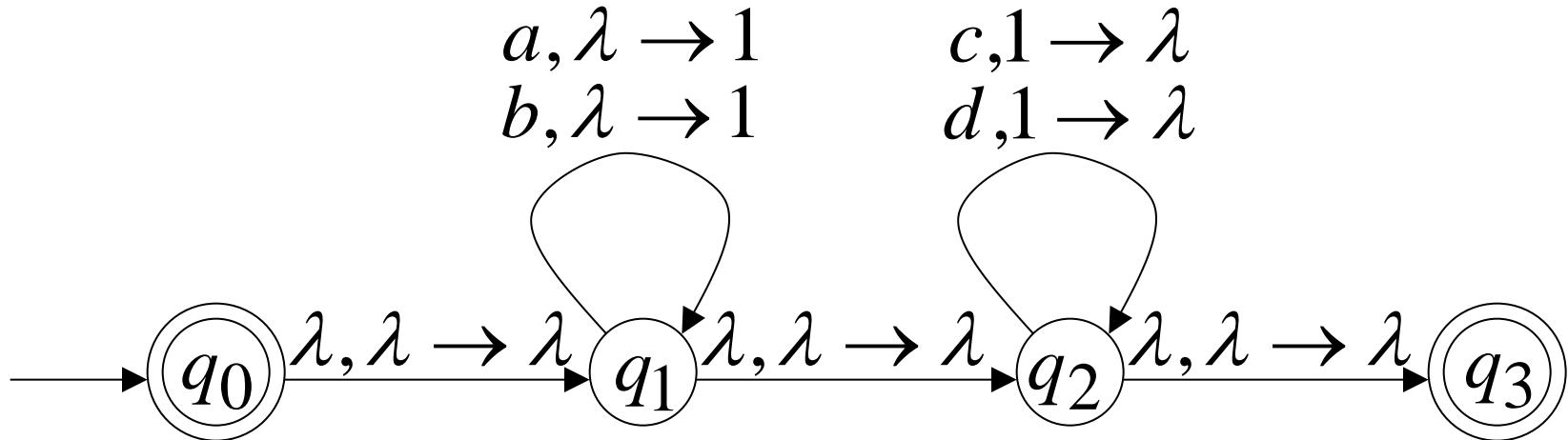
final states

Example:

context-free

$$L_1 = \{ w_1 w_2 : |w_1| = |w_2|, w_1 \in \{a,b\}^*, w_2 \in \{c,d\}^* \}$$

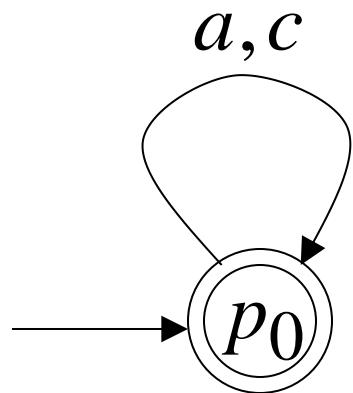
NPDA M_1



regular

$$L_2 = \{a, c\}^*$$

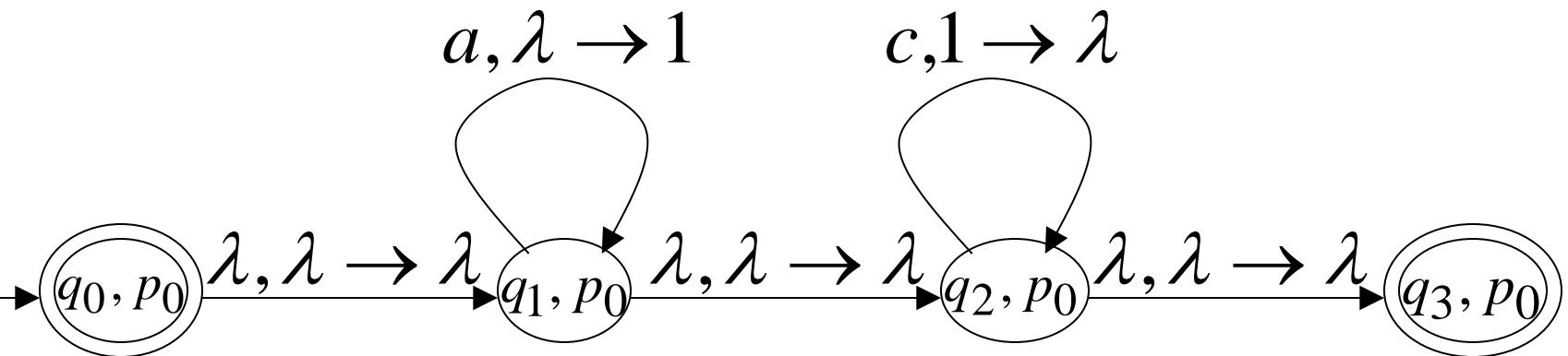
DFA M_2



context-free

Automaton for: $L_1 \cap L_2 = \{a^n c^n : n \geq 0\}$

NPDA M



In General:

M simulates in parallel M_1 and M_2

M accepts string w if and only if

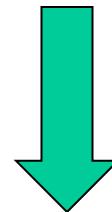
M_1 accepts string w and

M_2 accepts string w

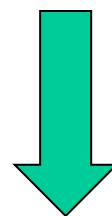
$$L(M) = L(M_1) \cap L(M_2)$$

Therefore:

M is NPDA



$L(M_1) \cap L(M_2)$ is context-free



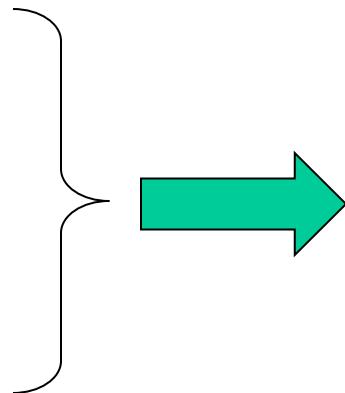
$L_1 \cap L_2$ is context-free

Applications of Regular Closure

The intersection of
a context-free language and
a regular language
is a context-free language

L_1 context free

L_2 regular



Regular Closure

$$L_1 \cap L_2$$

context-free

An Application of Regular Closure

Prove that: $L = \{a^n b^n : n \neq 100, n \geq 0\}$

is context-free

We know:

$\{a^n b^n : n \geq 0\}$ is context-free

We also know:

$$L_1 = \{a^{100}b^{100}\} \quad \text{is regular}$$



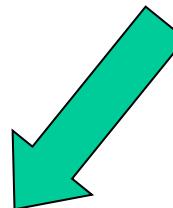
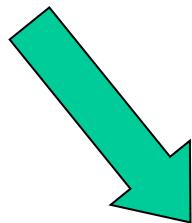
$$\overline{L_1} = \{(a+b)^*\} - \{a^{100}b^{100}\} \quad \text{is regular}$$

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

$$\overline{L_1} = \{(a+b)^*\} - \{a^{100}b^{100}\}$$

context-free

regular



(regular closure)

$$\{a^n b^n\} \cap \overline{L_1}$$

context-free



$$\{a^n b^n\} \cap \overline{L_1} = \{a^n b^n : n \neq 100, n \geq 0\} = L$$

is context-free

Another Application of Regular Closure

Prove that: $L = \{w : n_a = n_b = n_c\}$

is **not** context-free

If $L = \{w : n_a = n_b = n_c\}$ is context-free

(regular closure)

Then $L \cap \{a^*b^*c^*\} = \{a^n b^n c^n\}$

context-free

regular

context-free

Impossible!!!

Therefore, L is **not** context free

Pumping Lemma for Context-free Languages

Take an **infinite** context-free language



Generates an infinite number
of different strings

Example: $S \rightarrow ABE \mid bBd$

$A \rightarrow Aa \mid a$

$B \rightarrow bSD \mid cc$

$D \rightarrow Dd \mid d$

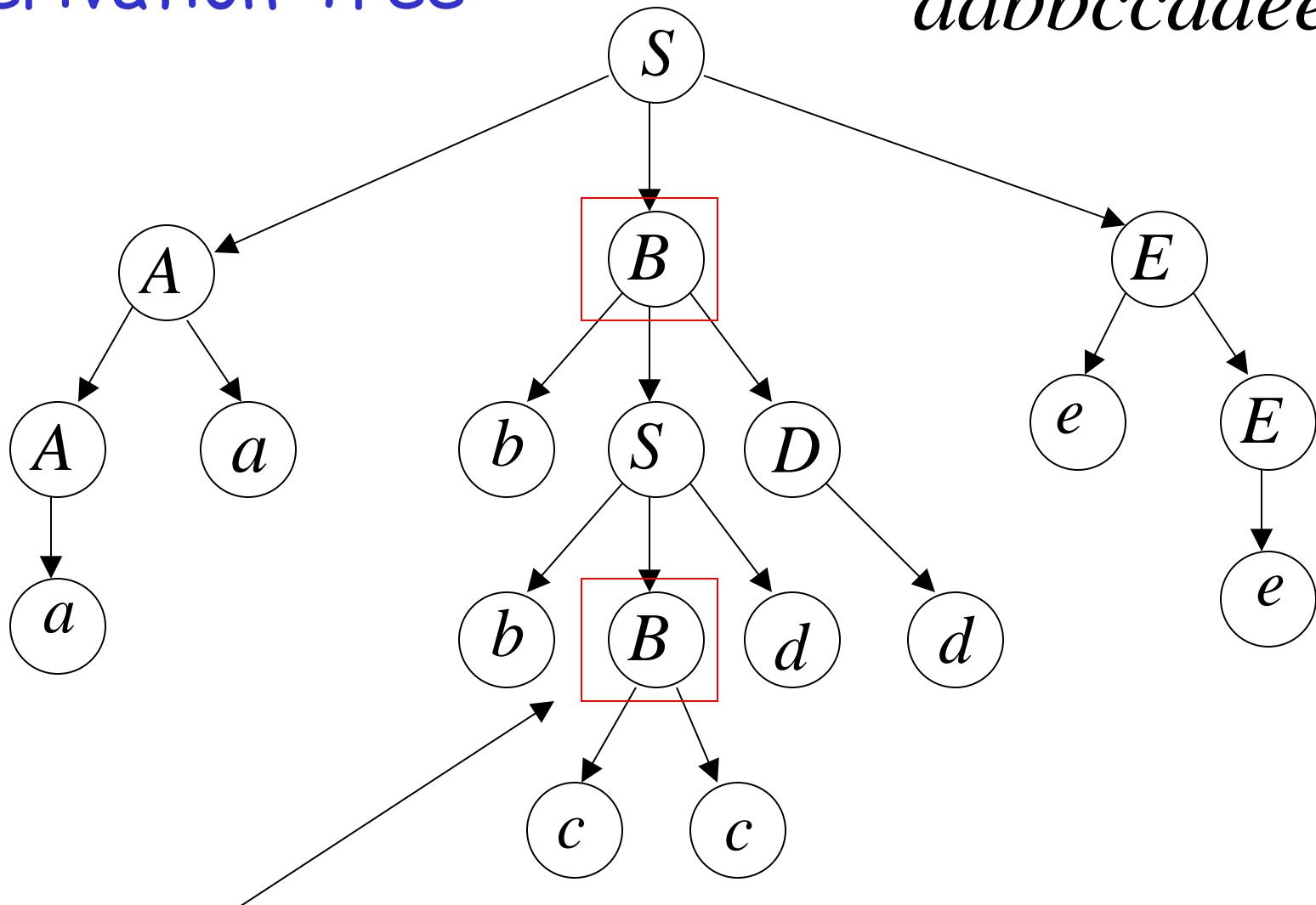
$E \rightarrow eE \mid e$

In a derivation of a “long” enough string, variables are repeated

A possible derivation:

$$\begin{aligned} S &\Rightarrow A\boxed{B}E \Rightarrow AaBE \Rightarrow aaBE \\ &\Rightarrow aabSDE \Rightarrow aabb\boxed{B}dDE \Rightarrow \\ &\Rightarrow aaabbcccdDE \Rightarrow aabbccddE \\ &\Rightarrow aabbccddeE \Rightarrow aabbccdddee \end{aligned}$$

Derivation Tree

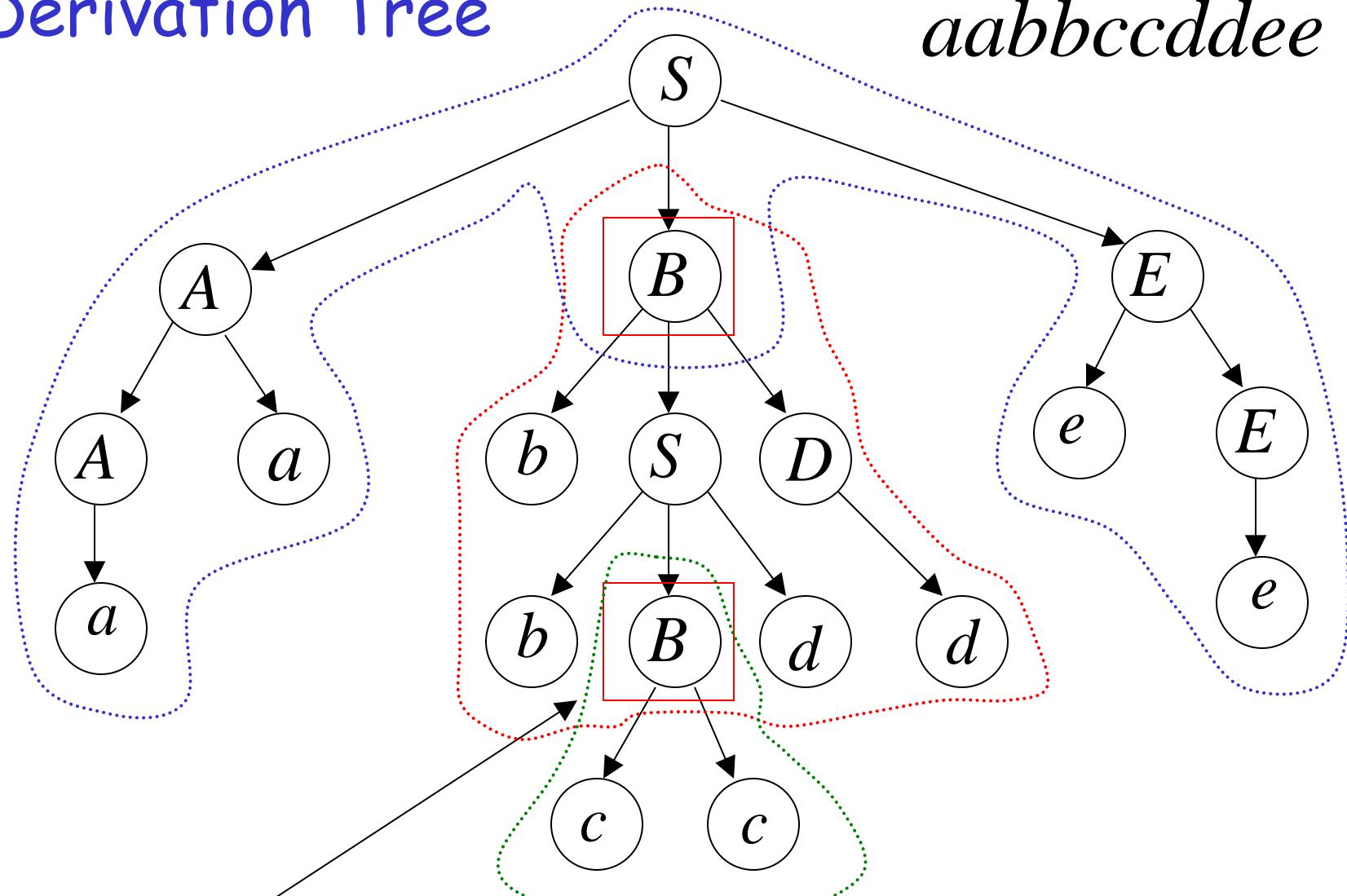


Repeated
variable

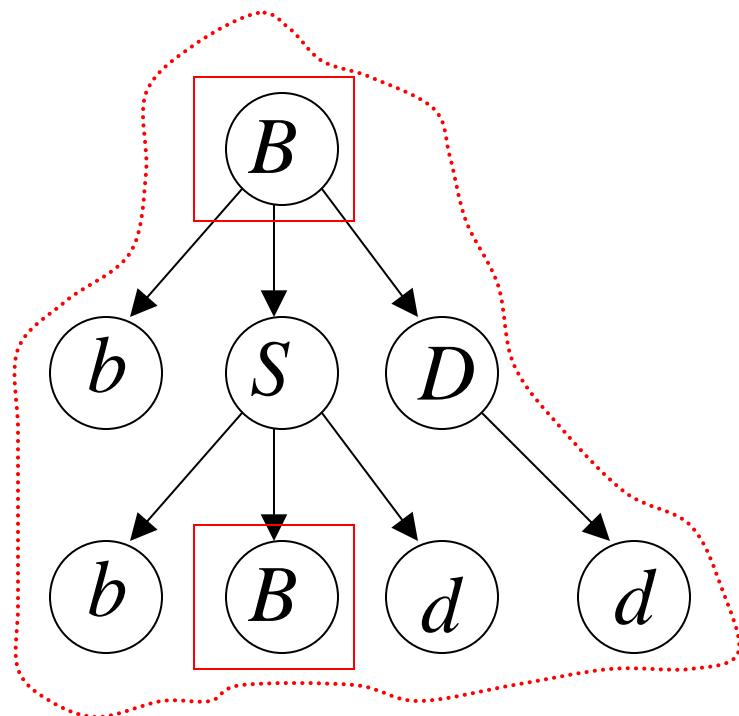
aabbcccddee

Derivation Tree

aabbcccddee

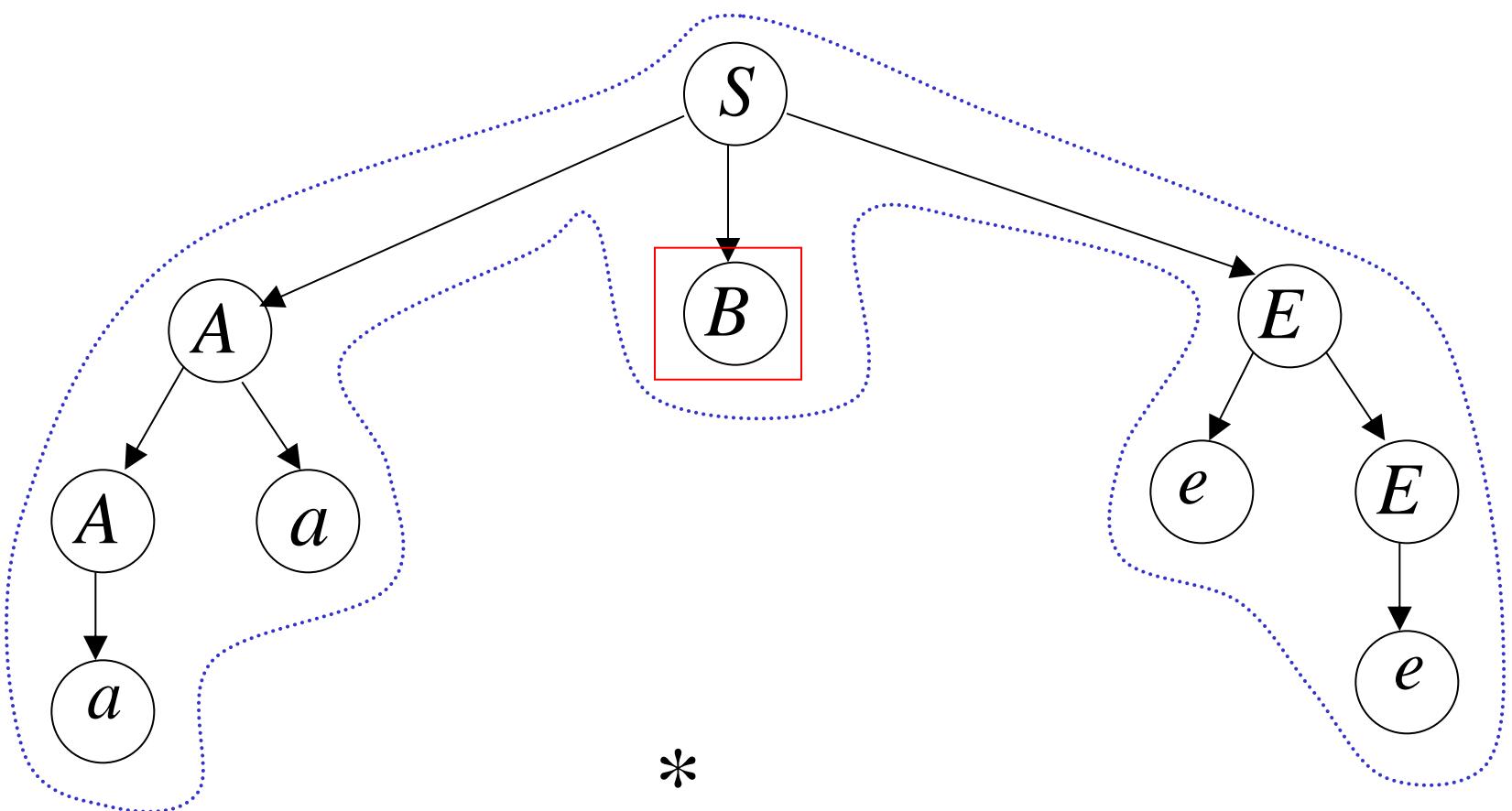


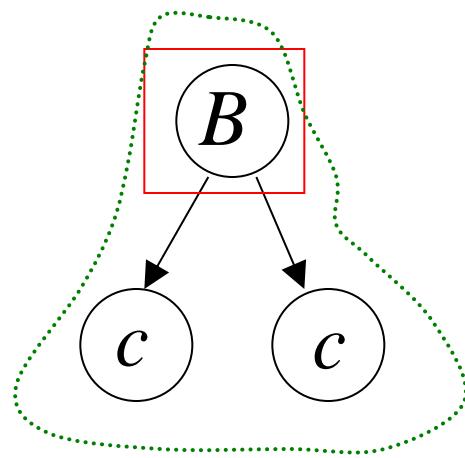
Repeated
variable

$$B \Rightarrow bSD \Rightarrow bbBdD \Rightarrow bbBdd$$


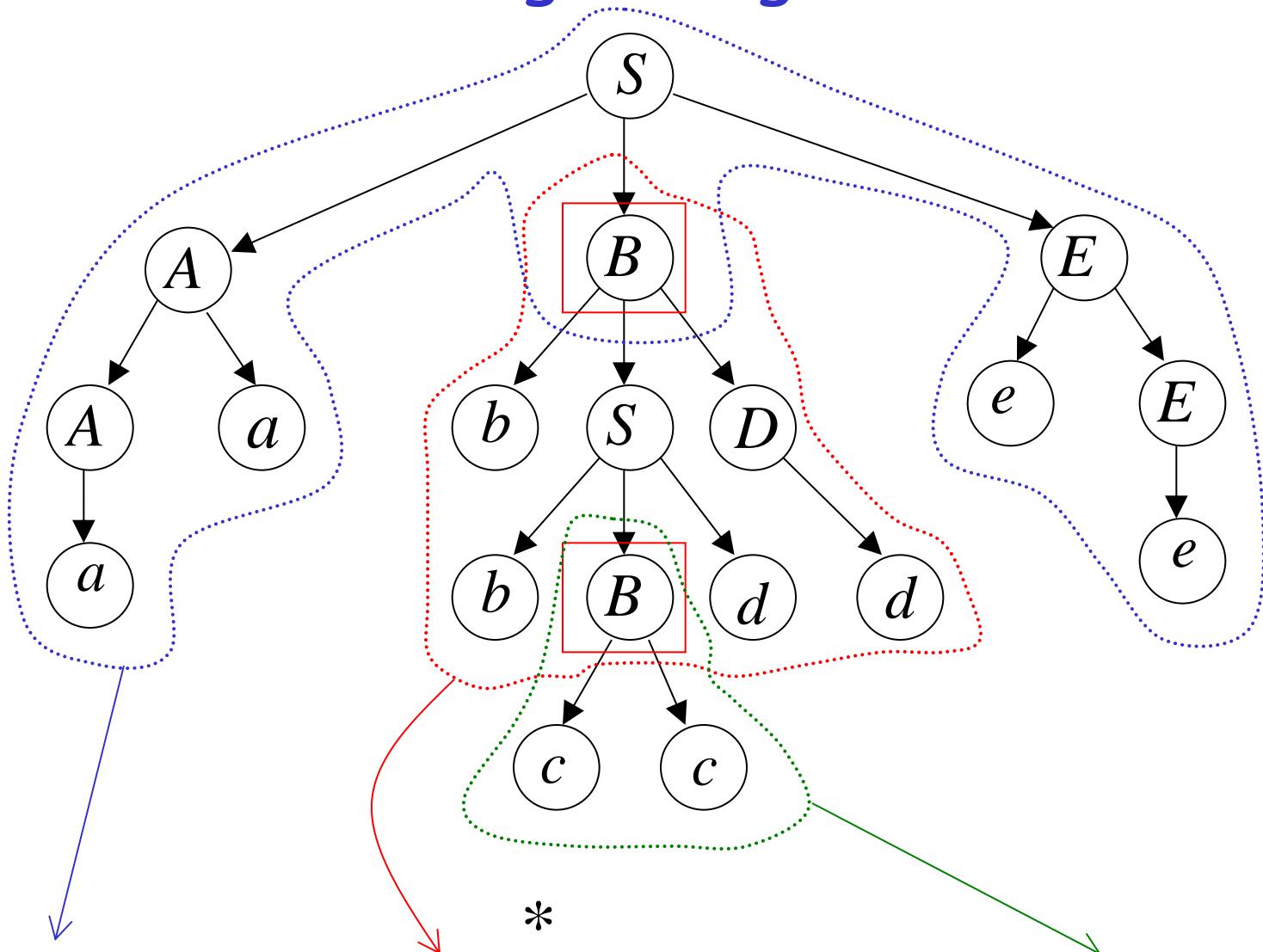
*

$$B \Rightarrow bbBdd$$

$$S \Rightarrow ABE \Rightarrow AaBE \Rightarrow aaBE \Rightarrow aaBeE \Rightarrow aaBee$$

$$S \Rightarrow aaBee$$


$$B \Rightarrow cc$$

Putting all together

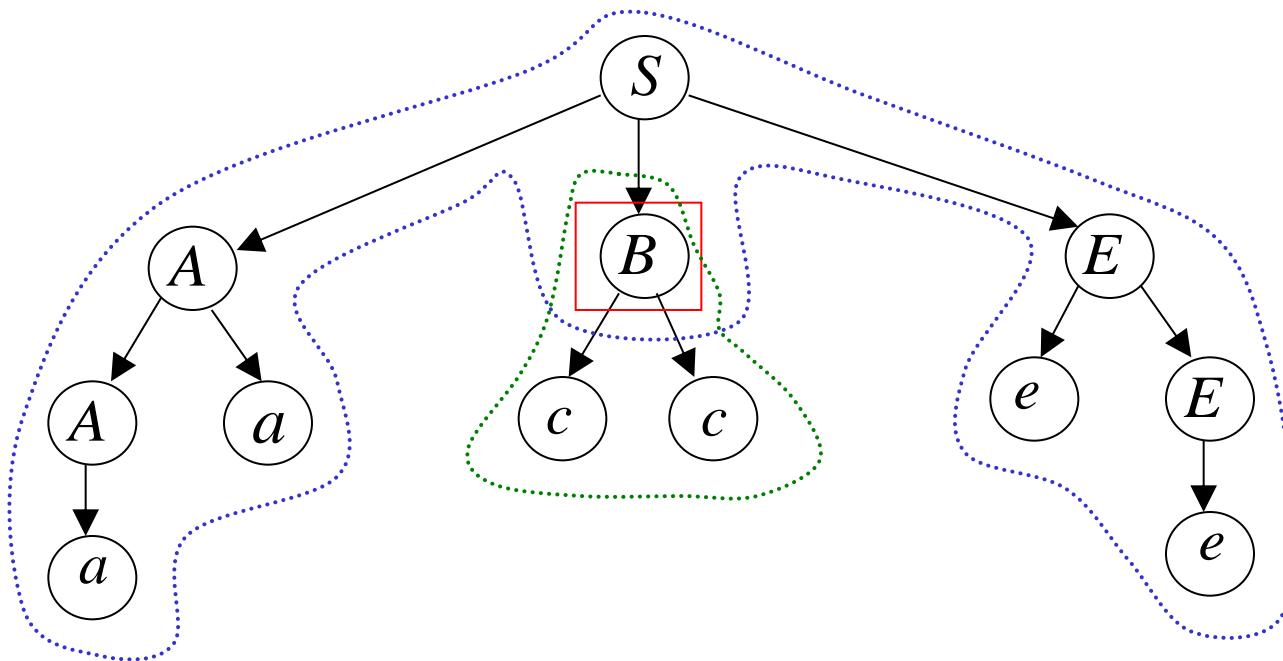


$S \Rightarrow aaBee$

$B \Rightarrow bbBdd$

$B \Rightarrow cc$

We can remove the middle part



$$* \\ S \Rightarrow aa(bb)^0 cc(dd)^0 ee$$

$*$ $*$

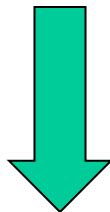
$$S \Rightarrow aaBee$$

$$B \Rightarrow bbBdd$$

$$B \Rightarrow cc$$

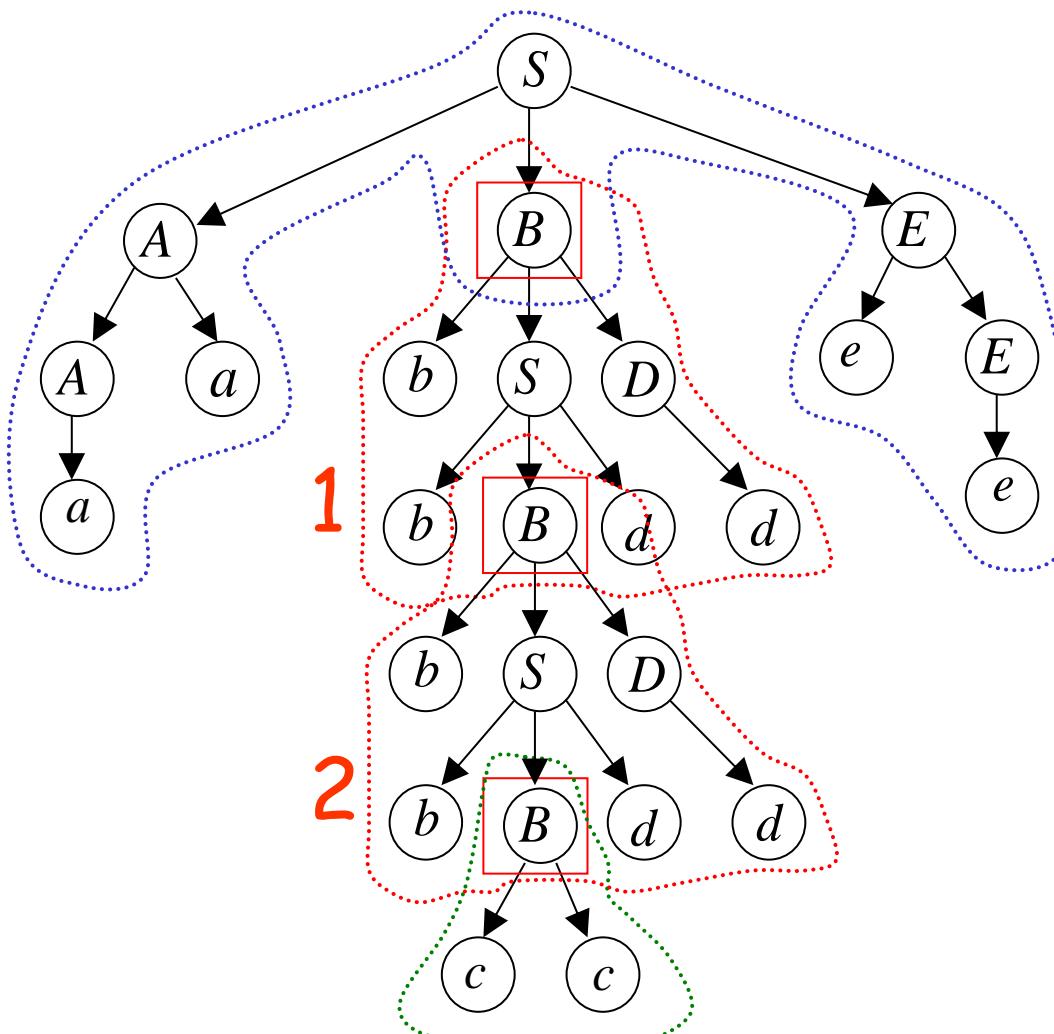
 $*$ $*$

$$S \Rightarrow aaBee \Rightarrow aaccee = aa(bb)^0 cc(dd)^0 ee$$



$$aa(bb)^0 cc(dd)^0 ee \in L(G)$$

We can repeated middle part two times



*

$$S \Rightarrow aa(bb)^2cc(dd)^2ee$$

$*$ $*$

$$S \Rightarrow aaBee$$

$$B \Rightarrow bbBdd$$

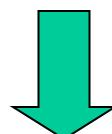
$$B \Rightarrow cc$$

 $*$ $*$

$$S \Rightarrow aaBee \Rightarrow aabbBddee$$

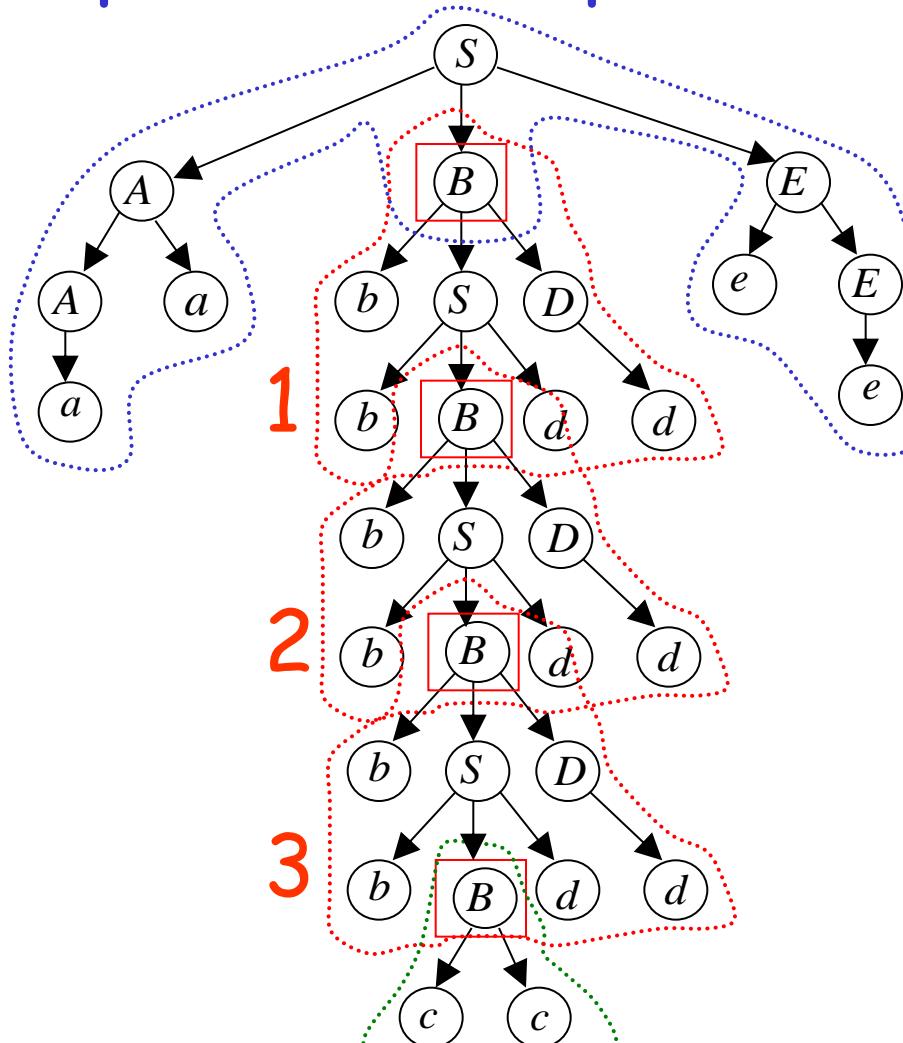
 $*$ $*$

$$\Rightarrow aa(bb)^2 B(dd)^2 ee \Rightarrow aa(bb)^2 cc(dd)^2 ee$$



$$aa(bb)^2 cc(dd)^2 ee \in L(G)$$

We can repeat middle part three times



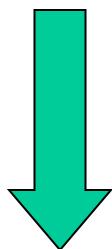
$$S \Rightarrow aa(bb)^3cc(dd)^3ee$$

$*$ $*$

$$S \Rightarrow aaBee$$

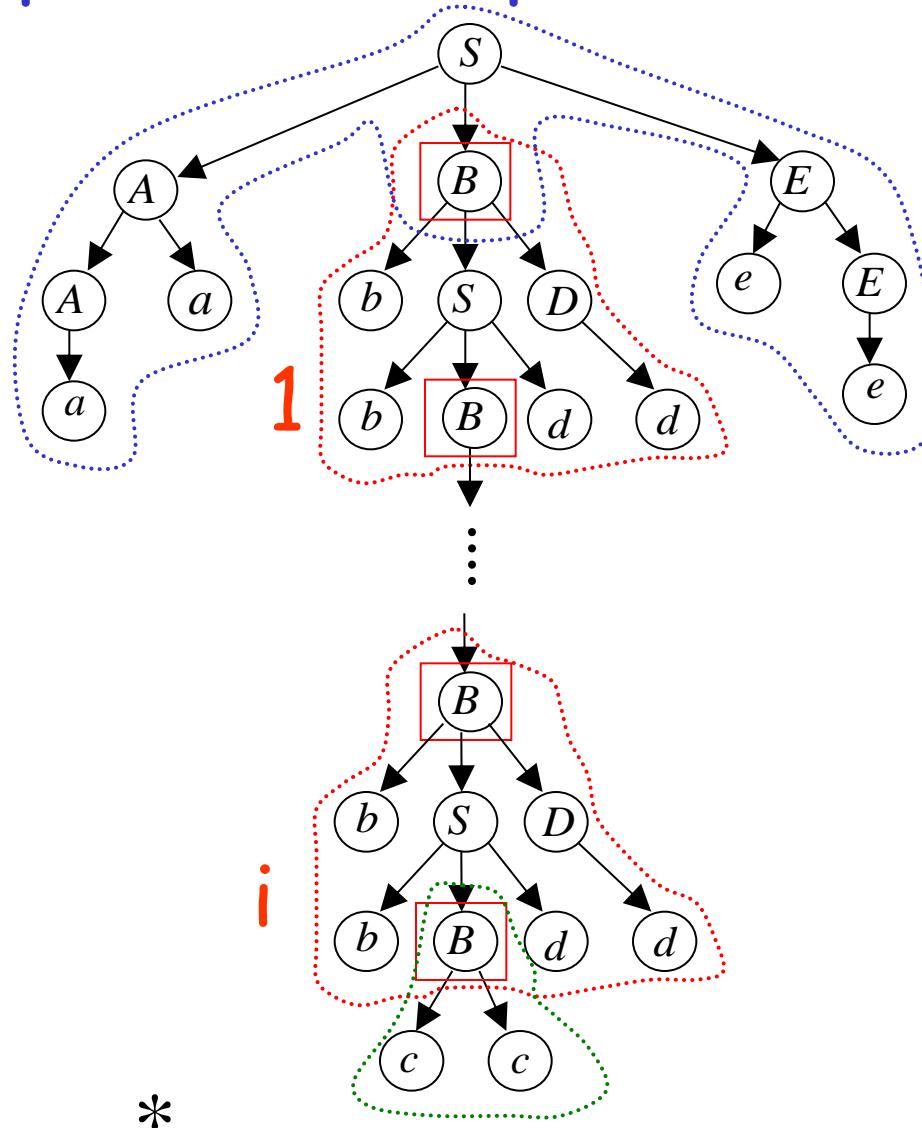
$$B \Rightarrow bbBdd$$

$$B \Rightarrow cc$$

 $*$

$$S \Rightarrow aa(bb)^3cc(dd)^3ee \in L(G)$$

Repeat middle part i times



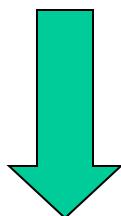
$$S \Rightarrow aa(bb)^i cc(dd)^i ee$$

$*$ $*$

$$S \xrightarrow{*} aaBee$$

$$B \xrightarrow{*} bbBdd$$

$$B \Rightarrow cc$$

 $*$

$$S \xrightarrow{*} aa(bb)^i cc(dd)^i ee \in L(G)$$

For any $i \geq 0$

From Grammar

$$S \rightarrow ABE \mid bBd$$

$$A \rightarrow Aa \mid a$$

$$B \rightarrow bSD \mid cc$$

$$D \rightarrow Dd \mid d$$

$$E \rightarrow eE \mid e$$

and given string

$$aabcccddee \in L(G)$$

We inferred that a family of strings is in $L(G)$

*

$$S \Rightarrow aa(bb)^i cc(dd)^i ee \in L(G) \text{ for any } i \geq 0$$

Arbitrary Grammars

Consider now an arbitrary infinite context-free language L

Let G be the grammar of $L - \{\lambda\}$

Take G so that it has no unit-productions
and no λ -productions

(remove them)

Let r be the number of variables

Let t be the maximum right-hand size
of any production

Example:

$$S \rightarrow ABE \mid bBd$$

$A \rightarrow Aa \mid a$

$$B \rightarrow bSD \mid cc$$

$D \rightarrow Dd \mid d$

$$E \rightarrow eE \mid e$$

$r = 5$

$t = 3$

```
graph LR; S["S → ABE | bBd"] --> r["r = 5"]; B["B → bSD | cc"] --> t["t = 3"]
```

Claim:

Take string $w \in L(G)$ with $|w| > t^r$.
Then in the derivation tree of w
there is a path from the root to a leaf
where a variable of G is repeated

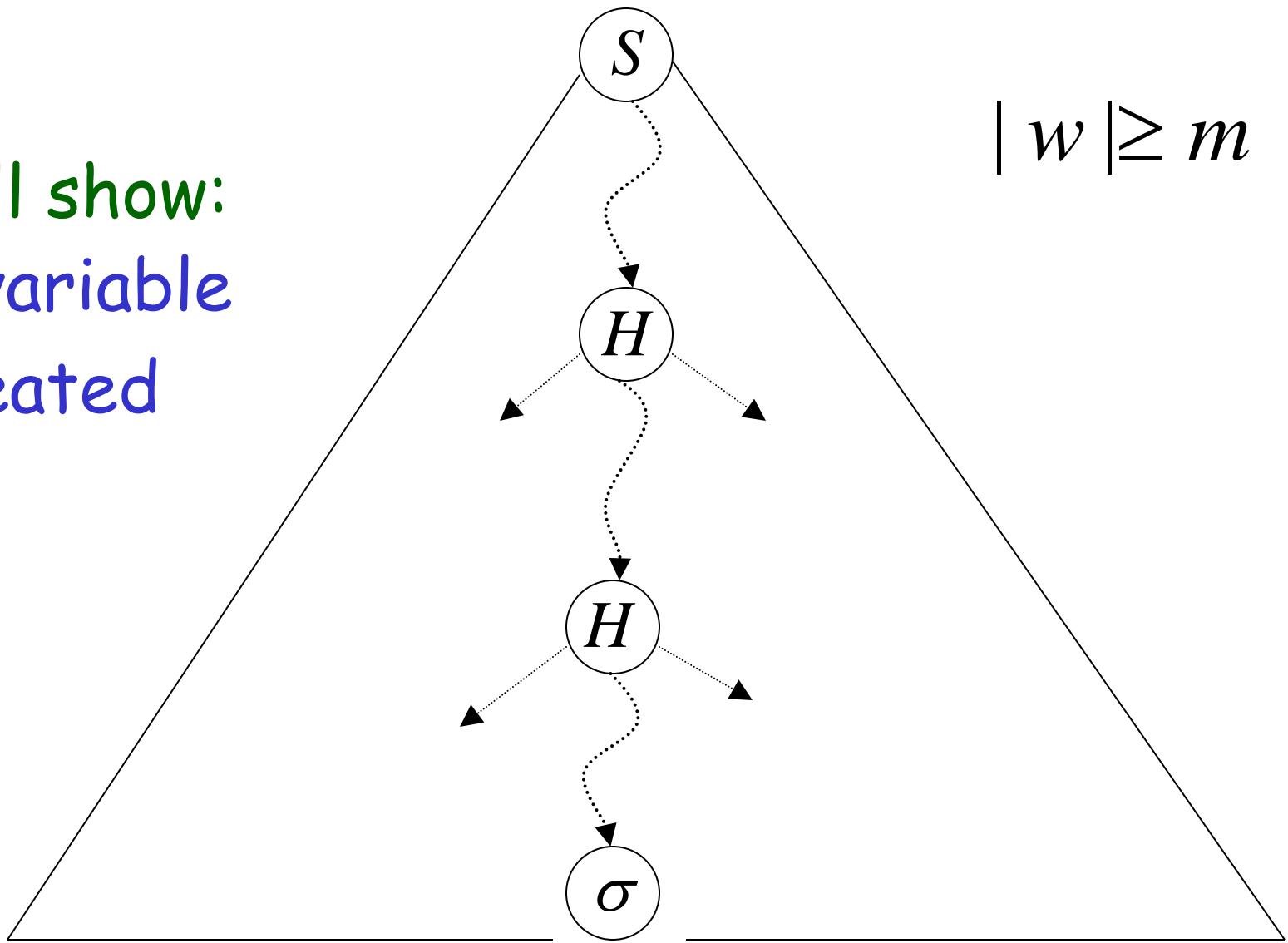
Proof:

Proof by contradiction

Derivation tree of w

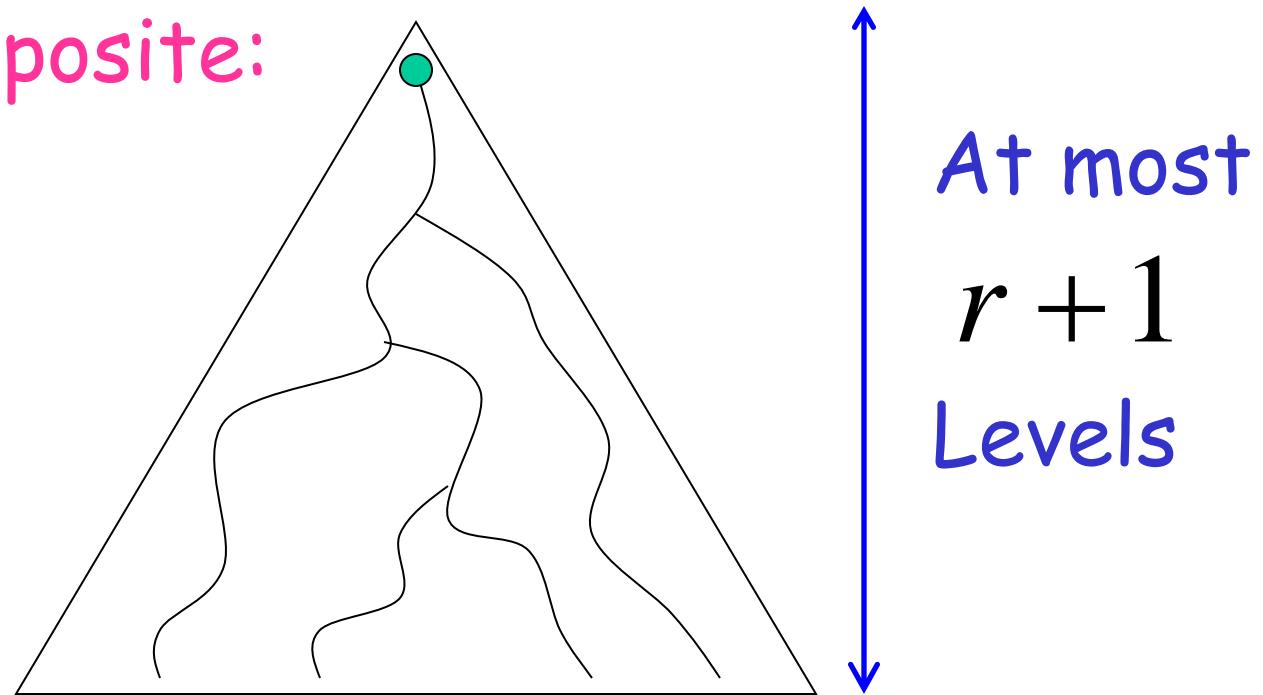
$$|w| \geq m$$

We will show:
some variable
is repeated

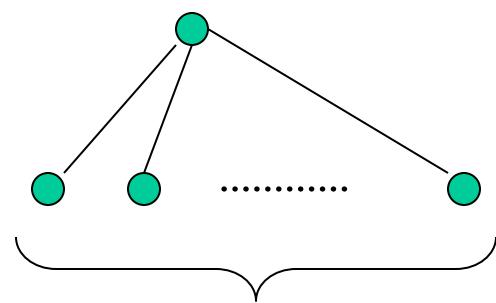


First we show that the tree of w has at least $r + 2$ levels of nodes

Suppose the opposite:



Maximum number of nodes per level



Level 0: 1 nodes

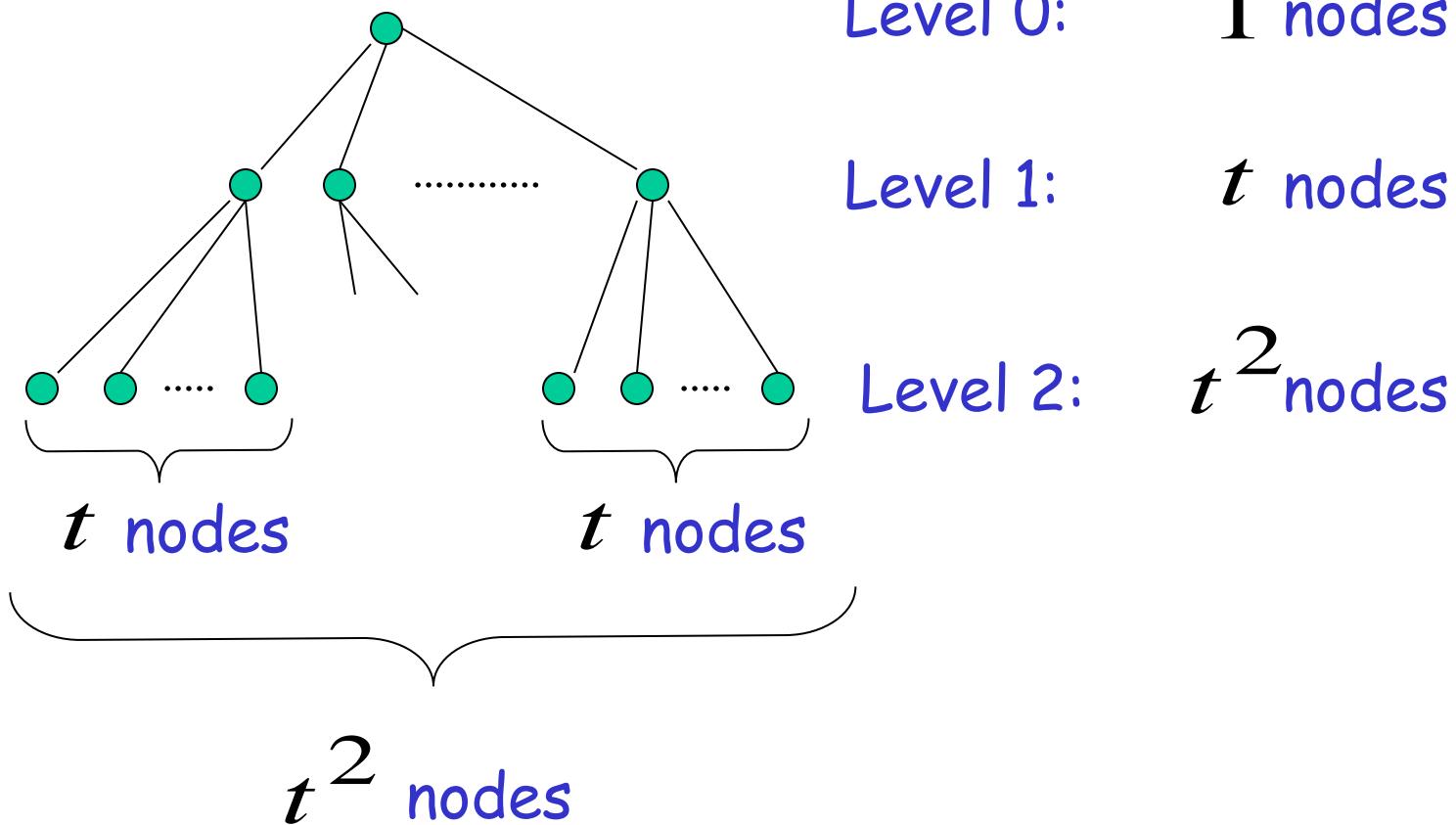
Level 1: t nodes

t nodes

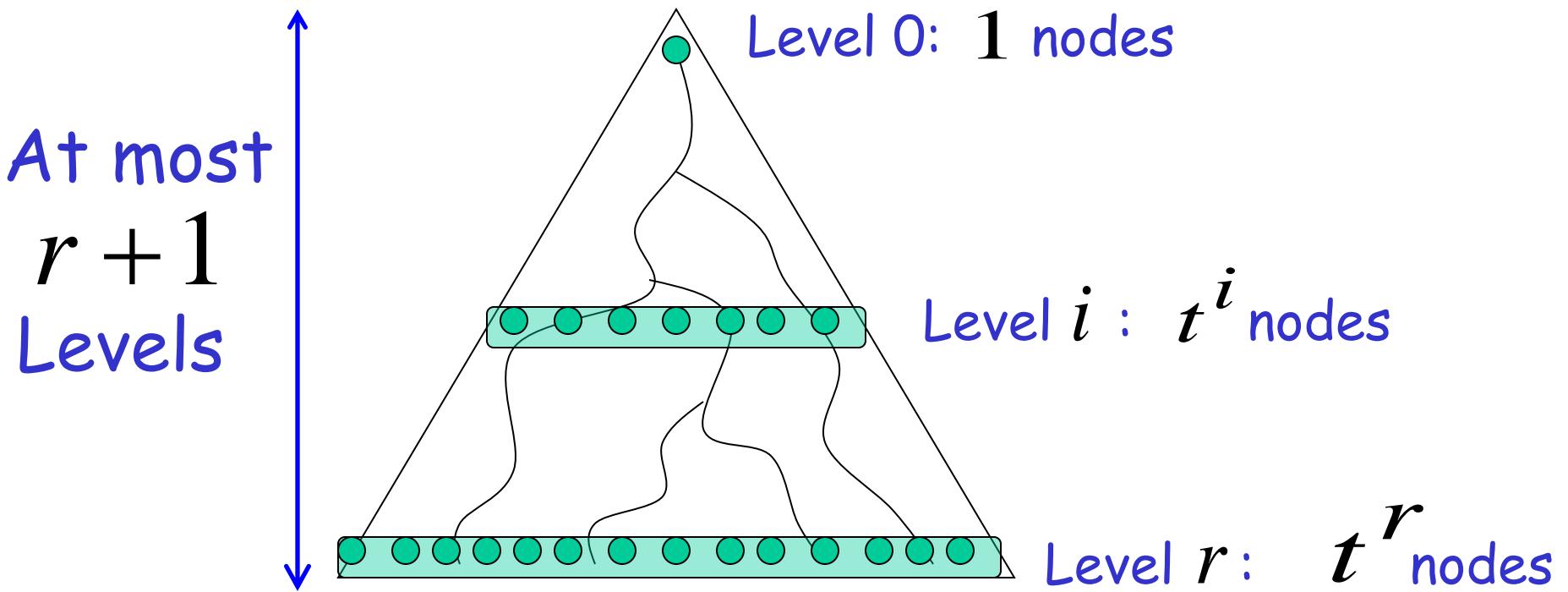


The maximum right-hand side of any production

Maximum number of nodes per level



Maximum number of nodes per level



Maximum possible string length

= max nodes at level r =

$$t^r$$

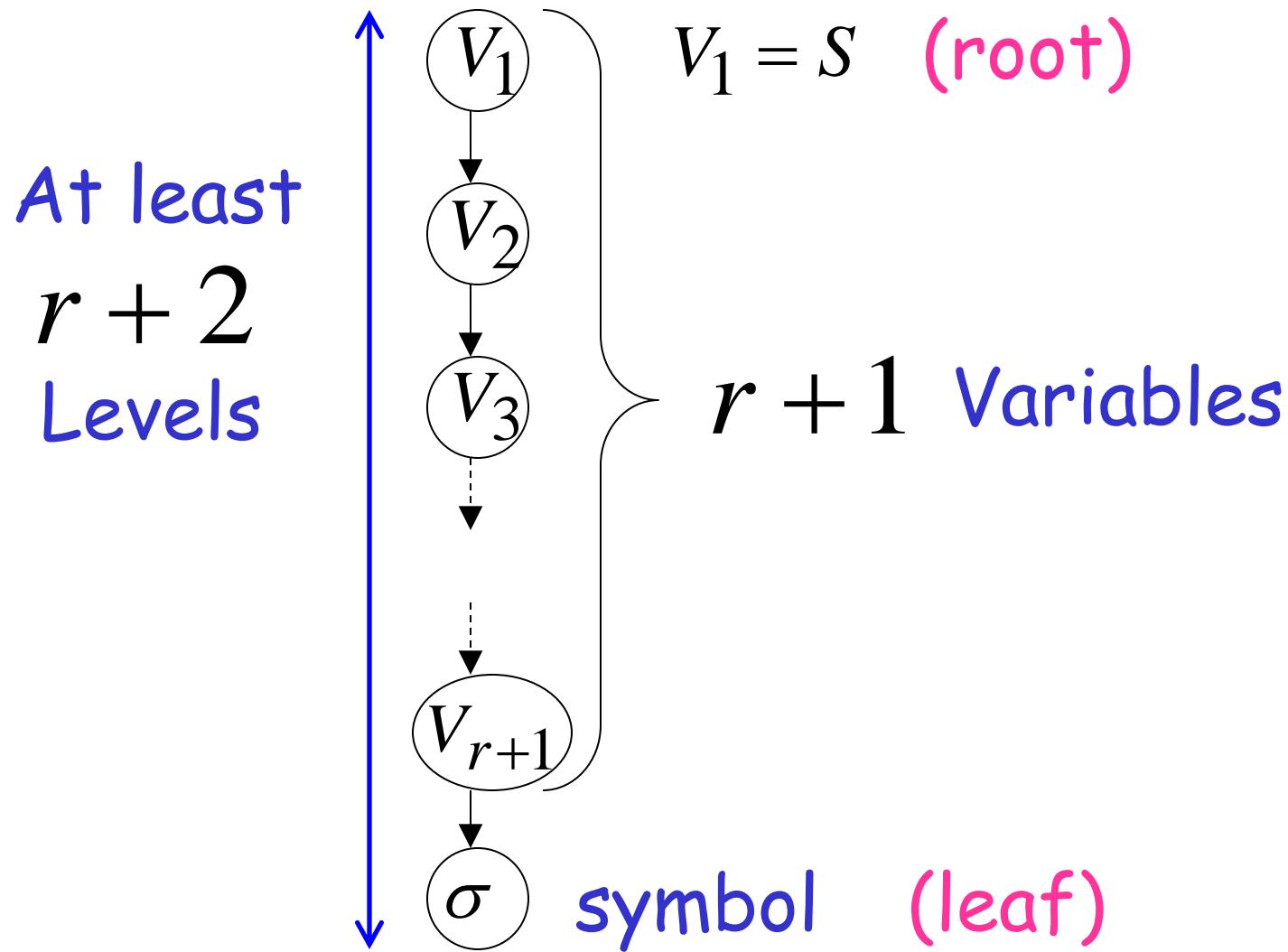
Therefore,
maximum length of string $w : |w| \leq t^r$

However we took, $|w| > t^r$

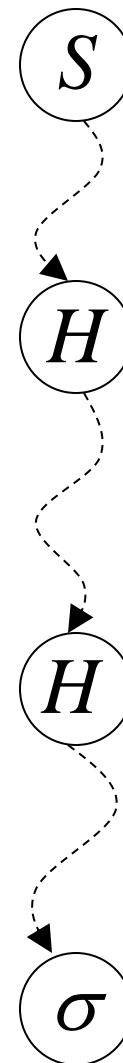
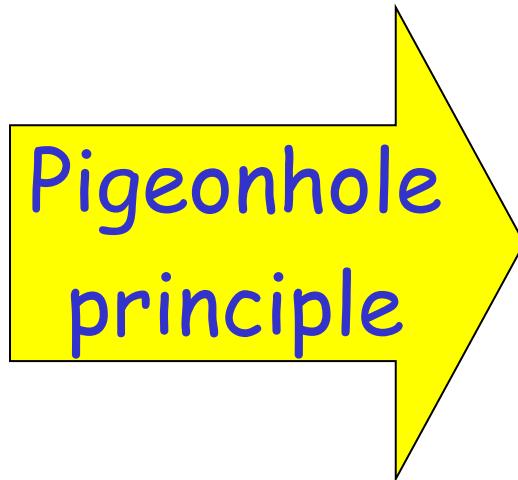
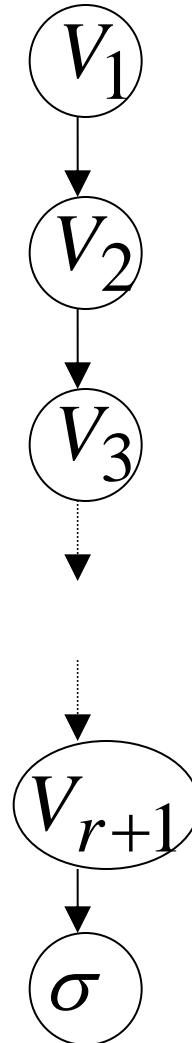
Contradiction!!!

Therefore,
the tree must have at least $r + 2$ levels

Thus, there is a path from the root to a leaf with at least $r + 2$ nodes



Since there are at most r different variables,
some variable is repeated

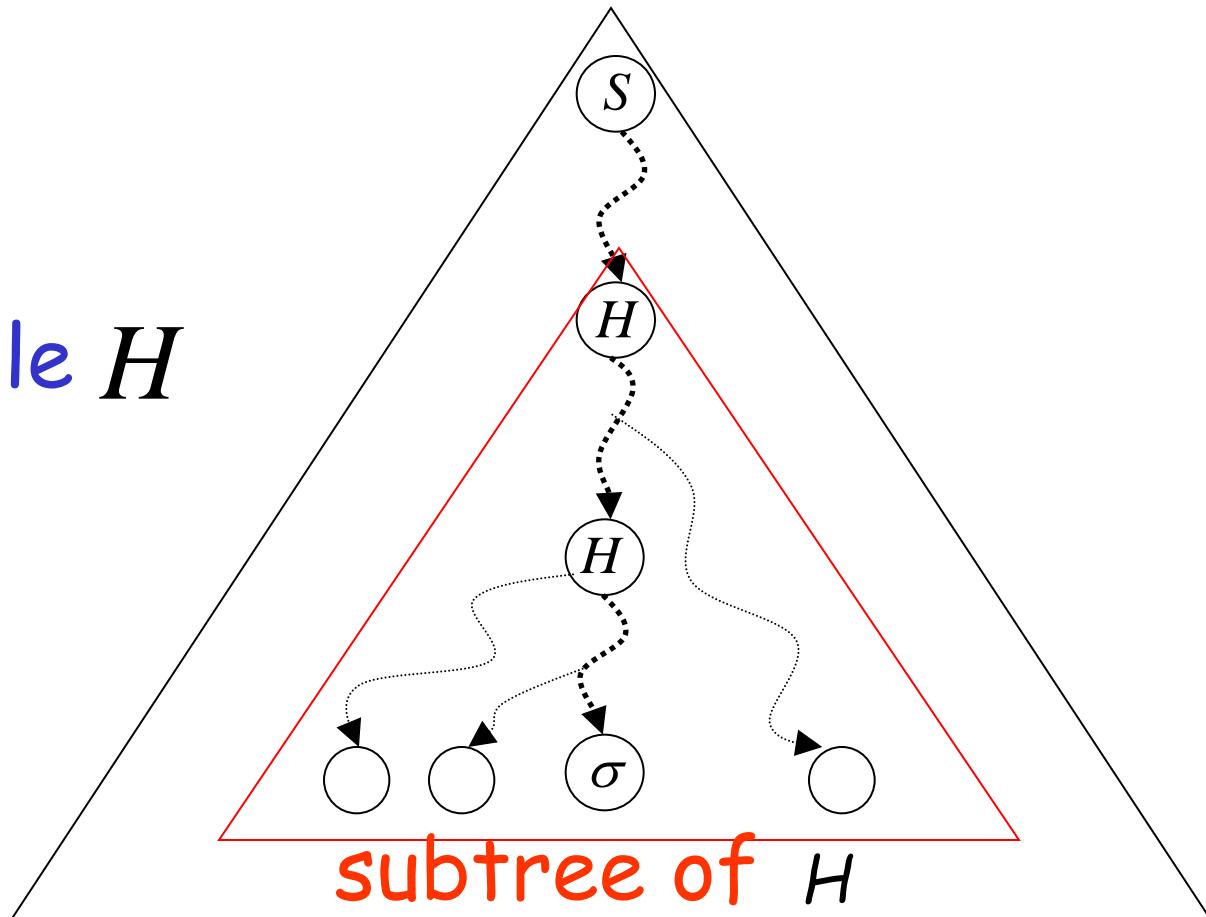


END OF CLAIM PROOF

Take now a string w with $|w| > t^r$

From claim:

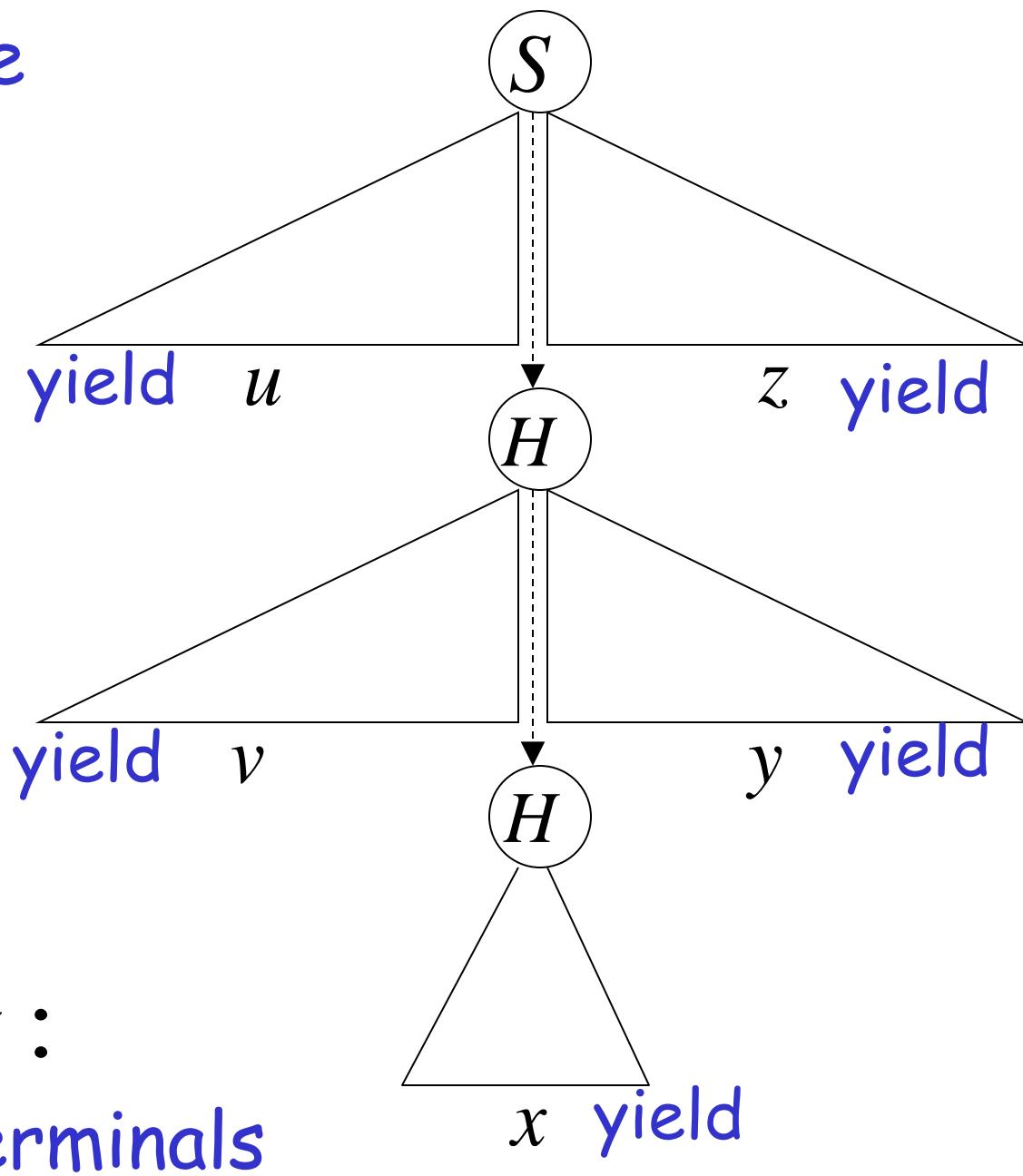
some variable H
is repeated



Take H to be the deepest, so that
only H is repeated in subtree

We can write

$$w = uvxyz$$



u, v, x, y, z :
Strings of terminals

Example:

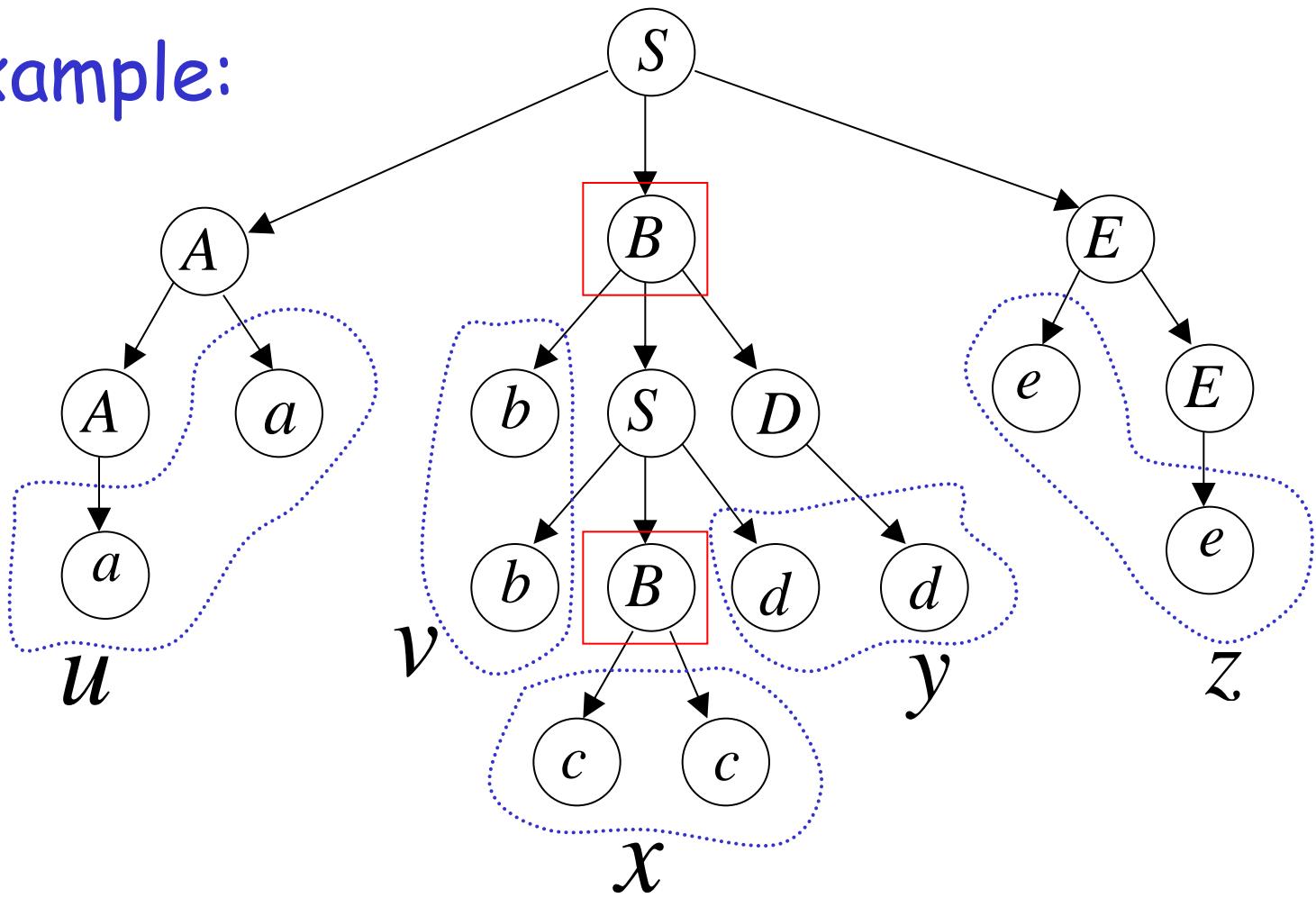
$$u = aa$$

$$v = bb$$

$$x = cc$$

$$y = dd$$

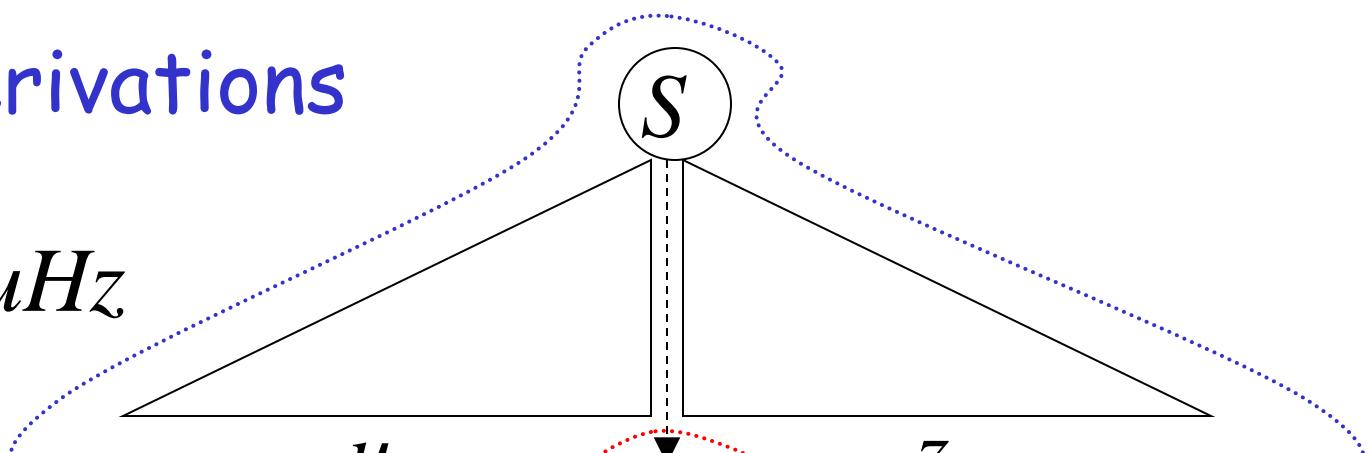
$$z = ee$$



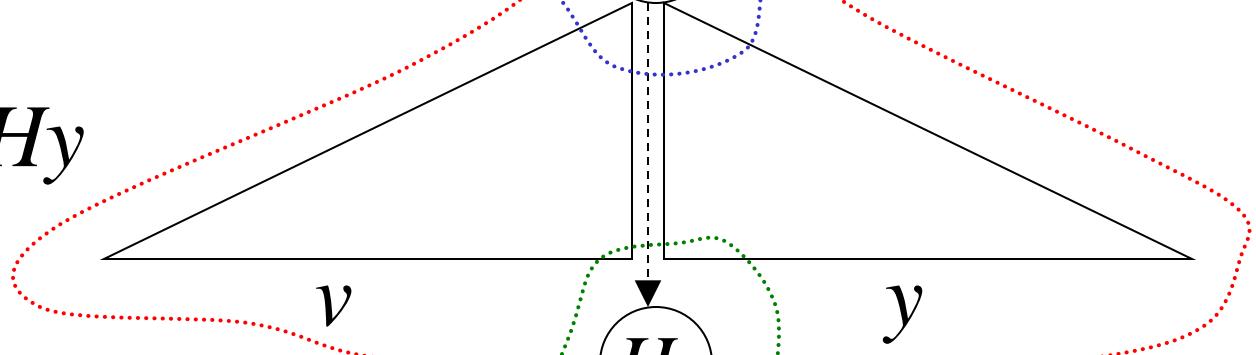
B correspond s to H

Possible derivations

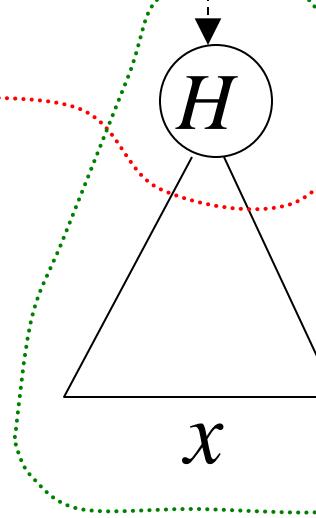
*

$$S \Rightarrow uHz$$


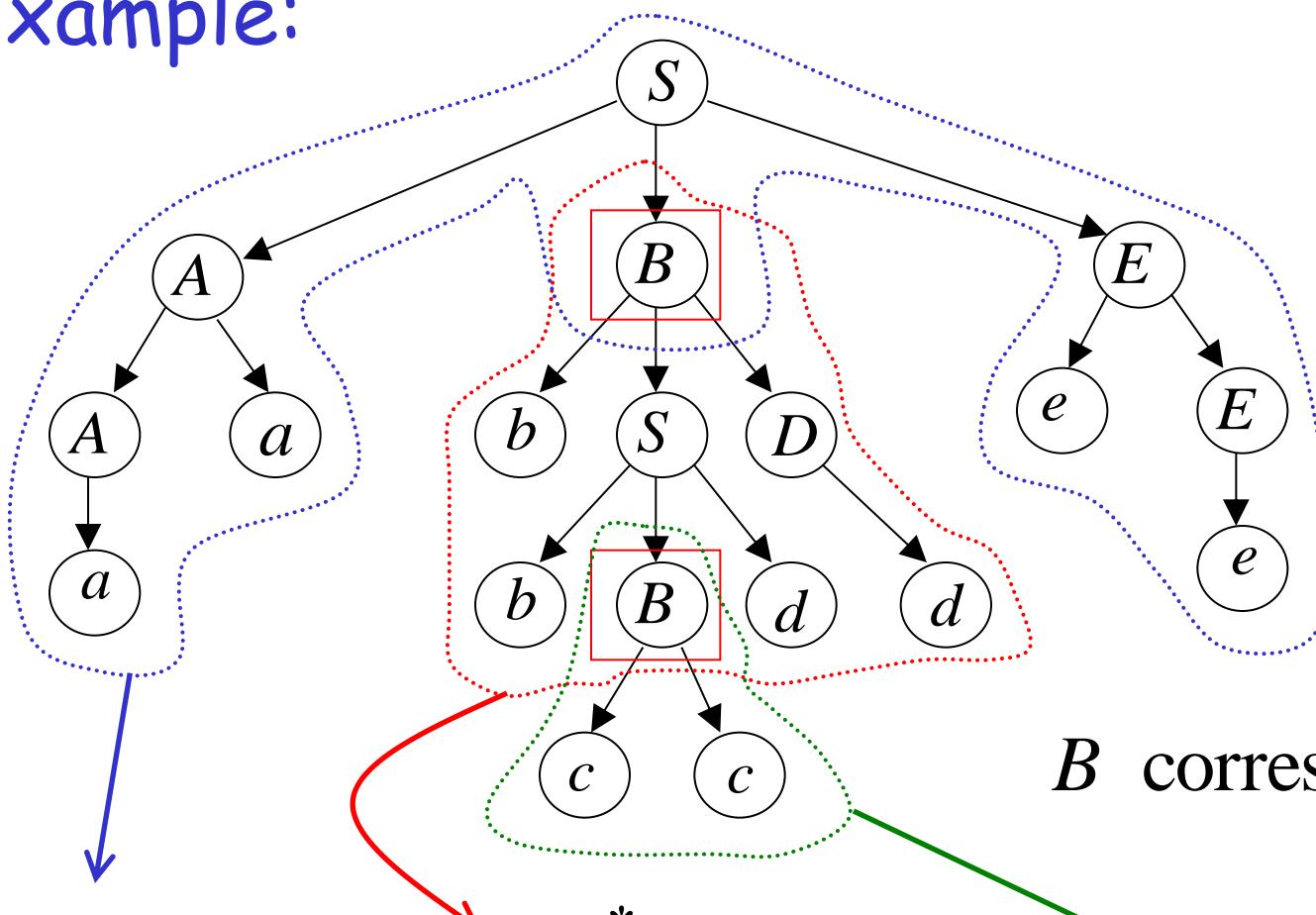
*

$$H \Rightarrow vHy$$


*

$$H \Rightarrow x$$


Example:



B correspond s to H

$$* \\ S \Rightarrow uHz$$

$$* \\ H \Rightarrow vHy$$

$$* \\ H \Rightarrow x$$

$$* \\ S \Rightarrow aaBee$$

$$* \\ B \Rightarrow bbBdd$$

$$B \Rightarrow cc$$

$$\begin{aligned} u &= aa \\ v &= bb \\ x &= cc \\ y &= dd \\ z &= ee \end{aligned}$$

Remove Middle Part

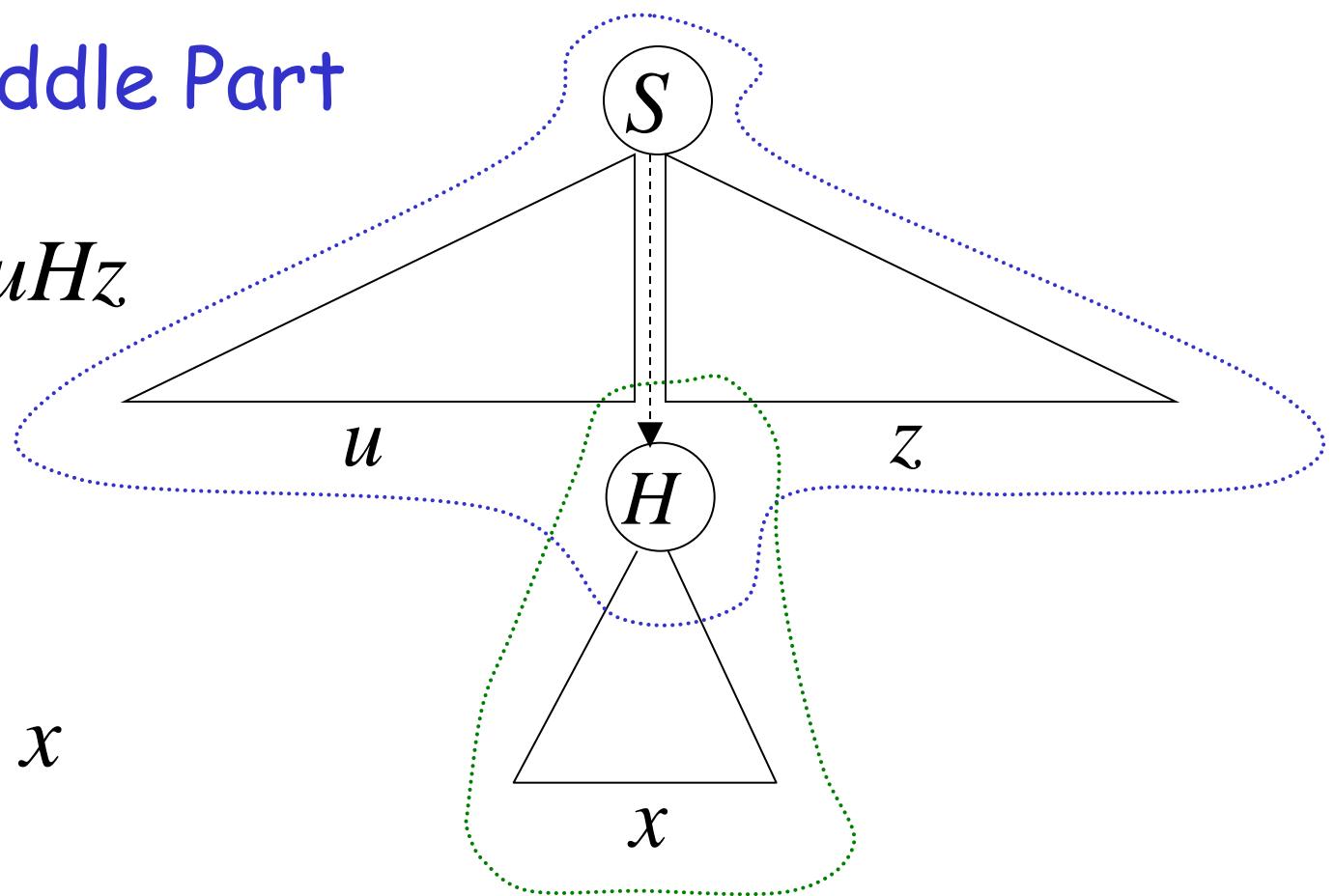
*

$$S \Rightarrow uHz$$

*

$$H \Rightarrow x$$

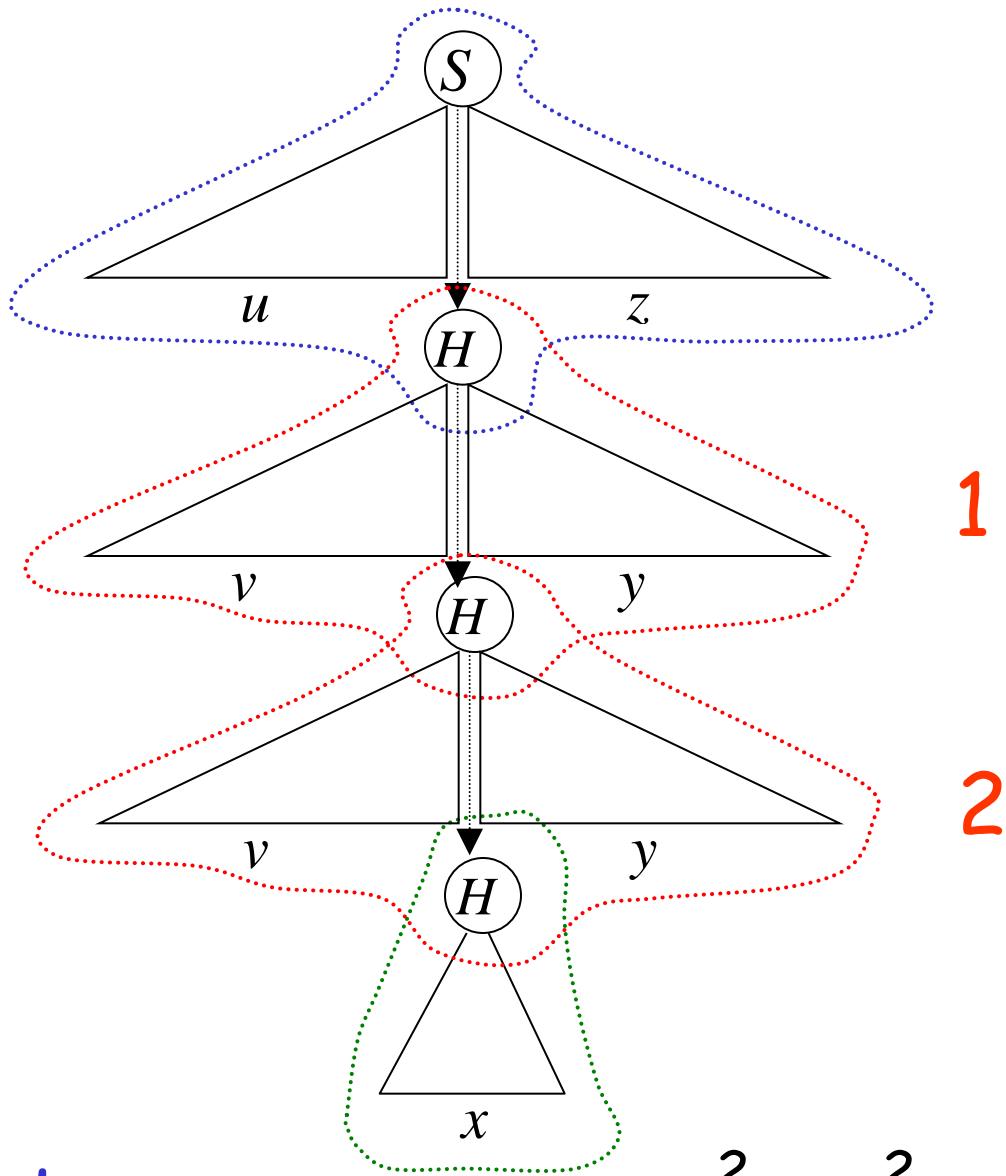
Yield: $uxz = uv^0xy^0z$

$$S \xrightarrow{*} uHz \xrightarrow{*} uxz = uv^0xy^0z \in L(G)$$


Repeat Middle part two times

*

$$S \Rightarrow uHz$$



*

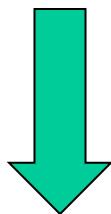
$$H \Rightarrow vHy$$

*

$$H \Rightarrow x$$

Yield: $uvvxyz = uv^2xy^2z$

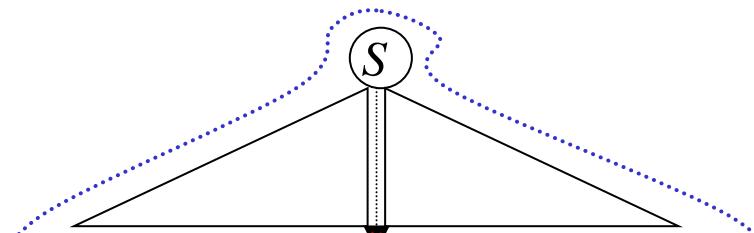
$$S \xrightarrow{*} uHz \quad H \xrightarrow{*} vHy \quad H \xrightarrow{*} x$$



$$\begin{aligned} & * & * & * \\ S \xrightarrow{*} uHz & \Rightarrow uvHyz \Rightarrow uvvHyyz \\ & * \\ \Rightarrow uvvxyz & = uv^2xy^2z \in L(G) \end{aligned}$$

Repeat Middle part i times

*

 $S \Rightarrow uHz$ 

*

 $H \Rightarrow vHy$

1

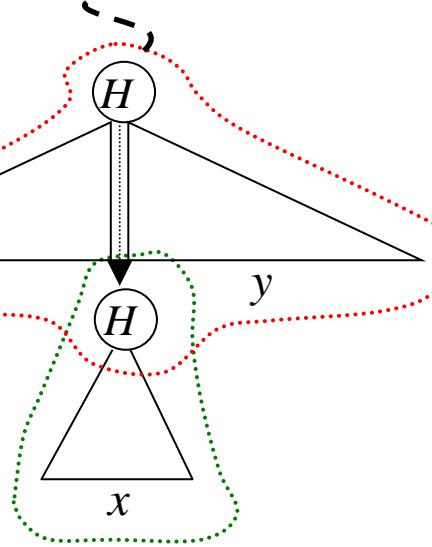
⋮

*

 $H \Rightarrow vHy$

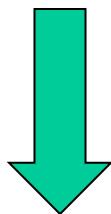
i

*

 $H \Rightarrow x$ 

Yield: $uv^i xy^i z$

* * *

$$S \xrightarrow{*} uHz$$
$$H \xrightarrow{*} vHy$$
$$H \xrightarrow{*} x$$

$$S \xrightarrow{*} uHz \xrightarrow{*} uvHyz \xrightarrow{*} uvvHyyz \xrightarrow{*} \dots$$
$$\xrightarrow{*} uv^i Hy^i z \xrightarrow{*} uv^i xy^i z \in L(G)$$

Therefore,

$$|w| \geq t^r$$

If we know that: $w = uvxyz \in L(G)$

then we also know: $uv^i xy^i z \in L(G)$

For all $i \geq 0$

since

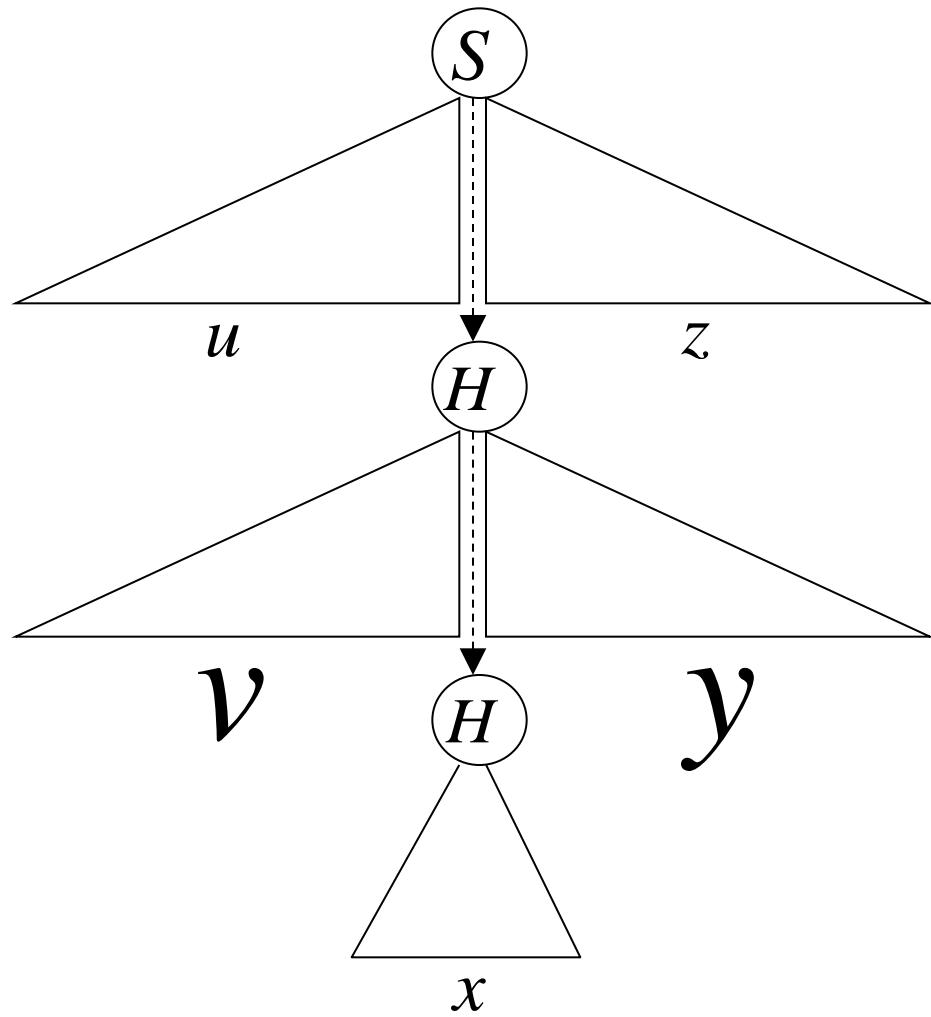
$$L(G) = L - \{\lambda\}$$

$$uv^i xy^i z \in L$$

Observation 1:

$$|vy| \geq 1$$

Since G has no unit and λ -productions

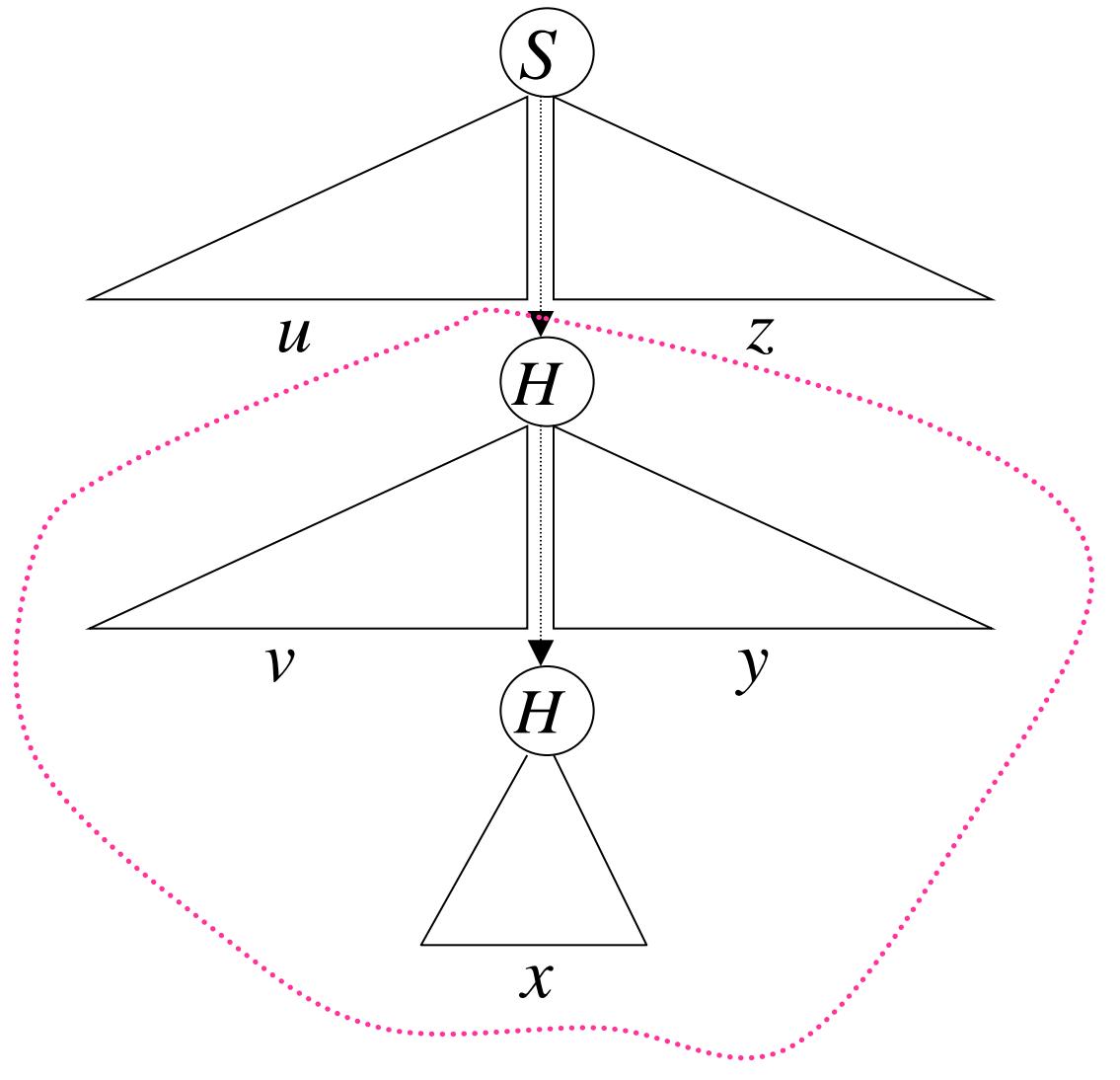


At least one of v or y is not λ

Observation 2:

$$|vxy| \leq t^{r+1}$$

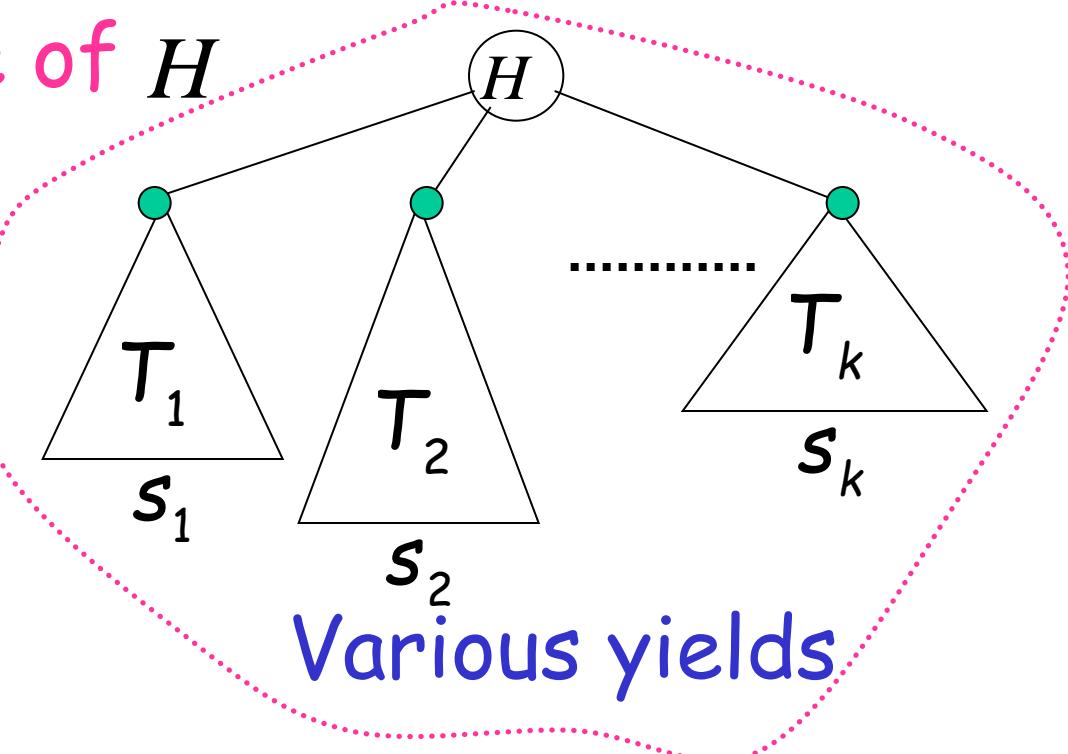
since in subtree
only variable H
is repeated



subtree of H

Explanation follows....

subtree of H



$$vxy = s_1 s_2 \cdots s_k$$

$|s_j| \leq t^r$ since no variable is repeated in T_j

$$|vxy| = \sum_{j=1}^k |s_j| \leq k \cdot t^r \leq t \cdot t^r = t^{r+1}$$

Maximum right-hand side of any production

Thus, if we choose critical length

$$m = t^{r+1} > t^r$$

then, we obtain the pumping lemma for context-free languages

The Pumping Lemma:

For any infinite context-free language L

there exists an integer m such that

for any string $w \in L$, $|w| \geq m$

we can write $w = uvxyz$

with lengths $|vxy| \leq m$ and $|vy| \geq 1$

and it must be that:

$uv^i xy^i z \in L$, for all $i \geq 0$

Applications of The Pumping Lemma

Non-context free languages

$\{a^n b^n c^n : n \geq 0\}$

Context-free languages

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

Theorem: The language

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

is **not** context free

Proof: Use the Pumping Lemma
for context-free languages

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

Assume for contradiction that L
is context-free

Since L is context-free and infinite
we can apply the pumping lemma

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

Let m be the critical length
of the pumping lemma

Pick any string $w \in L$ with length $|w| \geq m$

We pick: $w = a^m b^m c^m$

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

$$w = a^m b^m c^m$$

From pumping lemma:

we can write: $w = uvxyz$

with lengths $|vxy| \leq m$ and $|vy| \geq 1$

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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Pumping Lemma says:

$$uv^i xy^i z \in L \quad \text{for all } i \geq 0$$

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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

We examine all the possible locations
of string vxy in w

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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 1: vxy is in a^m

$m \quad m \quad m$

$a \dots aa \dots aa \dots a \ bbb \dots bbb \ ccc \dots ccc$

$u \quad vxy \quad z$

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

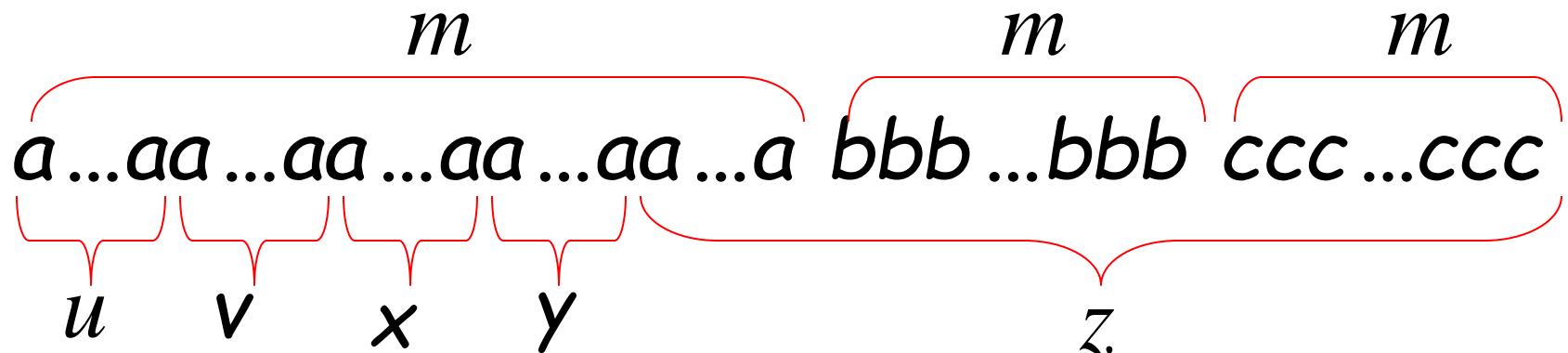
$$w = a^m b^m c^m$$

$$w = uvxyz$$

$$|vxy| \leq m$$

$$|vy| \geq 1$$

$$v = a^{k_1} \quad y = a^{k_2} \quad k_1 + k_2 \geq 1$$

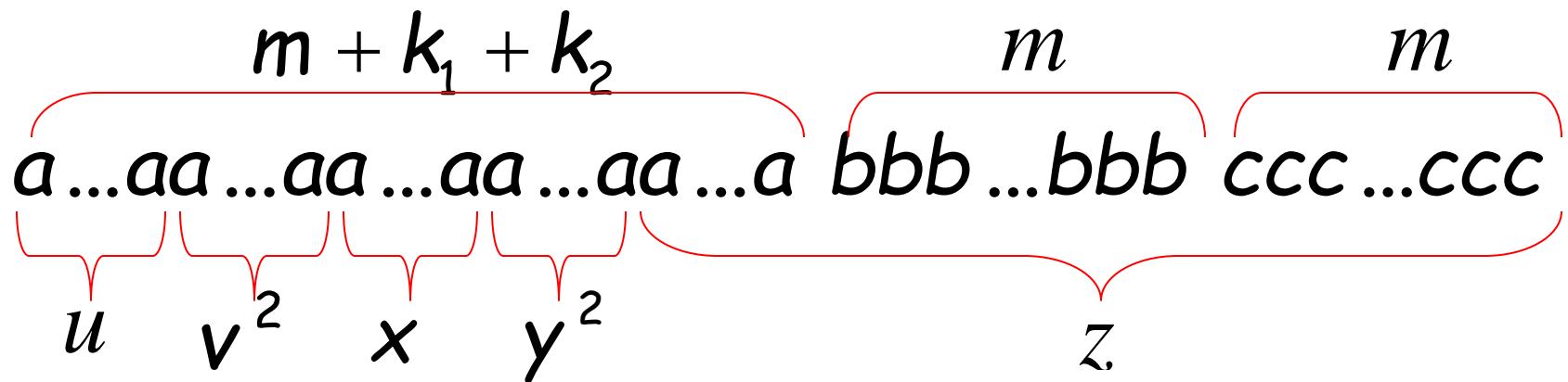


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

$$v = a^{k_1} \quad y = a^{k_2} \quad k_1 + k_2 \geq 1$$



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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

From Pumping Lemma: $uv^2xy^2z \in L$

$$k_1 + k_2 \geq 1$$

However: $uv^2xy^2z = a^{m+k_1+k_2}b^m c^m \notin L$

Contradiction!!!

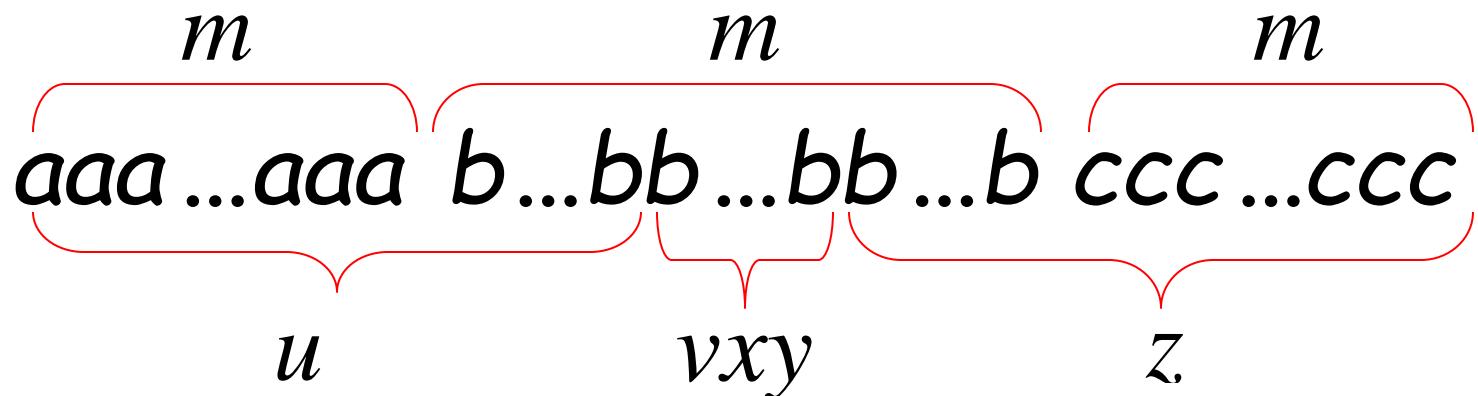
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 2: vxy is in b^m

Similar to case 1



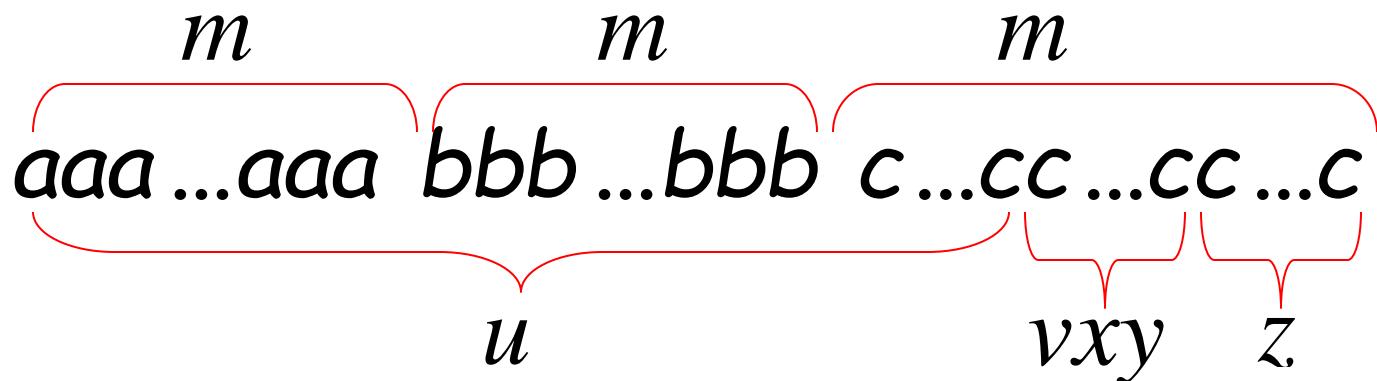
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 3: vxy is in c^m

Similar to case 1

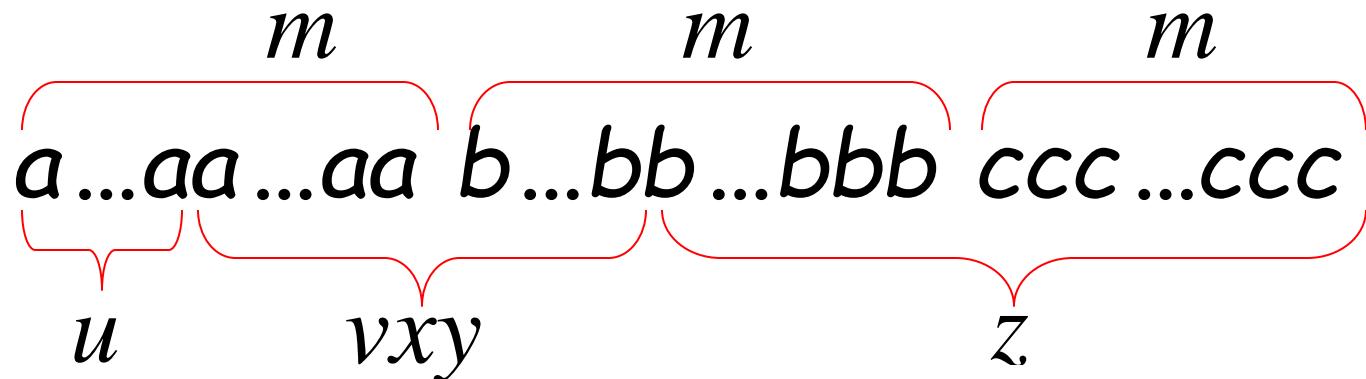


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 4: vxy overlaps a^m and b^m

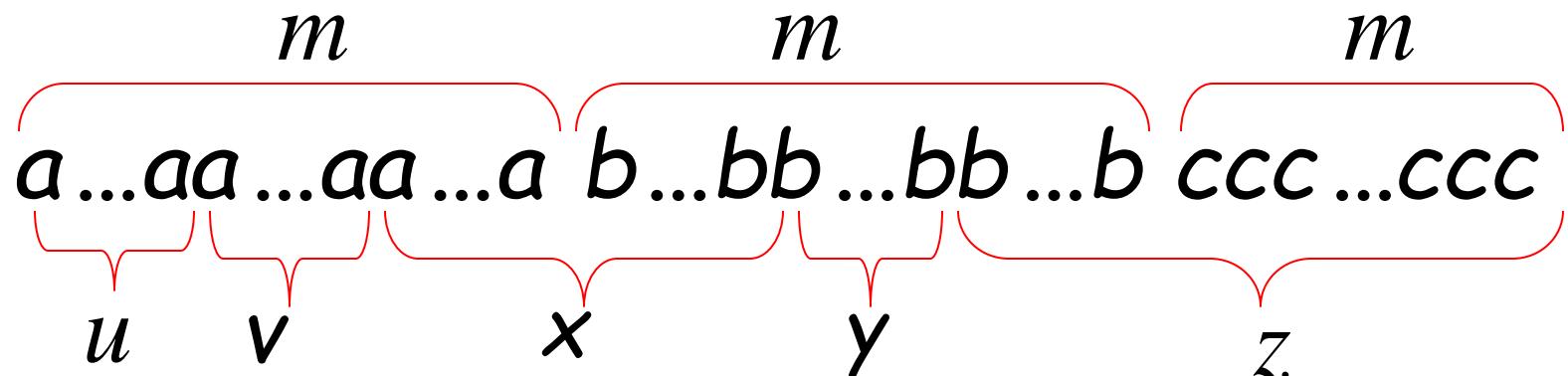


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Sub-case 1: v contains only a
 y contains only b



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

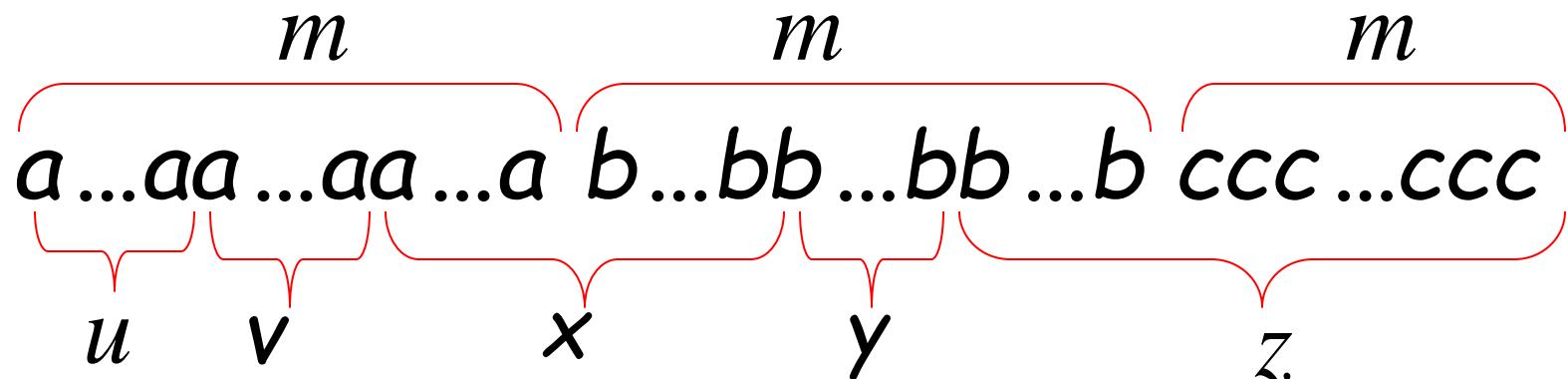
$$w = a^m b^m c^m$$

$$w = uvxyz$$

$$|vxy| \leq m$$

$$|vy| \geq 1$$

$$v = a^{k_1} \quad y = a^{k_2} \quad k_1 + k_2 \geq 1$$

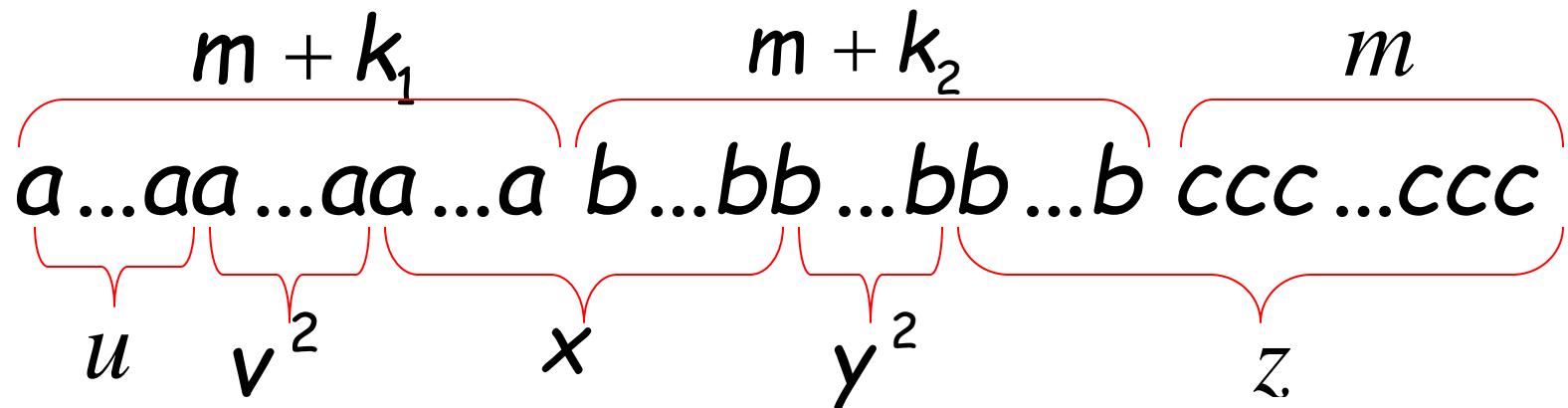


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

$$v = a^{k_1} \quad y = a^{k_2} \quad k_1 + k_2 \geq 1$$



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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

From Pumping Lemma: $uv^2xy^2z \in L$

$$k_1 + k_2 \geq 1$$

However: $uv^2xy^2z = a^{m+k_1}b^{m+k_2}c^m \notin L$

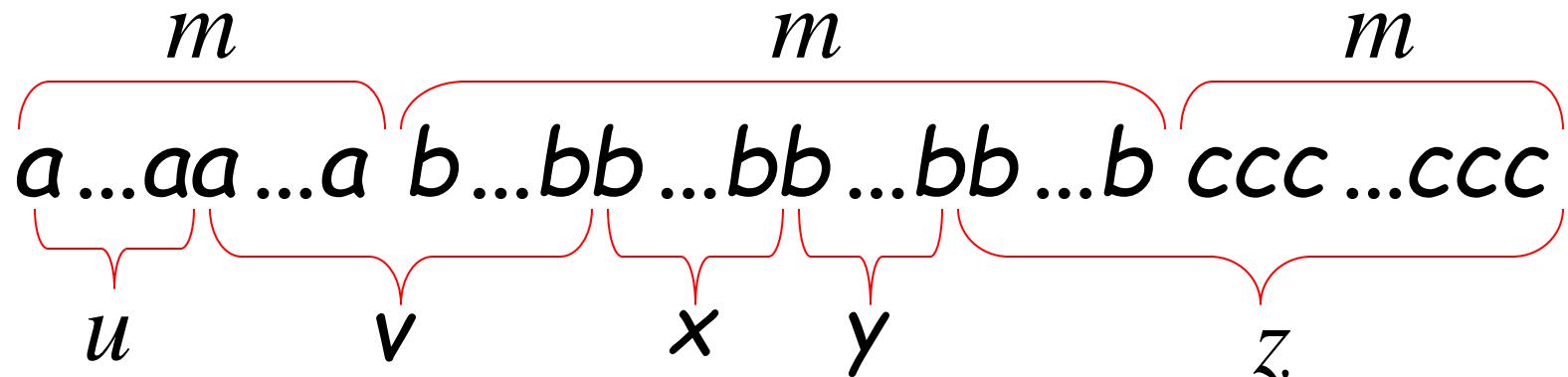
Contradiction!!!

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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Sub-case 2: v contains a and b
 y contains only b



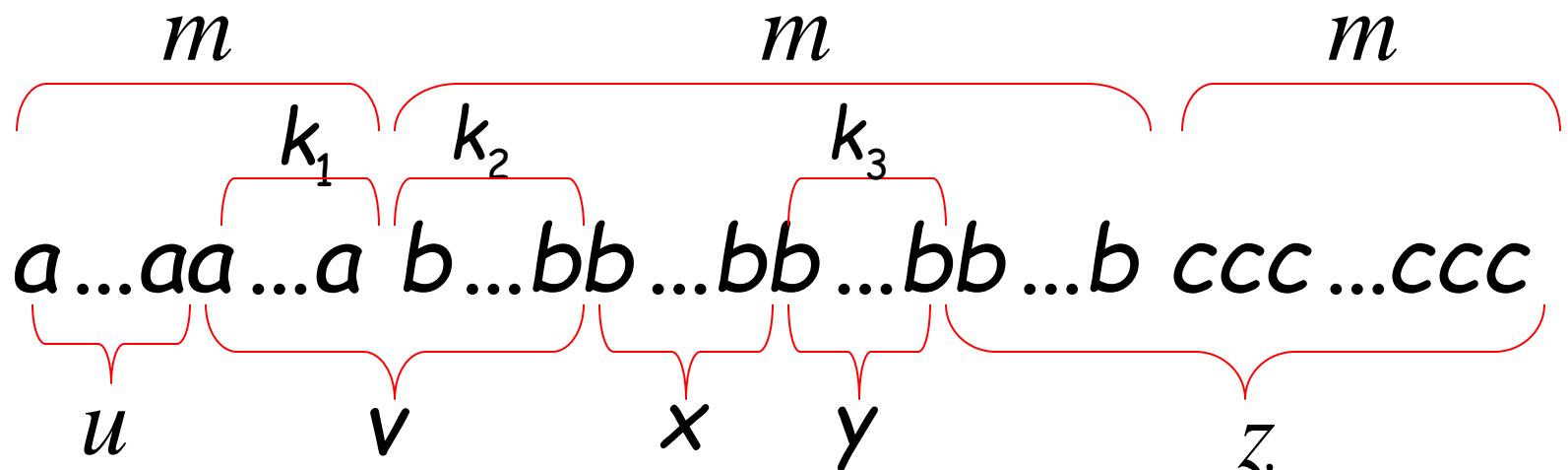
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

By assumption

$$v = a^{k_1} b^{k_2} \quad y = a^{k_3} \quad k_1, k_2 \geq 1$$

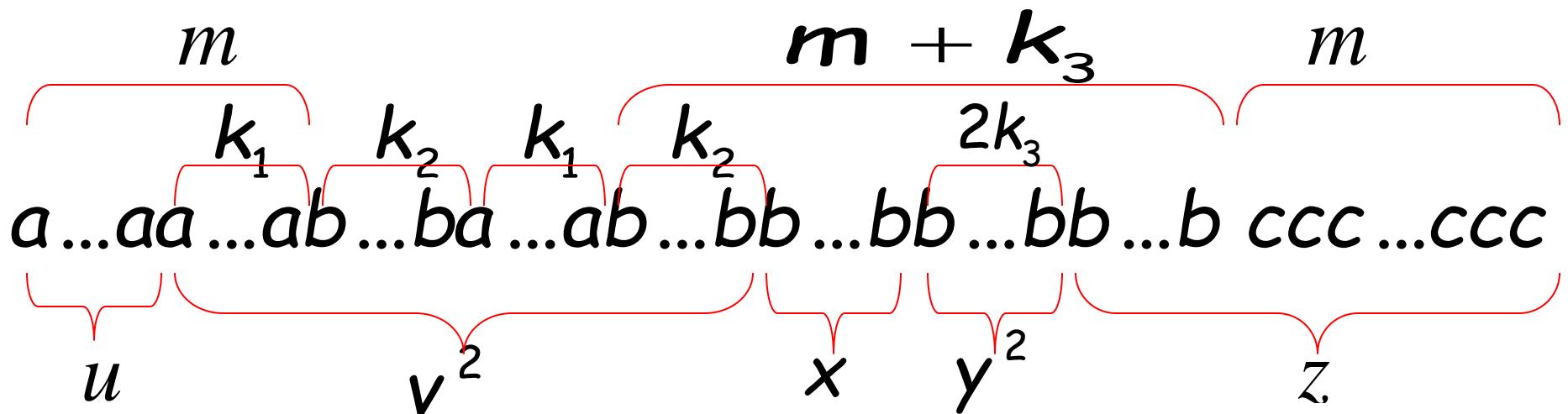


$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

$$v = a^{k_1} b^{k_2} \quad y = a^{k_3} \quad k_1, k_2 \geq 1$$



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

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

From Pumping Lemma: $uv^2xy^2z \in L$

$$k_1, k_2 \geq 1$$

However: $uv^2xy^2z = a^m b^{k_2} a^{k_1} b^{m+k_3} c^m \notin L$

Contradiction!!!

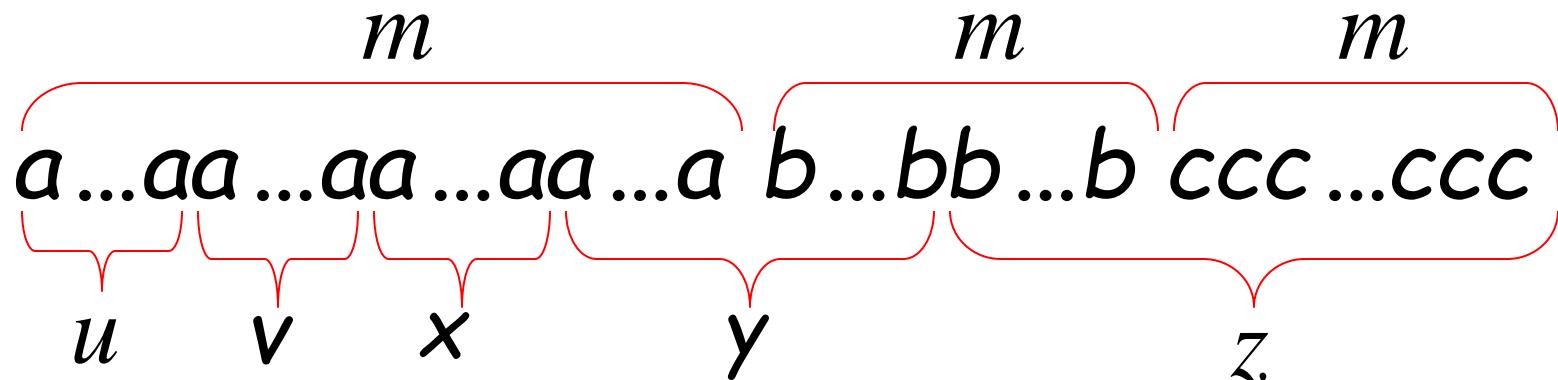
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Sub-case 3: v contains only a
 y contains a and b

Similar to sub-case 2



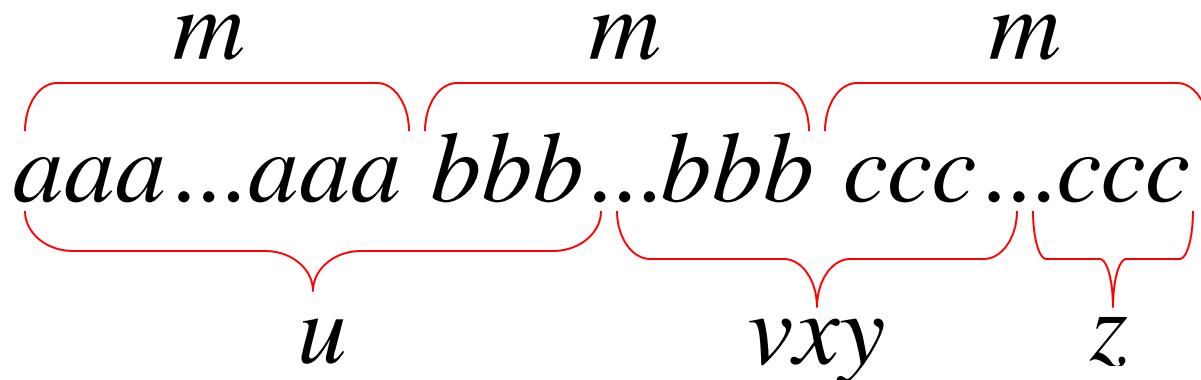
$$L = \{a^n b^n c^n : n \geq 0\}$$

$$w = a^m b^m c^m$$

$$w = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 5: vxy overlaps b^m and c^m

Similar to case 4



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

$$w = a^m b^m c^m$$

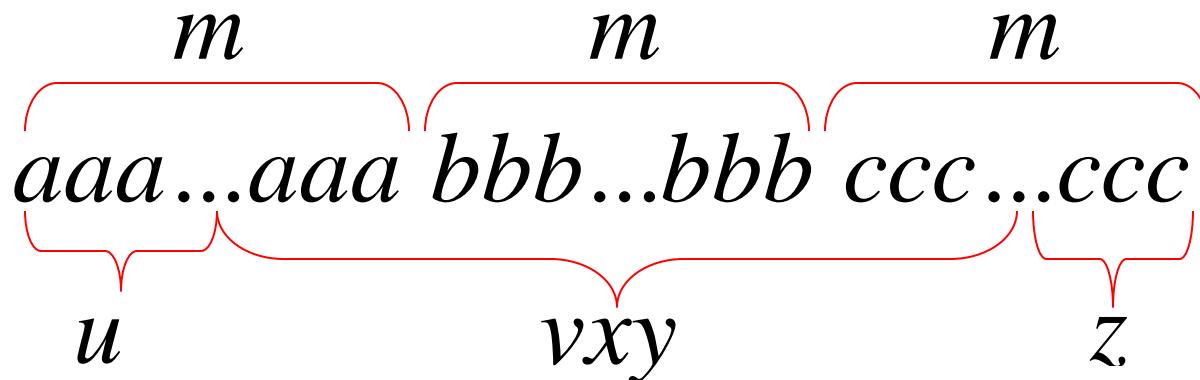
$$w = uvxyz$$

$$|vxy| \leq m$$

$$|vy| \geq 1$$

Case 6: vxy overlaps a^m , b^m and c^m

Impossible!



In all cases we obtained a contradiction

Therefore: the original assumption that

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

is context-free must be wrong

Conclusion: L is not context-free

More Applications of The Pumping Lemma

The Pumping Lemma:

For infinite context-free language L

there exists an integer m such that

for any string $w \in L$, $|w| \geq m$

we can write $w = uvxyz$

with lengths $|vxy| \leq m$ and $|vy| \geq 1$

and it must be:

$$uv^i xy^i z \in L, \quad \text{for all } i \geq 0$$

Non-context free languages

$\{a^n b^n c^n : n \geq 0\}$

$\{vv : v \in \{a,b\}^*\}$

Context-free languages

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

$\{ww^R : w \in \{a,b\}^*\}$

Theorem: The language

$$L = \{vv : v \in \{a,b\}^*\}$$

is not context free

Proof: Use the Pumping Lemma
for context-free languages

$$L = \{vv : v \in \{a,b\}^*\}$$

Assume for contradiction that L
is context-free

Since L is context-free and infinite
we can apply the pumping lemma

$$L = \{vv : v \in \{a,b\}^*\}$$

Pumping Lemma gives a magic number m such that:

Pick any string of L with length at least m

we pick: $a^m b^m a^m b^m \in L$

$$L = \{vv : v \in \{a,b\}^*\}$$

We can write: $a^m b^m a^m b^m = uvxyz$

with lengths $|vxy| \leq m$ and $|vy| \geq 1$

Pumping Lemma says:

$uv^i xy^i z \in L$ for all $i \geq 0$

$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

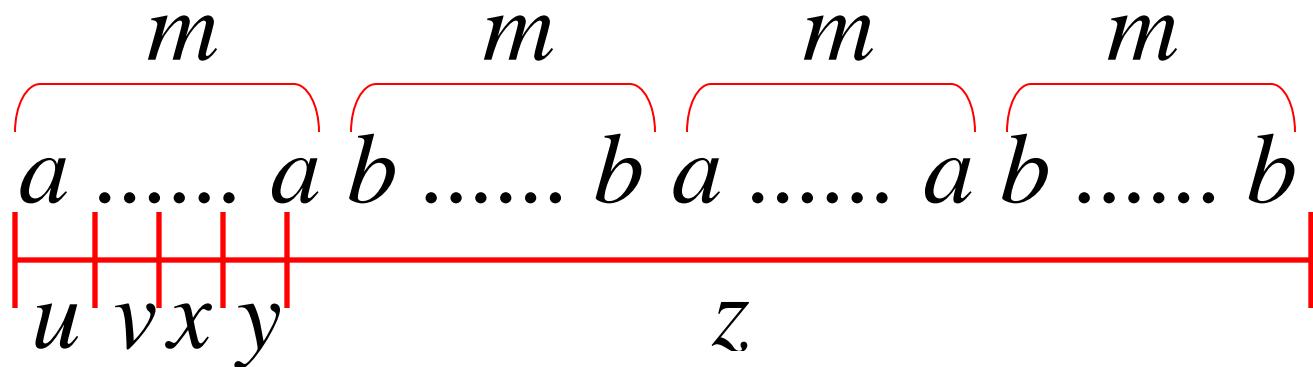
We examine all the possible locations
of string vxy in $a^m b^m a^m b^m$

$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 1: vxy is within the first a^m

$$v = a^{k_1} \quad y = a^{k_2} \quad k_1 + k_2 \geq 1$$

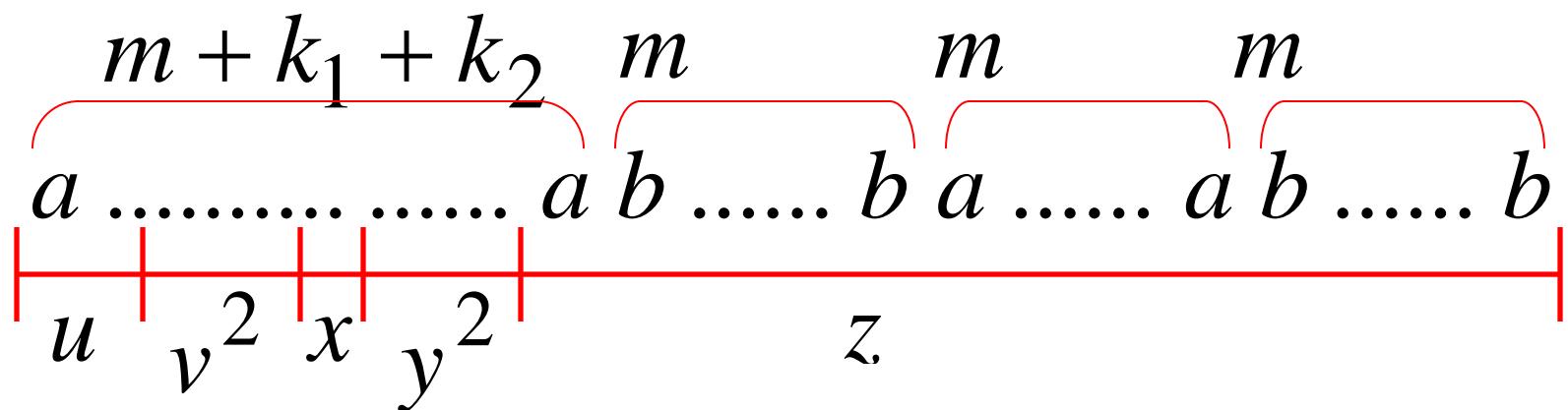


$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 1: vxy is within the first a^m

$$v = a^{k_1} \quad y = a^{k_2} \quad k_1 + k_2 \geq 1$$



$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 1: vxy is within the first a^m

$$a^{m+k_1+k_2} b^m a^m b^m = u v^2 x y^2 z \notin L$$

$$k_1 + k_2 \geq 1$$

$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 1: vxy is within the first a^m

$$a^{m+k_1+k_2} b^m a^m b^m = uv^2 xy^2 z \notin L$$

However, from Pumping Lemma: $uv^2 xy^2 z \in L$

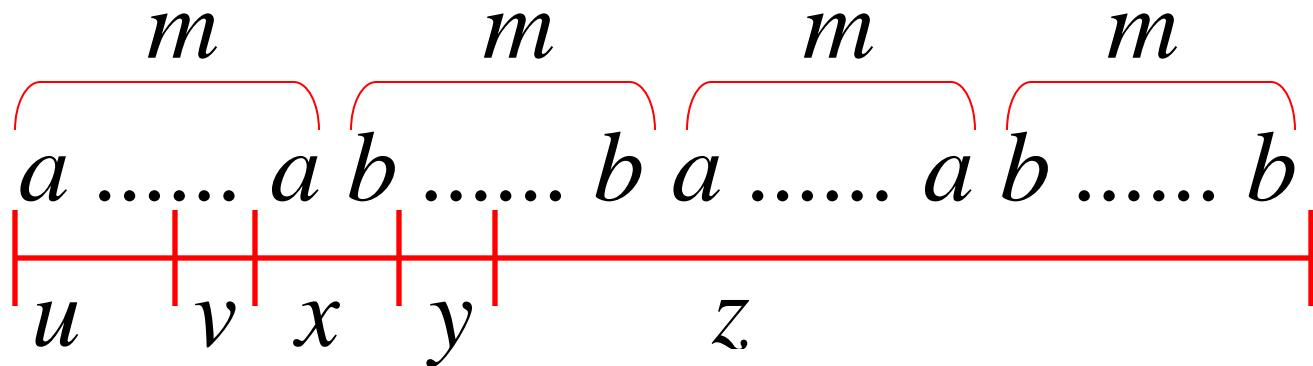
Contradiction!!!

$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 2: v is in the first a^m
 y is in the first b^m

$$v = a^{k_1} \quad y = b^{k_2} \quad k_1 + k_2 \geq 1$$

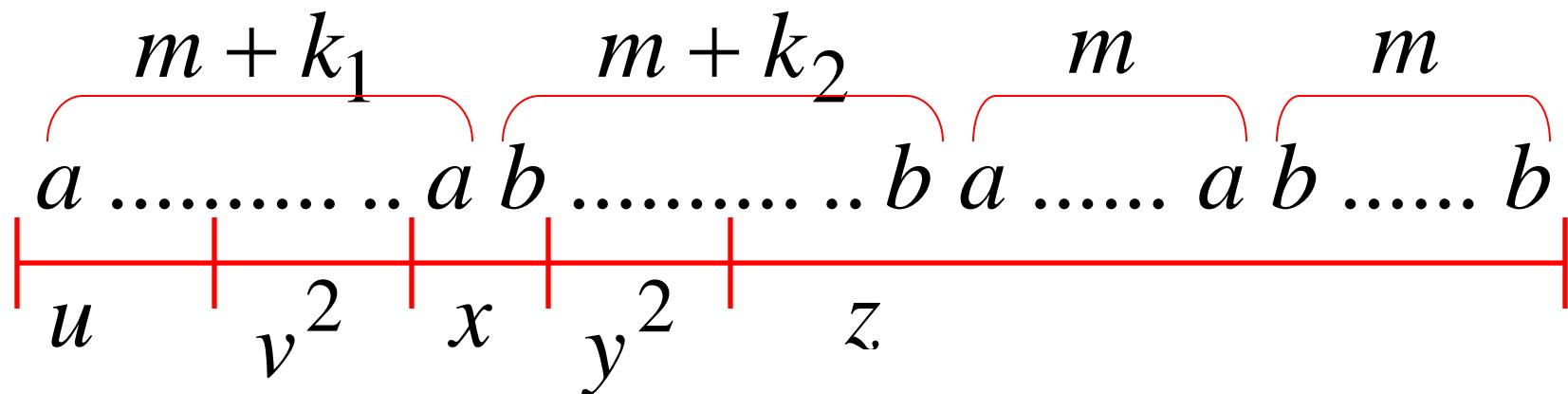


$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 2: v is in the first a^m
 y is in the first b^m

$$v = a^{k_1} \quad y = b^{k_2} \quad k_1 + k_2 \geq 1$$



$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 2: v is in the first a^m
 y is in the first b^m

$$a^{m+k_1} b^{m+k_2} a^m b^m = uv^2 xy^2 z \notin L$$

$$k_1 + k_2 \geq 1$$

$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 2: v is in the first a^m
 y is in the first b^m

$$a^{m+k_1} b^{m+k_2} a^m b^m = uv^2 xy^2 z \notin L$$

However, from Pumping Lemma: $uv^2 xy^2 z \in L$

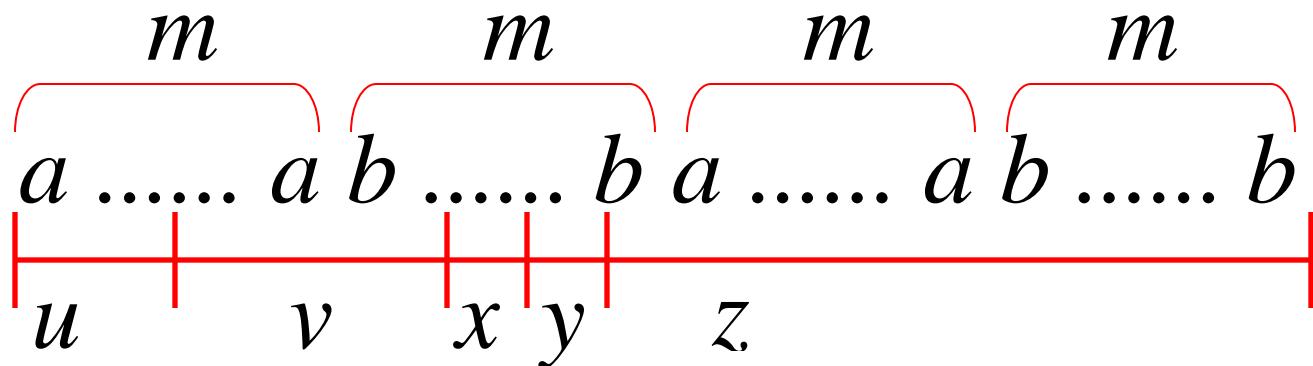
Contradiction!!!

$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 3: v overlaps the first $a^m b^m$
 y is in the first b^m

$$v = a^{k_1} b^{k_2} \quad y = b^{k_3} \quad k_1, k_2 \geq 1$$

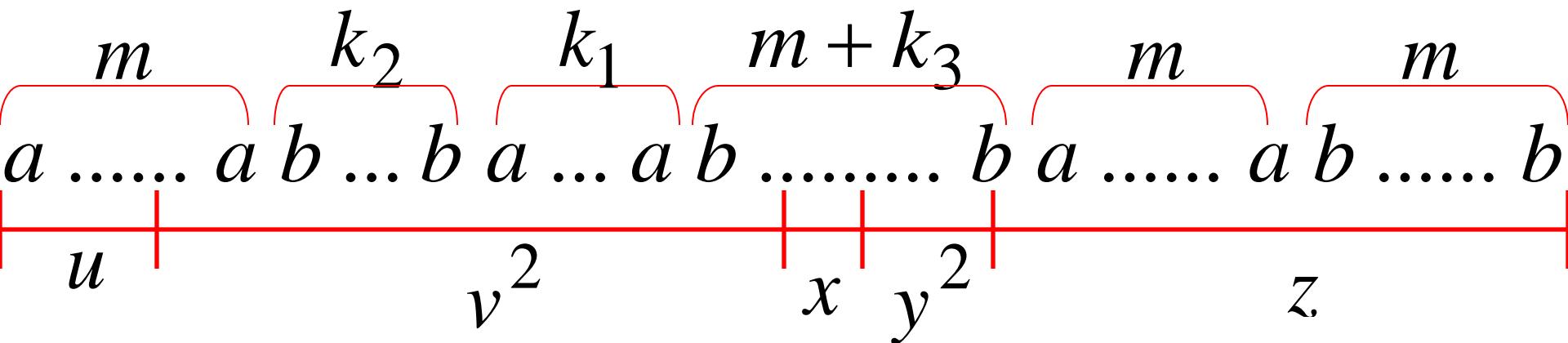


$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 3: v overlaps the first $a^m b^m$
 y is in the first b^m

$$v = a^{k_1} b^{k_2} \quad y = b^{k_3} \quad k_1, k_2 \geq 1$$



$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 3: v overlaps the first $a^m b^m$
 y is in the first b^m

$$a^m b^{k_2} a^{k_1} b^{m+k_3} a^m b^m = uv^2 xy^2 z \notin L$$

$$k_1, k_2 \geq 1$$

$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 3: v overlaps the first $a^m b^m$
 y is in the first b^m

$$a^m b^{k_2} a^{k_1} b^{k_3} a^m b^m = uv^2 xy^2 z \notin L$$

However, from Pumping Lemma: $uv^2 xy^2 z \in L$

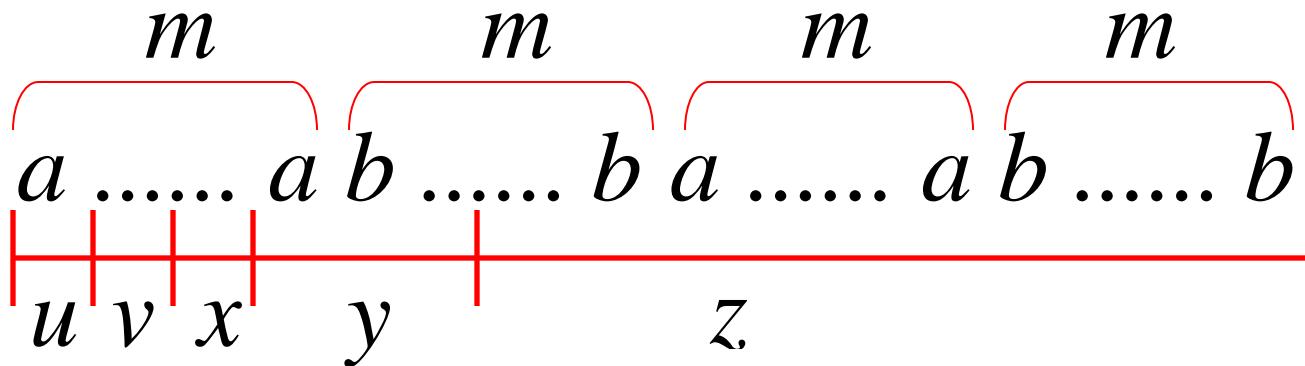
Contradiction!!!

$$L = \{vv : v \in \{a,b\}^*\}$$

$$a^m b^m a^m b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Case 4: v in the first a^m
 y Overlaps the first $a^m b^m$

Analysis is similar to case 3



Other cases: vxy is within $a^m b^m a^m b^m$

or

$a^m b^m a^m b^m$

or

$a^m b^m a^m b^m$

Analysis is similar to case 1:

$a^m b^m a^m b^m$

More cases:

vxy

overlaps

$a^m b^m a^m b^m$

or

$a^m b^m a^m b^m$

Analysis is similar to cases 2,3,4:

$a^m b^m a^m b^m$

There are no other cases to consider

Since $|vxy| \leq m$, it is impossible
 vxy to overlap:

$$a^m b^m a^m b^m$$

nor

$$a^m b^m a^m b^m$$

nor

$$a^m b^m a^m b^m$$

In all cases we obtained a contradiction

Therefore: The original assumption that

$$L = \{vv : v \in \{a,b\}^*\}$$

is context-free must be wrong

Conclusion: L is not context-free

Non-context free languages

$\{a^n b^n c^n : n \geq 0\}$

$\{ww : w \in \{a,b\}^*\}$

$\{a^{n!} : n \geq 0\}$

Context-free languages

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

$\{ww^R : w \in \{a,b\}^*\}$

Theorem: The language

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

is **not** context free

Proof:

Use the Pumping Lemma
for context-free languages

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

Assume for contradiction that L
is context-free

Since L is context-free and infinite
we can apply the pumping lemma

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

Pumping Lemma gives a magic number m such that:

Pick any string of L with length at least m

we pick: $a^{m!} \in L$

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

We can write: $a^{m!} = uvxyz$

with lengths $|vxy| \leq m$ and $|vy| \geq 1$

Pumping Lemma says:

$uv^i xy^i z \in L$ for all $i \geq 0$

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

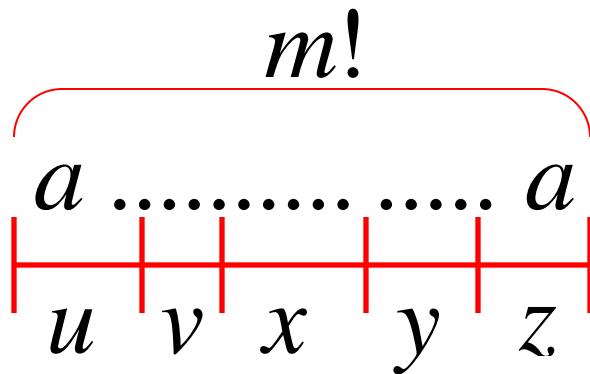
$$a^{m!} = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

We examine all the possible locations
of string vxy in $a^{m!}$

There is only one case to consider

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

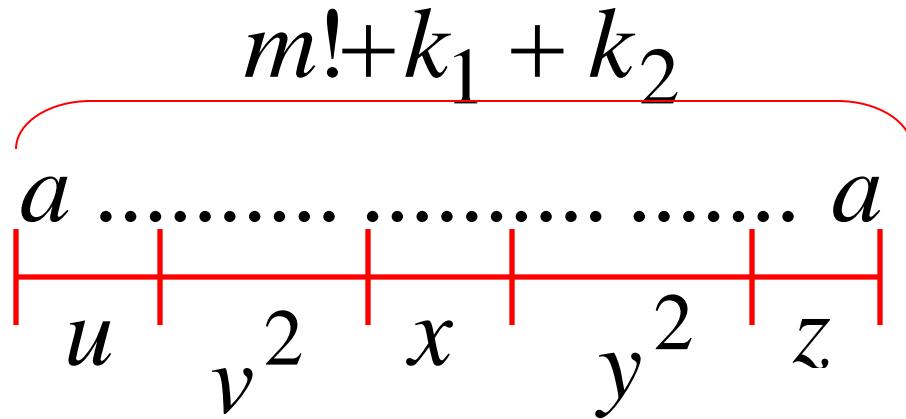
$$a^{m!} = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$



$$v = a^{k_1} \quad y = a^{k_2} \quad 1 \leq k_1 + k_2 \leq m$$

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

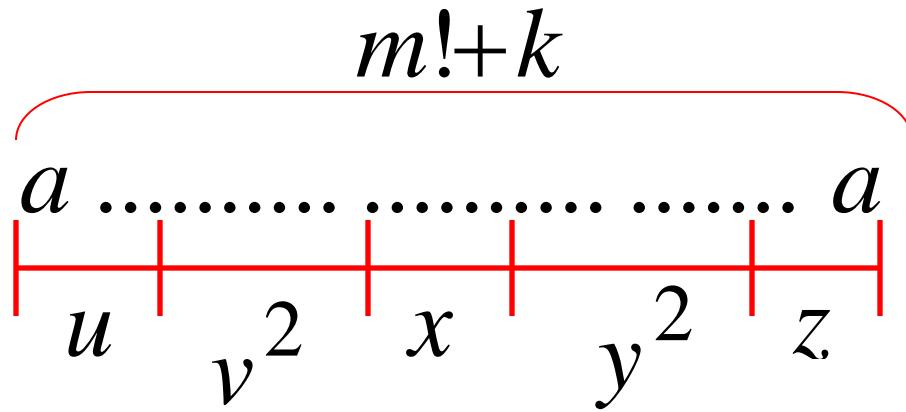
$$a^{m!} = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$



$$v = a^{k_1} \quad y = a^{k_2} \quad 1 \leq k_1 + k_2 \leq m$$

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

$$a^{m!} = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$



$$v = a^{k_1} \quad y = a^{k_2} \quad 1 \leq k \leq m$$

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

$$a^{m!}=uvxyz \hspace{1.5cm} |vxy|\leq m \hspace{1cm} |vy|\geq 1$$

$$a^{m!+k}=uv^2xy^2z$$

$$\frac{\partial}{\partial x} \phi(x) = \frac{\partial}{\partial x} \phi(x_0)$$

$$a^{m!+k}=uv^2xy^2z$$

$$(\mathbb{R}^d)^{\otimes k}$$

$$a^{m!+k}=uv^2xy^2z$$

$$1\leq k\leq m$$

$$(\mathbb{R}^d)^{\otimes k}$$

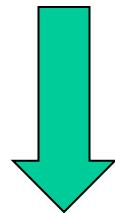
Since $1 \leq k \leq m$, for $m \geq 2$ we have:

$$m!+k \leq m!+m$$

$$< m!+m!m$$

$$= m!(1+m)$$

$$= (m+1)!$$

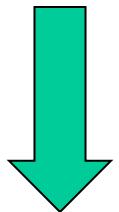


$$m! < m!+k < (m+1)!$$

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

$$a^{m!} = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

$$m! < m! + k < (m+1)!$$



$$a^{m!+k} = uv^2xy^2z \notin L$$

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

$$a^{m!} = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

However, from Pumping Lemma: $uv^2xy^2z \in L$

$$a^{m!+k} = uv^2xy^2z \notin L$$

Contradiction!!!

We obtained a contradiction

Therefore: The original assumption that

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

is context-free must be wrong

Conclusion: L is not context-free

Non-context free languages

$\{a^n b^n c^n : n \geq 0\}$

$\{ww : w \in \{a,b\}^*\}$

$\{a^{n^2} b^n : n \geq 0\}$

$\{a^{n!} : n \geq 0\}$

Context-free languages

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

$\{ww^R : w \in \{a,b\}^*\}$

Theorem: The language

$$L = \{a^{n^2} b^n : n \geq 0\}$$

is **not** context free

Proof:

Use the Pumping Lemma
for context-free languages

$$L = \{a^{n^2} b^n : n \geq 0\}$$

Assume for contradiction that L
is context-free

Since L is context-free and infinite
we can apply the pumping lemma

$$L = \{a^{n^2} b^n : n \geq 0\}$$

Pumping Lemma gives a magic number m such that:

Pick any string of L with length at least m

we pick: $a^{m^2} b^m \in L$

$$L = \{a^{n^2} b^n : n \geq 0\}$$

We can write: $a^{m^2} b^m = uvxyz$

with lengths $|vxy| \leq m$ and $|vy| \geq 1$

Pumping Lemma says:

$uv^i xy^i z \in L$ for all $i \geq 0$

$$L = \{a^{n^2} b^n : n \geq 0\}$$

$$a^{m^2} b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

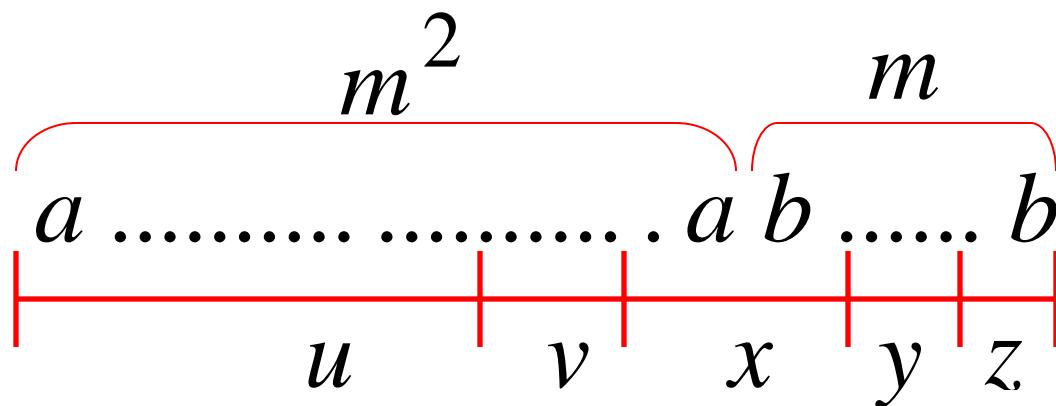
We examine all the possible locations

of string vxy in $a^{m^2} b^m$

$$L = \{a^{n^2} b^n : n \geq 0\}$$

$$a^{m^2} b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

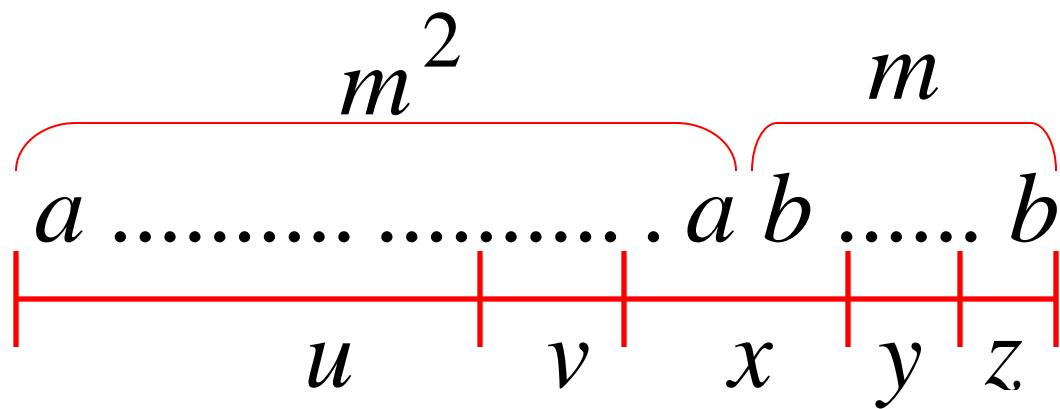
Most complicated case: v is in a^m
 y is in b^m



$$L = \{a^{n^2} b^n : n \geq 0\}$$

$$a^{m^2} b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

$$v = a^{k_1} \quad y = b^{k_2} \quad 1 \leq k_1 + k_2 \leq m$$

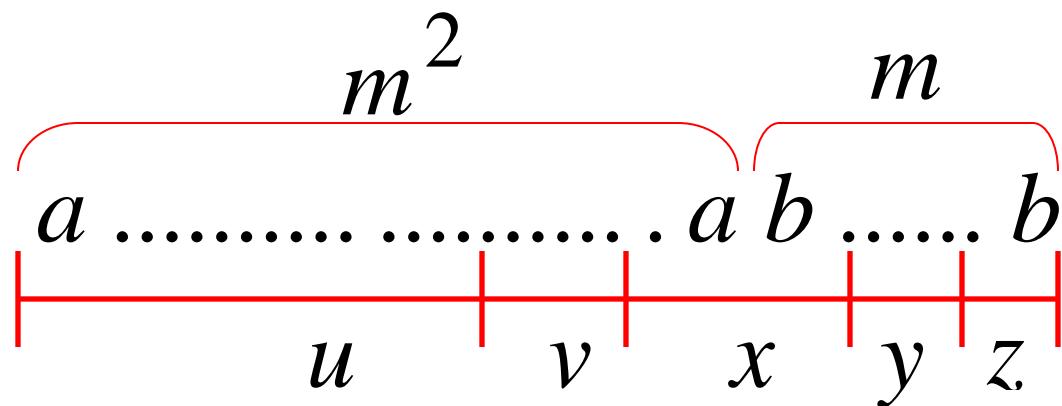


$$L = \{a^{n^2} b^n : n \geq 0\}$$

$$a^{m^2} b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Most complicated sub-case: $k_1 \neq 0$ and $k_2 \neq 0$

$$v = a^{k_1} \quad y = b^{k_2} \quad 1 \leq k_1 + k_2 \leq m$$

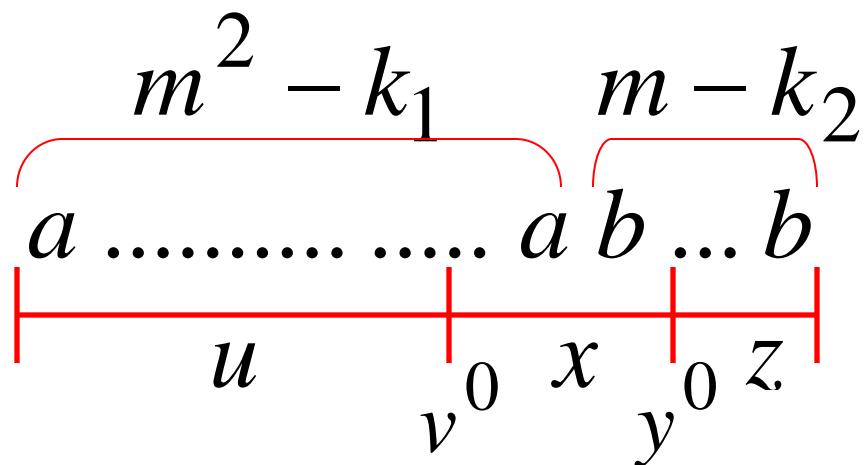


$$L = \{a^{n^2} b^n : n \geq 0\}$$

$$a^{m^2} b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Most complicated sub-case: $k_1 \neq 0$ and $k_2 \neq 0$

$$v = a^{k_1} \quad y = b^{k_2} \quad 1 \leq k_1 + k_2 \leq m$$



$$L = \{a^{n^2} b^n : n \geq 0\}$$

$$a^{m^2} b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

Most complicated sub-case: $k_1 \neq 0$ and $k_2 \neq 0$

$$v = a^{k_1} \quad y = b^{k_2} \quad 1 \leq k_1 + k_2 \leq m$$

$$a^{m^2 - k_1} b^{m - k_2} = uv^0 xy^0 z$$

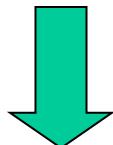
$$k_1 \neq 0 \text{ and } k_2 \neq 0 \quad 1 \leq k_1 + k_2 \leq m$$



$$(m - k_2)^2 \leq (m - 1)^2$$

$$= m^2 - 2m + 1$$

$$< m^2 - k_1$$

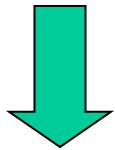


$$m^2 - k_1 \neq (m - k_2)^2$$

$$L = \{a^{n^2} b^n : n \geq 0\}$$

$$a^{m^2} b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

$$m^2 - k_1 \neq (m - k_2)^2$$



$$a^{m^2 - k_1} b^{m - k_2} = uv^0 xy^0 z \notin L$$

$$L = \{a^{n^2} b^n : n \geq 0\}$$

$$a^{m^2} b^m = uvxyz \quad |vxy| \leq m \quad |vy| \geq 1$$

However, from Pumping Lemma: $uv^0xy^0z \in L$

$$a^{m^2-k_1}b^{m-k_2} = uv^0xy^0z \notin L$$

Contradiction!!!

When we examine the rest of the cases
we also obtain a contradiction

In all cases we obtained a contradiction

Therefore: The original assumption that

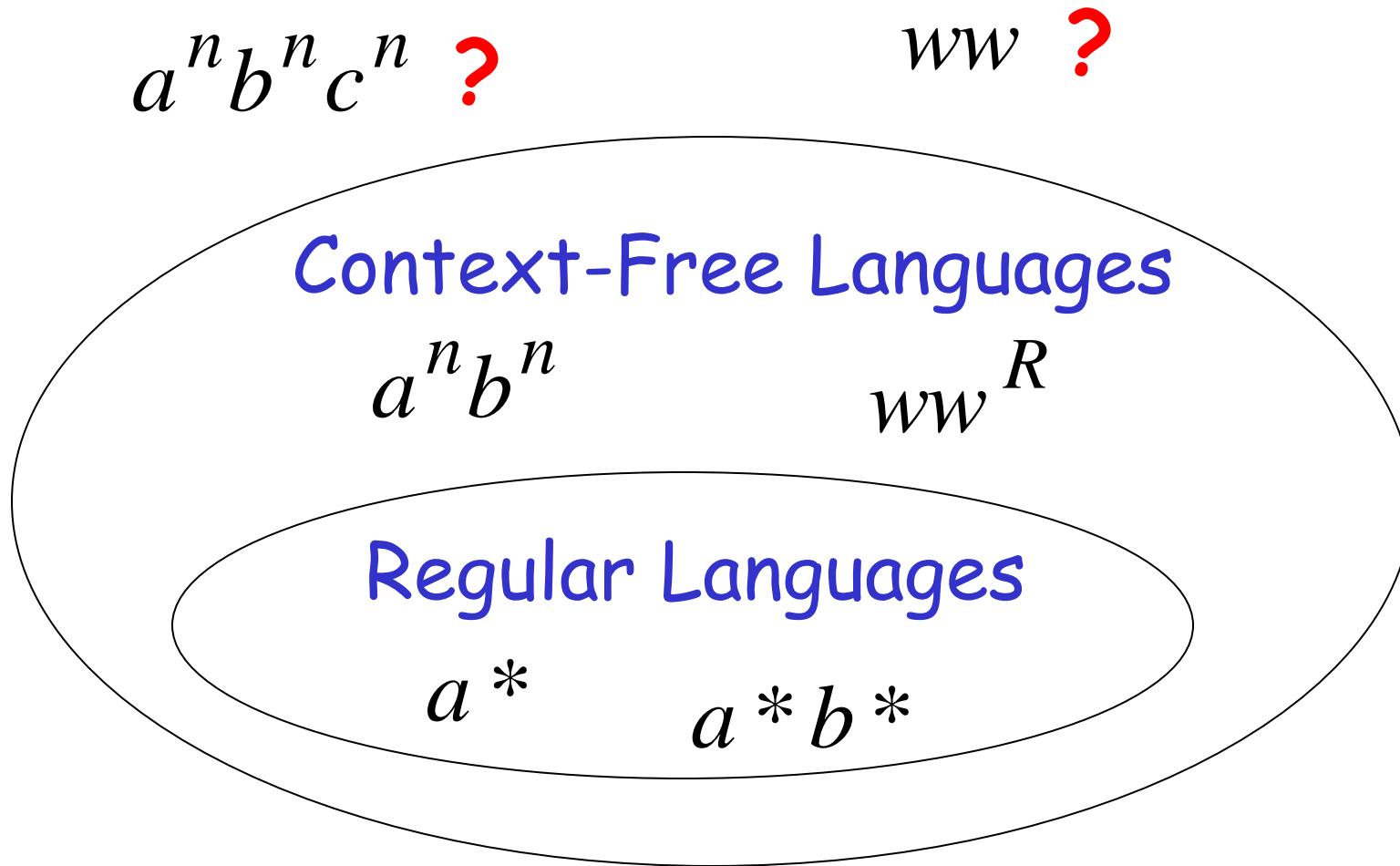
$$L = \{a^{n^2} b^n : n \geq 0\}$$

is context-free must be wrong

Conclusion: L is not context-free

Turing Machines

The Language Hierarchy



Languages accepted by Turing Machines

$a^n b^n c^n$

ww

Context-Free Languages

$a^n b^n$

ww^R

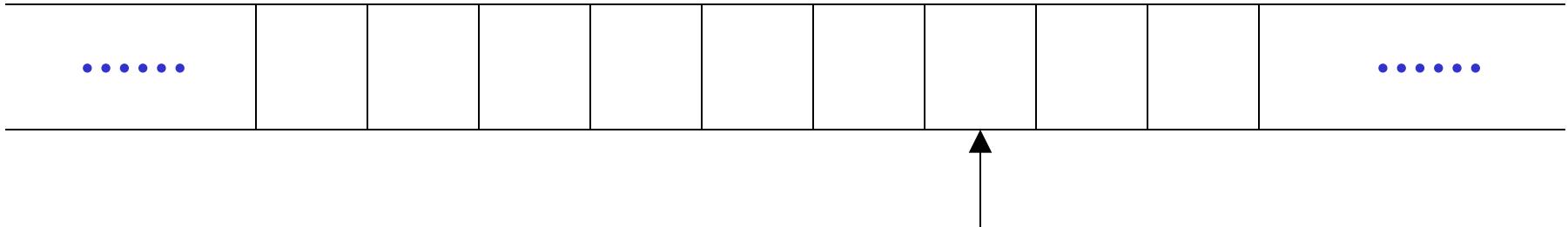
Regular Languages

a^*

$a^* b^*$

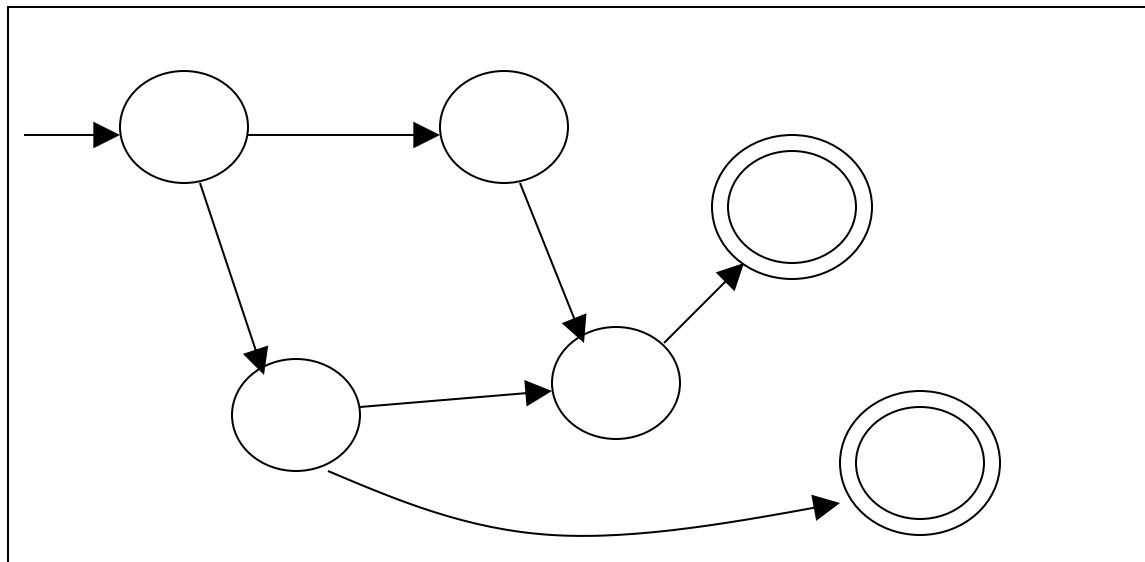
A Turing Machine

Tape



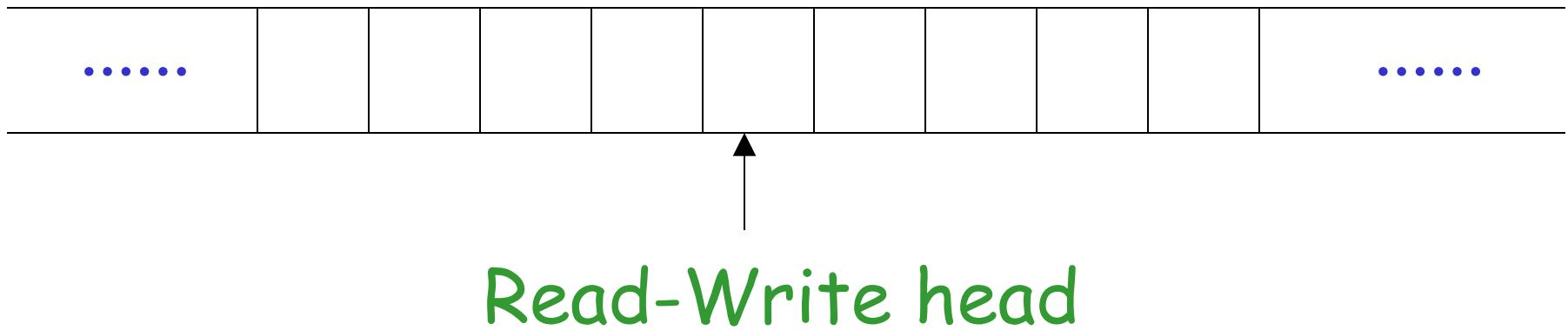
Read-Write head

Control Unit

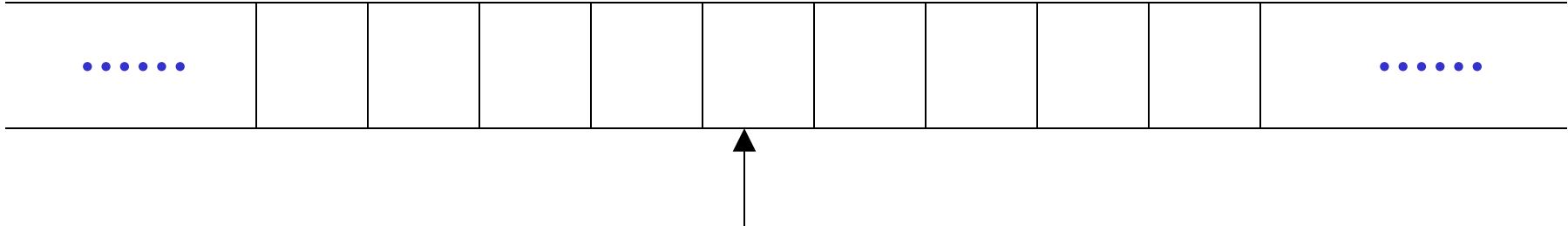


The Tape

No boundaries -- infinite length



The head moves Left or Right



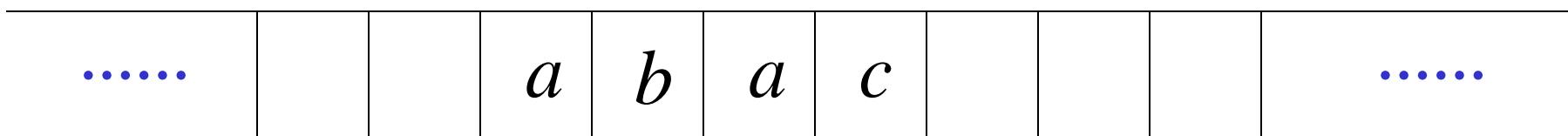
Read-Write head

The head at each transition (time step):

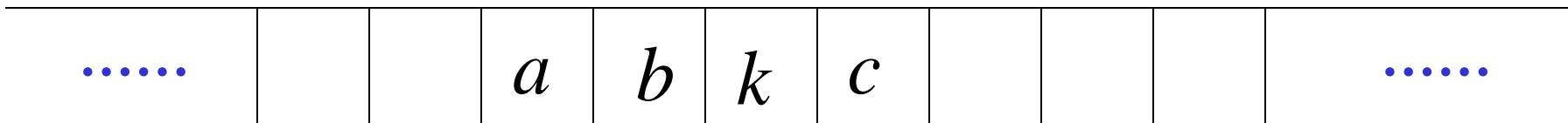
1. Reads a symbol
2. Writes a symbol
3. Moves Left or Right

Example:

Time 0



Time 1



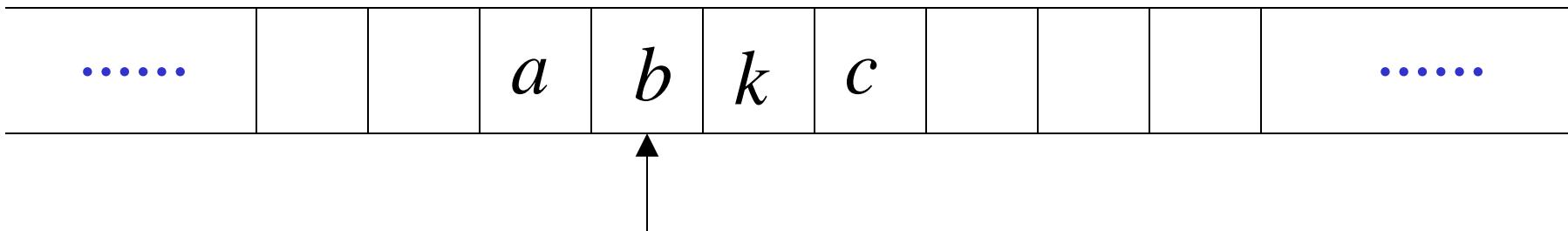
1. Reads a

2. Writes k

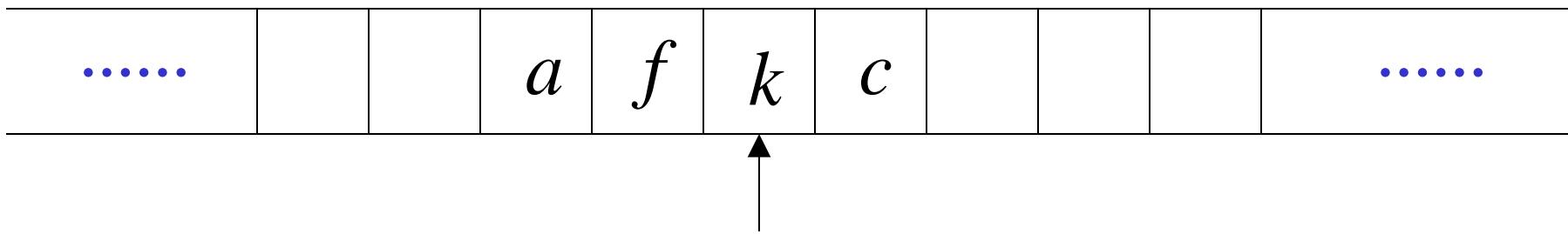
3. Moves Left

Example:

Time 1



Time 2

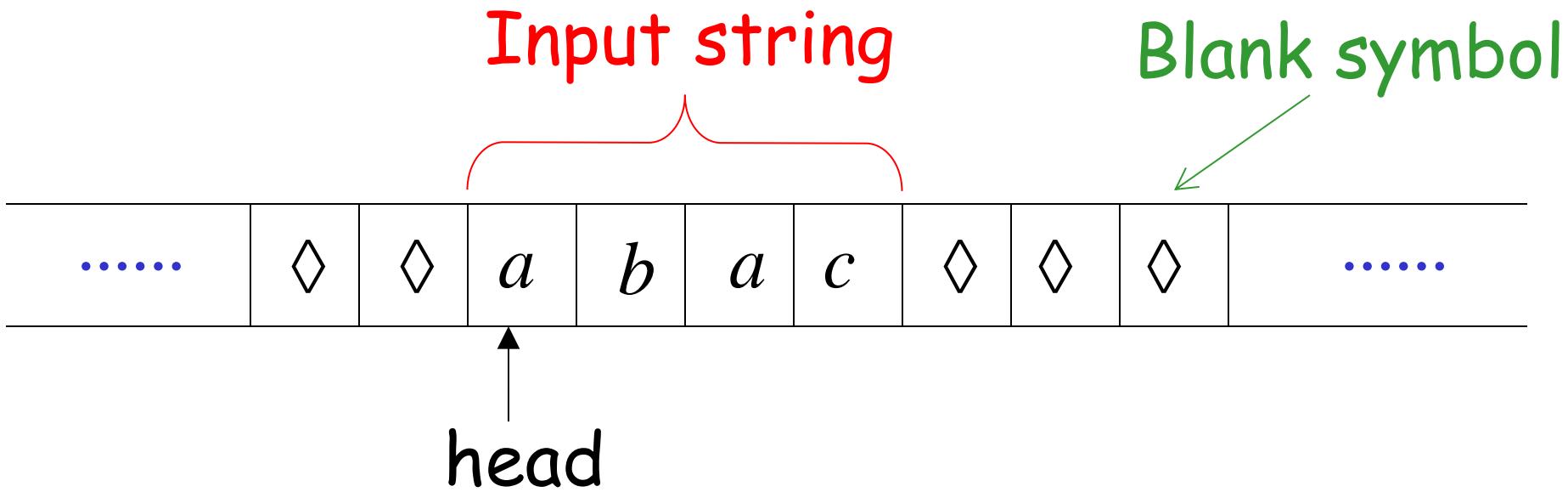


1. Reads b

2. Writes f

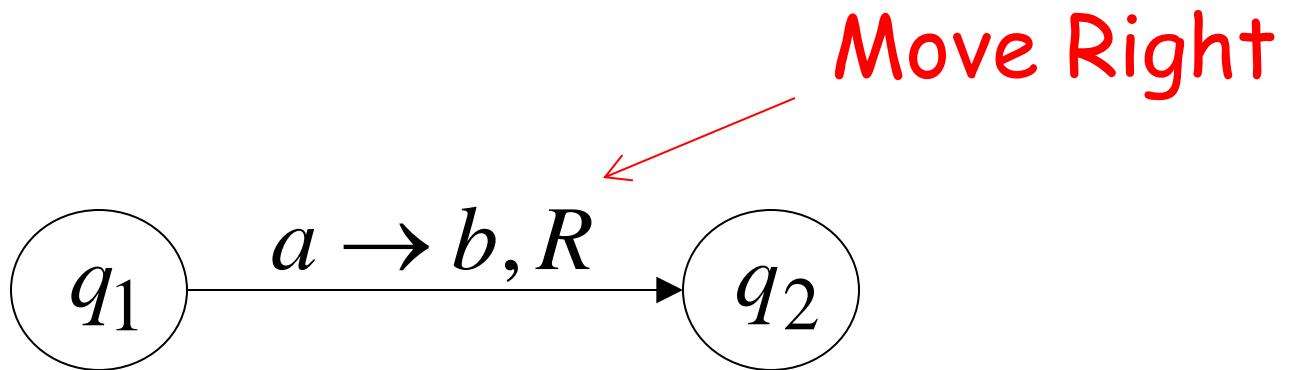
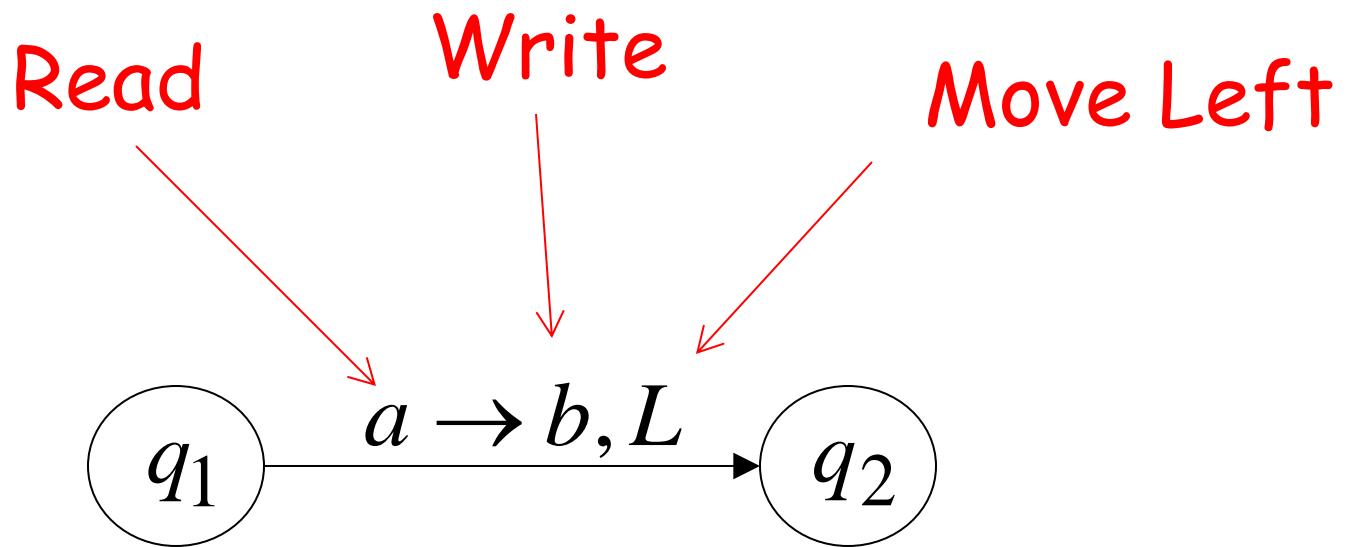
3. Moves Right

The Input String

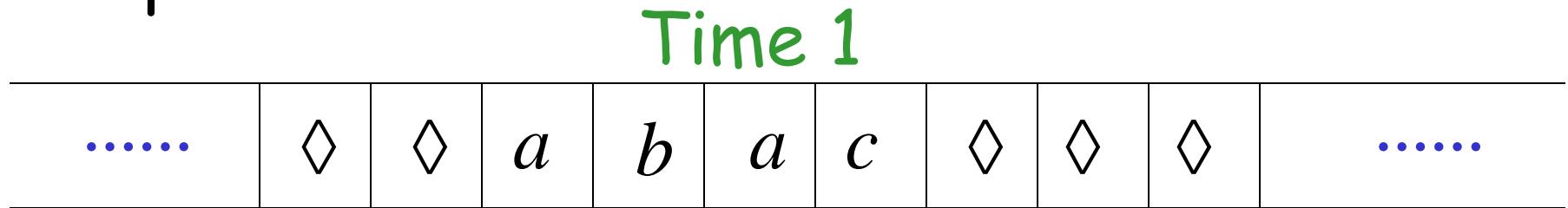


Head starts at the leftmost position
of the input string

States & Transitions

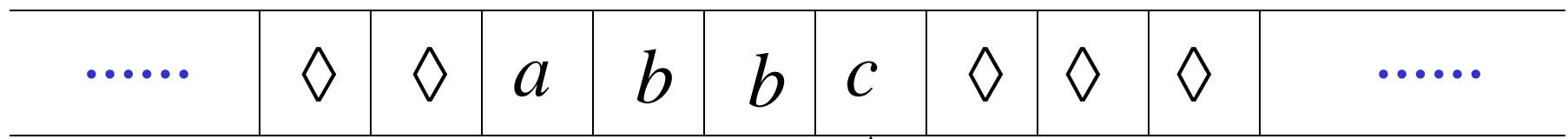


Example:

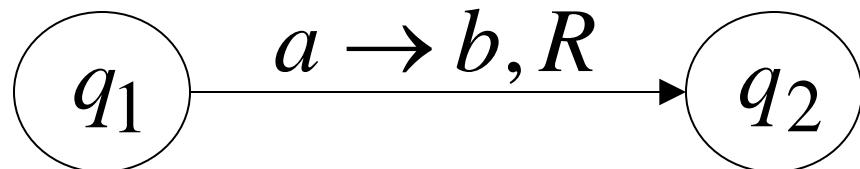


current state q_1

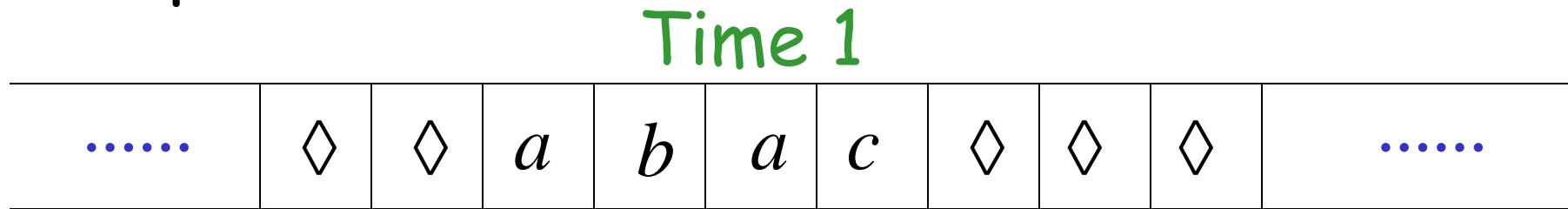
Time 2



q_2

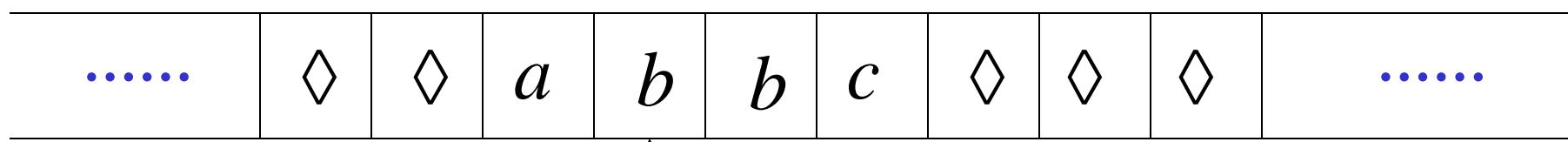


Example:

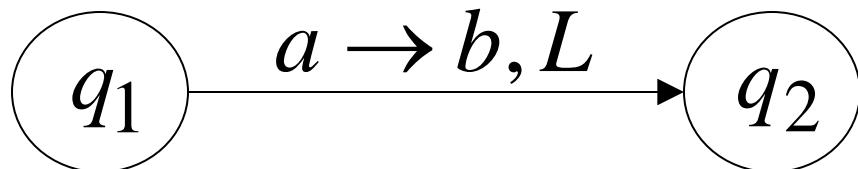


current state q_1

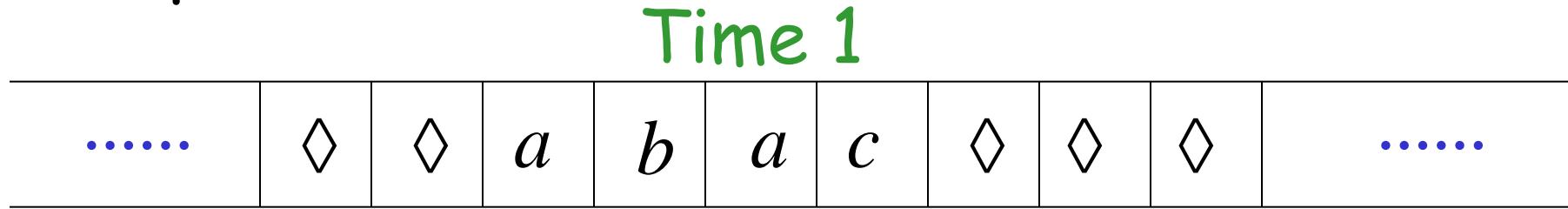
Time 2



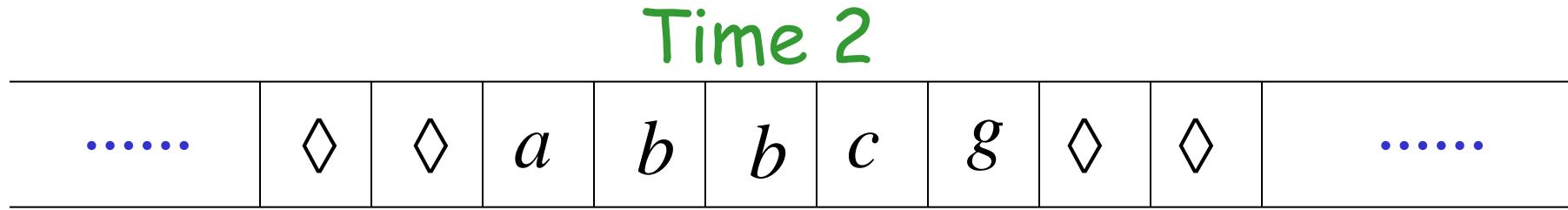
q_2



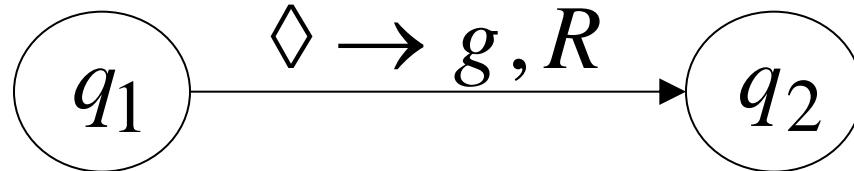
Example:



current state q_1



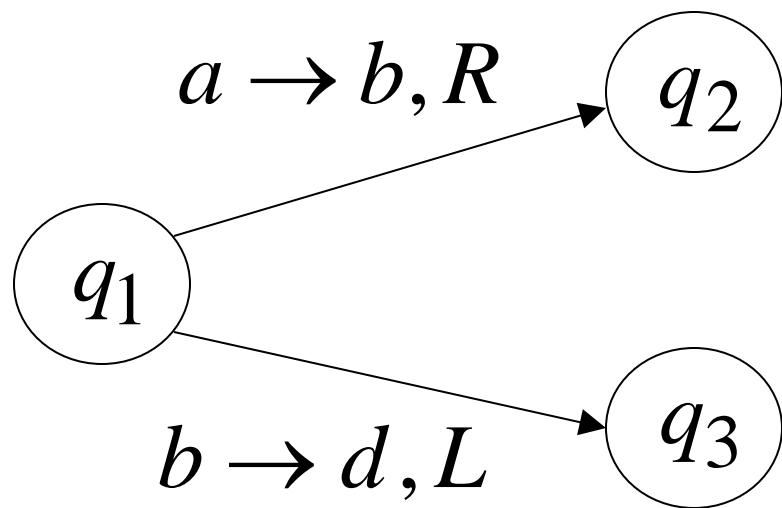
q_2



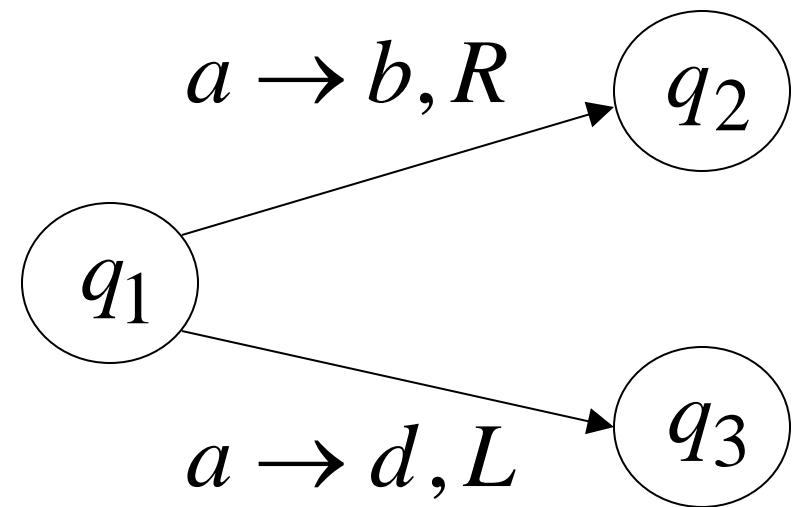
Determinism

Turing Machines are deterministic

Allowed



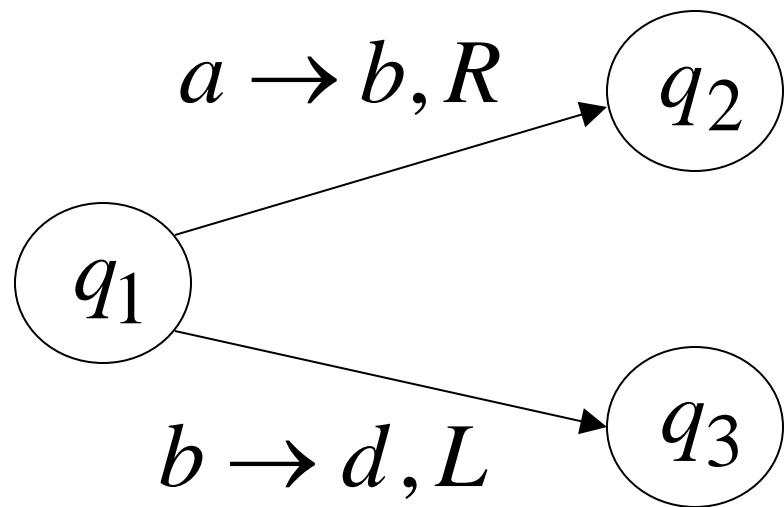
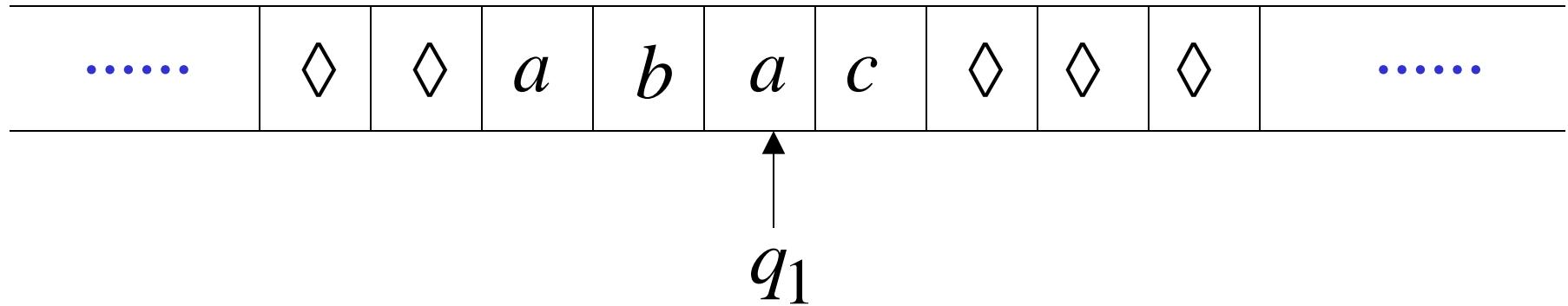
Not Allowed



No lambda transitions allowed

Partial Transition Function

Example:



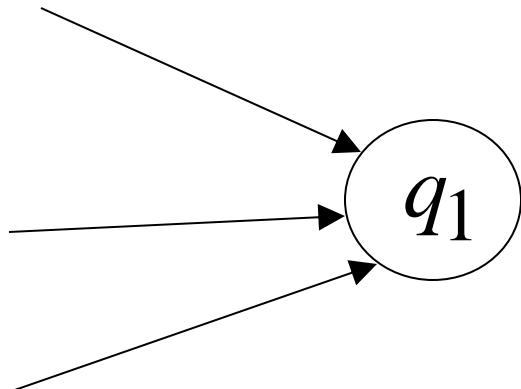
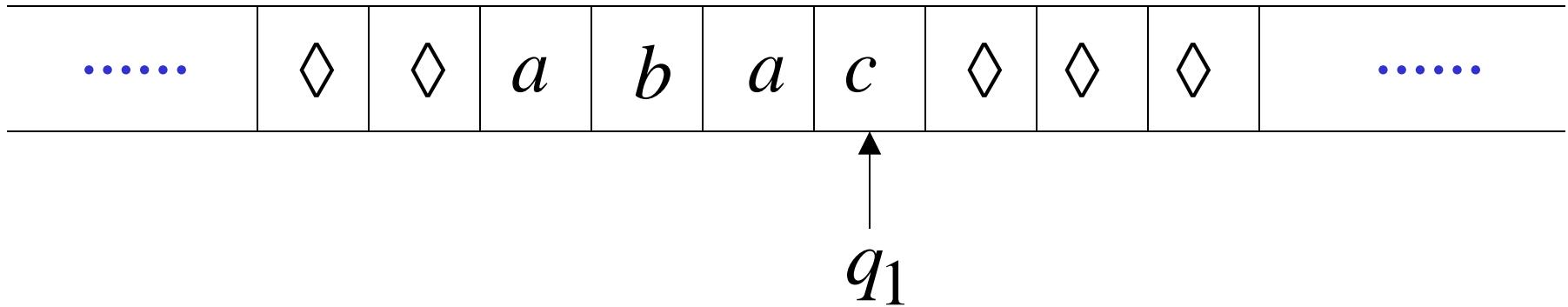
Allowed:

No transition
for input symbol c

Halting

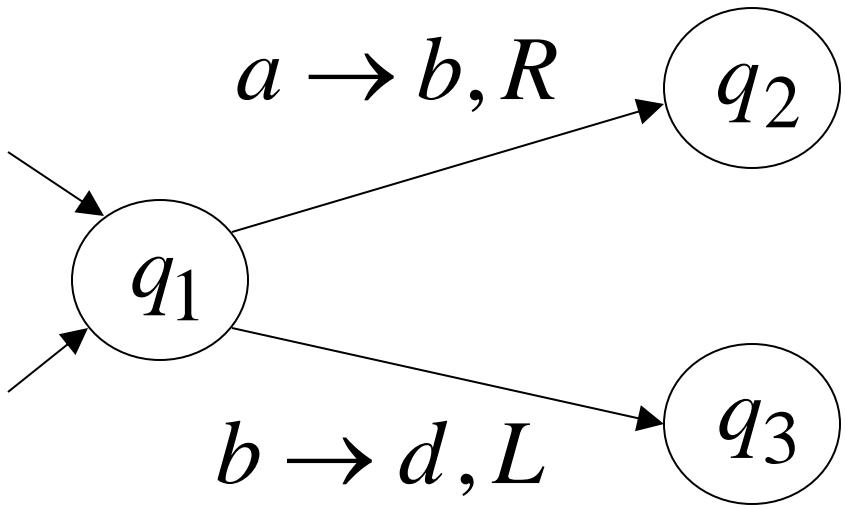
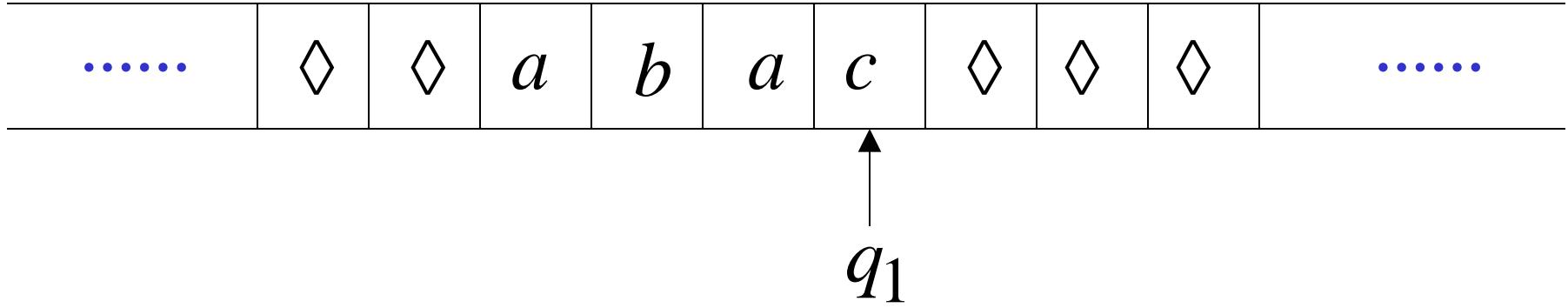
The machine *halts* in a state if there is no transition to follow

Halting Example 1:



No transition from q_1
HALT!!!

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
or
If machine enters
an *infinite loop*

Observation:

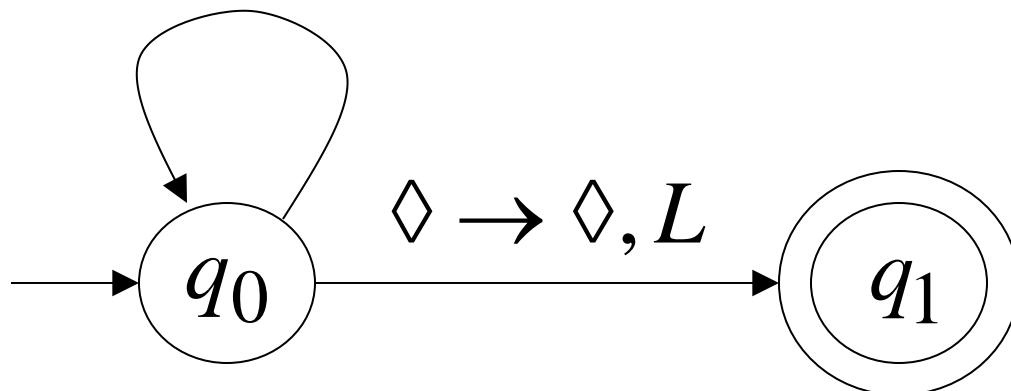
In order to accept an input string,
it is not necessary to scan all the
symbols in the string

Turing Machine Example

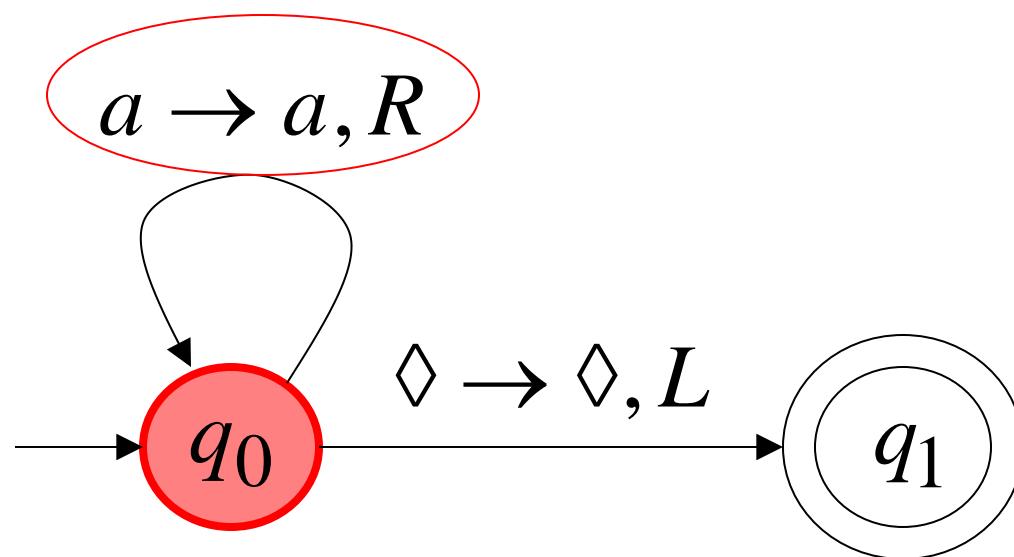
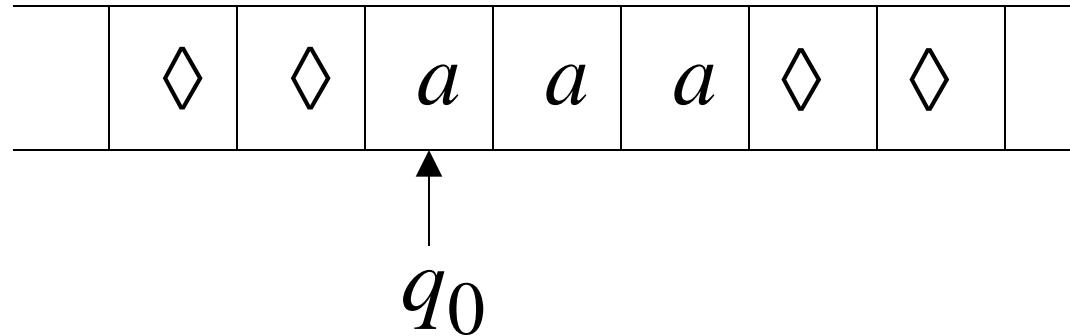
Input alphabet $\Sigma = \{a, b\}$

Accepts the language: a^*

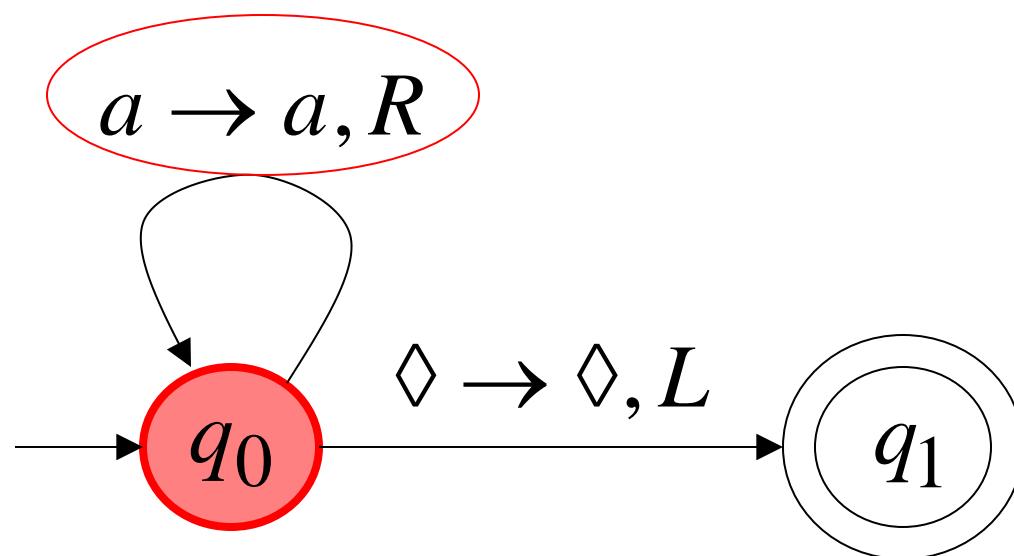
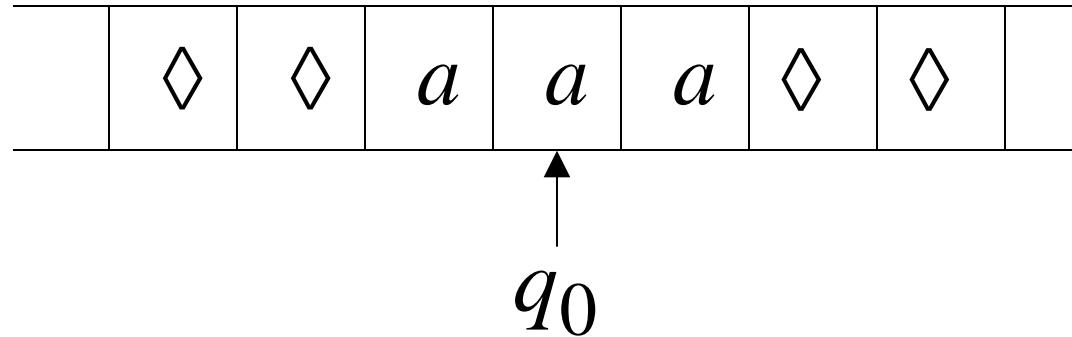
$$a \rightarrow a, R$$



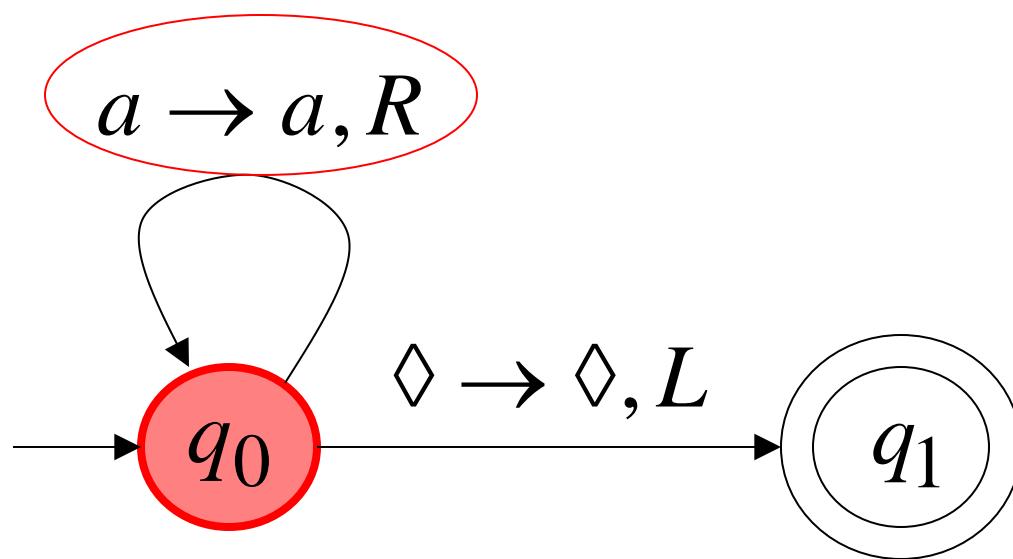
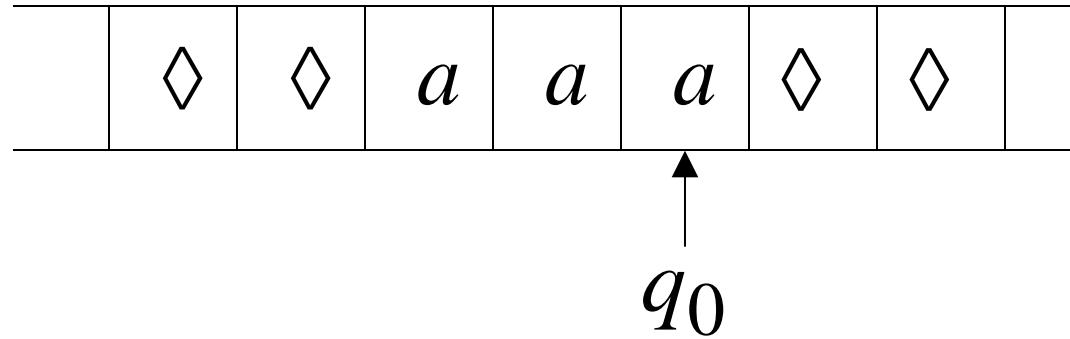
Time 0



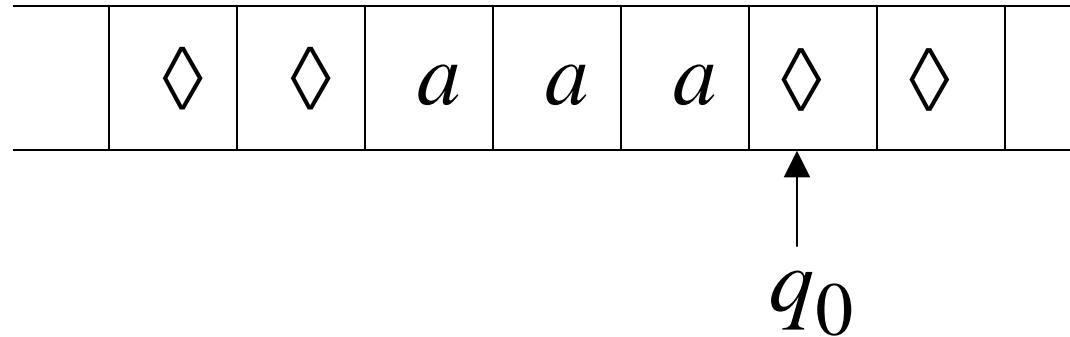
Time 1



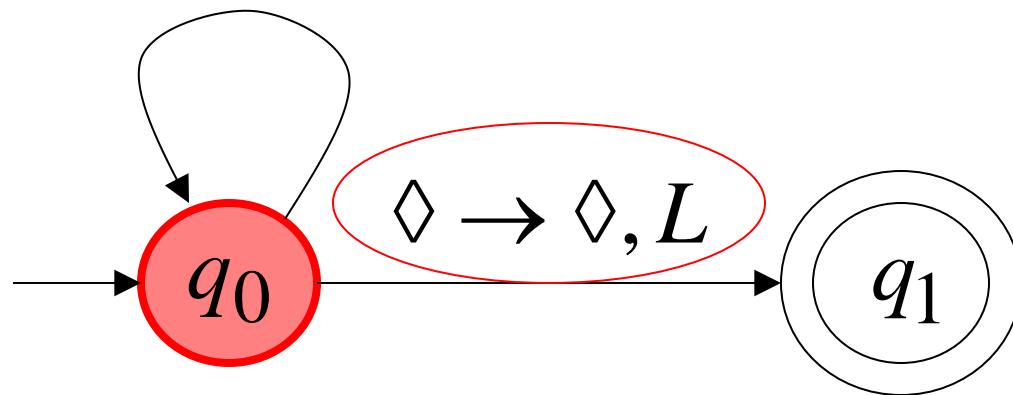
Time 2



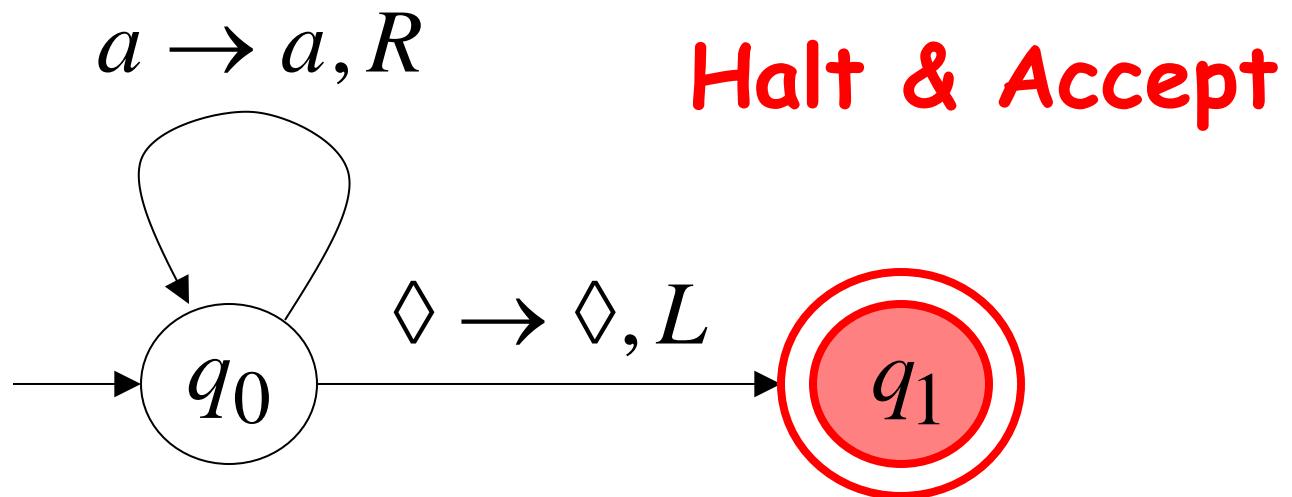
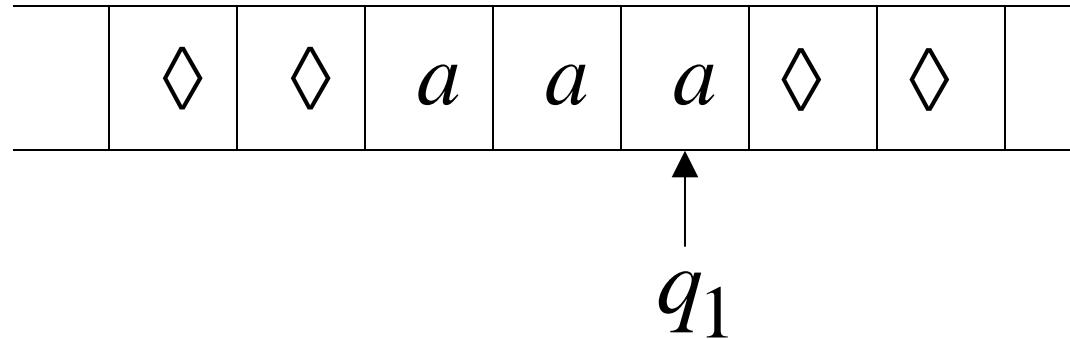
Time 3



$a \rightarrow a, R$

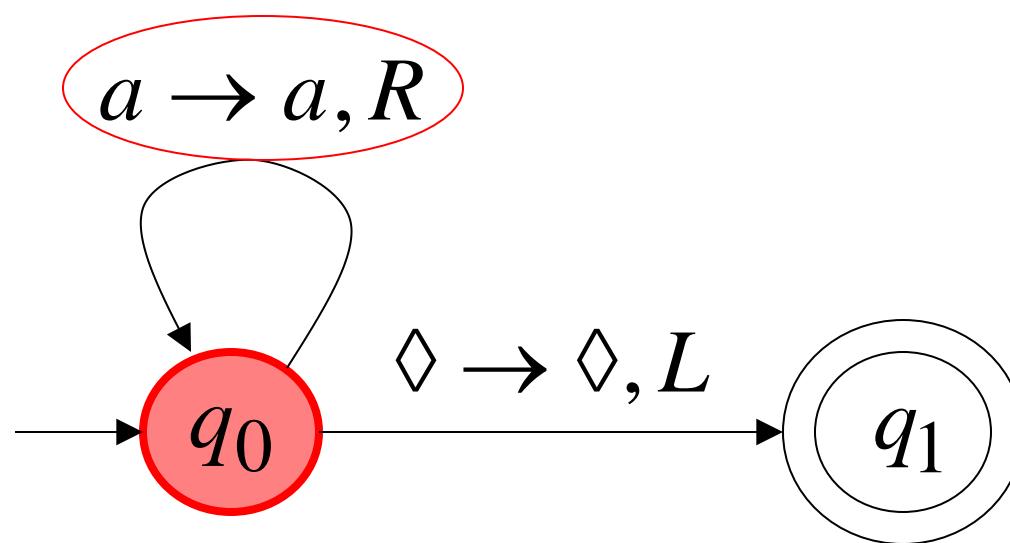
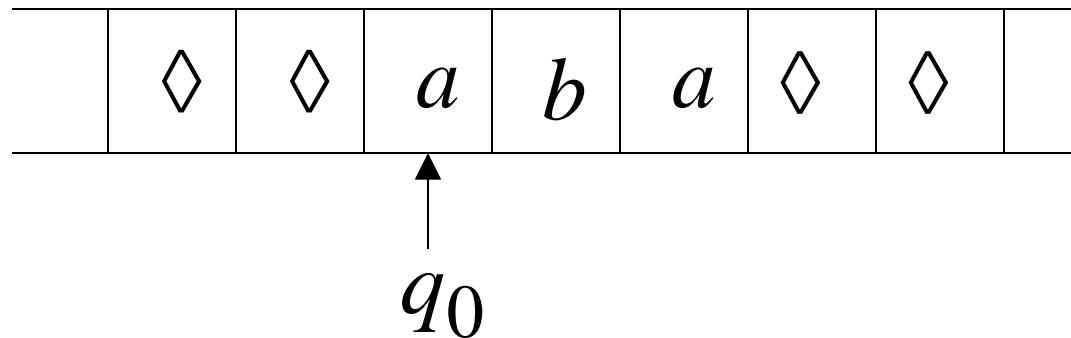


Time 4

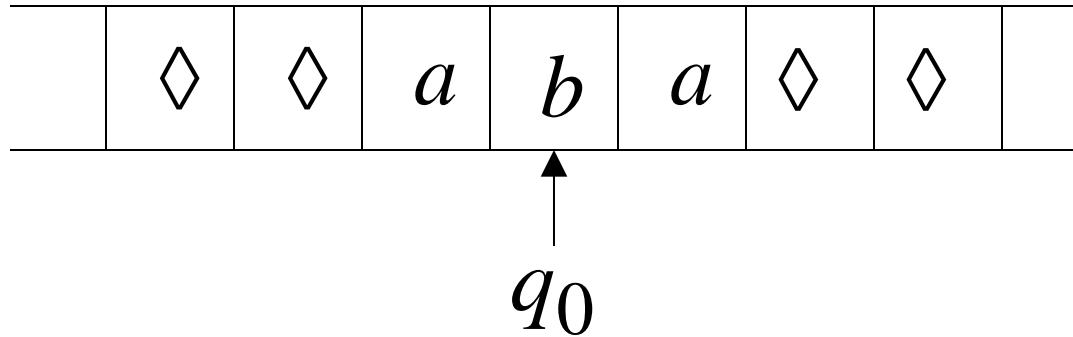


Rejection Example

Time 0



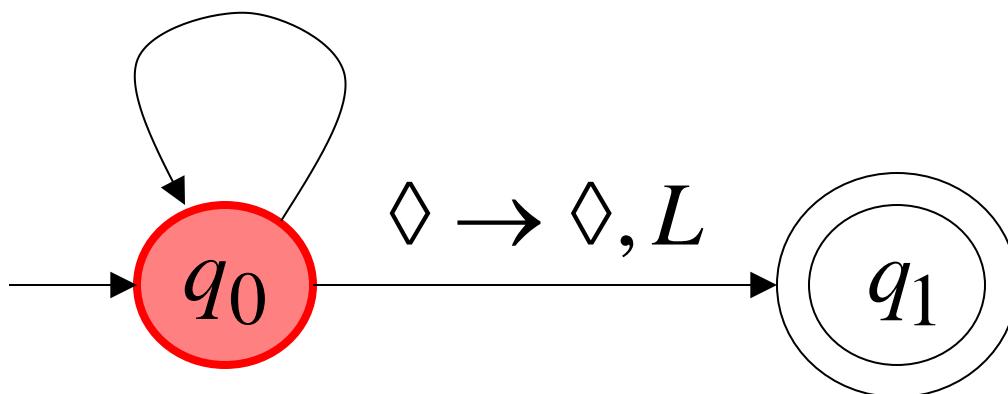
Time 1



No possible Transition

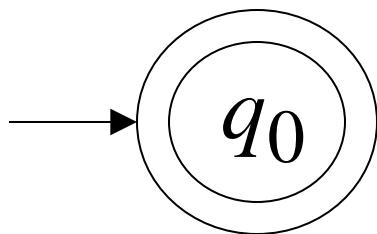
$a \rightarrow a, R$

Halt & Reject

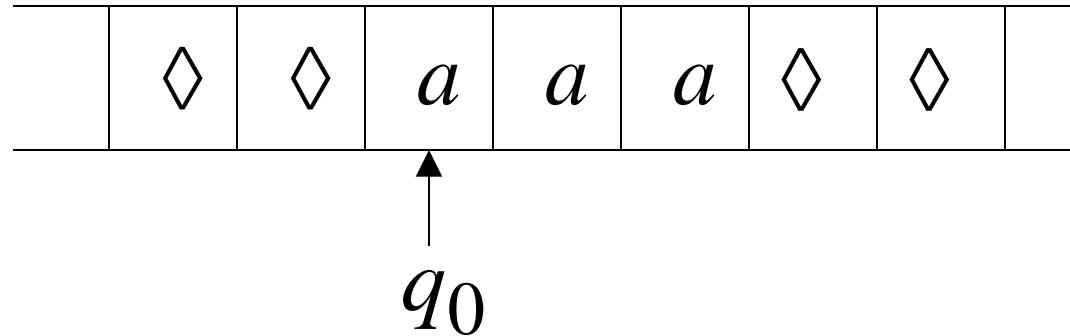


A simpler machine for same language
but for input alphabet $\Sigma = \{a\}$

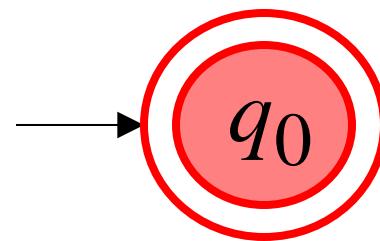
Accepts the language: a^*



Time 0



Halt & Accept



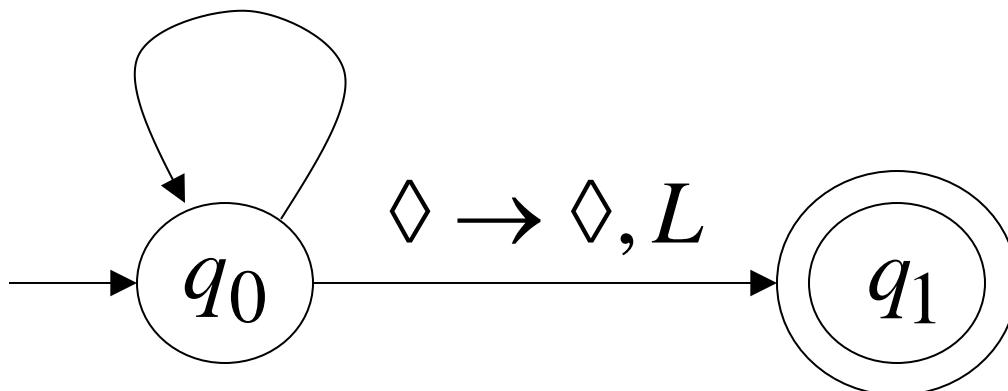
Not necessary to scan input

Infinite Loop Example

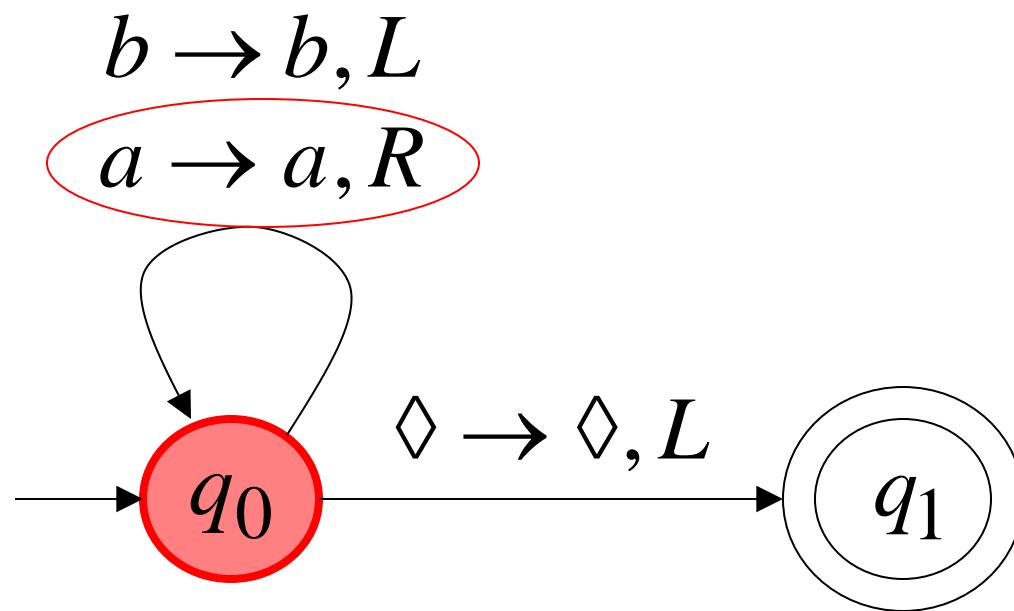
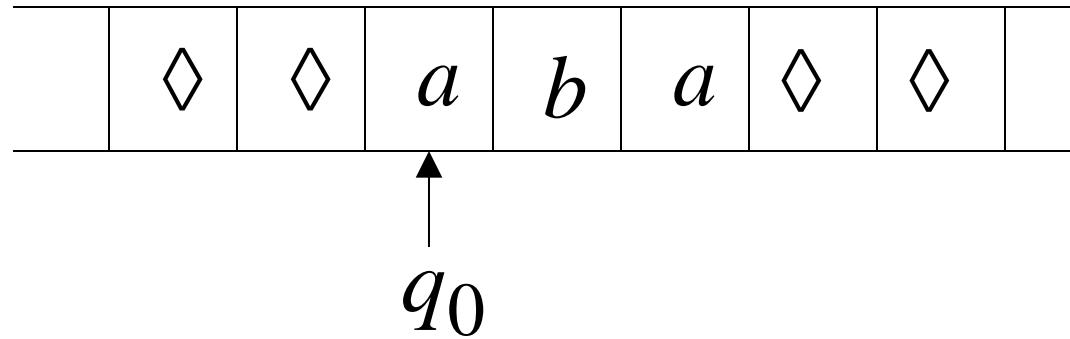
A Turing machine
for language $a^* + b(a+b)^*$

$b \rightarrow b, L$

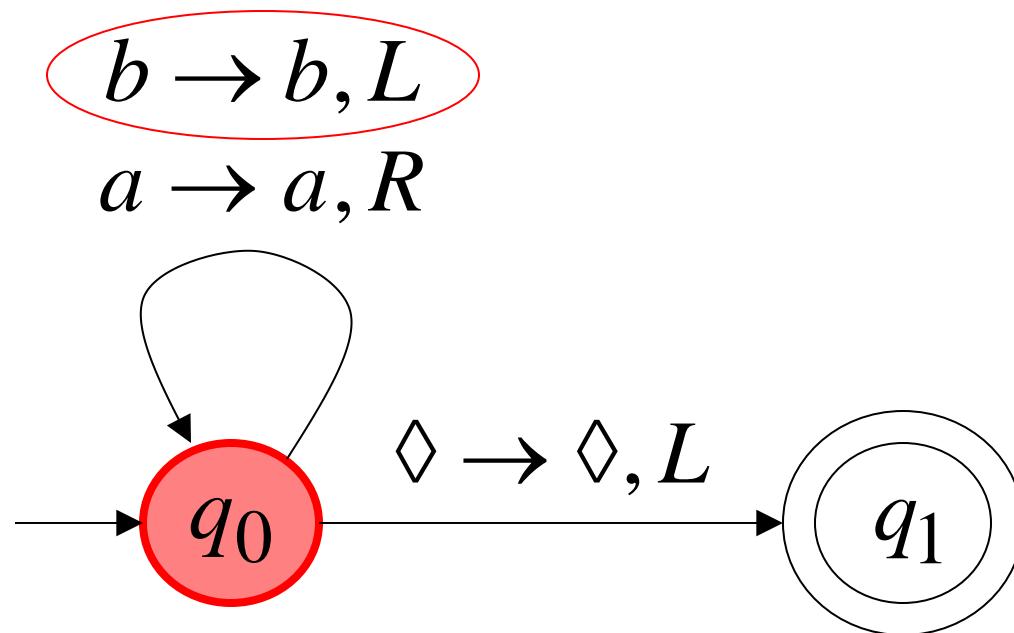
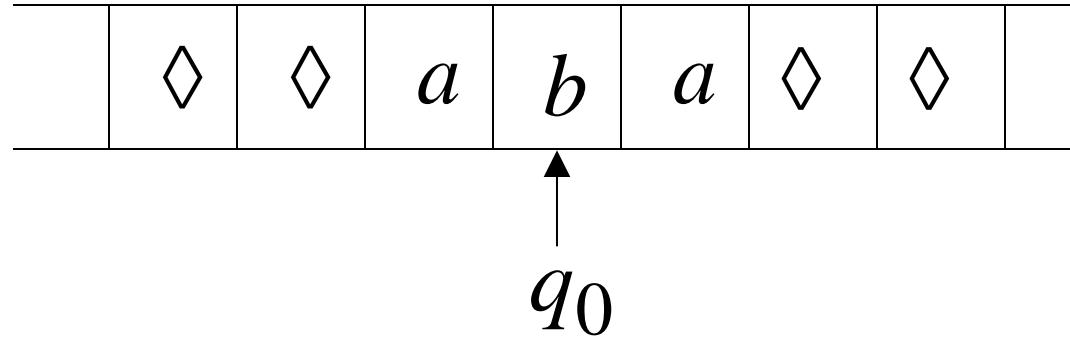
$a \rightarrow a, R$



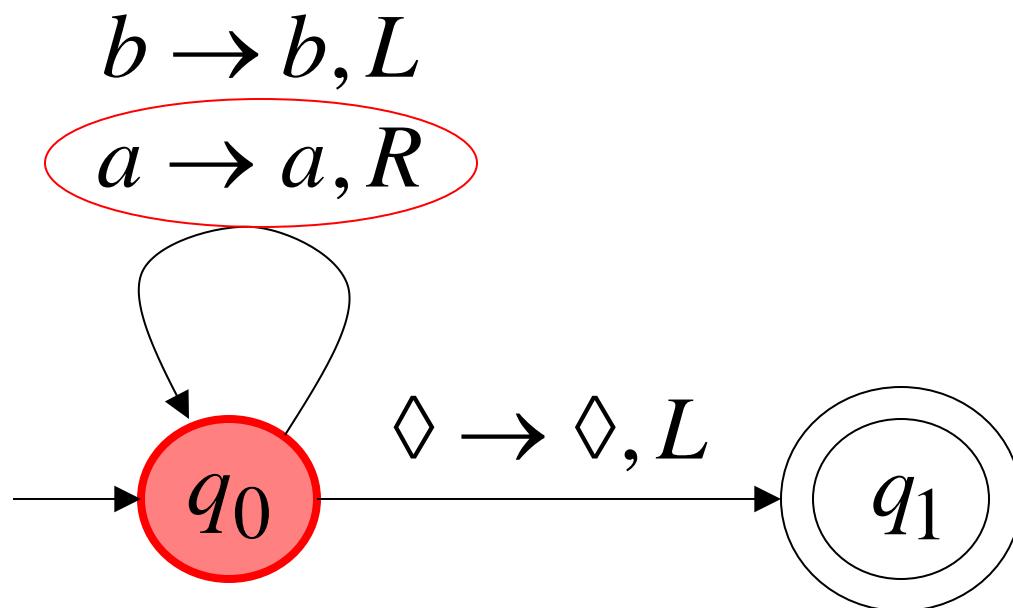
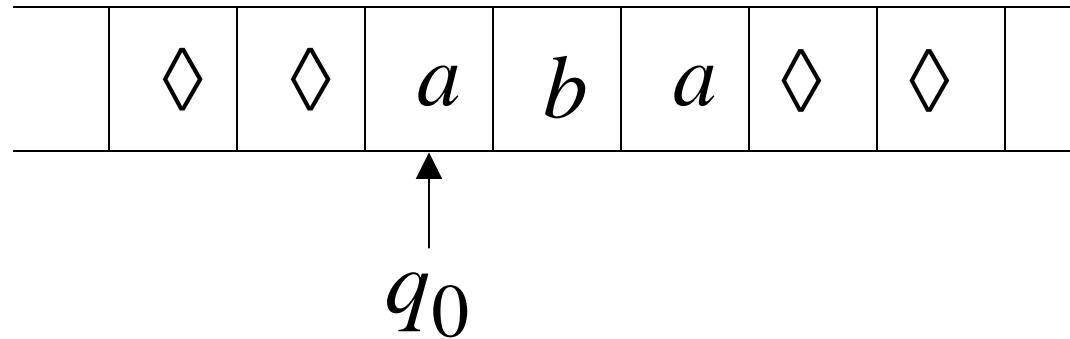
Time 0



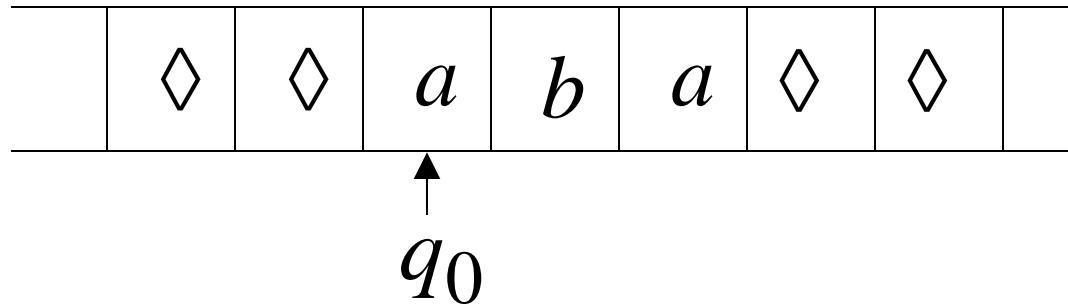
Time 1



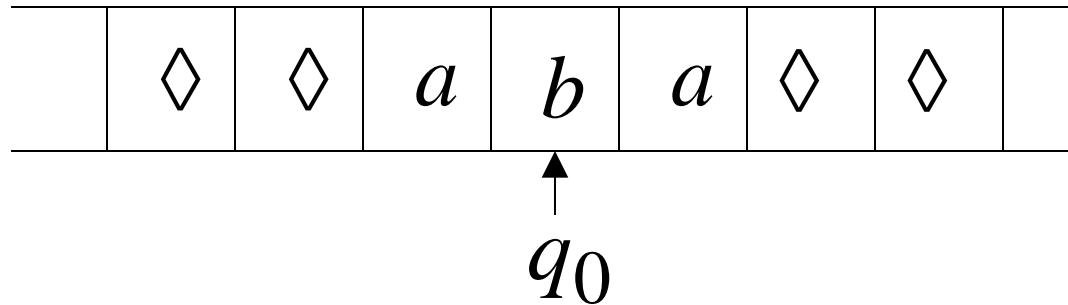
Time 2



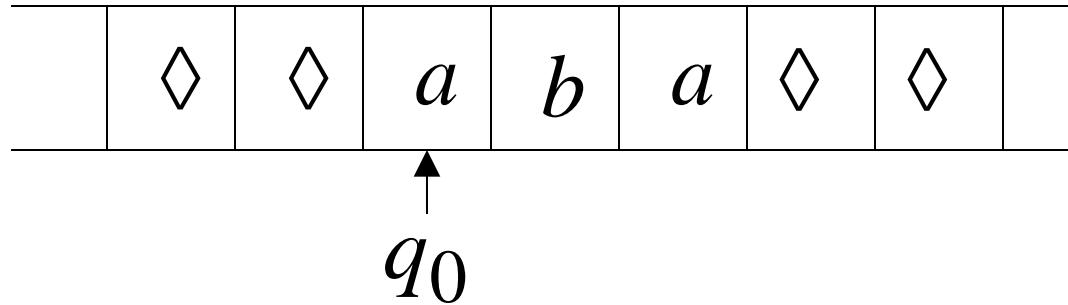
Time 2



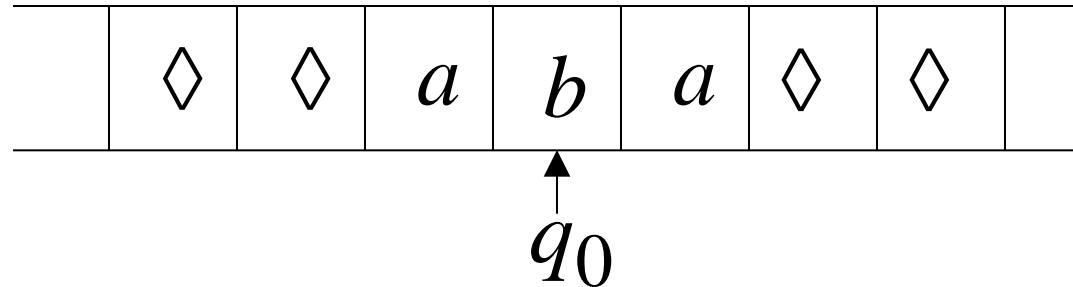
Time 3



Time 4



Time 5



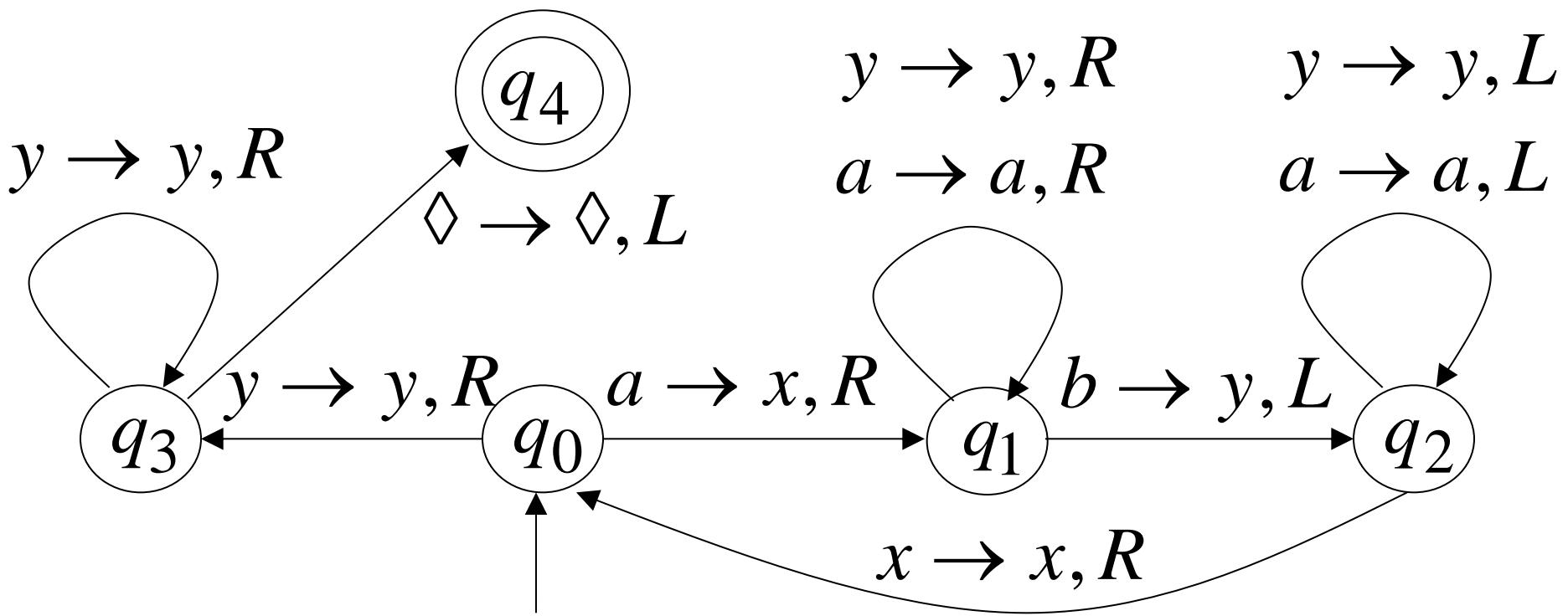
Infinite loop

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:

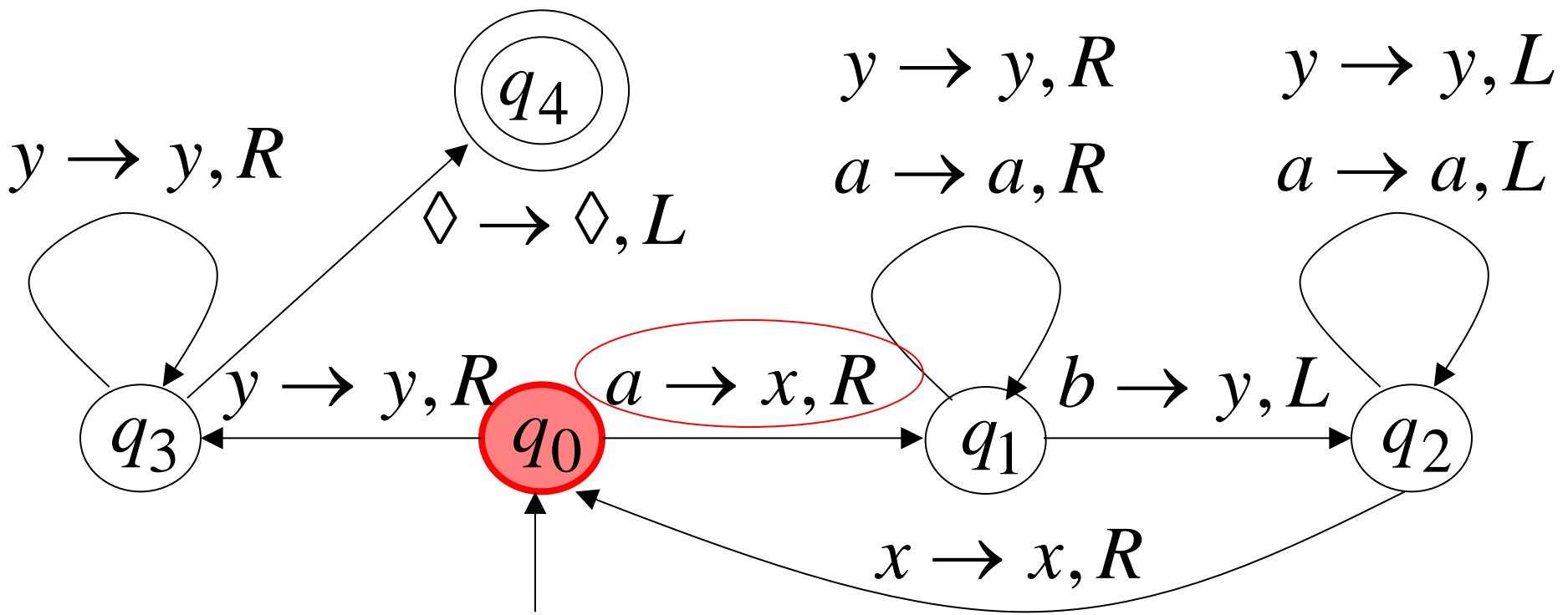
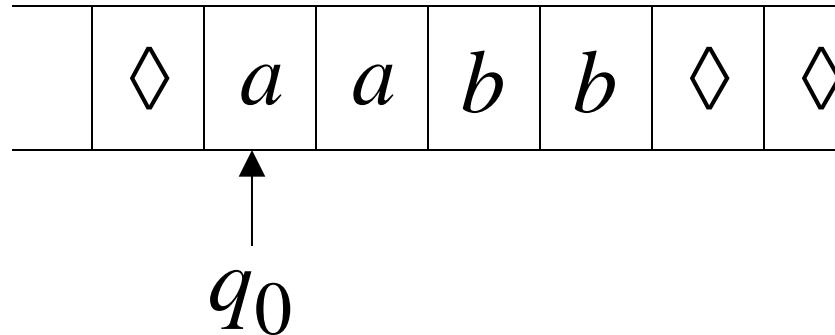
 replace leftmost a with x

 find leftmost b and replace it with y

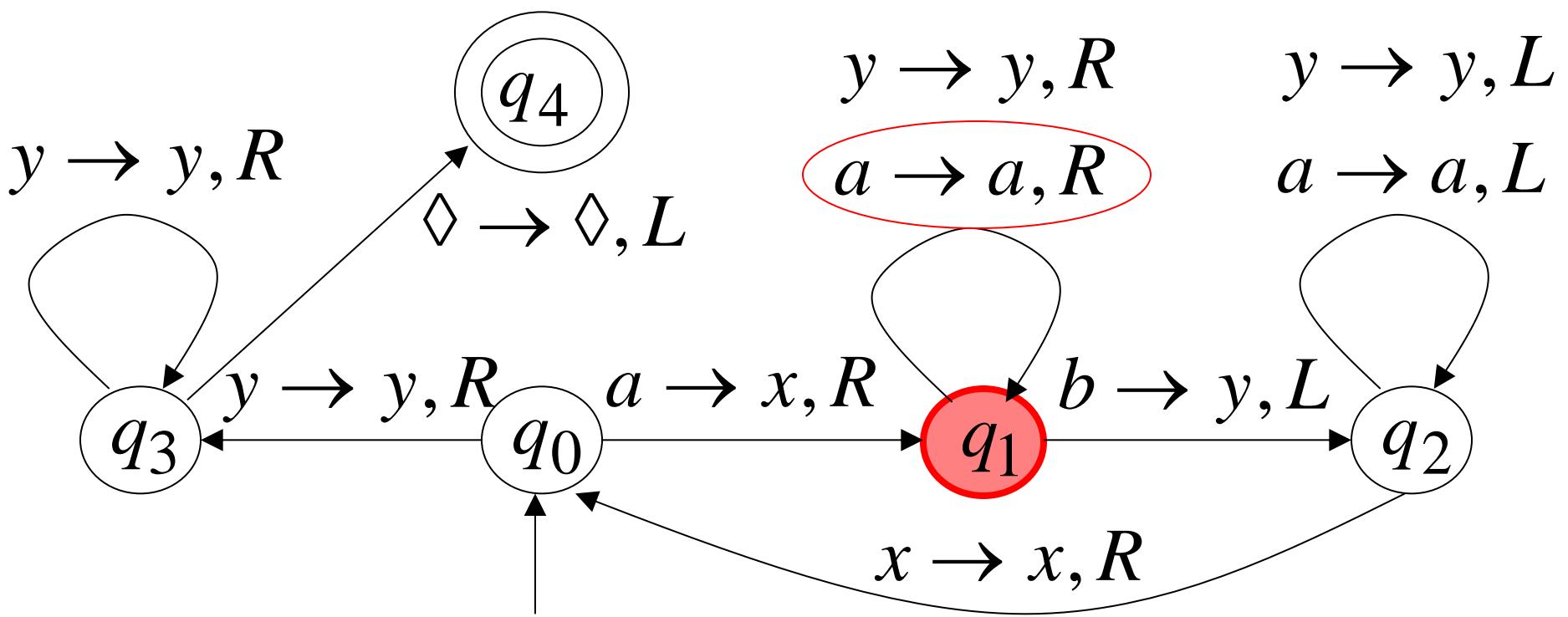
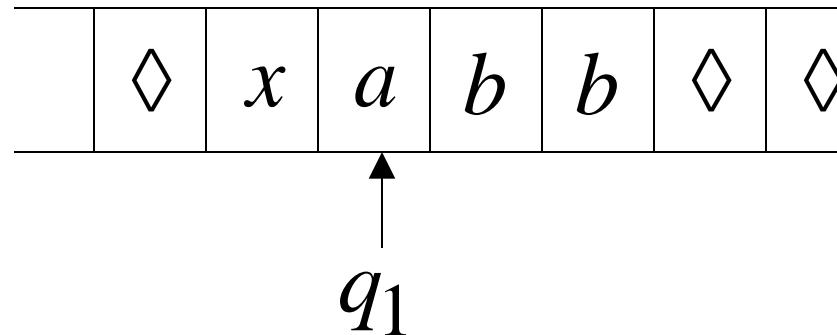
Until there are no more a's or b's

If there is a remaining a or b reject

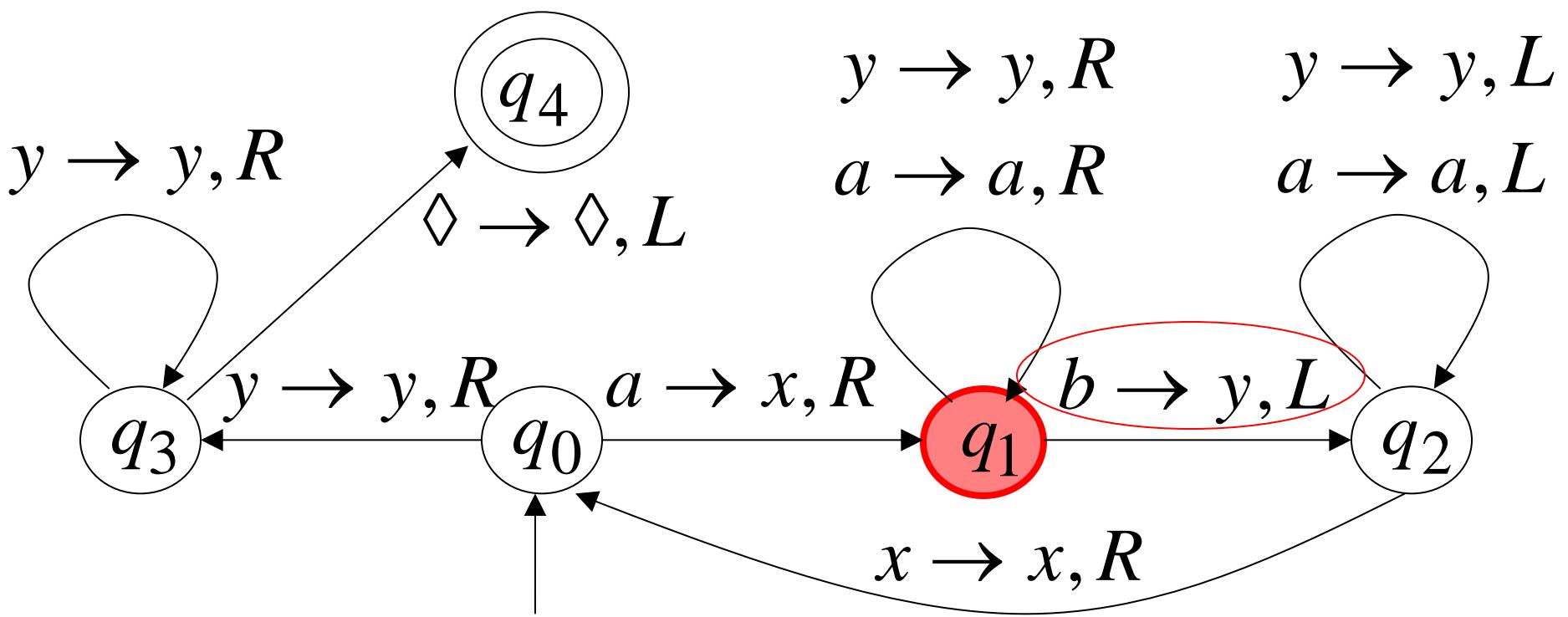
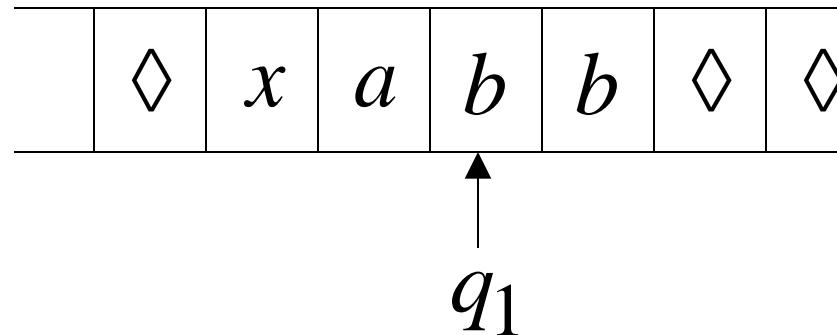
Time 0



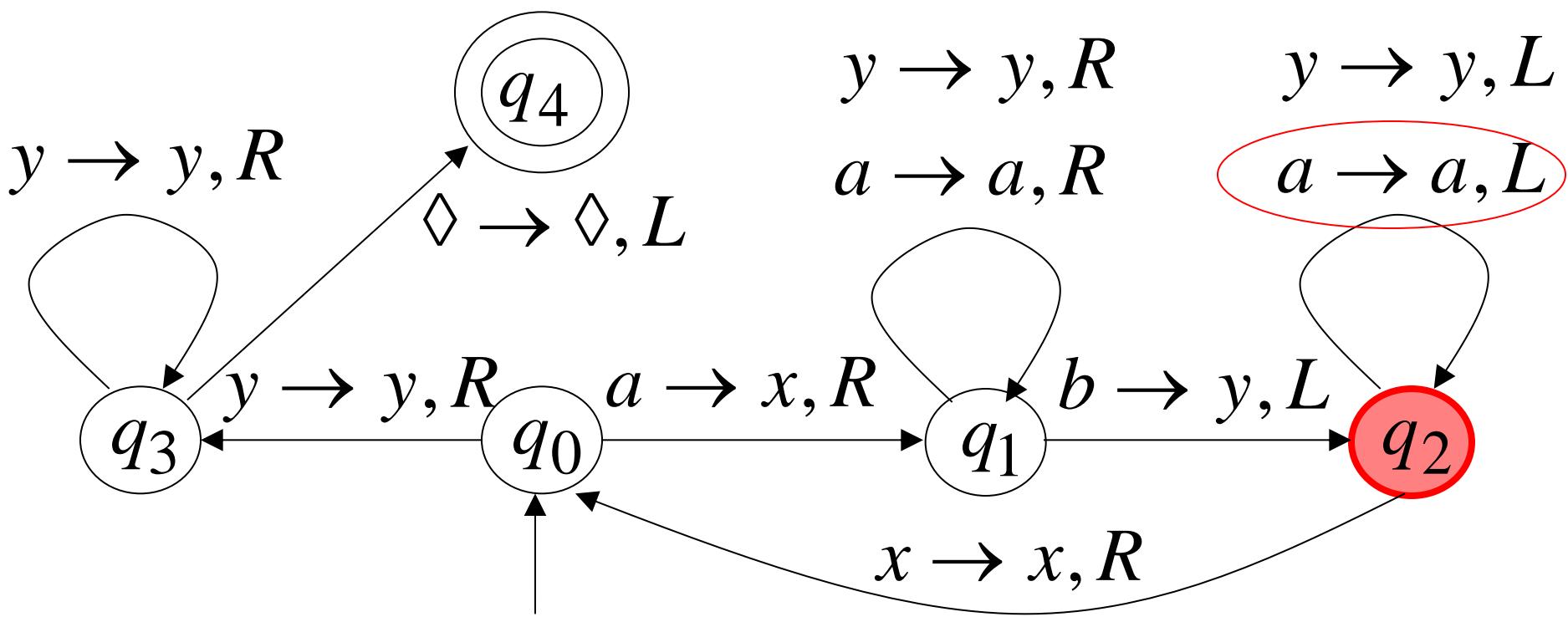
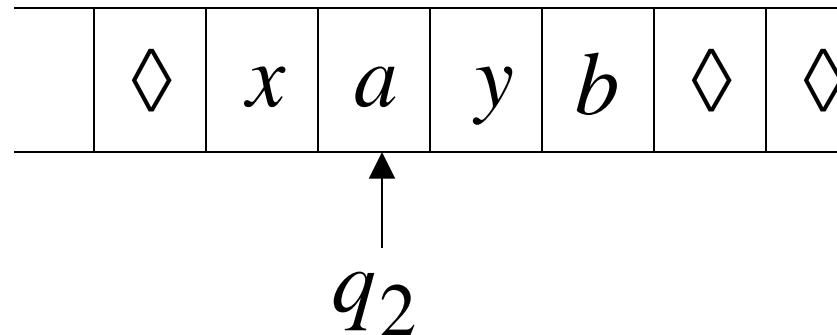
Time 1



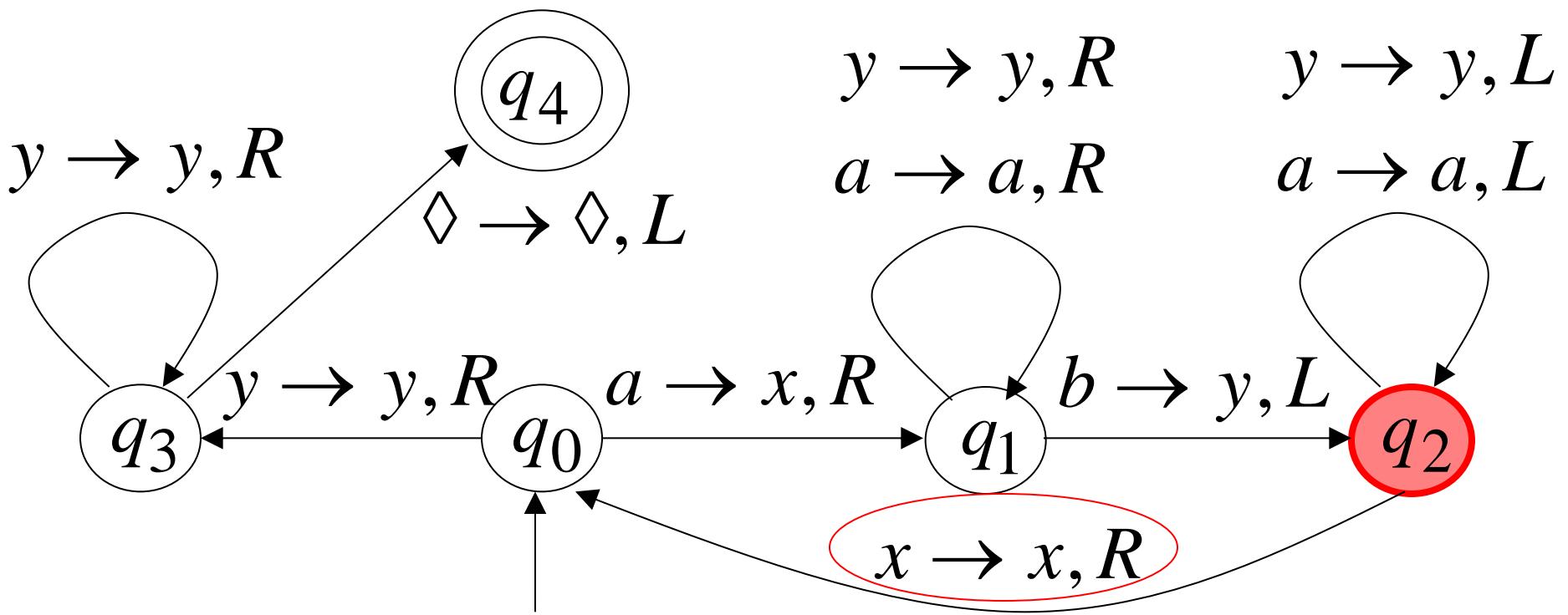
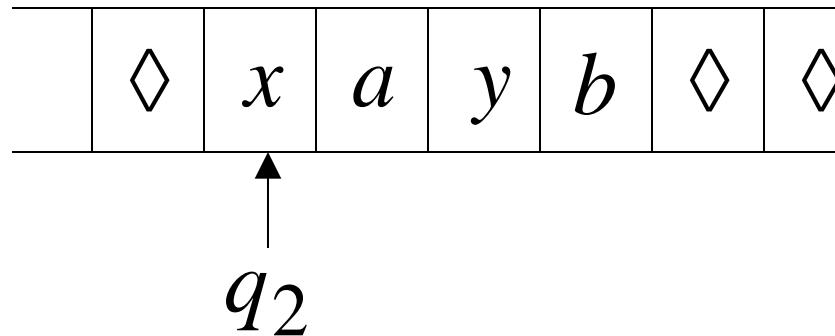
Time 2



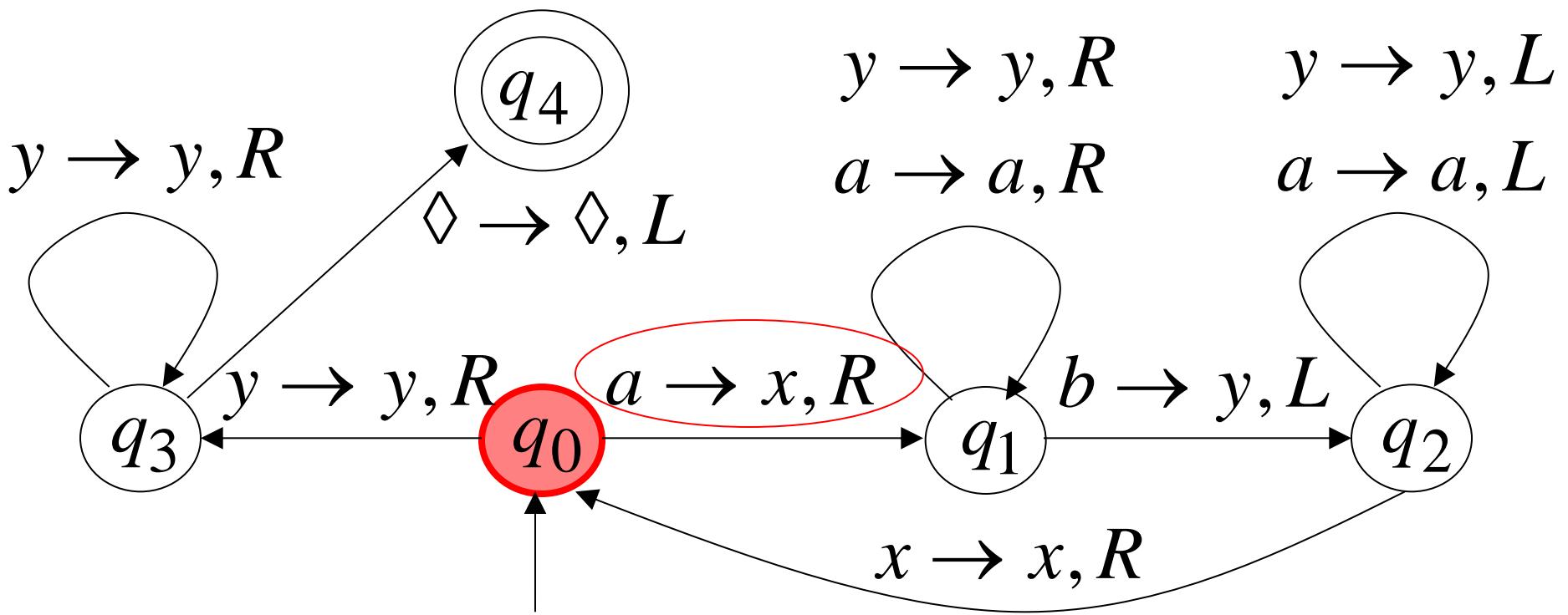
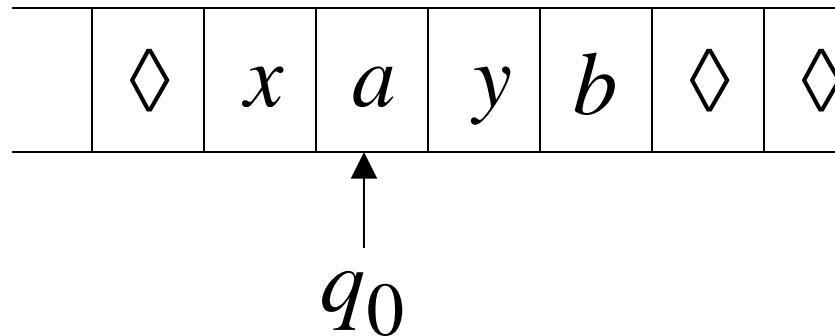
Time 3



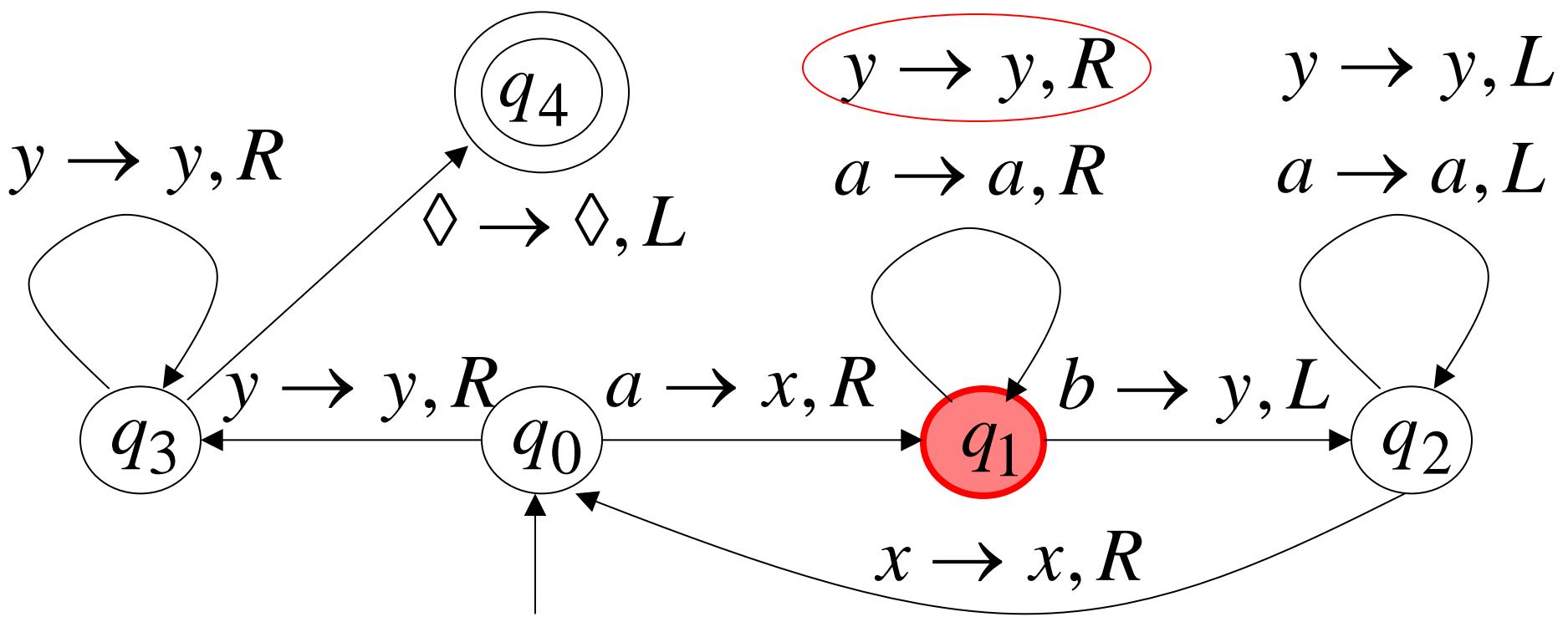
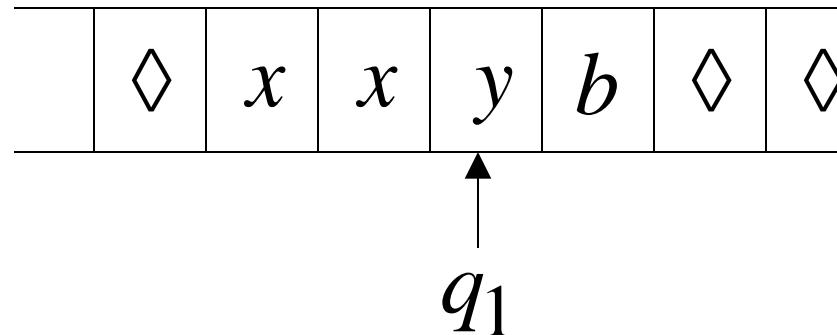
Time 4



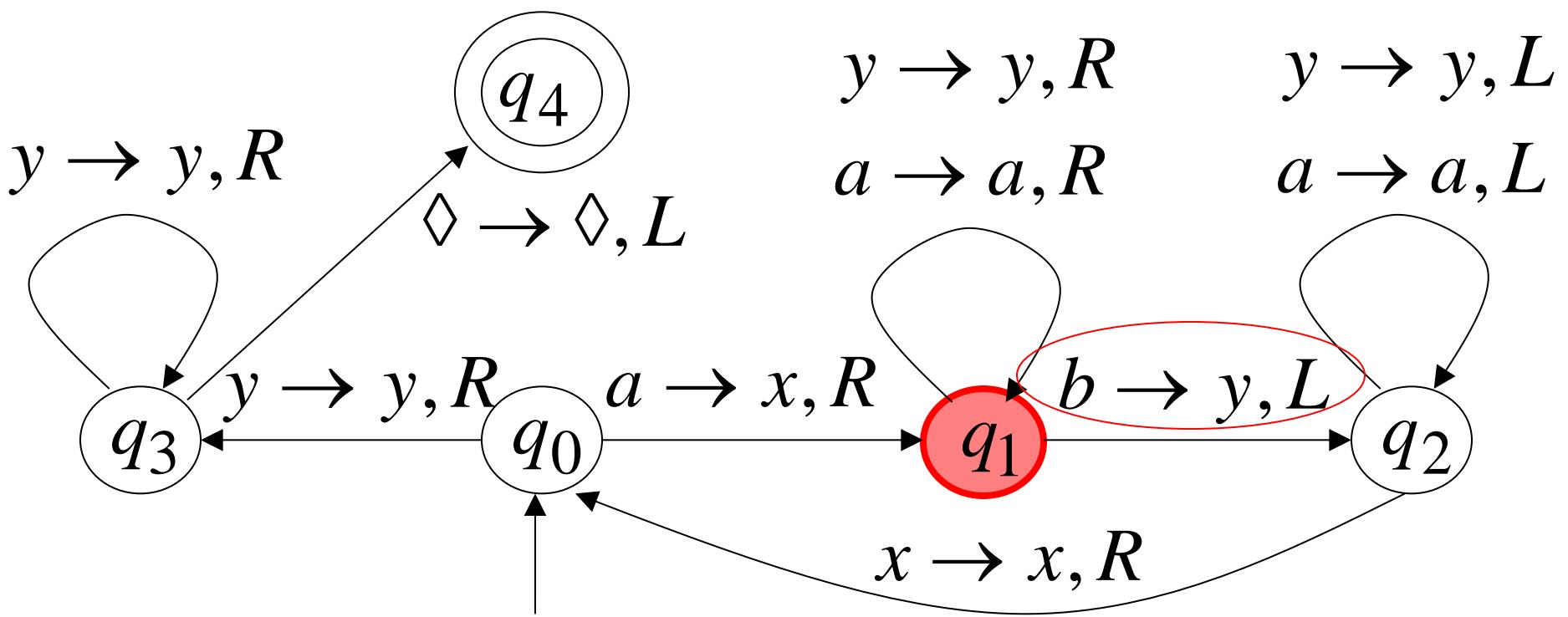
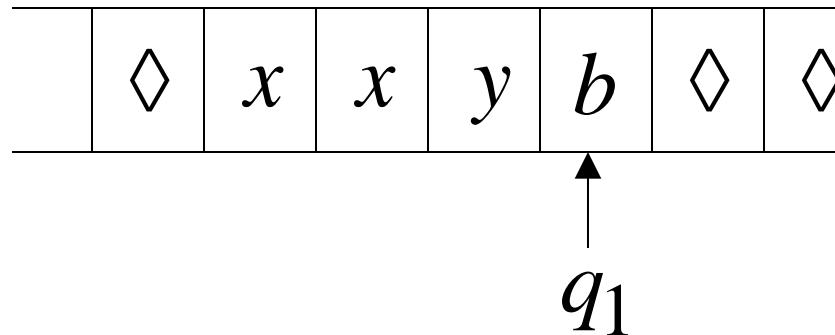
Time 5



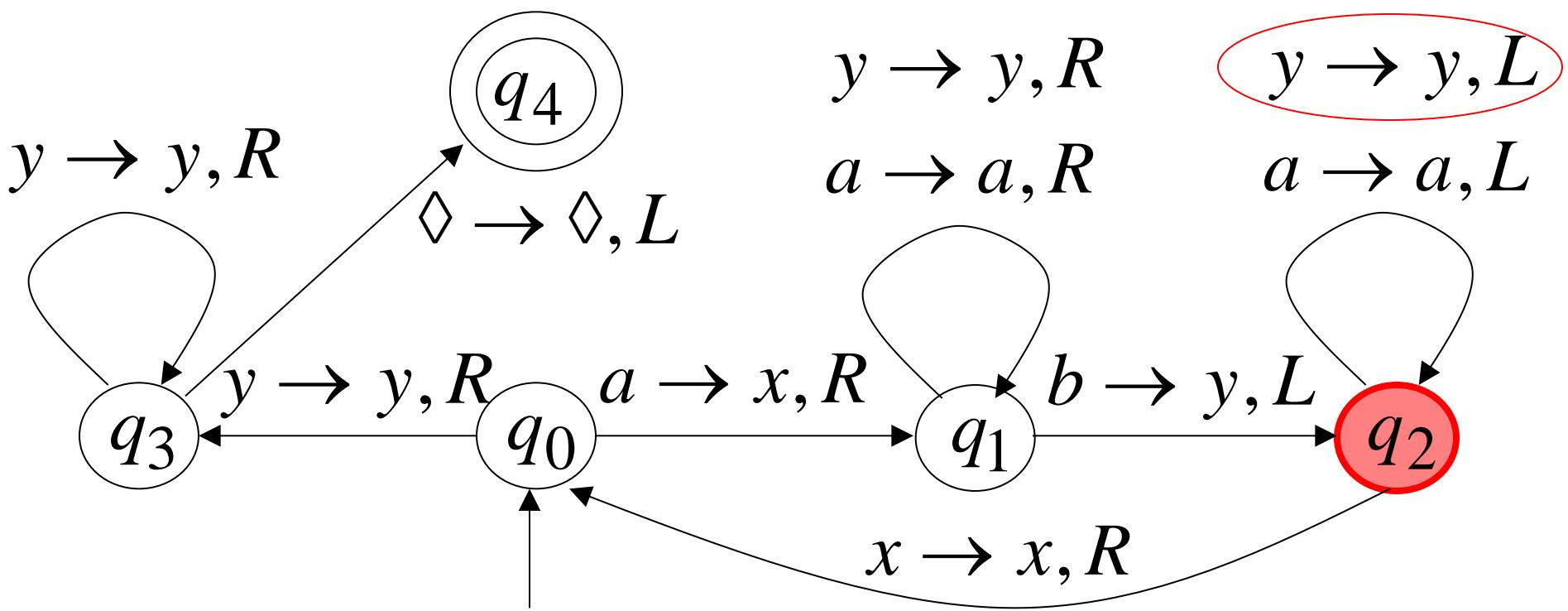
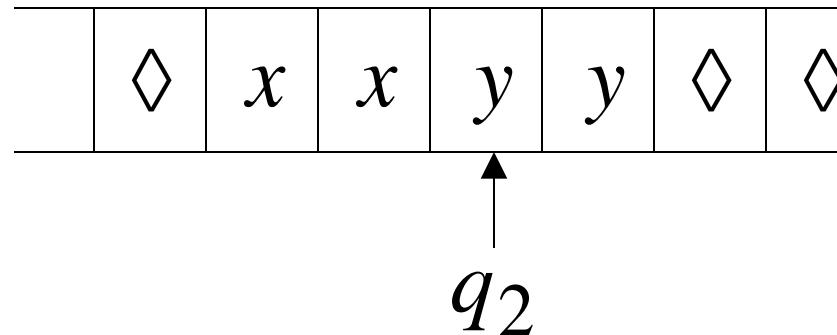
Time 6



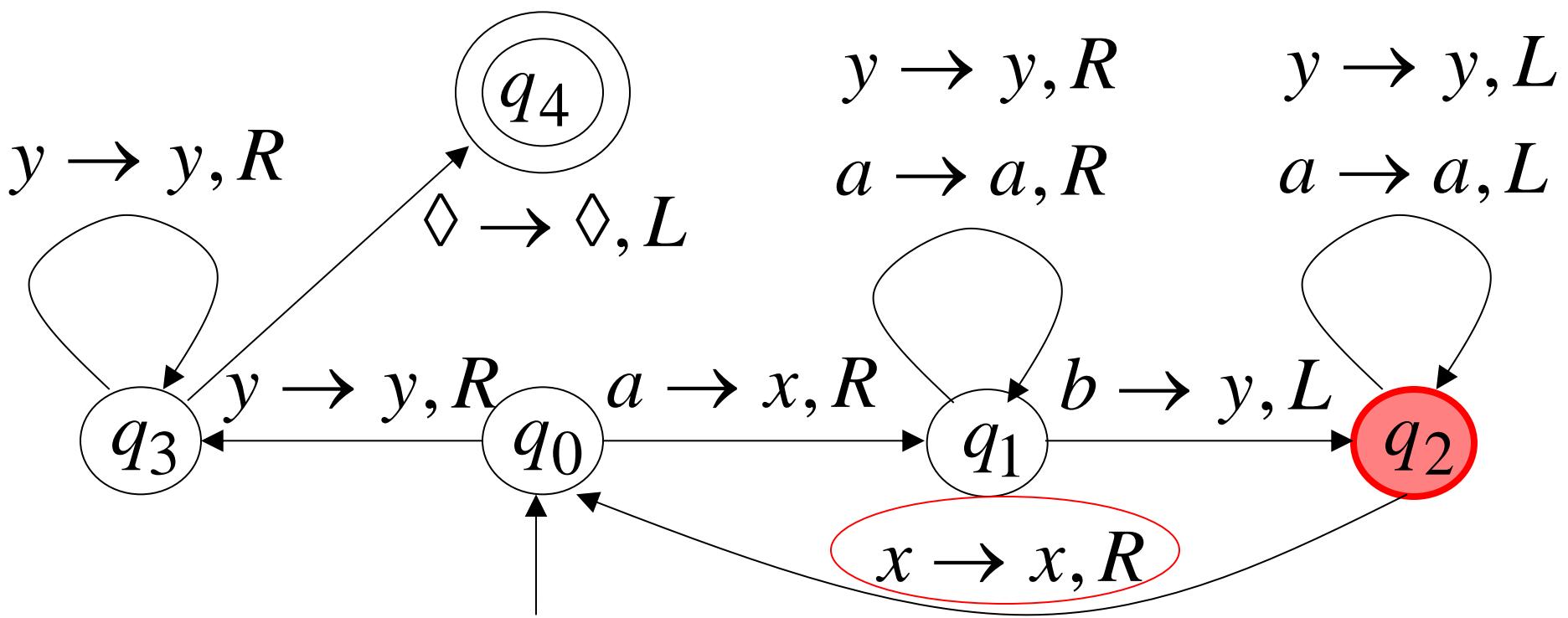
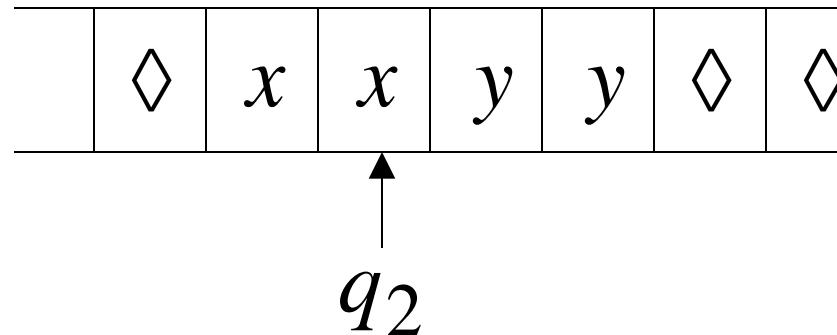
Time 7



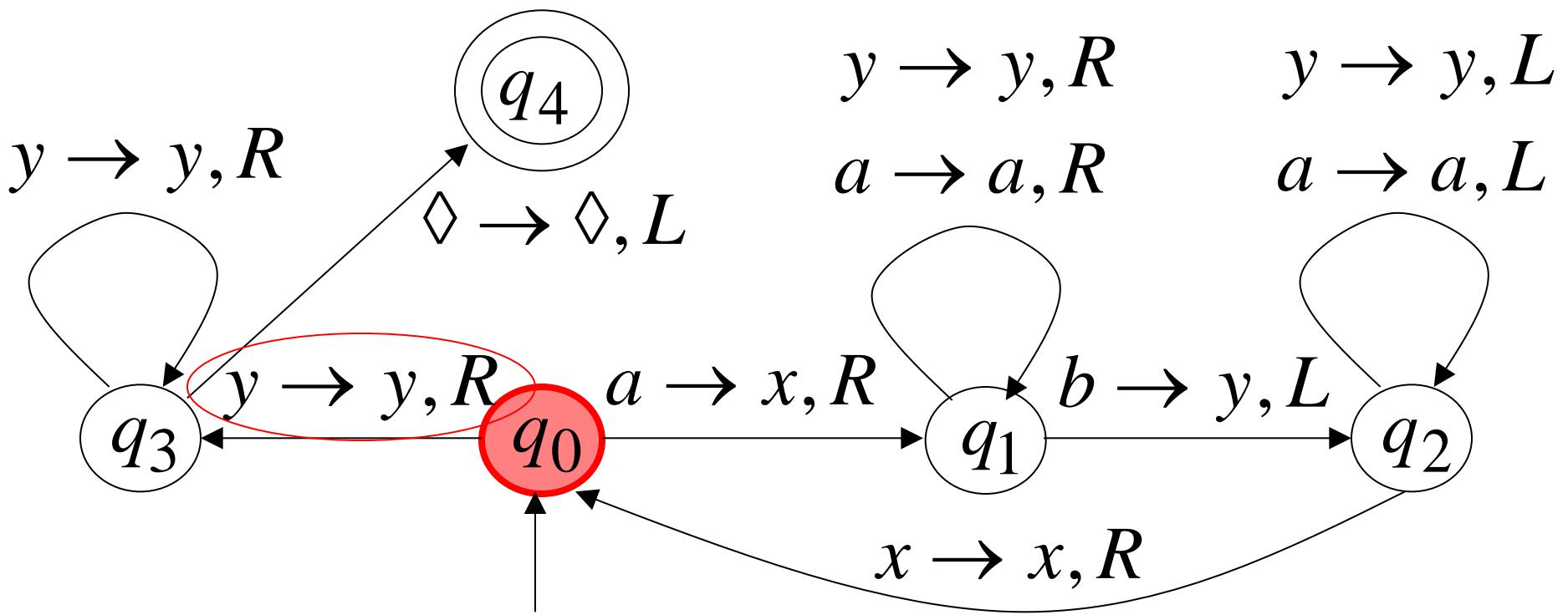
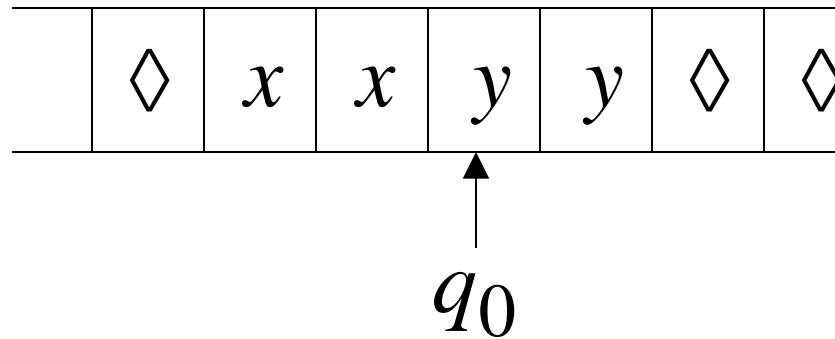
Time 8



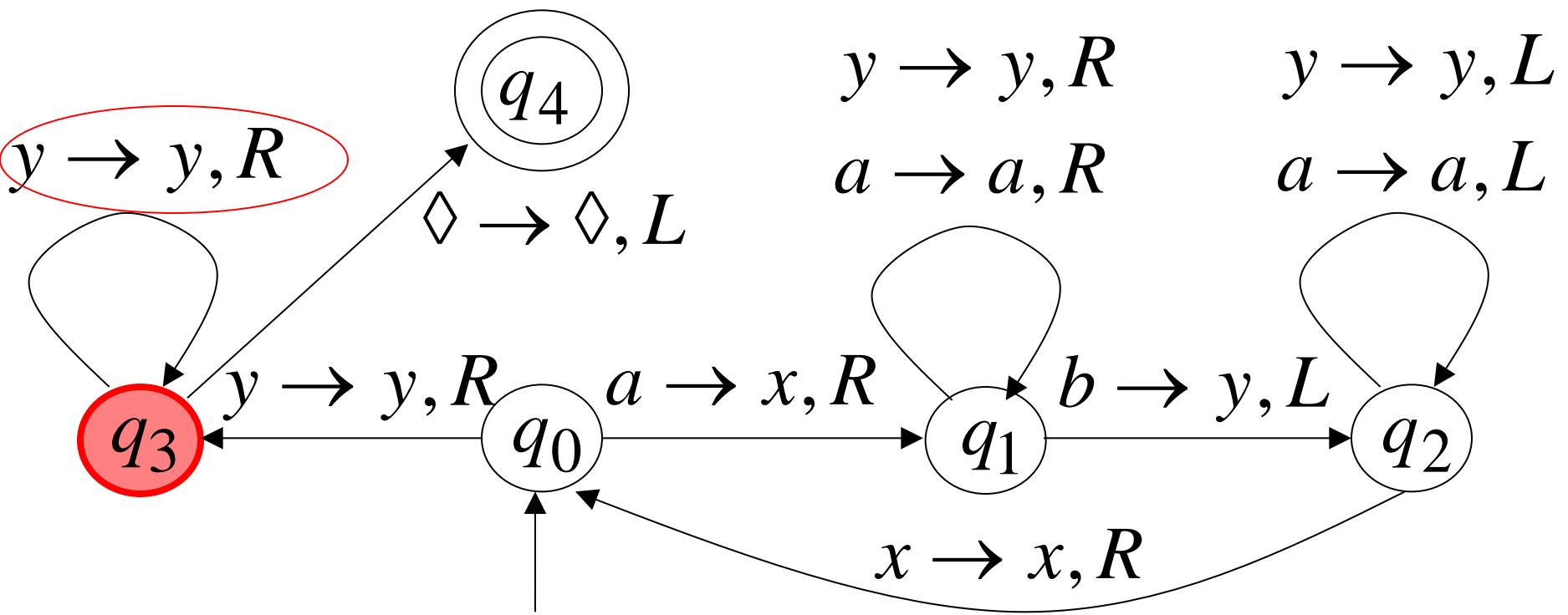
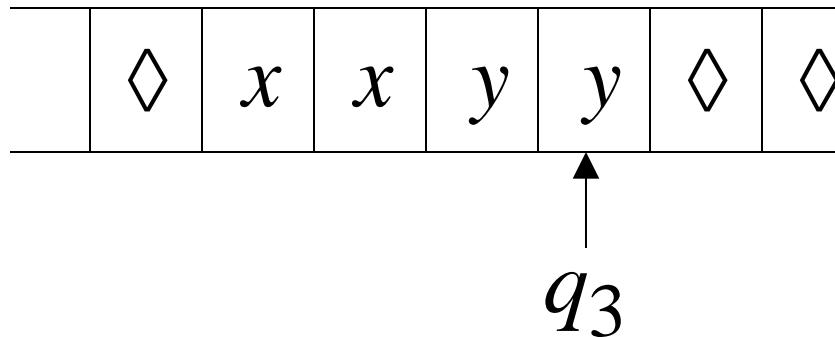
Time 9



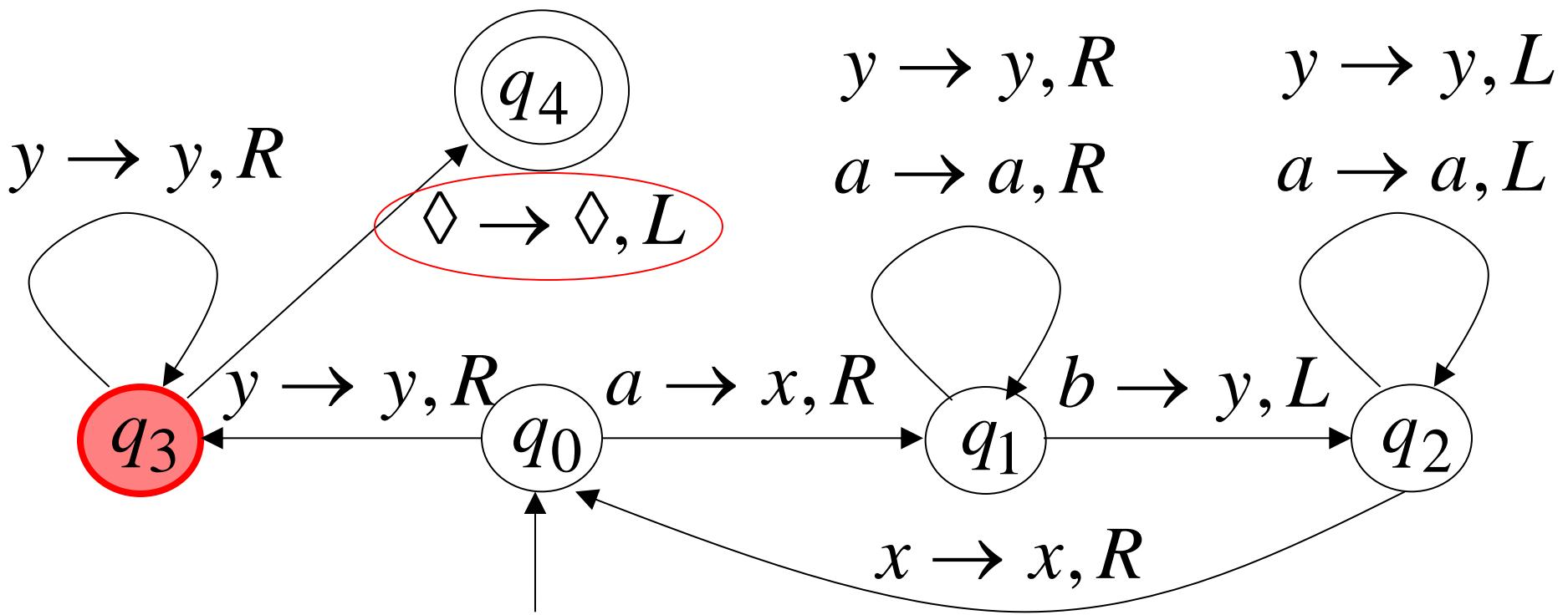
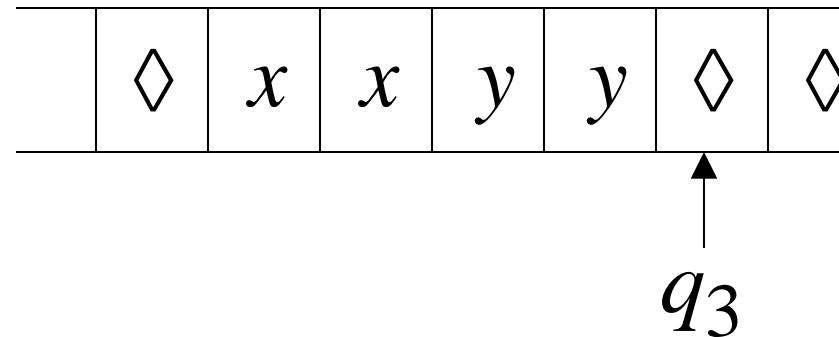
Time 10



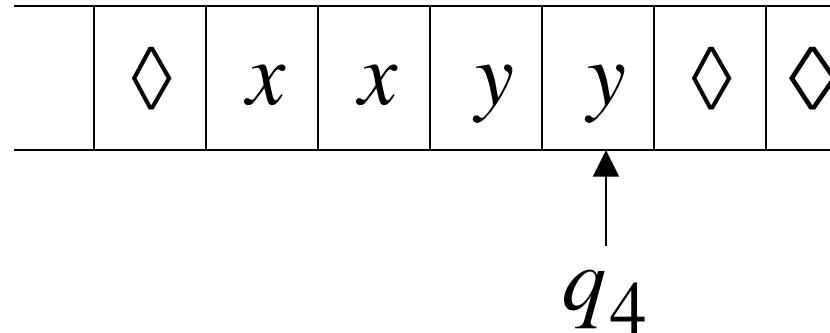
Time 11



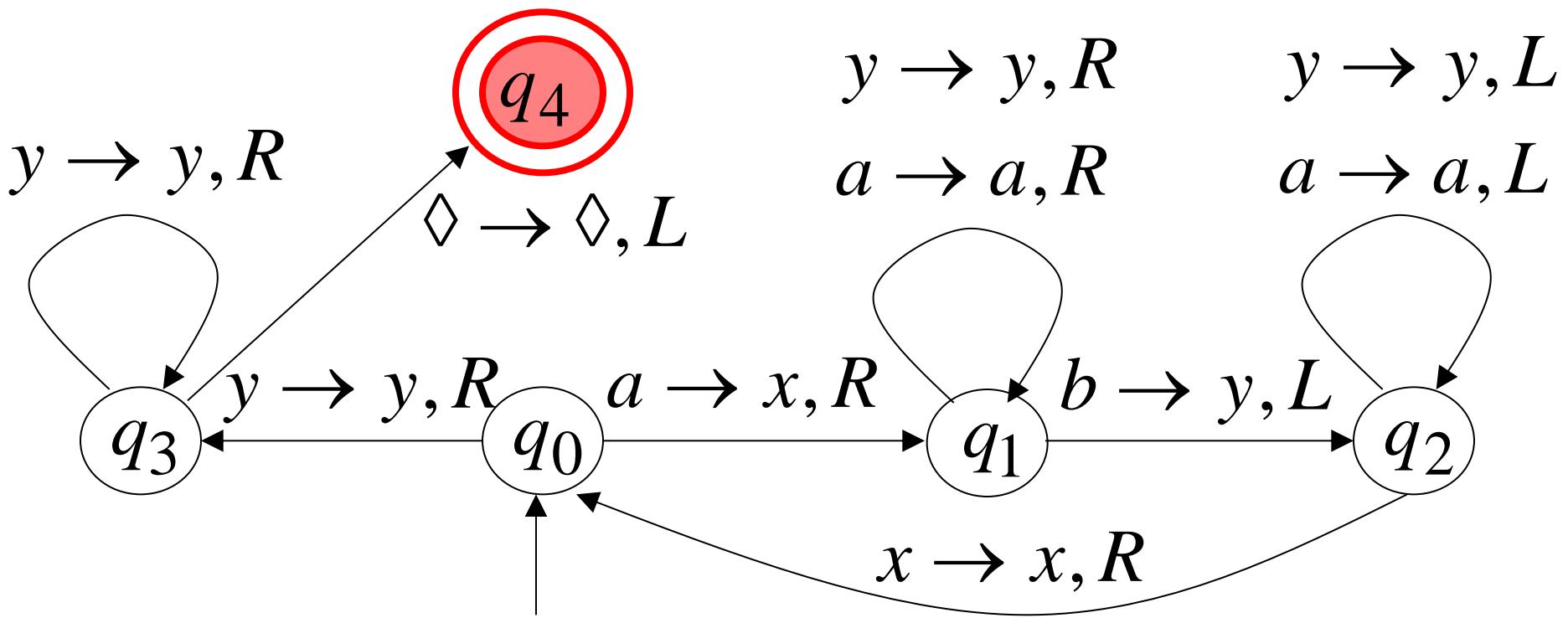
Time 12



Time 13



Halt & Accept



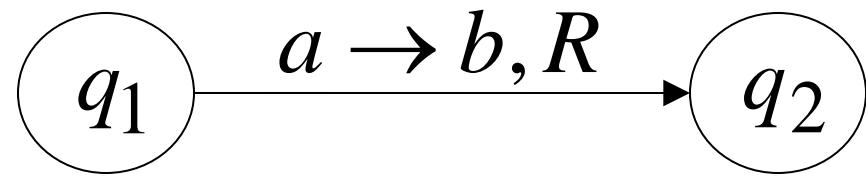
Observation:

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

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

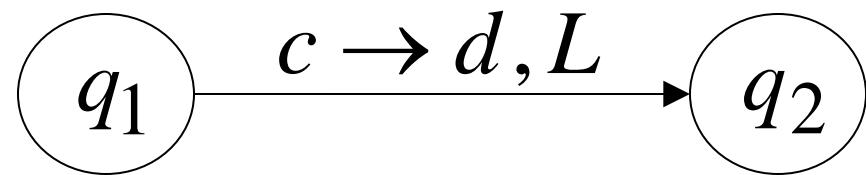
Formal Definitions for Turing Machines

Transition Function



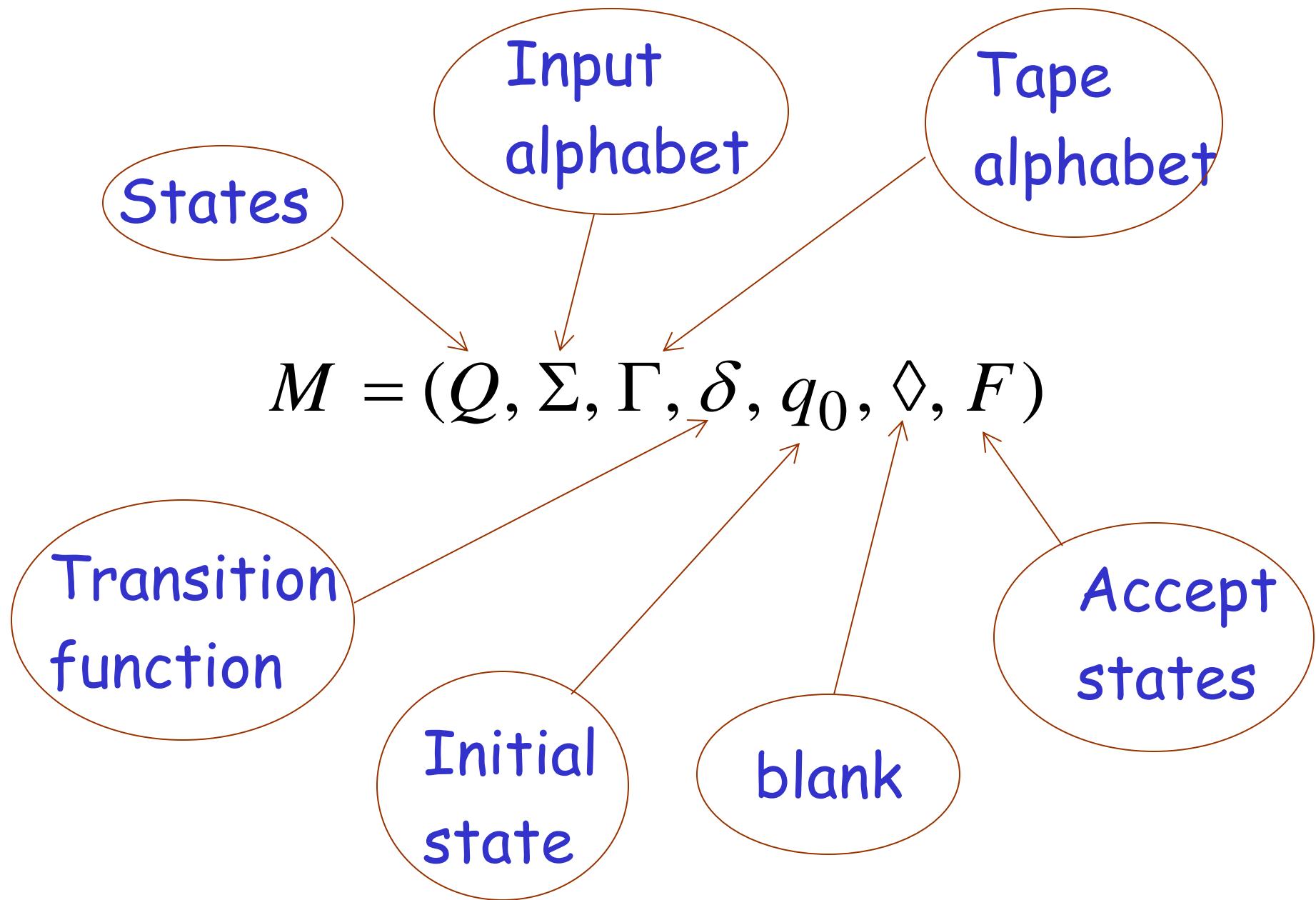
$$\delta(q_1, a) = (q_2, b, R)$$

Transition Function

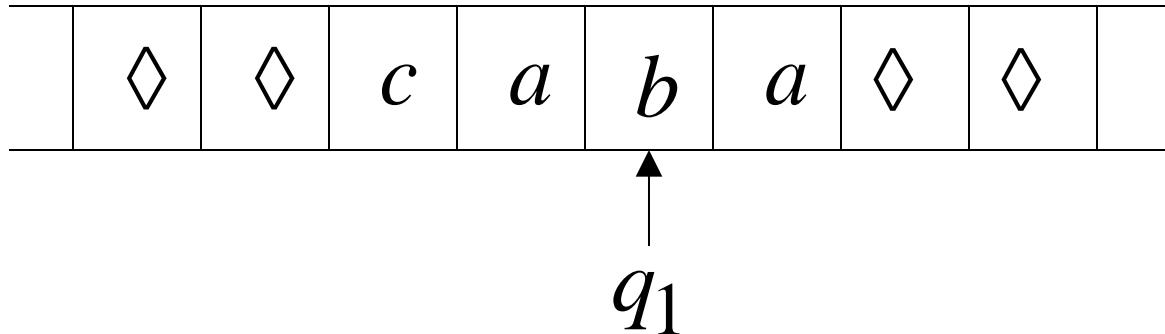


$$\delta(q_1, c) = (q_2, d, L)$$

Turing Machine:



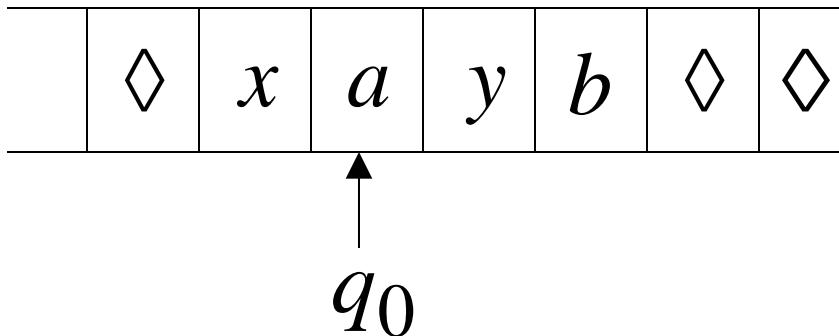
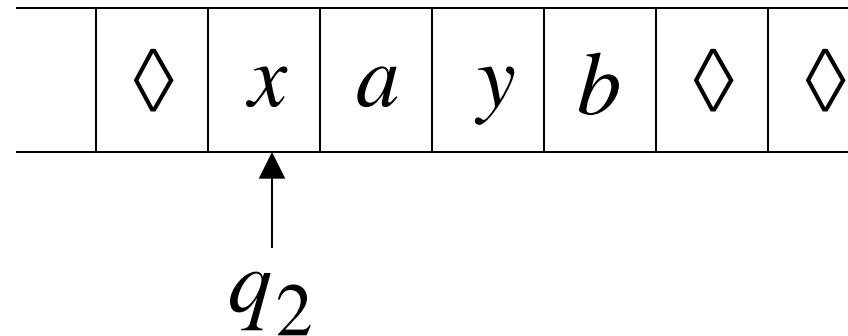
Configuration



Instantaneous description: $ca\ q_1\ ba$

Time 4

Time 5

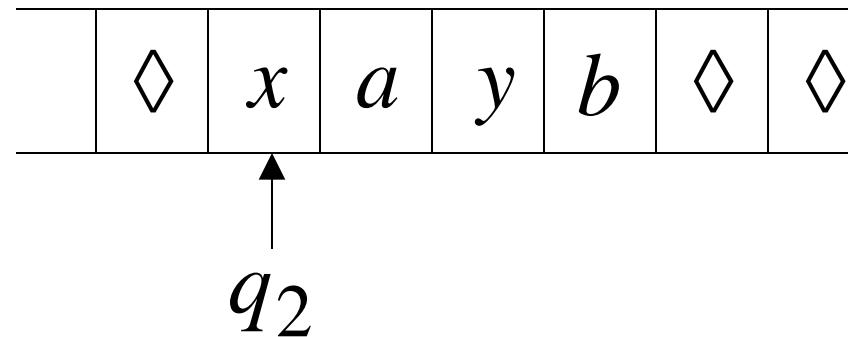


A Move:

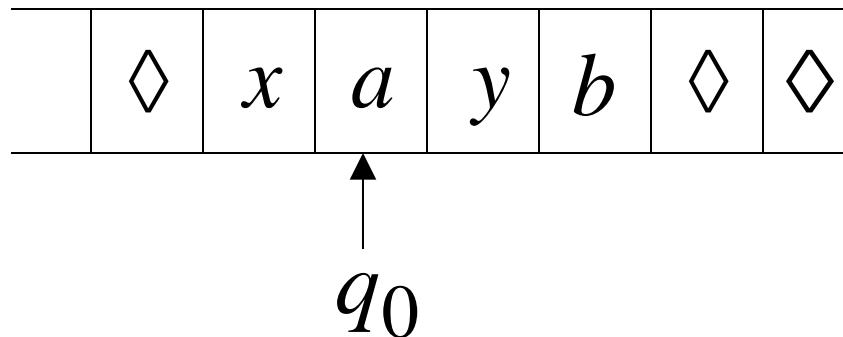
$$q_2 \ xayb \succ x q_0 \ ayb$$

(yields in one mode)

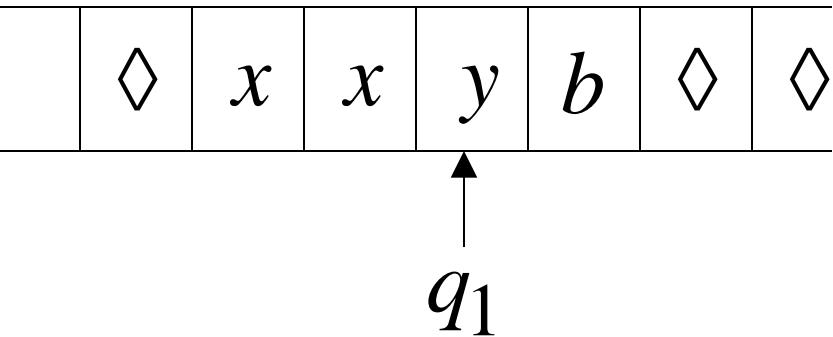
Time 4



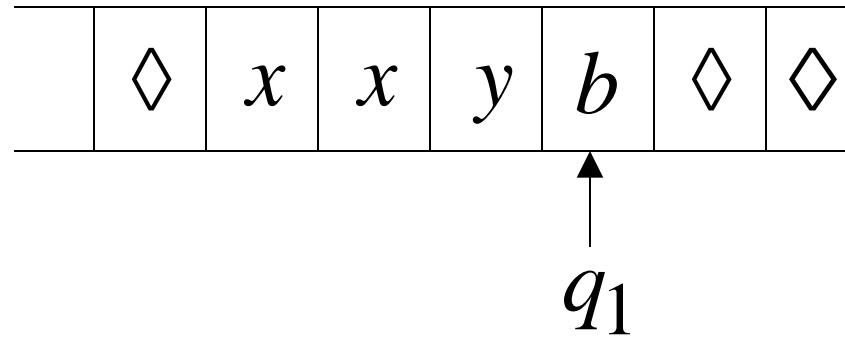
Time 5



Time 6



Time 7



A computation

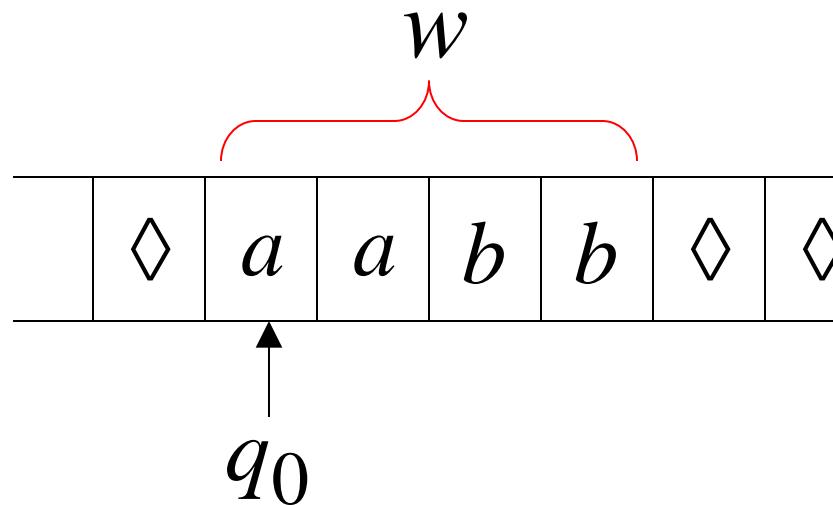
$q_2 \ xayb \succ x \ q_0 \ ayb \succ xx \ q_1 \ yb \succ xxy \ q_1 \ b$

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

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

Initial configuration: $q_0 \ w$

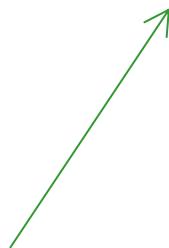
Input string



The Accepted Language

For any Turing Machine M

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



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

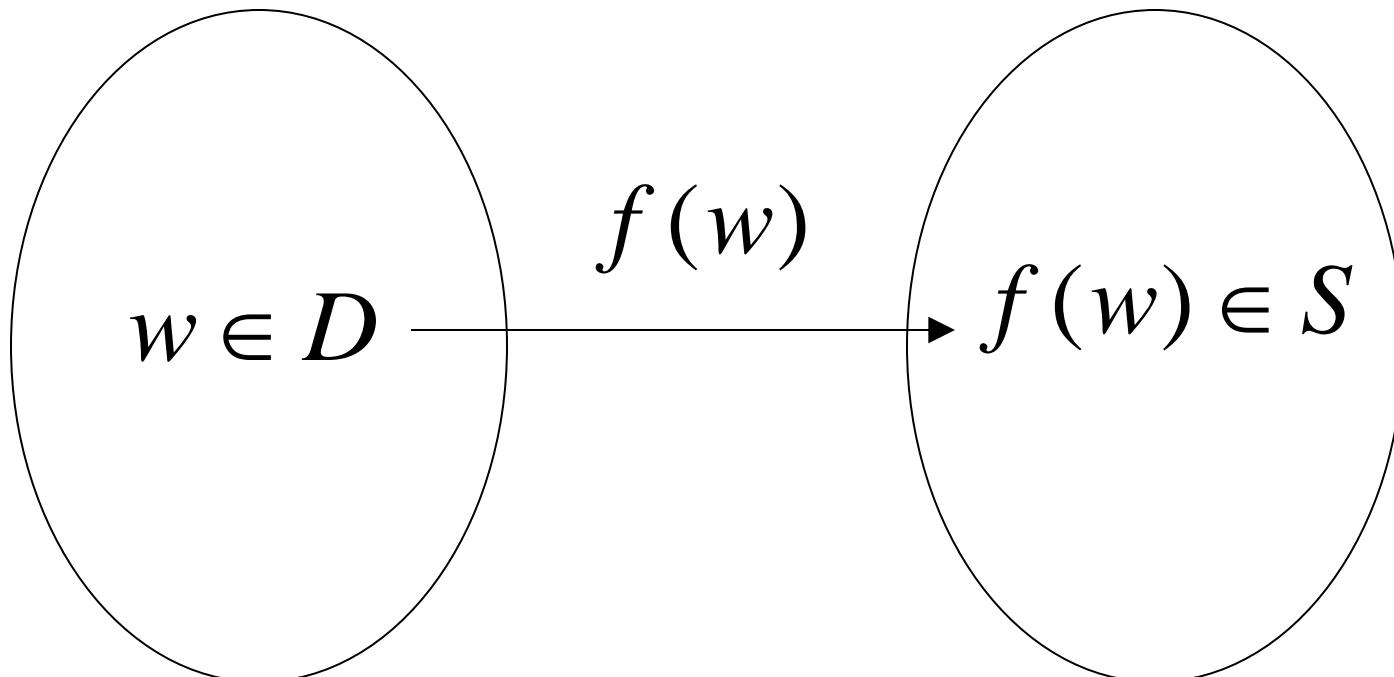
A function

$f(w)$

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

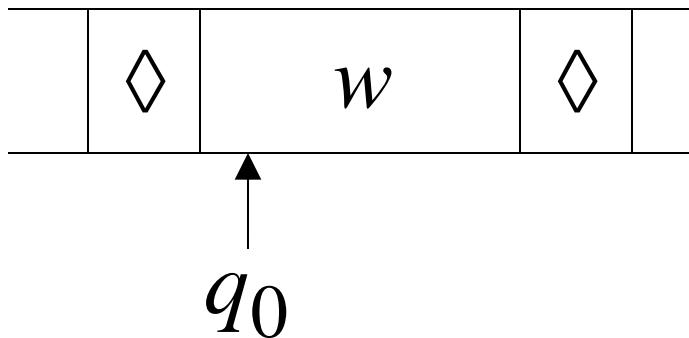
We prefer **unary** representation:

easier to manipulate with Turing machines

Definition:

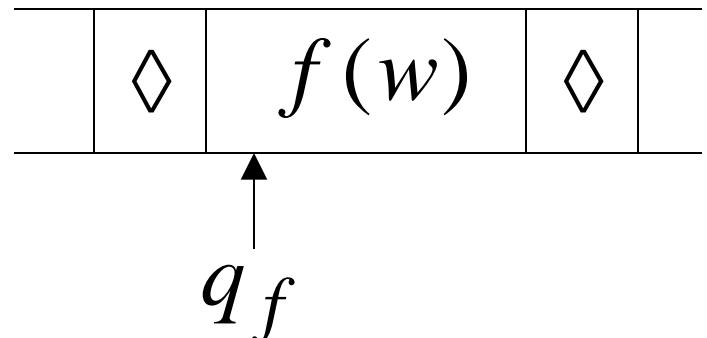
A function f is computable if there is a Turing Machine M such that:

Initial configuration



initial state

Final configuration

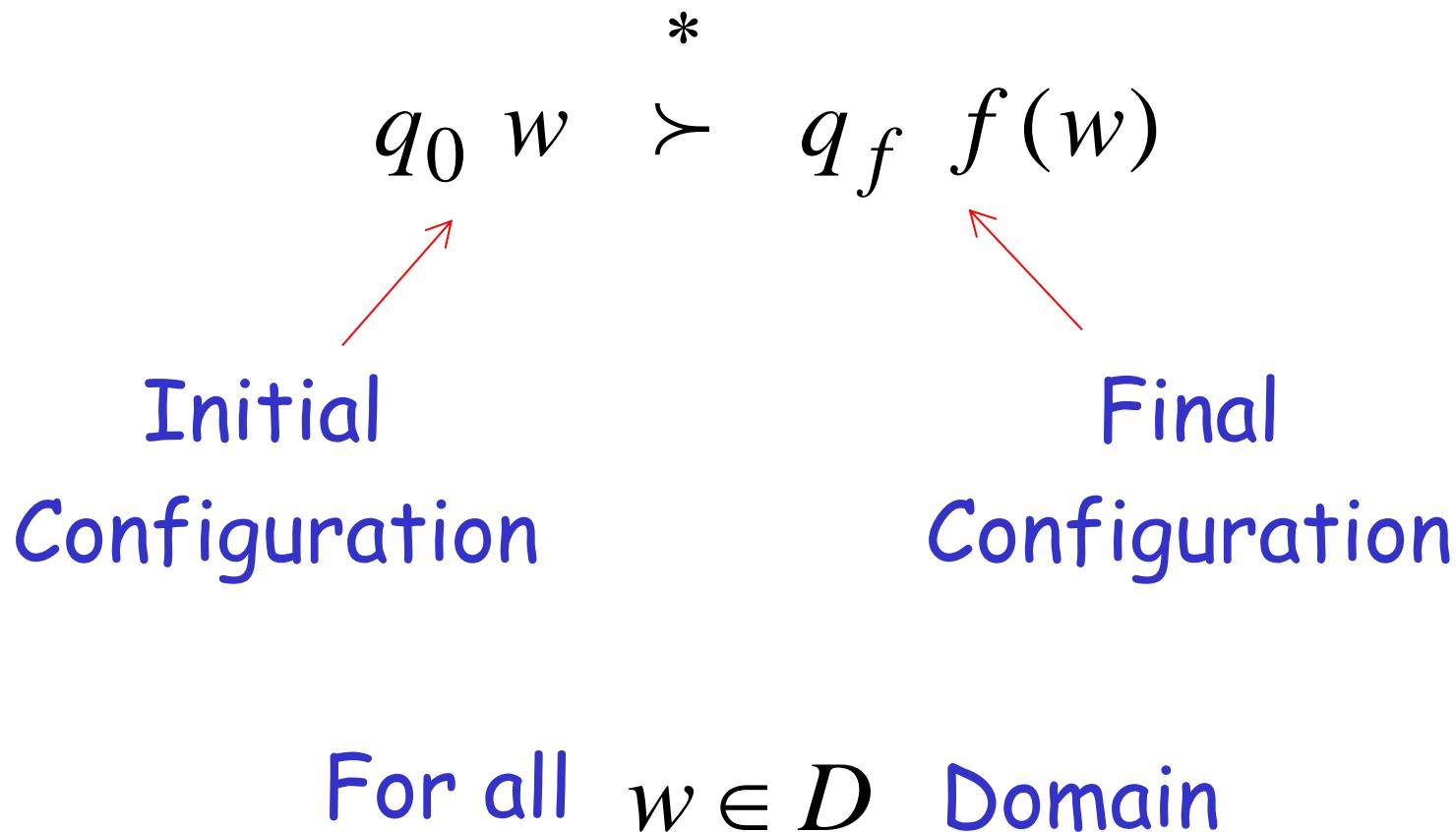


accept state

For all $w \in D$ Domain

In other words:

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



Example

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

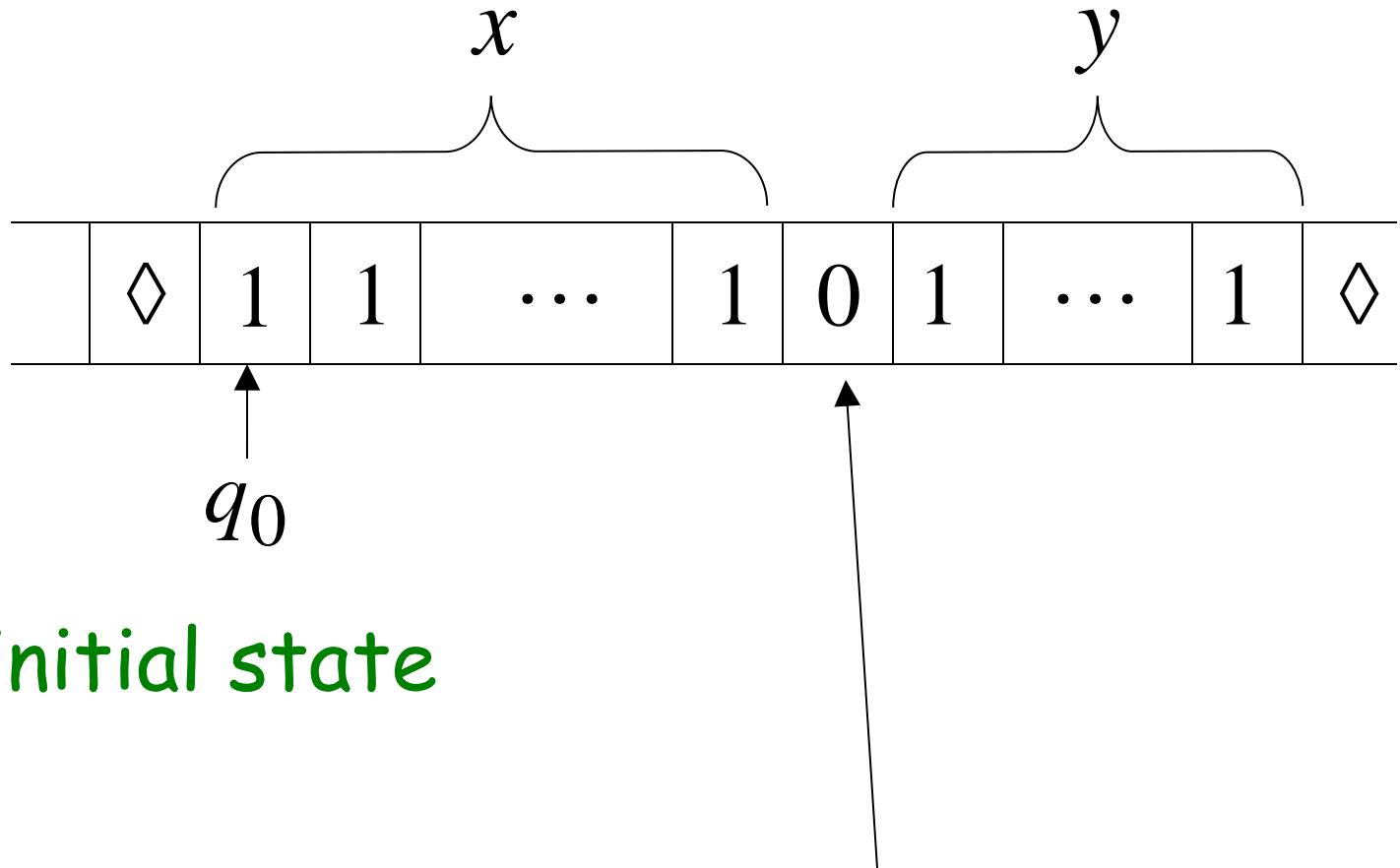
x, y are integers

Turing Machine:

Input string: $x0y$ unary

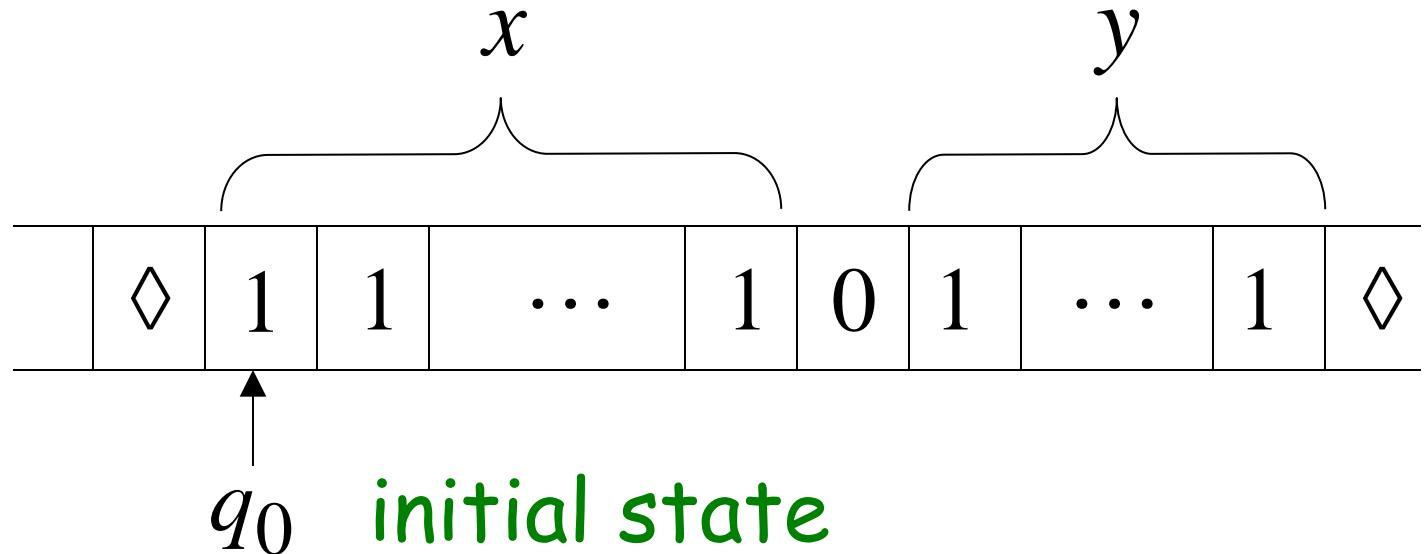
Output string: $xy0$ unary

Start

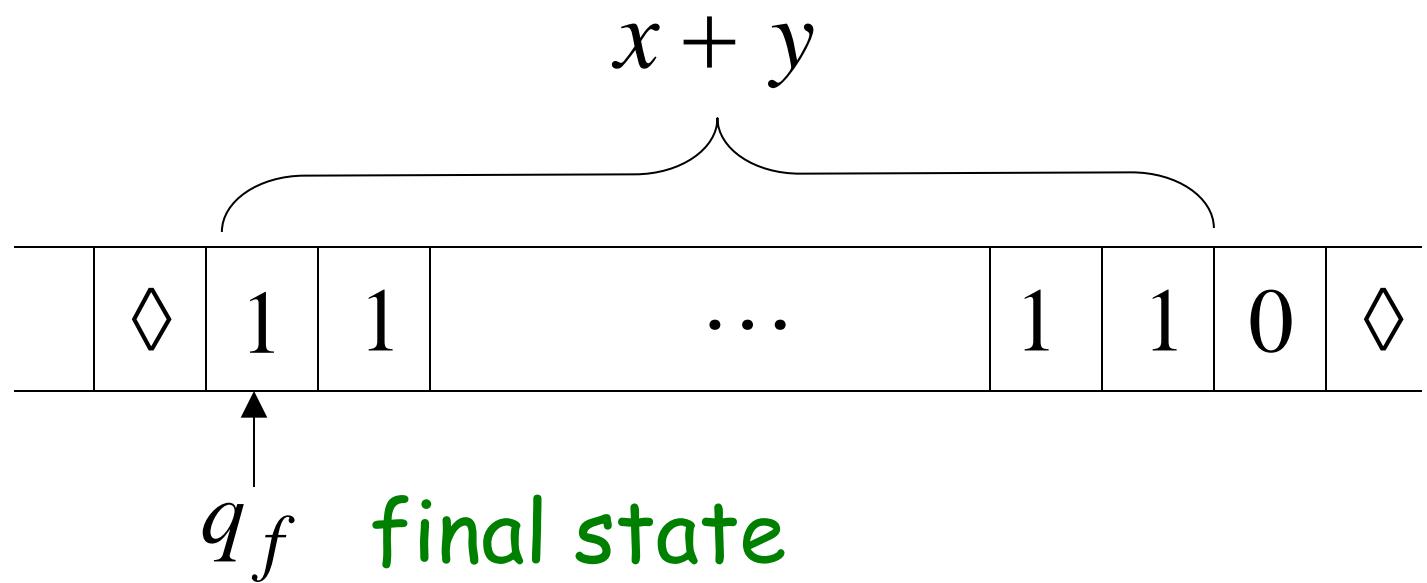


The 0 is the delimiter that separates the two numbers

Start

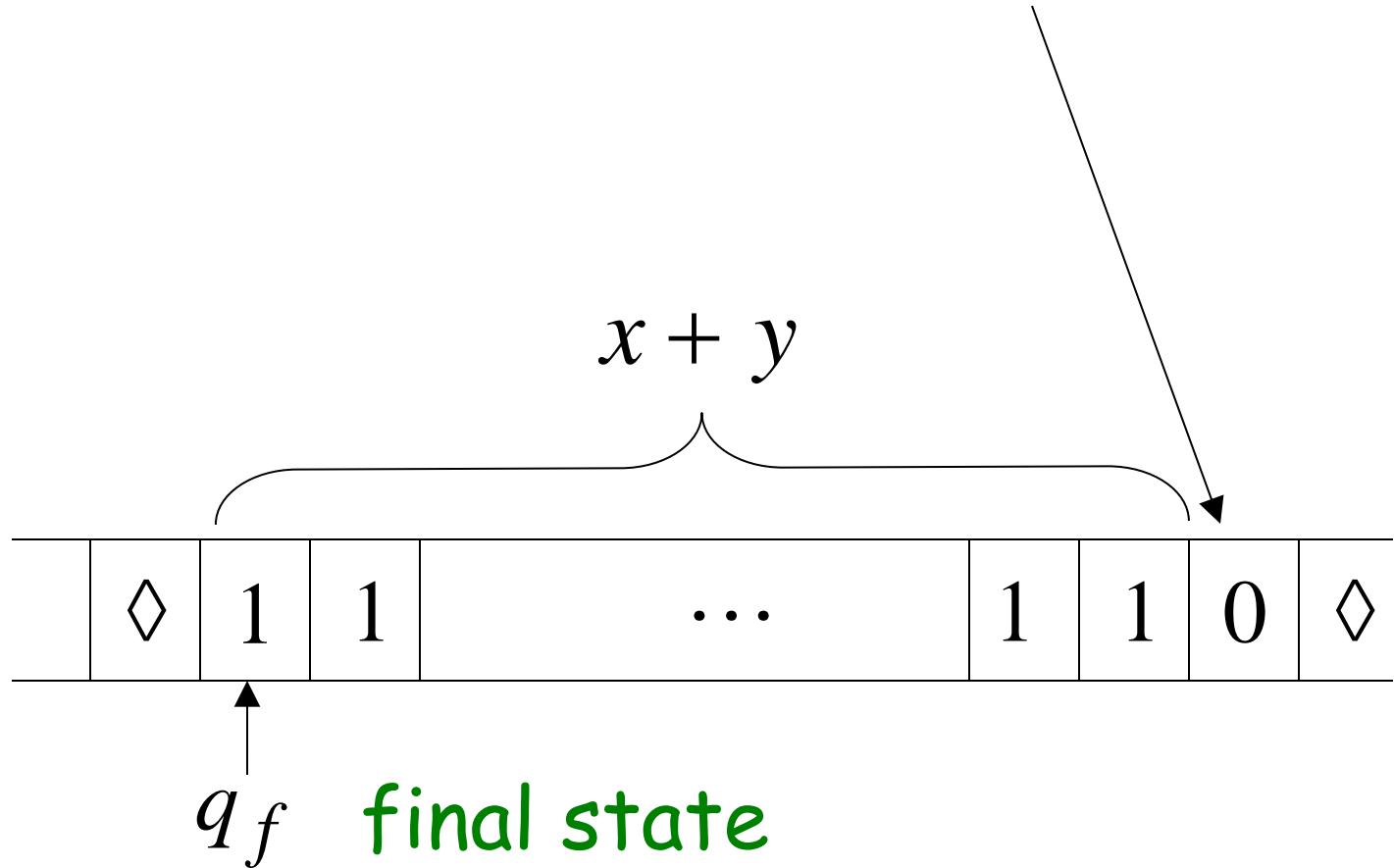


Finish

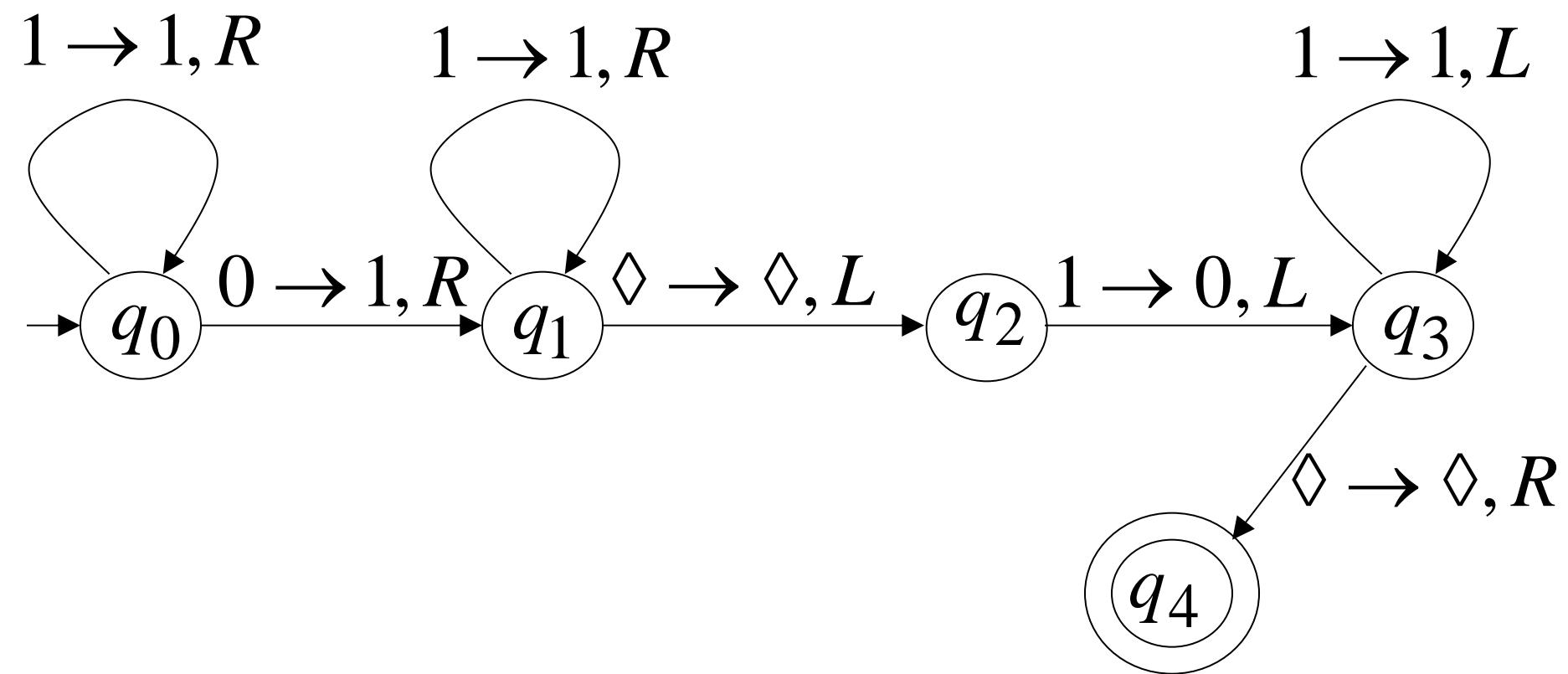


The 0 here helps when we use
the result for other operations

Finish



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



Execution Example:

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

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

Time 0

		x		y	
		◊	1	1	0

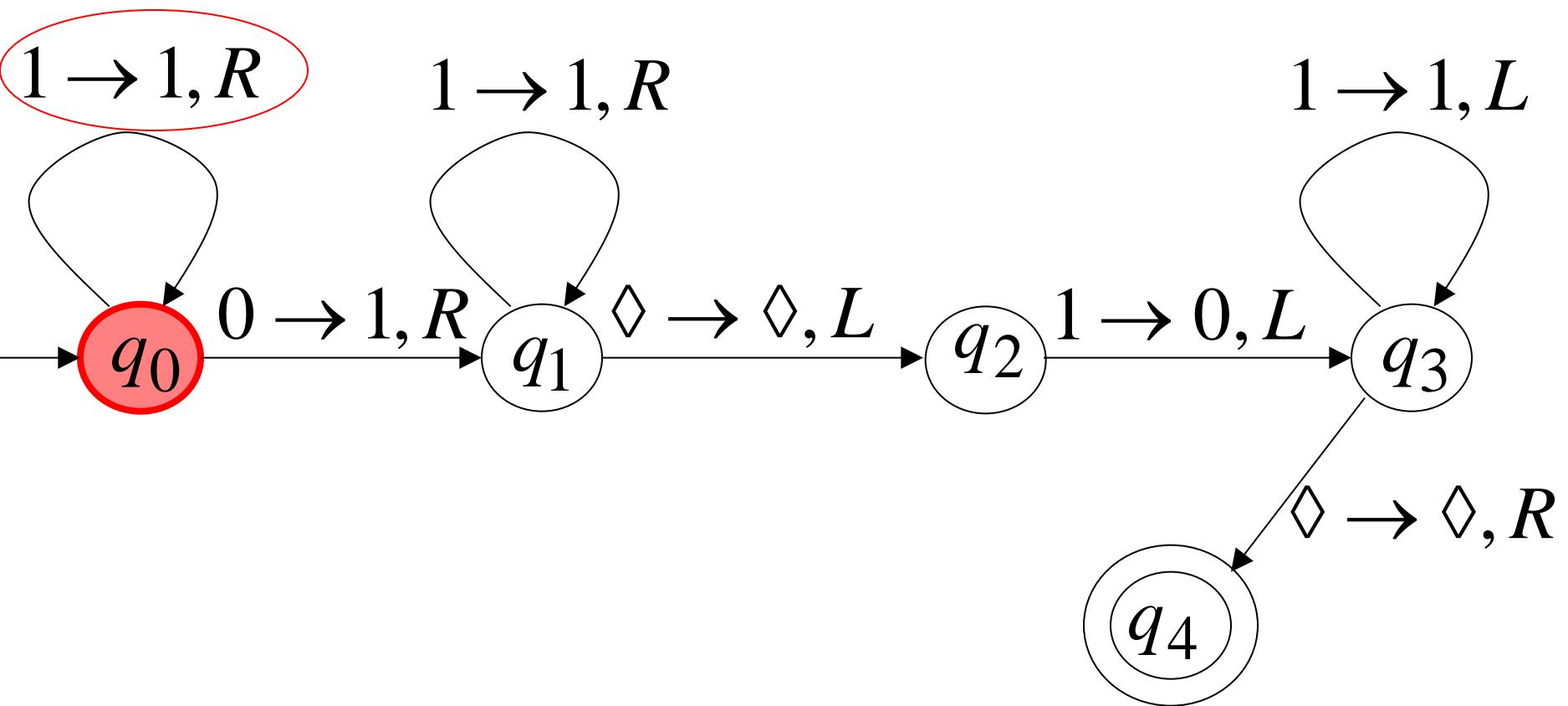
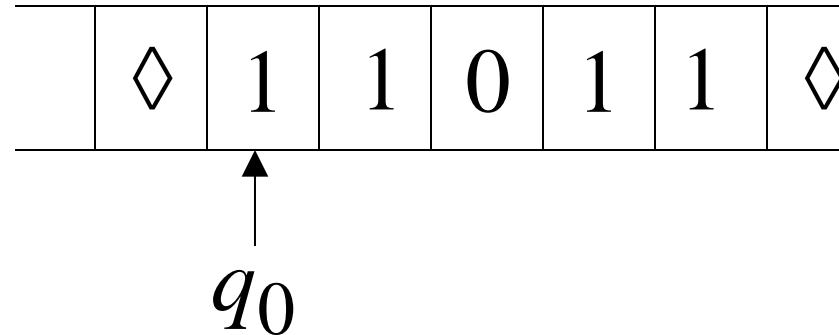
\uparrow
 q_0

Final Result

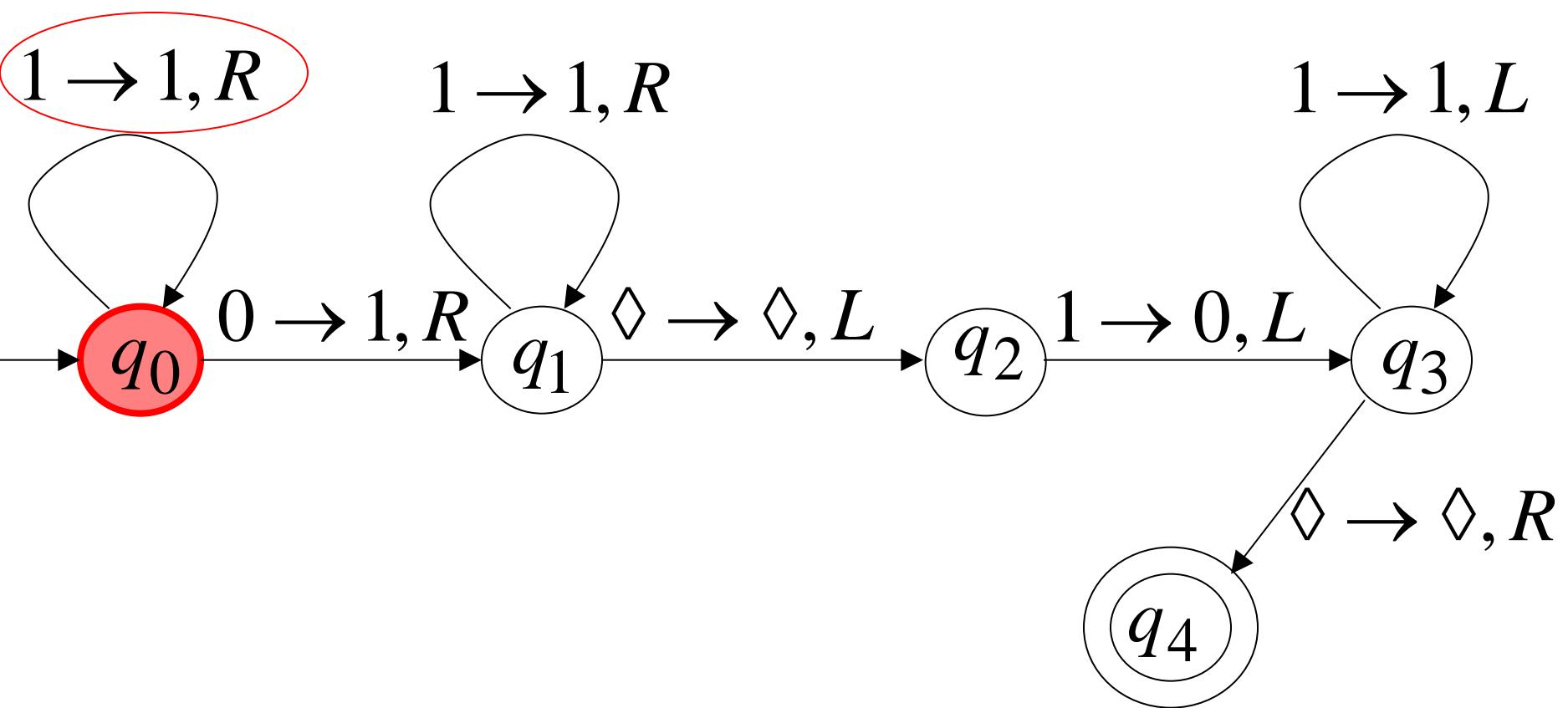
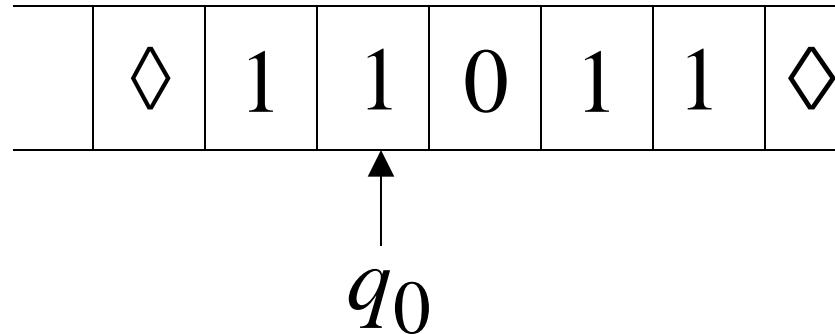
		$x + y$	
		◊	1

\uparrow
 q_4

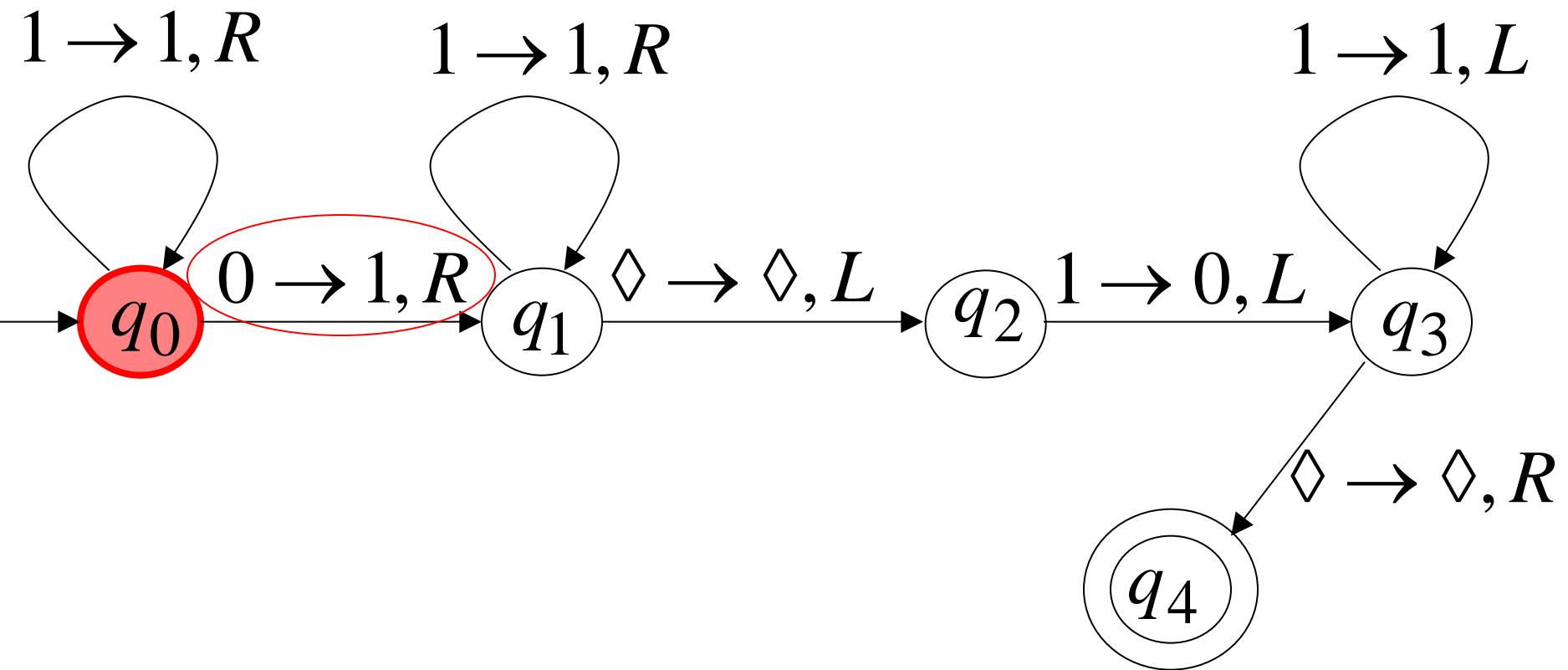
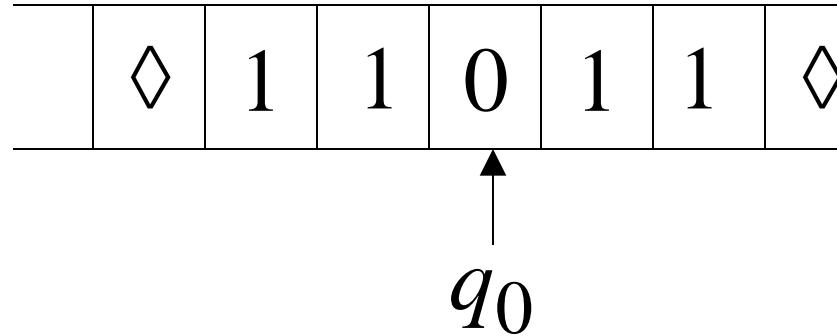
Time 0



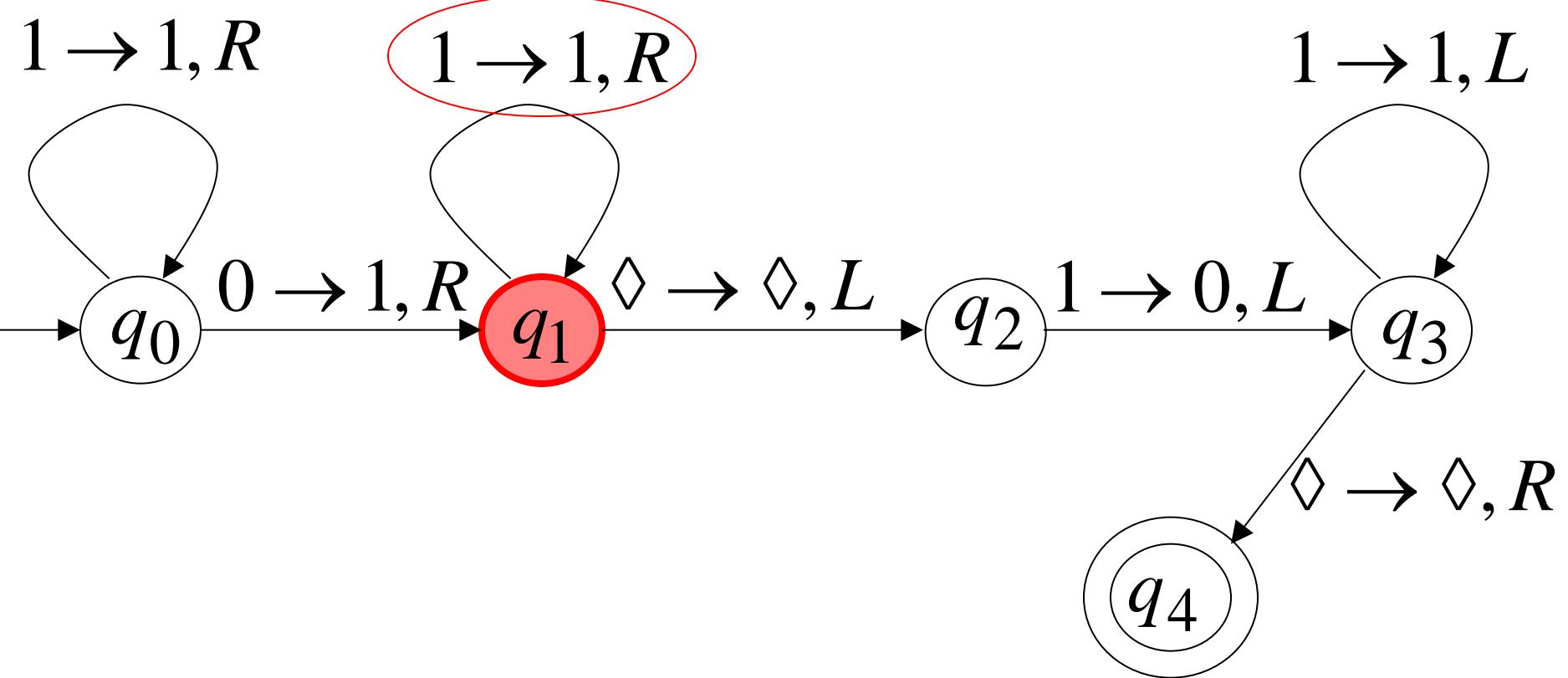
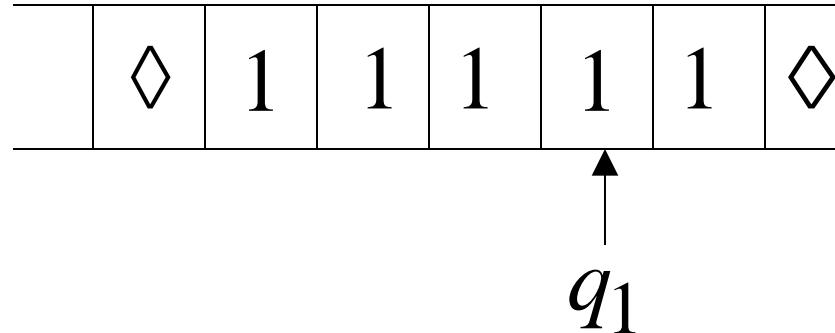
Time 1



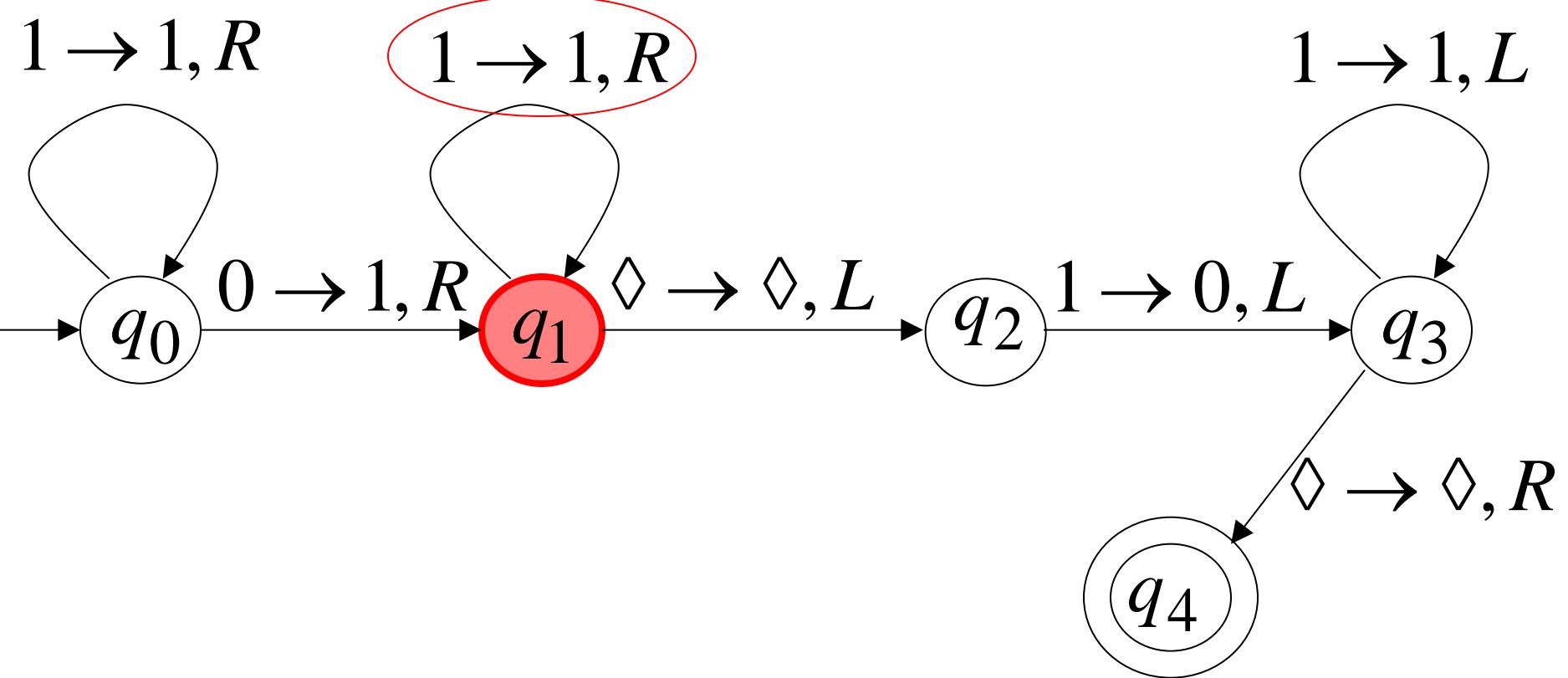
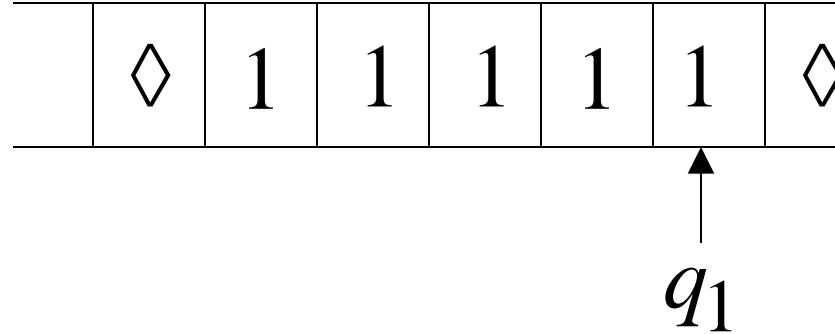
Time 2



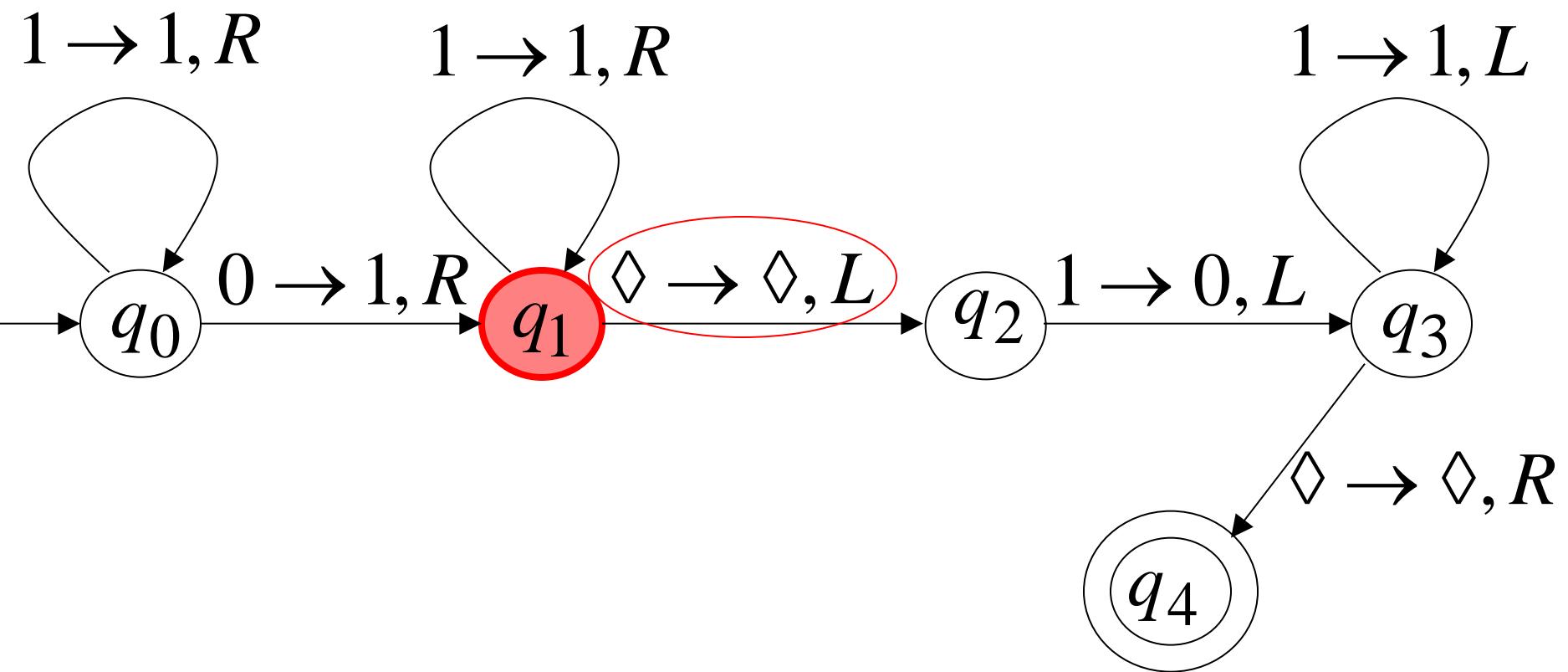
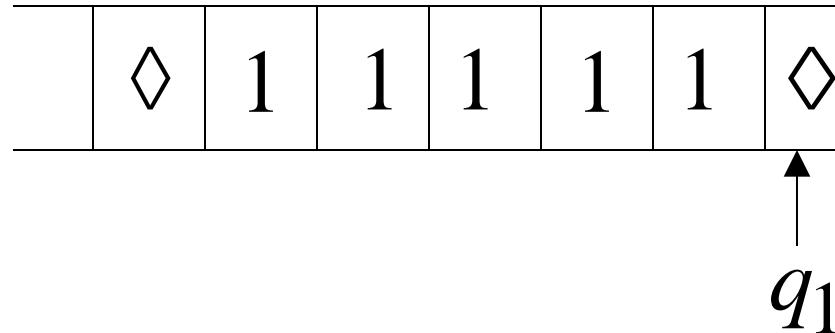
Time 3



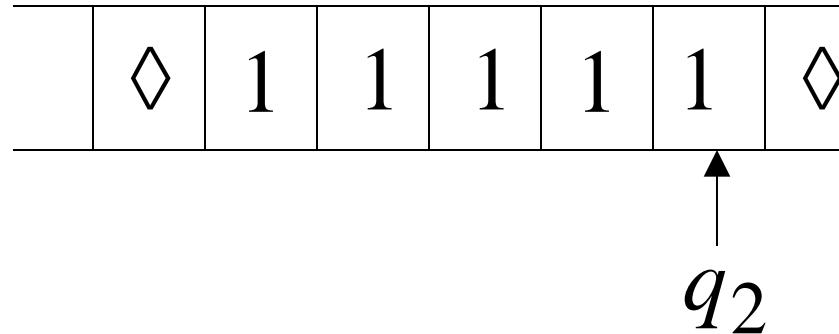
Time 4



Time 5



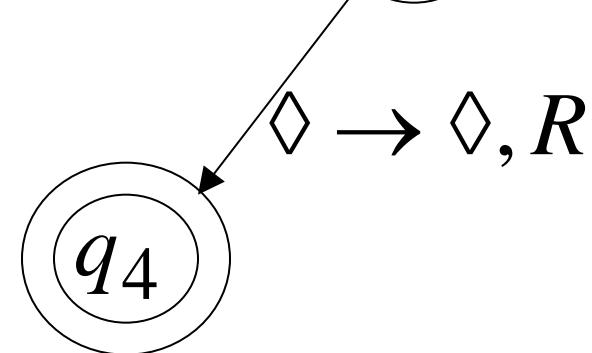
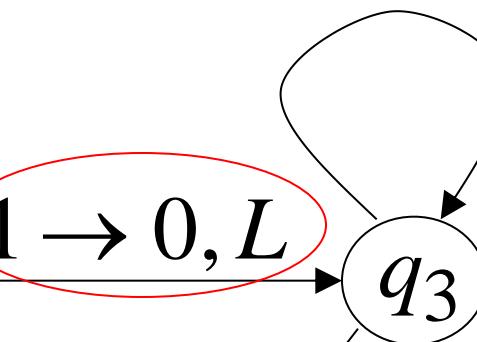
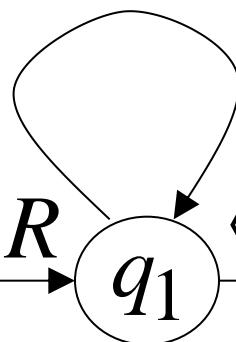
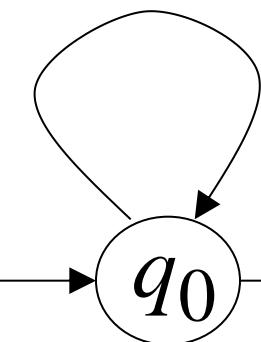
Time 6



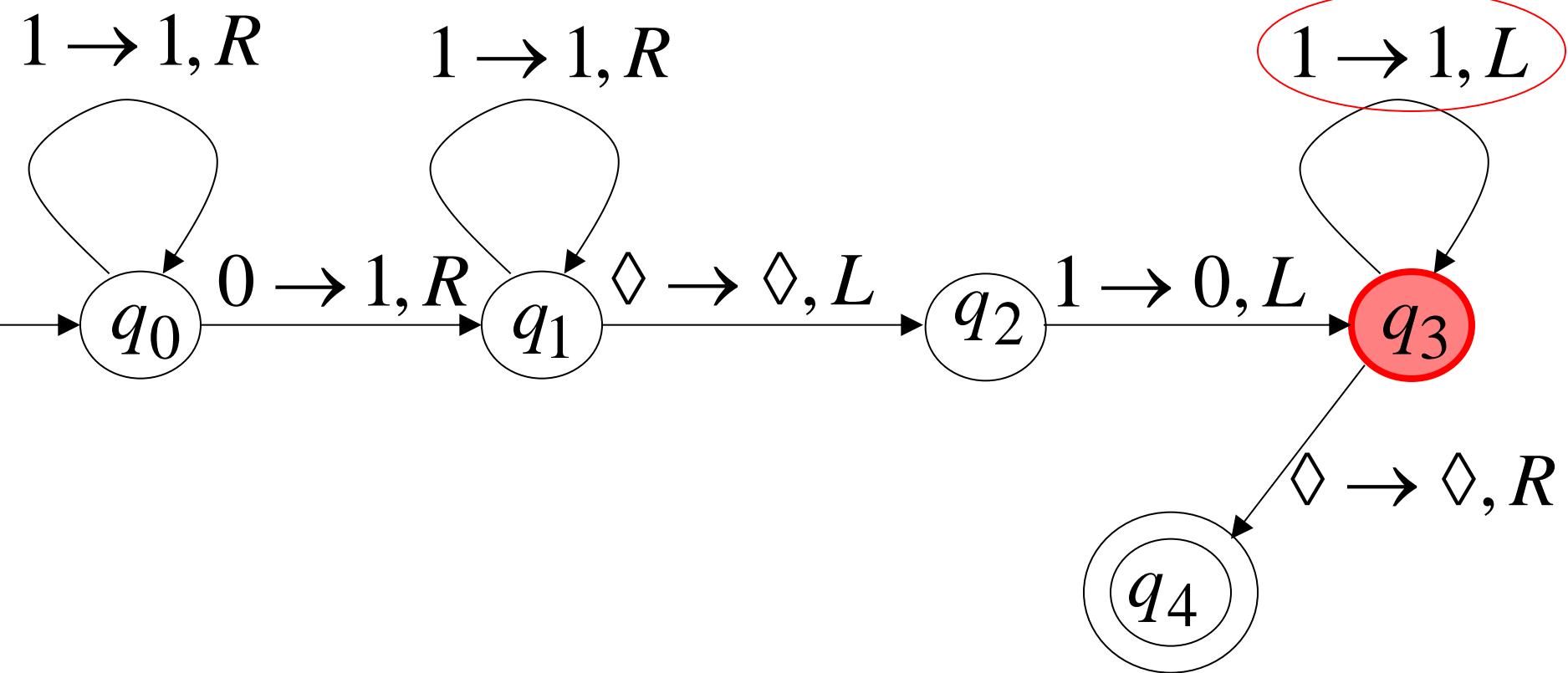
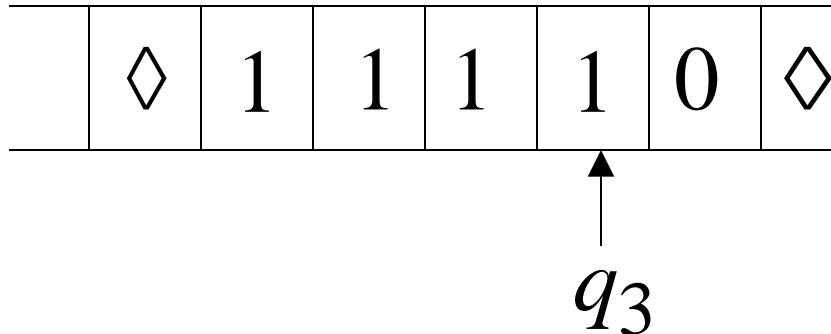
$1 \rightarrow 1, R$

$1 \rightarrow 1, R$

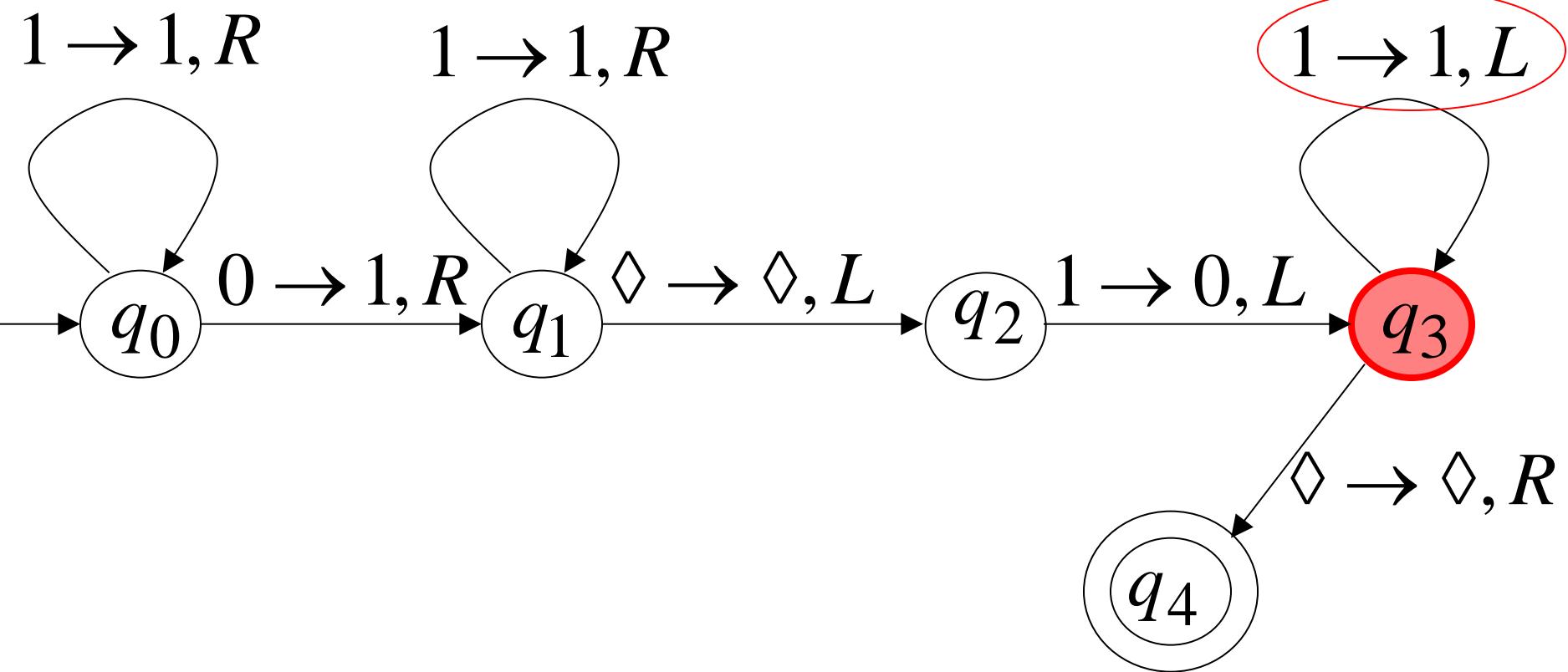
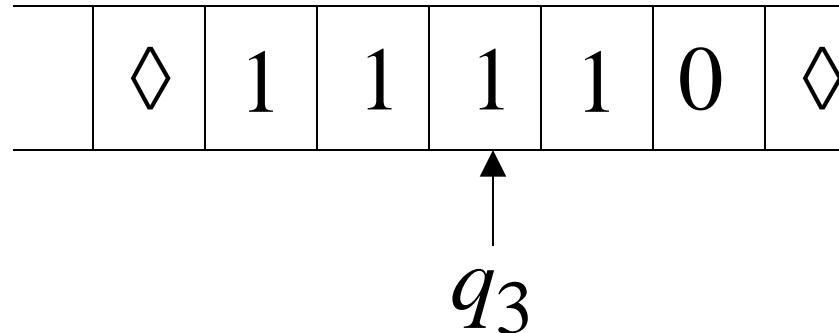
$1 \rightarrow 1, L$



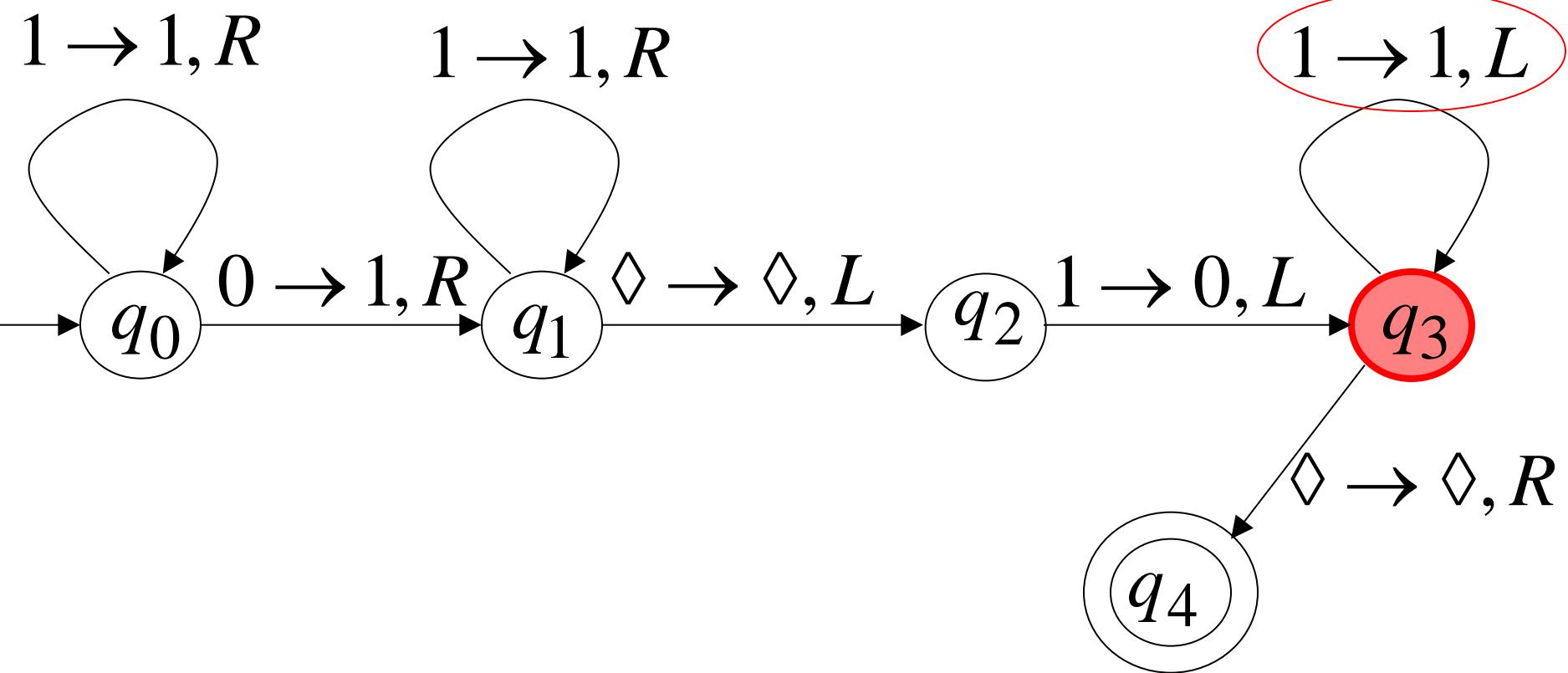
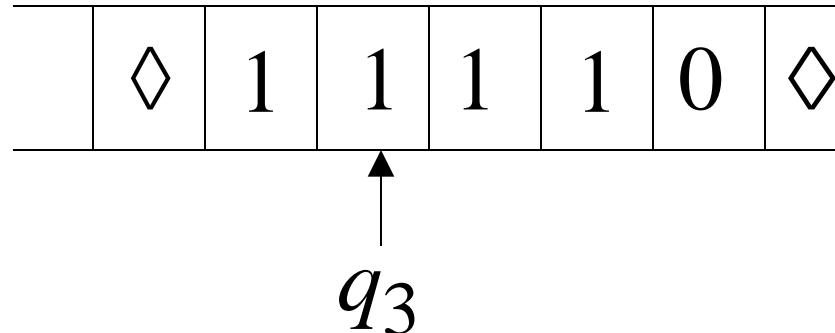
Time 7



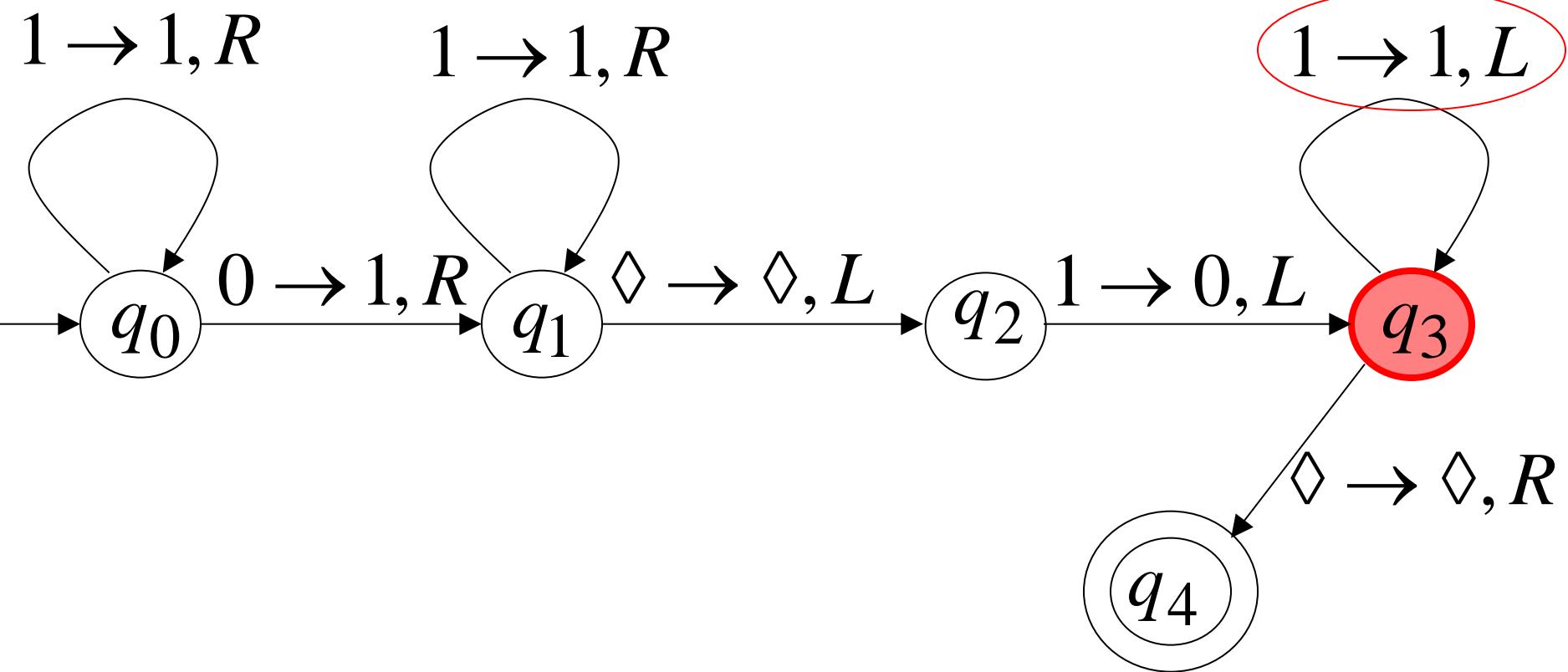
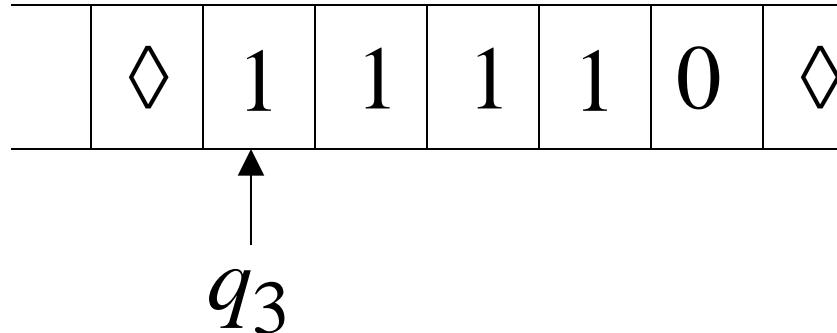
Time 8



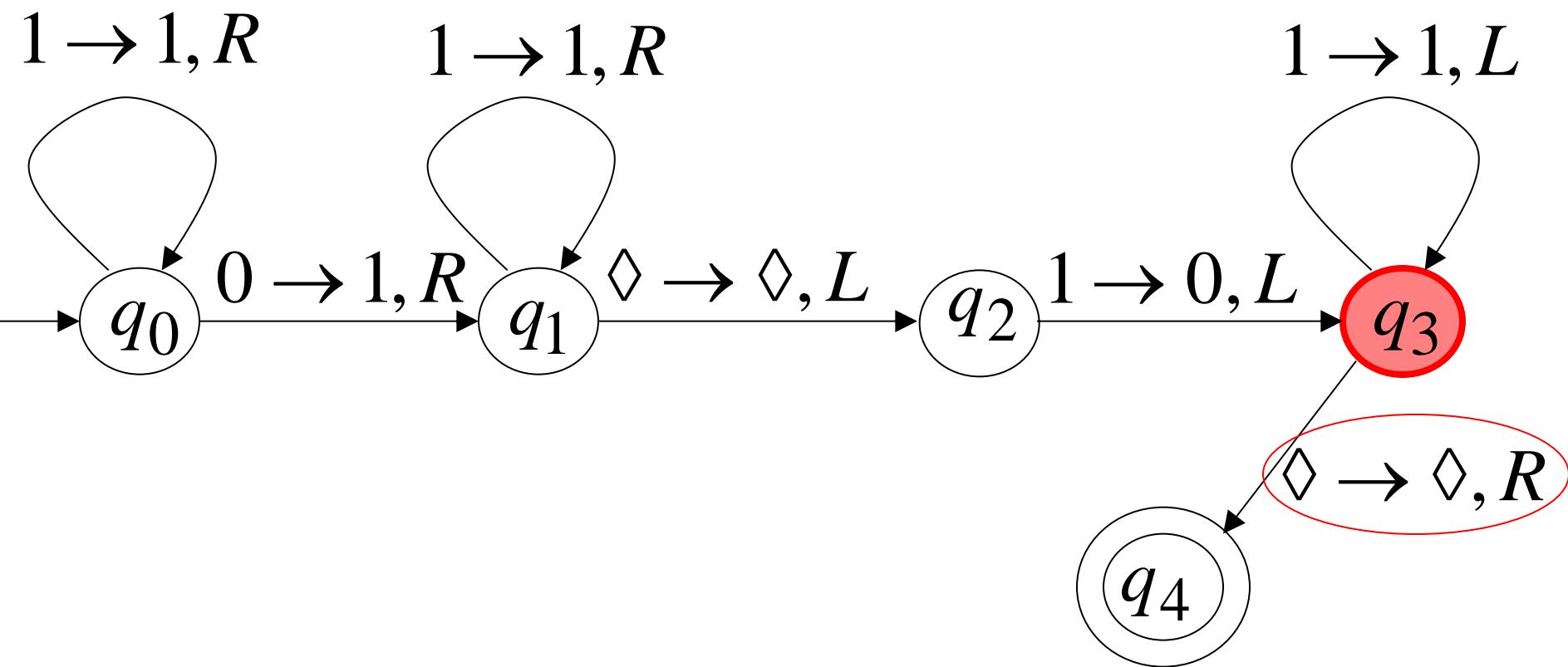
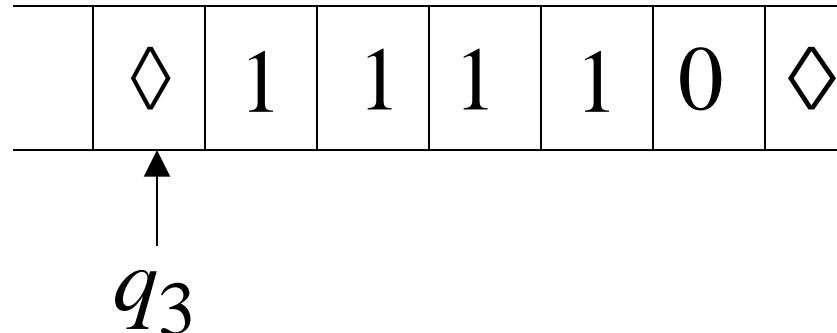
Time 9



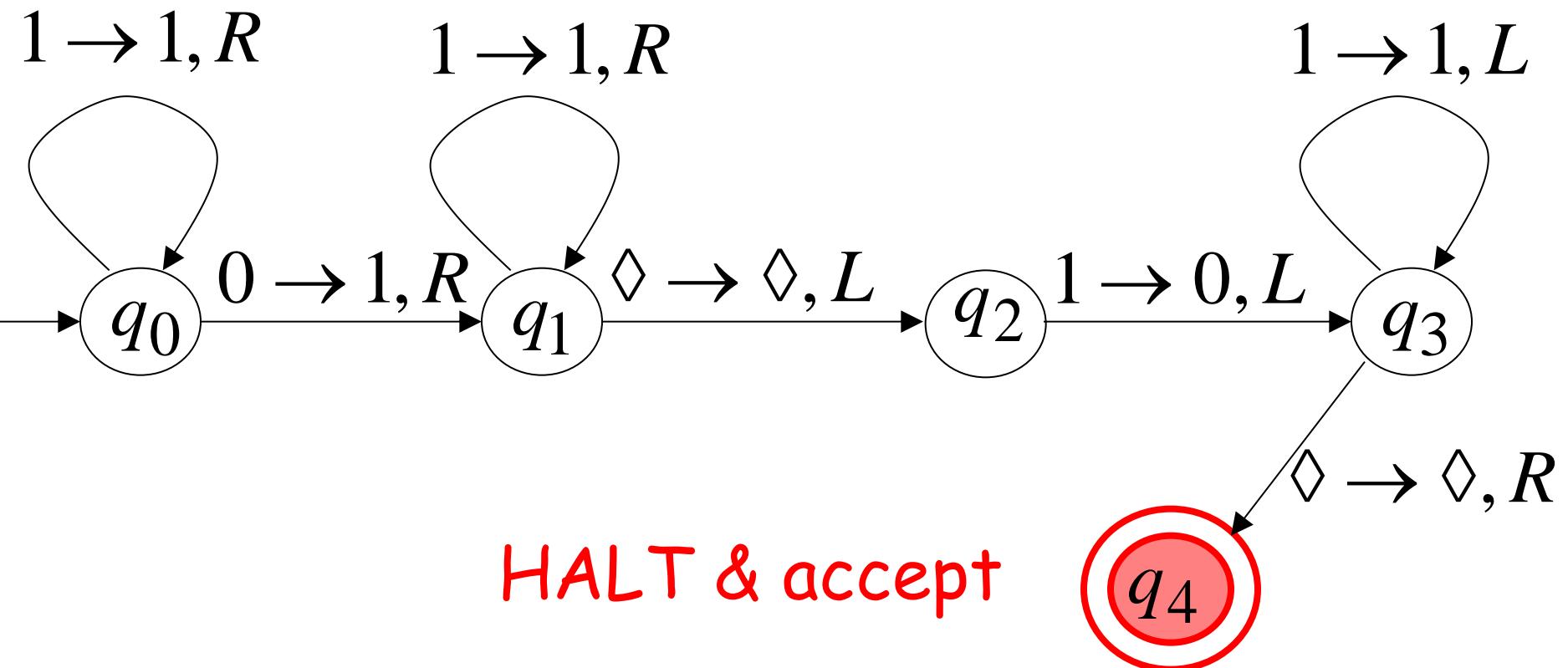
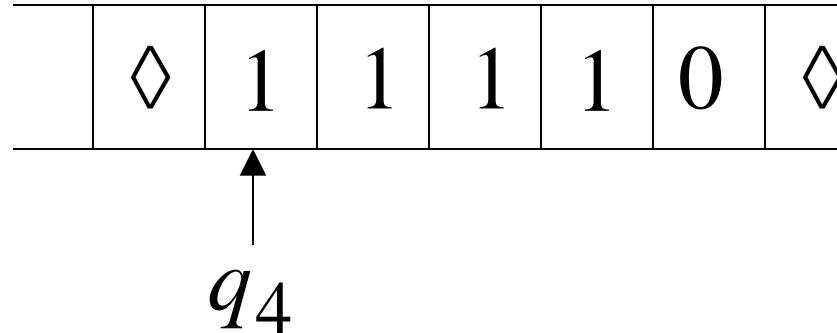
Time 10



Time 11



Time 12



Another Example

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

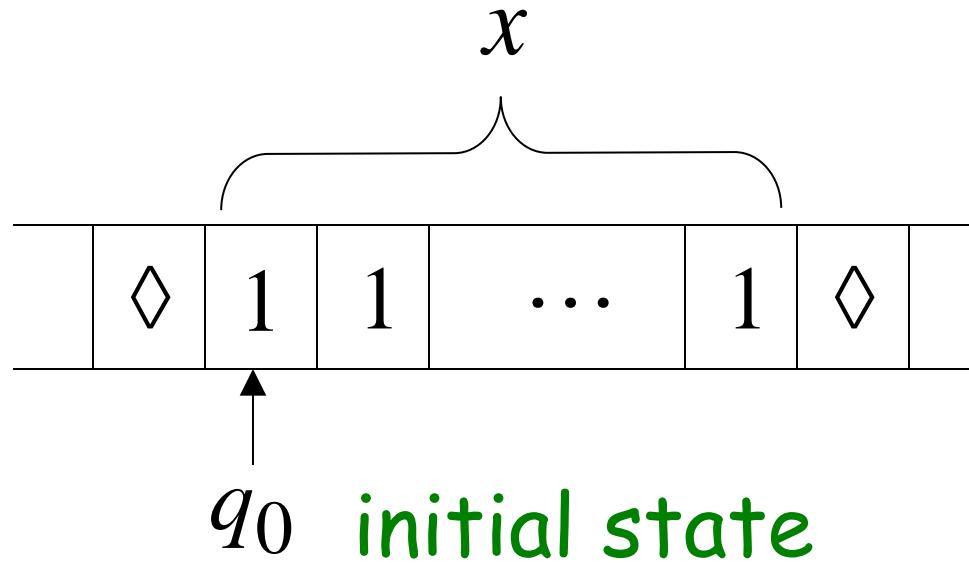
x is integer

Turing Machine:

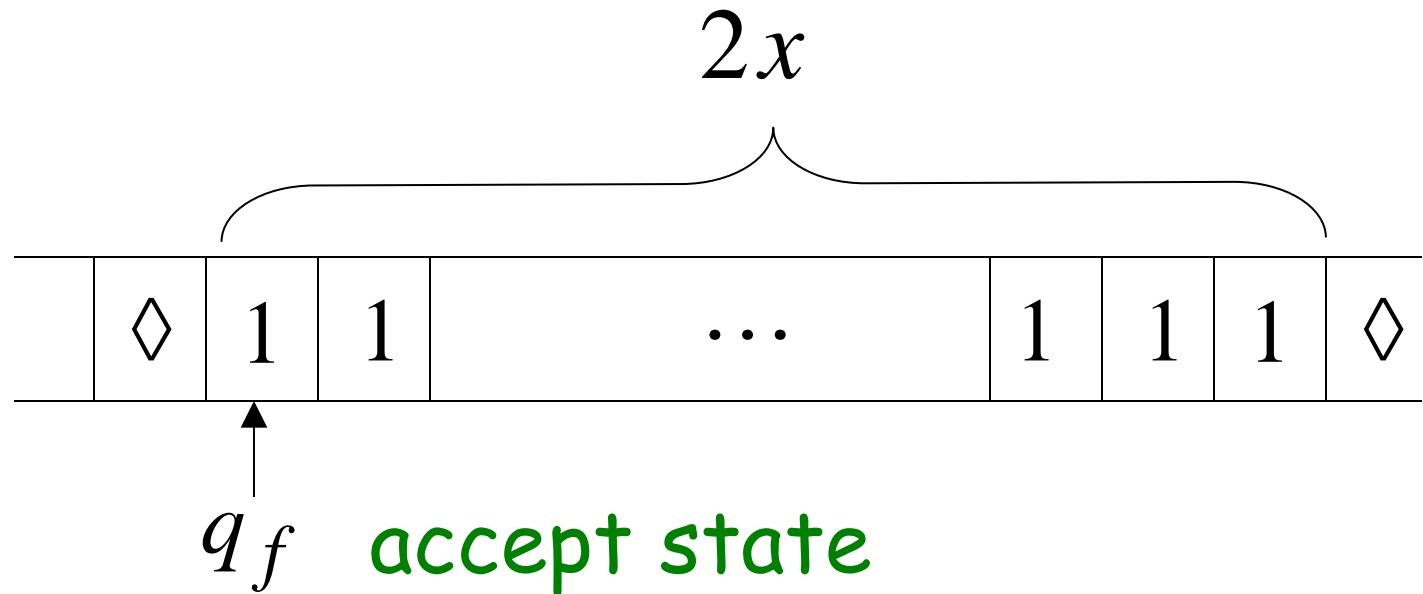
Input string: x unary

Output string: xx unary

Start



Finish

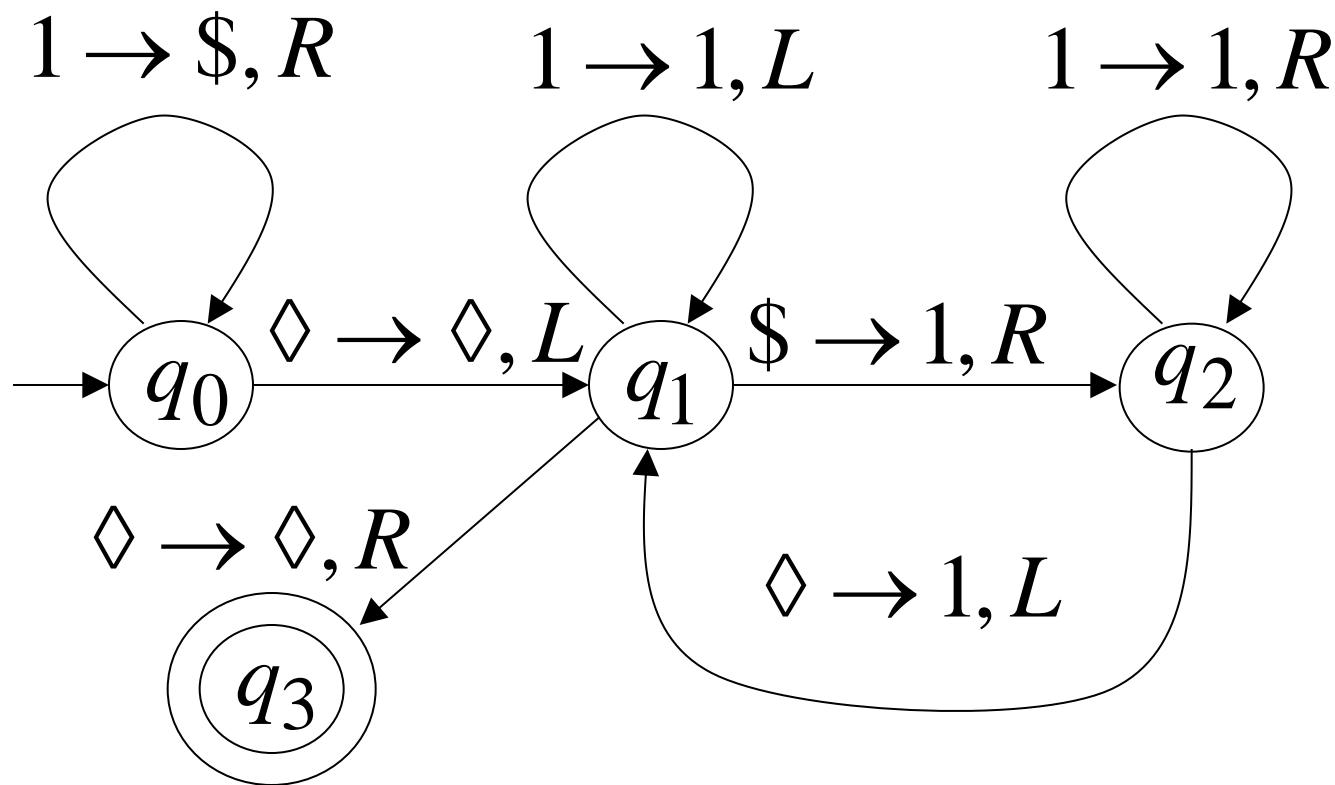


Turing Machine Pseudocode for $f(x) = 2x$

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

Until no more \$ remain

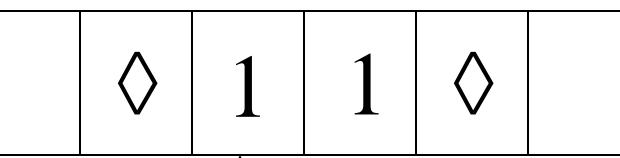
Turing Machine for $f(x) = 2x$



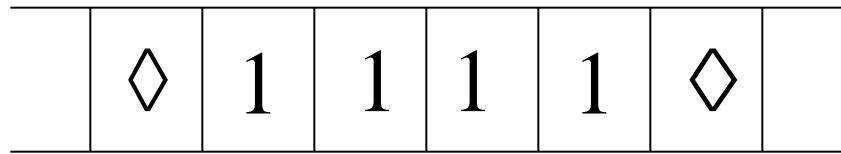
Example

Start

Finish



q_0

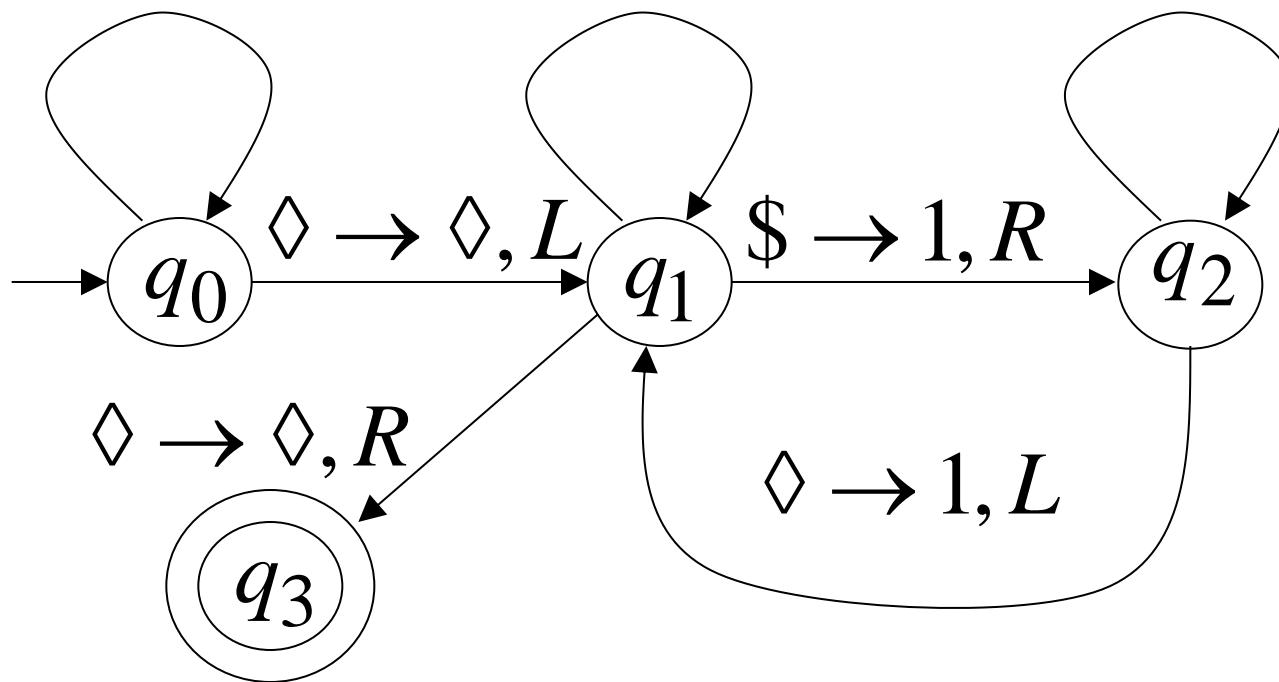


q_3

$1 \rightarrow \$, R$

$1 \rightarrow 1, L$

$1 \rightarrow 1, R$



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 ($x \leq y$)

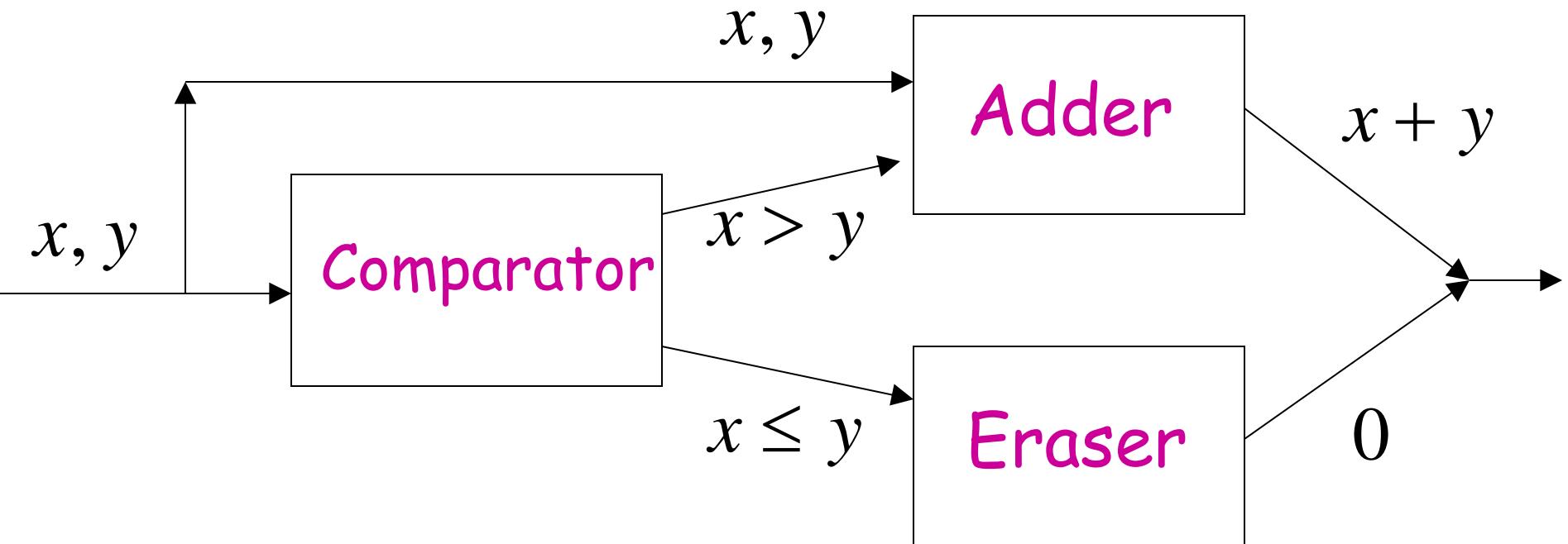
Combining Turing Machines

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

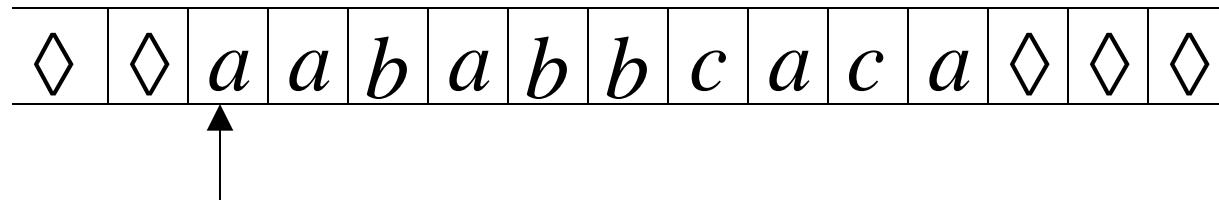
When we say: There exists an algorithm

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

Variations of the Turing Machine

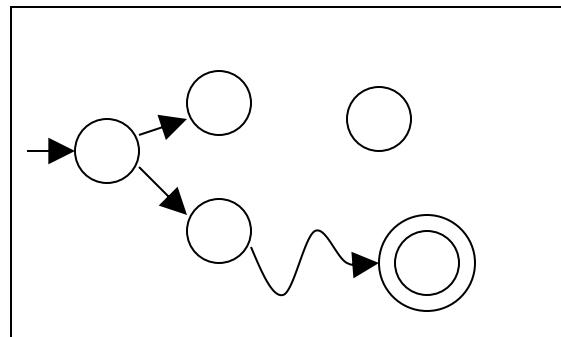
The Standard Model

Infinite Tape



Read-Write Head (Left or Right)

Control Unit



Deterministic

Variations of the Standard Model

- Turing machines with:
- Stay-Option
 - Semi-Infinite Tape
 - Off-Line
 - 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: $L(M_1) = L(M_2)$

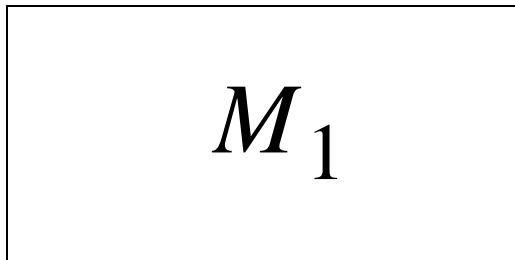
and vice-versa

Simulation: A technique to prove same power.

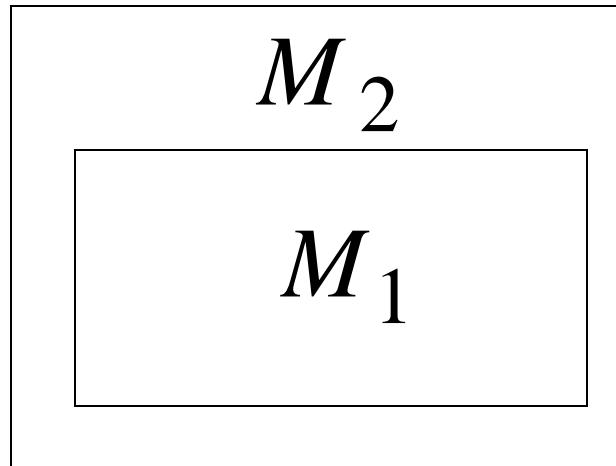
Simulate the machine of one class
with a machine of the other class

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

Original Machine:

M_1
 $d_0 \succ d_1 \succ \dots \succ d_n$

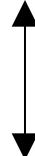


Simulation Machine:

$d'_0 \succ d'_1 \succ \dots \succ d'_n$
 M_2

Accepting Configuration

Original Machine:

 d_f 

Simulation Machine:

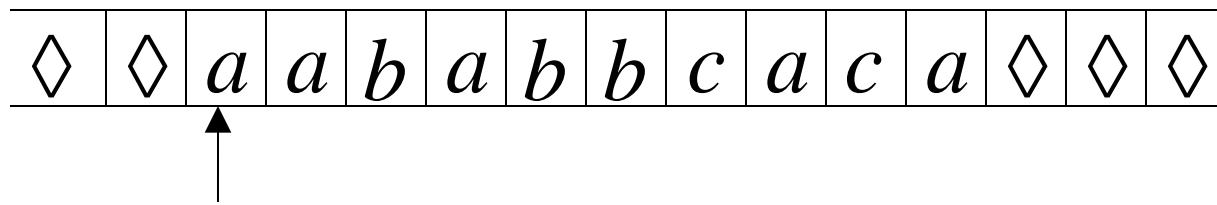
 d'_f

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

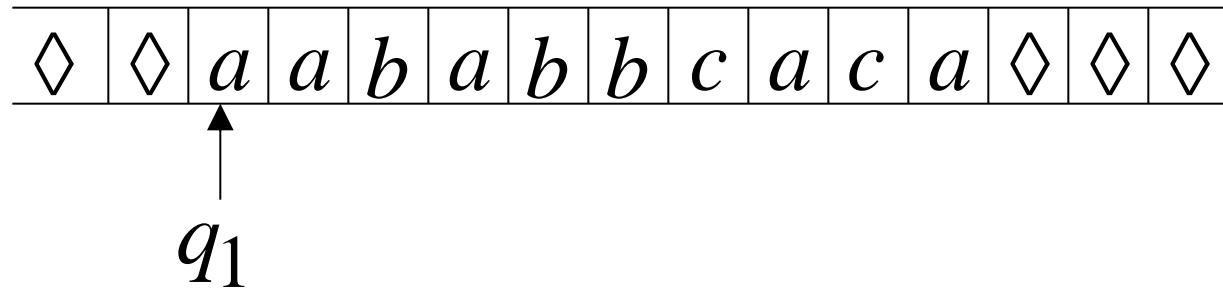


Left, Right, Stay

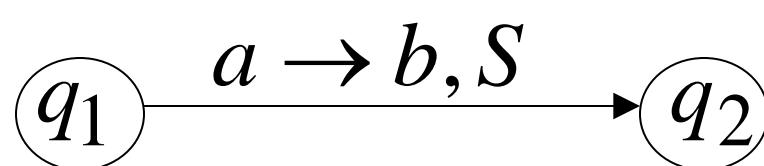
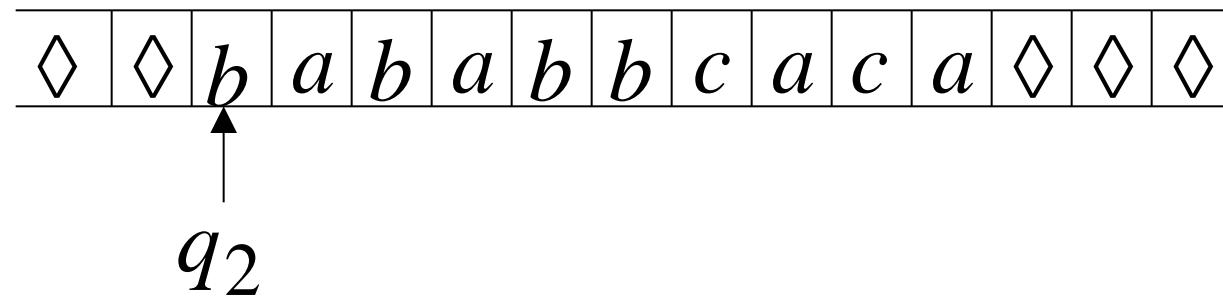
L,R,S: possible head moves

Example:

Time 1



Time 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

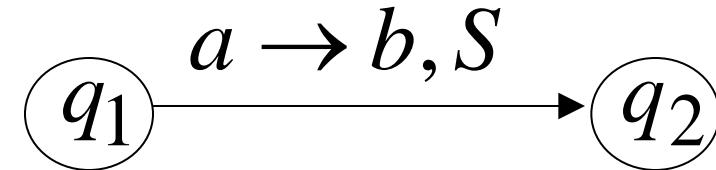
1. Stay-Option Machines simulate Standard Turing machines

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

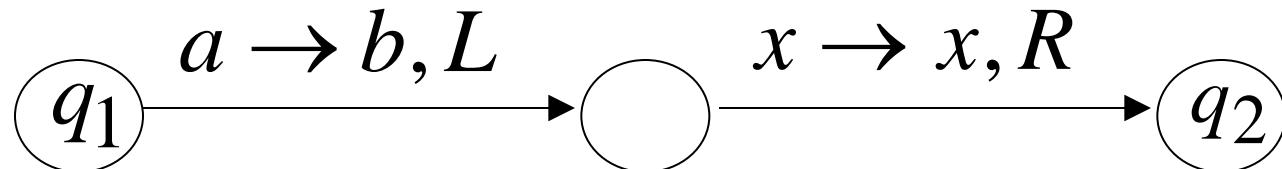
2. Standard Turing machines simulate Stay-Option machines

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

Stay-Option Machine



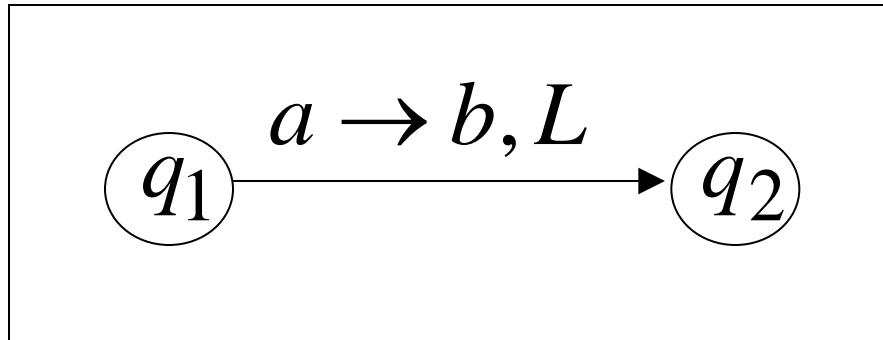
Simulation in Standard Machine



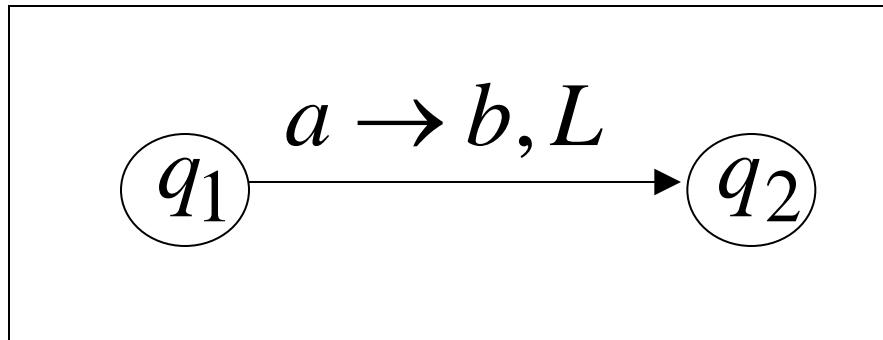
For every possible tape symbol x

For other transitions nothing changes

Stay-Option Machine



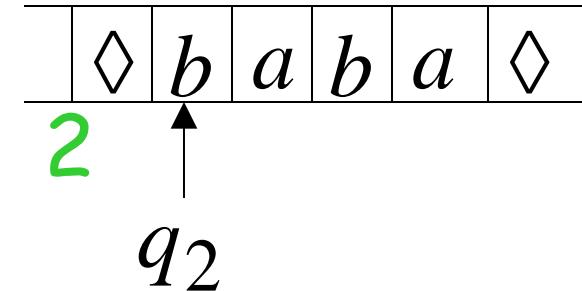
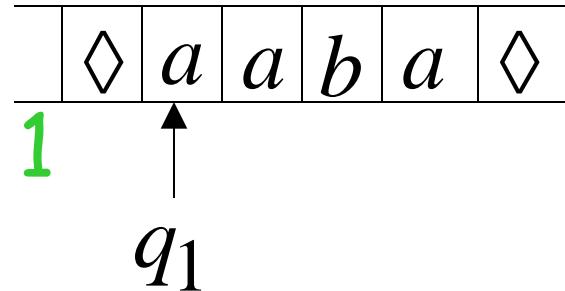
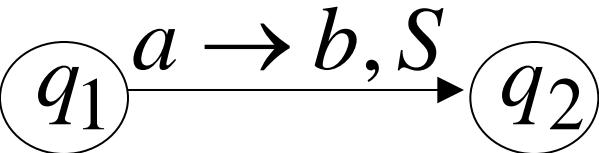
Simulation in Standard Machine



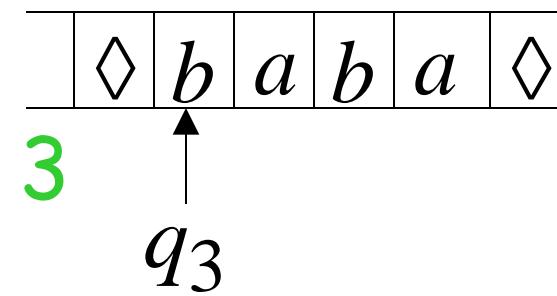
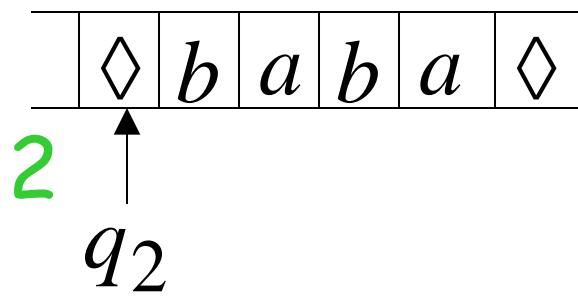
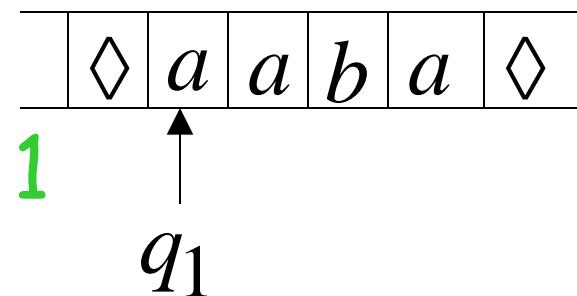
Similar for Right moves

example of simulation

Stay-Option Machine:



Simulation in Standard Machine:

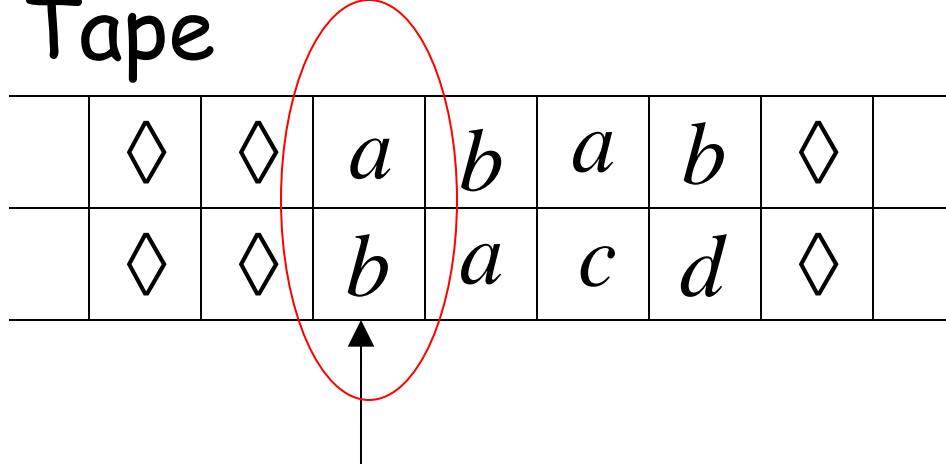


END OF PROOF

Multiple Track Tape

A useful trick to perform more complicated simulations

One Tape



One head

One symbol (a, b)

track 1

track 2

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

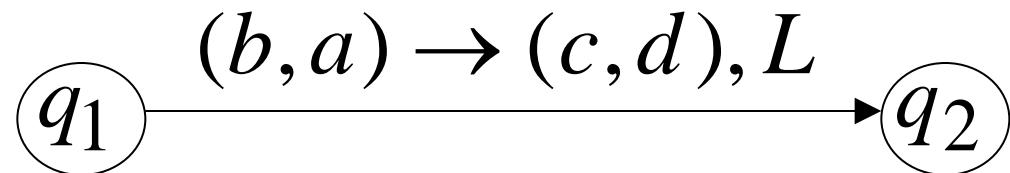
\uparrow
 q_1

track 1
track 2

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

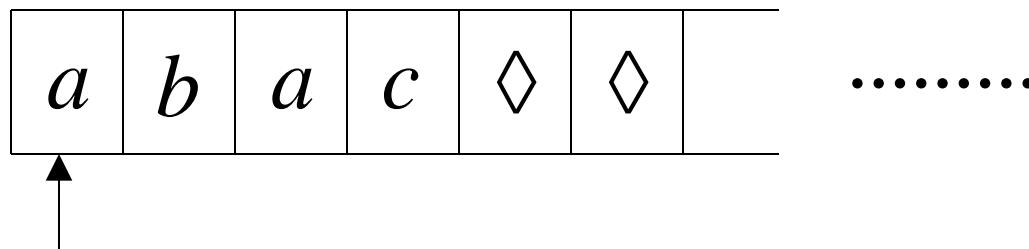
\uparrow
 q_2

track 1
track 2



Semi-Infinite Tape

The head extends infinitely only to the right



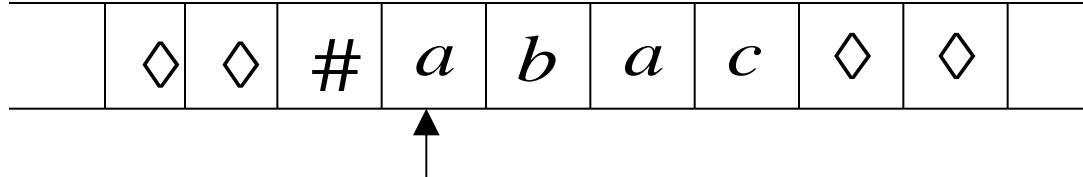
- Initial position is the leftmost cell
- When the head moves left from the border, it returns to the same position

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:

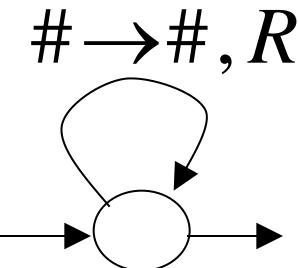


Standard Turing Machine

a. insert special symbol #

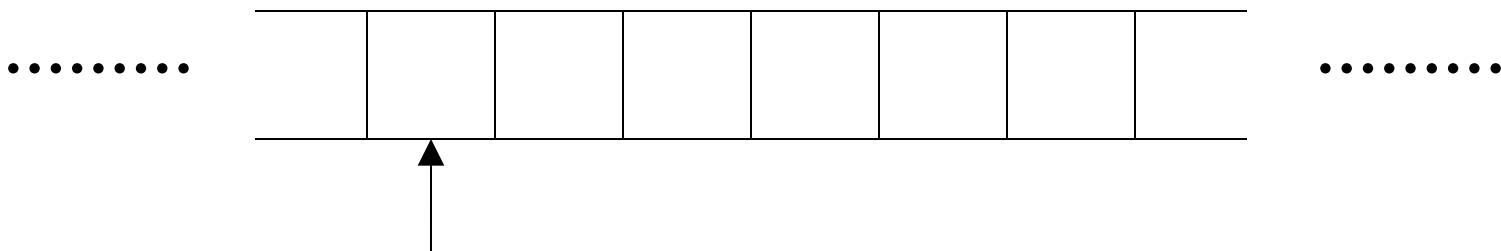
at left of input string

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

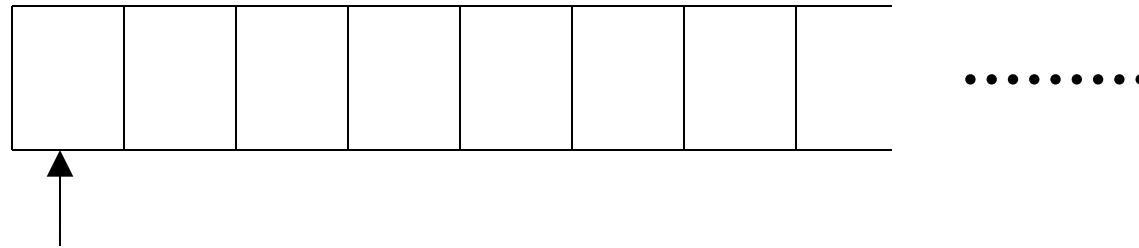


2. Semi-Infinite tape machines simulate Standard Turing machines:

Standard machine

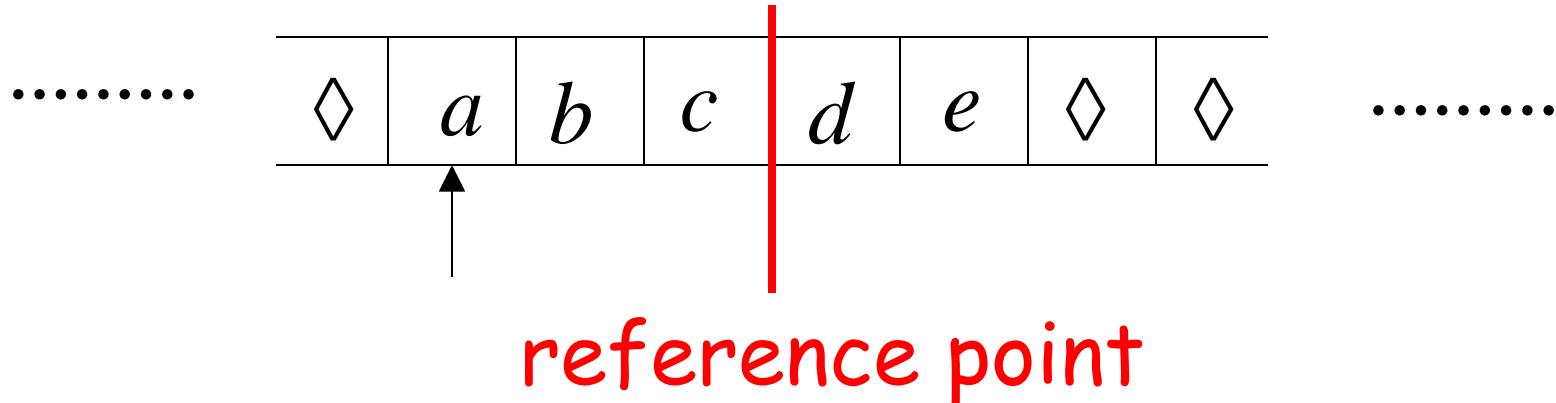


Semi-Infinite tape machine



Squeeze infinity of both directions
in one direction

Standard machine



Semi-Infinite tape machine with two tracks

Right part

#	d	e	◊	◊	◊	
#	c	b	a	◊	◊	

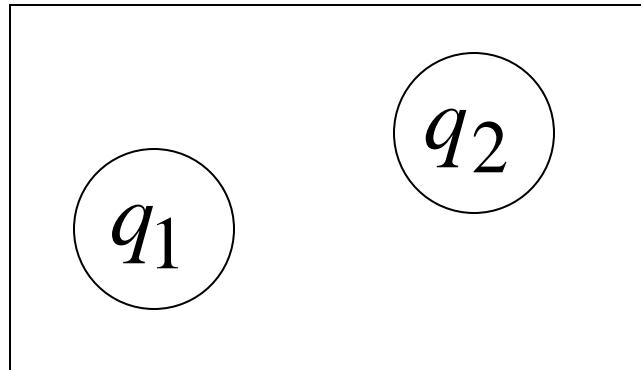
.....

Left part

#	d	e	◊	◊	◊	
#	c	b	a	◊	◊	

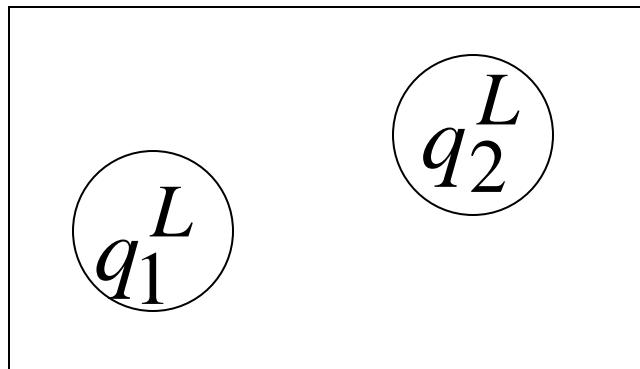


Standard machine

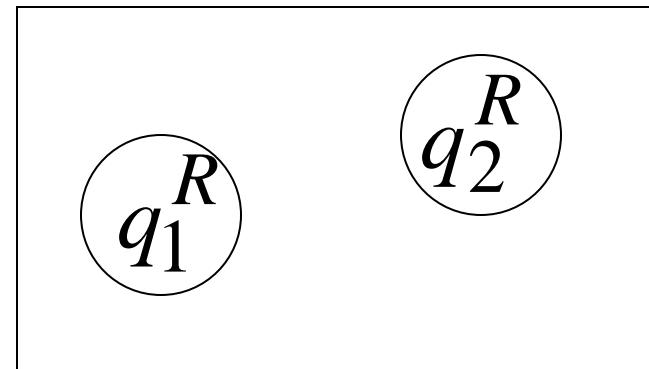


Semi-Infinite tape machine

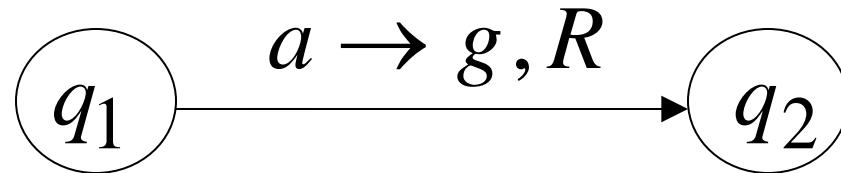
Left part



Right part

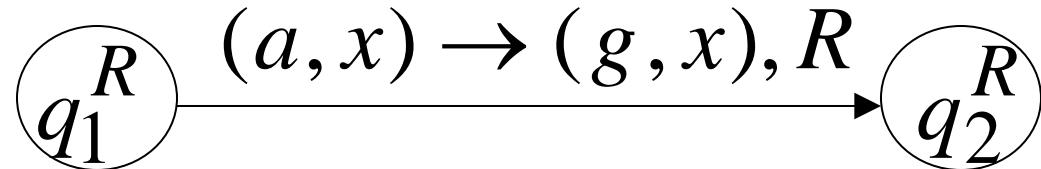


Standard machine

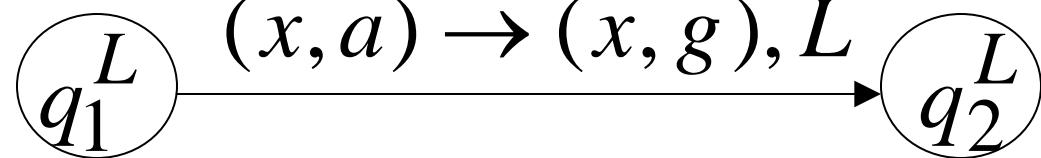


Semi-Infinite tape machine

Right part



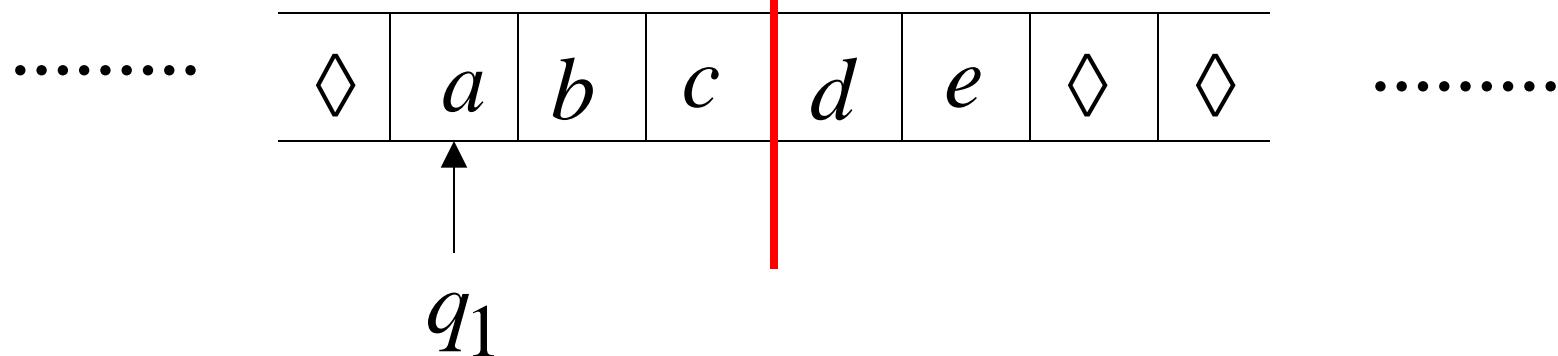
Left part



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	

.....

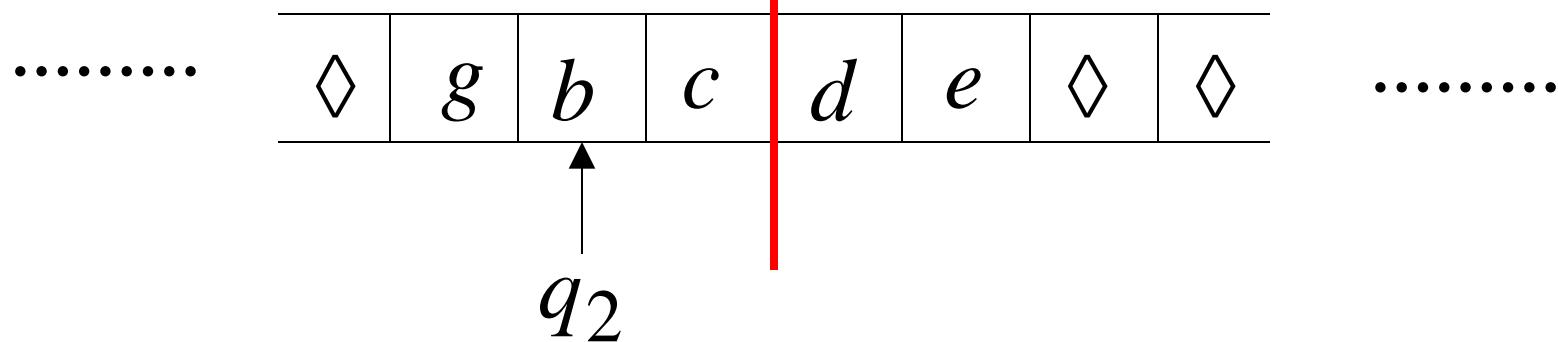
Left part

#	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

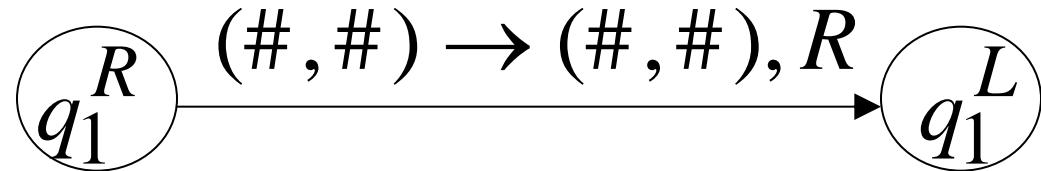
#	c	b	g	diamond	diamond	
#	c	b	g	diamond	diamond	

q_2^L

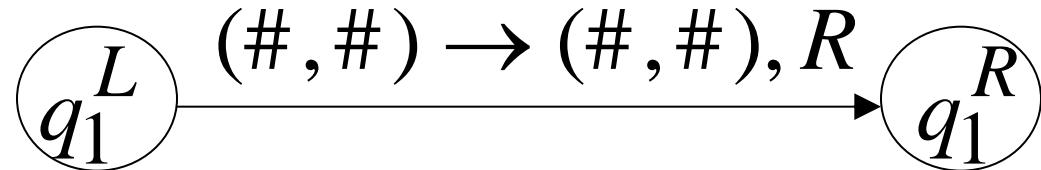
At the border:

Semi-Infinite tape machine

Right part



Left part



Semi-Infinite tape machine

Right part

Left part

Time 1

#	d	e	◊	◊	◊	
#	c	b	g	◊	◊	

\uparrow
 q_1^L

.....

Right part

Left part

Time 2

#	d	e	◊	◊	◊	
#	c	b	g	◊	◊	

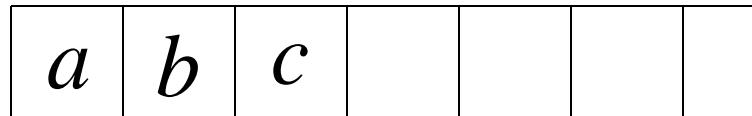
\uparrow
 q_1^R

.....

END OF PROOF

The Off-Line Machine

Input File read-only (once)

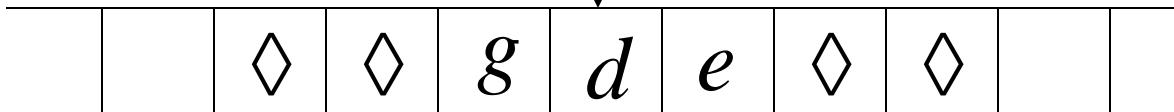


Input string

Input string
Appears on
input file only

Control Unit
(state machine)

Tape read-write



Theorem: Off-Line machines
have the same power with
Standard Turing machines

Proof: 1. Off-Line machines
simulate Standard Turing machines

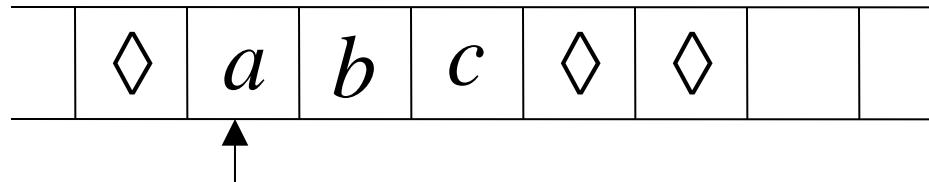
2. Standard Turing machines
simulate Off-Line machines

1. Off-line machines simulate Standard Turing Machines

Off-line machine:

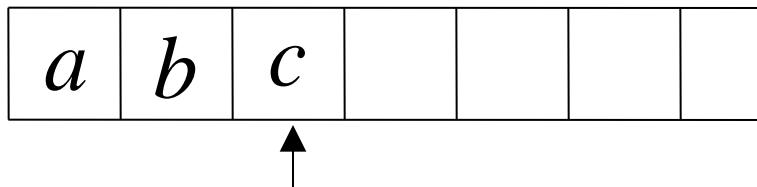
1. Copy input file to tape
2. Continue computation as in
Standard Turing machine

Standard machine

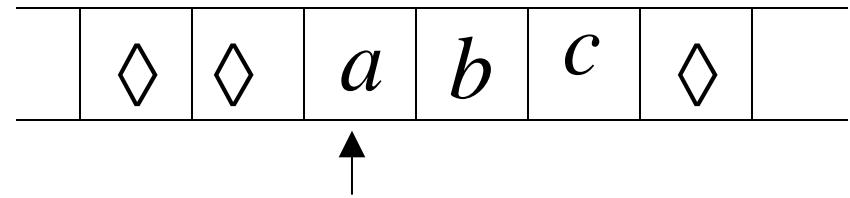


Off-line machine

Input File

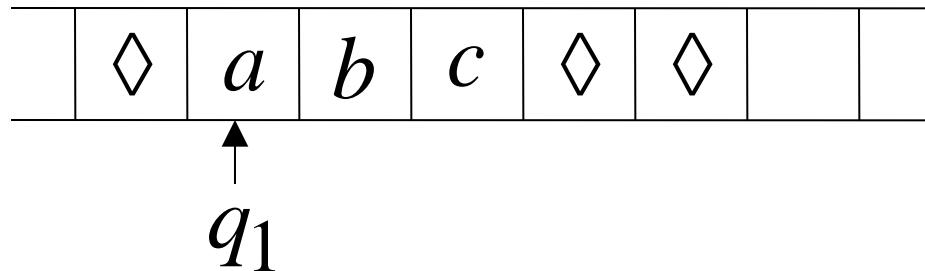


Tape



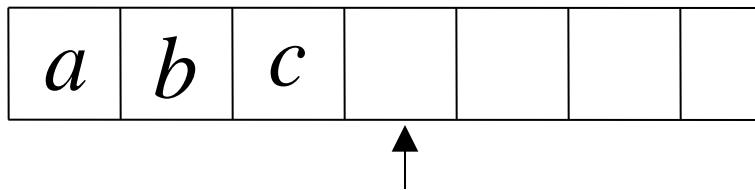
1. Copy input file to tape

Standard machine

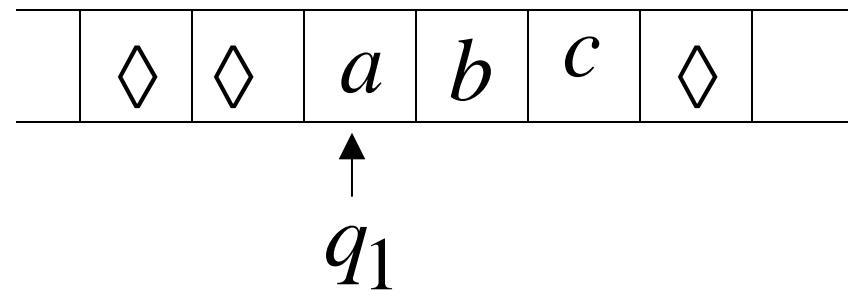


Off-line machine

Input File



Tape



2. Do computations as in Turing machine

2. Standard Turing machines simulate Off-Line machines:

Use a Standard machine with
a four-track tape to keep track of
the Off-line input file and tape contents

Off-line Machine

Input File

a	b	c	d			
---	---	---	---	--	--	--

↑

Tape

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

↑

Standard Machine -- Four track tape

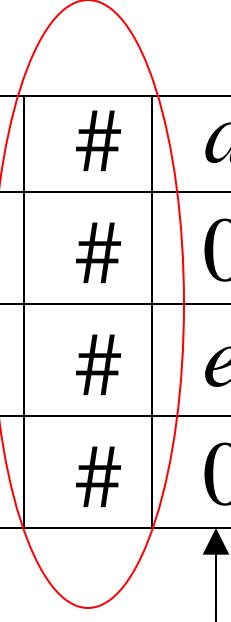
	#	a	b	c	d		
	#	0	0	1	0		
		e	f	g			
		0	1	0			

↑

Input File
head position
Tape
head position

Reference point (uses special symbol #)

#	a	b	c	d		
#	0	0	1	0		
#	e	f	g			
#	0	1	0			



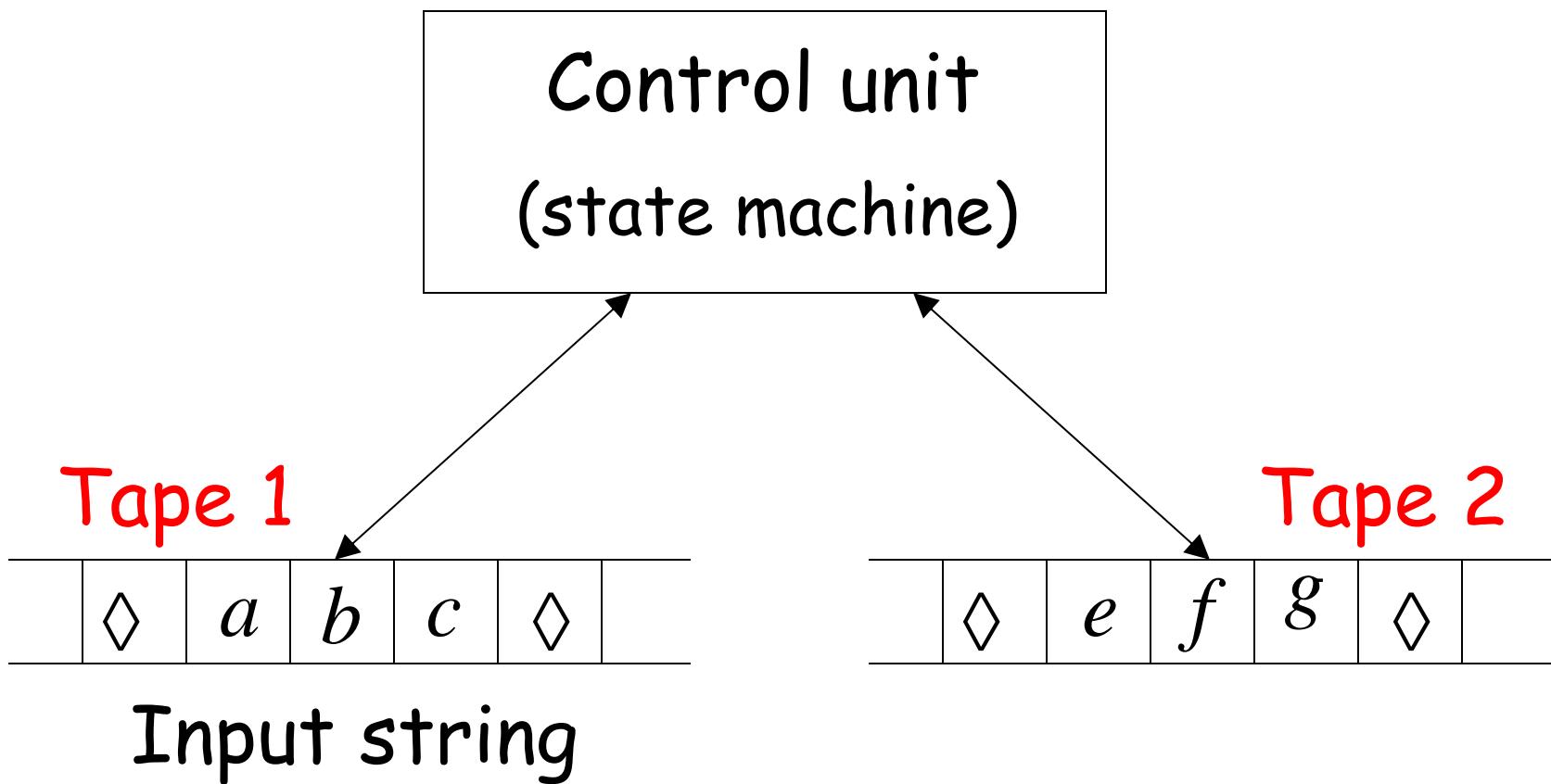
Input File
head position
Tape
head position

Repeat for each state transition:

1. Return to reference point
2. Find current input file symbol
3. Find current tape symbol
4. Make transition

END OF PROOF

Multi-tape Turing Machines



Input string appears on Tape 1

Tape 1

Time 1

Tape 2

	◊	a	b	c	◊	
--	---	---	---	---	---	--



q_1

	◊	e	f	g	◊	
--	---	---	---	---	---	--



q_1

Tape 1

Time 2

Tape 2

	◊	a	g	c	◊	
--	---	---	---	---	---	--

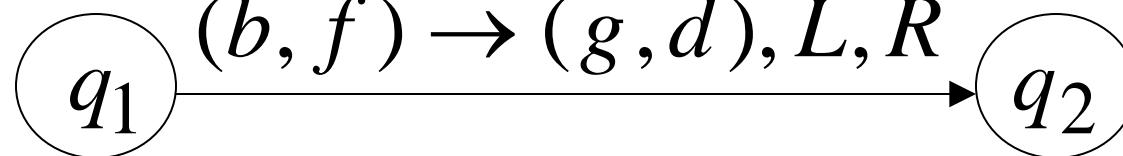


q_2

	◊	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 just 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		

↑

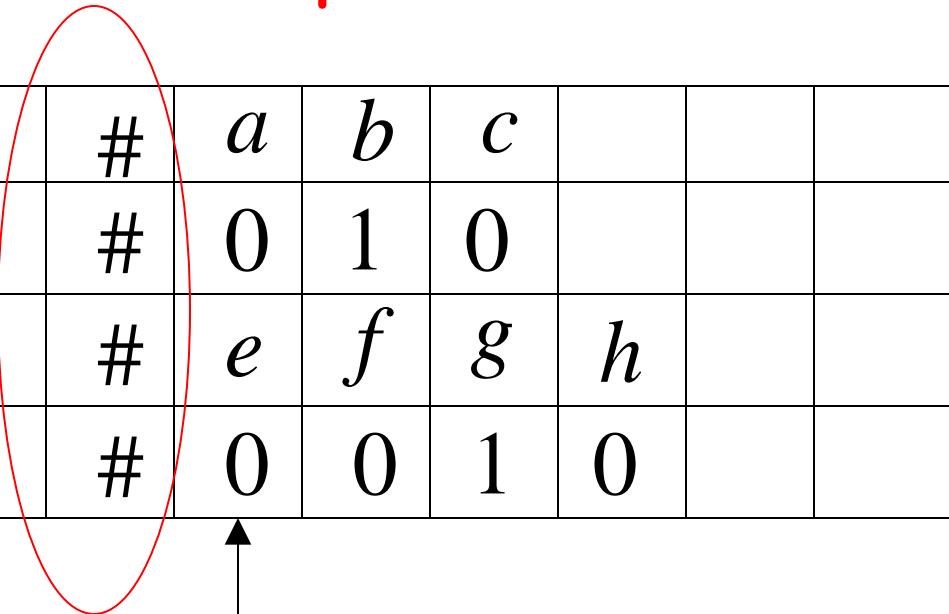
Tape 1

head position

Tape 2

head position

Reference point



A diagram illustrating the reference point for two tapes. On the left, there are two horizontal tapes represented by grids of cells. The top tape is labeled "Tape 1 head position" and has columns labeled "#", "a", "b", "c", and several empty columns. The bottom tape is labeled "Tape 2 head position" and has columns labeled "#", "e", "f", "g", "h", and several empty columns. A red oval highlights the first cell of Tape 1, which contains the symbol "#". A vertical arrow points upwards from the bottom of the grid towards this highlighted cell.

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

Tape 1
head position
Tape 2
head position

Repeat for each state transition:

1. Return to reference point
2. Find current symbol in Tape 1
3. Find current symbol in Tape 2
4. Make transition

END OF PROOF

Same power doesn't imply same speed:

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

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

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

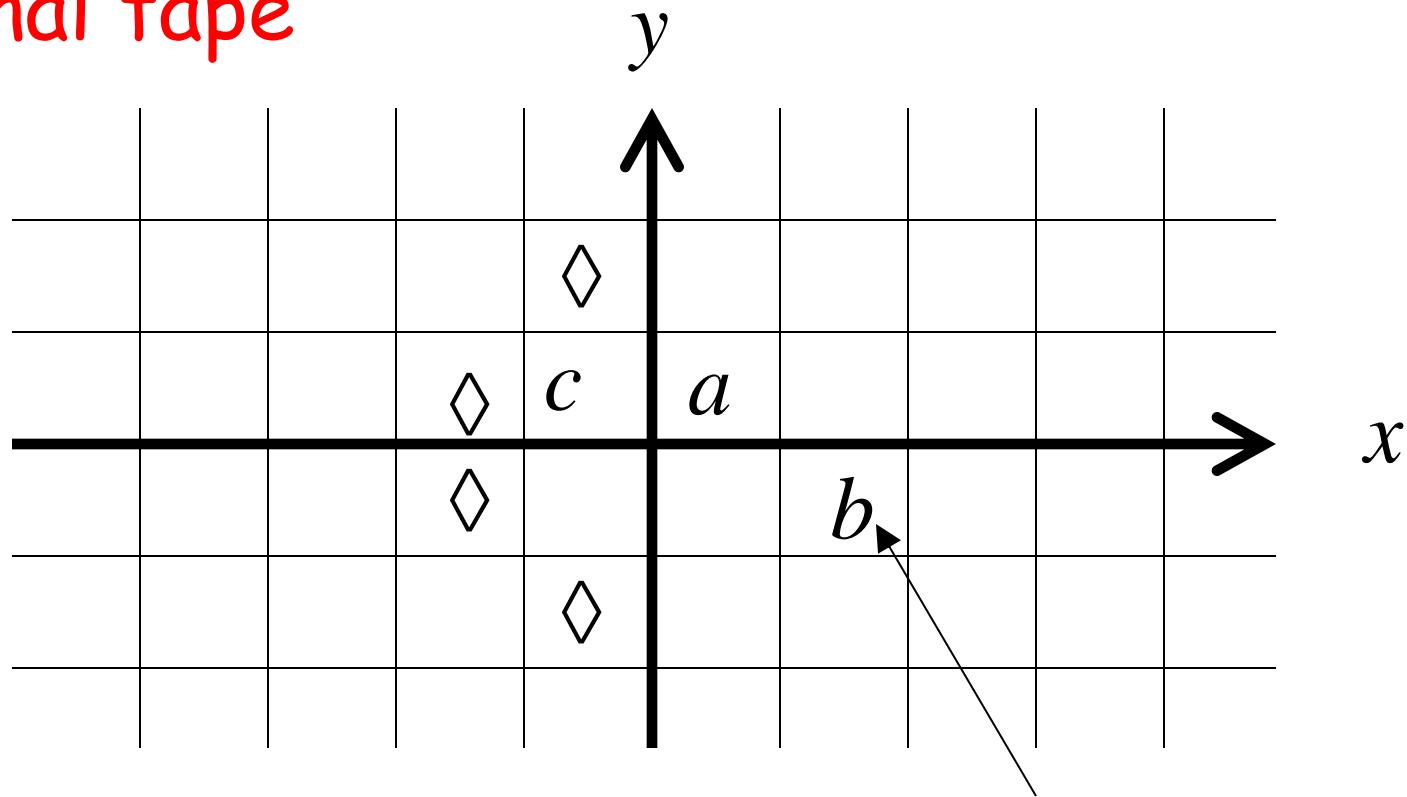
2-tape machine: $O(n)$ time

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

2. Compare a^n on tape 1
and b^n 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

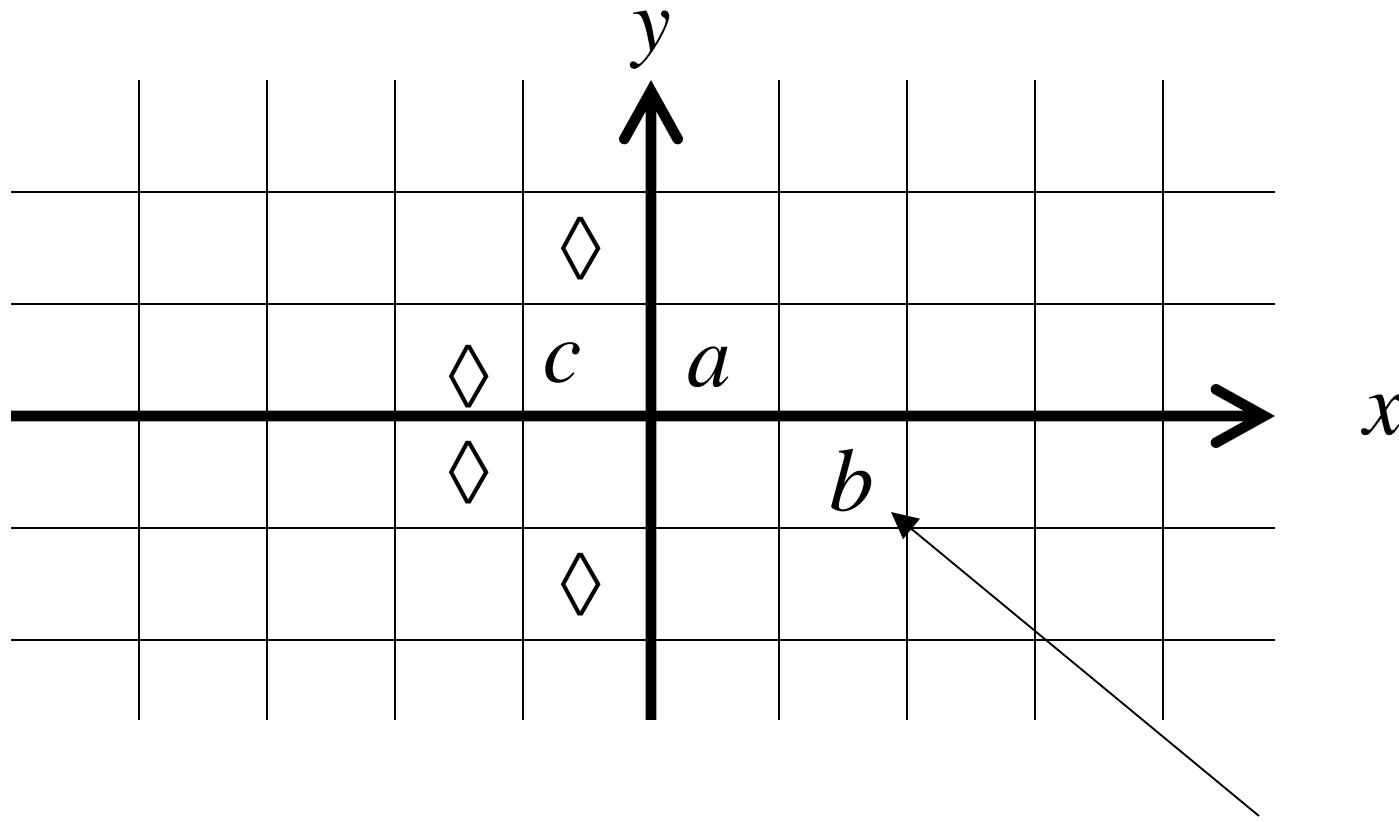
Trivial: Use one dimension

2. Standard Turing machines simulate Multidimensional machines

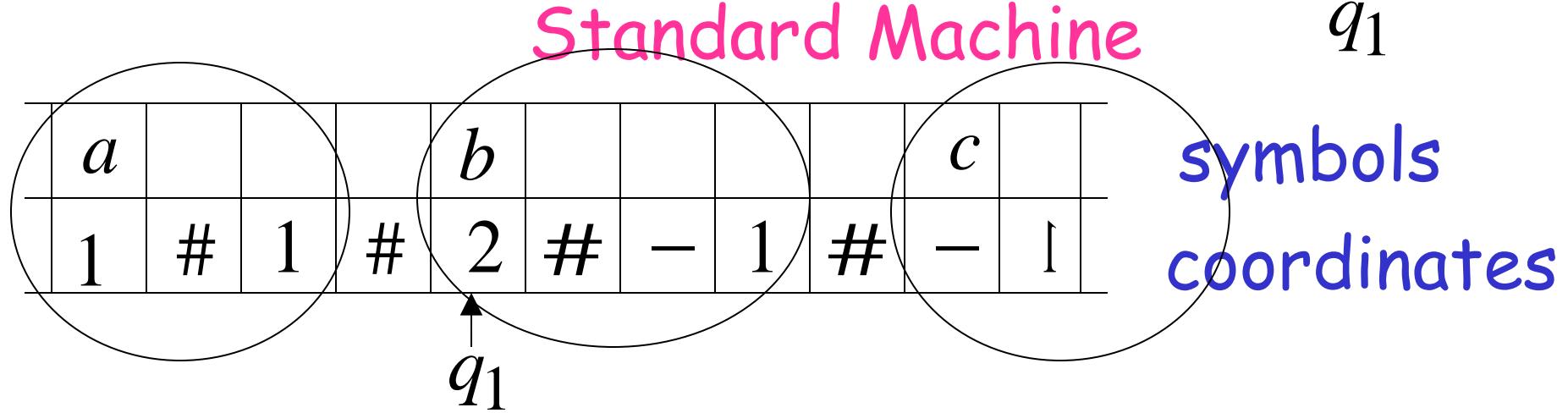
Standard machine:

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

2-dimensional machine



Standard Machine



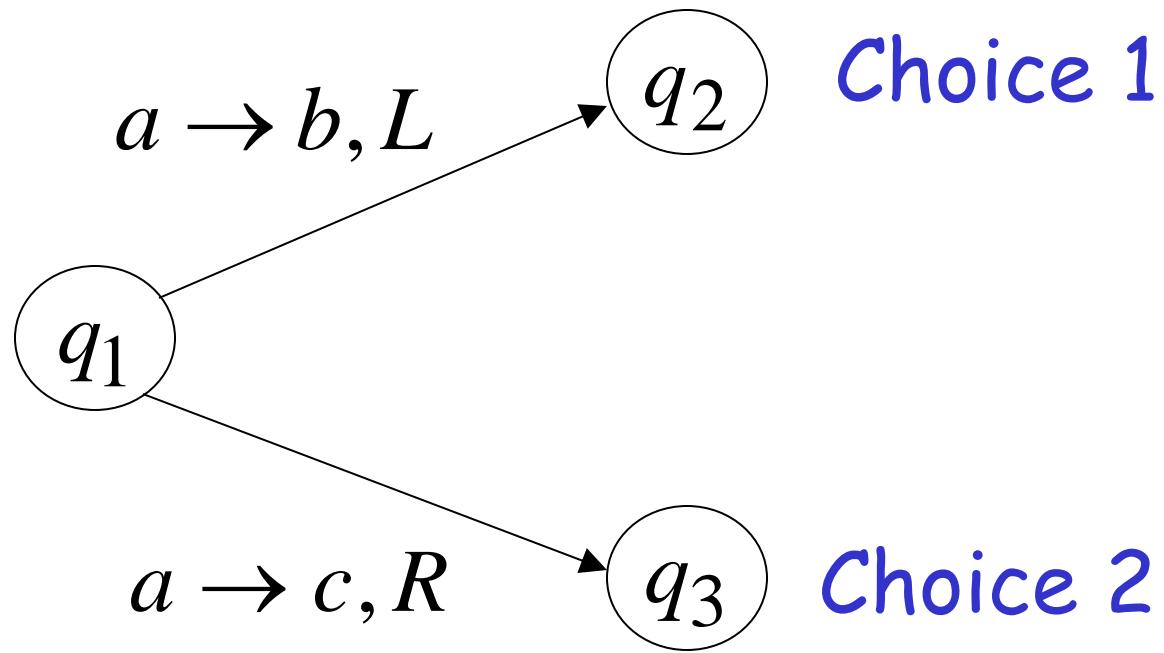
Standard machine:

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

1. Update current symbol
2. Compute coordinates of next position
3. Go to new position

END OF PROOF

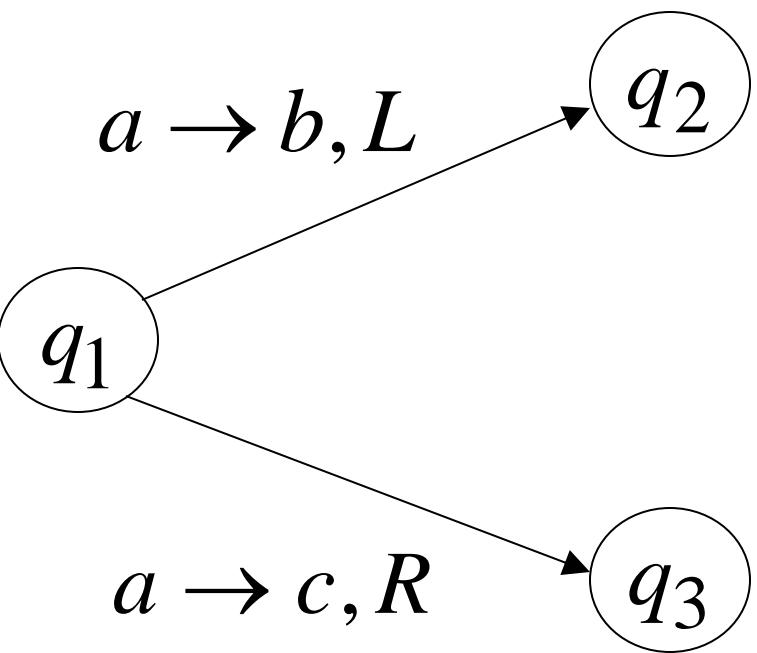
Nondeterministic Turing Machines



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				

q_3

Input string w is accepted if
there is a computation:

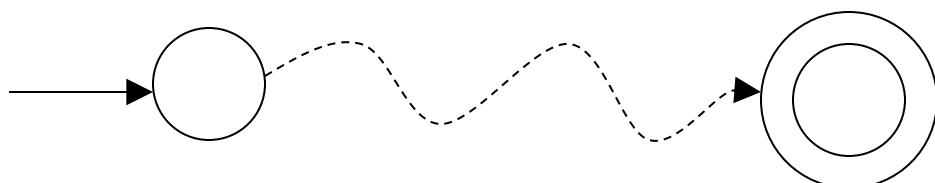
$$q_0 w \prec x q_f y$$

Initial configuration

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

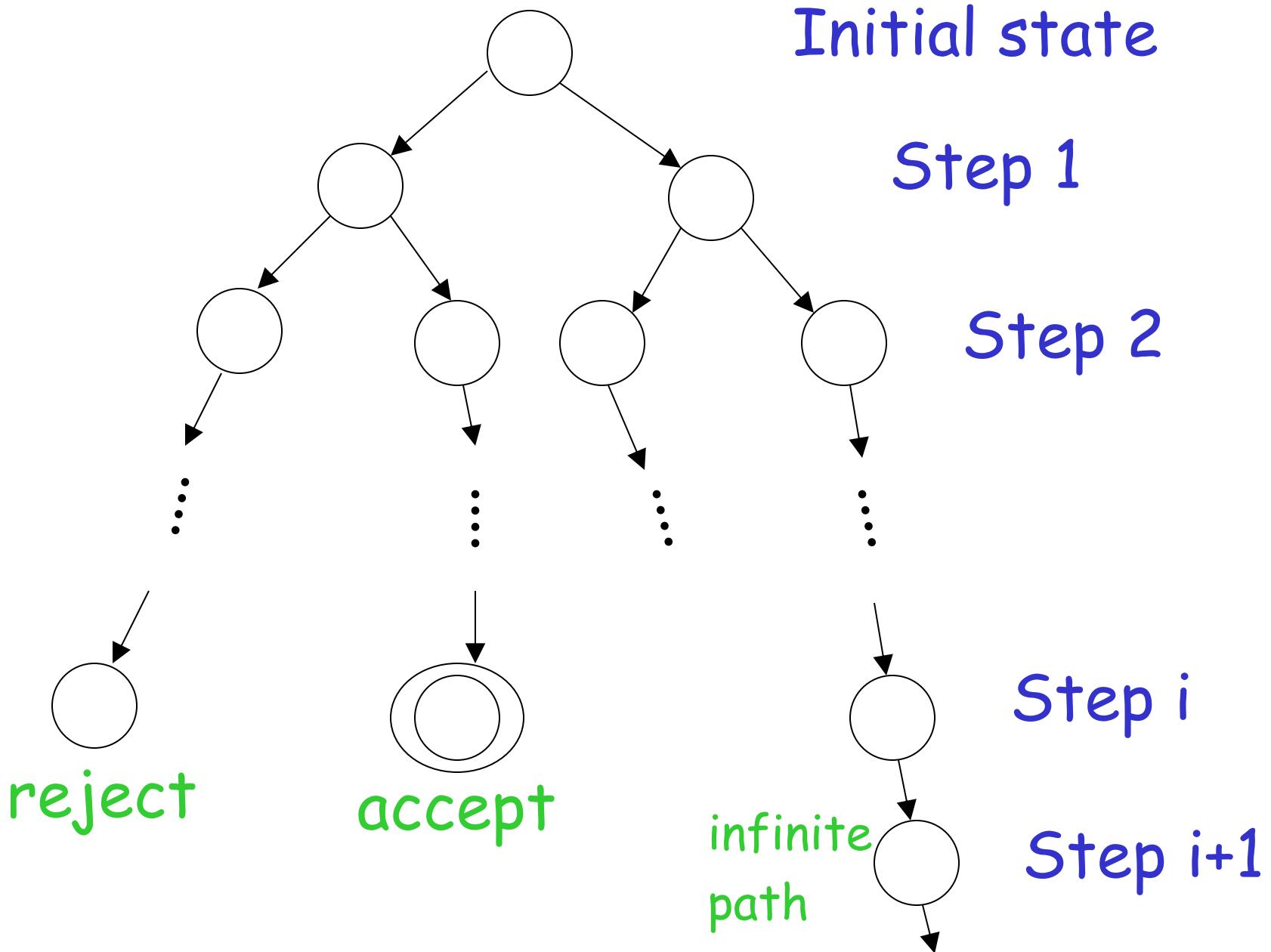
Trivial: every deterministic machine
is also nondeterministic

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

Deterministic machine:

- Uses a 2-dimensional tape
(which is equivalent to 1-dimensional tape)
- Stores all possible computations
of the non-deterministic machine
on the 2-dimensional tape

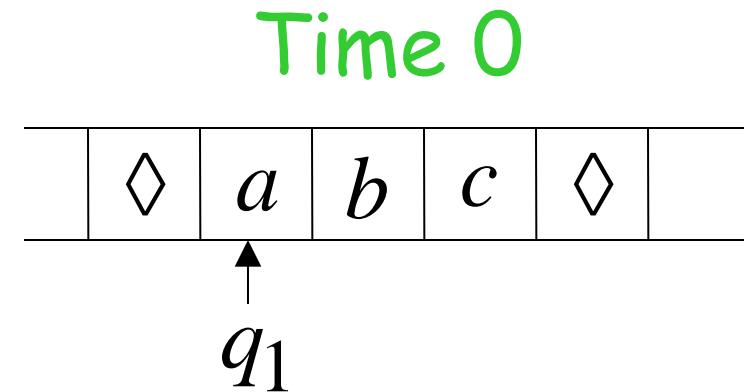
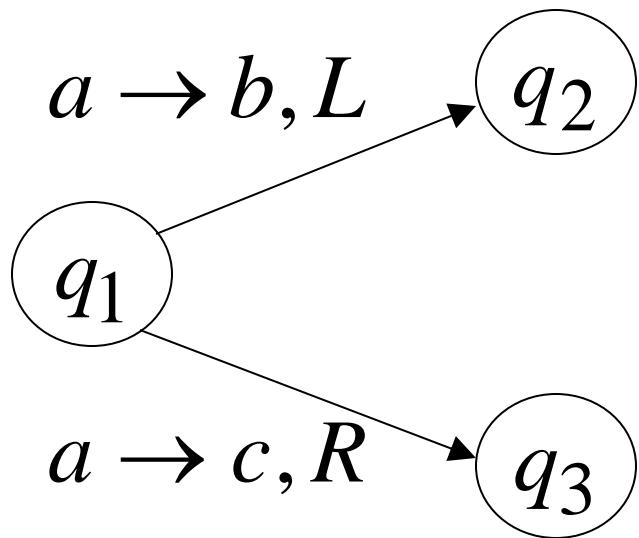
All possible computation paths



The Deterministic Turing machine
simulates all possible computation paths:

- simultaneously
- step-by-step
- in a breadth-first search fashion

NonDeterministic machine



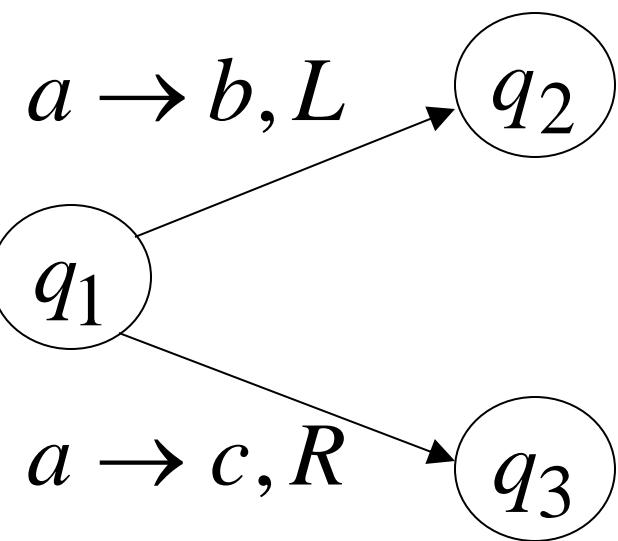
Deterministic machine

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

current
configuration

NonDeterministic machine

Time 1



	◊	b	b	c	◊	

q_2

Choice 1

	◊	c	b	c	◊	

q_3

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

END OF PROOF

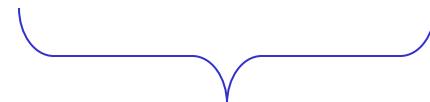
Remark:

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

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:

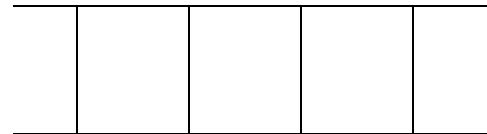
Description of transitions of M

Input string of M

Three tapes

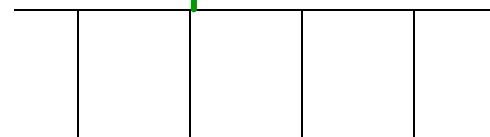


Tape 1



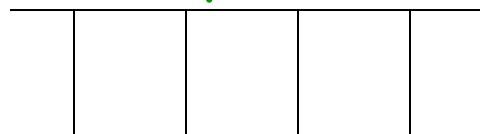
Description of M

Tape 2



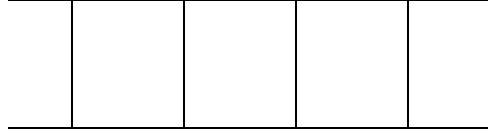
Tape Contents of M

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:	a	b	c	d	...
	\downarrow	\downarrow	\downarrow	\downarrow	

Encoding:	1	11	111	1111	

State Encoding

States:

q_1

q_2

q_3

q_4

\cdots



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:

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

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

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

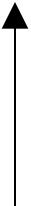
separator

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

$L = \{ 010100101,$ (Turing Machine 1)
 $00100100101111,$ (Turing Machine 2)
 $111010011110010101,$
..... }

Countable Sets

Infinite sets are either:

Countable

or

Uncountable

Countable set:

There is a one to one correspondence
of
elements of the set
to
Natural numbers (Positive Integers)

(every element of the set is mapped to a 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:

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

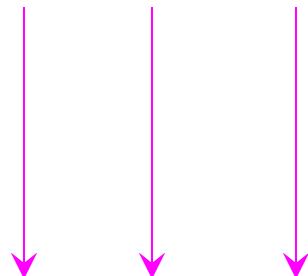
Naïve Approach

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

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

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

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

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

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



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

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

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

...

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

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

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

...

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

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

...

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

...

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



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

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

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

...

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



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

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

...

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

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

...

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

...

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



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

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

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

...

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



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

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

...

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

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

...

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

...

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



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

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

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

...

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



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

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

...

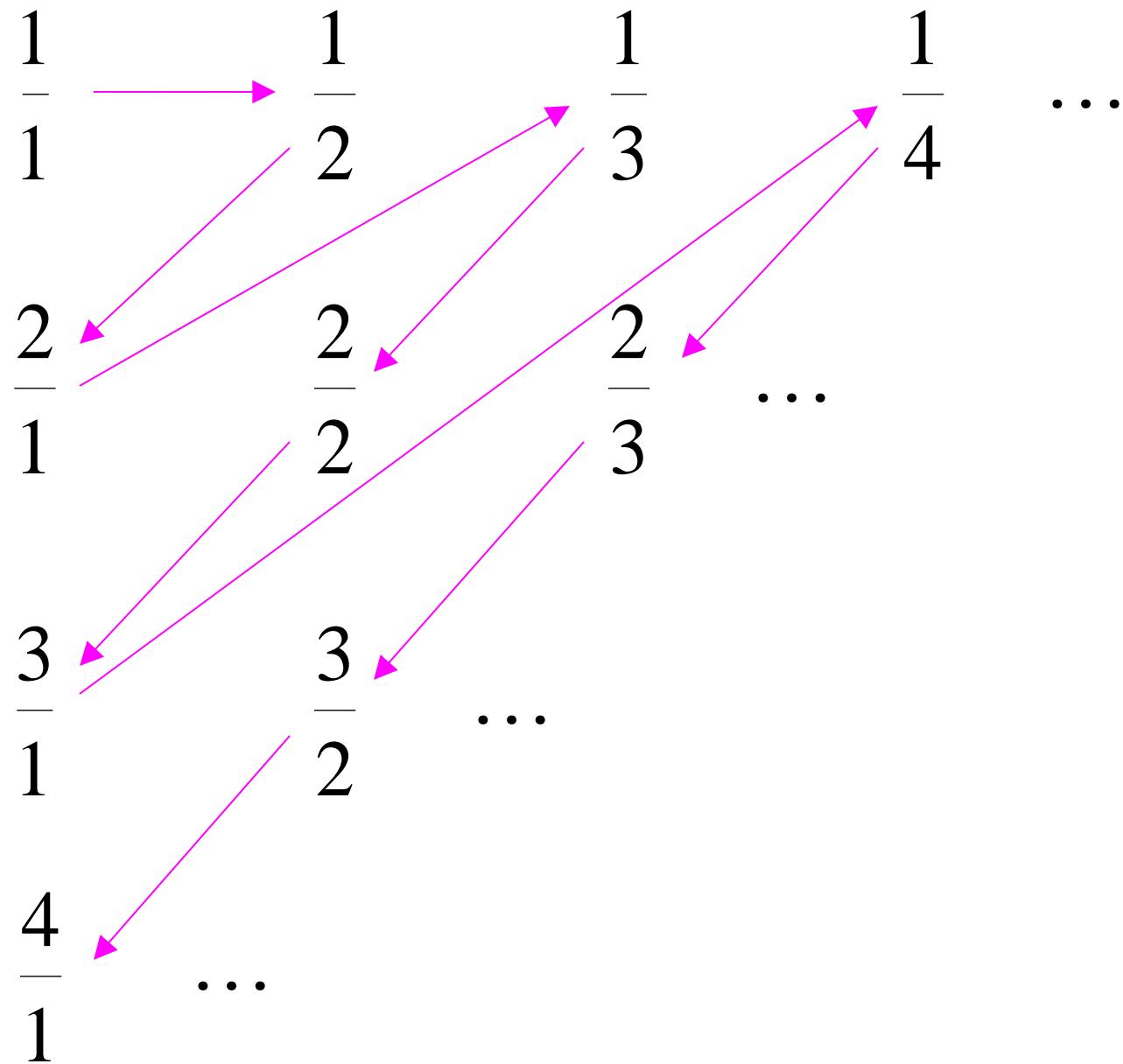
$$\frac{3}{1}$$

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

...

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

...



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

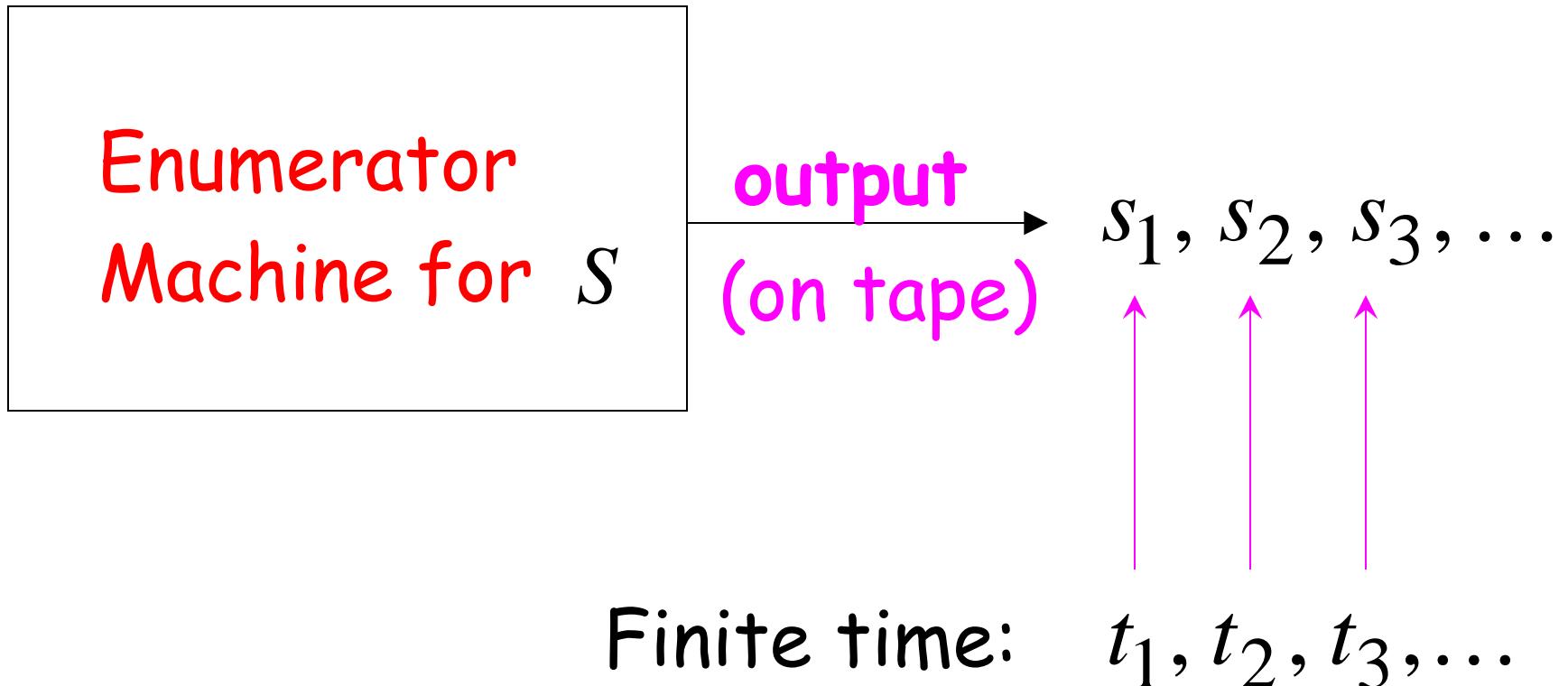
Let S be a set of strings (Language)

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

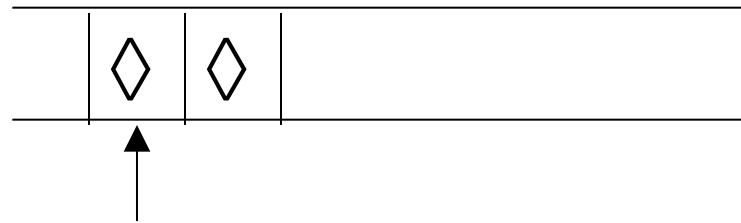
strings $s_1, s_2, s_3, \dots \in S$



Enumerator Machine

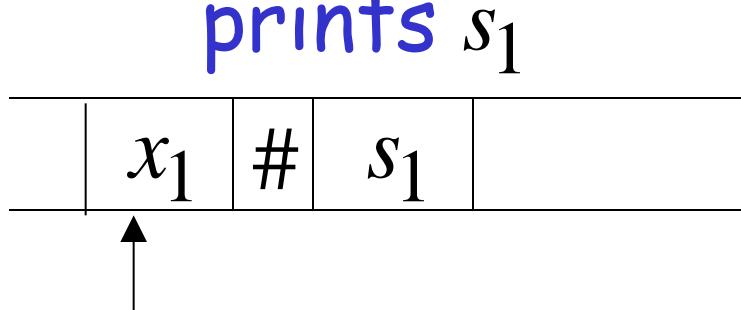
Configuration

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

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
-

$$\begin{aligned}s_1 &= a \\ s_2 &= b \\ \vdots &= c\end{aligned}\quad \left. \right\} \text{length 1}$$

Produce strings in
Proper Order:

$$\begin{aligned}aa \\ ab \\ ac \\ ba \\ bb \\ bc \\ ca \\ cb \\ cc\end{aligned}\quad \left. \right\} \text{length 2}$$
$$\begin{aligned}aaa \\ aab \\ aac \\ \dots\dots\end{aligned}\quad \left. \right\} \text{length 3}$$

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

1

00

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

:

:

s_1

Turing Machines

1 0 1 0 1 1 0 1 1 0 1

s_2

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

End of Proof

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)

Definition: A set is uncountable if it is not countable

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

Theorem:

If S is an infinite countable set, then

the powerset 2^S of S is uncountable.

(the powerset 2^S is the set whose elements
are all possible sets made from the elements of S)

Proof:

Since S is countable, we can write

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



Elements of S

Elements of the powerset 2^S have the form:

\emptyset

$\{s_1, s_3\}$

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

.....

We encode each element of the powerset
with a binary string of 0's and 1's

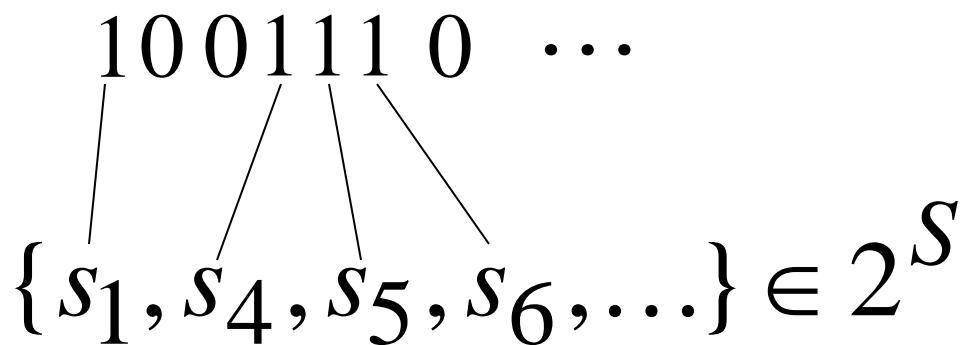
Powerset element (in arbitrary order)	Binary encoding				
	s_1	s_2	s_3	s_4	\dots
$\{s_1\}$	1	0	0	0	\dots
$\{s_2, s_3\}$	0	1	1	0	\dots
$\{s_1, s_3, s_4\}$	1	0	1	1	\dots

Observation:

Every infinite binary string corresponds to an element of the powerset:

Example:

Corresponds to:



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

Then: we can enumerate
the elements of the powerset

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

Powerset
element

suppose that this is the respective
Binary encoding

t_1 1 0 0 0 0 ...

t_2 1 1 0 0 0 ...

t_3 1 1 0 1 0 ...

t_4 1 1 0 0 1 ...

...

...

Take 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: $\mathbf{t} = 0011\dots$

(binary complement of diagonal)

The binary string

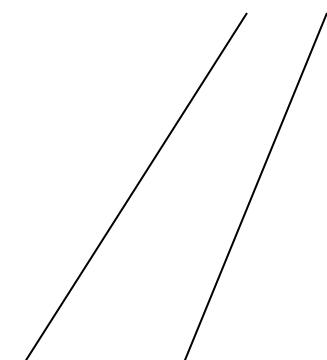
$$t = 0011\dots$$

corresponds

to an element of

the powerset 2^S :

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



Thus, t must be equal to some t_i

$$t = t_i$$

However,

the i -th bit in the encoding of t is
the complement of the i -th bit of t_i , thus:

$$t \neq t_i$$

Contradiction!!!

Since we have a contradiction:

The powerset 2^S of S is uncountable

End of proof

An Application: Languages

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

The set of all Strings:

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

infinite and countable

(we can enumerate the strings
in proper order)

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

The set of all Strings:

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

infinite and countable

Any language is a subset of S :

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

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

The set of all Strings:

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

infinite and countable

The powerset of S contains all languages:

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

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

countable

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 have $X \subset 2^S$)

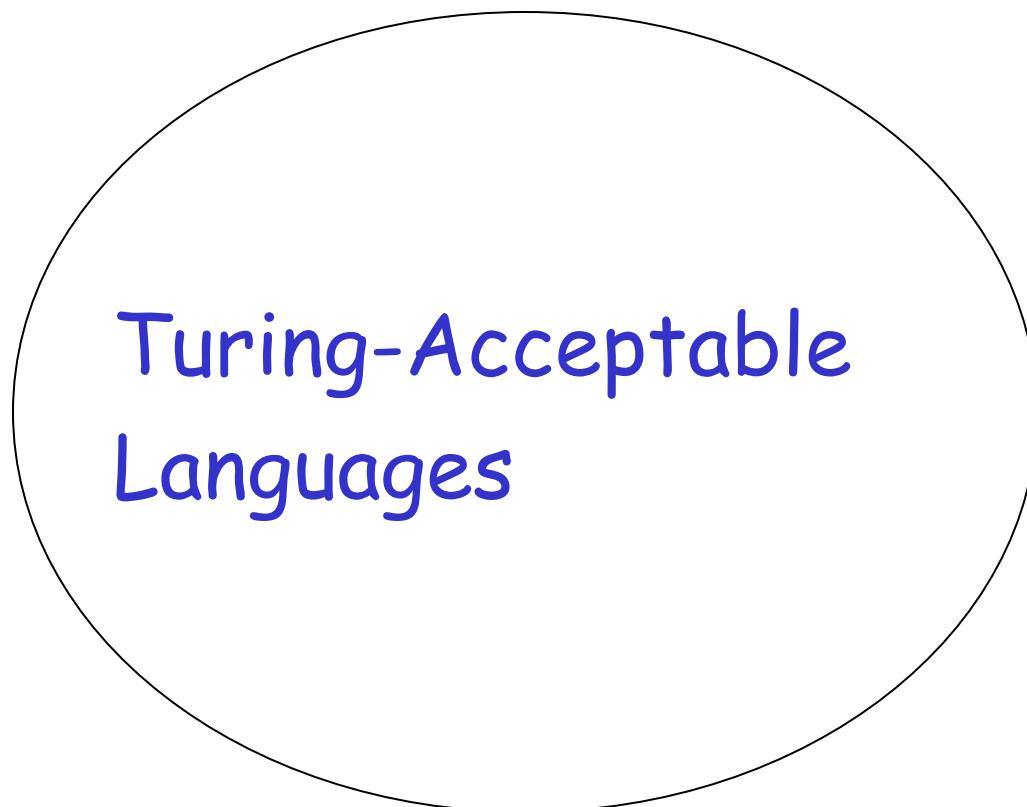
Conclusion:

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

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

(Language L' cannot be described by any algorithm)

Non Turing-Acceptable Languages



L'

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

The Chomsky Hierarchy

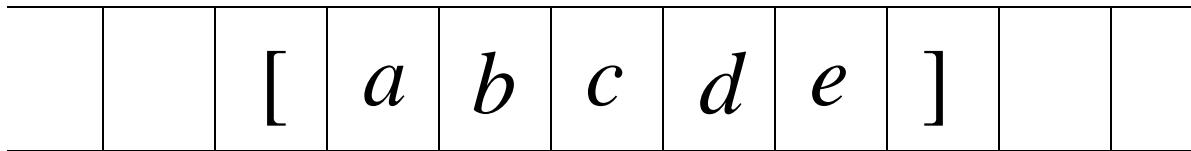
Linear-Bounded Automata:

Same as Turing Machines with one difference:

the input string tape space
is the only tape space allowed to use

Linear Bounded Automaton (LBA)

Input string



Working space
in tape

Left-end
marker

Right-end
marker

All computation is done between end markers

We define LBA's as NonDeterministic

Open Problem:

NonDeterministic LBA's
have same power as
Deterministic LBA's ?

Example languages accepted by LBAs:

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

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

LBA's have more power than PDA's
(pushdown automata)

LBA's have less power than Turing Machines

Unrestricted Grammars:

Productions

$$u \rightarrow v$$



String of variables
and terminals

String of variables
and terminals

Example unrestricted grammar:

$$S \rightarrow aBc$$

$$aB \rightarrow cA$$

$$Ac \rightarrow d$$

Theorem:

A language L is Turing-Acceptable if and only if L is generated by an unrestricted grammar

Context-Sensitive Grammars:

Productions

$$u \rightarrow v$$



String of variables
and terminals

String of variables
and terminals

and: $|u| \leq |v|$

The language $\{a^n b^n c^n\}$

is context-sensitive:

$$S \rightarrow abc \mid aAbc$$

$$Ab \rightarrow bA$$

$$Ac \rightarrow Bbcc$$

$$bB \rightarrow Bb$$

$$aB \rightarrow aa \mid aaA$$

Theorem:

A language L is context sensitive
if and only if
it is accepted by a Linear-Bounded automaton

Observation:

There is a language which is context-sensitive
but not decidable

The Chomsky Hierarchy

Non Turing-Acceptable

Turing-Acceptable

decidable

Context-sensitive

Context-free

Regular

Decidable Languages

Recall that:

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

For any 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**

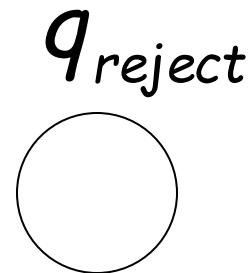
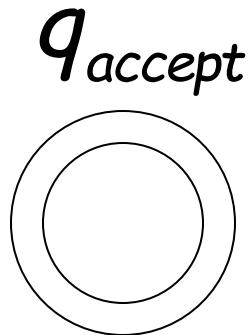
For any string w :

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

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

Every decidable language is Turing-Acceptable

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

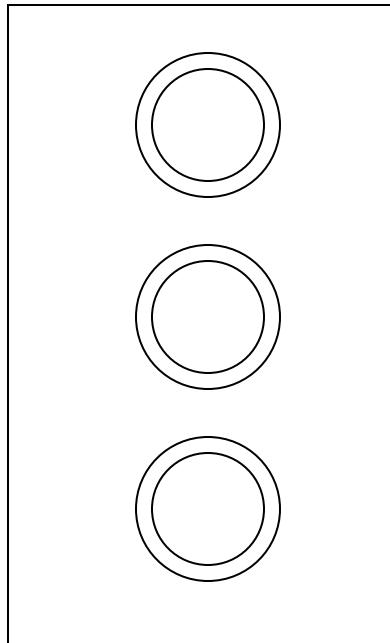


These are the only halting states

That result to possible
halting configurations

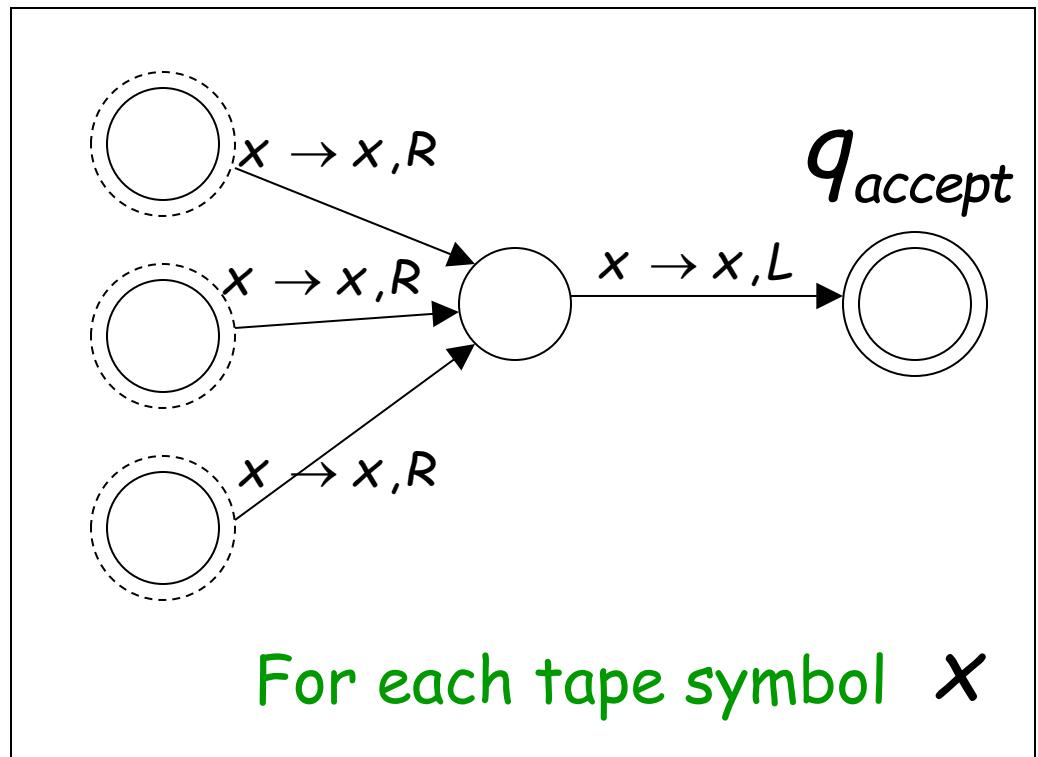
We can convert any Turing machine to have single accept and reject states

Old machine



Multiple
accept states

New machine

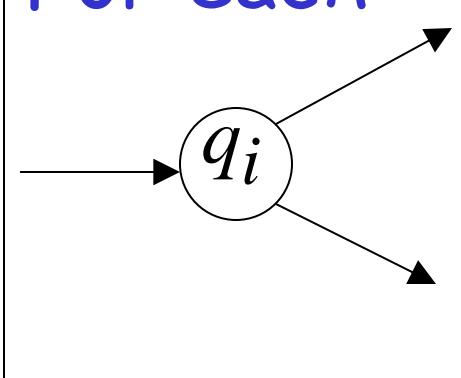


One accept state

Do the following for each possible halting state:

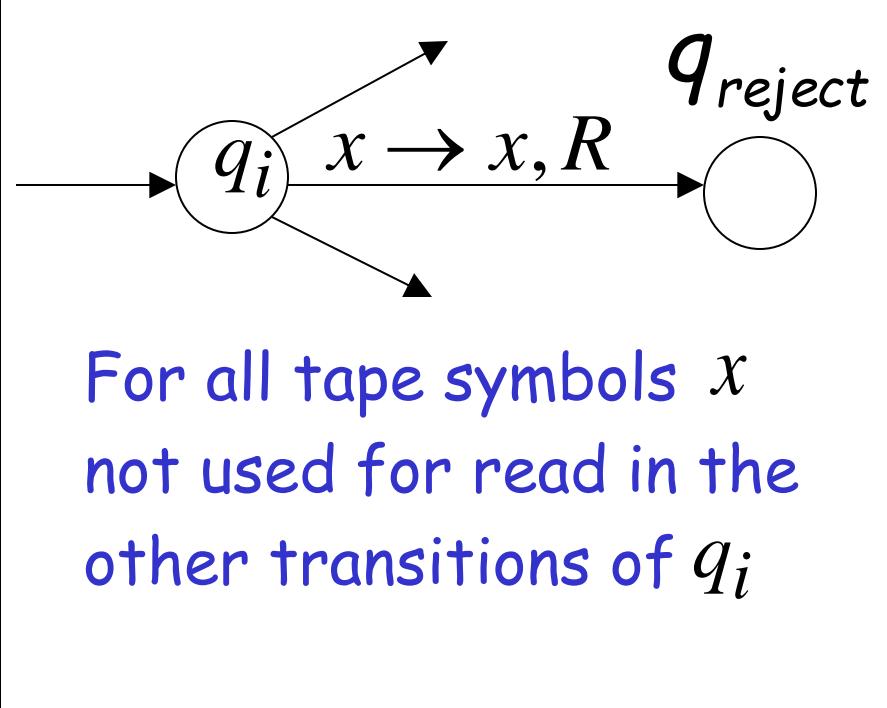
Old machine

For each



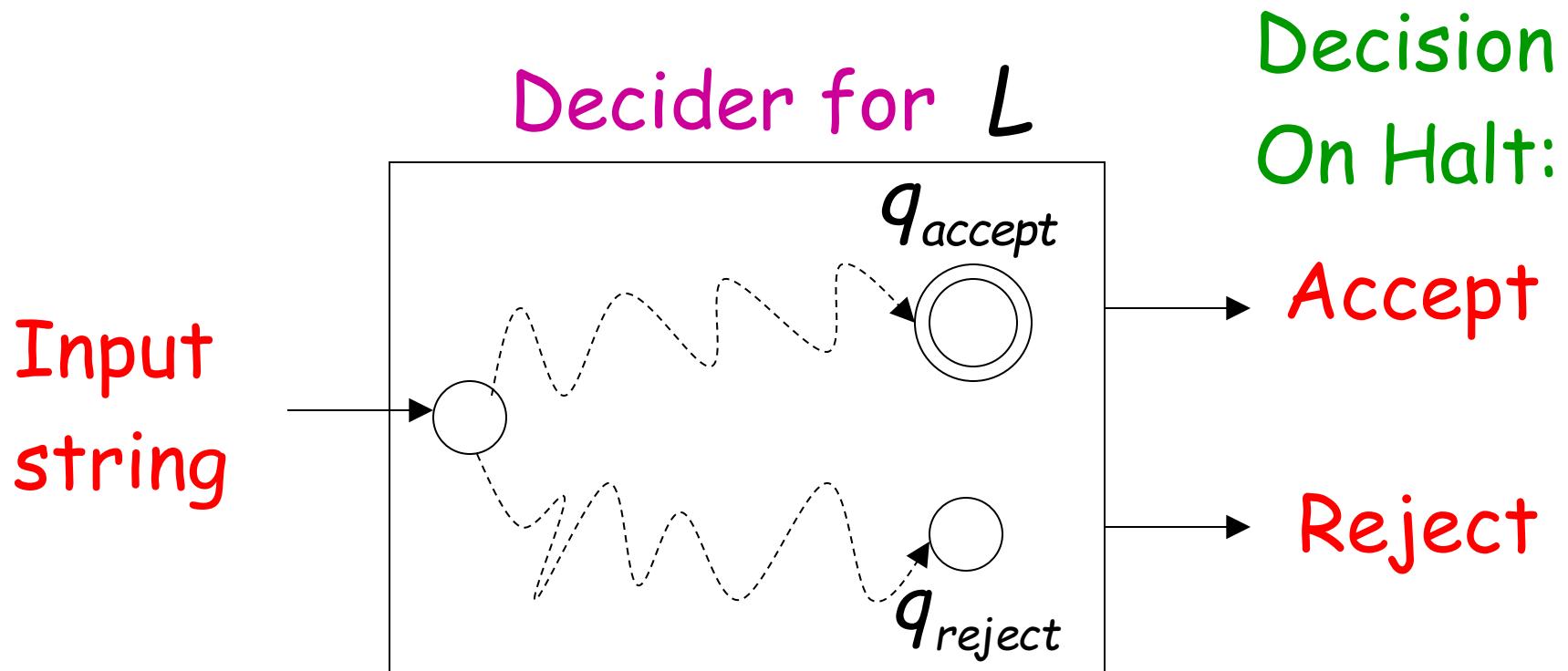
Multiple
reject states

New machine



One reject state

For a decidable language L :

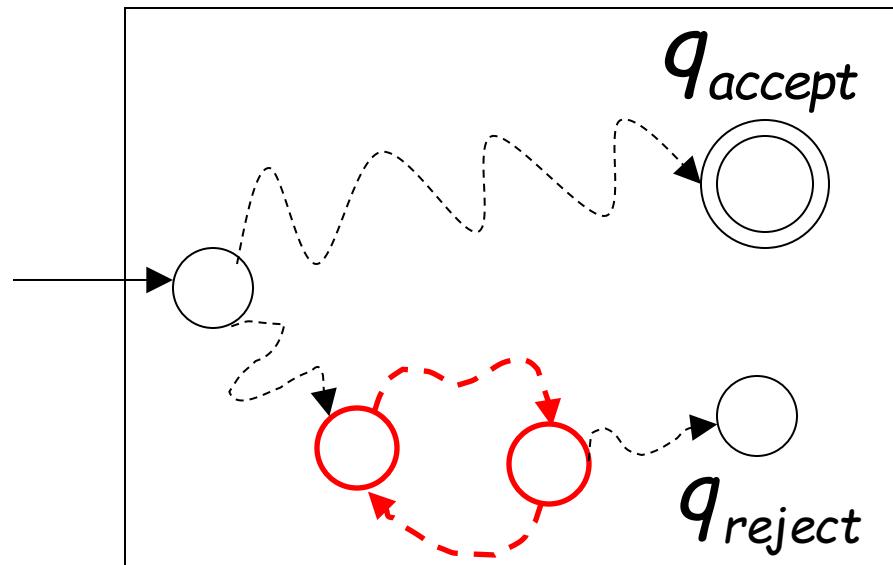


For each input string, the computation halts in the accept or reject state

For a Turing-Acceptable language L :

Turing Machine for L

Input
string



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

Problem: Is number x prime?

Corresponding language:

$$\text{PRIMES} = \{1, 2, 3, 5, 7, \dots\}$$

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

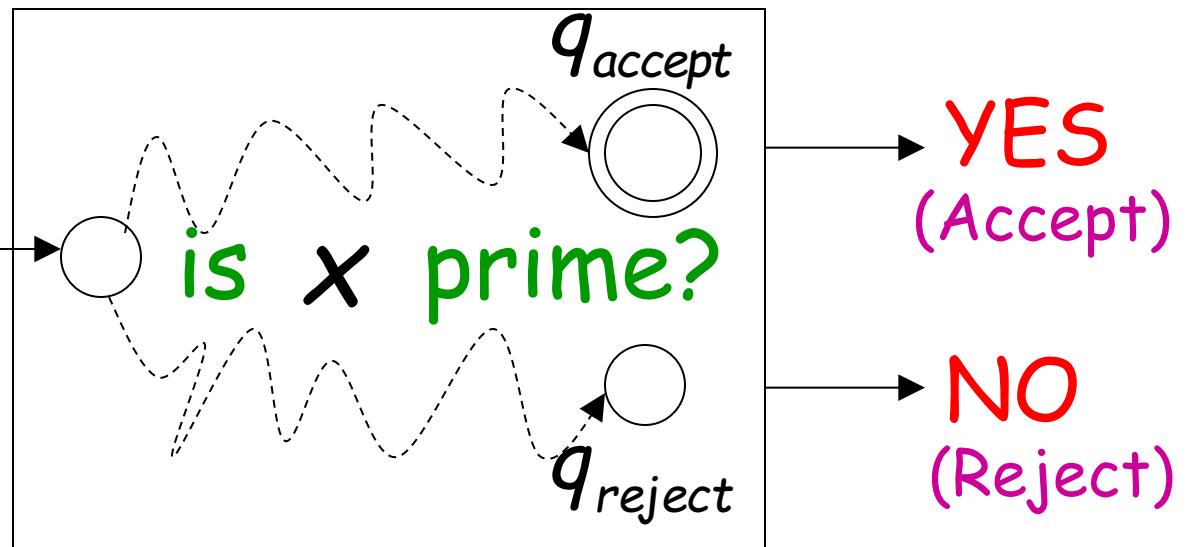
Then reject

Else accept

the decider for the language
solves the corresponding problem

Decider for PRIMES

Input number x
(Input string)



Theorem:

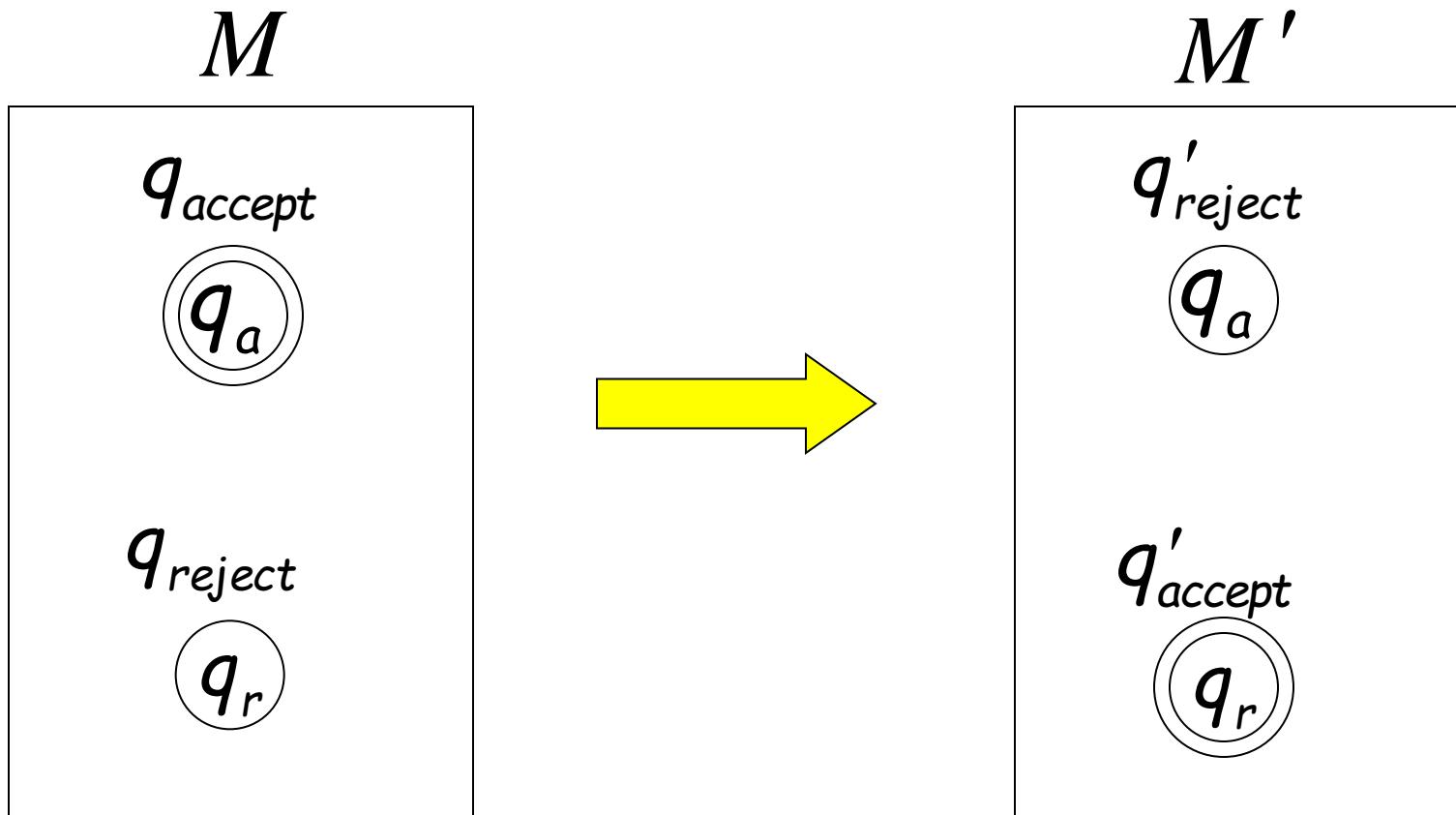
If a language L is decidable,
then its complement \bar{L} is decidable too

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 \bar{L} and halts on every input string

END OF PROOF

Undecidable Languages

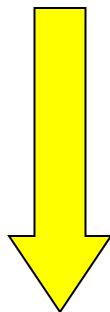
An undecidable language has no decider:
each Turing machine that accepts L
does not halt on some input string

We will show that:

There is a language which is
Turing-Acceptable and undecidable

We will prove that there is a language L :

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



the complement of a
decidable language is decidable

Therefore, L is undecidable

Non Turing-Acceptable \overline{L}

Turing-Acceptable L

Decidable

A Language which
is not
Turing Acceptable

Consider alphabet $\{a\}$

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

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

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

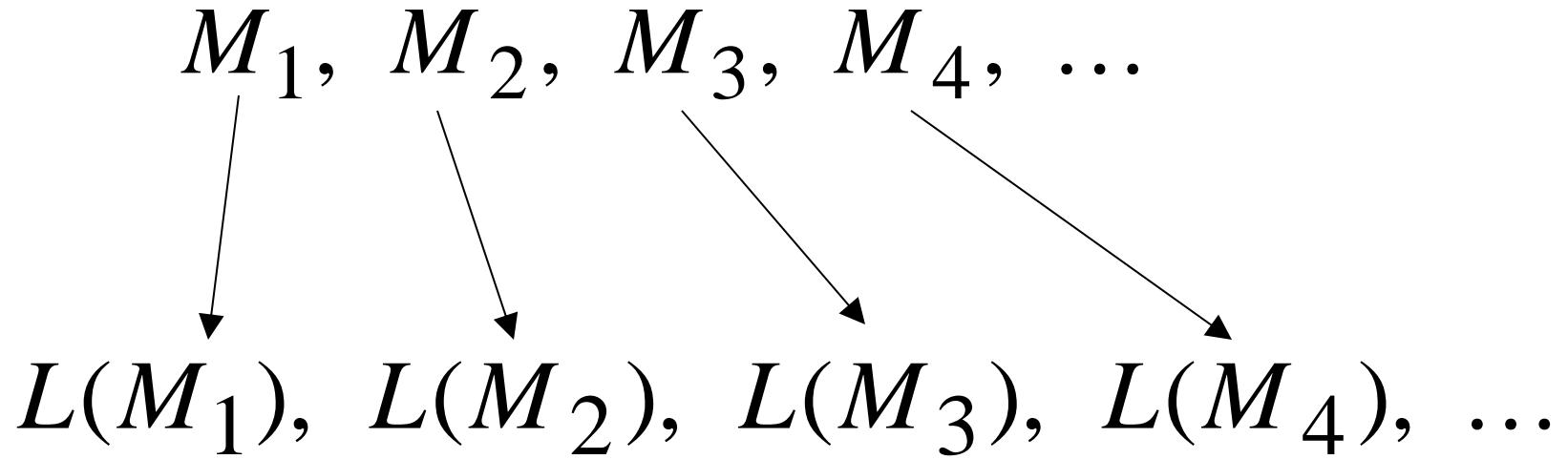
Consider Turing Machines
that accept languages over alphabet $\{a\}$

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

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

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

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	\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

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

	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

$$\overline{L} = \{a^1, a^2, \dots\}$$

Theorem:

Language \overline{L} is not Turing-Acceptable

Proof:

Assume for contradiction that

\overline{L} is Turing-Acceptable

There must exist some machine M_k
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

Question: $M_k = M_1$?

$$L(M_k) = \overline{L}$$

	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

$$a^1 \in L(M_k)$$

$$a^1 \notin L(M_1)$$

Answer: $M_k \neq M_1$

	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

Question: $M_k = M_2$?

$$L(M_k) = \overline{L}$$

	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

Answer: $M_k \neq M_2$

$$a^2 \in L(M_k)$$

$$a^2 \notin L(M_2)$$

	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

Question: $M_k = M_3$?

$$L(M_k) = \overline{L}$$

	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

Answer: $M_k \neq M_3$

$$a^3 \notin L(M_k)$$

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

Similarly: $M_k \neq M_i$ for any i

Because either:

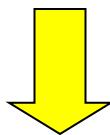
$$a^i \in L(M_k)$$

or

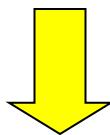
$$a^i \notin L(M_i)$$

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

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



the machine M_k cannot exist



\overline{L} is not Turing-Acceptable

End of Proof

Non Turing-Acceptable

\overline{L}

Turing-Acceptable

Decidable

A Language which is
Turing-Acceptable
and Undecidable

We will prove that the language

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

Is Turing-
Acceptable

Undecidable

There is a
Turing machine
that accepts L

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

	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

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

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

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

End of Proof

Observation:

Turing-Acceptable

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

Not Turing-acceptable

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

(Thus, \overline{L} is undecidable)

Non Turing-Acceptable \overline{L}

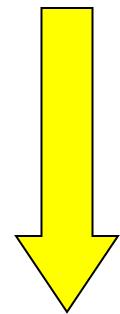
Turing-Acceptable L

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

Turing acceptable languages and Enumerators

We will prove:

(weak result)

- If a language is decidable then there is an enumerator for it

(strong result)

- A language is Turing-acceptable if and only if there is an enumerator for it

Theorem:

if a language L is decidable then
there is an enumerator for it

Proof:

Let M be the decider for L

Use M to build the enumerator for L

Let \tilde{M} be an enumerator that prints all strings from input alphabet in proper order

Example:
alphabet is $\{a, b\}$

a
 b
 aa
 ab
 ba (proper order)
 bb
 aaa
 aab
.....

Enumerator for L

Repeat:

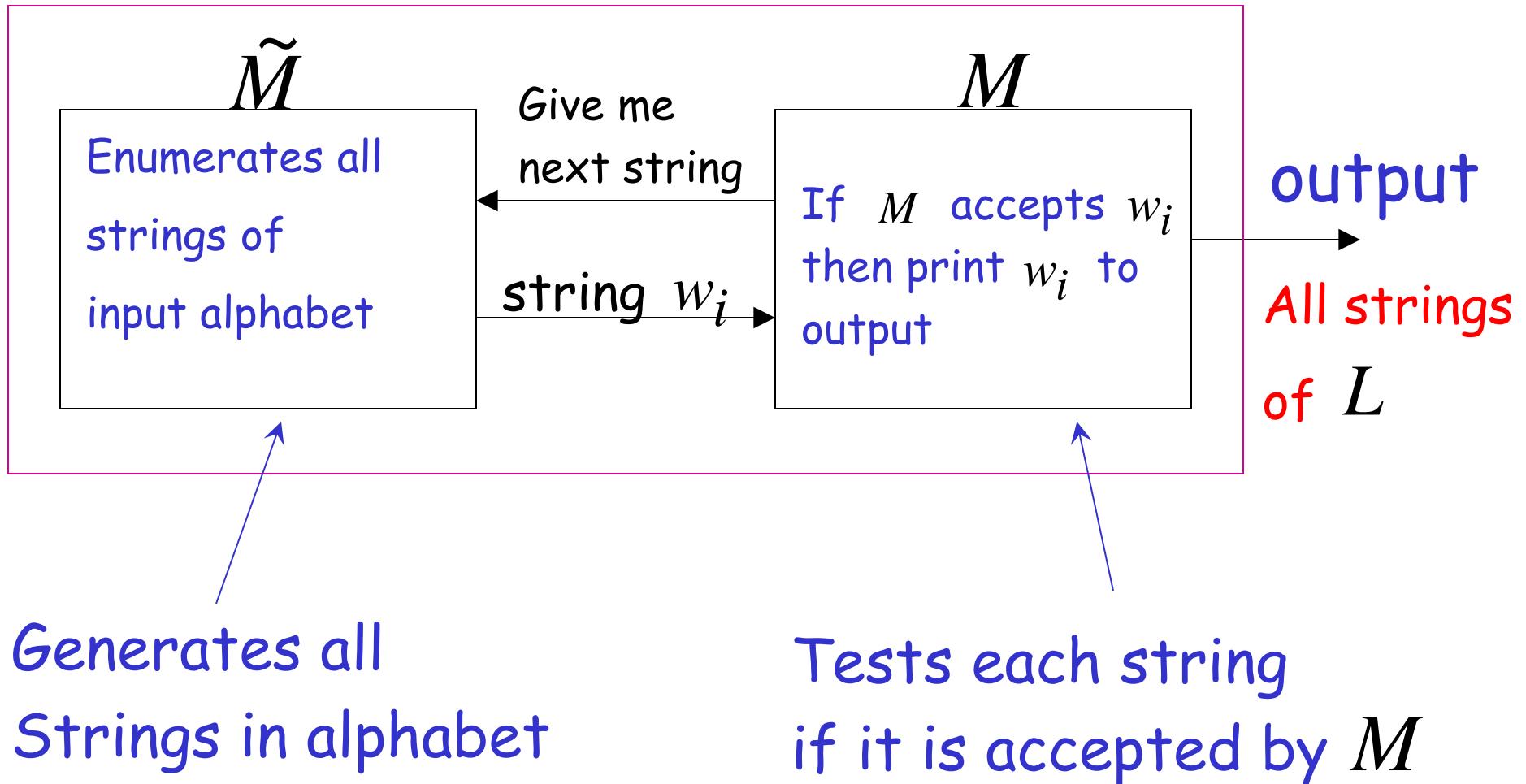
1. \tilde{M} generates a string w
2. M checks if $w \in L$

YES: print w to output

NO: ignore w

This part terminates,
because L is decidable

Enumerator for L



Example: $L = \{b, ab, bb, aaa, \dots\}$

	\tilde{M}	M	Enumeration Output
w_1	a	reject	
w_2	b	accept	b
w_3	aa	reject	
⋮	ab	accept	ab
⋮	ba	reject	
⋮	bb	accept	bb
	aaa	accept	aaa
	aab	reject	⋮
	⋮	⋮	⋮

END OF PROOF

Theorem:

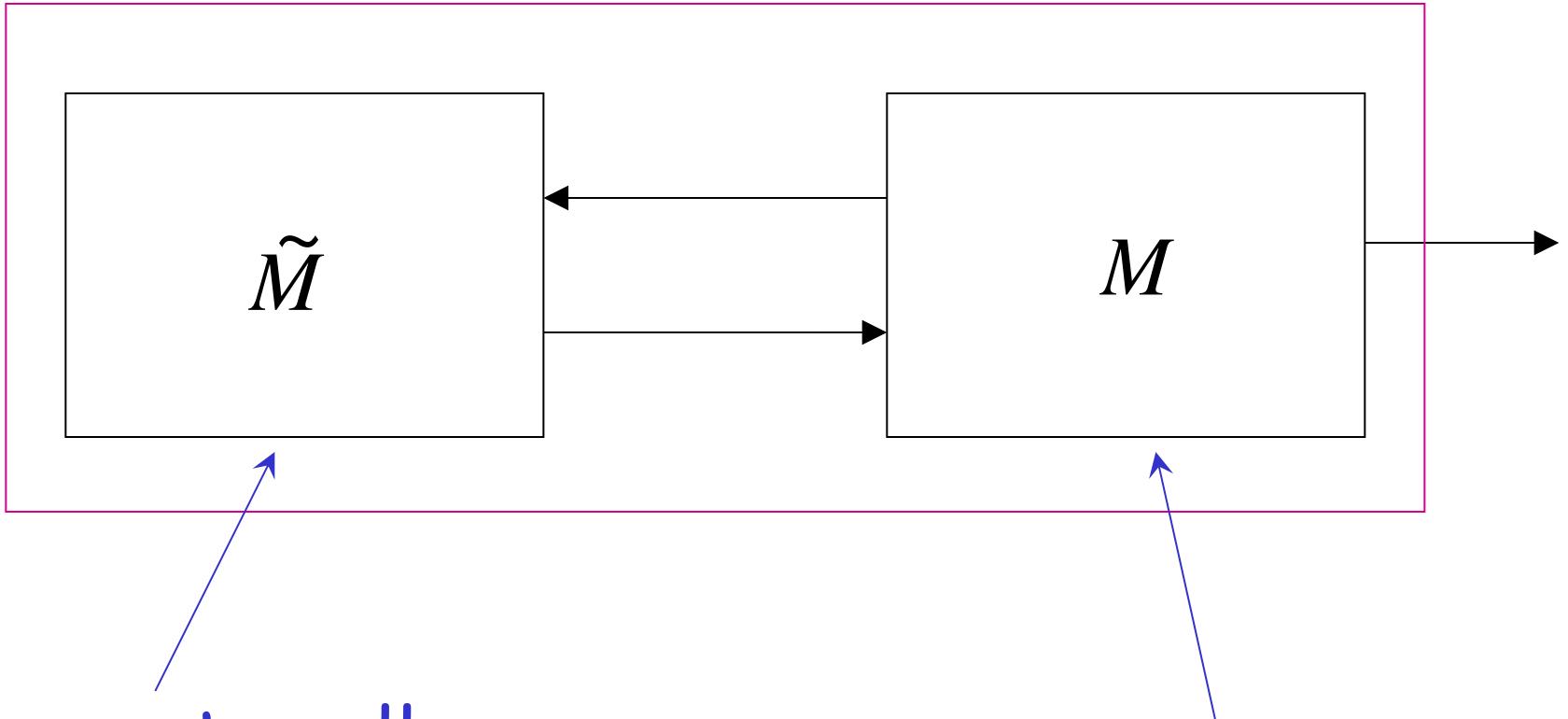
if language L is Turing-Acceptable then
there is an enumerator for it

Proof:

Let M be the Turing machine that accepts L

Use M to build the enumerator for L

Enumerator for L



Enumerates all
strings of input alphabet
in proper order

Accepts L

NAIVE APPROACH

Enumerator for L

Repeat: \tilde{M} generates a string w

M checks if $w \in L$

YES: print w to output

NO:

ignore w

Problem: If $w \notin L$

machine M may loop forever

BETTER APPROACH

\tilde{M} Generates first string w_1

M executes first step on w_1

\tilde{M} Generates second string w_2

M executes first step on w_2

second step on w_1

\tilde{M} Generates third string w_3

M executes first step on w_3

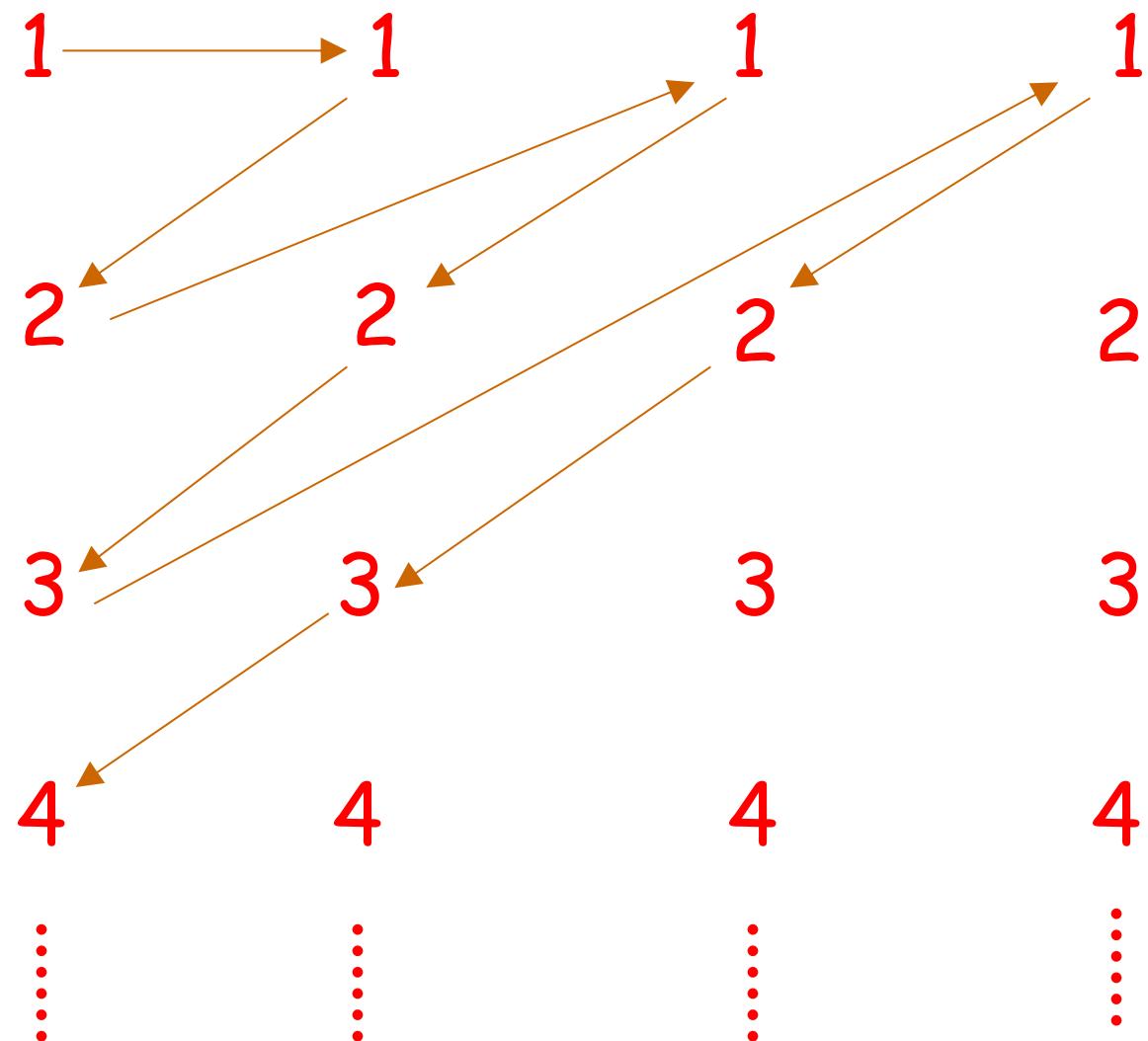
second step on w_2

third step on w_1

And so on.....

String: w_1 w_2 w_3 w_4

Step in
computation
of string



If for any string w_i
machine M halts in an accepting state
then print w_i on the output

End of Proof

Theorem:

If for language L
there is an enumerator
then L is Turing-Acceptable

Proof:

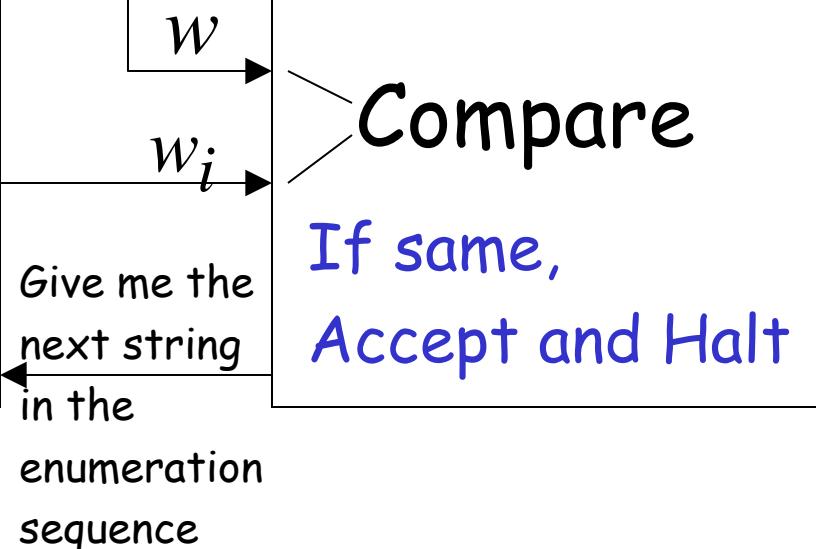
Using the enumerator for L
we will build a Turing machine
that accepts L

Input Tape

w

Turing Machine that accepts L

Enumerator
for L



Turing machine that accepts L

For any input string w

Loop:

- Using the enumerator of L , generate the next string of L
- Compare generated string with w
If same, accept and exit loop

End of Proof

By combining the last two theorems,
we have proven:

A language is Turing-Acceptable
if and only if
there is an enumerator for it