

# CS 302.1 - Automata Theory

## Lecture 01

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad



INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

H Y D E R A B A D

# Introduction

In this course, we will look at:

- **Which problems are computable?**
  - Can we characterize them?
  - What about natural problems? Computers are exotic physics experiments after all.
- **Design abstract models of computation and try to understand what problems can be solved by them.**
  - Small models that are limited in power and can solve a subset of computable problems.
  - We will build increasingly powerful computational models as we go along.
- **What are the limits of computational models?**
  - Are there problems that cannot be solved on the most powerful computers that will exist in the future.

# Introduction

In this course, we will look at:

- Which problems are computable?
- Design abstract models of computation and try to understand what problems can be solved by them.
- What are the limits of computational models?



Consider an (extremely) simple robot which

- has a button that turns it ON and OFF
- once turned on, can either move forward or backwards
- has a sensor that recognizes an obstacle and reverses the direction of the robot.

# Introduction



Consider an (extremely) simple robot which

- has a button that turns it ON and OFF
- once turned on, can either move forward or backwards
- has a sensor that recognizes an obstacle and reverses the direction of the robot.

States : {OFF, FORWARD, BACKWARD}

Inputs : {BUTTON, SENSOR}

Initial state: OFF

By accepting an INPUT (signal), the robot TRANSITIONS from one state to another

# Introduction



States : {OFF, FORWARD, BACKWARD}

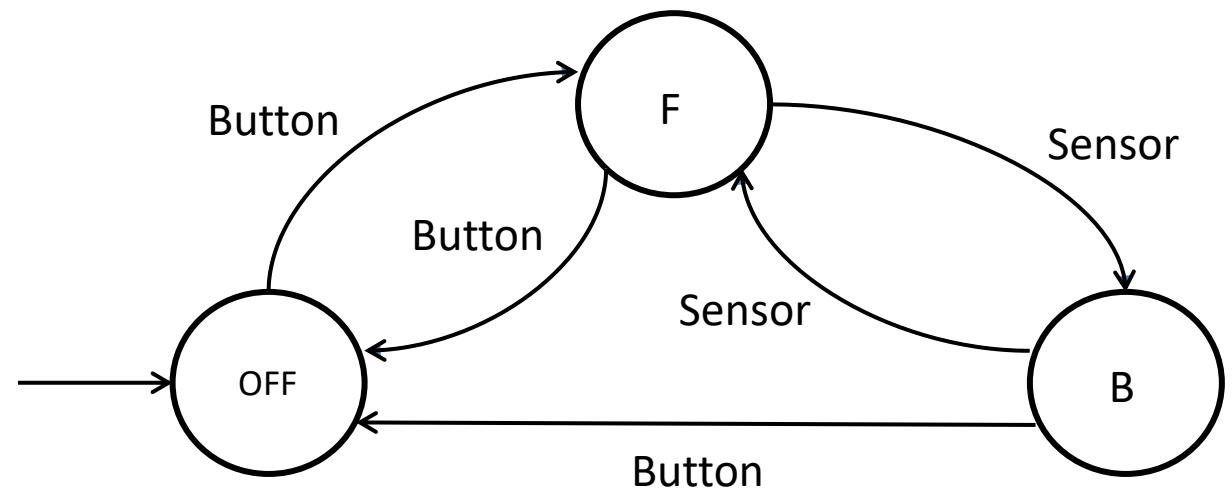
Inputs : {BUTTON, SENSOR}

Initial state: OFF

By accepting an INPUT (signal), the robot TRANSITIONS from one state to another

	BUTTON	SENSOR
OFF	F	X
F	OFF	B
B	OFF	F

State Transition Table

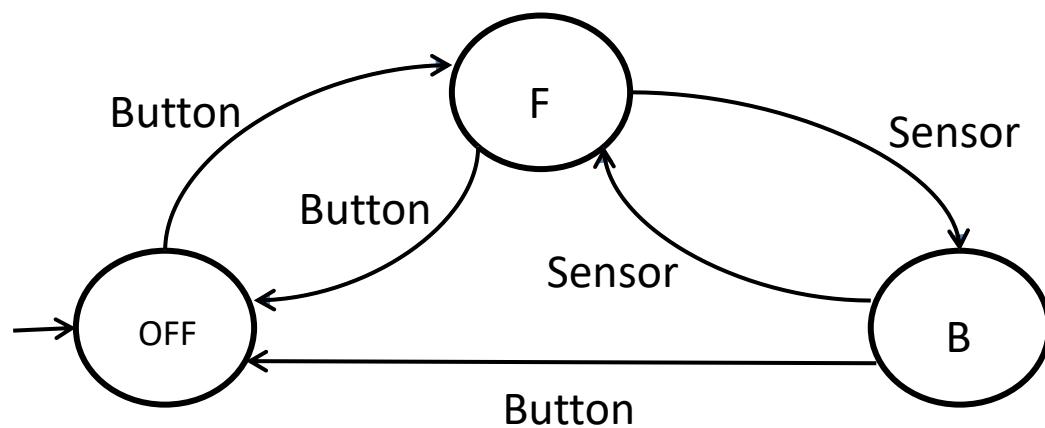


State diagram for the robot

# Introduction



	BUTTON	SENSOR
OFF	F	X
F	OFF	B
B	OFF	F



- Often computational tasks do not require an all powerful computer
- Examples: this robot, elevators, automatic doors, vending machines, ATMs etc.
- Design computational models with varying degrees of power and classify them.
- For a particular computational model, try to classify all the *problems* that can be solved by the model and those that can't be.

# Introduction

In this course, we will ask questions such as :

- Can a given problem be computed by a particular computational model?

Let us explore what is meant by this.

Problem	Problem Instance
$\int f(x)dx$	$\int \sin x dx$
Sorting	$\frac{\pi}{3}, \frac{1}{2}, 2, \dots$

Problem vs a specific instance of a problem

**Problem vs decision problem:** In order to answer these questions, we will always convert a given problem into a *decision (YES-NO) problem*. We will always do this!

# Introduction

- Can a given problem be computed by a particular computational model?

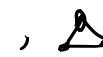
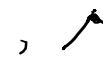
**Problem vs decision problem:** In order to answer these questions, we will always convert a given problem into a *decision (YES-NO) problem*. We will always do this!

Problem	Decision problem
Sorting	Is the array sorted?
Graph connectivity	Is the graph connected?

By converting a problem into a decision problem is that we obtain two sets :

A YES set containing all the *instances* where the answer is YES.  
A NO set containing all the *instances* where the answer is NO.

Problem: Graph Connectivity

YES set : {  ,  ,  , ..... }

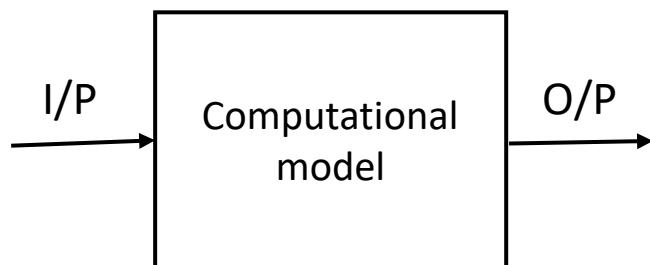
NO set : {  ,  ,  , ..... }

Given an input instance, the computer can simply check to which set it belongs to and output accordingly.

# Introduction

In this course, we will also ask questions such as :

- Can a given problem be computed by a particular computational model?



A computational model solves a problem P if,

(i) For all inputs belonging to the YES instance of P, the device outputs **YES**

AND

(ii) For all inputs belonging to the NO instance of P, the device outputs **NO**.

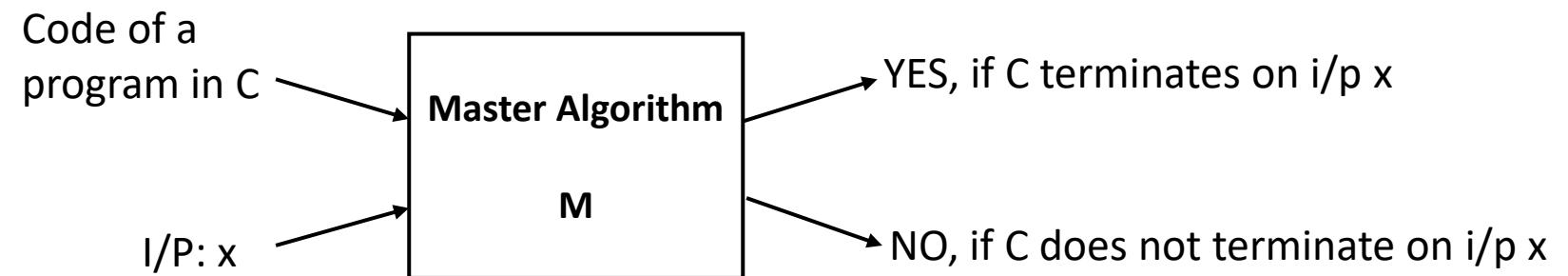
If (i) and (ii) hold, we say that the problem P is **computable** by this computational model.

# Introduction

What are the limits of computability?

Can we have problems that cannot be solved by ANY computer, no matter how powerful?

## Example 1: Master Algorithm

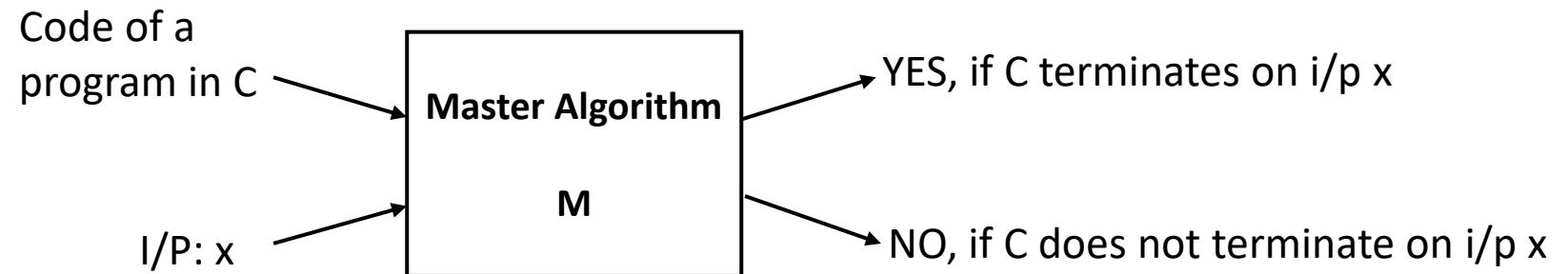


# Introduction

What are the limits of computability?

Can we have problems that cannot be solved by ANY computer, no matter how powerful?

## Example 1: Master Algorithm

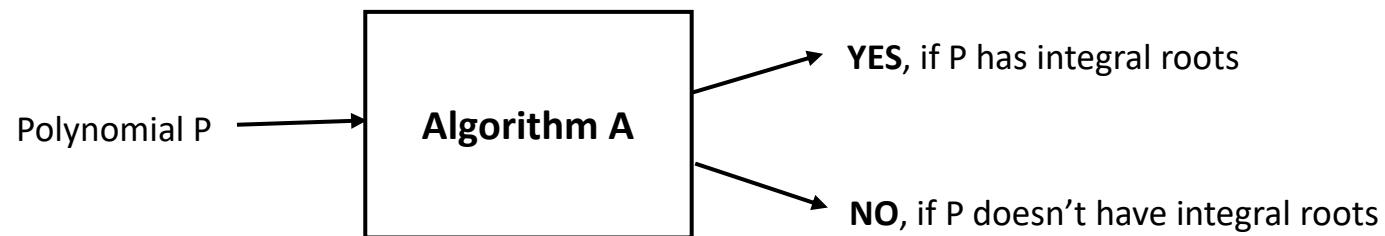


- **M terminates and outputs NO even if  $C(x)$  runs infinitely!**
- No such Algorithm M can be written. **Undecidable problem!**

**Key takeaway:** There are problems that are **not computable**.

# Introduction

**Example 2: Does a polynomial  $P(x,y)$  with integral coefficients have integral roots?**



Eg:      Input Polynomial P:  $x^3y^2 + xy^2 + 3x - 5 = 0$       O/P : YES as (-1,1) are solutions to P

- The algorithm A proceeds by checking whether for integers  $0, \pm 1, \pm 2, \dots$ . It terminates and outputs YES, whenever it finds the roots.
- What if P does not have integral roots? Algorithm A will run forever and will never terminate to output NO.
- **Undecidable problem! Key takeaway:** There are problems that are **not computable**.

# Introduction

In this course we will:

- We will consider different computational models and classify them based on the problems they can solve
- Start from simple models and gradually increase their power to accommodate real computers
- Identify the problems that are not computable.

In this course we will not:

- Deal with how much time or space (memory) an algorithm would need to solve a certain problem
- Classifying the hardness of computable problems falls under the purview of Complexity Theory

# Course Structure

- ❖ 13 Lectures in all
- ❖ Final Exam at the end (**35% weightage**)
- ❖ One theory assignment (**20% weightage**)
  - Assignment 1 – released after Lec 4/5 (Deadline: End of semester)
- ❖ Programming assignment (**25% weightage**)
  - Assignment 1 – Released after Lec 4 (Deadline: End of sem)
  - Assignment 2 – Released after Lec 6 (Deadline: 15 days)
- ❖ Quiz (**20% weightage**)

# Tutorials and TAs

- Tutorial sessions weekly: **Saturday 11:30 AM – 1 PM\***.
- Teaching Associates:
  - **Zeeshan Ahmed** ([zeeshan.ahmed@research.iiit.ac.in](mailto:zeeshan.ahmed@research.iiit.ac.in))
  - **Alapan Chaudhuri** ([alapan.chaudhuri@research.iiit.ac.in](mailto:alapan.chaudhuri@research.iiit.ac.in))
  - **Rutvij Menavlikar** ([rutvij.menavlikar@research.iiit.ac.in](mailto:rutvij.menavlikar@research.iiit.ac.in))
  - **Kushagra Garg** ([kushagra.garg@research.iiit.ac.in](mailto:kushagra.garg@research.iiit.ac.in))
  - **Mihir Bani** ([mihir.bani@research.iiit.ac.in](mailto:mihir.bani@research.iiit.ac.in))
  - **Rudransh Pratap Singh** ([rudransh.s@research.iiit.ac.in](mailto:rudransh.s@research.iiit.ac.in))
- Tutorial sessions **are not** just going to be doubt clearing sessions.
- Several **interesting topics** will be covered.
- **My email:** [shchakra@iiit.ac.in](mailto:shchakra@iiit.ac.in)
- **Lecture slides available at my homepage:** <https://sites.google.com/view/shchakra/teaching/m22-automata-theory>

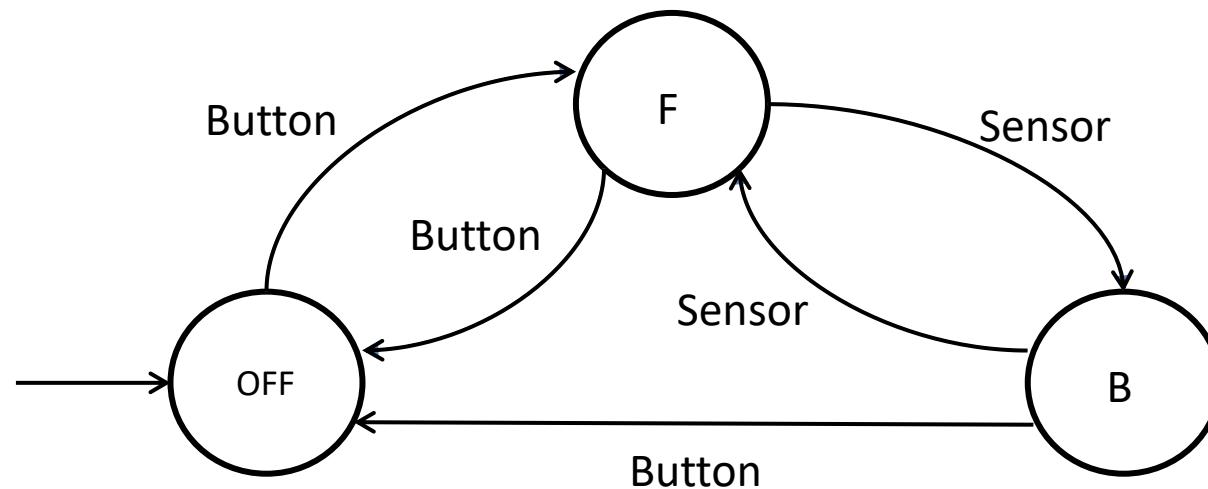
# Some terminology

Alphabet	Strings/Words	Language
Any finite, non-empty set of symbols	Finite sequence of symbols from an alphabet.	Set of words/strings from the current alphabet
$\Sigma_1 = \{0,1\}$	0110, 000, 10, 10000,.....	Even numbers
$\Sigma_2 = \{a, b, c, \dots, z\}$	any, word, revolution,.....	English

Generally the empty string is denoted by  $\epsilon$

# Models of computation

- Deterministic Finite Automata (DFA) Model

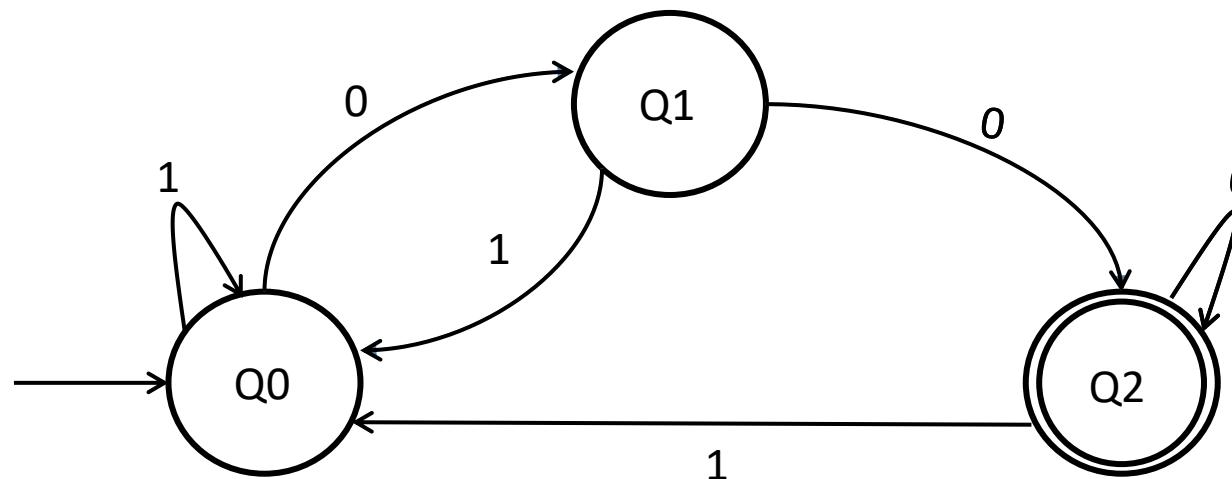


Characteristics: (i) Single start State, (ii) Unique Transitions, (iii) Zero or more final states

# Models of computation

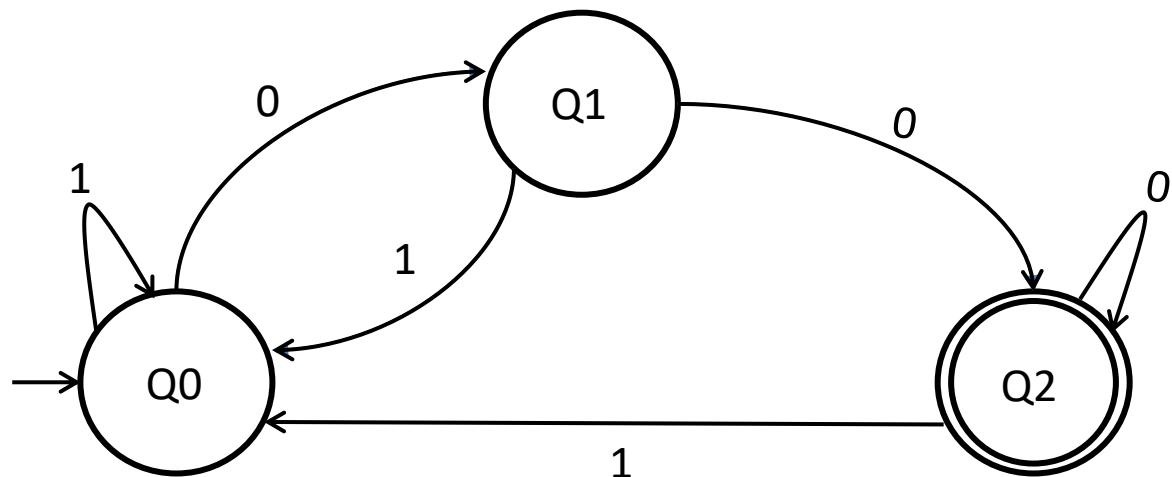
- Deterministic Finite Automata (DFA) Model

Q0: Start state, Q2: Final state



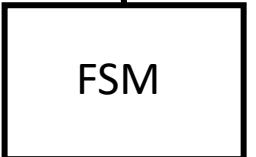
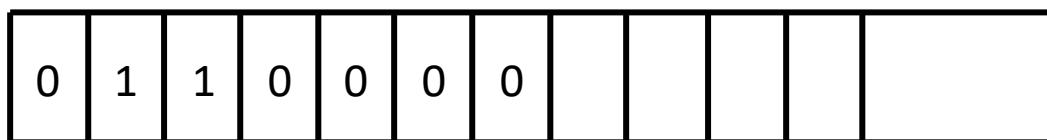
State transition diagram of the Finite State Machine

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine

One-way infinite tape



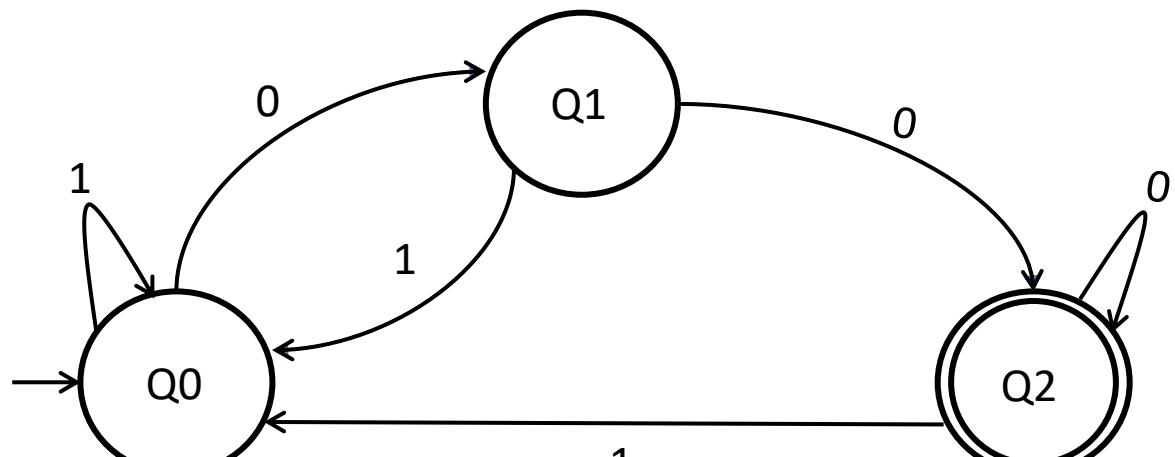
Input: Strings from alphabet  $\Sigma = \{0,1\}$

Q0: Start state, Q2: Final state

**Run:**

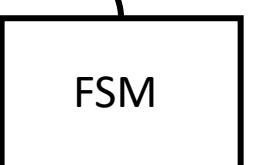
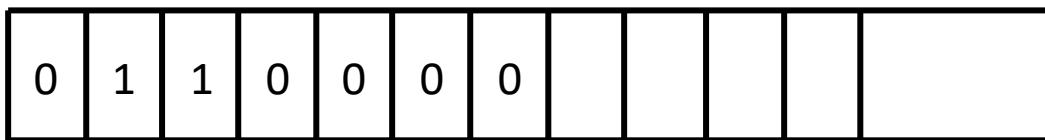
$Q0 \xrightarrow{0} Q1 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{0} Q2 \xrightarrow{0} Q2 \xrightarrow{0} Q2$

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine

One-way infinite tape



Input: Strings from alphabet  $\Sigma = \{0,1\}$

Q0: Start state, Q2: Final state

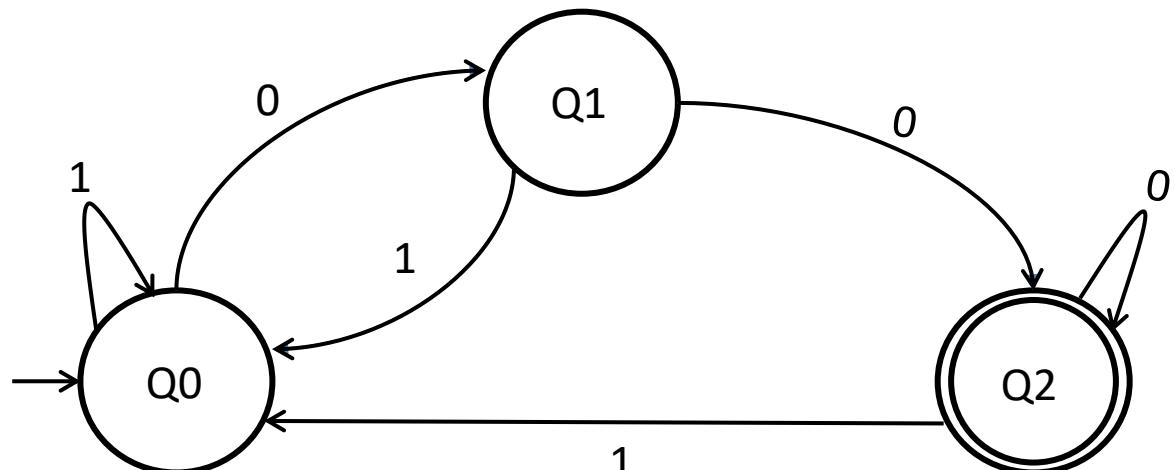
The DFA “accepts” an input string, if it corresponds to a *run* that ends up in the final state Q2. (**Accepting Run**)

The DFA “rejects” an input string, if it corresponds to a *run* that ends up in any non-final state. (**Rejecting Run**)

Run:

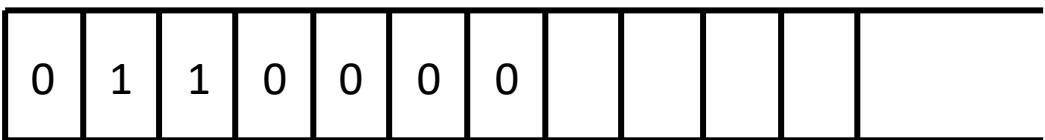
$Q0 \xrightarrow{0} Q1 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{0} Q2 \xrightarrow{0} Q2 \xrightarrow{0} Q2$

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine

One-way infinite tape



FSM

Input: Strings from alphabet  $\Sigma = \{0,1\}$

Q0: Start state, Q2: Final state

The DFA “accepts” an input string, if it corresponds to a *run* that ends up in the final state Q2. (**Accepting Run**)

The DFA “rejects” an input string, if it corresponds to a *run* that ends up in any non-final state. (**Rejecting Run**)

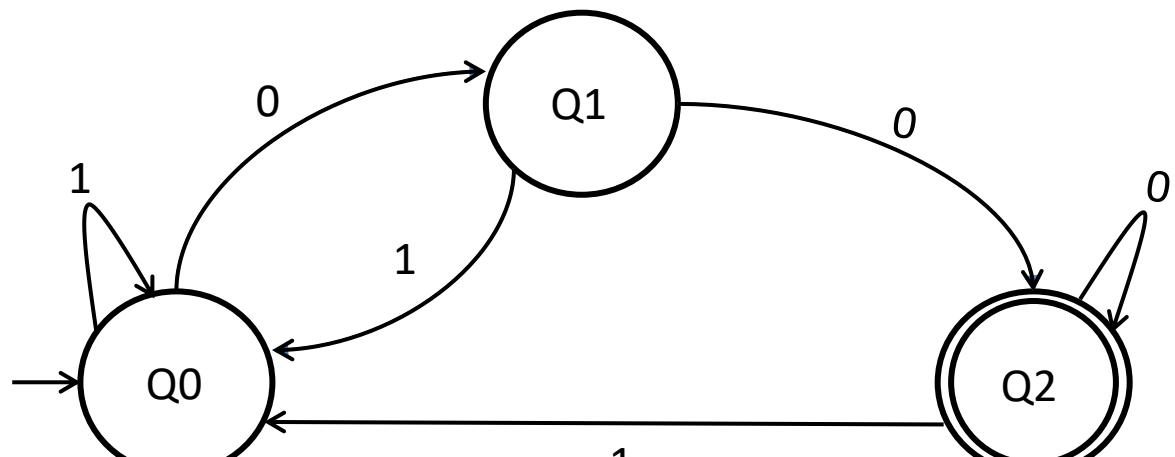
Run:

$$Q0 \xrightarrow{0} Q1 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{0} Q2 \xrightarrow{0} Q2 \xrightarrow{0} Q2$$

ACCEPT = {0110000, }

REJECT = {}

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine

One-way infinite tape



FSM

Input: Strings from alphabet  $\Sigma = \{0,1\}$

Q0: Start state, Q2: Final state

The DFA “accepts” an input string, if it corresponds to a *run* that ends up in the final state Q2. (**Accepting Run**)

The DFA “rejects” an input string, if it corresponds to a *run* that ends up in any non-final state. (**Rejecting Run**)

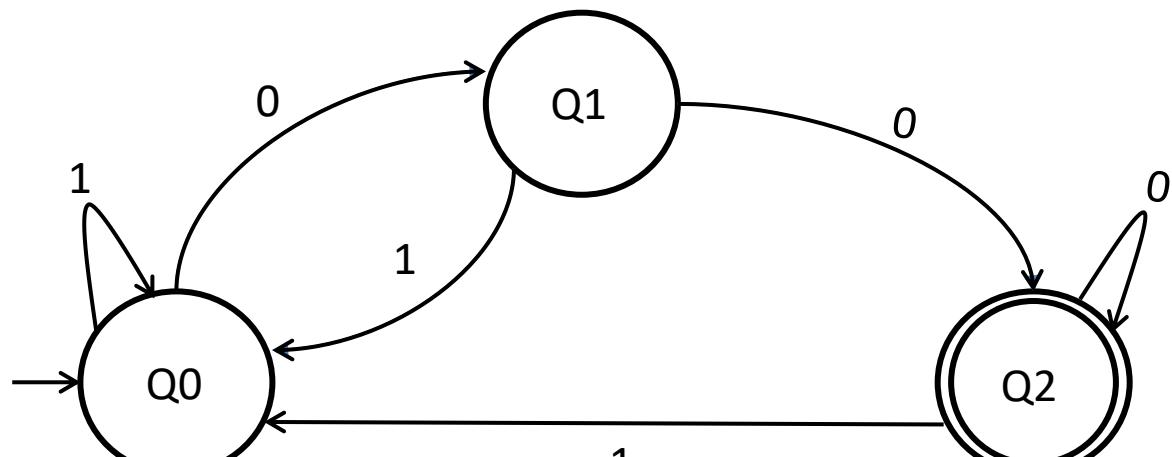
Run:

$Q0 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{1} Q0$

ACCEPT = {0111000, }

REJECT = {}

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine

One-way infinite tape



FSM

Input: Strings from alphabet  $\Sigma = \{0,1\}$

Q0: Start state, Q2: Final state

The DFA “accepts” an input string, if it corresponds to a *run* that ends up in the final state Q2. (**Accepting Run**)

The DFA “rejects” an input string, if it corresponds to a *run* that ends up in any non-final state. (**Rejecting Run**)

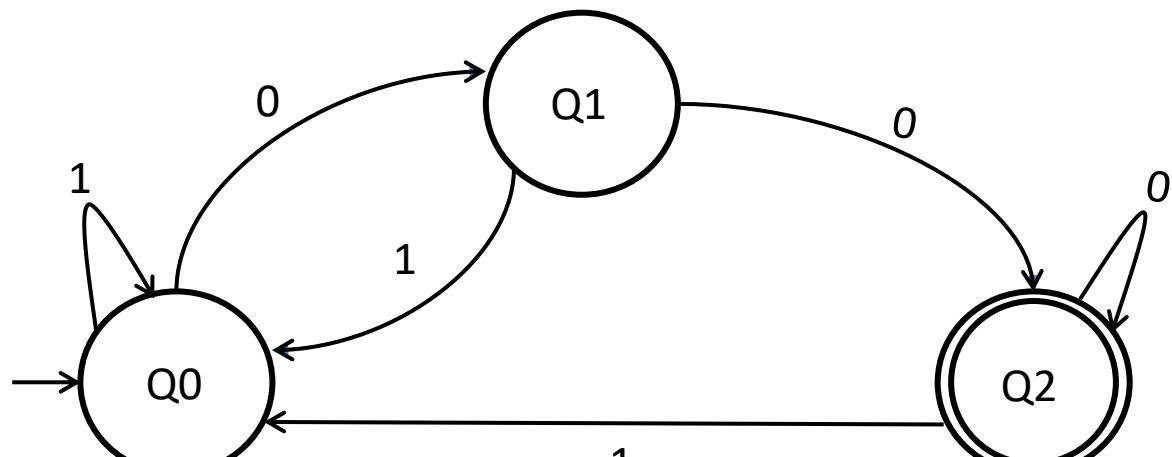
Run:

$$Q0 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{1} \textcolor{red}{Q0}$$

ACCEPT = {0111000, }

REJECT = {11101, }

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine

One-way infinite tape



FSM

Input: Strings from alphabet  $\Sigma = \{0,1\}$

Q0: Start state, Q2: Final state

The DFA “accepts” an input string, if it corresponds to a *run* that ends up in the final state Q2. (**Accepting Run**)

The DFA “rejects” an input string, if it corresponds to a *run* that ends up in any non-final state. (**Rejecting Run**)

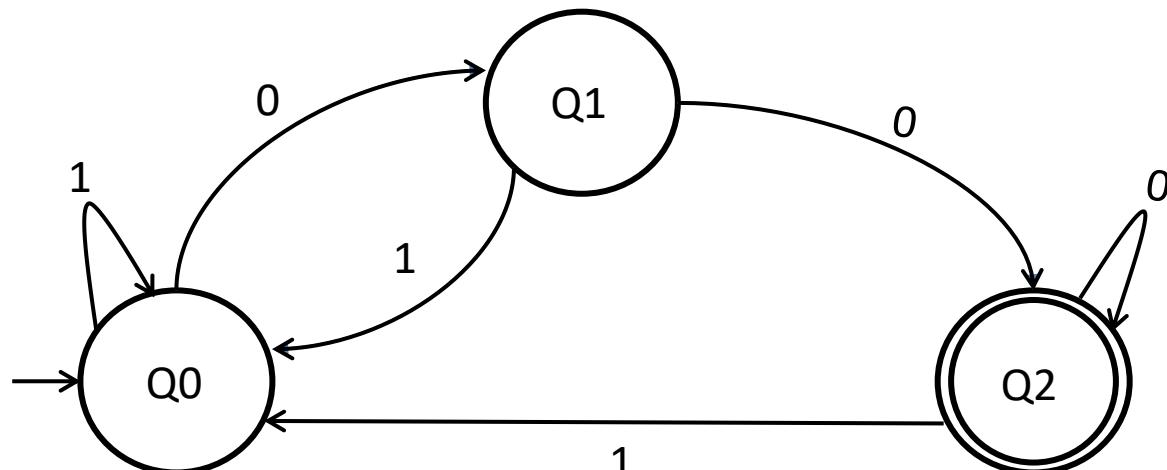
**Run:**

$$Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{0} Q2$$

ACCEPT = {0111000, 10100, ...}

REJECT = {11101, ....}

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine

One-way infinite tape



Input: Strings from alphabet  $\Sigma = \{0,1\}$

Q0: Start state, Q2: Final state

The DFA “accepts” an input string, if it corresponds to a *run* that ends up in the final state Q2. (**Accepting Run**)

The DFA “rejects” an input string, if it corresponds to a *run* that ends up in any non-final state. (**Rejecting Run**)

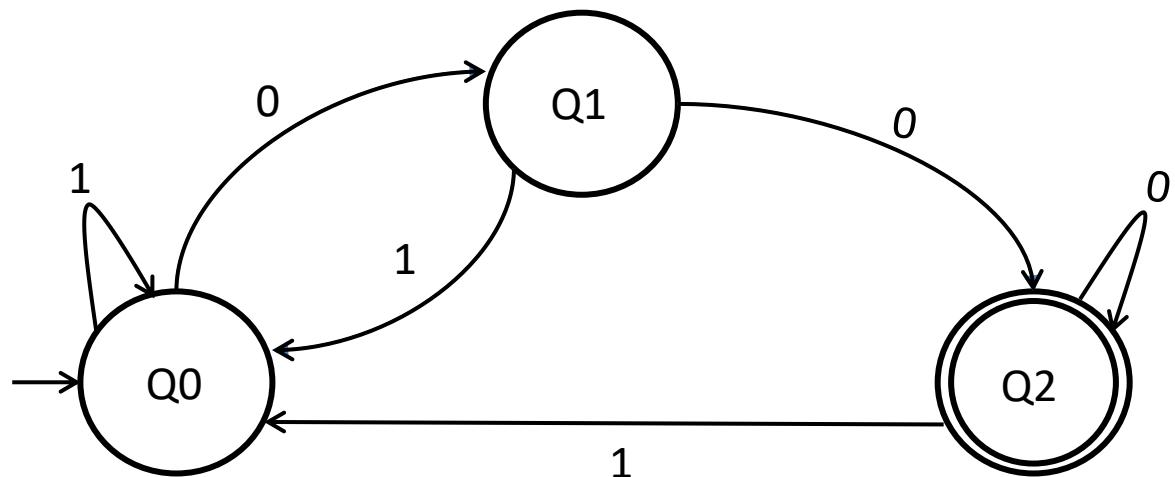
**Run:**

$$Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{0} Q2$$

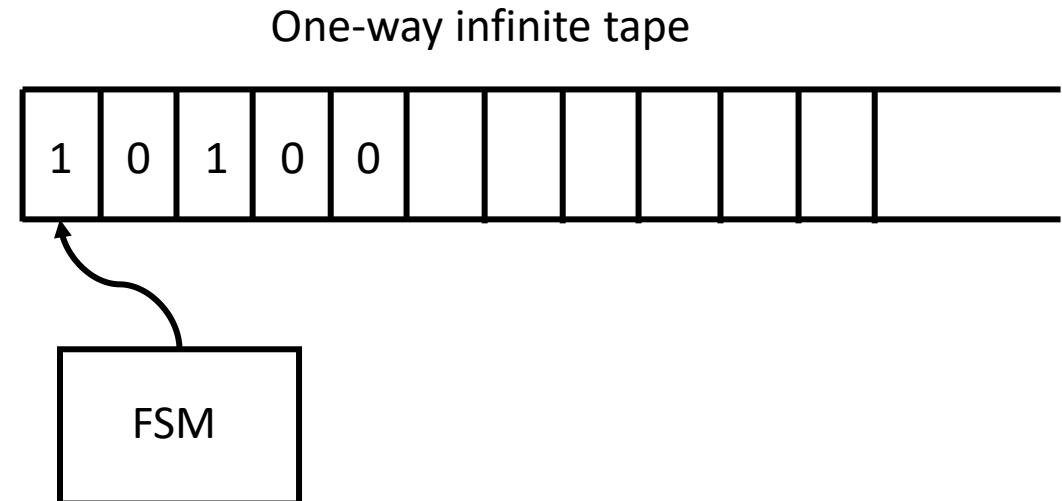
ACCEPT = {0111000, 10100, 0100, 00, 10000,...}

REJECT = {11101, 0, 1, 11, 001,.....}

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine



ACCEPT = {0111000, 10100, 0100, 00, 10000,...}  
REJECT = {11101, 0, 1, 11, 001,.....}

Let the DFA be M. Then, **language M accepts is**

$L(M) = \{\omega \mid \omega \text{ results in an accepting run}\}$ , i.e. the set of all strings  $\omega$  such that  $M(\omega)$  accepts

For the example above,  $L(M) = \{\omega \mid \omega \text{ ends in "00"}\}$

**Thank You!**

# CS 302.1 - Automata Theory

## Lecture 02

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad

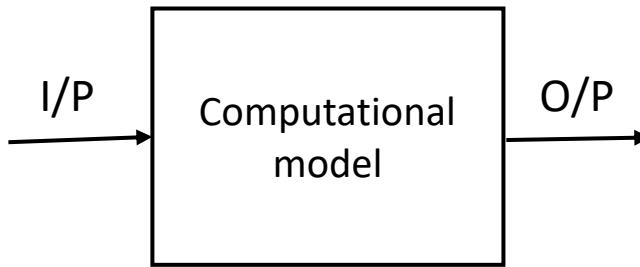


INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

H Y D E R A B A D

# A quick recap

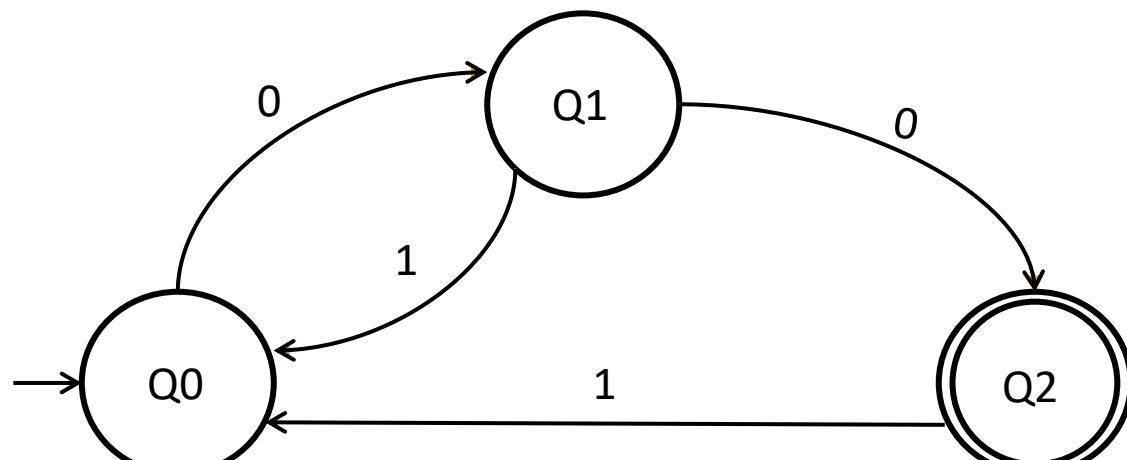
- Can a given problem be computed by a particular computational model?



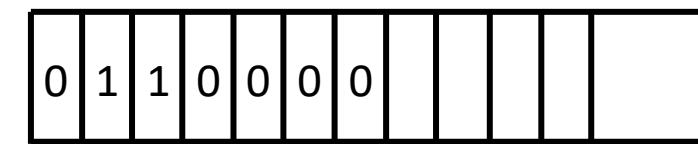
A computational model solves a problem P if,

- (i) For all inputs belonging to the YES instance of P, the device outputs **YES**
- (ii) For all inputs belonging to the NO instance of P, the device outputs **NO**.

If (i) and (ii) hold, we say that the problem **P** is **computable** by this computational model.



Deterministic Finite Automata (DFA)

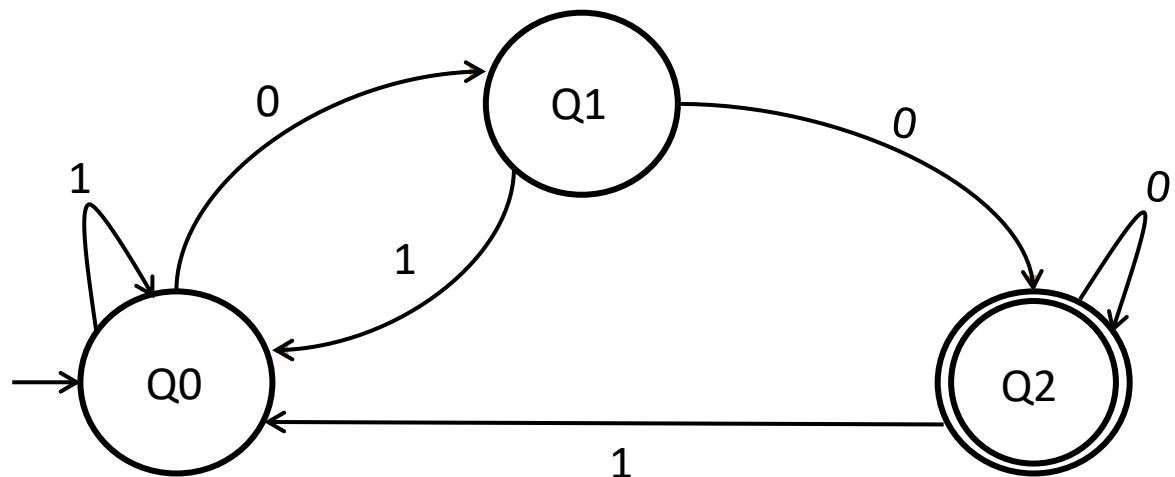


**Run:**

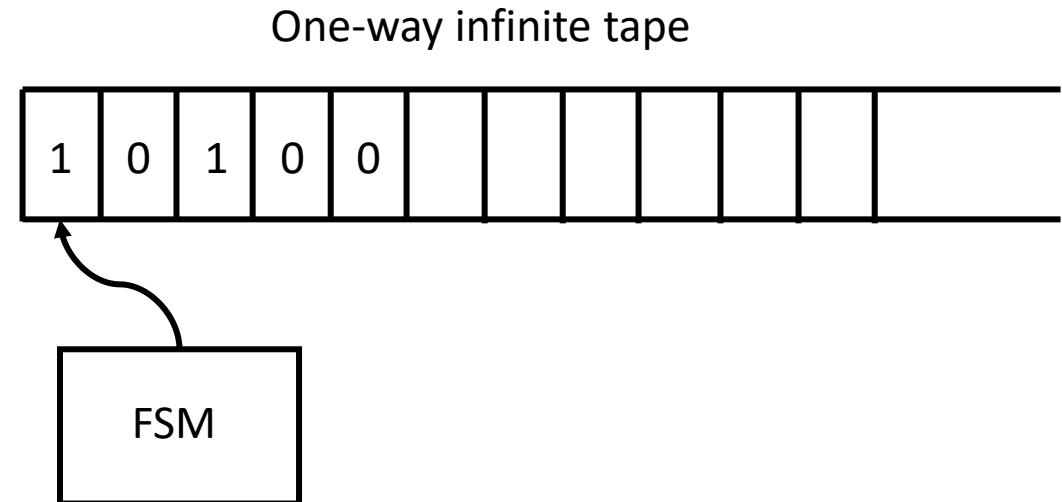
$$Q0 \xrightarrow{0} Q1 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{0} Q2 \xrightarrow{0} Q2 \xrightarrow{0} Q2$$

$$L(M) = \{\omega \mid \omega \text{ results in an accepting run}\}$$

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine



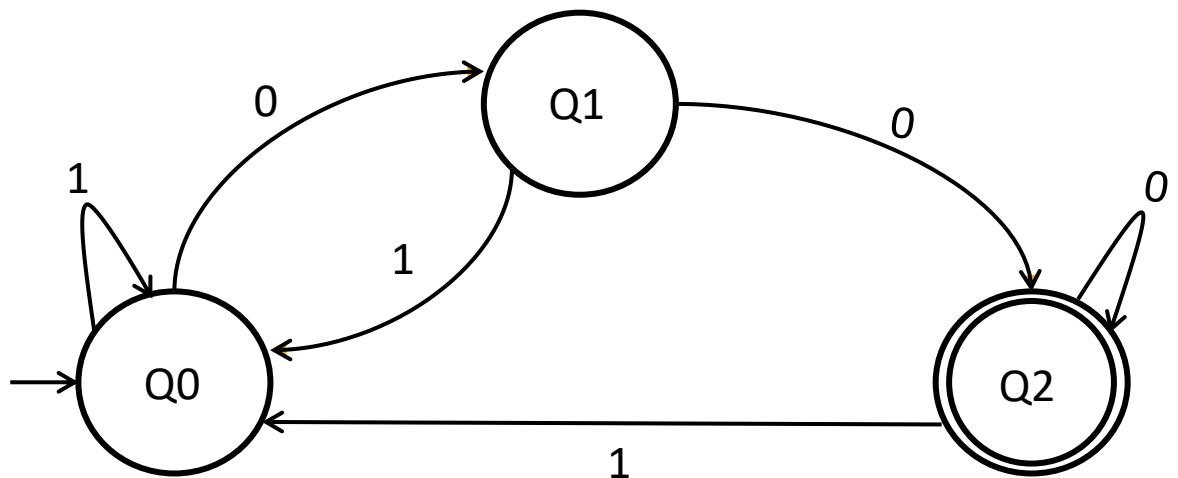
ACCEPT = {0111000, 10100, 0100, 00, 10000,...}  
REJECT = {11101, 0, 1, 11, 001,.....}

For any language  $L$ , we say  $M$  recognizes  $L$  if

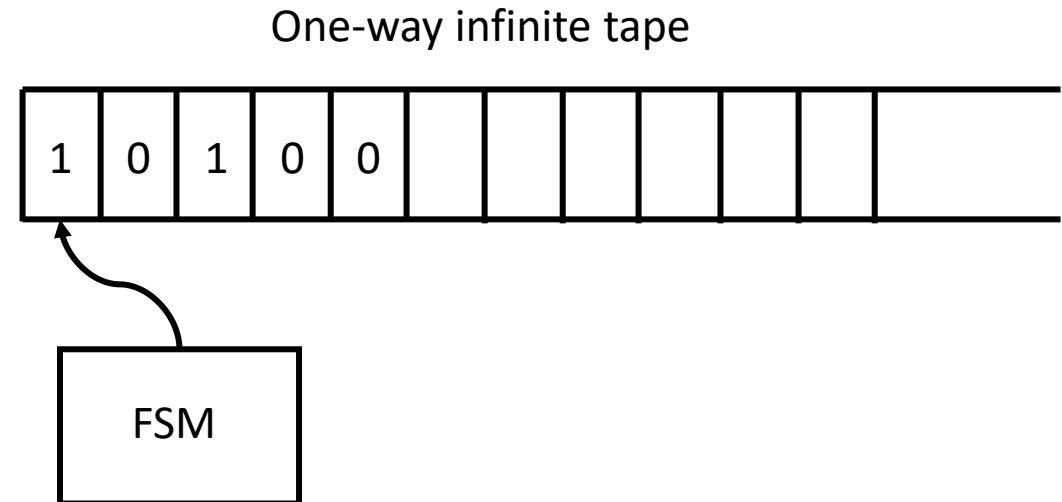
$$\forall \omega \in L, M(\omega) \text{ accepts}$$

For the example above,  $M$  recognizes  $L = \{\omega | \omega \text{ ends in "00"}\}$

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine



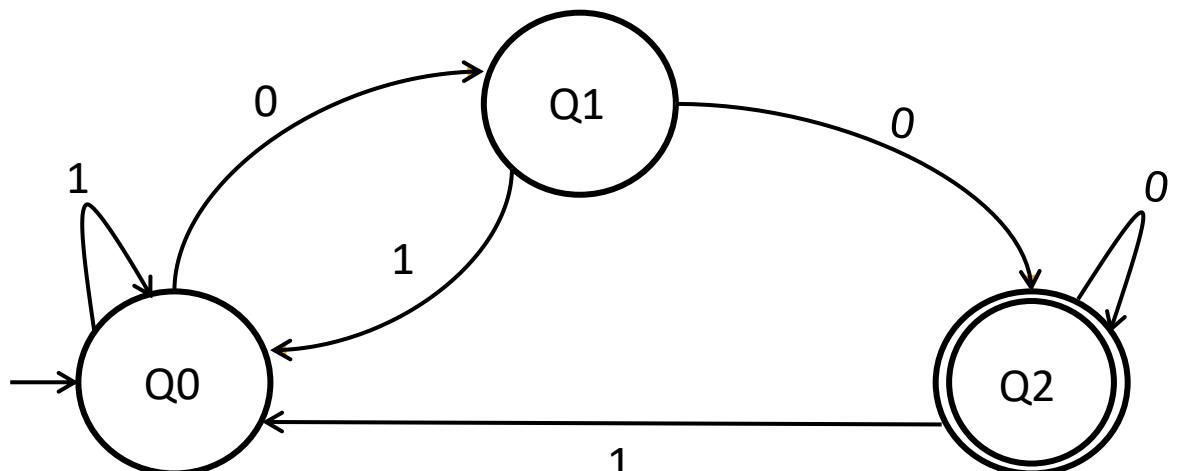
ACCEPT = {0111000, 10100, 0100, 00, 10000,...}  
REJECT = {11101, 0, 1, 11, 001,.....}

For any language  $L$ , we say **the problem  $M$  solves or  $M$  decides  $L$**  if

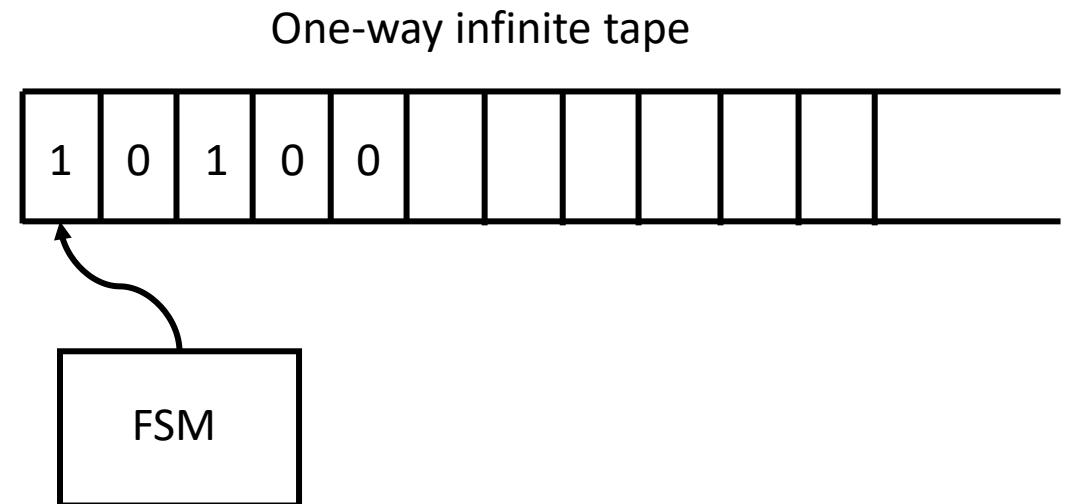
$$\begin{aligned}\forall \omega \in L, M(\omega) &\text{ accepts} \\ \forall \omega \notin L, M(\omega) &\text{ rejects}\end{aligned}$$

For the example above,  **$M$  decides  $L = \{\omega | \omega \text{ ends in "00"}\}$**

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine



For any language  $L$ , we say **M recognizes L** if

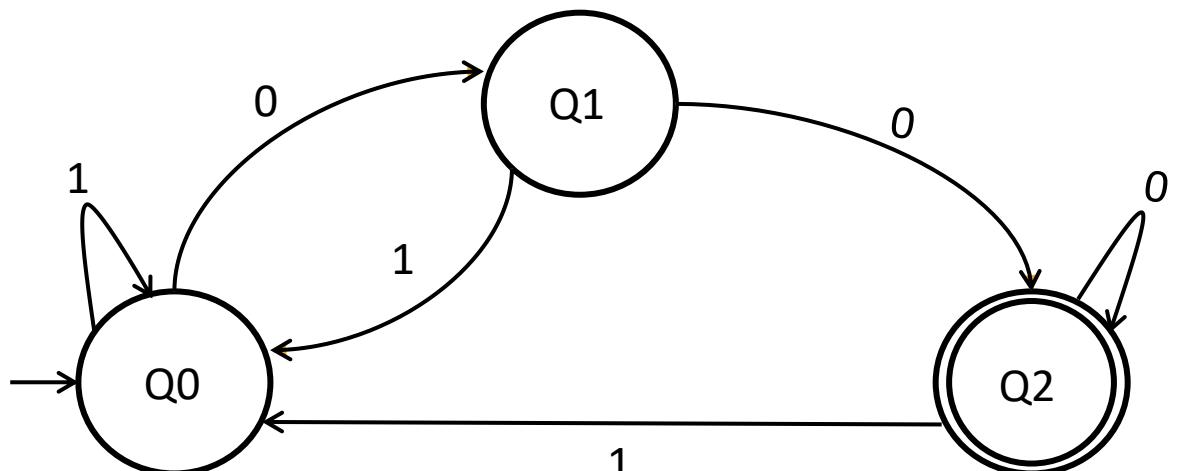
$\forall \omega \in L, M(\omega) \text{ accepts}$

For any language  $L$ , we say **M decides L** if

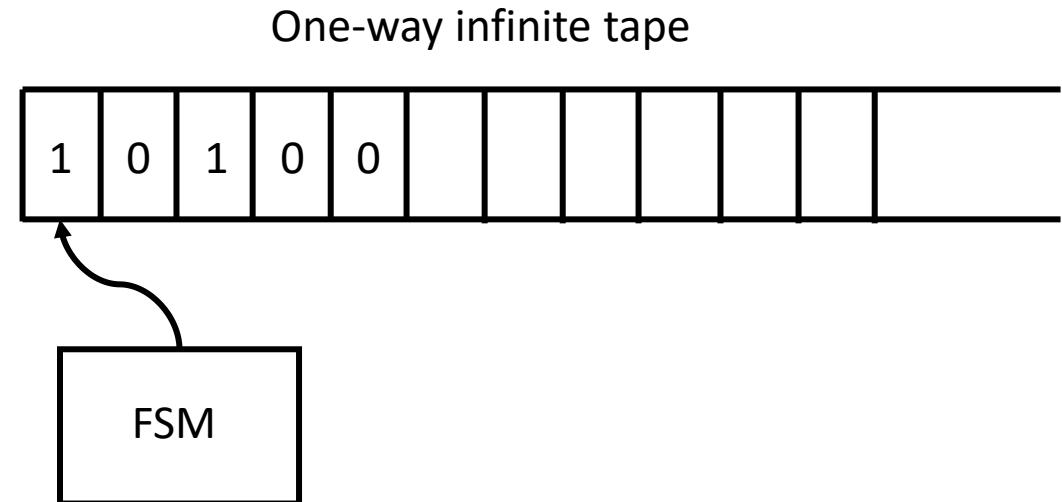
$\forall \omega \in L, M(\omega) \text{ accepts}$   
 $\forall \omega \notin L, M(\omega) \text{ rejects}$

For a DFA, the notions of **deciding a language** and **recognizing a language** are equivalent, but this may not be true for other, more powerful computational models

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine



Characteristics of DFA : (i) Single start state (ii) Unique transitions (iii) Zero or more final states

Formally, a finite automaton M is a 5-tuple  $(Q, \Sigma, \delta, q_0, F )$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is a finite set called the **alphabet**.
- $\delta: Q \times \Sigma \mapsto Q$  is the **transition function (unique)**.
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  are the **final/accepting states**.

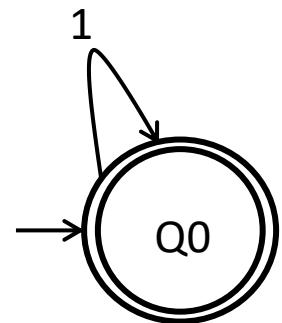
$$\begin{aligned} Q &= \{Q0, Q1, Q2\} \\ \Sigma &= \{0,1\} \\ (Q0,0) &\mapsto Q1; (Q0,1) \mapsto Q0, \dots, (Q2,1) \mapsto Q0 \\ q_0 &= Q0 \\ F &= Q2 \end{aligned}$$

# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ has an even number of } 0's\}$

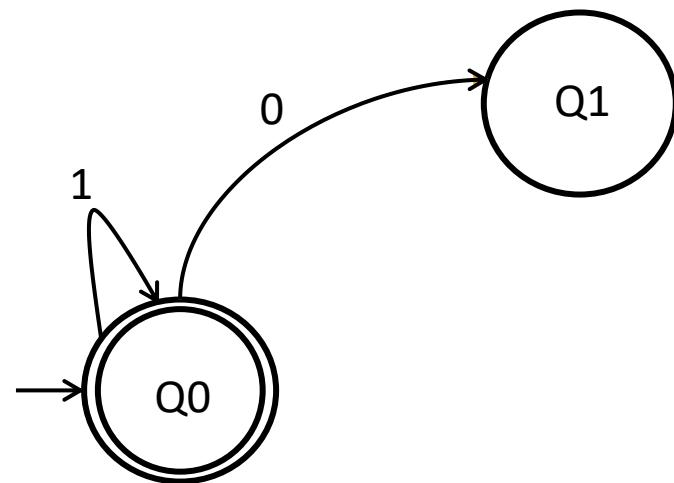
# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ has an even number of } 0's\}$



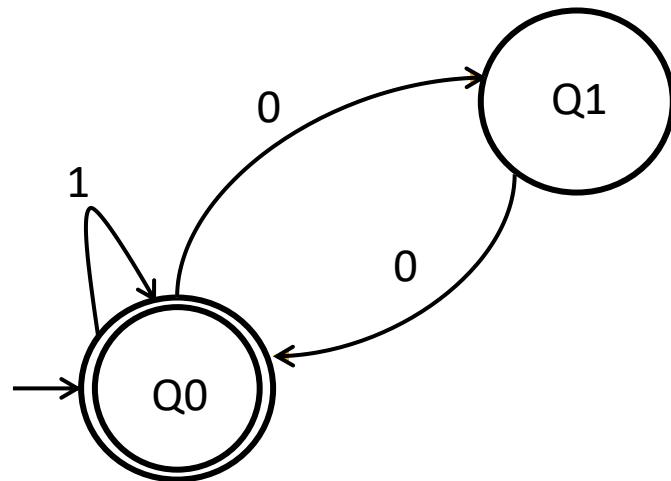
# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ has an even number of 0's}\}$



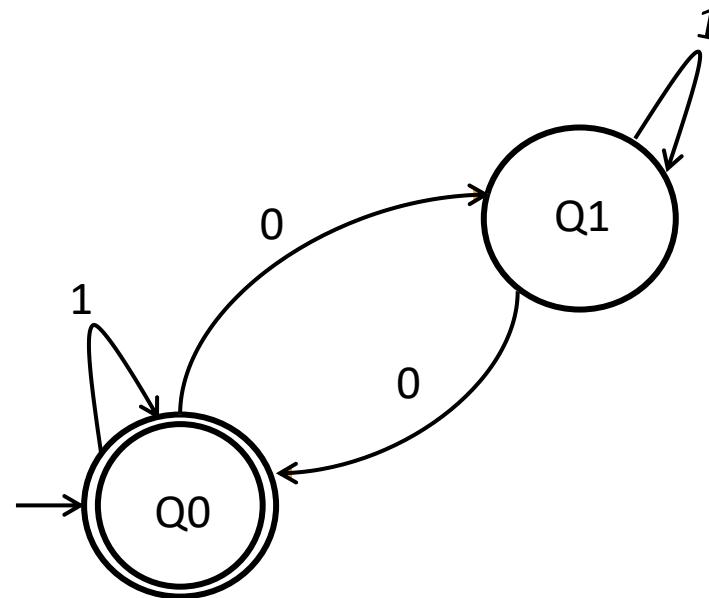
# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ has an even number of 0's}\}$



# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ has an even number of 0's}\}$



	0	1
Q0	Q1	Q0
Q1	Q0	Q1

# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ is divisible by } 3\}$

Any input string would leave three remainders: 0, 1 or 2.

# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ is divisible by } 3\}$

Any input string would leave three remainders: 0, 1 or 2.

Intuition: Let  $\omega$  be any substring of the input string divisible by 3, i.e.  $\omega = 0 \pmod{3}$

$$\omega 0 = 2 \times \text{value}(\omega) = 0 \pmod{3}$$

$$\omega 1 = 2 \times \text{value}(\omega) + 1 = 1 \pmod{3}$$

$$\omega 10 = 2 \times \text{value}(\omega 1) = 2 \pmod{3}$$

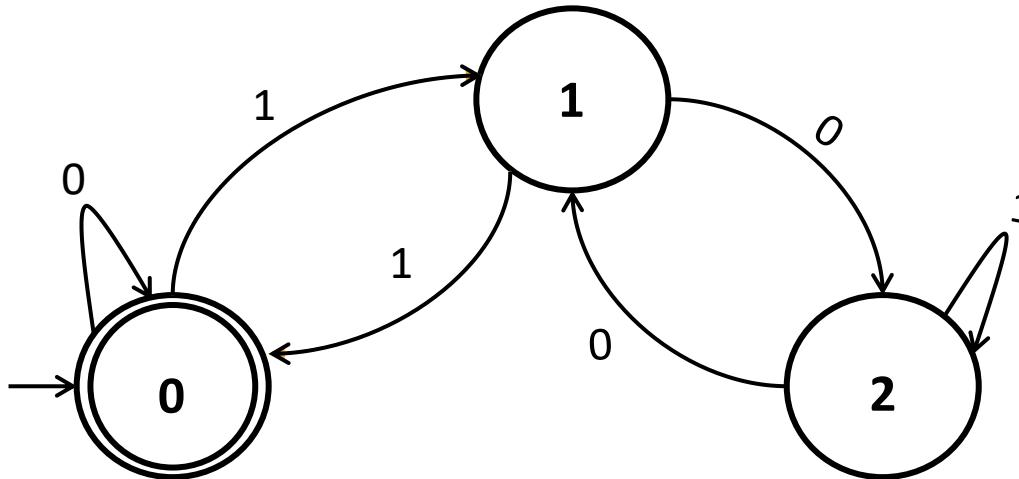
$$\omega 11 = 2 \times \text{value}(\omega 1) + 1 = 0 \pmod{3}$$

.... And so on

- The DFA will have three states, each corresponding to the remainder of  $\text{value}(\omega)/3$ .
- The final state =  $0 \pmod{3}$  – the string  $\omega$  is accepted if after reading it, the DFA ends in this state.

# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ is divisible by } 3\}$



Any input string would either leave remainders 0, 1 or 2.

Intuition: Let  $\omega$  be any substring of the input string divisible by 3, i.e.  $\omega = 0 \pmod{3}$

$$\omega 0 = 2 \times \text{value}(\omega) = 0 \pmod{3}$$

$$\omega 1 = 2 \times \text{value}(\omega) + 1 = 1 \pmod{3}$$

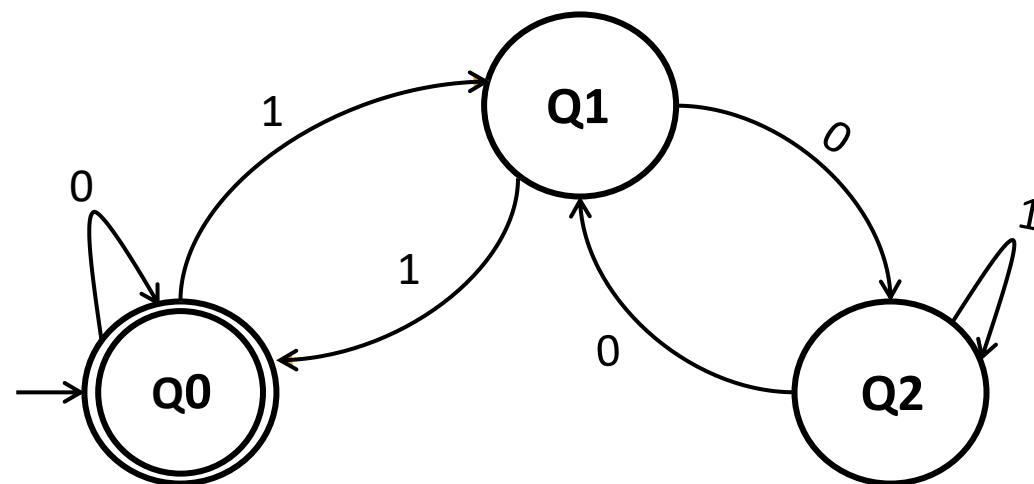
$$\omega 10 = 2 \times \text{value}(\omega 1) = 2 \pmod{3}$$

$$\omega 11 = 2 \times \text{value}(\omega 1) + 1 = 0 \pmod{3}$$

.... And so on

# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ is divisible by 3}\}$



	0	1
Q0	Q0	Q1
Q1	Q2	Q0
Q2	Q1	Q2

# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ is NOT divisible by } 3\}$

# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ is NOT divisible by } 3\}$

**Intuition** - Construct a **Toggled DFA**: Toggle the final states and the non-final states!

# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ is NOT divisible by } 3\}$

**Intuition** - Construct a **Toggled DFA**: Toggle the final states and the non-final states!

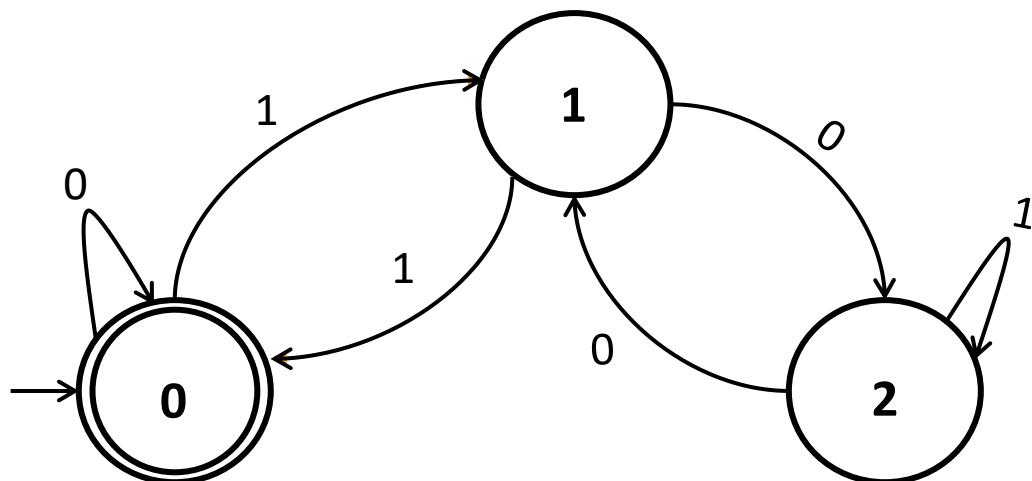
In fact if any DFA accepts  $L$ , the toggled DFA accepts  $\bar{L}$ , the complement of  $L$

# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ is NOT divisible by } 3\}$

**Intuition** - Construct a **Toggled DFA**: Toggle the final states and the non-final states!

In fact if any DFA accepts  $L$ , the toggled DFA accepts  $\bar{L}$ , the complement of  $L$

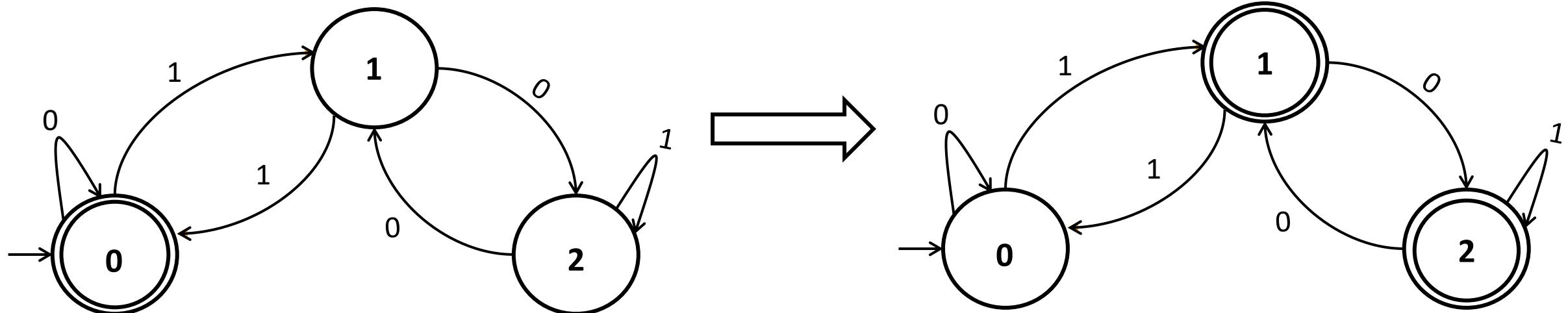


# Constructing DFA for a language

Examples:  $\Sigma = \{0, 1\}$ ,  $L(M) = \{\omega \mid \omega \text{ is NOT divisible by } 3\}$

**Intuition** - Construct a **Toggled DFA**: Toggle the final states and the non-final states!

In fact if any DFA accepts  $L$ , the toggled DFA accepts  $\bar{L}$ , the complement of  $L$



# Non-deterministic Finite Automata (NFA)

Characteristics of DFA : (i) Single start state (ii) Unique transitions (iii) Zero or more final states

# Non-deterministic Finite Automata (NFA)

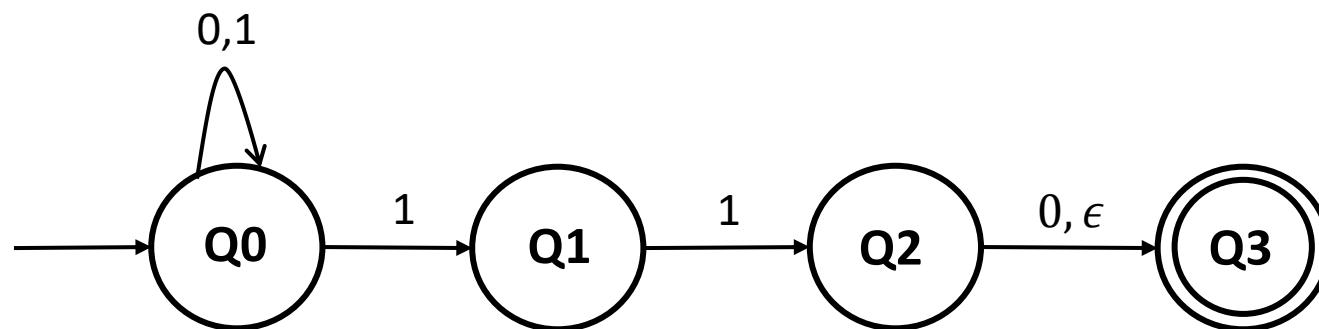
Characteristics of DFA : (i) Single start state (ii) Unique transitions (iii) Zero or more final states

**Characteristics of NFA :** (i) Single start state (ii) Zero or more final states

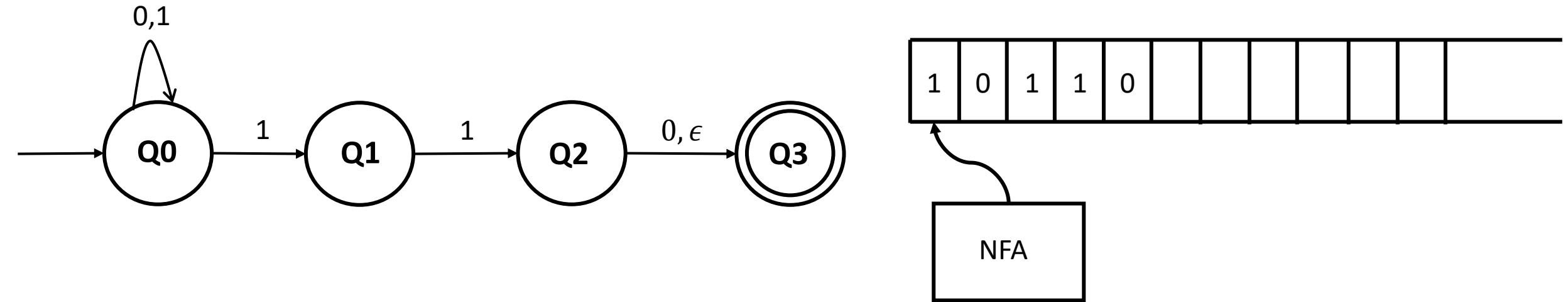
(iii) Multiple transitions are possible on the same input for a state

(iv) Some transitions might be missing

(v)  $\epsilon$  - transitions



# Non-deterministic Finite Automata (NFA)

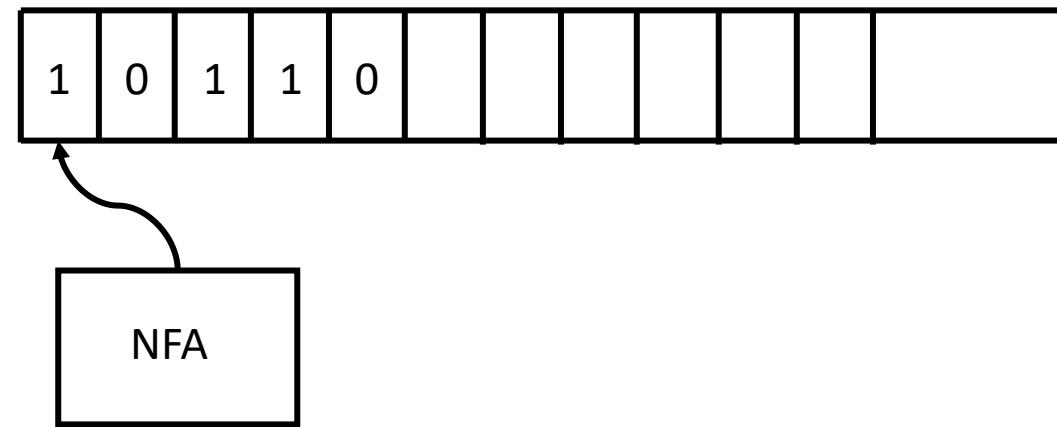
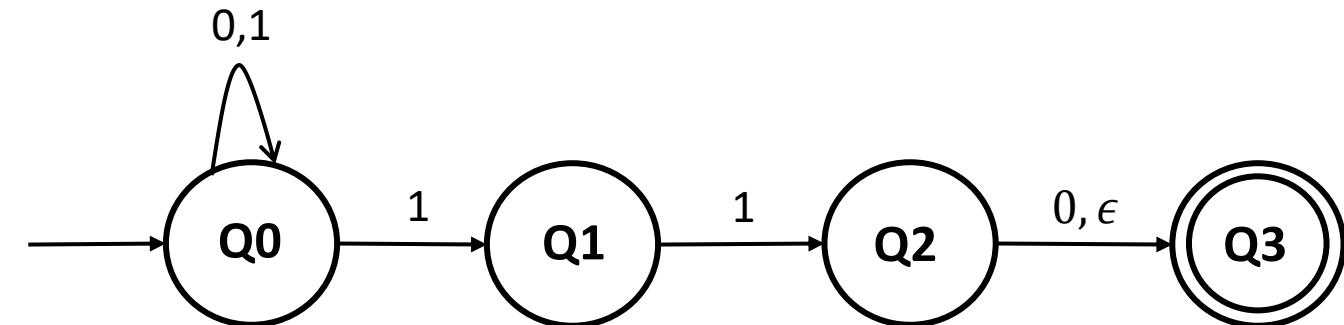


**Run 1:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0$  (**REJECT**)

**Run 2:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_1 \xrightarrow{1} Q_2 \xrightarrow{0} Q_3$  (**ACCEPT**)

Multiple **runs** per input is possible

# Non-deterministic Finite Automata (NFA)



**Run 1:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0$  (**REJECT**)

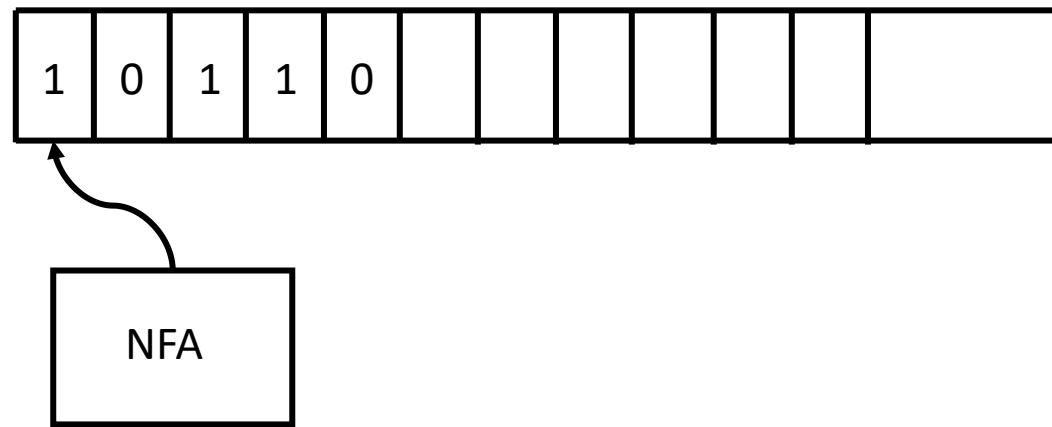
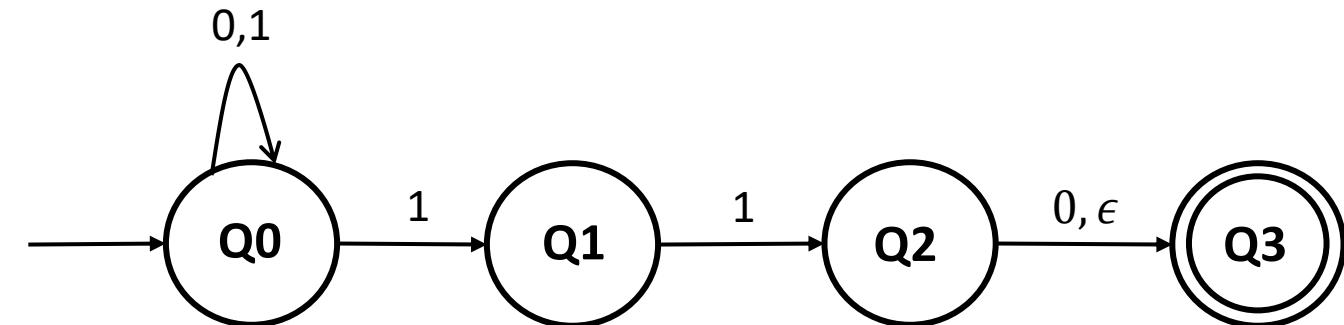
**Run 2:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_1 \xrightarrow{1} Q_2 \xrightarrow{0} Q_3$  (**ACCEPT**)

**Run 3:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_0 \xrightarrow{1} Q_1 \xrightarrow{0}$  **CRASH**

**Run 4:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_1 \xrightarrow{1} Q_2 \xrightarrow{\epsilon} Q_3 \xrightarrow{0}$  **CRASH**

**CRASH** is a Rejecting Run

# Non-deterministic Finite Automata (NFA)



**Run 1:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0$  (**REJECT**)

**Run 2:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_1 \xrightarrow{1} Q_2 \xrightarrow{0} Q_3$  (**ACCEPT**)

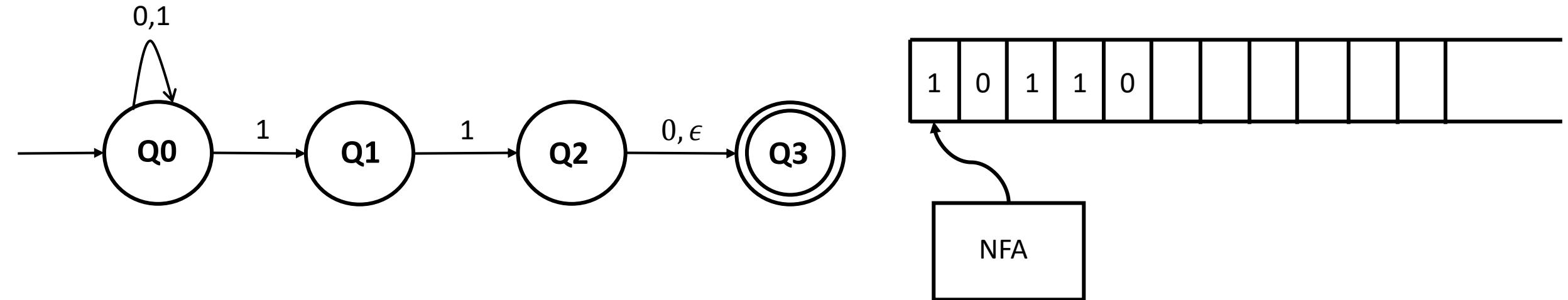
**Run 3:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_0 \xrightarrow{1} Q_1 \xrightarrow{0}$  CRASH (**REJECT**)

**Run 4:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_1 \xrightarrow{1} Q_2 \xrightarrow{\epsilon} Q_3 \xrightarrow{0}$  CRASH (**REJECT**)

The NFA “accepts” an input string, if it at **least one run** ends up in the final state. (**Accepting Run**)

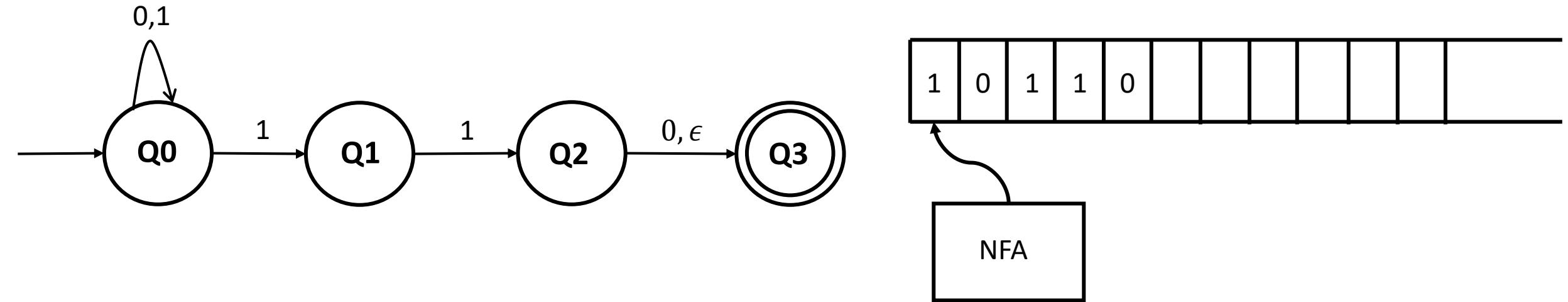
The NFA “rejects” an input string, if there are **no runs** that end up in a final state. (**Rejecting Run**)

# Non-deterministic Finite Automata (NFA)



	0	1	$\epsilon$
$Q_0$	$Q_0$	$Q_0, Q_1$	
$Q_1$		$Q_2$	
$Q_2$	$Q_3$		$Q_3$
$Q_3$			

# Non-deterministic Finite Automata (NFA)



Formally, a finite automaton  $M$  is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is a finite set called the **alphabet**.
- $\delta: Q \times \Sigma \mapsto P(Q)$  is the **transition function**.  $P(Q)$  is the power set of  $Q$
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **final/accepting states**.

	0	1	$\epsilon$
$Q_0$	$Q_0$	$Q_0, Q_1$	
$Q_1$		$Q_2$	
$Q_2$	$Q_3$		$Q_3$
$Q_3$			

# NFA vs DFA

- Are NFAs more powerful than DFAs?
- Intuitively, non-determinism seems to be adding more “power”.

# NFA vs DFA

- Are NFAs more powerful than DFAs?
- Intuitively, non-determinism seems to be adding more “power”.
- Let  $L_1$  be the language accepted by NFAs and  $L_2$  be the language accepted by DFAs
- Is  $L_2 \subseteq L_1$ ?

# NFA vs DFA

- Are NFAs more powerful than DFAs?
- Intuitively, non-determinism seems to be adding more “power”.
- Let  $L_1$  be the language accepted by NFAs and  $L_2$  be the language accepted by DFAs
- Is  $L_2 \subseteq L_1$ ? Clearly true, because a DFA is just a special case of an NFA.

# NFA vs DFA

- Are NFAs more powerful than DFAs?
- Intuitively, non-determinism seems to be adding more “power”.
- Let  $L_1$  be the language accepted by NFAs and  $L_2$  be the language accepted by DFAs
- Is  $L_2 \subseteq L_1$ ? Clearly true, because a DFA is just a special case of an NFA.
- Surprisingly, what we will show next is that  $L_1 \subseteq L_2$ !
- That is, **given an NFA, we can convert it to a DFA that accepts the same language.**
- Such a DFA is called a “**Remembering DFA**”.

Thus, DFAs and NFAs are completely equivalent and  $L_1 = L_2$ !

# Converting an NFA to a DFA

Intuitive idea for the construction of a Remembering DFA from an NFA:

- Let  $R$  be the Remembering DFA corresponding to an NFA  $N$ .
- $R$  on an input enters a state that is labelled by all possible states that  $N$  can enter on that input.
- Note that this “trims away” the non-determinism of the NFA  $N$  without “losing” the language it accepts.
- Also note that if  $N$  has  $k$  states, then  $R$  has at most  $2^k$  states. Why?

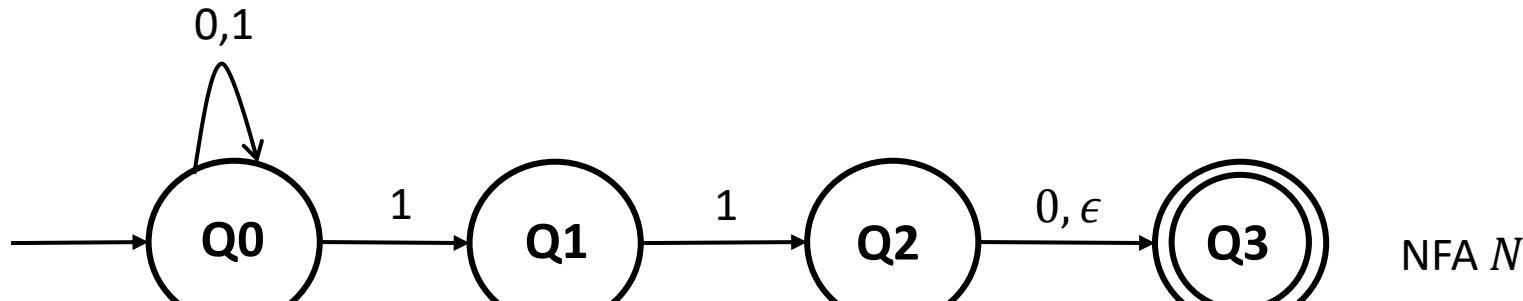
# Converting an NFA to a DFA

Intuitive idea for the construction of a Remembering DFA from an NFA:

- Let  $R$  be the Remembering DFA corresponding to an NFA  $N$ .
- $R$  on an input enters a state that is labelled by all possible states that  $N$  can enter on that input.
- Note that this “trims away” the non-determinism of the NFA  $N$  without “losing” the language it accepts.
- Also note that if  $N$  has  $k$  states, then  $R$  has at most  $2^k$  states. Why?
- Any label in the Remembering DFA is a subset of  $\{Q_0, Q_1, Q_2, \dots, Q_{k-1}\}$ , where  $Q_i$  = State of the NFA.
- There are at most  $2^k$  labels for the DFA.

# Converting an NFA to a DFA

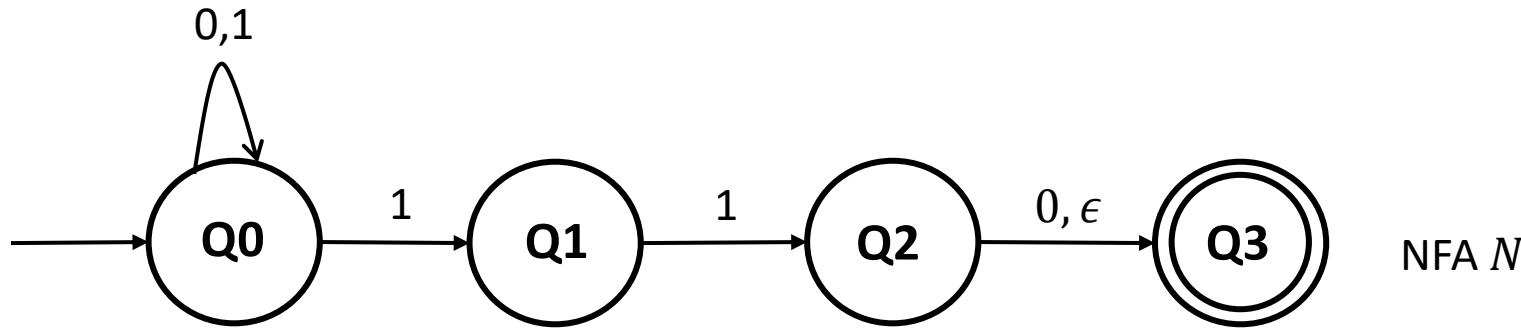
- $R$  on an input enters a state that is labelled by all possible states that  $N$  can enter on that input.



	0	1	$\epsilon$
Q0	Q0	Q0, Q1	
Q1		Q2	
Q2	Q3		Q3
Q3			

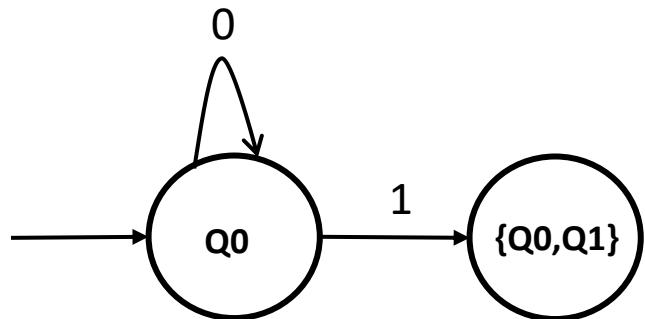
# Converting an NFA to a DFA

- $R$  on an input enters a state that is labelled by all possible states that  $N$  can enter on that input.



NFA  $N$

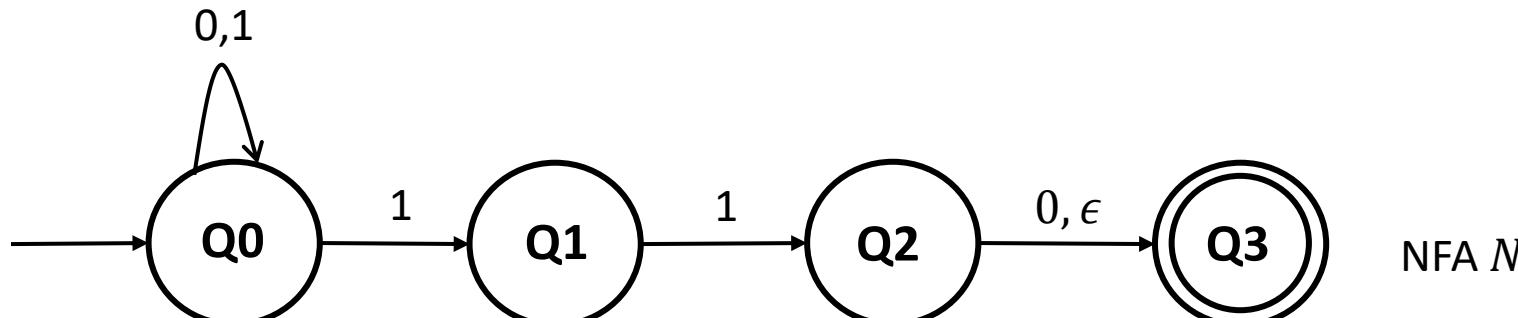
	0	1	$\epsilon$
$Q_0$	$Q_0$	$Q_0, Q_1$	
$Q_1$		$Q_2$	
$Q_2$	$Q_3$		$Q_3$
$Q_3$			



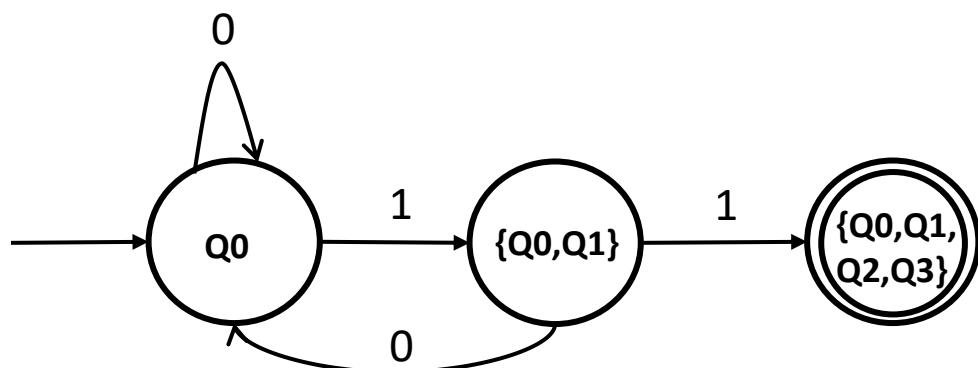
Remembering DFA  $R$

# Converting an NFA to a DFA

- $R$  on an input enters a state that is labelled by all possible states that  $N$  can enter on that input.



NFA  $N$



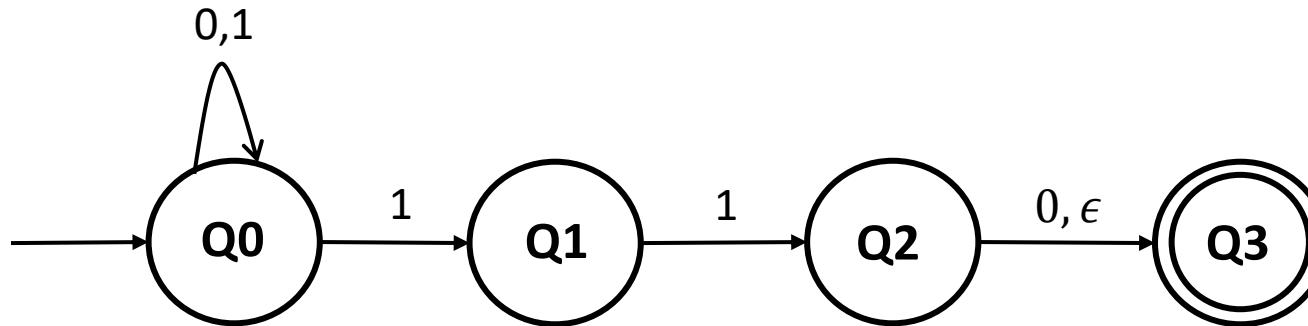
Remembering DFA  $R$

**Any state of  $R$  that contains in its label, an accepting state of  $N$  is an accepting state of  $R$ .**

	0	1	$\epsilon$
$Q_0$	$Q_0$	$Q_0, Q_1$	
$Q_1$		$Q_2$	
$Q_2$	$Q_3$		$Q_3$
$Q_3$			

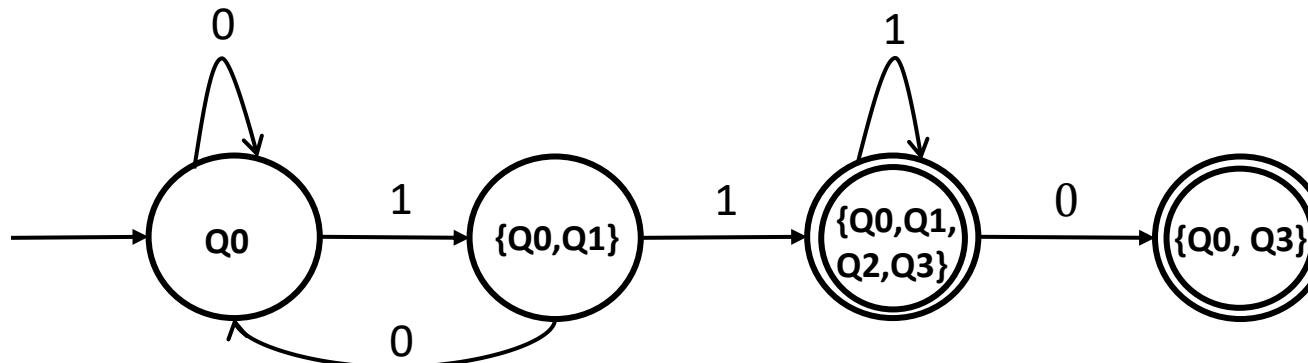
# Converting an NFA to a DFA

- $R$  on an input enters a state that is labelled by all possible states that  $N$  can enter on that input.



NFA  $N$

	0	1	$\epsilon$
$Q_0$	$Q_0$	$Q_0, Q_1$	
$Q_1$		$Q_2$	
$Q_2$	$Q_3$		$Q_3$
$Q_3$			

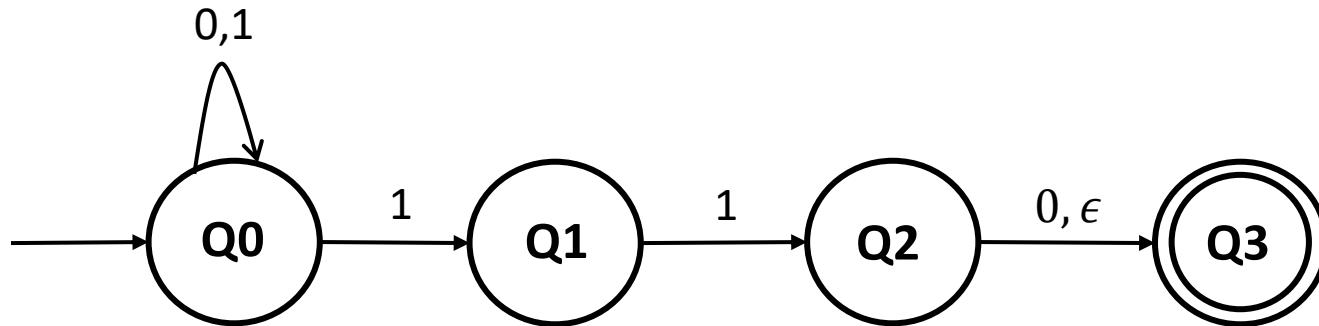


Remembering DFA  $R$

Any state of  $R$  that contains in its label, an accepting state of  $N$  is an accepting state of  $R$ .

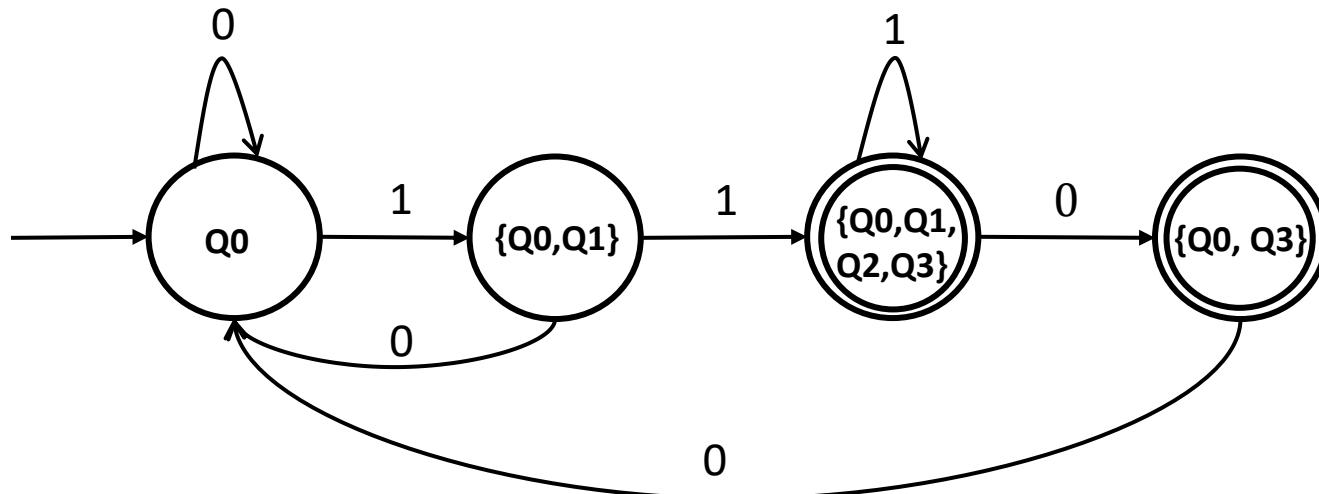
# Converting an NFA to a DFA

- $M_2$  on an input enters a state that is labelled by all possible states that  $M_1$  can enter on that input.



NFA  $N$

	0	1	$\epsilon$
$Q_0$	$Q_0$	$Q_0, Q_1$	
$Q_1$		$Q_2$	
$Q_2$	$Q_3$		$Q_3$
$Q_3$			

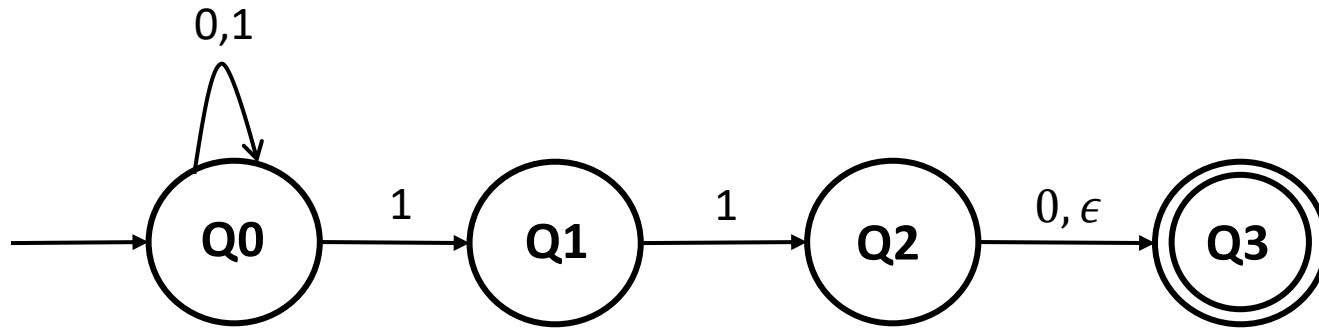


Remembering DFA  $R$

Any state of  $R$  that contains in its label, an accepting state of  $N$  is an accepting state of  $R$ .

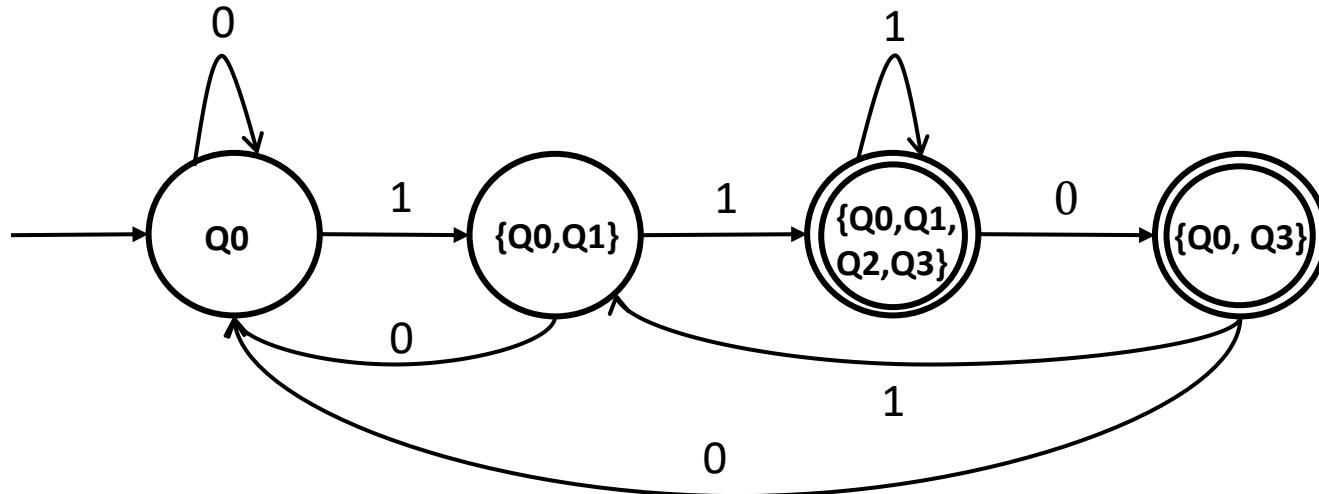
# Converting an NFA to a DFA

- $M_2$  on an input enters a state that is labelled by all possible states that  $M_1$  can enter on that input.



NFA  $N$

	0	1	$\epsilon$
$Q_0$	$Q_0$	$Q_0, Q_1$	
$Q_1$		$Q_2$	
$Q_2$	$Q_3$		$Q_3$
$Q_3$			

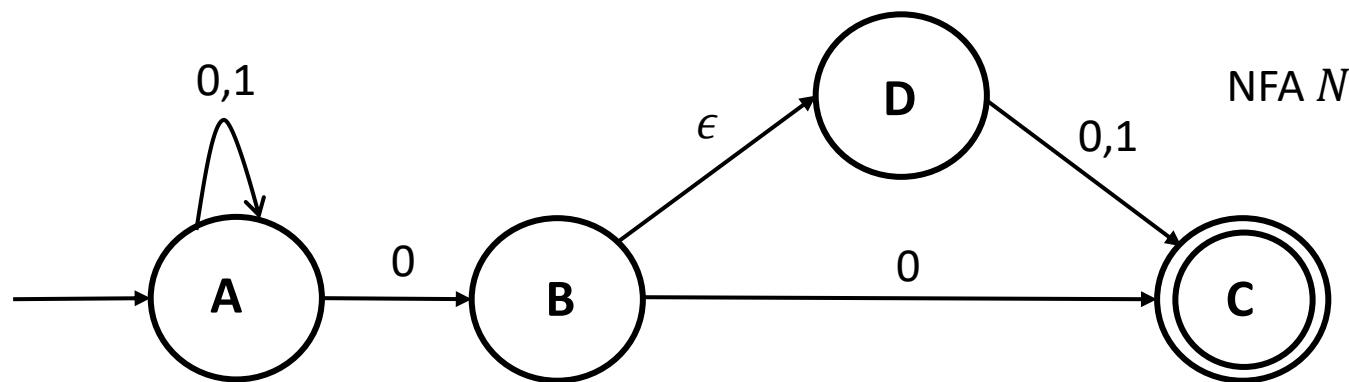


Remembering DFA  $R$

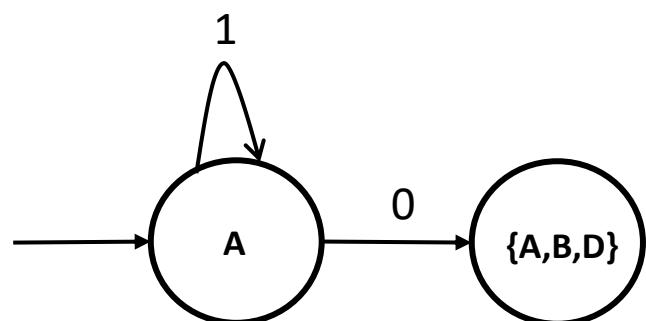
Any state of  $R$  that contains in its label, an accepting state of  $N$  is an accepting state of  $R$ .

# Converting an NFA to a DFA

- $M_2$  on an input enters a state that is labelled by all possible states that  $M_1$  can enter on that input.



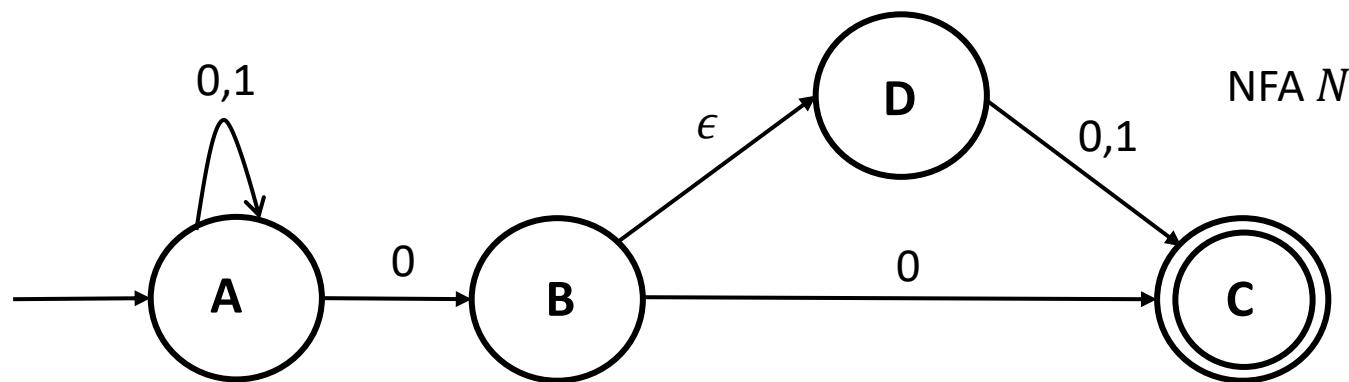
	0	1	$\epsilon$
A	A, B	A	
B	C		D
C			
D	C	C	



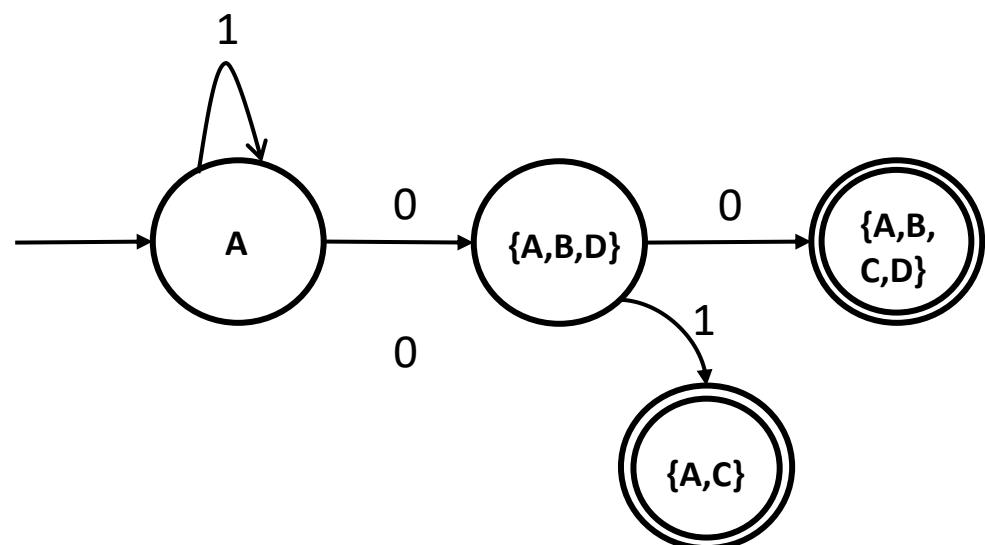
Remembering DFA  $R$

# Converting an NFA to a DFA

- $M_2$  on an input enters a state that is labelled by all possible states that  $M_1$  can enter on that input.



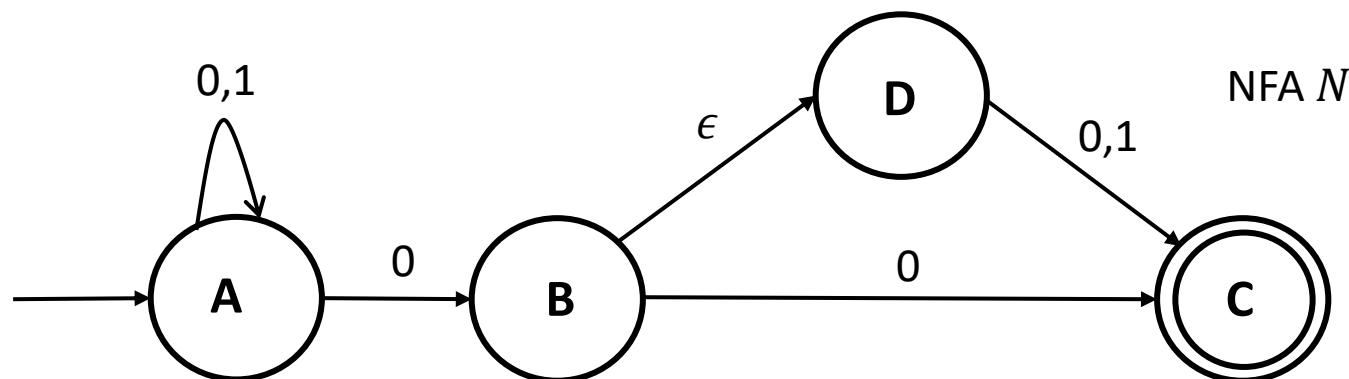
	0	1	$\epsilon$
A	A, B	A	
B	C		D
C			
D	C	C	



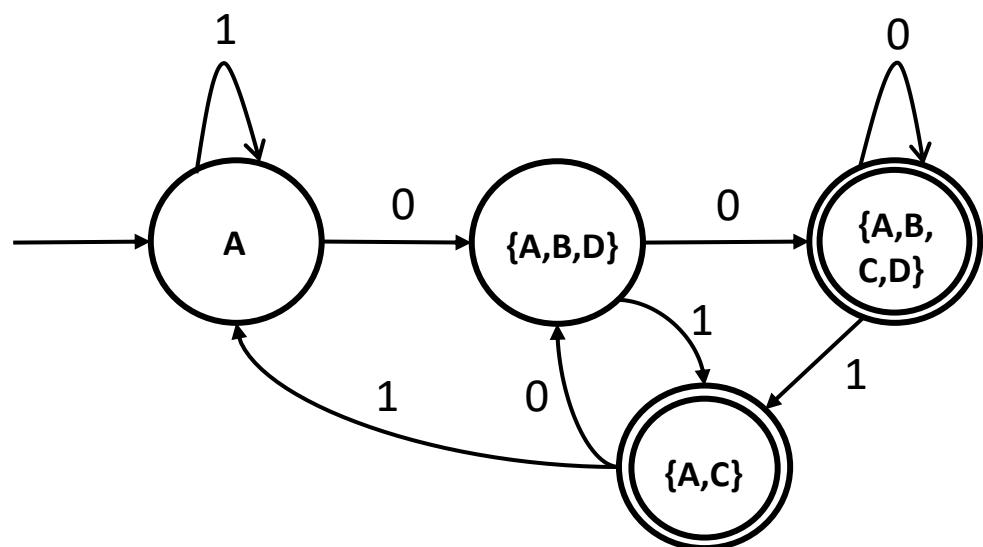
Remembering DFA  $R$

# Converting an NFA to a DFA

- $M_2$  on an input enters a state that is labelled by all possible states that  $M_1$  can enter on that input.



	0	1	$\epsilon$
A	A, B	A	
B	C		D
C			
D	C	C	



Remembering DFA  $R$

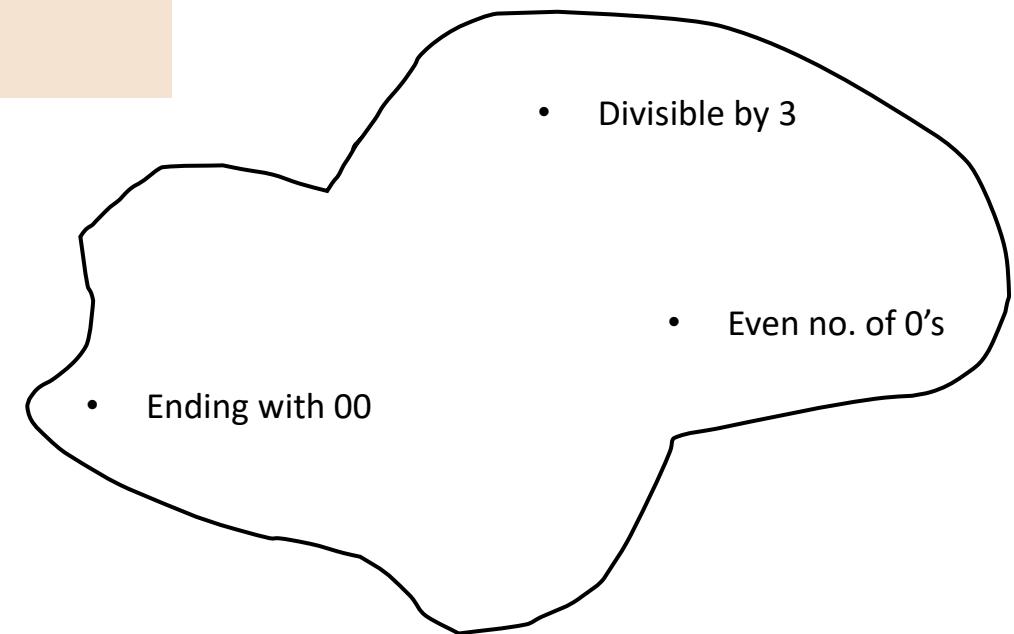
# Regular Languages

A language is called a **Regular Language** if there exists some finite automata recognizing it.

If  $M$  be a finite automaton (DFA/NFA) and,

$$L(M) = \{\omega \mid \omega \text{ is accepted by } M\}$$

**$L(M)$  is regular.**



Set of all regular Languages

# Regular Languages

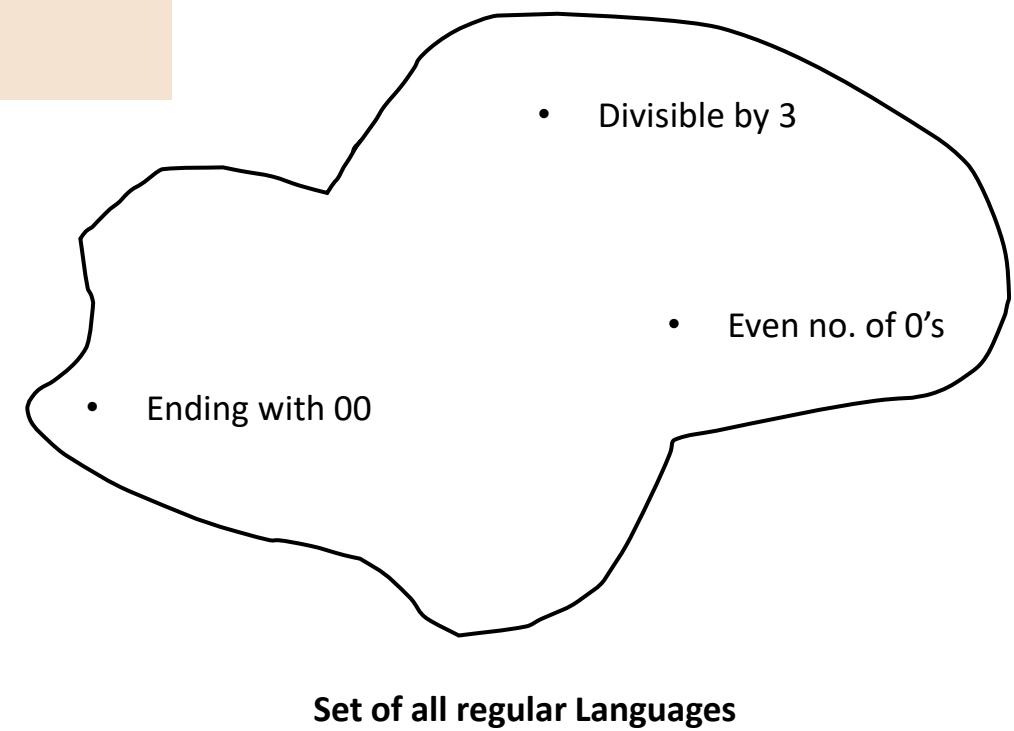
A language is called a **Regular Language** if there exists some finite automata recognizing it.

If  $M$  be a finite automaton (DFA/NFA) and,

$$L(M) = \{\omega \mid \omega \text{ is accepted by } M\}$$

**$L(M)$  is regular.**

- Any language has associated with it, a set of operations that can be performed on it.
- These operations help us to understand the properties of that language, e.g. closure properties
- For regular languages, this will help us prove that certain languages are non-regular and hence we cannot hope to design a finite automaton for them

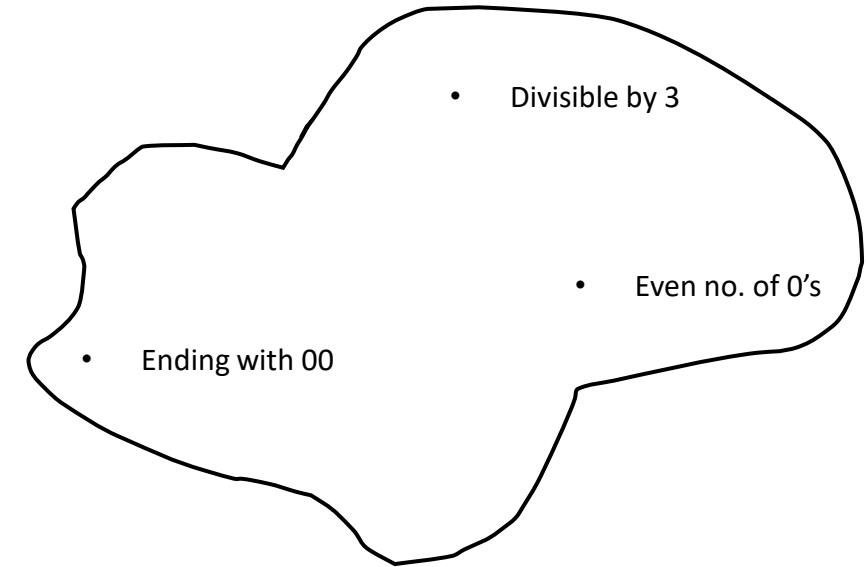


# Regular Languages

## Regular Operations:

Let  $L_1$  and  $L_2$  be languages. The following are the *regular operations*:

- **Union:**  $L_1 \cup L_2 = \{x|x \in L_1 \text{ or } x \in L_2\}$
- **Concatenation:**  $L_1 \cdot L_2 = \{xy|x \in L_1 \text{ and } y \in L_2\}$
- **Star:**  $L_1^* = \{x_1x_2 \cdots x_k | k \geq 0 \text{ and each } x_i \in L_1\}$



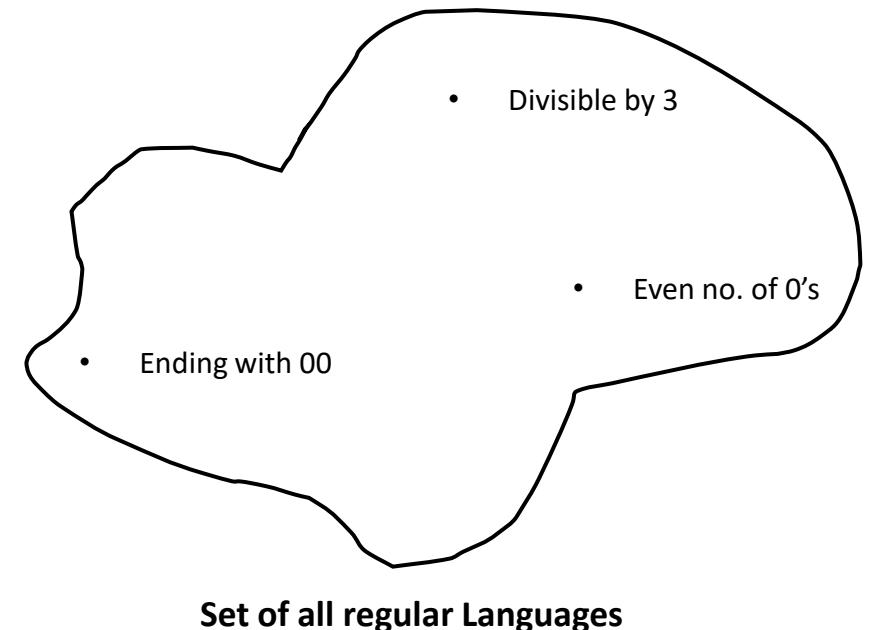
Set of all regular Languages

# Regular Languages

## Regular Operations:

Let  $L_1$  and  $L_2$  be languages. The following are the *regular operations*:

- **Union:**  $L_1 \cup L_2 = \{x|x \in L_1 \text{ or } x \in L_2\}$
- **Concatenation:**  $L_1 \cdot L_2 = \{xy|x \in L_1 \text{ and } y \in L_2\}$
- **Star:**  $L_1^* = \{x_1x_2 \cdots x_k | k \geq 0 \text{ and each } x_i \in L_1\}$



**Star operation:** It is an unary operation (unlike the other two) and involves putting together *any number of strings in  $L_1$  together to obtain a new string*.

**Note:** Any number of strings includes “0” as a possibility and so the empty string  $\epsilon$  is a member of  $L_1^*$ .

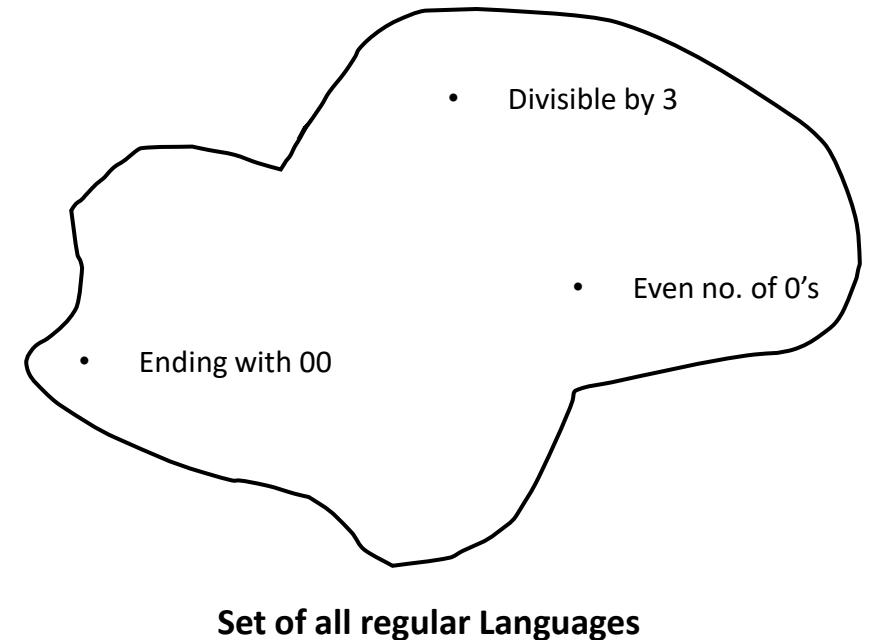
If  $\Sigma = \{a\}$ ,  $\Sigma^* = \{\epsilon, a, aa, aaa, \dots \dots\}$ ; If  $\Sigma = \{\Phi\}$ ,  $\Sigma^* = \{\epsilon\}$

# Regular Languages

## Regular Operations:

Let  $L_1$  and  $L_2$  be languages. The following are the *regular operations*:

- **Union:**  $L_1 \cup L_2 = \{x|x \in L_1 \text{ or } x \in L_2\}$
- **Concatenation:**  $L_1 \cdot L_2 = \{xy|x \in L_1 \text{ and } y \in L_2\}$
- **Star:**  $L_1^* = \{x_1x_2 \cdots x_k | k \geq 0 \text{ and each } x_i \in L_1\}$



**Star operation:** It is an unary operation (unlike the other two) and involves putting together *any number of strings in  $L_1$  together to obtain a new string*.

**Note:** Any number of strings includes “0” as a possibility and so the empty string  $\epsilon$  is a member of  $L_1^*$ .

If  $\Sigma = \{0,1\}$ , we have that  $\Sigma^* = \{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots \dots \}$

# Regular Languages

**Regular Operations:** Let  $L_1$  and  $L_2$  be languages.

- **Union:**  $L_1 \cup L_2 = \{x|x \in L_1 \text{ or } x \in L_2\}$
- **Concatenation:**  $L_1 \cdot L_2 = \{xy|x \in L_1 \text{ and } y \in L_2\}$
- **Star:**  $L_1^* = \{x_1 x_2 \cdots x_k | k \geq 0 \text{ and each } x_i \in L_1\}$

**Example:** Let the alphabet  $\Sigma = \{a, b, \dots, z\}$ . If  $L_1 = \{\textit{social}, \textit{economic}\}$  and  $L_2 = \{\textit{justice}, \textit{reform}\}$ , then

- $L_1 \cup L_2 = \{\textit{social}, \textit{economic}, \textit{justice}, \textit{reform}\}$

# Regular Languages

**Regular Operations:** Let  $L_1$  and  $L_2$  be languages.

- **Union:**  $L_1 \cup L_2 = \{x|x \in L_1 \text{ or } x \in L_2\}$
- **Concatenation:**  $L_1 \cdot L_2 = \{xy|x \in L_1 \text{ and } y \in L_2\}$
- **Star:**  $L_1^* = \{x_1x_2 \cdots x_k | k \geq 0 \text{ and each } x_i \in L_1\}$

**Example:** Let the alphabet  $\Sigma = \{a, b, \dots, z\}$ . If  $L_1 = \{\textit{social}, \textit{economic}\}$  and  $L_2 = \{\textit{justice}, \textit{reform}\}$ , then

- $L_1 \cup L_2 = \{\textit{social}, \textit{economic}, \textit{justice}, \textit{reform}\}$
- $L_1 \cdot L_2 = \{\textit{socialjustice}, \textit{socialreform}, \textit{economicjustice}, \textit{economicreform}\}$

# Regular Languages

**Regular Operations:** Let  $L_1$  and  $L_2$  be languages.

- **Union:**  $L_1 \cup L_2 = \{x | x \in L_1 \text{ or } x \in L_2\}$
- **Concatenation:**  $L_1 \cdot L_2 = \{xy | x \in L_1 \text{ and } y \in L_2\}$
- **Star:**  $L_1^* = \{x_1 x_2 \cdots x_k | k \geq 0 \text{ and each } x_i \in L_1\}$

**Example:** Let the alphabet  $\Sigma = \{a, b, \dots, z\}$ . If  $L_1 = \{\text{social, economic}\}$  and  $L_2 = \{\text{justice, reform}\}$ , then

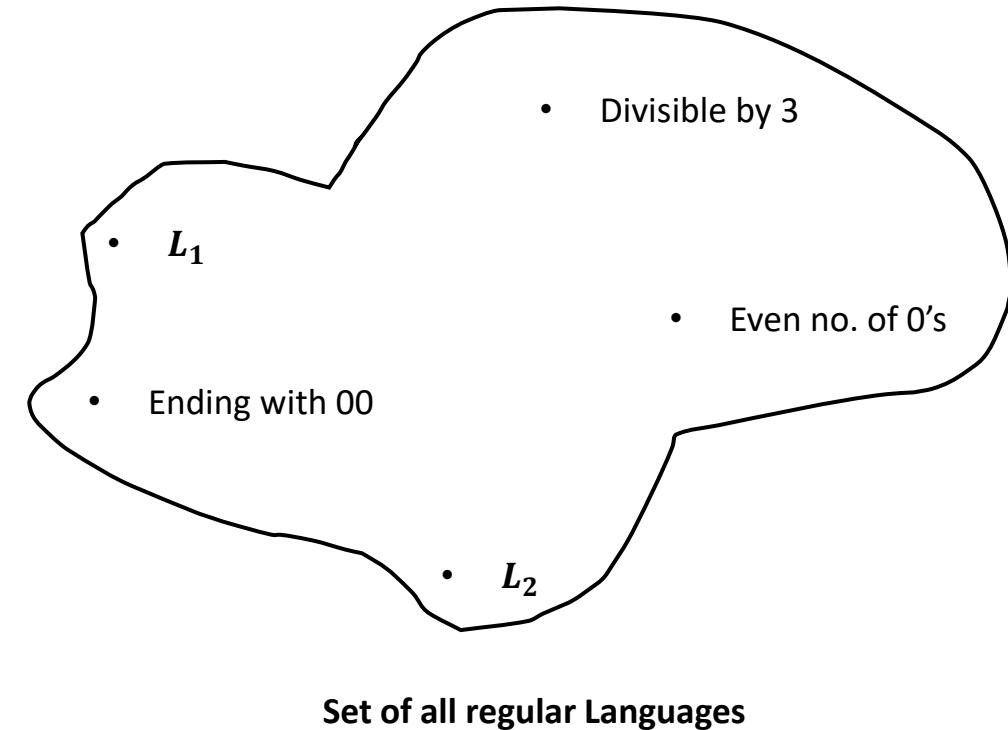
- $L_1 \cup L_2 = \{\text{social, economic, justice, reform}\}$
- $L_1 \cdot L_2 = \{\text{socialjustice, socialreform, economicjustice, economicreform}\}$
- $L_1^* = \{\epsilon, \text{social, economic, socialsocial, socialeconomic, economicsocial, economiceconomic, socialsocialsocial, socialsocialeconomic, socialeconomiceconomic, .....}\}$
- $L_2^* = \{\epsilon, \text{justice, reform, justicejustice, justicereform, reformjustice, reformreform, justicejusticejustice, .....}\}$

# Closure of Regular Languages

We want to check whether the set of regular languages are **closed** under some operations.

What does this mean?

- We pick up points within the set of all regular languages (say  $L_1$  and  $L_2$ )
- Perform *set operations* such as Union, concatenation, Star, intersection, reversal, complement etc on them.
- Observe whether the resulting language still belongs to the set of all regular languages.
- If so, we say, regular languages are **closed** under that operation.

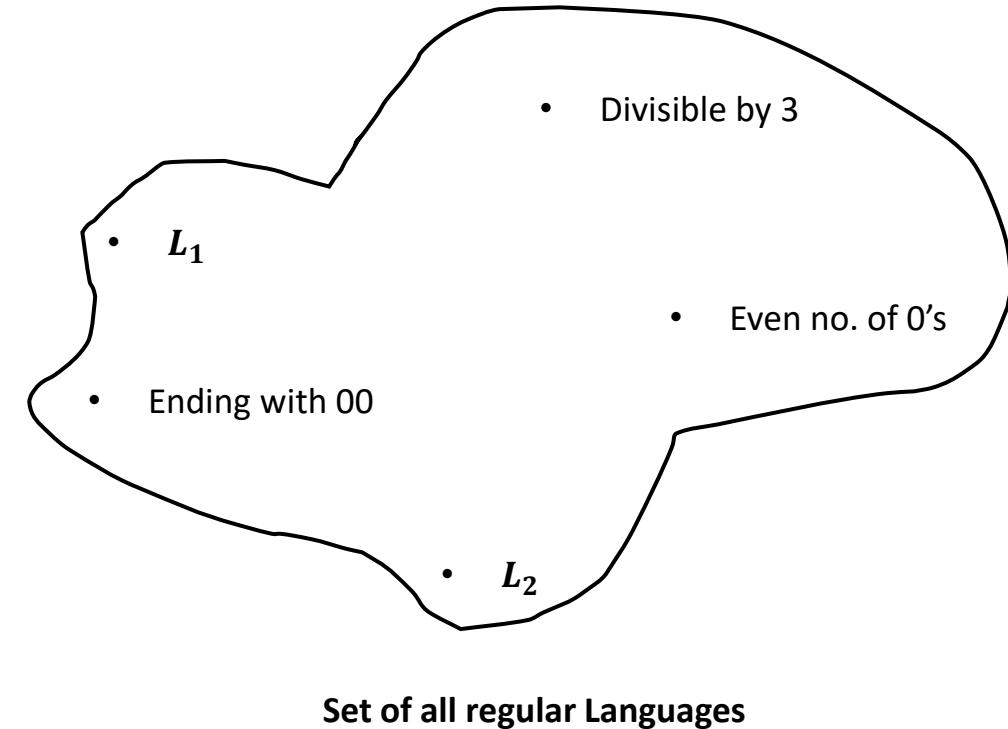


# Closure of Regular Languages

We want to check whether the set of regular languages are **closed** under some operations.

What does this mean?

- We pick up points within the set of all regular languages (say  $L_1$  and  $L_2$ )
- Perform *set operations* such as Union, concatenation, Star, intersection, reversal, complement etc on them.
- Observe whether the resulting language still belongs to the set of all regular languages.
- If so, we say, regular languages are **closed** under that operation.

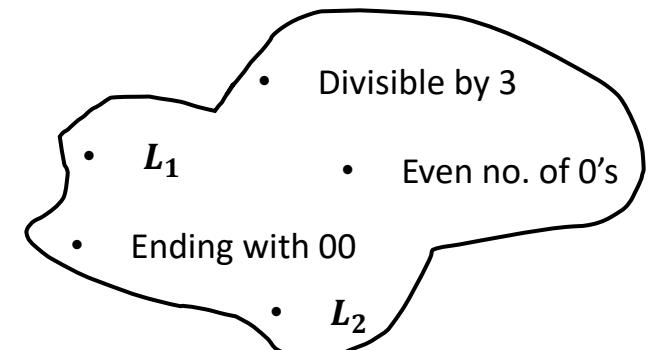


For example, the **natural numbers** are closed under addition/multiplication and not under subtraction/division.

# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under union?**

Suppose  $L_1$  and  $L_2$  are regular languages. Is  $L = L_1 \cup L_2$  also regular?



Set of all regular Languages

# Closure of Regular Languages

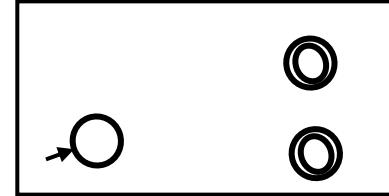
**Q:** Is the set of all regular languages **closed under union?**

Suppose  $L_1$  and  $L_2$  are regular languages. Is  $L = L_1 \cup L_2$  also regular?

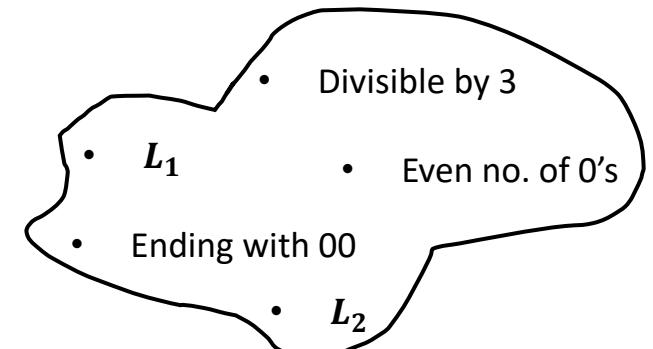
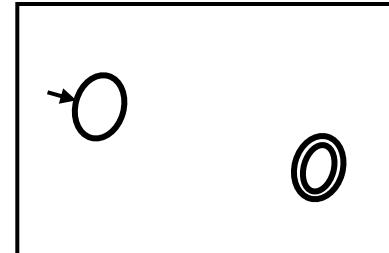
**Proof:** Since  $L_1$  and  $L_2$  are regular, there must be a DFA  $M_1$  that accepts  $L_1$ , i.e.  $L(M_1) = L_1$  and a DFA  $M_2$  that accepts  $L_2$ , i.e.  $L(M_2) = L_2$ .

Using  $M_1$  and  $M_2$ , we will show how to construct an NFA  $M$  that accepts  $L = L_1 \cup L_2$ , i.e.  $L(M) = L_1 \cup L_2$ .

Suppose the DFA for  $M_1$  is



And the DFA for  $M_2$  is



Set of all regular Languages

# Closure of Regular Languages

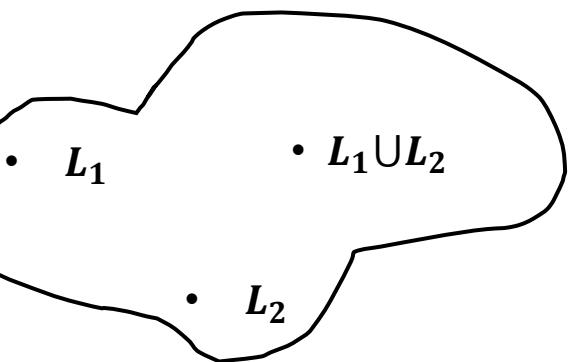
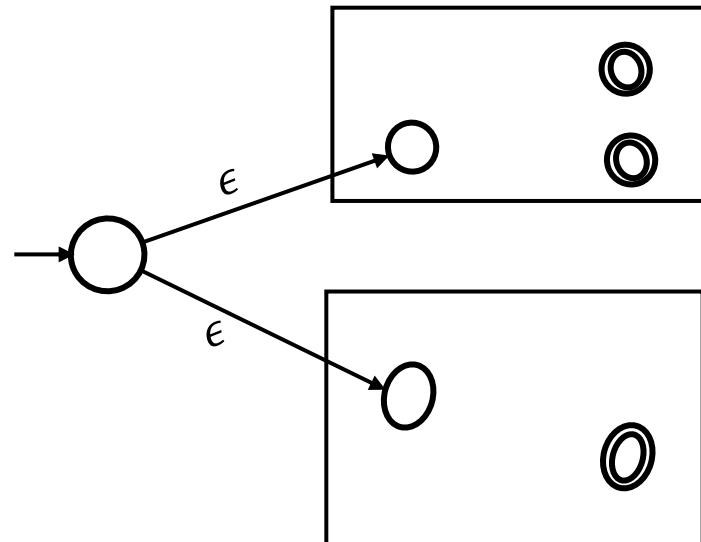
**Q:** Is the set of all regular languages **closed under union?**

Suppose  $L_1$  and  $L_2$  are regular languages. Is  $L = L_1 \cup L_2$  also regular?

**Proof:** Since  $L_1$  and  $L_2$  are regular, there must be a DFA  $M_1$  that accepts  $L_1$ , i.e.  $L(M_1) = L_1$  and a DFA  $M_2$  that accepts  $L_2$ , i.e.  $L(M_2) = L_2$ .

Using  $M_1$  and  $M_2$ , we will show how to construct an NFA  $M$  that accepts  $L = L_1 \cup L_2$ , i.e.  $L(M) = L_1 \cup L_2$ .

NFA  $M$  accepting  $L = L_1 \cup L_2$



Set of all regular Languages

# Closure of Regular Languages

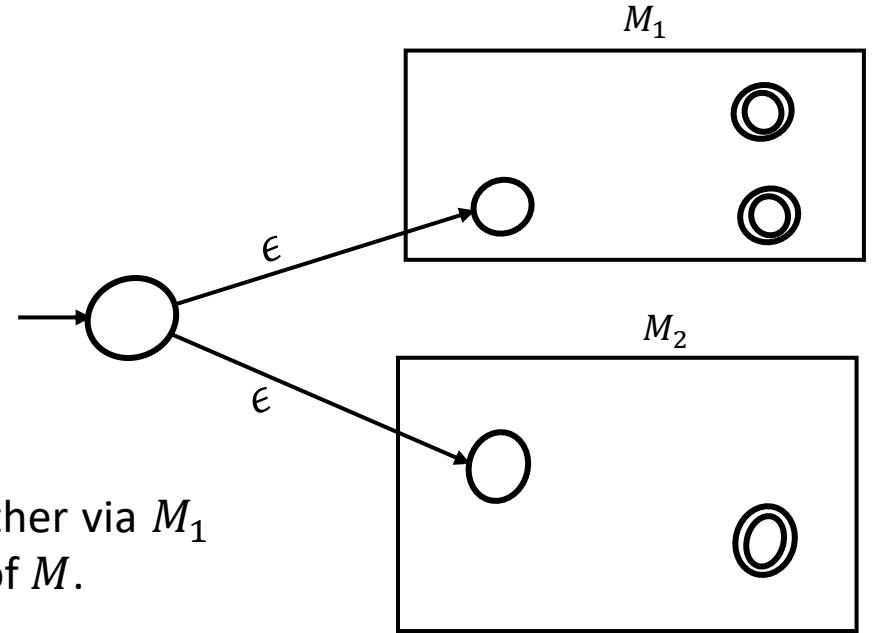
**Q:** Is the set of all regular languages **closed under union?**

Suppose  $L_1$  and  $L_2$  are regular languages. Is  $L = L_1 \cup L_2$  also regular?

**Proof:** In order to prove that  $L(M) = L_1 \cup L_2$ , we show two things:

(i)  $L \subseteq L_1 \cup L_2$

Let  $\omega \in L$ , i.e.  $\omega$  is accepted by  $M$ . The final state for  $L$  can be reached either via  $M_1$  or  $M_2$ . Thus  $\omega$  must be accepted by either of them to reach the final state of  $M$ .



# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under union?**

Suppose  $L_1$  and  $L_2$  are regular languages. Is  $L = L_1 \cup L_2$  also regular?

**Proof:** In order to prove that  $L(M) = L_1 \cup L_2$ , we show two things:

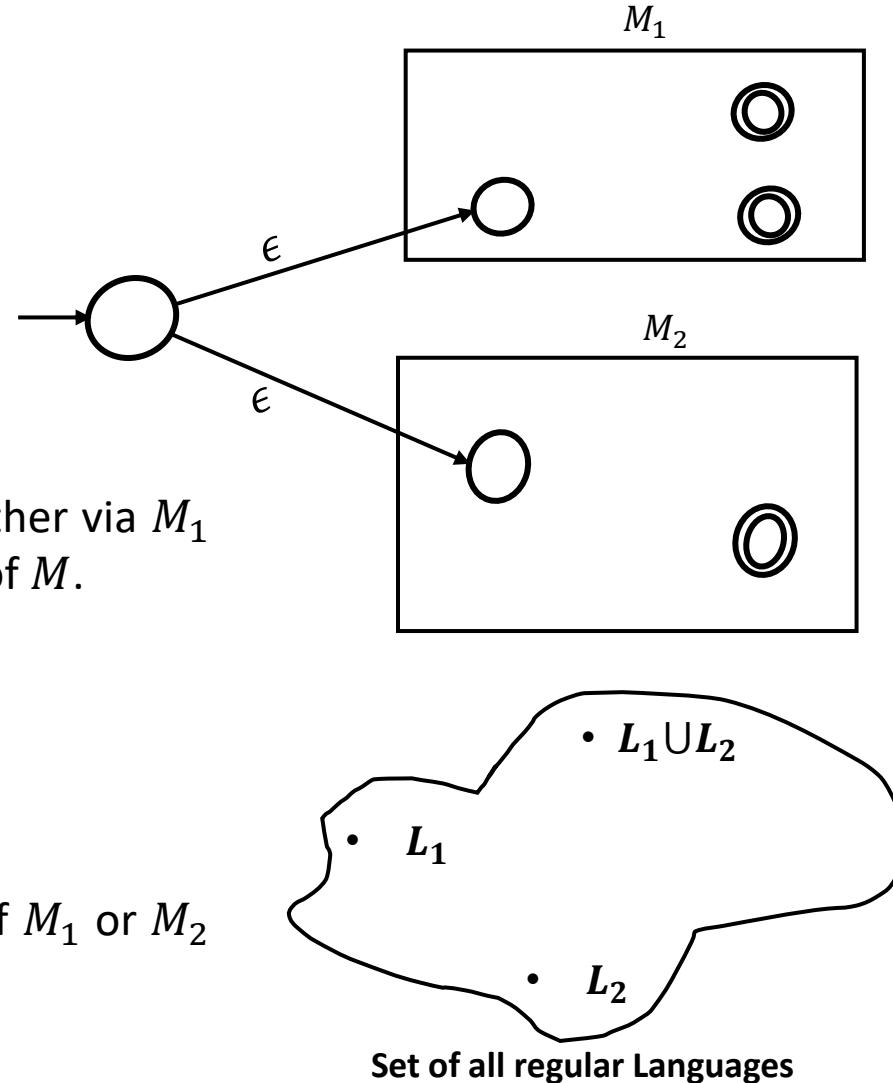
(i)  $L \subseteq L_1 \cup L_2$

Let  $\omega \in L$ , i.e.  $\omega$  is accepted by  $M$ . The final state for  $L$  can be reached either via  $M_1$  or  $M_2$ . Thus  $\omega$  must be accepted by either of them to reach the final state of  $M$ .

(ii)  $L_1 \cup L_2 \subseteq L$

Let  $\omega \in L_1 \cup L_2$ . Then,  $\omega \in L_1$  or  $\omega \in L_2$ .

Thus,  $\omega$  must reach the final state of  $M_1$  or  $M_2$ . But since the start state of  $M_1$  or  $M_2$  can be reached from the start state of  $M$  by taking an  $\epsilon$ -transition,  $\omega \in L$ .

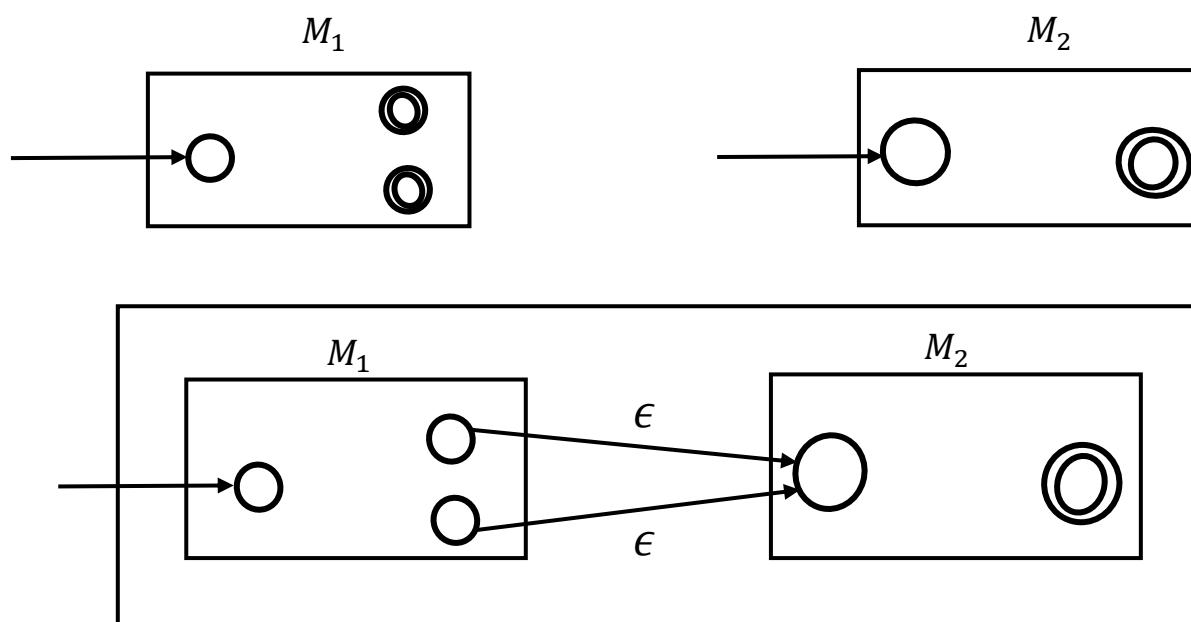


# Closure of Regular Languages

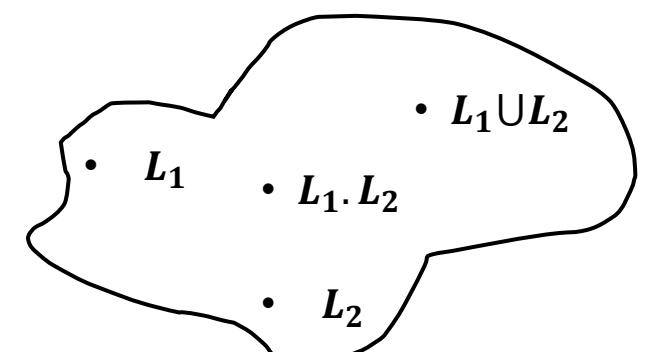
**Q:** Is the set of all regular languages **closed under concatenation**? Suppose  $L_1$  and  $L_2$  are regular languages. Is  $L = L_1 \cdot L_2$  also regular?

**Proof:** Since  $L_1$  and  $L_2$  are regular, there must be a DFA  $M_1$  that accepts  $L_1$ , i.e.  $L(M_1) = L_1$  and a DFA  $M_2$  that accepts  $L_2$ , i.e.  $L(M_2) = L_2$ .

Using  $M_1$  and  $M_2$ , we will show how to construct an NFA  $M$  that accepts  $L = L_1 \cdot L_2$ .



NFA  $M$  accepting  $L = L_1 \cdot L_2$



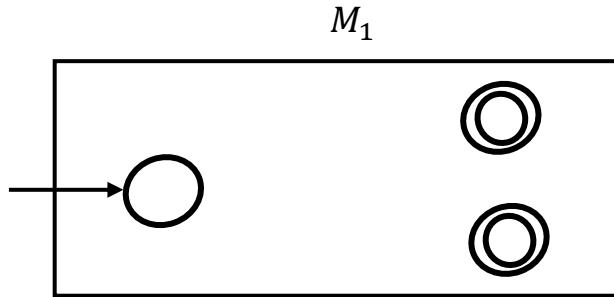
Set of all regular Languages

$$L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$$

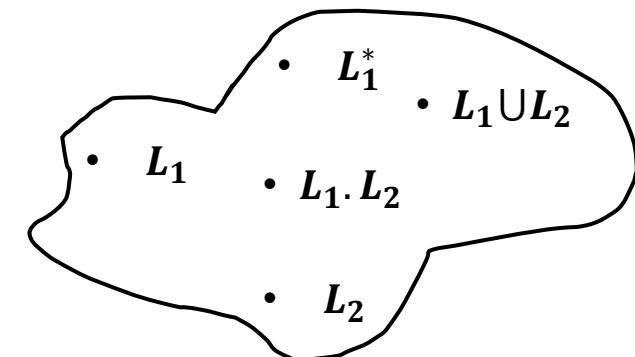
# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under star**? Suppose  $L_1$  is a regular language. Is  $L_1^*$  also regular?

**Proof:** Since  $L_1$  is regular, there must be a DFA  $M_1$  that accepts  $L_1$ , i.e.  $L(M_1) = L_1$ . Using  $M_1$ , we will show how to construct an NFA  $M$  that accepts  $L = L_1^*$ .



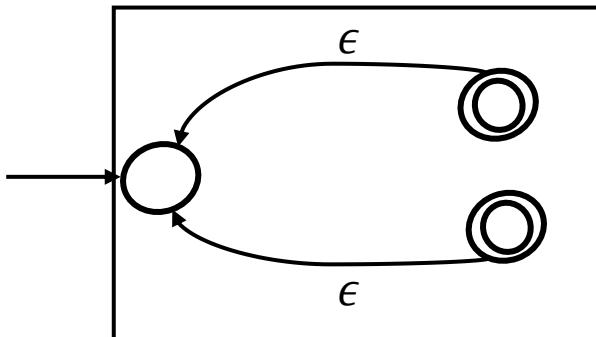
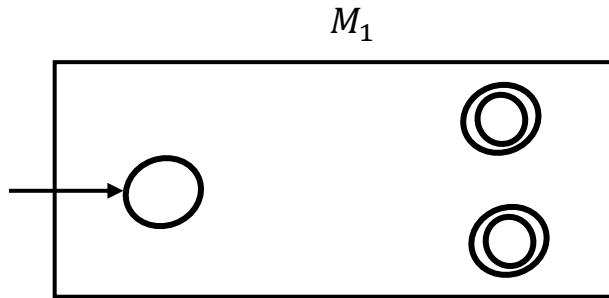
$$L_1^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$$



# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under star**? Suppose  $L_1$  is a regular language. Is  $L_1^*$  also regular?

**Proof:** Since  $L_1$  is regular, there must be a DFA  $M_1$  that accepts  $L_1$ , i.e.  $L(M_1) = L_1$ . Using  $M_1$ , we will show how to construct an NFA  $M$  that accepts  $L = L_1^*$ .

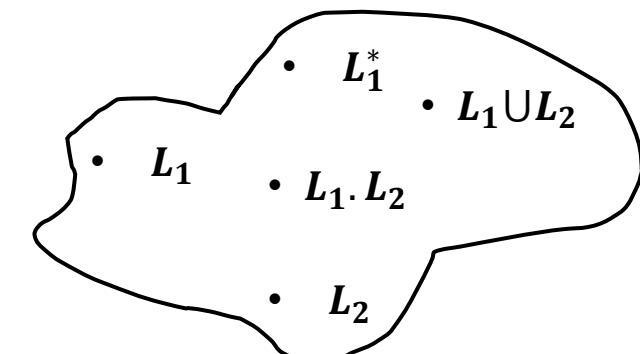


NFA accepting  $L = L_1^*$

$$L_1^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$$

**Steps:**

- Make  $\epsilon$ -transitions from the final states of  $L_1$  to the initial state of  $L_1$ .

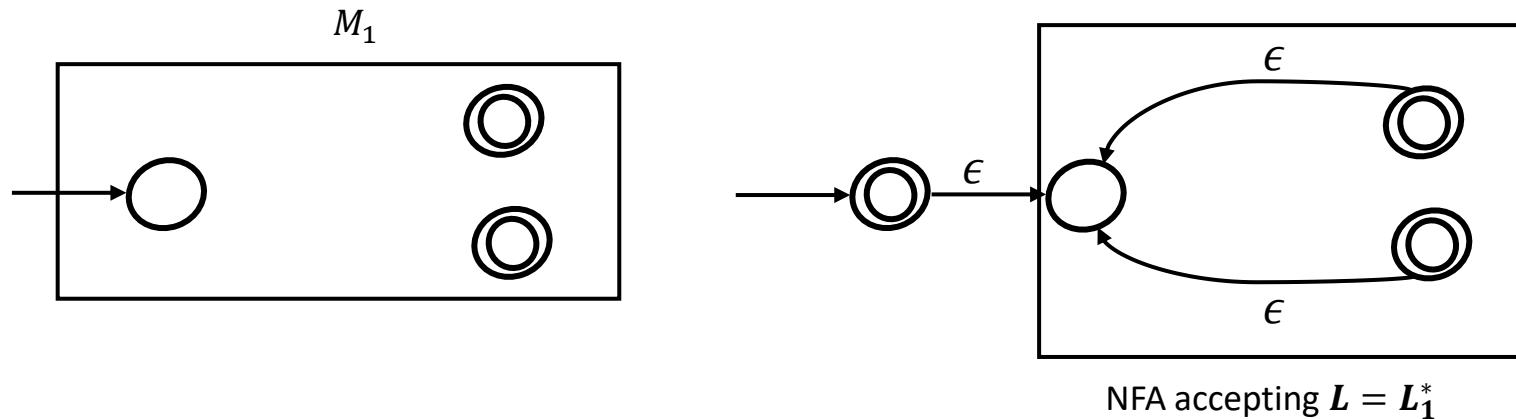


Set of all regular Languages

# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under star**? Suppose  $L_1$  is a regular language. Is  $L_1^*$  also regular?

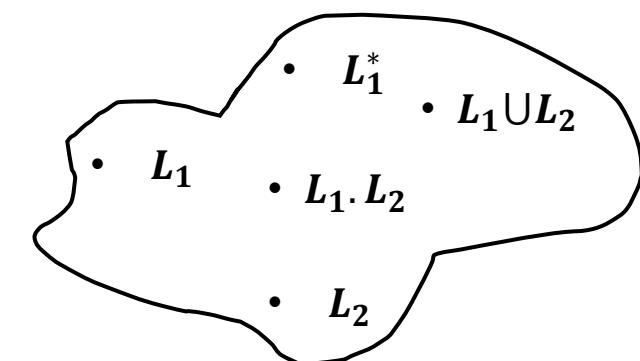
**Proof:** Since  $L_1$  is regular, there must be a DFA  $M_1$  that accepts  $L_1$ , i.e.  $L(M_1) = L_1$ . Using  $M_1$ , we will show how to construct an NFA  $M$  that accepts  $L = L_1^*$ .



$$L_1^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$$

**Steps:**

- Make  $\epsilon$ -transitions from the final states of  $L_1$  to the initial state of  $L_1$ .
- Make a new final state as the start state and make an  $\epsilon$ -transition from this state to the previous start state of  $L_1$ .



Set of all regular Languages

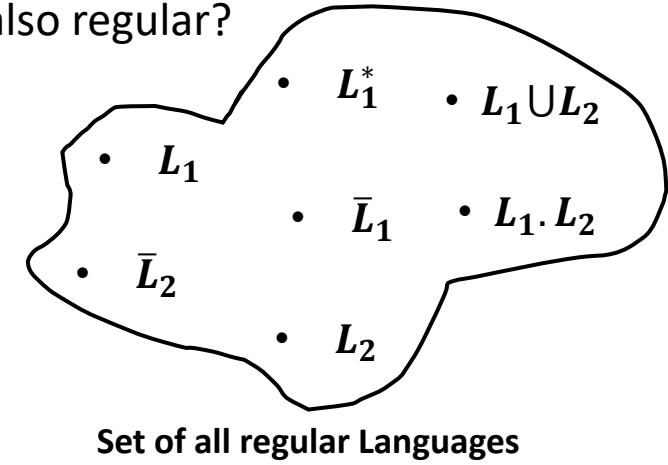
# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under complement**? If  $L$  is regular, then is  $\bar{L}$  also regular?

**Proof:** Given a DFA  $M$ , such that  $L(M) = L$ , construct the **toggled DFA  $M'$**  from  $M$ , by

- (i) changing all the non-final states of  $M$  to be the final states of  $M'$  and
- (ii) changing all the final states  $M$  to be the non-final states of  $M'$ .

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



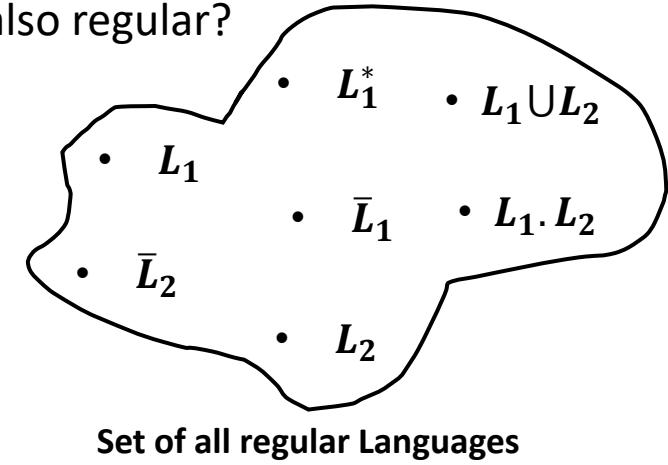
# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under complement**? If  $L$  is regular, then is  $\bar{L}$  also regular?

**Proof:** Given a DFA  $M$ , such that  $L(M) = L$ , construct the **toggled DFA  $M'$**  from  $M$ , by

- (i) changing all the non-final states of  $M$  to be the final states of  $M'$  and
- (ii) changing all the final states  $M$  to be the non-final states of  $M'$ .

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



**Q:** If  $L$  is the language accepted by an NFA, does “toggling” its states result in an NFA that accepts  $\bar{L}$ ?

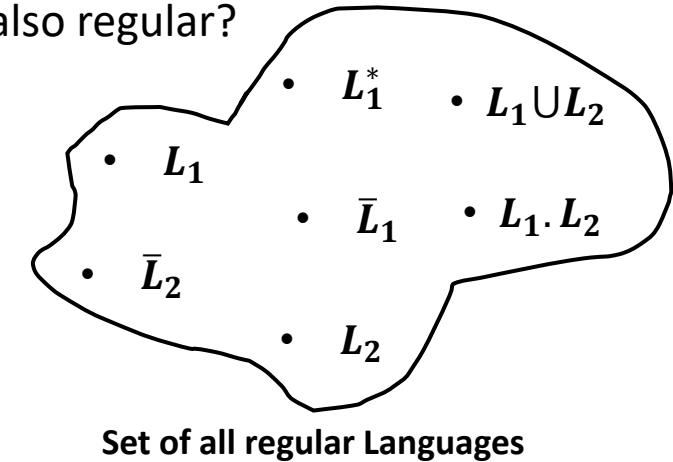
# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under complement**? If  $L$  is regular, then is  $\bar{L}$  also regular?

**Proof:** Given a DFA  $M$ , such that  $L(M) = L$ , construct the **toggled DFA  $M'$**  from  $M$ , by

- (i) changing all the non-final states of  $M$  to be the final states of  $M'$  and
- (ii) changing all the final states  $M$  to be the non-final states of  $M'$ .

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



**Q:** If  $L$  is the language accepted by an NFA, does “toggling” its states result in an NFA that accepts  $\bar{L}$ ?

**Proof:** Consider that for an input string  $x \in L$ , such that  $N$  accepts it. Suppose there is an rejecting run and an accepting run for input  $x$ . (See Table)

	NFA $N$	Toggled NFA $N'$
Run 1	Rejecting	
Run 2	Accepting	

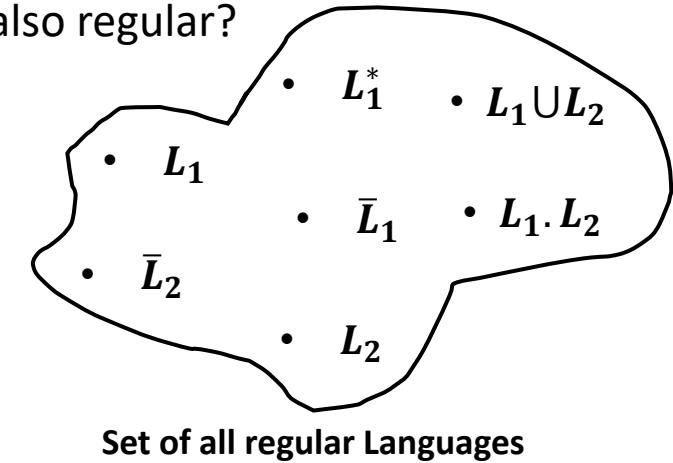
# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under complement**? If  $L$  is regular, then is  $\bar{L}$  also regular?

**Proof:** Given a DFA  $M$ , such that  $L(M) = L$ , construct the **toggled DFA  $M'$**  from  $M$ , by

- (i) changing all the non-final states of  $M$  to be the final states of  $M'$  and
- (ii) changing all the final states  $M$  to be the non-final states of  $M'$ .

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



**Q:** If  $L$  is the language accepted by an NFA, does “toggling” its states result in an NFA that accepts  $\bar{L}$ ?

**Proof:** Consider that for an input string  $x \in L$ , such that  $N$  accepts it. Suppose there is an rejecting run and an accepting run for input  $x$ . (See Table)

For toggled NFA  $N'$  too, there are two runs for  $x$ . However, the rejecting run  $N$  is an accepting run for  $N'$ . Thus  $x$  is accepted by both  $N$  and  $N'$ .

	NFA $N$	Toggled NFA $N'$
Run 1	Rejecting	Accepting
Run 2	Accepting	Rejecting

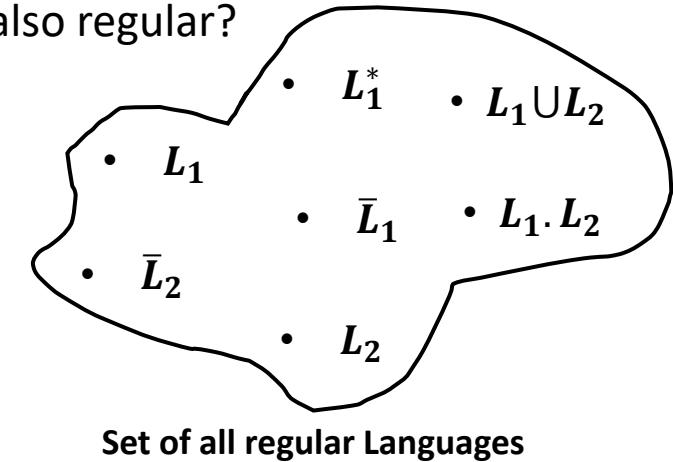
# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under complement**? If  $L$  is regular, then is  $\bar{L}$  also regular?

**Proof:** Given a DFA  $M$ , such that  $L(M) = L$ , construct the **toggled DFA  $M'$**  from  $M$ , by

- (i) changing all the non-final states of  $M$  to be the final states of  $M'$  and
- (ii) changing all the final states  $M$  to be the non-final states of  $M'$ .

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



**Q:** If  $L$  is the language accepted by an NFA, does “toggling” its states result in an NFA that accepts  $\bar{L}$ ?

**Proof:** Consider that for an input string  $x \in L$ , such that  $N$  accepts it. Suppose there is an rejecting run and an accepting run for input  $x$ . (See Table)

For toggled NFA  $N'$  too, there are two runs for  $x$ . However, the rejecting run  $N$  is an accepting run for  $N'$ . Thus  $x$  is accepted by both  $N$  and  $N'$ .

**Contradiction!** So No, the **toggled NFA does not accept  $\bar{L}$** .

	NFA $N$	Toggled NFA $N'$
Run 1	Rejecting	Accepting
Run 2	Accepting	Rejecting

**Thank You!**

# CS 302.1 - Automata Theory

## Lecture 03

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad



INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

H Y D E R A B A D

# Quick Recap

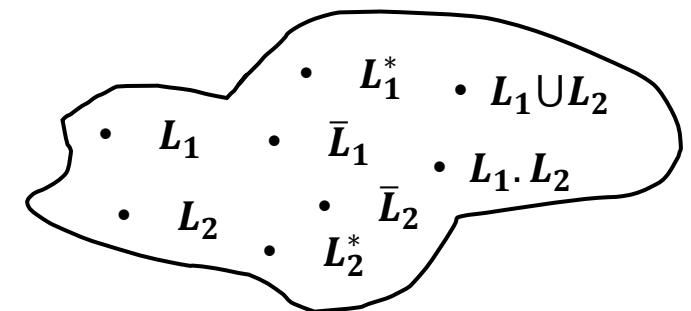
- DFAs and NFAs are equivalent
- For every NFA we can obtain a “Remembering DFA” that accepts the same language.
- The language accepted by finite automata are called Regular Languages.
- Regular operations: Union, Complement, Concatenation, **Star**.

# Quick Recap

- DFAs and NFAs are equivalent
- For every NFA we can obtain a “Remembering DFA” that accepts the same language.
- The language accepted by finite automata are called Regular Languages.
- Regular operations: Union, Complement, Concatenation, **Star**.
- **Star:**  $L_1^* = \{x_1x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$ . Examples:
  - If  $\Sigma = \{a\}$ ,  $\Sigma^* = \{\epsilon, a, aa, aaa, \dots \dots\}$

# Quick Recap

- DFAs and NFAs are equivalent
- For every NFA we can obtain a “Remembering DFA” that accepts the same language.
- The language accepted by finite automata are called Regular Languages.
- Regular operations: Union, Complement, Concatenation, **Star**.
- **Star:**  $L_1^* = \{x_1x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$ . Examples:
  - If  $\Sigma = \{a\}$ ,  $\Sigma^* = \{\epsilon, a, aa, aaa, \dots \dots\}$
  - If  $\Sigma = \{\Phi\}$ ,  $\Sigma^* = \{\epsilon\}$
- Regular Languages are closed under: Union, Star, Concatenation, Complement,...



Set of all regular Languages

# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under intersection**? If  $L_1$  and  $L_2$  are regular, then is  $L = L_1 \cap L_2$  also regular?

**Proof:** We shall use the fact that regular languages are **closed** under union and complement.

# Closure of Regular Languages

**Q:** Is the set of all regular languages **closed under intersection**? If  $L_1$  and  $L_2$  are regular, then is  $L = L_1 \cap L_2$  also regular?

**Proof:** We shall use the fact that regular languages are **closed** under union and complement.

Note that using De Morgan's laws:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

# Closure of Regular Languages

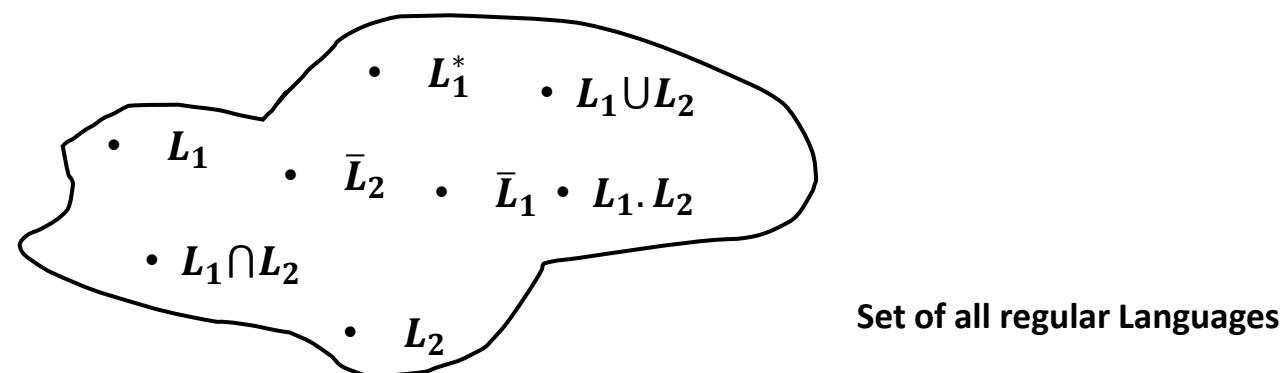
**Q:** Is the set of all regular languages **closed under intersection**? If  $L_1$  and  $L_2$  are regular, then is  $L = L_1 \cap L_2$  also regular?

**Proof:** We shall use the fact that regular languages are **closed** under union and complement.

Note that using De Morgan's laws:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Given a DFA for  $L_1$  and a DFA for  $L_2$ , we know how to construct an NFA for  $\overline{L_1}$ ,  $\overline{L_2}$  as well as for  $L_1 \cup L_2$ . Using these constructions and the aforementioned relationship, we can construct an NFA for  $L = L_1 \cap L_2$ .

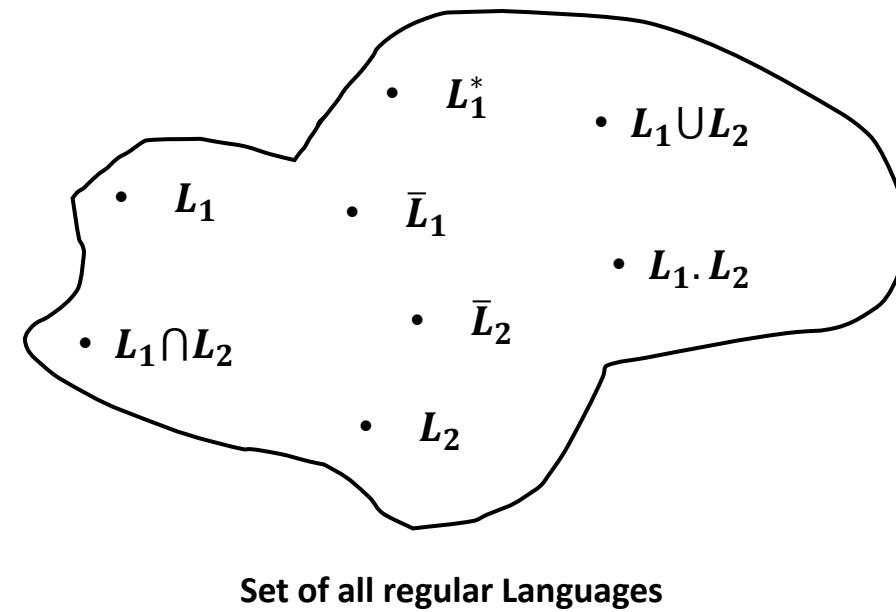


# Closure of Regular Languages

## Summary:

Regular Languages are closed under:

- **Union**
- **Intersection**
- **Star**
- **Complement**
- **Concatenation**



# Regular Languages

If  $\Sigma$  is an alphabet, then

- $\Sigma^0 = \{\epsilon\}$
- $\Sigma^2 = \{a_1 a_2 | a_1 \in \Sigma, a_2 \in \Sigma\}$
- $\Sigma^k = \{a_1 a_2 \cdots a_k | a_i \in \Sigma \mid 1 \leq i \leq k\}$
- $\Sigma^* = \{\bigcup_{i \geq 0} \Sigma^i\} = \{\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cdots\} = \{a_1 a_2 \cdots a_k | k \in \{0, 1, \dots\} \text{ & } a_j \in \Sigma, \forall j \in \{1, 2, \dots, k\}\}$

A Language  $L \subset \Sigma^*$  and  $L^* = \{\bigcup_{i \geq 0} L^i\}$

# Regular Languages

If  $\Sigma$  is an alphabet, then

- $\Sigma^0 = \{\epsilon\}$
- $\Sigma^2 = \{a_1 a_2 | a_1 \in \Sigma, a_2 \in \Sigma\}$
- $\Sigma^k = \{a_1 a_2 \cdots a_k | a_i \in \Sigma | 1 \leq i \leq k\}$
- $\Sigma^* = \{\cup_{i \geq 0} \Sigma^i\} = \{\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cdots\} = \{a_1 a_2 \cdots a_k | k \in \{0, 1, \dots\} \text{ & } a_j \in \Sigma, \forall j \in \{1, 2, \dots, k\}\}$

A Language  $L \subset \Sigma^*$  and  $L^* = \{\cup_{i \geq 0} L^i\}$

**Regular Language (alternate definition):** Let  $\Sigma$  be an alphabet. Then the following are the regular languages over  $\Sigma$ :

- The empty language  $\Phi$  is regular
- For each  $a \in \Sigma, \{a\}$  is regular.
- Let  $L_1, L_2$  be regular languages. Then  $L_1 \cup L_2, L_1 \cdot L_2, L_1^*$  are regular languages.

# Regular Expressions

A regular expression describes regular languages algebraically. The algebraic formulation also provides a powerful set of tools which will be leveraged to prove

- languages are regular
- derive properties of regular languages

# Regular Expressions

A regular expression describes regular languages algebraically. The algebraic formulation also provides a powerful set of tools which will be leveraged to prove

- languages are regular
- derive properties of regular languages

**Syntax for regular expressions (Recursive definition):** R is said to be a regular expression if it has one of the following forms:

- $\Phi$  is a regular expression,  $L(\Phi) = \Phi$
- $\epsilon$  is a regular expression,  $L(\epsilon) = \{\epsilon\}$
- Any  $a \in \Sigma$  is a regular expression,  $L(a) = \{a\}$

# Regular Expressions

A regular expression describes regular languages algebraically. The algebraic formulation also provides a powerful set of tools which will be leveraged to prove

- languages are regular
- derive properties of regular languages

**Syntax for regular expressions (Recursive definition):** R is said to be a regular expression if it has one of the following forms:

- $\Phi$  is a regular expression,  $L(\Phi) = \Phi$
- $\epsilon$  is a regular expression,  $L(\epsilon) = \{\epsilon\}$
- Any  $a \in \Sigma$  is a regular expression,  $L(a) = \{a\}$
- $R_1 + R_2$  is a regular expression if  $R_1$  and  $R_2$  are regular expressions,  $L(R_1 + R_2) = L(R_1) \cup L(R_2)$
- $R^*$  is a regular expression if  $R$  is a regular expression,  $L(R^*) = (L(R))^*$

# Regular Expressions

A regular expression describes regular languages algebraically. The algebraic formulation also provides a powerful set of tools which will be leveraged to prove

- languages are regular
- derive properties of regular languages

**Syntax for regular expressions (Recursive definition):** R is said to be a regular expression if it has one of the following forms:

- $\Phi$  is a regular expression,  $L(\Phi) = \Phi$
- $\epsilon$  is a regular expression,  $L(\epsilon) = \{\epsilon\}$
- Any  $a \in \Sigma$  is a regular expression,  $L(a) = \{a\}$
- $R_1 + R_2$  is a regular expression if  $R_1$  and  $R_2$  are regular expressions,  $L(R_1 + R_2) = L(R_1) \cup L(R_2)$
- $R^*$  is a regular expression if  $R$  is a regular expression,  $L(R^*) = (L(R))^*$
- $R_1 R_2$  is a regular expression if  $R_1$  and  $R_2$  are regular expressions,  $L(R_1 R_2) = L(R_1) \cdot L(R_2)$
- $(R)$  is a regular expression if  $R$  is a regular expression,  $L((R)) = R$

# Regular Expressions

Syntax for regular expressions:

Regular Expression	Regular Language	Comment
$\Phi$	$\{\}$	The empty set
$\epsilon$	$\{\epsilon\}$	The set containing $\epsilon$ only
$a$	$\{a\}$	Any $a \in \Sigma$
$R_1 + R_2$	$L(R_1) \cup L(R_2)$	For regular expressions $R_1$ and $R_2$
$R_1 R_2$	$L(R_1).L(R_2)$	For regular expressions $R_1$ and $R_2$
$R^*$	$(L(R))^*$	For regular expressions $R$
$(R)$	$L(R)$	For regular expressions $R$

Order of precedence:  $(\cdot)$ ,  $^*$ ,  $\cdot$ ,  $+$

A language  $L$  is regular if and only if for some regular expression  $R$ ,  $L(R) = L$ .

RE's are equivalent in power to NFAs/DFAs

# Regular Expressions

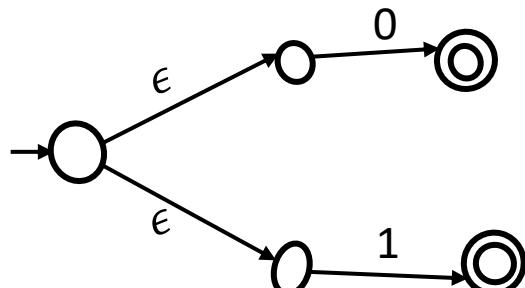
Syntax for regular expressions:

Regular Expression R	$L(R)$
01	{01}
$01 + 1$	{01,1}
$(0 + 1)^*$	{ $\epsilon$ , 0, 1, 00, 01, ... }
$(01 + \epsilon)1$	{011,1}
$(0 + 1)^*01$	{01, 001, 101, 0001, ... }
$(0 + 10)^*(\epsilon + 1)$	{ $\epsilon$ , 0, 10, 00, 001, 010, 0101, ... }

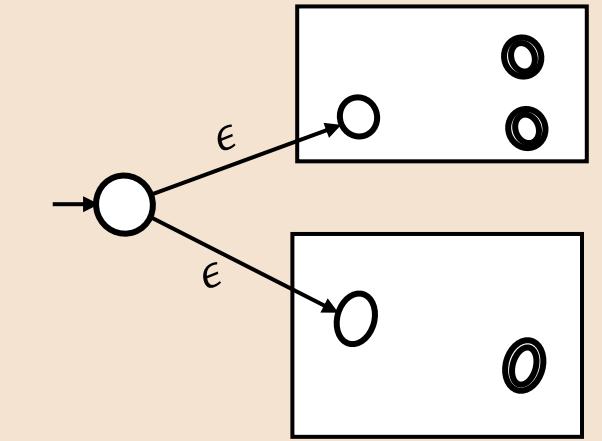
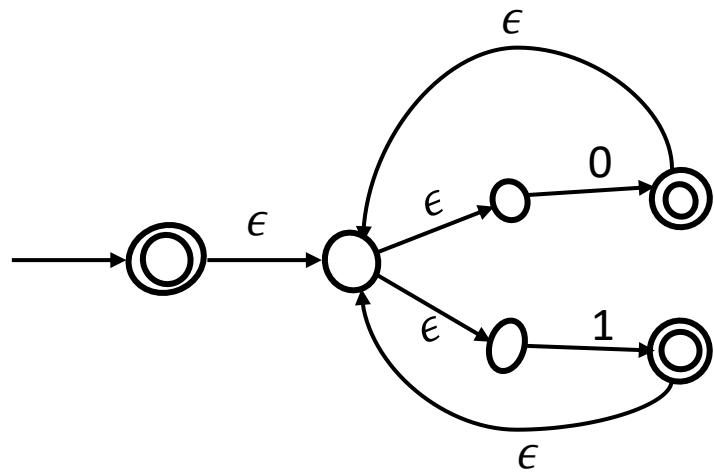
# Regular Expressions

NFA for RE:  $(0 + 1)^*01$

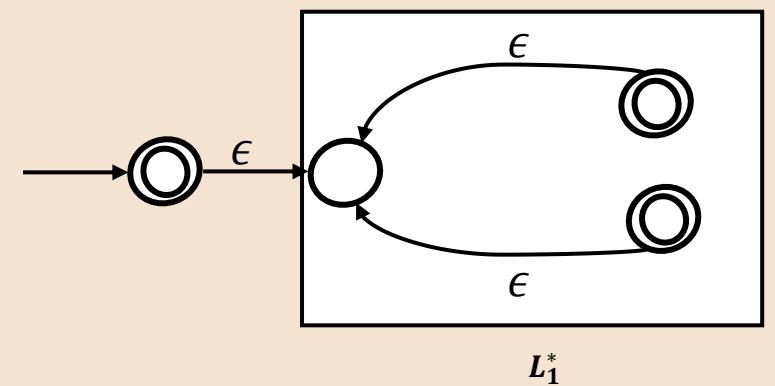
(i) NFA for  $(0 + 1)$



(ii) NFA for  $(0 + 1)^*$



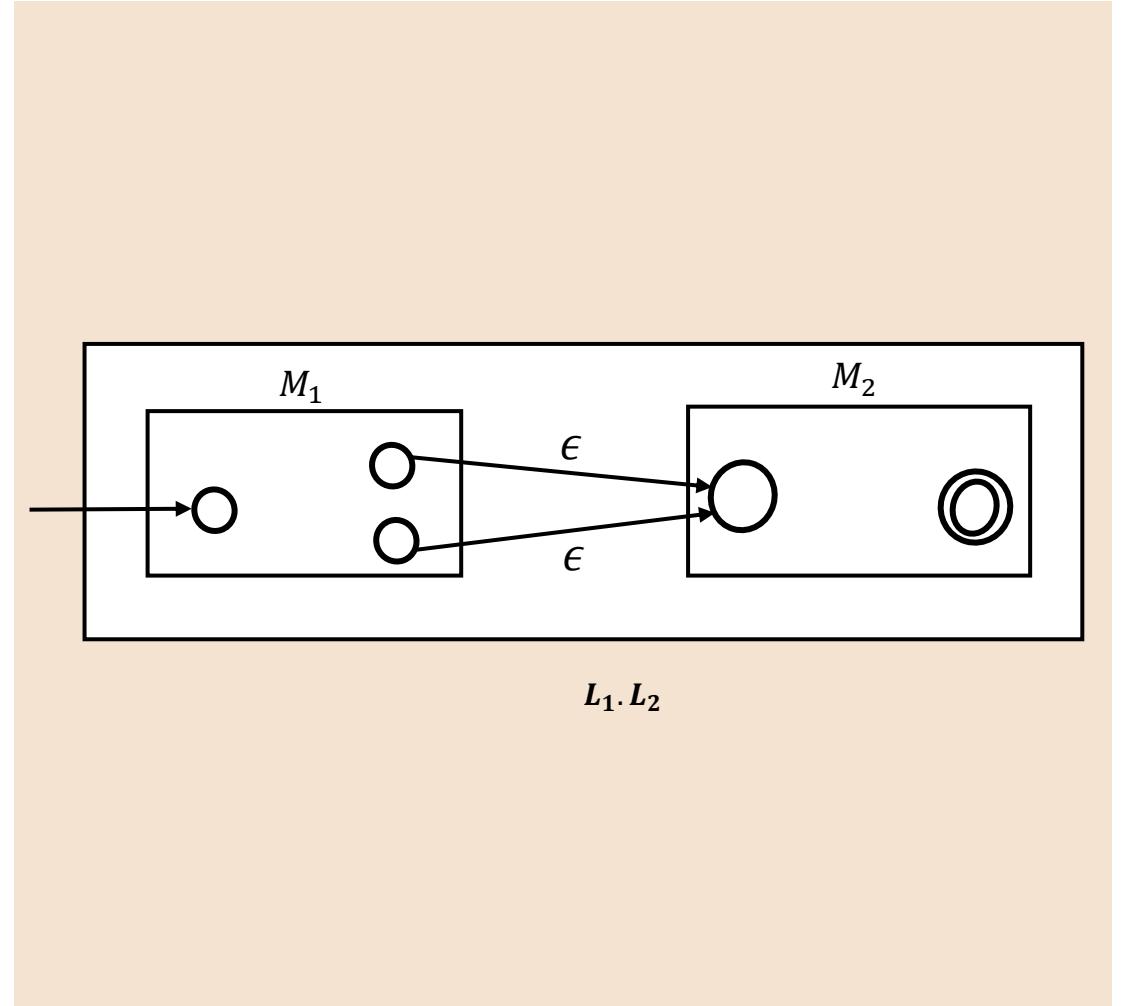
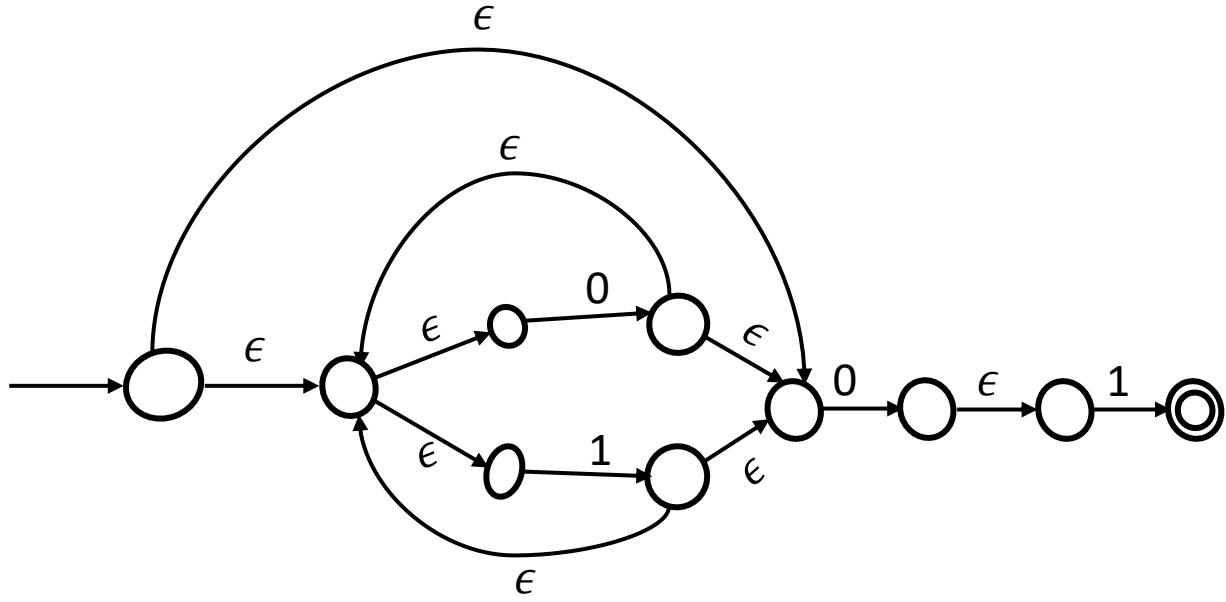
$L_1 \cup L_2$



$L_1^*$

# Regular Expressions

NFA for  $(0 + 1)^*01$



# Regular Expressions

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

Language	Regular Expression
$\{\omega \mid \omega \text{ ends in "ab"}\}$	$(a + b)^*ab$
$\{\omega \mid \omega \text{ has a single } a\}$	$b^*ab^*$
$\{\omega \mid \omega \text{ has at most one } a\}$	$b^* + b^*ab^*$
$\{\omega \mid  \omega  \text{ is even}\}$	$((a + b)(a + b))^* = (aa + bb + ab + ba)^*$
$\{\omega \mid \omega \text{ has "ab" as a substring}\}$	$(a + b)^*ab(a + b)^*$
$\{\omega \mid  \omega  \text{ is a multiple of 3}\}$	$((a + b)(a + b)(a + b))^*$

# Regular Expressions

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

Language	Regular Expression
$\{\omega \mid \omega \text{ ends in "ab"}\}$	$(a + b)^*ab$
$\{\omega \mid \omega \text{ has a single } a\}$	$b^*ab^*$
$\{\omega \mid \omega \text{ has at most one } a\}$	$b^* + b^*ab^*$
$\{\omega \mid  \omega  \text{ is even}\}$	$((a + b)(a + b))^* = (aa + bb + ab + ba)^*$
$\{\omega \mid \omega \text{ has "ab" as a substring}\}$	$(a + b)^*ab(a + b)^*$
$\{\omega \mid  \omega  \text{ is a multiple of 3}\}$	$((a + b)(a + b)(a + b))^*$

## Some algebraic properties of Regular Expressions:

- $R_1 + (R_2 + R_3) = (R_1 + R_2) + R_3$
- $R_1(R_2R_3) = (R_1R_2)R_3$
- $R_1(R_2 + R_3) = R_1R_2 + R_1R_3$
- $(R_1 + R_2)R_3 = R_1R_3 + R_2R_3$
- $R_1 + R_2 = R_2 + R_1$
- $R_1^*R_1^* = R_1^*$
- $(R_1^*)^* = R_1^*$
- $R\epsilon = \epsilon R = R$
- $R\Phi = \Phi R = \Phi$
- $R + \Phi = R$
- $\epsilon + RR^* = \epsilon + R^*R = R^*$
- $(R_1 + R_2)^* = (R_1^*R_2^*)^* = (R_1^* + R_2^*)^*$

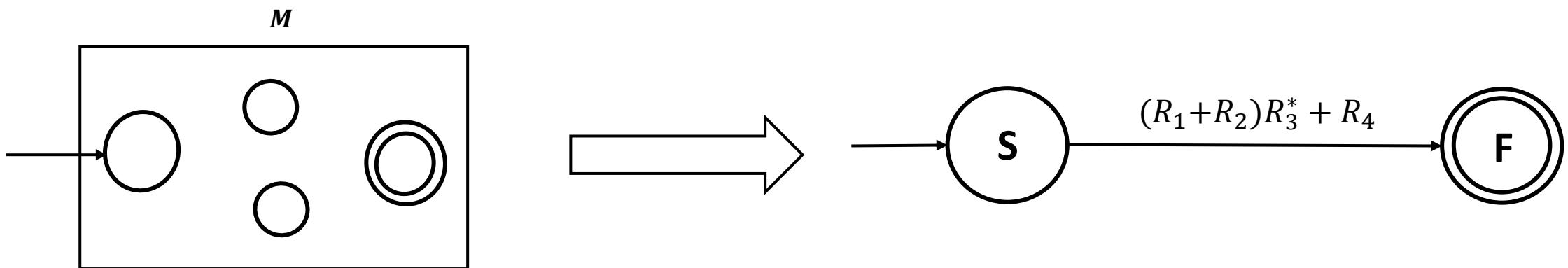
# DFA to Regular Expressions

If a language is regular then it accepts a regular expression. We could draw equivalent NFAs for Regular Expressions.

How can we obtain Regular expressions given a DFA?

Given a DFA  $M$ , we **recursively** construct a two-state **Generalized NFA** (GNFA) with

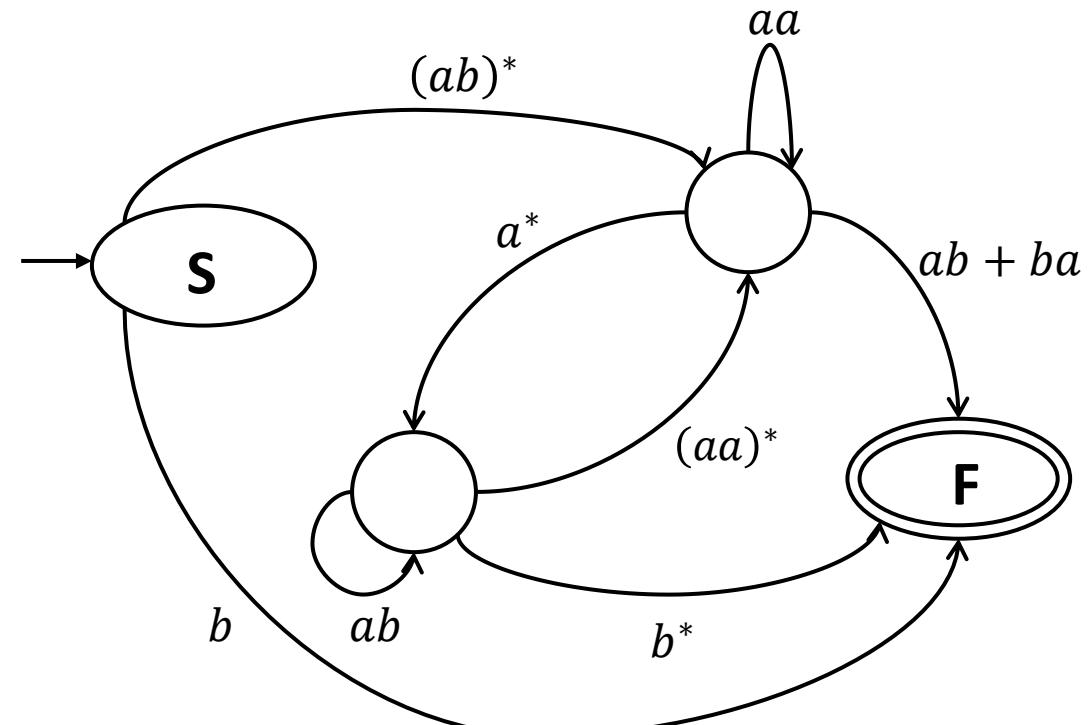
- A start state and a final state
- A single arrow goes from the start state to the final state
- The label of this arrow is the regular expression corresponding to the language accepted by the DFA  $M$ .



# DFA to Regular Expressions: GNFA

What are GNFs? They are simply NFAs such that

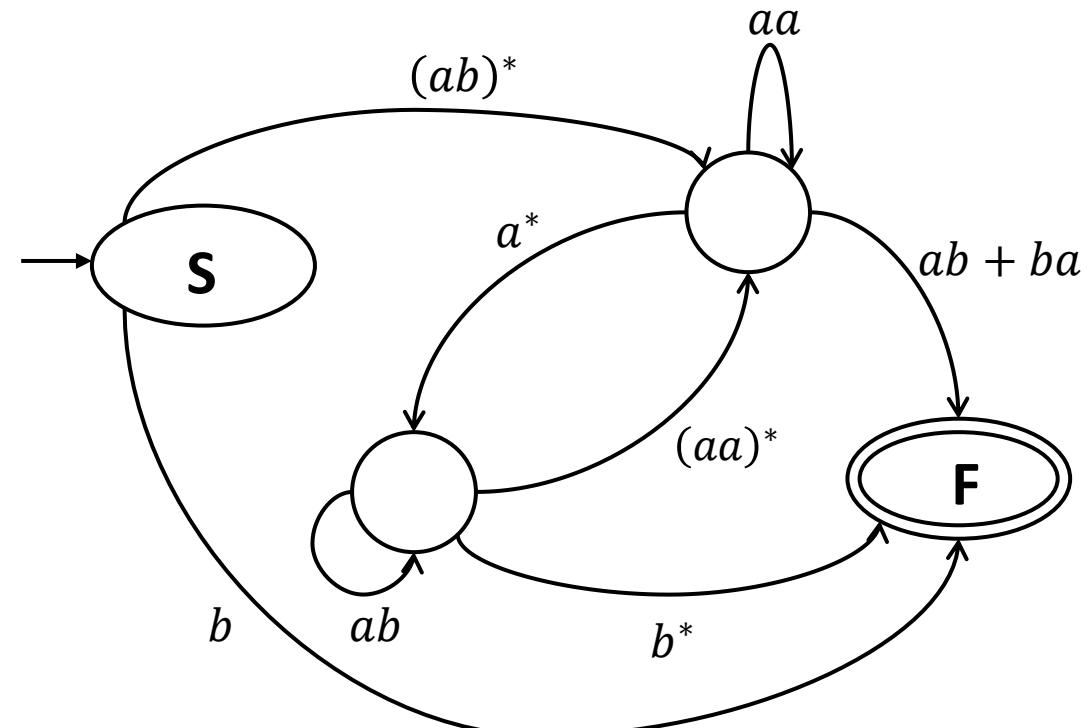
- The transitions may have regular expressions
- A unique start state that has arrows going to other states, but has no incoming arrows
- A unique final state that has arrows incoming from other states, but has no outgoing arrows
- For an input string, **runs** on a GNFA are similar to that of an NFA, except now a block of symbols are read corresponding to the Regular Expressions on the transitions.
- $b$ ,  $abababab$ ,  $abaaaba$  are some input strings that have accepting runs for the GNFA on the right



# DFA to Regular Expressions: GNFA

What are GNFs? They are simply NFAs such that

- The transitions may have regular expressions
- A unique start state that has arrows going to other states, but has no incoming arrows
- A unique final state that has arrows incoming from other states, but has no outgoing arrows
- For an input string, **runs** on a GNFA are similar to that of an NFA, except now a block of symbols are read corresponding to the Regular Expressions on the transitions.
- $b$ ,  $abababab$ ,  $abaaaba$  are some input strings that have accepting runs for the GNFA on the right

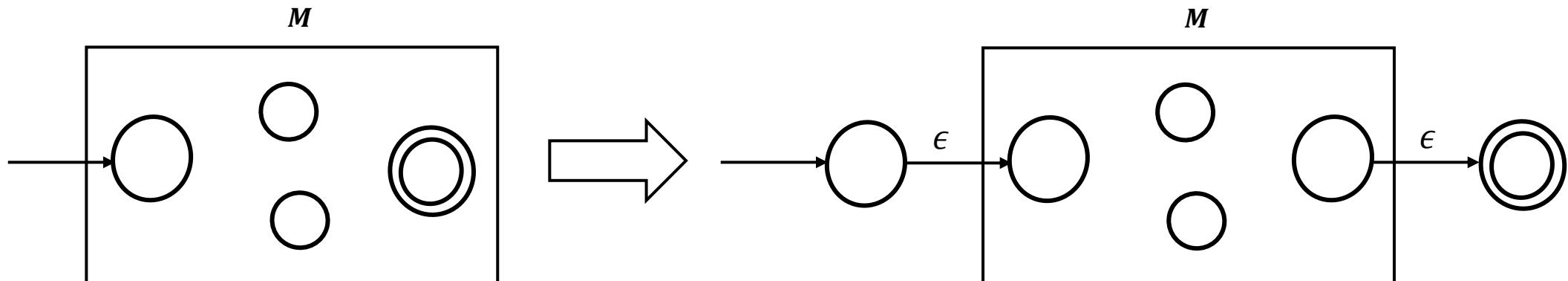


Starting from a DFA we will begin by constructing a GNFA with  $k$  states. We then outline a recursive procedure by which at each step, we will construct a GNFA with one less state. This step will be repeated until we obtain the **2-state GNFA**.

# DFA to Regular Expressions: GNFA

Starting from the DFA  $M$ ,

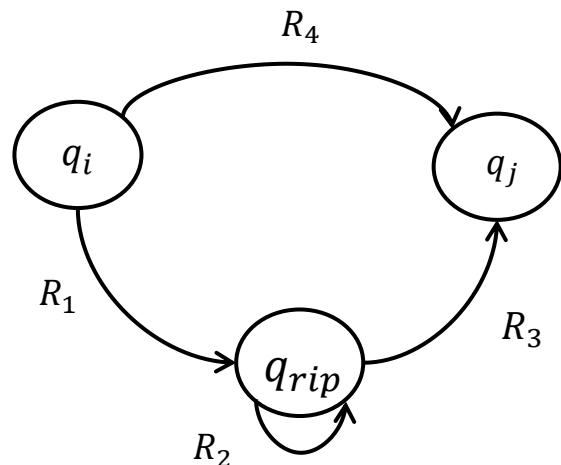
- Add a new start state with an  $\epsilon$  arrow to the old start state.
- Add a new final state by with an  $\epsilon$  arrow to the old final state.



# DFA to Regular Expressions: GNFA

The crucial step is to convert a GNFA with  $k (>2)$  states to a GNFA with  $k - 1$  states. This is what we shall show next.

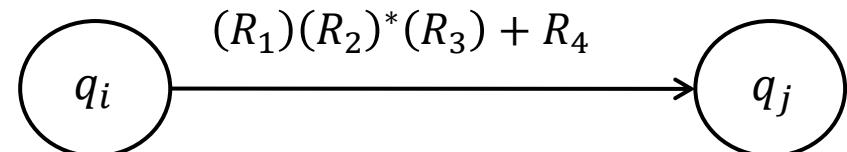
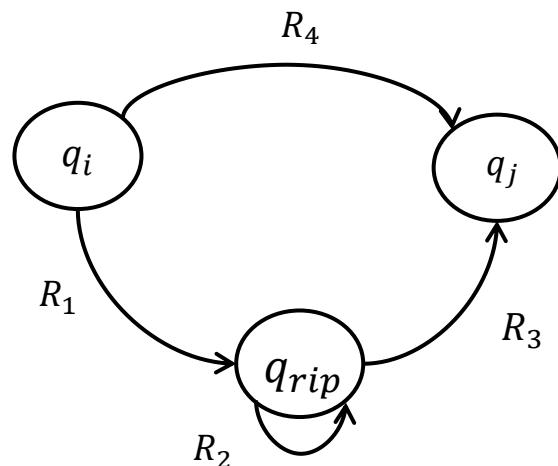
- Start by picking any state of the GNFA (except the new start and final states)
- Let us call this state  $q_{rip}$ . We “rip”  $q_{rip}$  out of the machine and create a GNFA with  $k - 1$  states.
- Of course, we need to “repair” the machine by altering the regular expressions that label each of the remaining arrows.
- The new labels compensate for the loss of  $q_{rip}$ .



# DFA to Regular Expressions: GNFA

The crucial step is to convert a GNFA with  $k (>2)$  states to a GNFA with  $k - 1$  states. This is what we shall show next.

- Start by picking any state of the GNFA (except the new start and final states)
- Let us call this state  $q_{rip}$ . We “rip”  $q_{rip}$  out of the machine and create a GNFA with  $k - 1$  states.
- Of course, we need to “repair” the machine by altering the regular expressions that label each of the remaining arrows.
- The new labels compensate for the loss of  $q_{rip}$ .



# DFA to Regular Expressions: GNFA

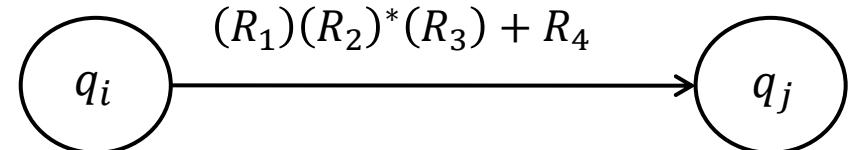
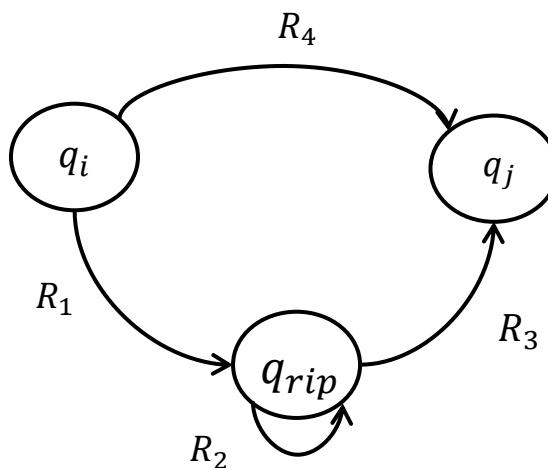
The crucial step is to convert a GNFA with  $k (>2)$  states to a GNFA with  $k - 1$  states.

How do we remove  $q_{rip}$ ? In the old machine if

- $q_i$  goes to  $q_{rip}$  with an arrow labelled  $R_1$
- $q_{rip}$  goes to itself with an arrow labelled  $R_2$
- $q_{rip}$  goes to  $q_j$  with an arrow labelled  $R_3$
- $q_i$  goes to  $q_j$  with an arrow labelled  $R_4$

**Repeat this until  $k = 2$**

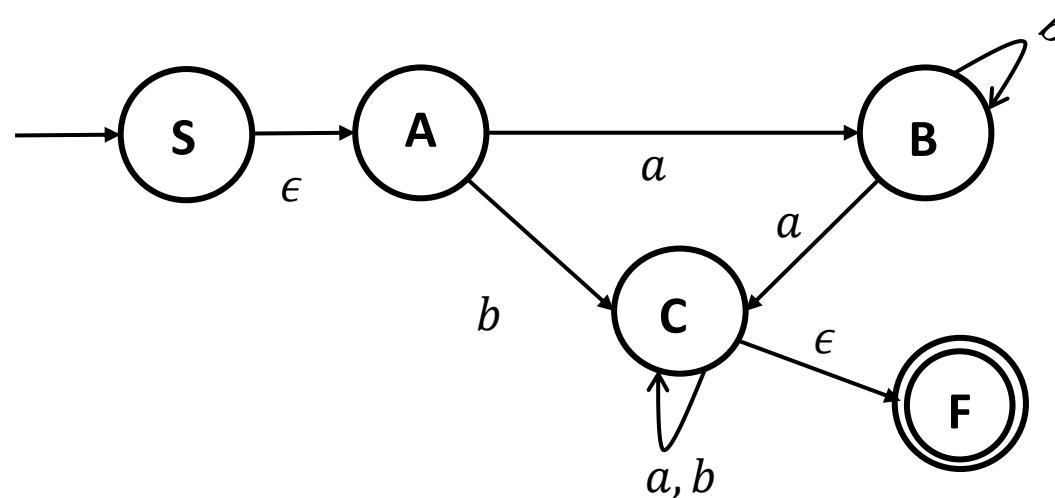
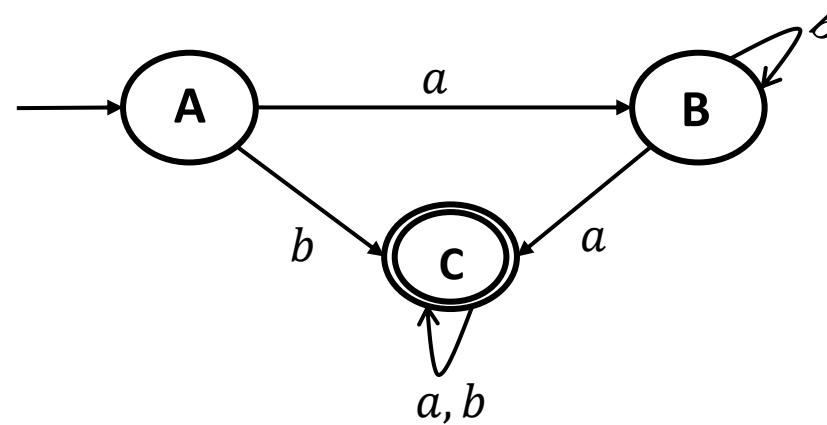
then in the new machine, the arrow from  $q_i$  to  $q_j$  has the label  $(R_1)(R_2)^*(R_3) + R_4$



This should be done for **every pair** of arrows outgoing and incoming  $q_{rip}$

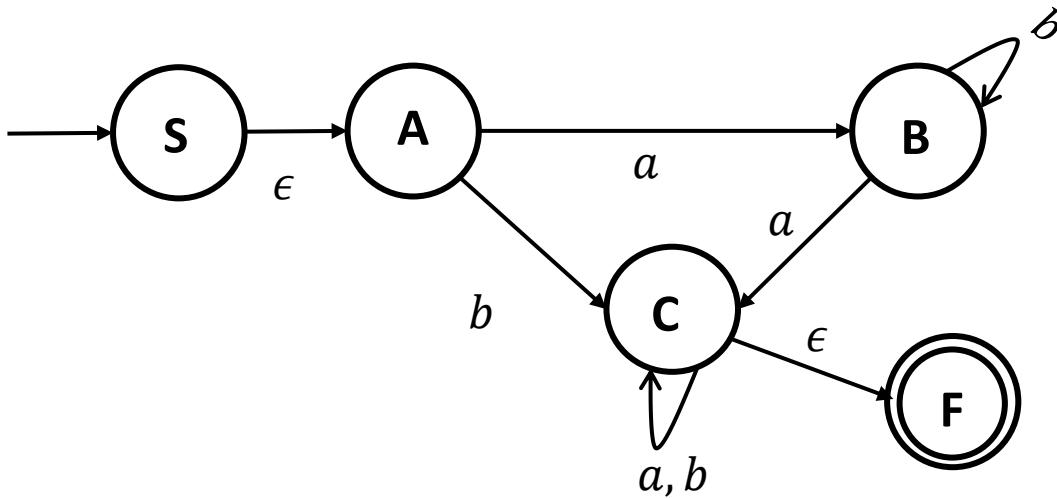
# DFA to Regular Expressions: GNFA

Let us look at an example. Consider the original DFA  $M$  below and find the regular expression corresponding to  $L(M)$ .

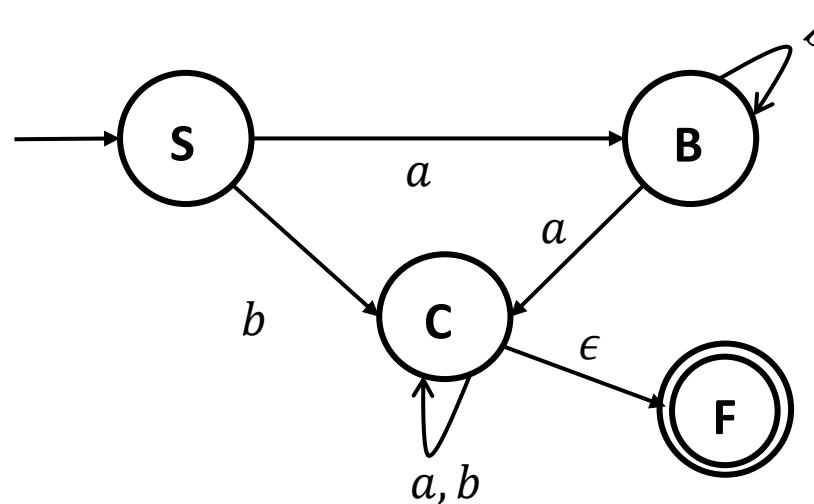


**Step 1: Add new start and final states**

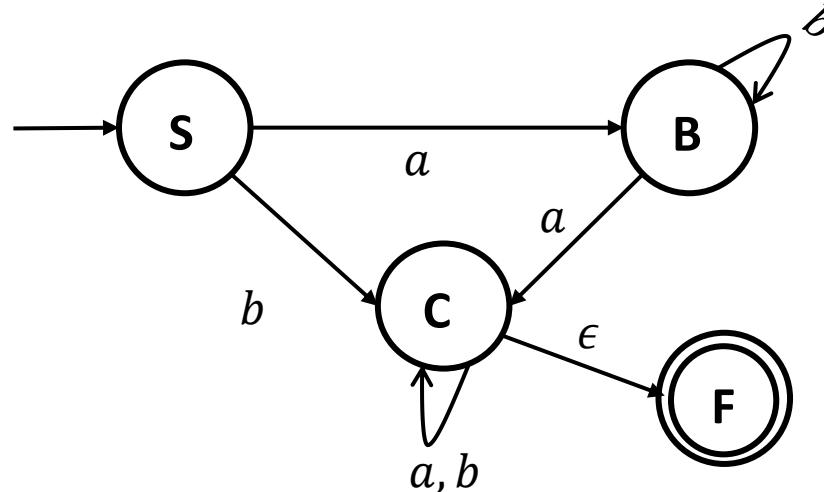
# DFA to Regular Expressions: GNFA



**Step 2: Eliminate A**

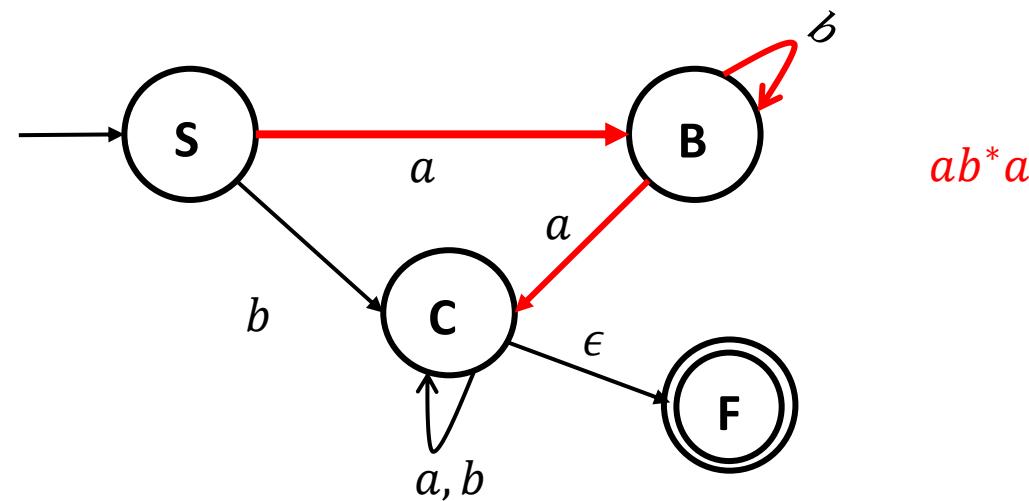


# DFA to Regular Expressions: GNFA

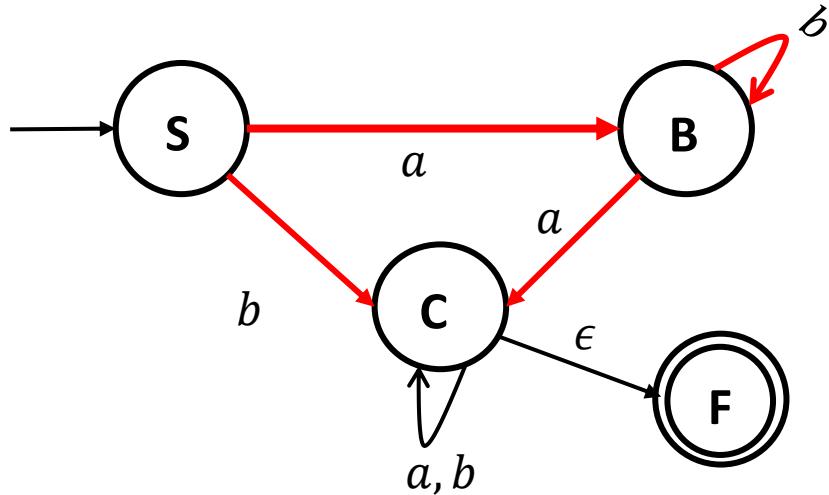


**Step 2: Eliminate B**

$S \rightarrow C$  via  $B$ , RE:  $ab^*a$



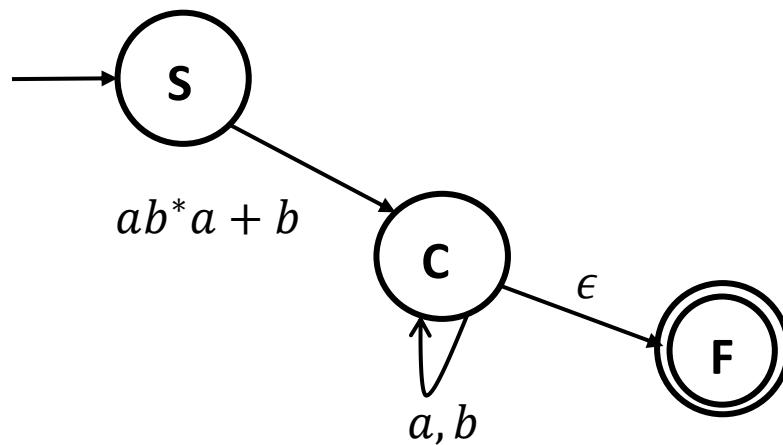
# DFA to Regular Expressions: GNFA



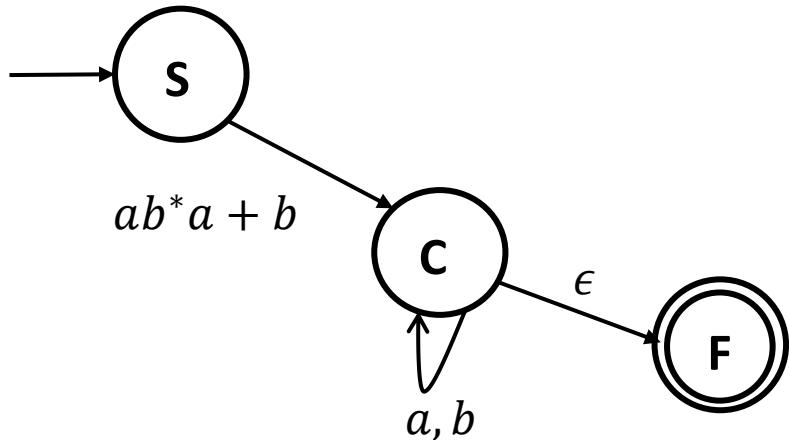
**Step 2: Eliminate  $B$**

$S \rightarrow C$  via  $B$ , RE:  $ab^*a$

Overall RE for  $S \rightarrow C$ :  $ab^*a + b$

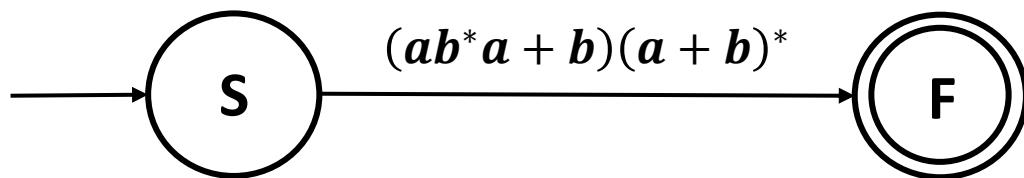


# DFA to Regular Expressions: GNFA

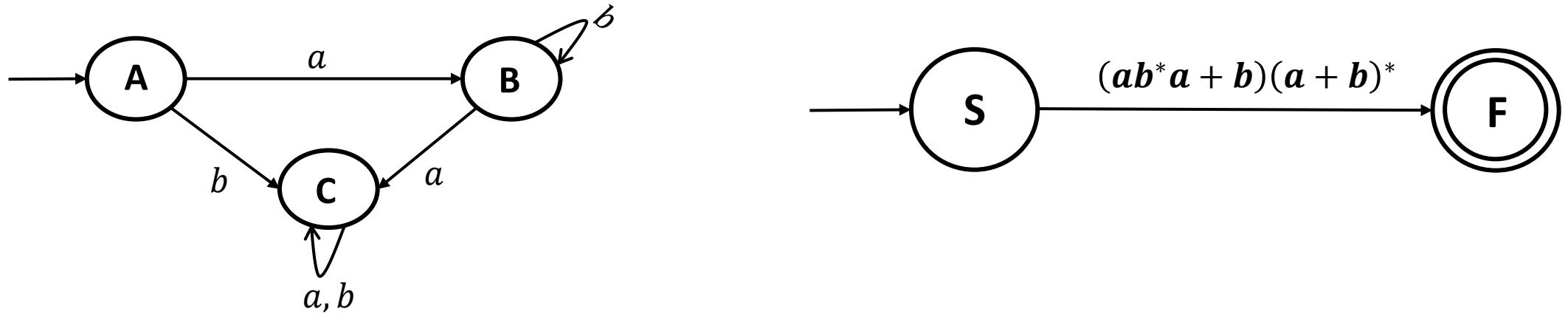


**Step 2: Eliminate C**

$S \rightarrow F$  via C, RE:  $(ab^*a + b)(a + b)^*$



# DFA to Regular Expressions: GNFA



Recursively, we managed to convert the DFA  $M$  to a 2-state GNFA such that the label from the start state to the final state of the GNFA is the Regular Expression corresponding to  $L(M)$ .

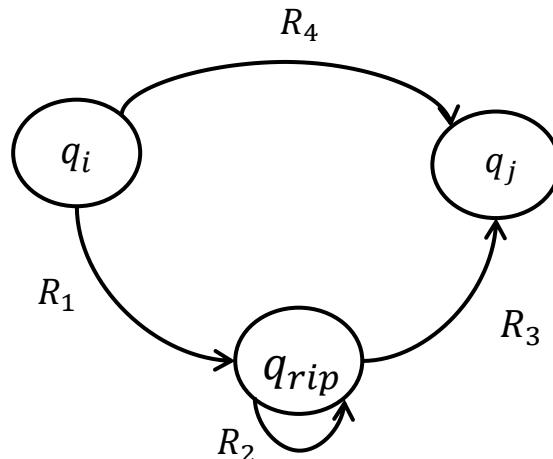
# DFA to Regular Expressions: GNFA

Formally, a GNFA is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where

- $Q$  is a finite set of states.
- $\Sigma$  is the input alphabet.
- $\delta: Q - \{q_0\} \times Q - \{F\} \mapsto \mathcal{R}$  is the transition function.
- $q_0$  is the start state.
- $F$  is the final state.

## Convert $k$ -state GNFA to a 2-state GNFA:

We provide a recursive algorithm  $\text{CONVERT}(G)$  for this.



### **CONVERT( $G$ ):**

1. Let  $k$  be the number of states of  $G$ .
2. If  $k = 2$ , then return the label  $R$  of the arrow between the start and the final state.
3. If  $k > 2$ , select any state  $Q$  different from  $q_0$  and  $F$  and let  $G'$  be the GNFA  $(Q', \Sigma, \delta', q_0, F)$ , where

$$Q' = Q - \{q_{rip}\},$$

and for any  $q_i \in Q' - \{q_0\}$  and any  $q_j \in Q' - \{q_0\}$ , let

$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) + R_4,$$

for  $R_1 = \delta(q_i, q_{rip})$ ,  $R_2 = \delta(q_{rip}, q_{rip})$ ,  $R_3 = \delta(q_{rip}, q_j)$  and  $R_4 = \delta(q_i, q_j)$

4. Compute  $\text{CONVERT}(G')$  and return its value.

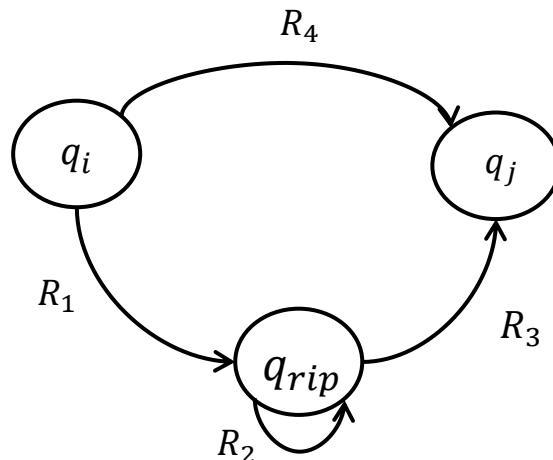
# DFA to Regular Expressions: GNFA

Formally, a GNFA is a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$  where

- $Q$  is a finite set of states.
- $\Sigma$  is the input alphabet.
- $\delta: Q - \{q_0\} \times Q - \{F\} \mapsto \mathcal{R}$  is the transition function.
- $q_0$  is the start state.
- $F$  is the final state.

**Convert  $k$ -state GNFA to a 2-state GNFA:**

We provide a recursive algorithm  
CONVERT( $G$ ) for this.



**DFA, NFA, Regular Expressions have equal power and all of them correspond to Regular Languages**

**How do Non-regular languages look like?  
How can we prove that certain languages are not regular?**

# Pumping Lemma

Recall that so far, we have proven that the following statements are all equivalent:

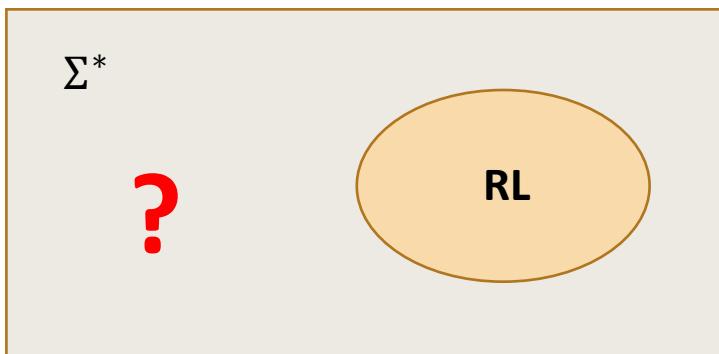
- $L$  is a regular language.
- There is a DFA  $D$  such that  $\mathcal{L}(D) = L$ .
- There is an NFA  $N$  such that  $\mathcal{L}(N) = L$ .
- There is a regular expression  $R$  such that  $\mathcal{L}(R) = L$ .
- Not all languages are regular.



# Pumping Lemma

Recall that so far, we have proven that the following statements are all equivalent:

- $L$  is a regular language.
- There is a DFA  $D$  such that  $\mathcal{L}(D) = L$ .
- There is an NFA  $N$  such that  $\mathcal{L}(N) = L$ .
- There is a regular expression  $R$  such that  $\mathcal{L}(R) = L$ .
- Not all languages are regular.



# Pumping Lemma

How do we prove that certain languages are non-regular? We start with an example

Let  $\Sigma = \{0,1\}$ . Consider the language  $L = \{0^n 1^n \mid n \geq 0\}$  and the following conversation between Karl and Mil.

# Pumping Lemma

How do we prove that certain languages are non-regular? We start with an example

Let  $\Sigma = \{0,1\}$ . Consider the language  $L = \{0^n 1^n \mid n \geq 0\}$  and the following conversation between Karl and Mil.

**Mil:** I have a DFA for  $L$ .

**Karl:** How many states are there?

**Mil:**  $n$ -states (say  $n = 10$ )

# Pumping Lemma

How do we prove that certain languages are non-regular? We start with an example

Let  $\Sigma = \{0,1\}$ . Consider the language  $L = \{0^n 1^n \mid n \geq 0\}$  and the following conversation between Karl and Mil.

**Mil:** I have a DFA for  $L$ .

**Karl:** How many states are there?

**Mil:**  $n$ -states (say  $n = 10$ )

**Karl:** Then  $0^{10} 1^{10}$  must be accepted.

By the **pigeonhole principle**, while reading the first ( $n = 10$ ) symbols, some states need to be revisited. Otherwise  $n + 1 = 11$  states would have been present. Hence some loop must be present. How many states are there in the loop?

# Pumping Lemma

How do we prove that certain languages are non-regular? We start with an example

Let  $\Sigma = \{0,1\}$ . Consider the language  $L = \{0^n 1^n \mid n \geq 0\}$  and the following conversation between Karl and Mil.

**Mil:** I have a DFA for  $L$ .

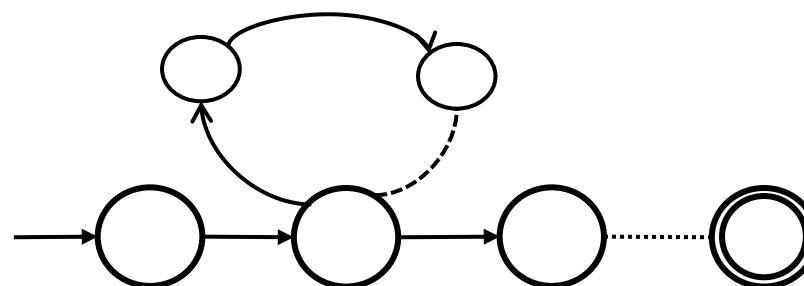
**Karl:** How many states are there?

**Mil:**  $n$ -states (say  $n = 10$ )

**Karl:** Then  $0^{10} 1^{10}$  must be accepted. By the **pigeonhole principle**, while reading the first ( $n = 10$ ) symbols, some states need to be revisited. Otherwise  $n + 1 = 11$  states would have been present. Hence some loop must be present. How many states are there in the loop?

**Mil:**  $t$ -states (say  $t = 3$ ).

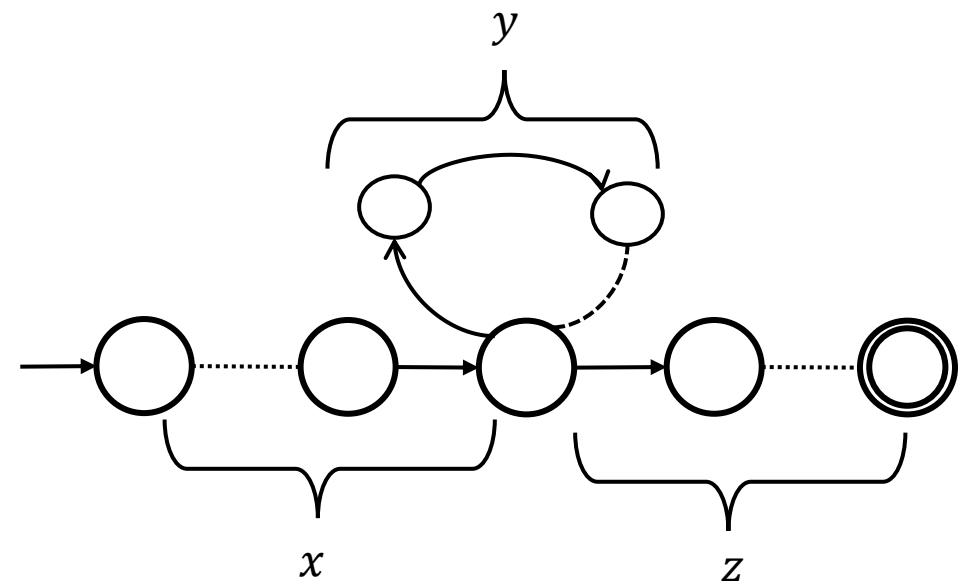
**Karl:** If your DFA accepts  $0^n 1^n$ , it must also accept  $0^{n+t} 1^n$ . This is because, if we take the loop one extra time, we read  $t$  more 0's.



**Contradiction as  $0^{n+t} 1^n \notin L$ . So Mil, you never had a DFA for  $L$  and in fact,  $L$  is not regular.**

# Pumping Lemma

If  $L$  is a regular language, all strings in the language, larger than a certain length (pumping length) , can be *pumped*: the string contains a certain section that can be repeated *any number of times* and the resulting string still  $\in L$ .

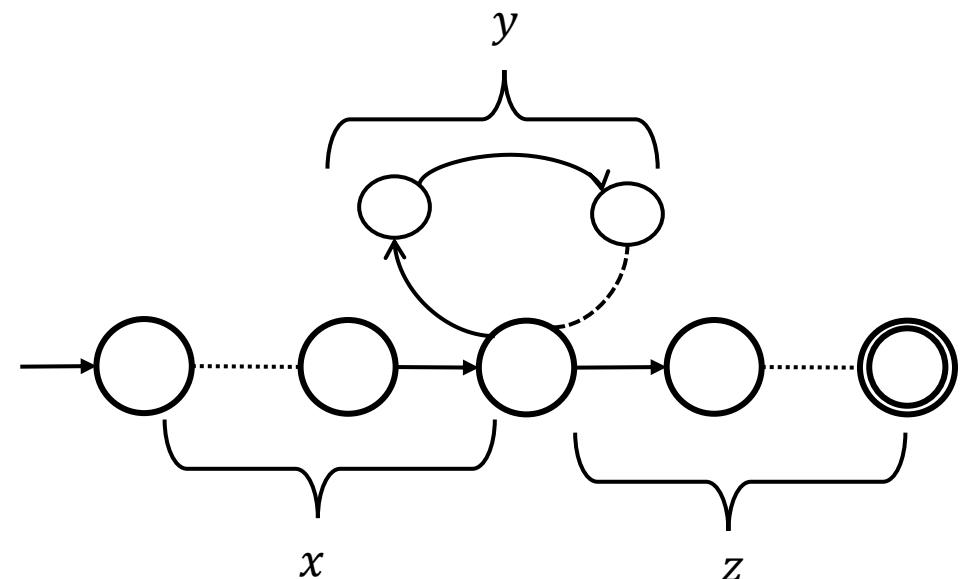


# Pumping Lemma

If  $L$  is a regular language, all strings in the language, larger than a certain length (pumping length) , can be *pumped*: the string contains a certain section that can be repeated *any number of times* and the resulting string still  $\in L$ .

**(Pumping Lemma)** If  $L$  is a regular language, then there exists a number  $p$  (the pumping length) where for all  $s \in L$  of length at least  $p$ , there exists  $x, y, z$  such that  $s = xyz$ , such that

1.  $|xy| \leq p$ .
2.  $|y| \geq 1$
3.  $\forall i \geq 0, xy^i z \in L$ .



# Pumping Lemma

If  $L$  is a regular language, all strings in the language, larger than a certain length (pumping length) , can be *pumped*: the string contains a certain section that can be repeated *any number of times* and the resulting string still  $\in L$ .

**(Pumping Lemma)** If  $L$  is a regular language, then there exists a number  $p$  (the pumping length) where for all  $s \in L$  of length at least  $p$ , there exists  $x, y, z$  such that  $s = xyz$ , such that

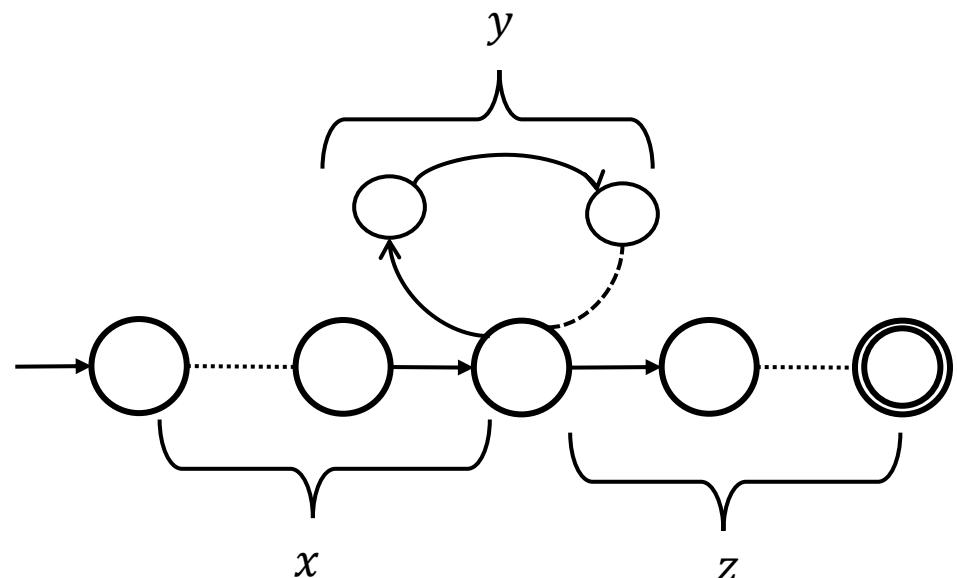
1.  $|xy| \leq p$ .
2.  $|y| \geq 1$
3.  $\forall i \geq 0, xy^i z \in L$ .

**Note:**  $(A \Rightarrow B) \equiv (\neg B) \Rightarrow (\neg A)$

If  $L$  is regular then, pumping property is satisfied

$\equiv$

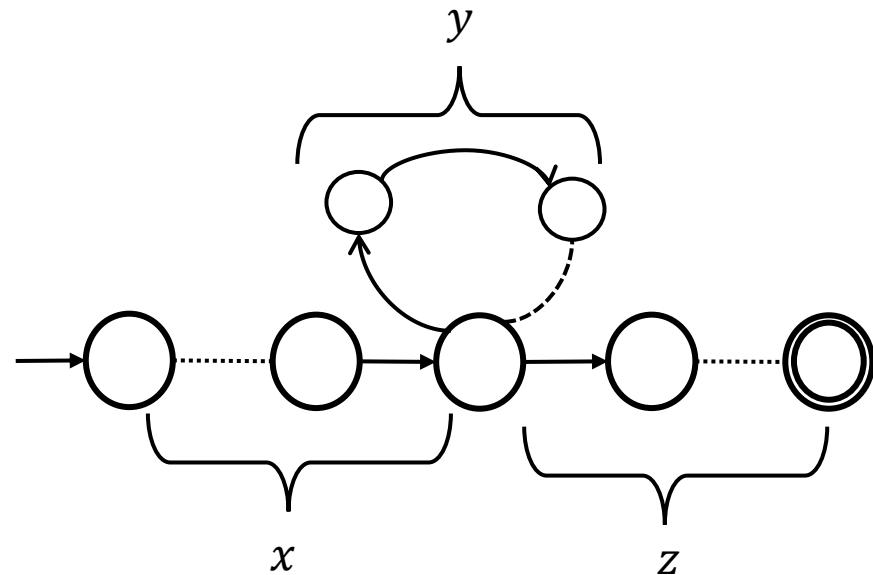
If pumping property is NOT satisfied, then  $L$  is NOT regular.



# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.



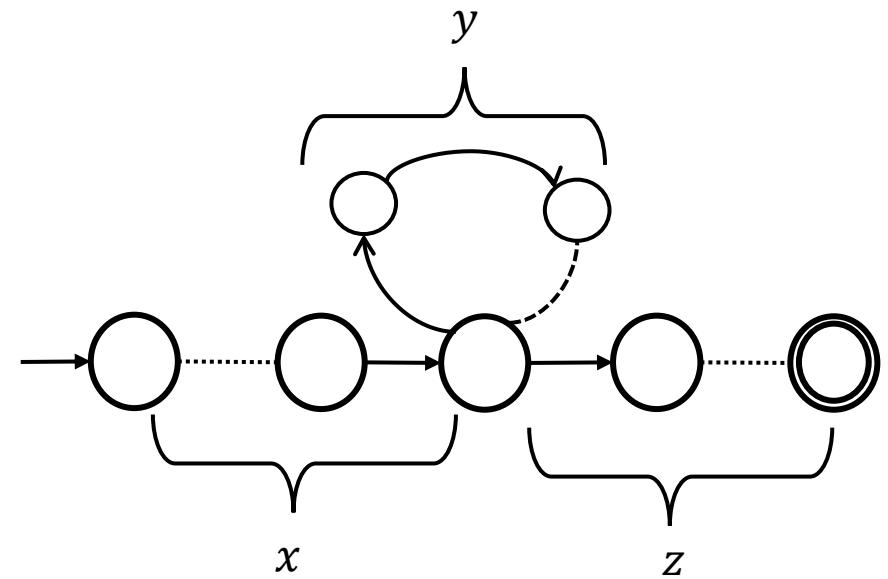
# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .



# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

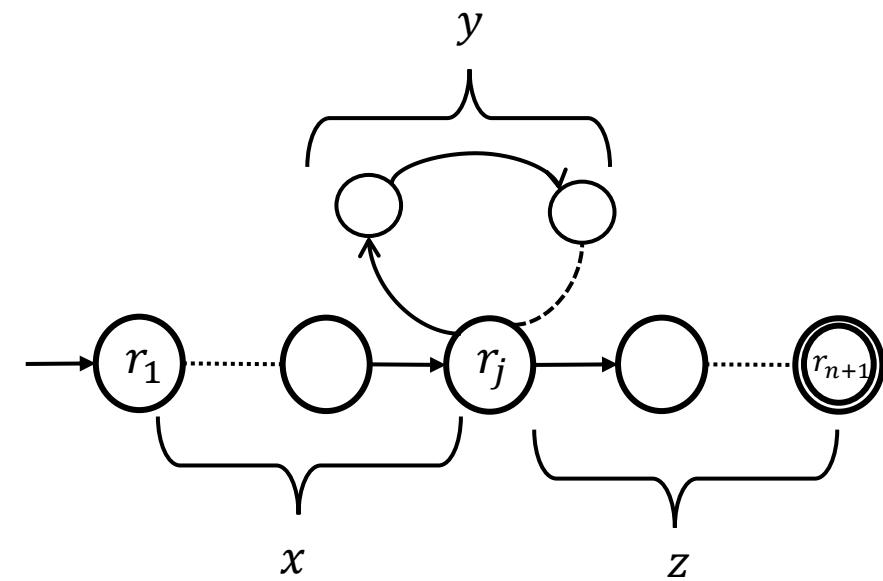
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .

So we can divide the  $s$  into three parts,  $x = s_1 \dots s_{j-1}$ ,  $y = s_j \dots s_{l-1}$ ,  $z = s_l \dots s_n$ . For a run on  $M$ , due to  $s$

- the  $x$  part takes us from  $r_1$  to  $r_j$
- the  $y$  part belongs to the loop part (we go from  $r_j$  to  $r_j$ )
- $z$  takes us from  $r_j$  to  $r_{n+1}$ , which is a final state if  $s \in L$ .



# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

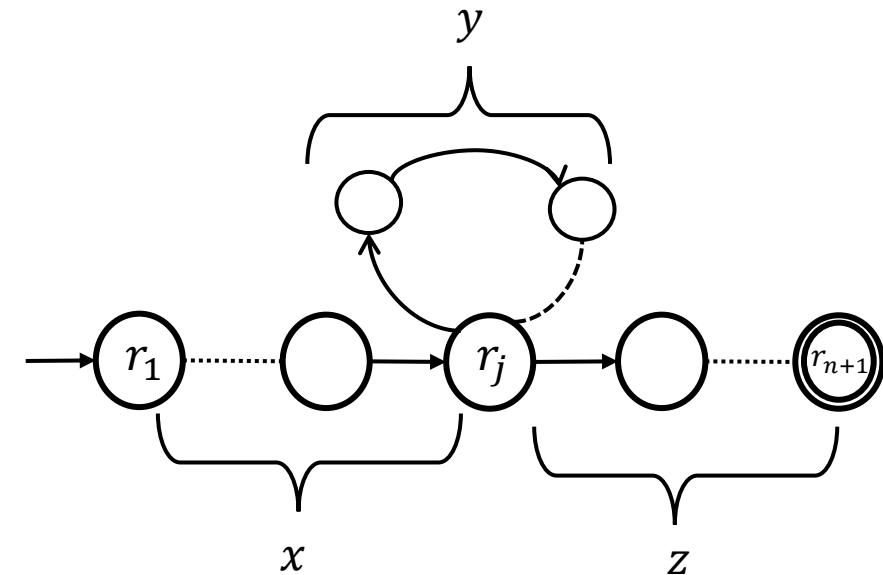
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .

So we can divide the  $s$  into three parts,  $x = s_1 \dots s_{j-1}$ ,  $y = s_j \dots s_{l-1}$ ,  $z = s_l \dots s_n$ . For a run on  $M$ , due to  $s$

- the  $x$  part takes us from  $r_1$  to  $r_j$
- the  $y$  part belongs to the loop part (we go from  $r_j$  to  $r_j$ )
- $z$  takes us from  $r_j$  to  $r_{n+1}$ , which is a final state if  $s \in L$ .



- We can traverse the loop bit any number of times and so  $\forall i \geq 0, xy^i z \in L$ .

# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

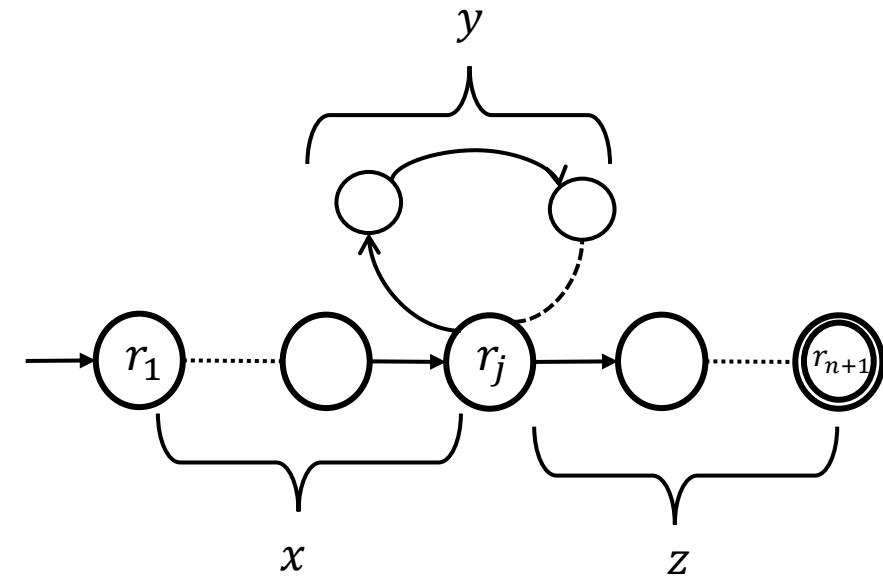
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .

So we can divide the  $s$  into three parts,  $x = s_1 \dots s_{j-1}$ ,  $y = s_j \dots s_{l-1}$ ,  $z = s_l \dots s_n$ . For a run on  $M$ , due to  $s$

- the  $x$  part takes us from  $r_1$  to  $r_j$
- the  $y$  part belongs to the loop part (we go from  $r_j$  to  $r_j$ )
- $z$  takes us from  $r_j$  to  $r_{n+1}$ , which is a final state if  $s \in L$ .



- We can traverse the loop bit any number of times and so  $\forall i \geq 0, xy^i z \in L$ .
- Also, as  $j \neq l$ ,  $|y| \geq 1$
- While reading the input, within the first  $p$  symbols of  $s$ , some state must be repeated.

# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

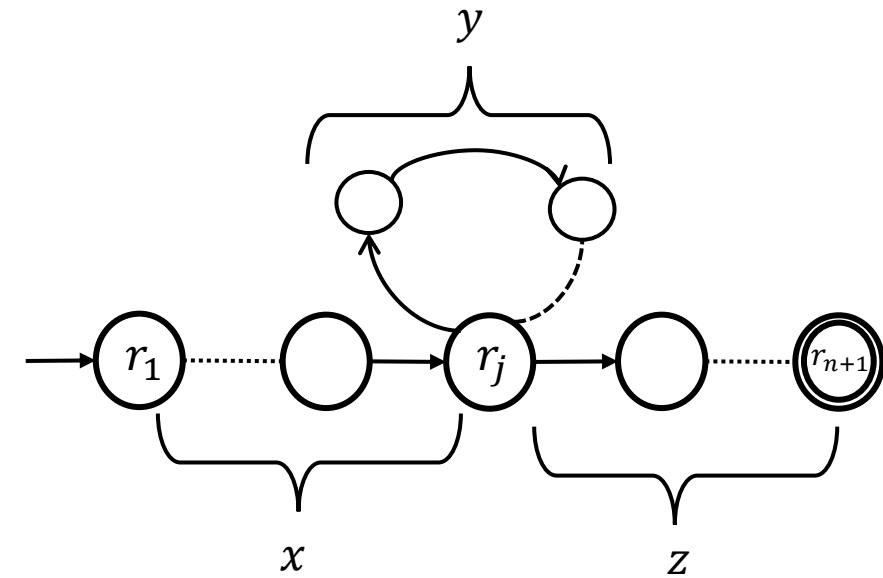
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .

So we can divide the  $s$  into three parts,  $x = s_1 \dots s_{j-1}$ ,  $y = s_j \dots s_{l-1}$ ,  $z = s_l \dots s_n$ . For a run on  $M$ , due to  $s$

- the  $x$  part takes us from  $r_1$  to  $r_j$
- the  $y$  part belongs to the loop part (we go from  $r_j$  to  $r_j$ )
- $z$  takes us from  $r_j$  to  $r_{n+1}$ , which is a final state if  $s \in L$ .



- We can traverse the loop bit any number of times and so  $\forall i \geq 0, xy^i z \in L$ .
- Also, as  $j \neq l$ ,  $|y| \geq 1$ , and
- The DFA reads  $|xy|$  by then and so  $|xy| \leq p$ .

# Pumping Lemma

In order to prove that a language is non-regular,

- Assume that it is regular and obtain a contradiction.
- Find a string in the language of length  $\geq p$  (pumping length) that cannot be pumped.

Examples of languages that are NOT regular:

- $\{0^p \mid p \text{ is prime}\}$
- $\{0^n 1^n \mid n \geq 0\}$
- $\{\omega \mid \omega \text{ has equal number of 0's and 1's}\}$
- $\{\omega \mid \omega \text{ is palindrome}\}$

⋮

⋮

Refer to Sipser (or some other textbook) for proofs using Pumping lemma

# The story so far...

- We have built devices (DFAs/NFAs) that decides some languages.
- Regular languages are precisely the ones that are accepted by finite automata.
- For any  $L \in RL$ , we have DFA/NFA  $M$  such that  $L(M) = L$ .
- Regular expressions describe regular languages algebraically.
- There are languages that are not regular.

DFA  $\equiv$  NFA  $\equiv$  Regular Expressions

Next up:

- How do we generate the strings in a language?
- **Syntax:** What are the set of legal strings in a language?
- Think of the English language (Rules of grammar)

**Thank You!**

# CS 302.1 - Automata Theory

## Lecture 04

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad



INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

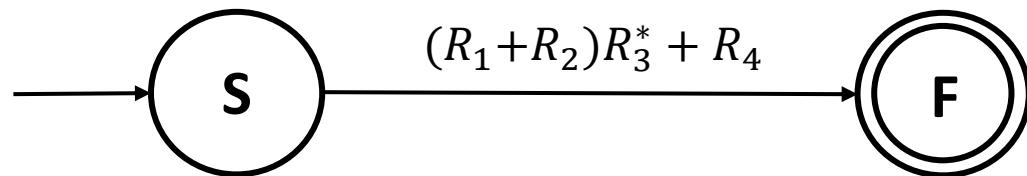
H Y D E R A B A D

# Quick Recap

- RL can also be derived from first principles.
- Regular expressions provide an elegant algebraic framework to represent regular languages.
- We can construct NFAs given a Regular Expression.

A Generalized NFA (GNFA) is similar to an NFA except that transitions contain regular expressions.

Given a DFA  $M$ , we obtain the regular expression corresponding to  $L(M)$  by constructing a 2-state GNFA via a recursive algorithm.



DFA, NFA, Regular Expressions  
have equal power and all of them  
correspond to Regular Languages

# Pumping Lemma

How do we prove that certain languages are non-regular? We start with an example

Let  $\Sigma = \{0,1\}$ . Consider the language  $L = \{0^n 1^n \mid n \geq 0\}$  and the following conversation between Karl and Mil.

**Mil:** I have a DFA for  $L$ .

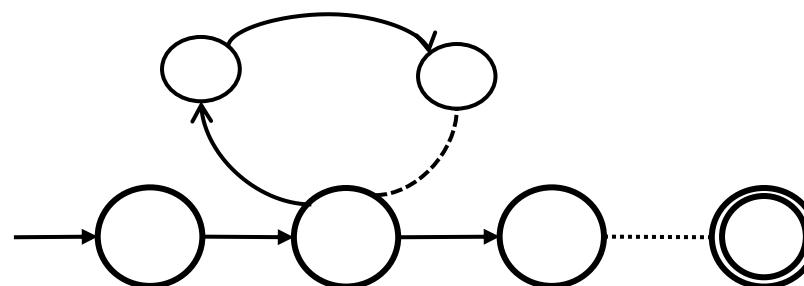
**Karl:** How many states are there?

**Mil:**  $n$ -states (say  $n = 10$ )

**Karl:** Then  $0^{10} 1^{10}$  must be accepted. By the **pigeonhole principle**, while reading the first ( $n = 10$ ) symbols, some states need to be revisited. Otherwise  $n + 1 = 11$  states would have been present. Hence some loop must be present. How many states are there in the loop?

**Mil:**  $t$ -states (say  $t = 3$ ).

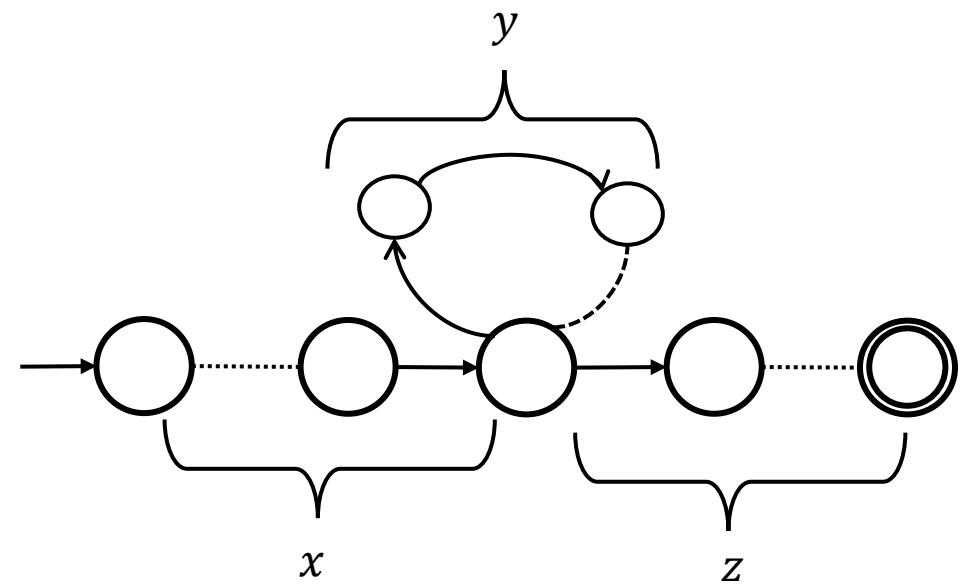
**Karl:** If your DFA accepts  $0^n 1^n$ , it must also accept  $0^{n+t} 1^n$ . This is because, if we take the loop one extra time, we read  $t$  more 0's.



**Contradiction as  $0^{n+t} 1^n \notin L$ . So Mil, you never had a DFA for  $L$  and in fact,  $L$  is not regular.**

# Pumping Lemma

If  $L$  is a regular language, all strings in the language, larger than a certain length (pumping length) , can be *pumped*: the string contains a certain section that can be repeated *any number of times* and the resulting string still  $\in L$ .

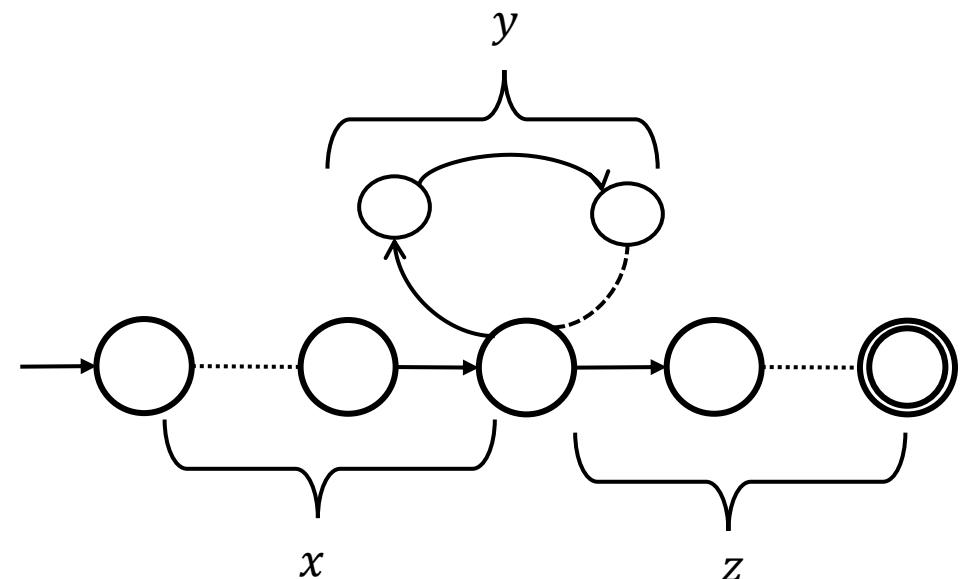


# Pumping Lemma

If  $L$  is a regular language, all strings in the language, larger than a certain length (pumping length) , can be *pumped*: the string contains a certain section that can be repeated *any number of times* and the resulting string still  $\in L$ .

**(Pumping Lemma)** If  $L$  is a regular language, then there exists a number  $p$  (the pumping length) where for all  $s \in L$  of length at least  $p$ , there exists  $x, y, z$  such that  $s = xyz$ , such that

1.  $|xy| \leq p$ .
2.  $|y| \geq 1$
3.  $\forall i \geq 0, xy^i z \in L$ .



# Pumping Lemma

If  $L$  is a regular language, all strings in the language, larger than a certain length (pumping length) , can be *pumped*: the string contains a certain section that can be repeated *any number of times* and the resulting string still  $\in L$ .

**(Pumping Lemma)** If  $L$  is a regular language, then there exists a number  $p$  (the pumping length) where for all  $s \in L$  of length at least  $p$ , there exists  $x, y, z$  such that  $s = xyz$ , such that

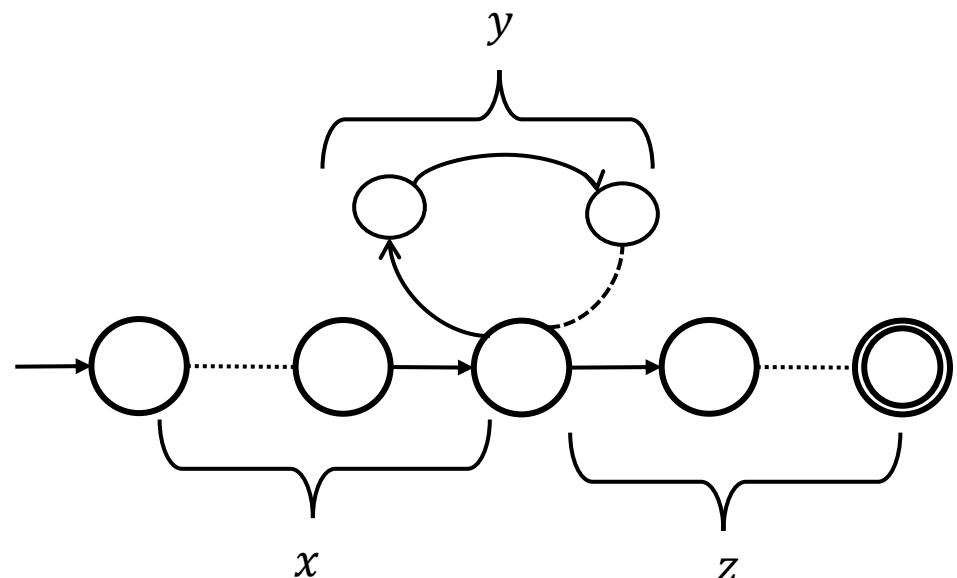
1.  $|xy| \leq p$ .
2.  $|y| \geq 1$
3.  $\forall i \geq 0, xy^i z \in L$ .

**Note:**  $(A \Rightarrow B) \equiv (\neg B) \Rightarrow (\neg A)$

If  $L$  is regular then, pumping property is satisfied

$\equiv$

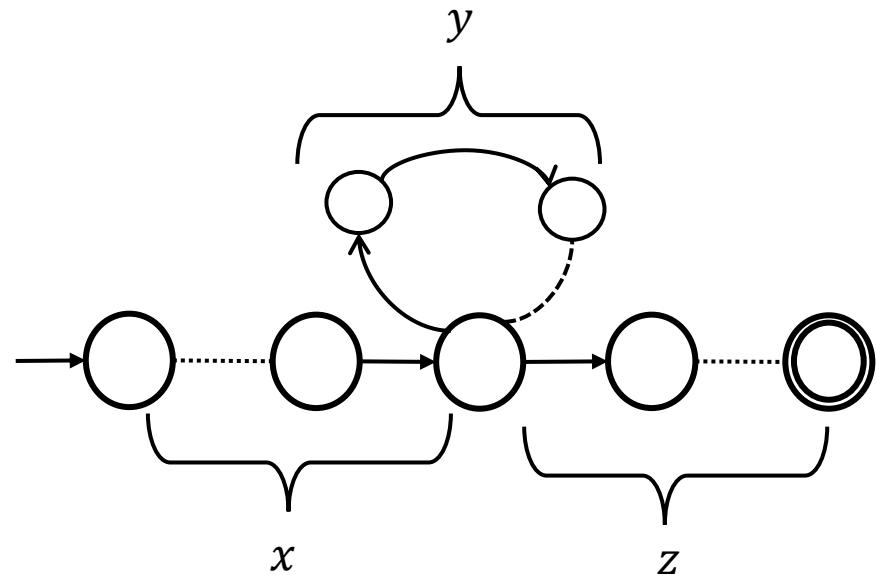
If pumping property is NOT satisfied, then  $L$  is NOT regular.



# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.



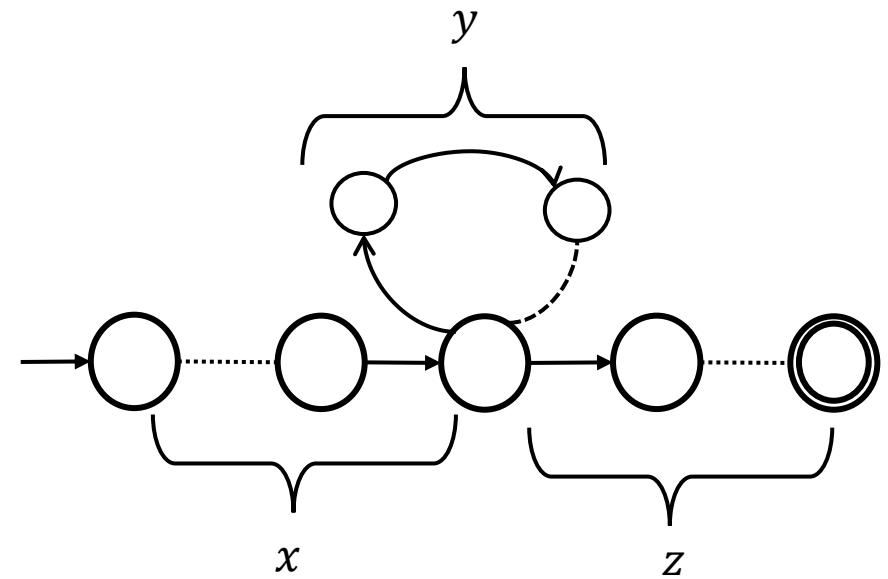
# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .



# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

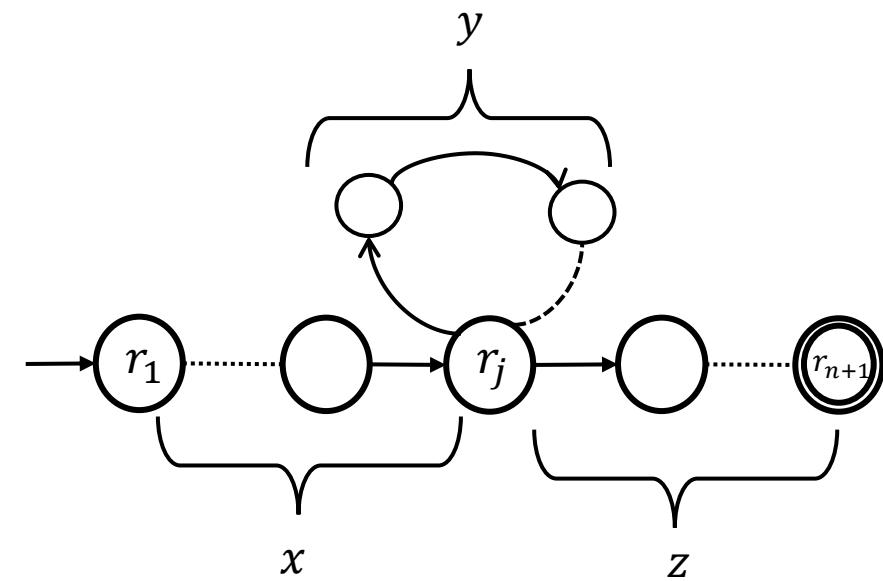
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .

So we can divide the  $s$  into three parts,  $x = s_1 \dots s_{j-1}$ ,  $y = s_j \dots s_{l-1}$ ,  $z = s_l \dots s_n$ . For a run on  $M$ , due to  $s$

- the  $x$  part takes us from  $r_1$  to  $r_j$
- the  $y$  part belongs to the loop part (we go from  $r_j$  to  $r_j$ )
- $z$  takes us from  $r_j$  to  $r_{n+1}$ , which is a final state if  $s \in L$ .



# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

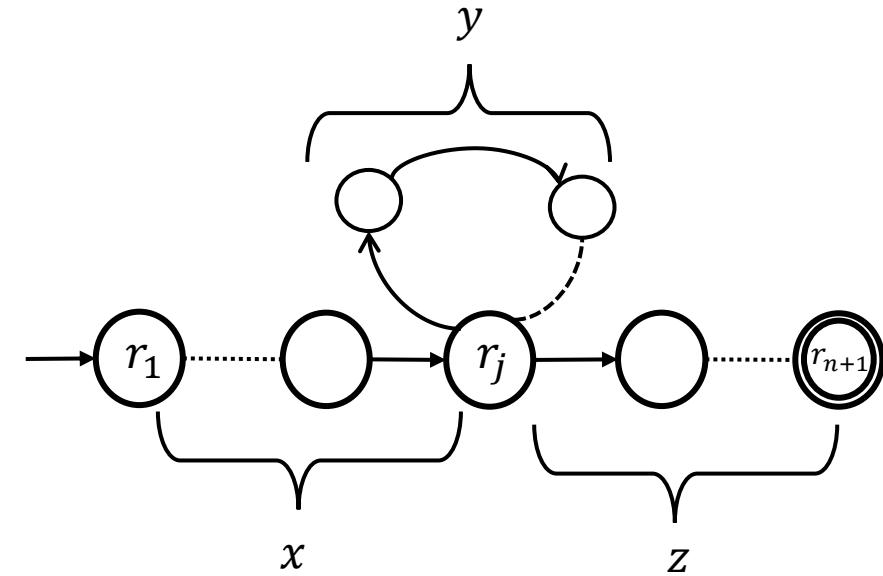
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .

So we can divide the  $s$  into three parts,  $x = s_1 \dots s_{j-1}$ ,  $y = s_j \dots s_{l-1}$ ,  $z = s_l \dots s_n$ . For a run on  $M$ , due to  $s$

- the  $x$  part takes us from  $r_1$  to  $r_j$
- the  $y$  part belongs to the loop part (we go from  $r_j$  to  $r_j$ )
- $z$  takes us from  $r_j$  to  $r_{n+1}$ , which is a final state if  $s \in L$ .



- We can traverse the loop bit any number of times and so  $\forall i \geq 0, xy^i z \in L$ .

# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

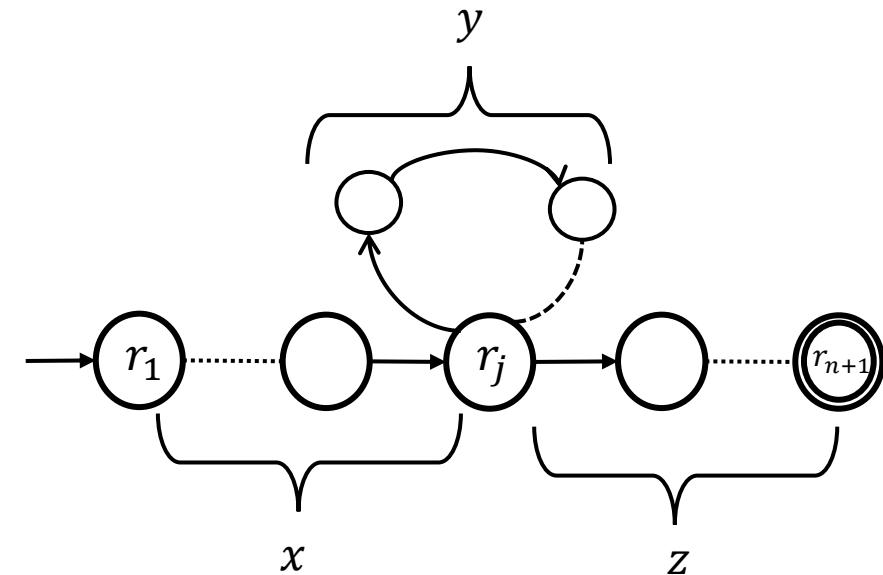
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .

So we can divide the  $s$  into three parts,  $x = s_1 \dots s_{j-1}$ ,  $y = s_j \dots s_{l-1}$ ,  $z = s_l \dots s_n$ . For a run on  $M$ , due to  $s$

- the  $x$  part takes us from  $r_1$  to  $r_j$
- the  $y$  part belongs to the loop part (we go from  $r_j$  to  $r_j$ )
- $z$  takes us from  $r_j$  to  $r_{n+1}$ , which is a final state if  $s \in L$ .



- We can traverse the loop bit any number of times and so  $\forall i \geq 0, xy^i z \in L$ .
- Also, as  $j \neq l$ ,  $|y| \geq 1$
- While reading the input, within the first  $p$  symbols of  $s$ , some state must be repeated.

# Pumping Lemma

**Proof sketch:** Suppose that we have a DFA  $M$  of  $p$  states. Then any run in the DFA corresponding to strings of length at least  $p$ , some states are repeated.

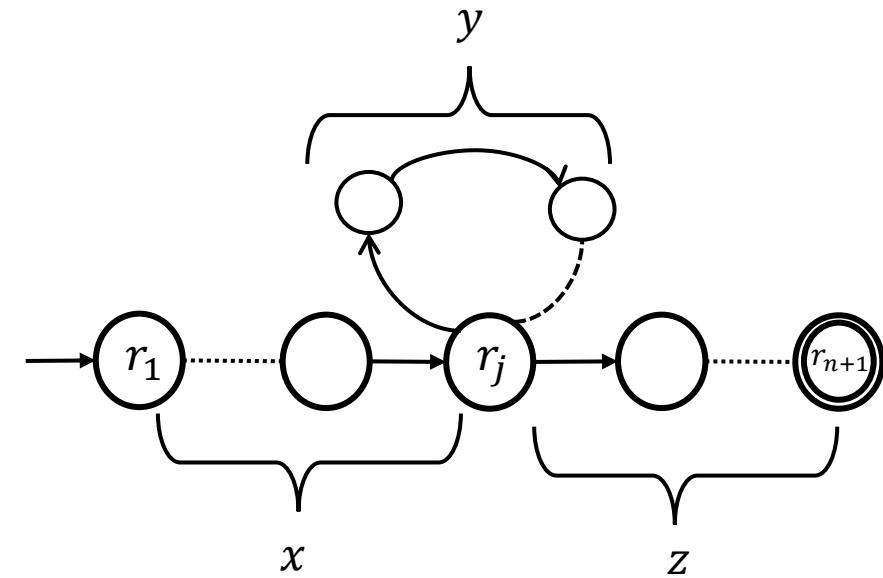
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

Suppose  $s = s_1s_2 \dots s_n$  be any such string of length  $n$  ( $\geq p$ ) and suppose  $r_1r_2 \dots r_{n+1}$  be the sequence of states encountered, while implementing a run of  $s$  in  $M$ .

As  $n + 1 \geq p + 1$ , in the above sequence at least two states must be repeated. Let them be  $r_j$  and  $r_l$ , i.e.,  $r_j = r_l$ , but  $j \neq l$ .

So we can divide the  $s$  into three parts,  $x = s_1 \dots s_{j-1}$ ,  $y = s_j \dots s_{l-1}$ ,  $z = s_l \dots s_n$ . For a run on  $M$ , due to  $s$

- the  $x$  part takes us from  $r_1$  to  $r_j$
- the  $y$  part belongs to the loop part (we go from  $r_j$  to  $r_j$ )
- $z$  takes us from  $r_j$  to  $r_{n+1}$ , which is a final state if  $s \in L$ .



- We can traverse the loop bit any number of times and so  $\forall i \geq 0, xy^i z \in L$ .
- Also, as  $j \neq l$ ,  $|y| \geq 1$ , and
- The DFA reads  $|xy|$  by then and so  $|xy| \leq p$ .

# Pumping Lemma

In order to prove that a language is non-regular,

- Assume that it is regular and obtain a contradiction.
- Find a string in the language of length  $\geq p$  (pumping length) that cannot be pumped.

Examples of languages that are NOT regular:

- $\{0^p \mid p \text{ is prime}\}$
- $\{0^n 1^n \mid n \geq 0\}$
- $\{\omega \mid \omega \text{ has equal number of 0's and 1's}\}$
- $\{\omega \mid \omega \text{ is palindrome}\}$

⋮

⋮

Refer to Sipser (or some other textbook) for proofs using Pumping lemma

# The story so far...

- We have built devices (DFAs/NFAs) that decides some languages.
- Regular languages are precisely the ones that are accepted by finite automata.
- For any  $L \in RL$ , we have DFA/NFA  $M$  such that  $L(M) = L$ .
- Regular expressions describe regular languages algebraically.
- There are languages that are not regular.

**DFA  $\equiv$  NFA  $\equiv$  Regular Expressions**

Next up:

- How do we generate the strings in a language?
- **Syntax:** What are the set of legal strings in a language?
- Think of the English language (Rules of grammar)

# Grammars

- **Grammars** provide a way to generate strings belonging to a language. The set of all strings generated by the grammar is the *language* of the grammar.
- **Grammars generate languages:** Grammars consist of a set of **rules** that allow you to construct strings of the language.
- For some classes of grammars, one can build automata that recognizes the language generated by the grammar.
- In fact, these concepts have been fundamental in attempts to formalize natural languages.

# Grammars

- **Grammars** provide a way to generate strings belonging to a language. The set of all strings generated by the grammar is the *language* of the grammar.
- **Grammars generate languages:** Grammars consist of a set of **rules** that allow you to construct strings of the language.
- For some classes of grammars, one can build automata that recognizes the language generated by the grammar.
- Consider these rules

*Sentence* → *Subject Verb Object*

*Subject* → *Noun. phrase*

*Object* → *Noun. phrase*

*Noun. phrase* → *Article Noun|Noun*

*Article* → **the**

*Noun* → **boy|girl|soccer|poetry**

*Verb* → **loves|plays**

# Grammars

- **Grammars** provide a way to generate strings belonging to a language. The set of all strings generated by the grammar is the *language* of the grammar.
- **Grammars generate languages:** Grammars consist of a set of **rules** that allow you to construct strings of the language.
- For some classes of grammars, one can build automata that recognizes the language generated by the grammar.
- Consider these rules

*Sentence* → *Subject Verb Object*

*Subject* → *Noun. phrase*

*Object* → *Noun. phrase*

*Noun. phrase* → *Article Noun|Noun*

*Article* → **the**

*Noun* → **boy|girl|soccer|poetry**

*Verb* → **loves|plays**

**Terminals** consist of strings over the alphabet corresponding to the language that the Grammar generates

**Variables:** {*Sentence*, *Subject*, *Verb*, *Object*, *Noun*, *Noun. phrase*, *Article*}, **Terminals:** {*the*, *girl*, *loves*, *plays*, *soccer*, *poetry*}

**Start Variable:** *Sentence*

# Grammars

- **Grammars** provide a way to generate strings belonging to a language. The set of all strings generated by the grammar is the *language* of the grammar.
- **Grammars generate languages:** Grammars consist of a set of **rules** that allow you to construct strings of the language.
- For some classes of grammars, one can build automata that recognizes the language generated by the grammar.
- Consider these rules

*Sentence* → *Subject Verb Object*

*Subject* → *Noun. phrase*

*Object* → *Noun. phrase*

*Noun. phrase* → *Article Noun|Noun*

*Article* → **the**

*Noun* → **boy|girl|soccer|poetry**

*Verb* → **loves|plays**

The sentence “**the girl plays soccer**” can be derived from this set of rules.

**Variables:** {*Sentence*, *Subject*, *Verb*, *Object*, *Noun*, *Noun. phrase*, *Article*}, **Terminals:** {*the*, *girl*, *loves*, *plays*, *soccer*, *poetry*}

**Start Variable:** *Sentence*

# Grammars

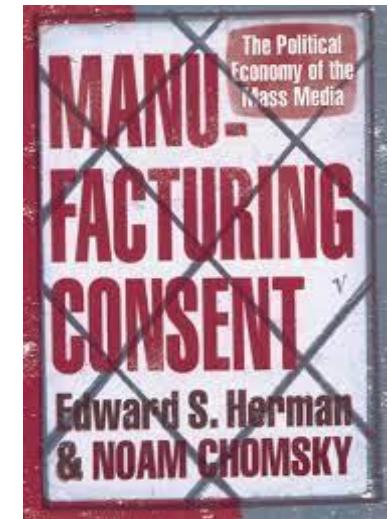
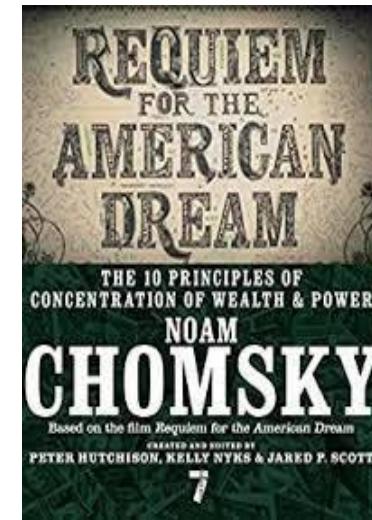
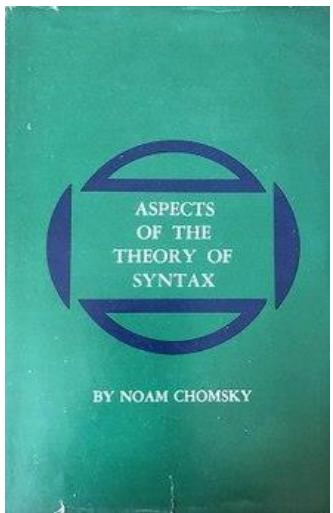
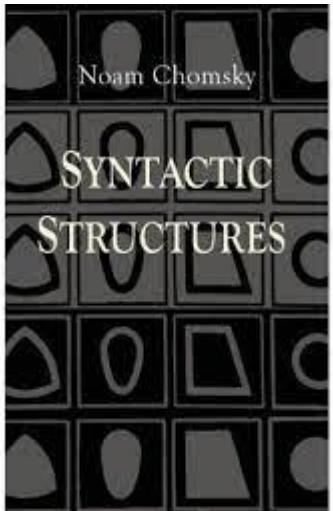
- **Grammars** provide a way to generate strings belonging to a language. The set of all strings generated by the grammar is the *language* of the grammar.
- **Grammars generate languages:** Grammars consist of a set of **rules** that allow you to construct strings of the language.
- For some classes of grammars, one can build automata that recognizes the language generated by the grammar.
- Consider these rules

*Sentence* → Subject Verb Object  
*Subject* → Noun. phrase  
*Object* → Noun. phrase  
*Noun. phrase* → Article Noun|Noun  
*Article* → **the**  
*Noun* → boy|girl|soccer|poetry  
*Verb* → loves|plays

*Sentence* → Subject Verb Object  
→ Noun. phrase Verb Object  
→ Article Noun Verb Object  
→ **the** Noun Verb Object  
→ **the girl** Verb Object  
→ **the girl plays** Object  
→ **the girl plays** Noun. phrase  
→ **the girl plays** Noun  
→ **the girl plays soccer**

**Variables:** {*Sentence*, *Subject*, *Verb*, *Object*, *Noun*, *Noun. phrase*, *Article*}, **Terminals:** {*The*, *girl*, *loves*, *plays*, *soccer*, *poetry*}  
**Start Variable:** *Sentence*

# Grammars



Noam Chomsky

- Noam Chomsky did pioneering work on linguistics and formalized many of these concepts.
- Also made great contributions to political economy and has been a champion of anti-imperialist, anti-capitalist, social justice struggles across the globe.

# Grammars

**(Grammar)** Formally, a *Grammar*  $G$  is a 5-tuple  $(V, \Sigma, P, S)$  such that

- $V$  is the set of **Variables**
- $\Sigma$  is the set of **Terminals** (disjoint from  $V$ )
- $P$  is the set of production **Rules**      [  $(V \cup \Sigma)^* V (V \cup \Sigma)^* \rightarrow (V \cup \Sigma)^*$  ]  
•  $S$  is the **Start Variable**                [ The variable in the LHS of the first rule is generally the start variable ]

Eg: Consider the grammar  $G$

$$X \rightarrow 1X$$

$$X \rightarrow 0Y$$

$$Y \rightarrow 0X$$

$$Y \rightarrow 1Y$$

$$Y \rightarrow \epsilon$$

**X is the start variable of the Grammar.** Variables:  $\{X, Y\}$ , Terminals:  $\{\epsilon, 0, 1\}$

# Grammars

**(Grammar)** Formally, a *Grammar*  $G$  is a 5-tuple  $(V, \Sigma, P, S)$  such that

- $V$  is the set of **Variables**
- $\Sigma$  is the set of **Terminals** (disjoint from  $V$ )
- $P$  is the set of production **Rules** [  $(V \cup \Sigma)^* V (V \cup \Sigma)^* \rightarrow (V \cup \Sigma)^*$  ]  
[ The variable in the LHS of the first rule is generally the start variable ]
- $S$  is the **Start Variable**

**Grammars can be used to derive strings.**

The sequence of **substitutions** (using the rules of  $G$ ) required to obtain a certain string is called a **derivation**.

- Begin the **derivation** from the **Start variable**.
- Replace any variable according to a rule. Repeat until only terminals remain.
- The generated string is **derived by the grammar**.

Eg: Consider the grammar  $G$

$$X \rightarrow 1X$$

$$X \rightarrow 0Y$$

$$Y \rightarrow 1Y$$

$$Y \rightarrow 0X$$

$$Y \rightarrow \epsilon$$

$X$ : Start Variable

$\{X, Y\}$ : Variables

$\{\epsilon, 0, 1\}$ : Terminals

The following is a derivation

$$X \rightarrow 1X \rightarrow 11X \rightarrow 110Y \rightarrow 1101Y \rightarrow \mathbf{1101}$$

# Grammars

**(Grammar)** Formally, a *Grammar*  $G$  is a 4-tuple  $(V, \Sigma, P, S)$  such that

- $V$  is the set of **Variables**
- $\Sigma$  is the set of **Terminals**
- $P$  is the set of production **Rules**
- $S$  is the **Start Variable**

$$[ (V \cup T)^* V (V \cup T)^* \rightarrow (V \cup T)^* ]$$

[ The variable in the LHS of the first rule is generally the start variable ]

- To show that a string  $w \in L(G)$ , we show that there exists a **derivation ending up in  $w$** . The fact that  $w$  can be derived using the rules of  $G$ , is expressed as  $S \xrightarrow{*} w$ .
- The **language of the grammar**,  $L(G)$  is  $\{w \in \Sigma^* | S \xrightarrow{*} w\}$

# Grammars

**(Grammar)** Formally, a *Grammar*  $G$  is a 5-tuple  $(V, \Sigma, P, S)$  such that

- $V$  is the set of **Variables**
- $\Sigma$  is the set of **Terminals**
- $P$  is the set of production **Rules**
- $S$  is the **Start Variable**

$$[ (V \cup T)^* V (V \cup T)^* \rightarrow (V \cup T)^* ]$$

[ The variable in the LHS of the first rule is generally the start variable ]

- To show that a string  $w \in L(G)$ , we show that there exists a **derivation ending up in  $w$** . The fact that  $w$  can be derived using the rules of  $G$ , is expressed as  $S \xrightarrow{*} w$ .
- The **language of the grammar**,  $L(G)$  is  $\{w \in \Sigma^* | S \xrightarrow{*} w\}$

Eg: Consider the grammar  $G$

$$X \rightarrow 1X$$

$$X \rightarrow 0Y$$

$$Y \rightarrow 1Y$$

$$Y \rightarrow 0X$$

$$Y \rightarrow \epsilon$$

The string **1101**  $\in L(G)$  because there exists the following derivation

$$X \rightarrow 1X \rightarrow 11X \rightarrow 110Y \rightarrow 1101Y \rightarrow 1101$$

# Grammars for Regular Languages

**Regular grammar:** If the *rules* of the underlying grammar  $G$  are of the form

$$Var \rightarrow Ter\ Var$$

$$Var \rightarrow Ter$$

$$Var \rightarrow \epsilon$$

then the language of the grammar is **regular**. Also known as **Right-linear grammar** (all variables are to the right of terminals in the RHS).

## Right linear Grammar to DFA

Eg: Consider the grammar  $G$

$$X \rightarrow 1X$$

$$X \rightarrow 0Y$$

$$Y \rightarrow 1Y$$

$$Y \rightarrow 0X$$

$$Y \rightarrow \epsilon \text{ (indicates that } Y \text{ is the final state)}$$

# Grammars for Regular Languages

**Regular grammar:** If the *rules* of the underlying grammar  $G$  are of the form

$$Var \rightarrow Ter\ Var$$

$$Var \rightarrow Ter$$

$$Var \rightarrow \epsilon$$

then the language of the grammar is **regular**. Also known as **Right-linear grammar** (all variables are to the right of terminals in the RHS).

## Right linear Grammar to DFA

Eg: Consider the grammar  $G$

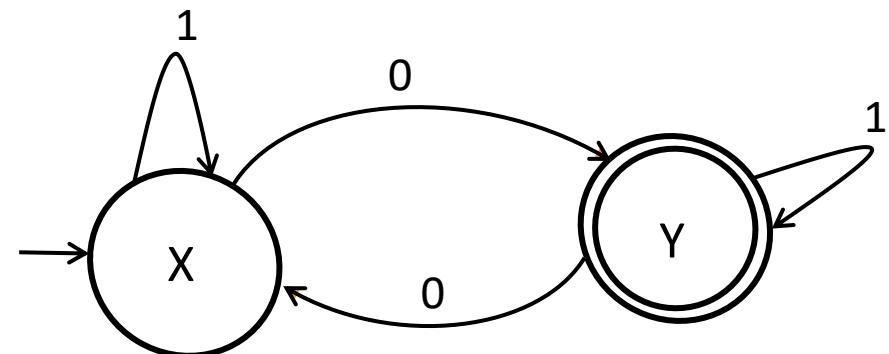
$$X \rightarrow 1X$$

$$X \rightarrow 0Y$$

$$Y \rightarrow 1Y$$

$$Y \rightarrow 0X$$

$Y \rightarrow \epsilon$  (indicates that  $Y$  is the final state)



# Grammars for Regular Languages

**Regular grammar:** If the *rules* of the underlying grammar  $G$  are of the form

$$Var \rightarrow Ter\ Var$$

$$Var \rightarrow Ter$$

$$Var \rightarrow \epsilon$$

then the language of the grammar is **regular**. Also known as **Right-linear grammar** (all variables are to the right of terminals in the RHS).

## Right linear Grammar to DFA

Eg: Consider the grammar  $G$

$$X \rightarrow 1X$$

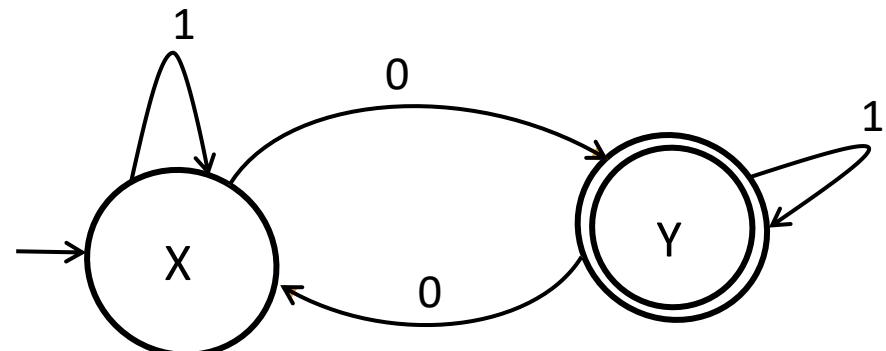
$$X \rightarrow 0Y$$

$$Y \rightarrow 1Y$$

$$Y \rightarrow 0X$$

$Y \rightarrow \epsilon$  (indicates that  $Y$  is the final state)

A **run** in a DFA model is analogous to a **derivation** in a linear grammar.



For the string **1101**:

**Derivation:**  $X \rightarrow 1X \rightarrow 11X \rightarrow 110Y \rightarrow 1101Y \rightarrow 1101$ . So  $1101 \in L(G)$

**Run:**  $X \xrightarrow{1} X \xrightarrow{1} X \xrightarrow{0} Y \xrightarrow{1} Y$  (Accepting Run and so  $1101 \in L(M)$ ).

# Grammars for Regular Languages

**Regular grammar:** If the *rules* of the underlying grammar  $G$  are of the form

$$Var \rightarrow Ter\ Var$$

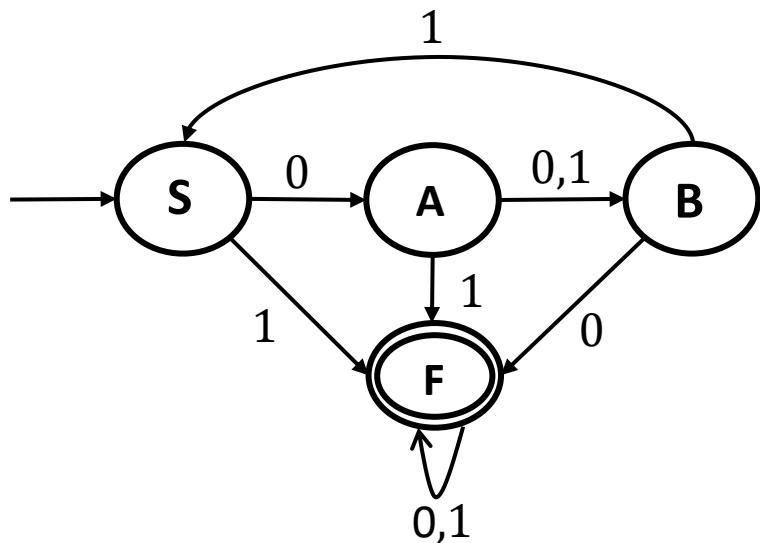
$$Var \rightarrow Ter$$

$$Var \rightarrow \epsilon$$

then the language of the grammar is **regular**. Also known as **Right-linear grammar** (all variables are to the right of terminals in the RHS).

## DFA to Right linear Grammar

Consider the following DFA  $M$



The right-linear grammar  $G$  for  $M$

$$S \rightarrow 0A$$

$$A \rightarrow 01B$$

$$B \rightarrow 1S$$

$$F \rightarrow 01F$$

$$A \rightarrow 1F$$

$$B \rightarrow 0F$$

$$S \rightarrow 1F$$

$$F \rightarrow \epsilon$$

# Grammars for Regular Languages

**Right-linear grammar  $\equiv$  DFA  $\equiv$  NFA  $\equiv$  Regular Expressions**

**Left linear grammar:** If the *rules* of the underlying grammar  $G$  are of the form

$$Var \rightarrow Var\ Ter$$

$$Var \rightarrow Ter$$

$$Var \rightarrow \epsilon$$

then such a grammar is called **Left-linear** (all Variables are to the left of terminals in the RHS).

**Right linear grammars are equivalent to Left-linear grammar** (We won't be proving it here )

# Grammars for Regular Languages

**Right-linear grammar  $\equiv$  DFA  $\equiv$  NFA  $\equiv$  Regular Expressions**

**Left linear grammar:** If the *rules* of the underlying grammar  $G$  are of the form

$$Var \rightarrow Var\ Ter$$

$$Var \rightarrow Ter$$

$$Var \rightarrow \epsilon$$

then such a grammar is called **Left-linear** (all Variables are to the left of terminals in the RHS).

**Right linear grammars are equivalent to Left-linear grammar** (We won't be proving it here)

**Right-linear grammars and Left-linear grammars generate Regular Languages.**

Note that mixing left-linear grammars and right-linear grammars in the same set of rules **won't generate regular languages**. (e.g:  $S \rightarrow aX, X \rightarrow Sb, S \rightarrow \epsilon$ )

**Left-linear grammar  $\equiv$  Right-linear grammar  $\equiv$  DFA  $\equiv$  NFA  $\equiv$  Regular Expressions**

# Context free Grammars

**(Grammar)** Formally, a *Grammar*  $G$  is a 4-tuple  $(V, \Sigma, P, S)$  such that

- $V$  is the set of **Variables**
- $\Sigma$  is the set of **Terminals**
- $P$  is the set of production **Rules**
- $S$  is the **Start Variable**

$$[ (V \cup T)^* V (V \cup T)^* \rightarrow (V \cup T)^* ]$$

[ The variable in the LHS of the first rule is generally the start variable ]

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (V \cup T)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammar is called a ***context-free language***.

Immediately we find that the *rules* are less restrictive than left-linear grammars and right-linear grammars. Context free grammars allow

$$Var \rightarrow Anything$$

$$Var \rightarrow String\ of\ Variables | String\ of\ Terminals | Strings\ of\ Variables\ and\ Terminals | \epsilon$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammars is called a *context-free language*.

Immediately we find that the *rules* are less restrictive than left-linear grammars and right-linear grammars. Context free grammars allow

$$Var \rightarrow Anything$$

$$Var \rightarrow String\ of\ Variables | String\ of\ Terminals | Strings\ of\ Variables\ and\ Terminals | \epsilon$$

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages  $\subset$  Context Free Languages.**

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammars is called a ***context-free language***.

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages**  $\subset$  **Context Free Languages**.

Consider the Grammar  $G$  with the following rules:

$$\begin{aligned} S &\rightarrow 0S1 \\ S &\rightarrow \epsilon \end{aligned}$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammars is called a ***context-free language***.

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages**  $\subset$  **Context Free Languages**.

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|\epsilon$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammars is called a ***context-free language***.

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages**  $\subset$  **Context Free Languages**.

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|\epsilon$$

What is the language generated by this grammar?

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammars is called a ***context-free language***.

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages  $\subset$  Context Free Languages.**

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|\epsilon$$

Strings that can be derived from  $G$ :

$$S \rightarrow \epsilon$$

What is the language generated by this grammar?

$$\{\epsilon\}$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammars is called a ***context-free language***.

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages  $\subset$  Context Free Languages.**

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1 | \epsilon$$

Strings that can be derived from  $G$ :

$$S \rightarrow 0S1 \rightarrow 01$$

What is the language generated by this grammar?

$$\{\epsilon, 01\}$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammars is called a ***context-free language***.

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages  $\subset$  Context Free Languages.**

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|\epsilon$$

Strings that can be derived from  $G$ :

$$S \rightarrow 0S1 \rightarrow 00S11 \rightarrow 0011$$

What is the language generated by this grammar?

$$\{\epsilon, 01, 0011\}$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammars is called a ***context-free language***.

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages  $\subset$  Context Free Languages.**

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|\epsilon$$

Strings that can be derived from  $G$ :

$$S \rightarrow 0S1 \rightarrow 00S11 \rightarrow 000S111 \rightarrow 000111$$

What is the language generated by this grammar?

$$\{\epsilon, 01, 0011, 000111\}$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

Any language generated by a context-free grammars is called a ***context-free language***.

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages  $\subset$  Context Free Languages.**

Consider the Grammar  $G$  with the following rules:

Strings that can be derived from  $G$ :

$$S \rightarrow 0S1|\epsilon$$

$$\{\epsilon, 01, 0011, 000111, 0^41^4, \dots\}$$

What is the language generated by this grammar?

$$L(G) = \{\omega | \omega = 0^n 1^n, n \geq 0\}$$

**So although  $L(G)$  is not regular, it is context-free.**

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

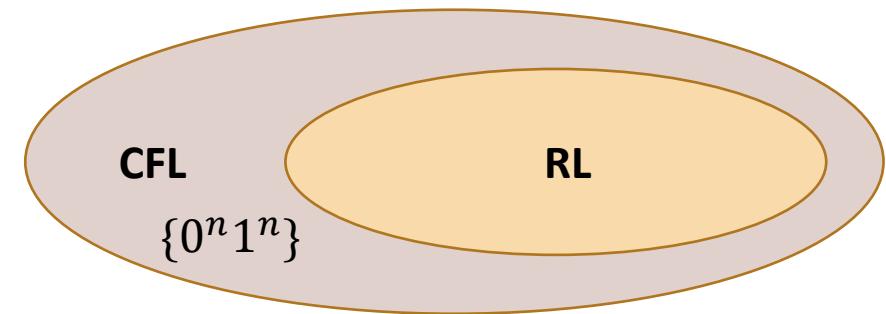
Any language generated by a context-free grammars is called a ***context-free language***.

- So Left linear grammars and Right linear grammars are also context-free grammars.
- **Regular languages  $\subset$  Context Free Languages.**

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|\epsilon$$

What is the language generated by this grammar?



$$L(G) = \{\omega | \omega = 0^n 1^n, n \geq 0\}$$

So although  $L(G)$  is not regular, it is context-free.

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (V \cup T)^*$$

then such a grammar is called **Context-Free**.

**Regular languages**  $\subset$  **Context Free Languages**.

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$S \rightarrow \epsilon$$

$$\{\epsilon\}$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (V \cup T)^*$$

then such a grammar is called **Context-Free**.

**Regular languages**  $\subset$  **Context Free Languages**.

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1 | SS | \epsilon$$

Strings that can be derived by  $G$ :

$$S \rightarrow 0S1 \rightarrow 00S11 \dots$$

$$\{\epsilon, 01, 0011, \dots 0^n1^n\}$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (V \cup T)^*$$

then such a grammar is called **Context-Free**.

**Regular languages**  $\subset$  **Context Free Languages**.

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$S \rightarrow \mathbf{0}S\mathbf{1} \rightarrow 0SS1 \rightarrow 00S1S1 \rightarrow 001S1 \rightarrow 0010S11 \rightarrow 001011$$

$$\{\epsilon, 01, 0011, \dots 0^n1^n, 001011, \dots\}$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (V \cup T)^*$$

then such a grammar is called **Context-Free**.

**Regular languages**  $\subset$  **Context Free Languages**.

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$S \rightarrow \mathbf{0}S\mathbf{1} \rightarrow 0SS1 \rightarrow 00S1S1 \rightarrow 001S1 \rightarrow 0010S11 \rightarrow 001011$$

$$\{\epsilon, 01, 0011, \dots 0^n1^n, 001011, \dots\}$$

Show that the string  $010101 \in L(G)$ .

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (V \cup T)^*$$

then such a grammar is called **Context-Free**.

**Regular languages**  $\subset$  **Context Free Languages**.

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$S \rightarrow SS \rightarrow SSS \rightarrow 0S1SS \rightarrow 0S10S1S \rightarrow 0S10S10S1 \rightarrow 010101$$

$$\{\epsilon, 01, 0011, \dots 0^n1^n, 001011, 010101, \dots\}$$

# Context free Grammars

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (V \cup T)^*$$

then such a grammar is called **Context-Free**.

**Regular languages**  $\subset$  **Context Free Languages**.

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$S \rightarrow SS \rightarrow SSS \rightarrow 0S1SS \rightarrow 0S10S1S \rightarrow 0S10S10S1 \rightarrow 010101$$

$$\{\epsilon, 01, 0011, \dots 0^n1^n, 001011, 010101, \dots\}$$

What is  $L(G)$ ?

# Context free Grammars

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$\{\epsilon, 01, 0011, \dots 0^n1^n, 001011, 010101, \dots\}$$

**What is  $L(G)$ ?**

# Context free Grammars

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$\{\epsilon, 01, 0011, \dots 0^n1^n, 001011, 010101, \dots\}$$

What is  $L(G)$ ?

You can see what the language is, if you replace **0** with ( and **1** with )

# Context free Grammars

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$$

**What is  $L(G)$ ?**

You can see what the language is, if you replace **0** with ( and **1** with )

Strings that can be derived by  $G$ :  $\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$

$$\{\epsilon, ()\}$$

# Context free Grammars

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$$

What is  $L(G)$ ?

You can see what the language is, if you replace **0** with ( and **1** with )

Strings that can be derived by  $G$ :  $\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$

$$\{\epsilon, ( ), (( )), \dots, \}$$

# Context free Grammars

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$$

What is  $L(G)$ ?

You can see what the language is, if you replace **0** with ( and **1** with )

Strings that can be derived by  $G$ :  $\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$

$$\{\epsilon, ( ), (( )), \dots, (((\dots)))\}$$

# Context free Grammars

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$$

What is  $L(G)$ ?

You can see what the language is, if you replace **0** with ( and **1** with )

Strings that can be derived by  $G$ :  $\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$

$$\{\epsilon, ( ), (( )), \dots, (((\dots)))\}, (( ))(( )), \dots\}$$

# Context free Grammars

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Strings that can be derived by  $G$ :

$$\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$$

**What is  $L(G)$ ?**

You can see what the language is, if you replace **0** with **(** and **1** with **)**

Strings that can be derived by  $G$ :  $\{\epsilon, 01, 0011, \dots, 0^n1^n, 001011, 010101, \dots\}$

$$\{\epsilon, ( ), (( )), \dots, (((\dots))), (( ))( )), 000, \dots\}$$

**So,  $L(G)$  is the language of all strings of properly nested parentheses.**

$$L(G) = \{\omega | \omega \text{ is a correctly nested parenthesis}\}$$

# Context free Grammars

## Constructing CFG corresponding to a Language.

There is no fixed recipe for doing this. Requires some level of creativity.

Some tips might come in handy:

- Check if the CFL is a union of simpler languages. If  $L(G) = L(G_1) \cup L(G_2)$  and  $G_1$  and  $G_2$  are known. If  $S_1$  is the start variable for  $G_1$  and  $S_2$  is the start variable for  $G_2$  then the rules of  $G$ :

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow \dots \dots \\ S_2 &\rightarrow \dots \dots \end{aligned}$$

# Context free Grammars

## Constructing CFG corresponding to a Language.

There is no fixed recipe for doing this. Requires some level of creativity.

Some tips might come in handy:

- Check if the CFL is a union of simpler languages. If  $L(G) = L(G_1) \cup L(G_2)$  and  $G_1$  and  $G_2$  are known. If  $S_1$  is the start variable for  $G_1$  and  $S_2$  is the start variable for  $G_2$  then the rules of  $G$ :

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow \dots \dots \\ S_2 &\rightarrow \dots \dots \end{aligned}$$

- Grammars with rules such as  $S \rightarrow aSb$  help generate strings where the corresponding machine would need unbounded memory to *remember* the number of  $a$ 's needed to verify that there are an equal number of  $b$ 's. This was not possible with regular expressions/linear grammars.

# Context free Grammars

**Constructing CFG corresponding to a Language.**

- Check if the CFL is a union of simpler languages.
- Grammars with rules such as  $S \rightarrow aSb$  help generate where the portions of  $a$  and  $b$  are equal.

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega \mid \omega \text{ has equal number of 0's and 1's}\}$

# Context free Grammars

## Constructing CFG corresponding to a Language.

- Check if the CFL is a union of simpler languages.
- Grammars with rules such as  $S \rightarrow aSb$  help generate where the portions of  $a$  and  $b$  are equal.

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega \mid \omega \text{ has equal number of 0's and 1's}\}$

- The first thing to notice is that  $L_1 = \{0^n 1^n, n \geq 0\} \subset L(G)$ . We know the grammar for this language.
- Any string  $\omega \in L_1$  has a series of 0's followed by an equal number of 1's.
- Again, consider  $L_2$  to comprise all strings that start with a series of 1's followed by an equal number of 0's, i.e.

$$L_2 = \{1^n 0^n, n \geq 0\}$$

- The grammar for  $L_2$  is similar to that of  $L_1$ : replace the 0's with 1's and vice versa. Importantly,  $L_2 = \{1^n 0^n, n \geq 0\} \subset L(G)$  also.
- Also,  $L_1 \cup L_2 \subset L(G)$

# Context free Grammars

## Constructing CFG corresponding to a Language.

- Check if the CFL is a union of simpler languages.
- Grammars with rules such as  $S \rightarrow aSb$  help generate where the portions of  $a$  and  $b$  are equal.

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega \mid \omega \text{ has equal number of 0's and 1's}\}$

- So  $L'(G') = \{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\} \subset L(G)$
- Grammar for  $L_1$ :  $S \rightarrow 0S1|\epsilon$
- Grammar for  $L_2$ :  $S \rightarrow 1S0|\epsilon$
- Grammar for  $L_1 \cup L_2$ :

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_20|\epsilon \end{aligned}$$

# Context free Grammars

## Constructing CFG corresponding to a Language.

- Check if the CFL is a union of simpler languages.
- Grammars with rules such as  $S \rightarrow aSb$  help generate where the portions of  $a$  and  $b$  are equal.

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega \mid \omega \text{ has equal number of 0's and 1's}\}$

- Grammar for  $L_1 \cup L_2$ :

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_20|\epsilon \end{aligned}$$

- **Is that all? Is  $L_1 \cup L_2 = L(G)$ ?**  $L_1 \cup L_2$  contains all strings that have equal number 0's followed by equal number of 1's or vice versa.

# Context free Grammars

## Constructing CFG corresponding to a Language.

- Check if the CFL is a union of simpler languages.
- Grammars with rules such as  $S \rightarrow aSb$  help generate where the portions of  $a$  and  $b$  are equal.

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega \mid \omega \text{ has equal number of 0's and 1's}\}$

- Grammar for  $L_1 \cup L_2$ :

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_20|\epsilon \end{aligned}$$

- **Is that all? Is  $L_1 \cup L_2 = L(G)$ ?**  $L_1 \cup L_2$  contains all strings that have equal number 0's followed by equal number of 1's or vice versa.
- What about strings such as  $s_1 = 0101 \dots$  and  $s_2 = 1010 \dots$ ? For this we need to be able to go from

$$0S_11 \rightarrow 0S_21 \rightarrow 01S_201 \rightarrow \dots$$

# Context free Grammars

**Constructing CFG corresponding to a Language.**

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega | \omega \text{ has equal number of 0's and 1's}\}$

- Grammar for  $L_1 \cup L_2$ :

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_20|\epsilon \end{aligned}$$

- What about strings such as  $s_1 = 0101\cdots$  and  $s_2 = 1010\cdots$ ? Add transitions  $S_1 \rightarrow S_2$  and  $S_2 \rightarrow S_1$ .

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_20|\epsilon \\ S_1 &\rightarrow S_2 \\ S_2 &\rightarrow S_1. \end{aligned}$$

# Context free Grammars

**Constructing CFG corresponding to a Language.**

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega | \omega \text{ has equal number of 0's and 1's}\}$

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_20|\epsilon \\ S_1 &\rightarrow S_2 \\ S_2 &\rightarrow S_1 \end{aligned}$$

- Can't we simplify this? We can replace  $S_1$  and  $S_2$  with a single Start variable as follows:  $S \rightarrow 0S1|1S0|\epsilon$
- What kind of strings does the grammar generate? Well if we use Rule  $S \rightarrow 0S1$ ,  $m$  times, we get to rules such as  $0^mS1^m$ .
- Now applying the rule  $S \rightarrow 1S0$ ,  $k$  times, we get  $0^m1^kS0^k1^m$ .
- So the strings we obtain are of the form:

$$\{0^{m_1}1^{n_1}0^{m_2}1^{n_2} \dots 0^{n_2}1^{m_2}0^{n_1}1^{m_1}\} \cup \{1^{m_1}0^{n_1}1^{m_2}0^{n_2} \dots 1^{n_2}0^{m_2}1^{n_1}0^{m_1}\} \in L(G)$$

# Context free Grammars

**Constructing CFG corresponding to a Language.**

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega | \omega \text{ has equal number of 0's and 1's}\}$

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_20|\epsilon \\ S_1 &\rightarrow S_2 \\ S_2 &\rightarrow S_1 \end{aligned}$$

- Simplified grammar:

$$S \rightarrow 0S1|1S0|\epsilon$$

# Context free Grammars

**Constructing CFG corresponding to a Language.**

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega | \omega \text{ has equal number of 0's and 1's}\}$

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_20|\epsilon \\ S_1 &\rightarrow S_2 \\ S_2 &\rightarrow S_1 \end{aligned}$$

- Simplified grammar:

$$S \rightarrow 0S1|1S0|\epsilon$$

- Is that all? What about strings such as **{0110, 00111100}**?
- More generally, what about strings that are a concatenation of  $L_1$  and  $L_2$ ?

# Context free Grammars

**Constructing CFG corresponding to a Language.**

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega \mid \omega \text{ has equal number of 0's and 1's}\}$

$$\begin{aligned} S &\rightarrow S_1 | S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_20|\epsilon \\ S_1 &\rightarrow S_2 \\ S_2 &\rightarrow S_1 \end{aligned}$$

- Simplified grammar:

$$S \rightarrow 0S1|1S0|\epsilon$$

- Is that all? What about strings such as  $\{0\mathbf{1}10, 0011\mathbf{1}100\}$ ?
- More generally, what about strings that are a concatenation of  $L_1$  and  $L_2$ ?
- Adding transitions like  $S \rightarrow S_1S_2$  incorporates this.

# Context free Grammars

**Constructing CFG corresponding to a Language.**

Example: Construct the grammar  $G$  such that  $L(G) = \{\omega \mid \omega \text{ has equal number of 0's and 1's}\}$

$$S \rightarrow S_1 | S_2 | S_1 S_2$$

$$S_1 \rightarrow 0S_11 | \epsilon$$

$$S_2 \rightarrow 1S_20 | \epsilon$$

$$S_1 \rightarrow S_2$$

$$S_2 \rightarrow S_1$$

- Simplify this further.

$$\mathbf{G: } S \rightarrow SS | 0S1 | 1S0 | \epsilon$$

**Thank You!**

# CS 302.1 - Automata Theory

Lecture 05

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad



INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

H Y D E R A B A D

# Quick Recap

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (V \cup T)^*$$

then such a grammar is called **Context-Free**.

- $\Sigma$  is the set of **Terminals**
- $P$  is the set of production **Rules**  $[ (V \cup T)^* V (V \cup T)^* \rightarrow (V \cup T)^* ]$
- $S$  is the **Start Variable** [ The variable in the LHS of the first rule is generally the start variable ]
- To show that a string  $w \in L(G)$ , we show that there exists a **derivation ending up in  $w$**  ( $S \xrightarrow{*} w$ ).
- The **language of the grammar,  $L(G)$**  is  $\{w \in \Sigma^* | S \xrightarrow{*} w\}$

**Right Linear grammar:** If the *rules* of the underlying grammar  $G$  are of the form

$$Var \rightarrow Ter \ Var$$

$$Var \rightarrow Ter$$

$$Var \rightarrow \epsilon$$

then it is **Right-linear grammar**.

**Left linear grammar:** If the *rules* of the underlying grammar  $G$  are of the form

$$Var \rightarrow \ Var \ Ter$$

$$Var \rightarrow Ter$$

$$Var \rightarrow \epsilon$$

then such a grammar is called **Left-linear grammar**.

**Left-linear grammar  $\equiv$  Right-linear grammar  $\equiv$  DFA  $\equiv$  NFA  $\equiv$  Regular Expressions**

# Quick Recap

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

- $\Sigma$  is the set of **Terminals**
  - $P$  is the set of production **Rules**  $[ (V \cup T)^* V (V \cup T)^* \rightarrow (V \cup T)^* ]$
  - $S$  is the **Start Variable** [ The variable in the LHS of the first rule is generally the start variable ]
- 
- To show that a string  $w \in L(G)$ , we show that there exists a **derivation ending up in  $w$**  ( $S \xrightarrow{*} w$ ).
  - The **language of the grammar,  $L(G)$**  is  $\{w \in \Sigma^* | S \xrightarrow{*} w\}$

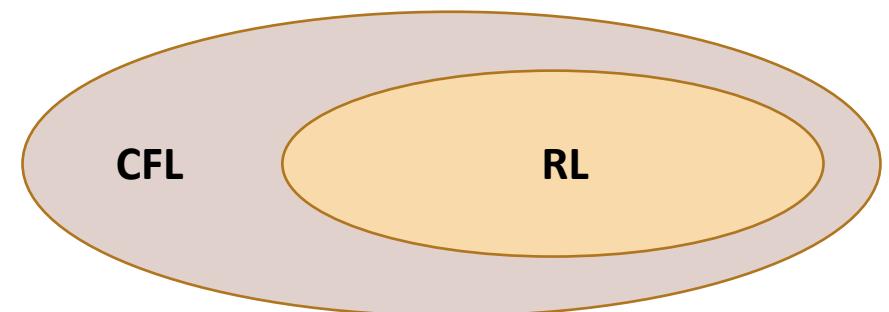
**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form

$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.

$$L(G) = \{\omega | \omega = 0^n 1^n, n \geq 0\}$$

So although  $L(G)$  is not regular, it is context-free.



# Parse trees for CFG

Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

**Parse trees:** These are ordered trees that provide alternative representations of the derivation of a grammar.

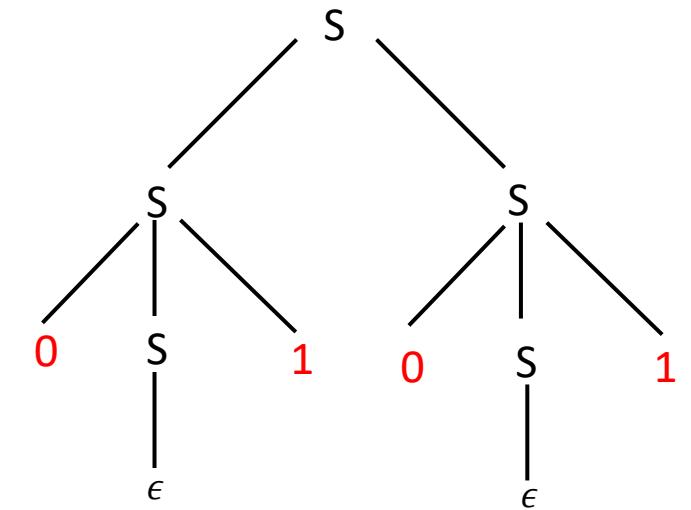
**Parsing** is a useful technique for compilers (Analysis of syntax eg: take sequence of tokens as input & output parse trees which provides structural representation of the input while checking for the correct syntax).

**Features:**

- The root node is the **Start variable**
- Branch out to nodes of the next level by following any of the rules of the grammar
- Stop when all the leaf nodes of the tree are terminals
- Read the terminals in the leaves from left to right.
- If  $w$  is the string obtained, then  $S \xrightarrow{*} w$  and  $w \in L(G)$

One derivation:

$$S \rightarrow SS \rightarrow \mathbf{0S1S} \rightarrow 0S1\mathbf{0S1} \rightarrow \mathbf{0101}$$



# Parse trees for CFG

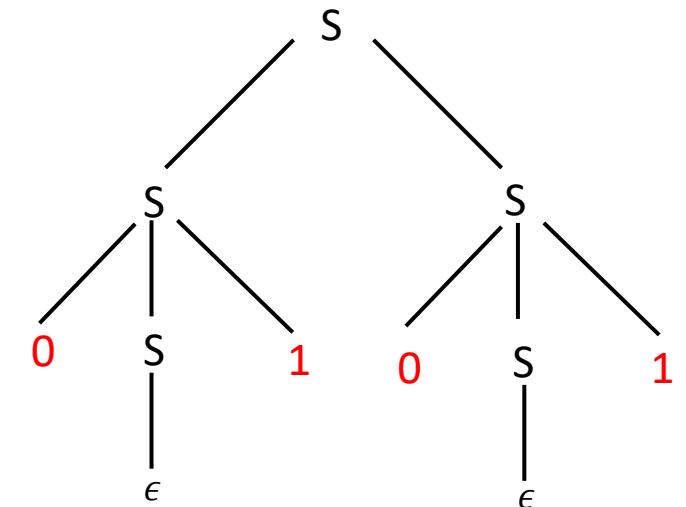
Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Consider the following derivations for 0101:

1.  $S \rightarrow SS \rightarrow 0S1S \rightarrow 0S10S1 \rightarrow 0101$
2.  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 010S1 \rightarrow 0101$
3.  $S \rightarrow SS \rightarrow S0S1 \rightarrow S01 \rightarrow 0S101 \rightarrow 0101$

- The parse trees for all these derivations are the same.



# Parse trees for CFG

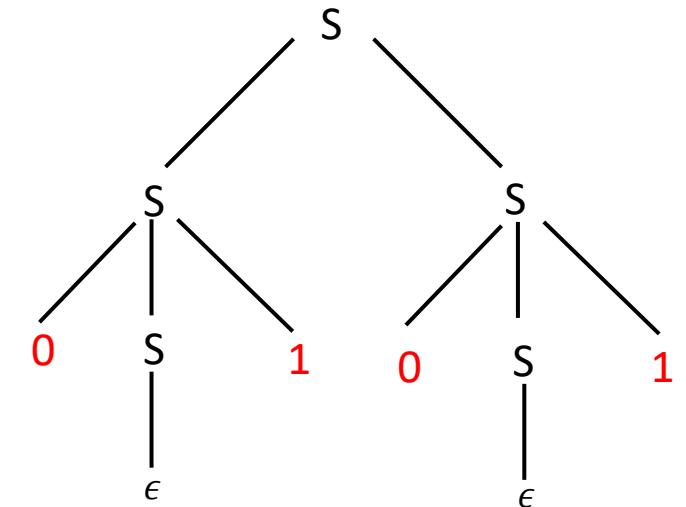
Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Consider the following derivations for 0101:

1.  $S \rightarrow SS \rightarrow 0S1S \rightarrow 0S10S1 \rightarrow 0101$
2.  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 010S1 \rightarrow 0101$
3.  $S \rightarrow SS \rightarrow S0S1 \rightarrow S01 \rightarrow 0S101 \rightarrow 0101$

- The parse trees for all these derivations are the same.
- If a string is derived by replacing only the leftmost variable at every step, then the derivation is a **leftmost derivation**. (e.g. derivation 2.)
- .....rightmost variable = **rightmost derivation** (e.g. derivation 3.)
- Derivations may not always be **leftmost** or **rightmost** (e.g. derivation 1.)



# Parse trees for CFG

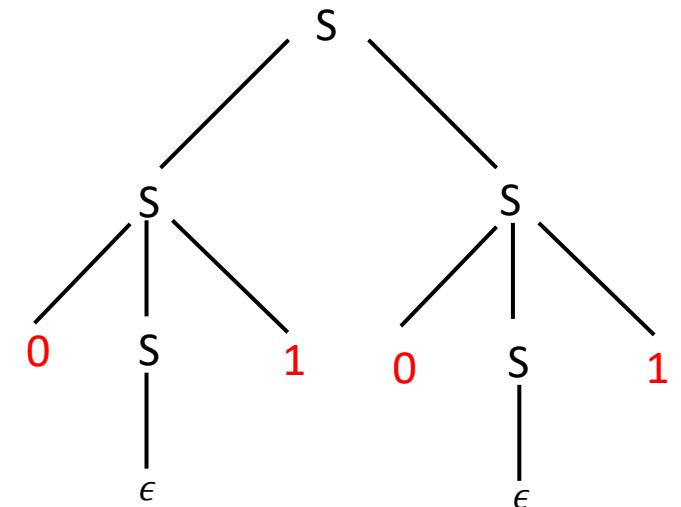
Consider the Grammar  $G$  with the following rules:

$$S \rightarrow 0S1|SS|\epsilon$$

Consider the following derivations for 0101:

1.  $S \rightarrow SS \rightarrow 0S1S \rightarrow 0S10S1 \rightarrow 0101$
2.  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 010S1 \rightarrow 0101$
3.  $S \rightarrow SS \rightarrow S0S1 \rightarrow S01 \rightarrow 0S101 \rightarrow 0101$

- The parse trees for all these derivations are the same.
- If a string is derived by replacing only the leftmost variable at every step, then the derivation is a **leftmost derivation**. (e.g. derivation 2.)
- .....rightmost variable = **rightmost derivation** (e.g. derivation 3.)
- Derivations may not always be **leftmost** or **rightmost** (e.g. derivation 1.)



**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$ .**

# Parse trees for CFG

**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.

# Parse trees for CFG

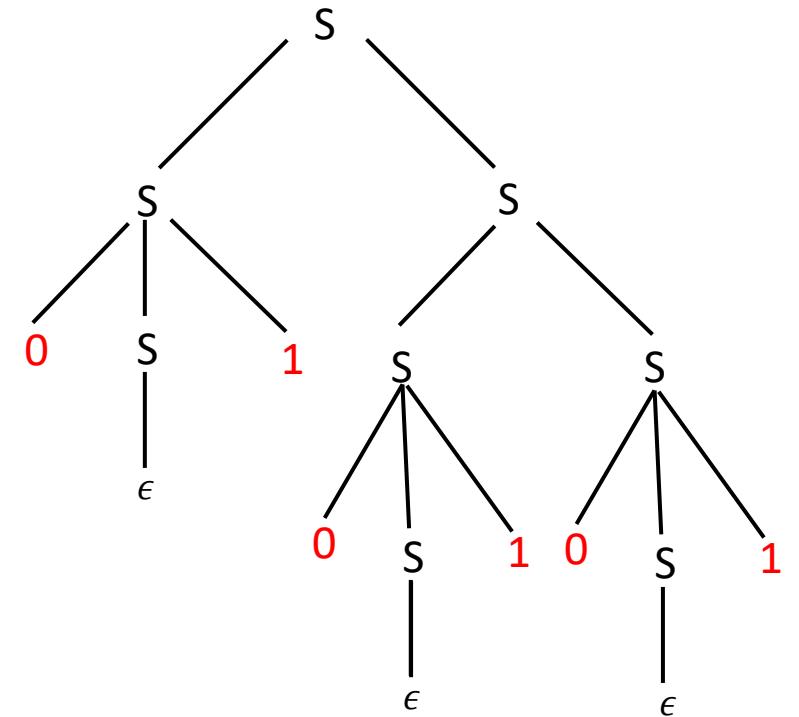
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS$

# Parse trees for CFG

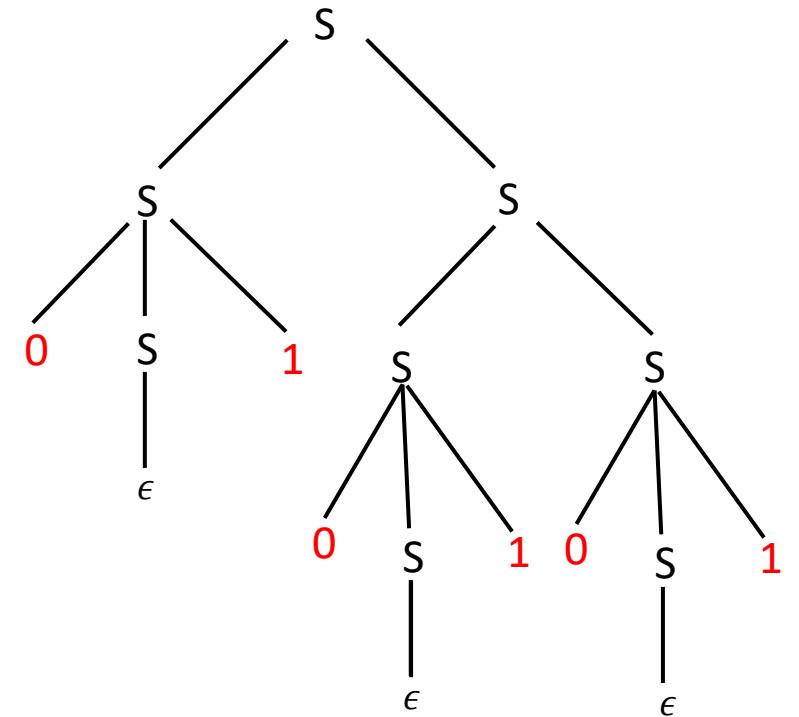
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow \textcolor{red}{SS}$

# Parse trees for CFG

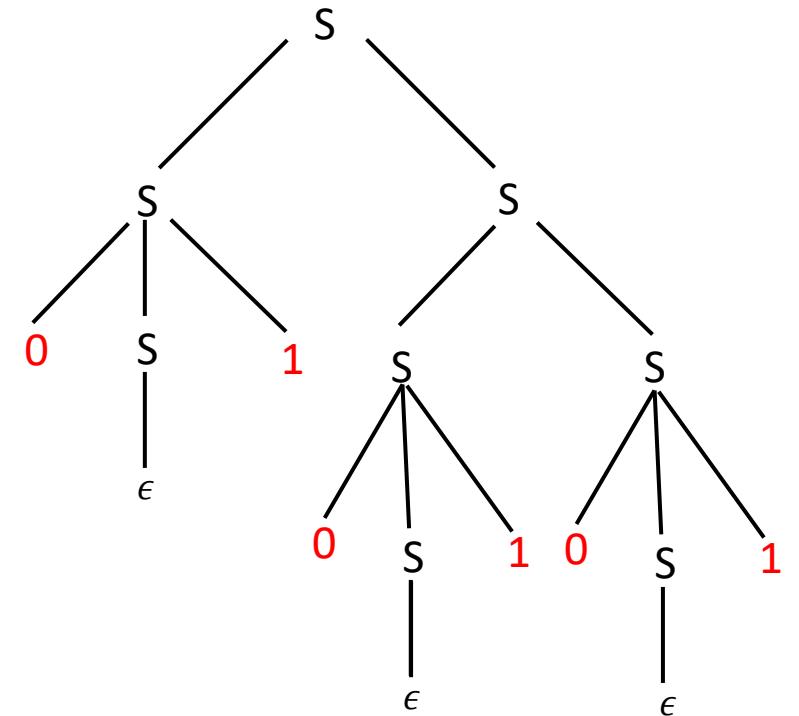
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S$

# Parse trees for CFG

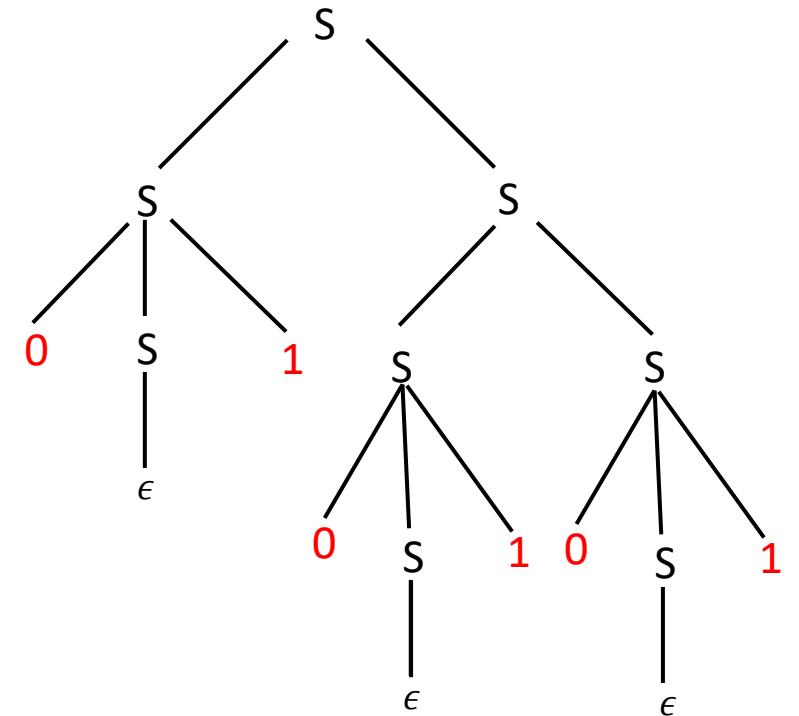
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0\mathbf{S}1S$

# Parse trees for CFG

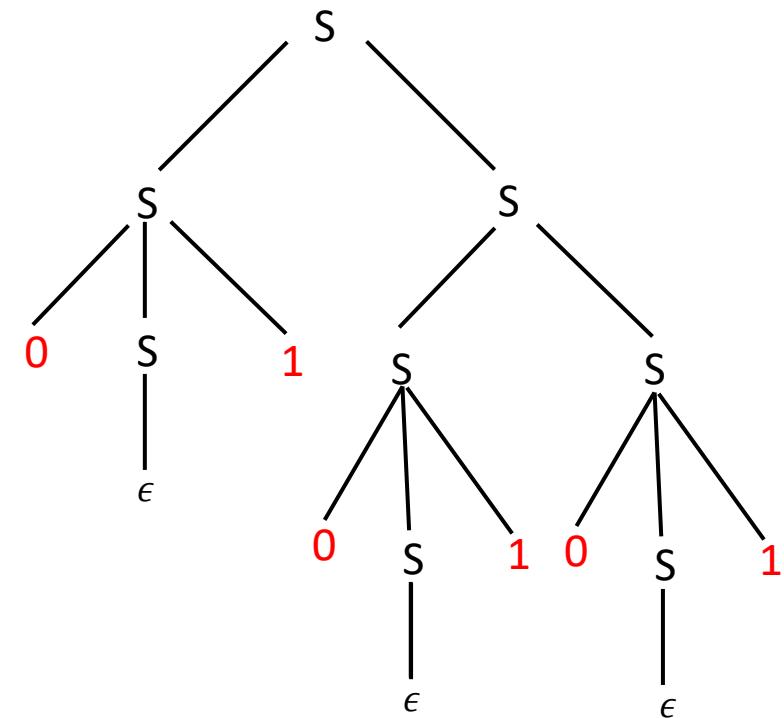
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S$

# Parse trees for CFG

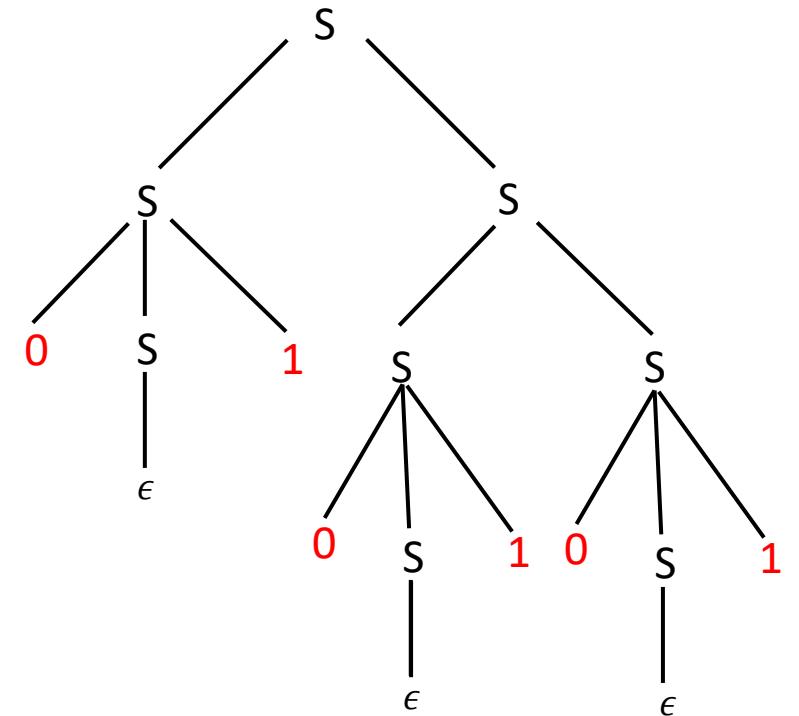
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01\textcolor{red}{S}$

# Parse trees for CFG

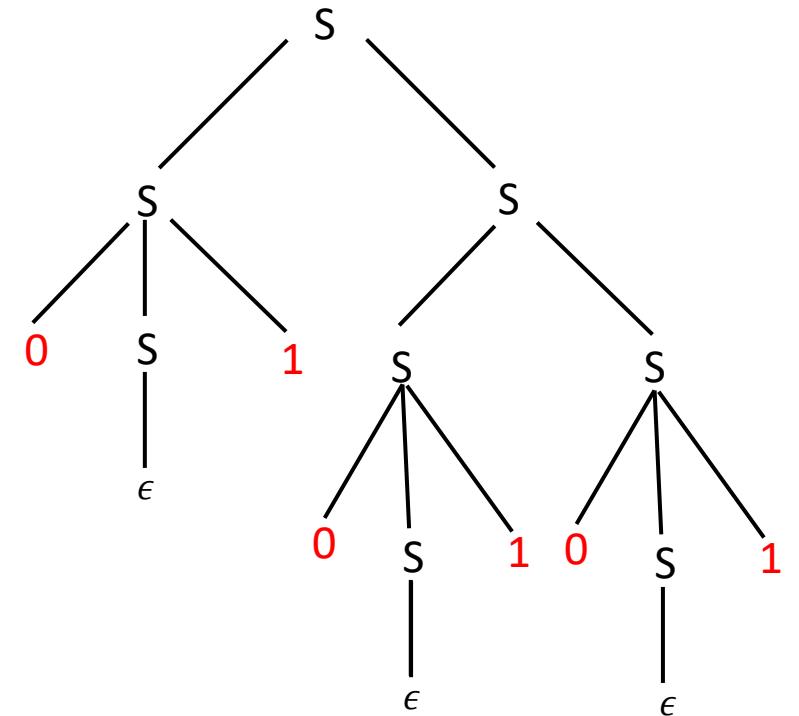
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 01SS$

# Parse trees for CFG

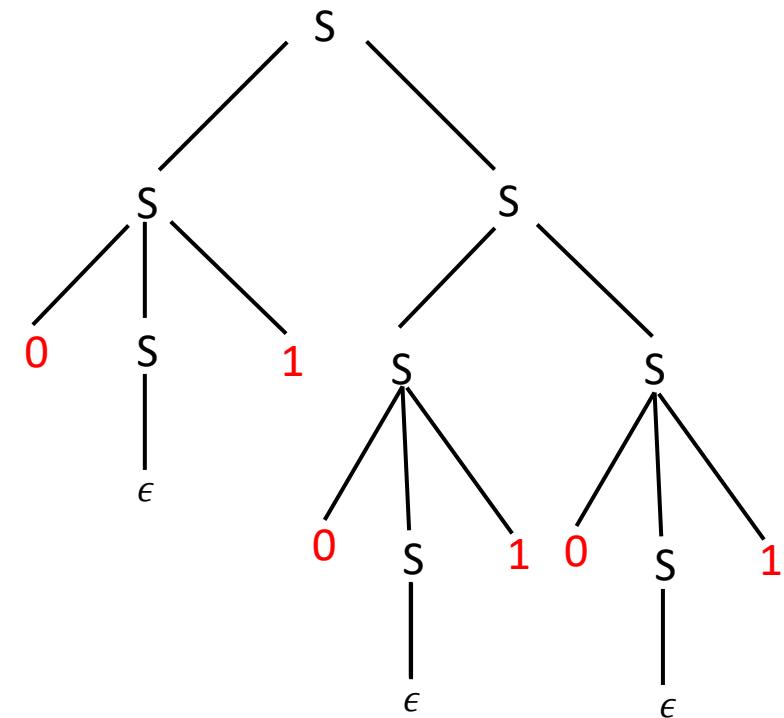
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 01SS$

# Parse trees for CFG

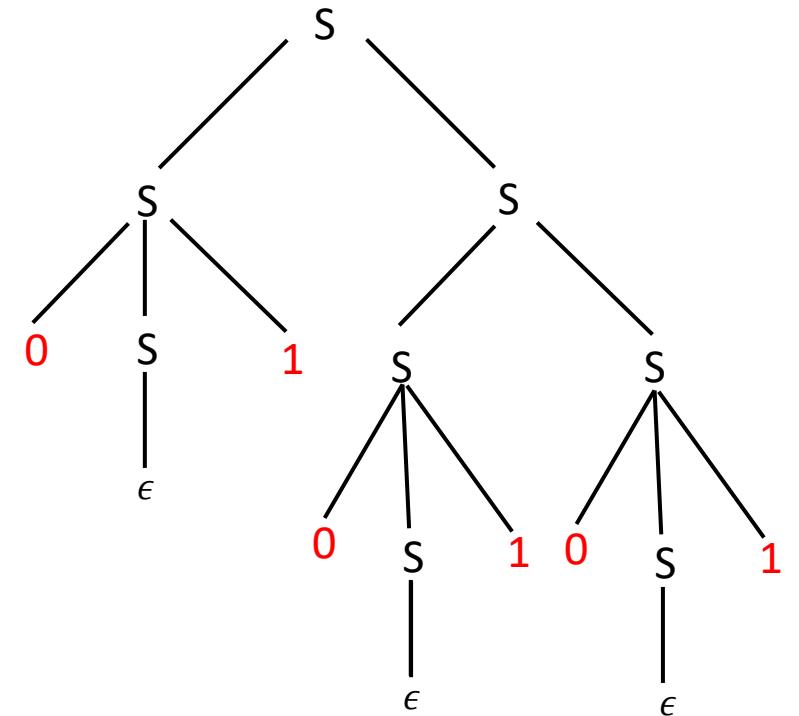
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 01SS \rightarrow 010S1S$

# Parse trees for CFG

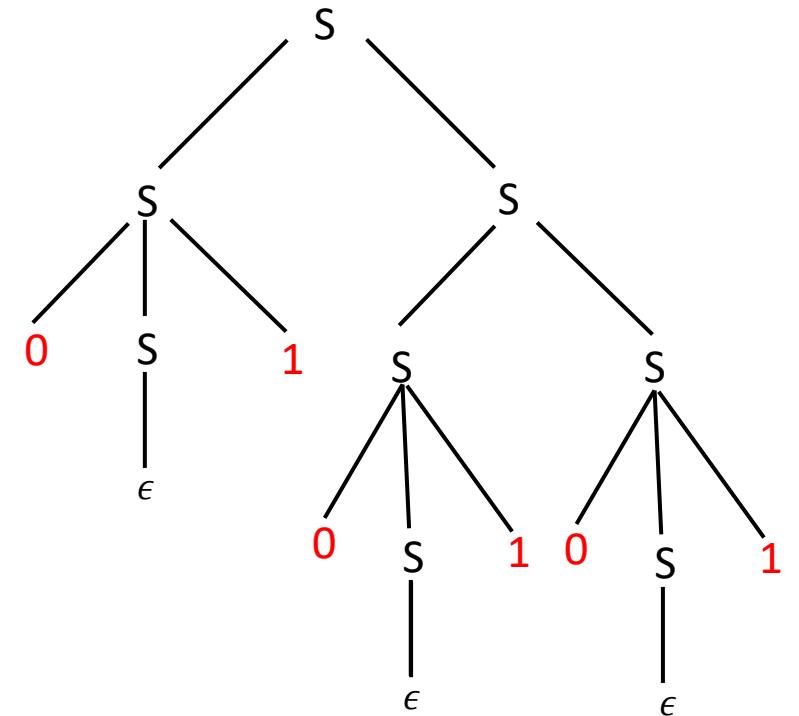
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 01SS \rightarrow 010\textcolor{red}{S}1S$

# Parse trees for CFG

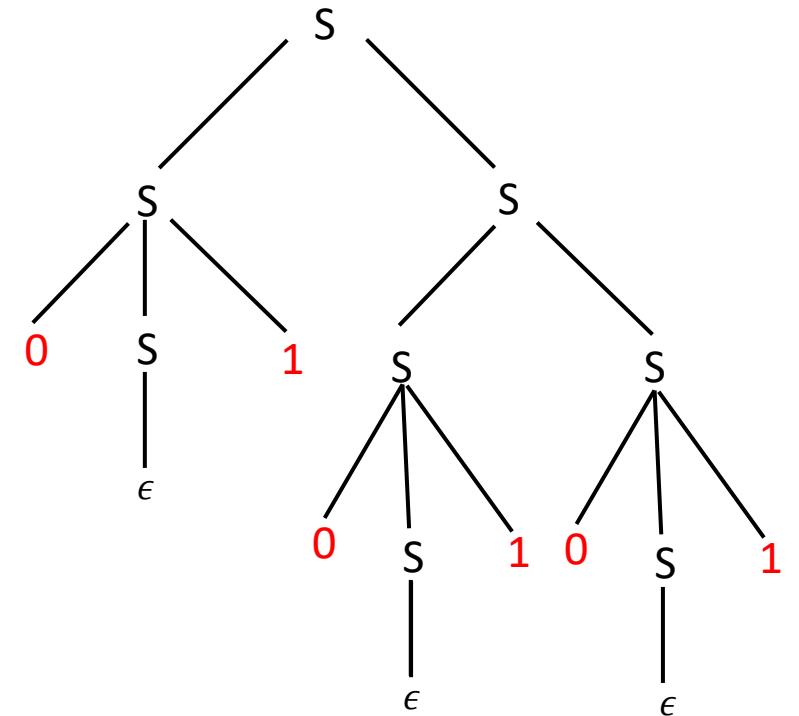
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 01SS \rightarrow 010S1S \rightarrow 0101S$

# Parse trees for CFG

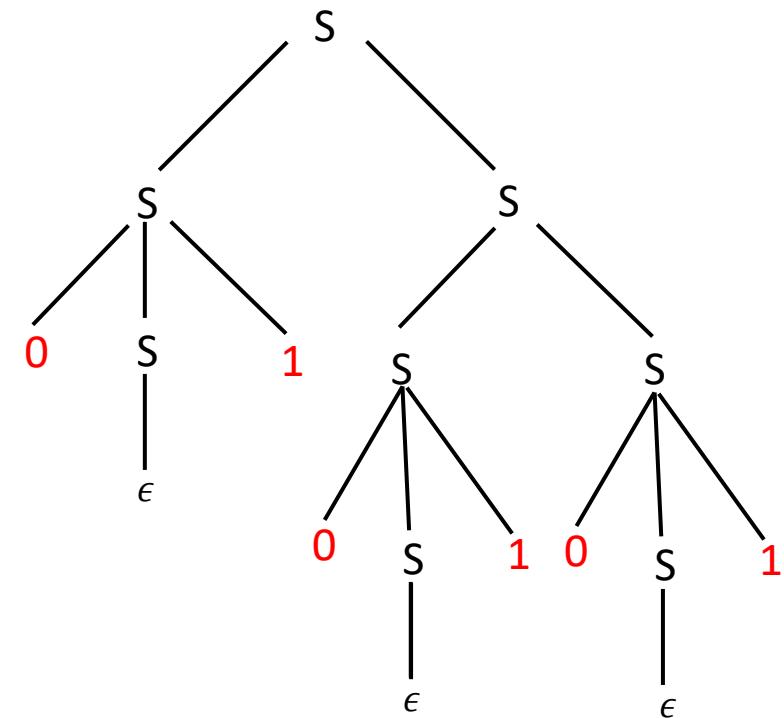
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 01SS \rightarrow 010S1S \rightarrow 0101S$

# Parse trees for CFG

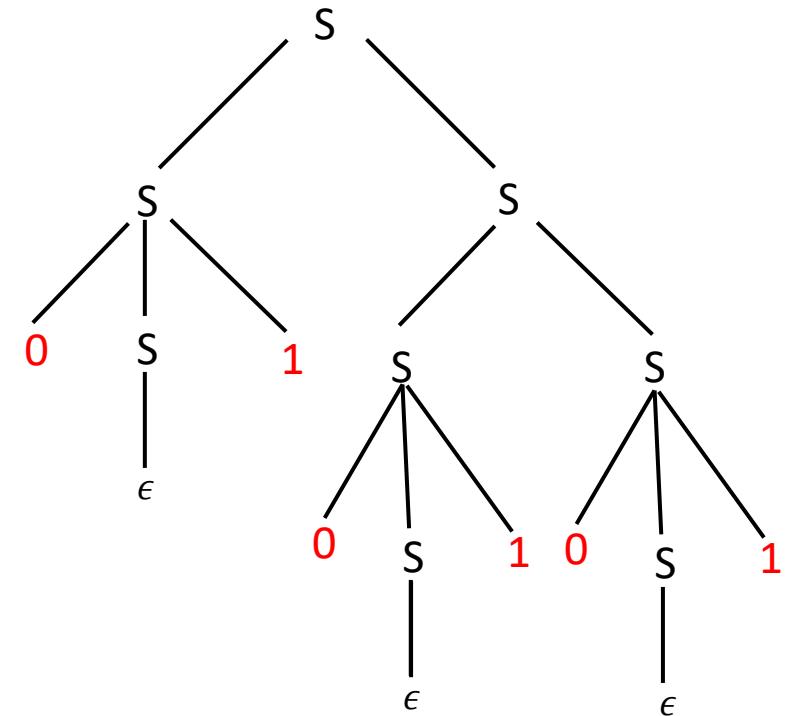
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 01SS \rightarrow 010S1S \rightarrow 0101S \rightarrow 01010S1$

# Parse trees for CFG

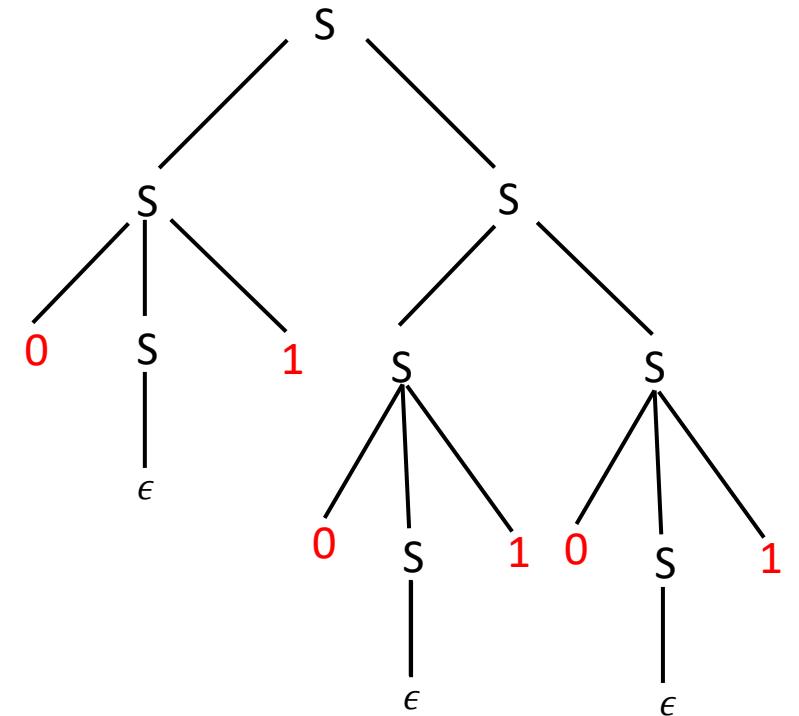
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Show that Grammar  $G$  is ambiguous, i.e.  $\exists \omega \in L(G)$ , such that there are two or more parse trees for  $\omega$ .

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.



Leftmost Derivation:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 01SS \rightarrow 010S1S \rightarrow 0101S \rightarrow 01010S1 \rightarrow 010101$

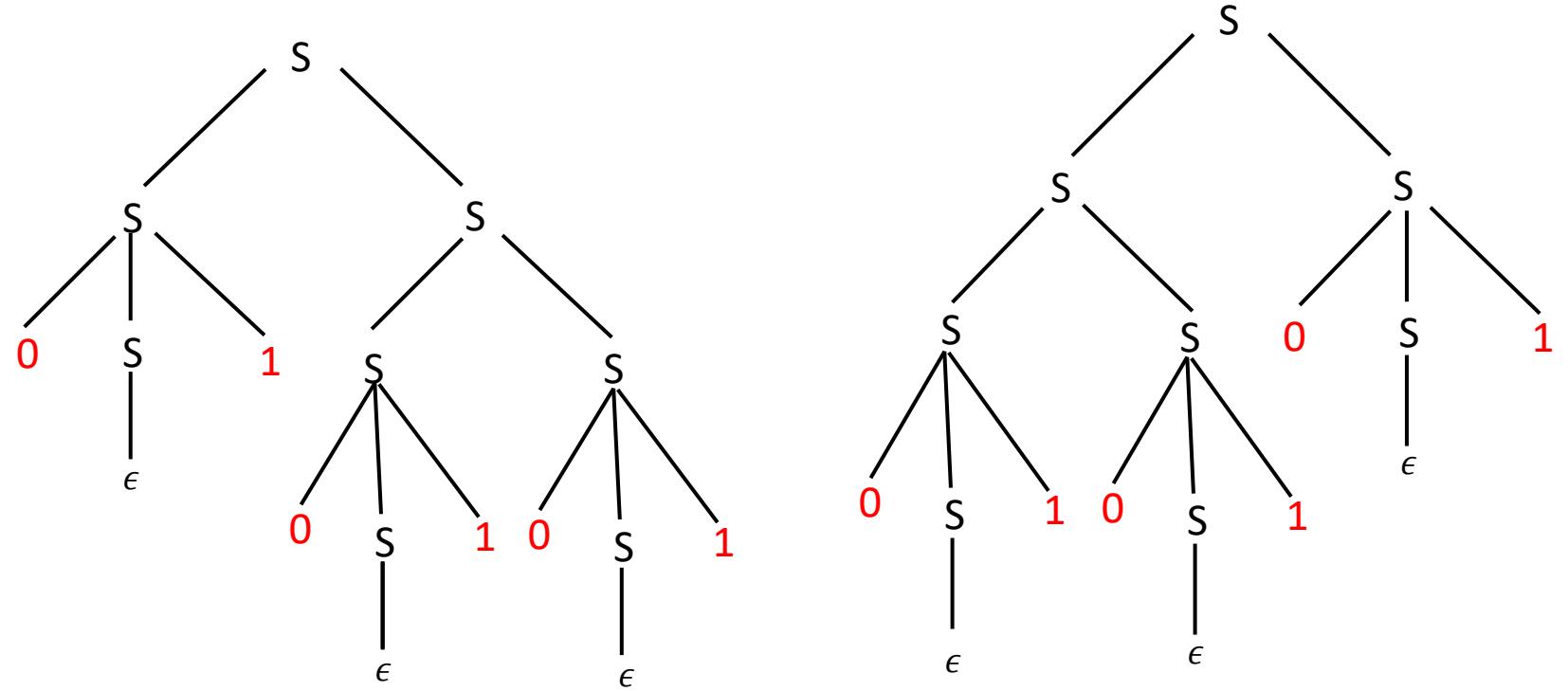
# Parse trees for CFG

**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

Consider the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$

Consider the string  $\omega = 010101$ :

- Show that there exist two different parse trees for **010101**.
- Show that there exist two leftmost derivations for **010101**.

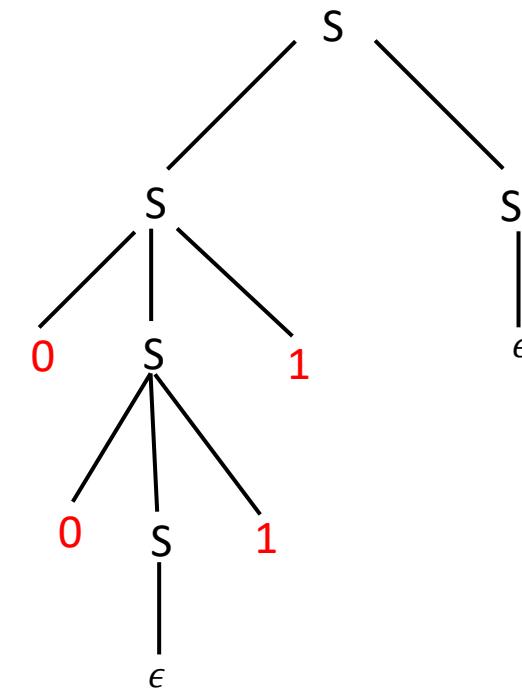
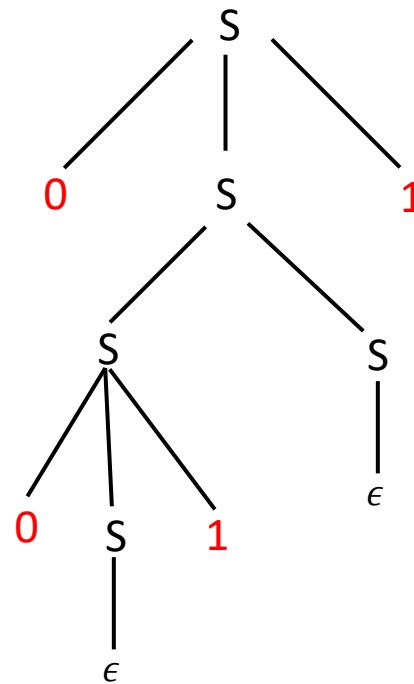
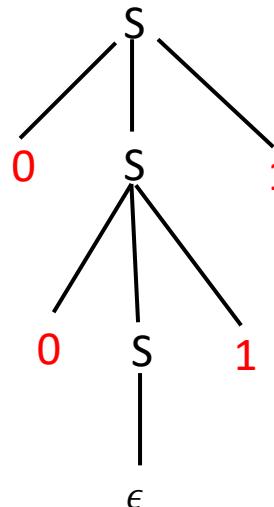


Leftmost Derivation:  $S \rightarrow SS \rightarrow SSS \rightarrow 0S1SS \rightarrow 01SS \rightarrow 010S1S \rightarrow 0101S \rightarrow 01010S1 \rightarrow 010101$

# Parse trees for CFG

Show that the Grammar  $G$  with the following rules:  $S \rightarrow 0S1|SS|\epsilon$  is ambiguous.

Consider string  $\omega = 0011$



LD:  $S \rightarrow 0S1 \rightarrow 00S11 \rightarrow 0011$

LD:  $S \rightarrow 0S1 \rightarrow 0SS1 \rightarrow 00S1S1 \rightarrow 001S1 \rightarrow 0011$

LD:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 00S11S \rightarrow 0011S \rightarrow 0011$

# Ambiguity

**Unique** structures are important. For example:

- The syntax of a programming language can be represented by a CFG.
- A compiler
  - translates the code written in the programming language into a form that is suitable for execution.
  - checks if the underlying programming language is syntactically correct.
- Parse trees are data structures that represent such structures.
- Parse tree for the code helps analyze the syntax. So ambiguity might lead to different interpretations and hence, different outcomes for the same code.

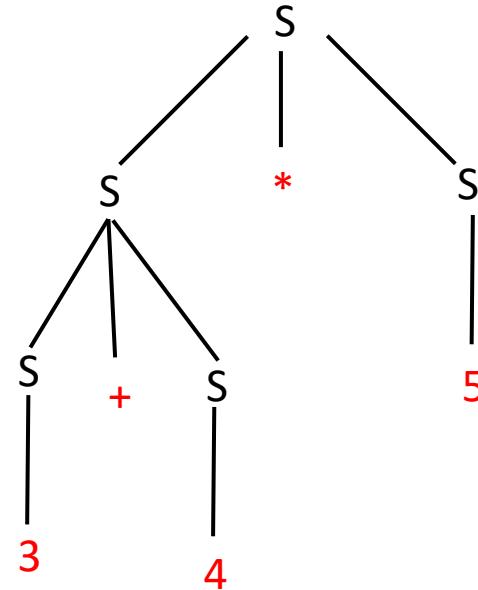
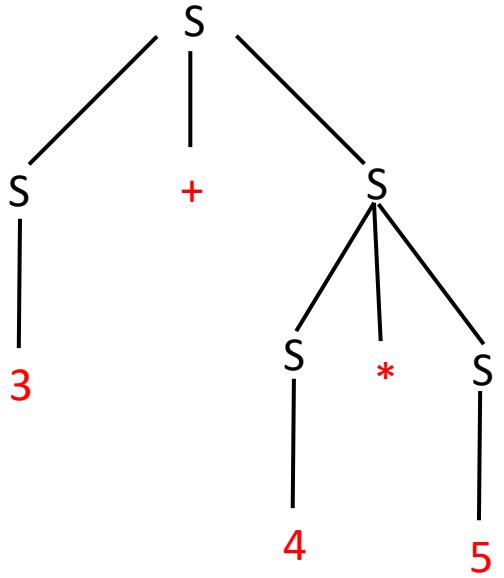
**Ambiguity may not be desirable.**

# Ambiguity

**Ambiguity may not be desirable.**

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

and the derivation of the string  $3 + 4 * 5$



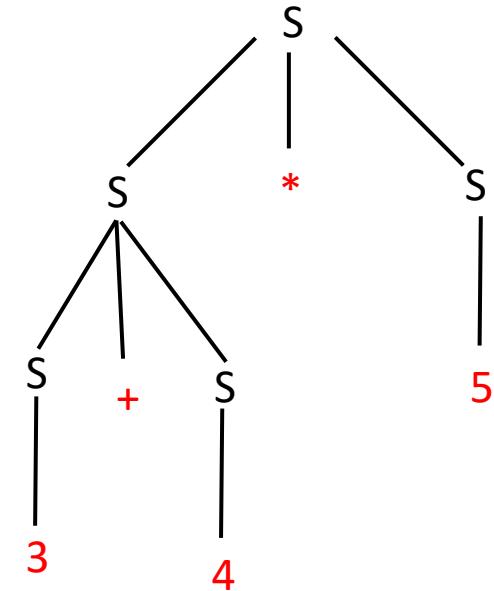
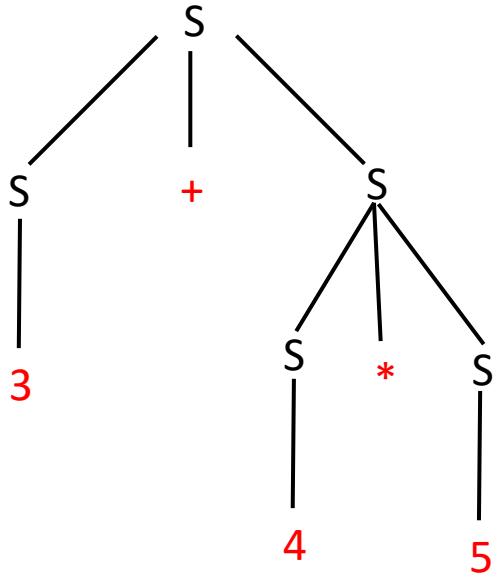
- The grammar contains no information on the precedence relations of the various arithmetic operations.
- The grammar may group  $+$  before  $*$

# Ambiguity

Ambiguity may not be desirable.

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

and the derivation of the string  $3 + 4 * 5$



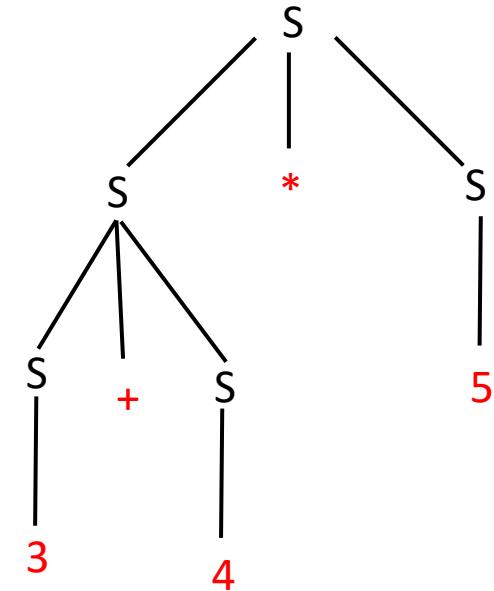
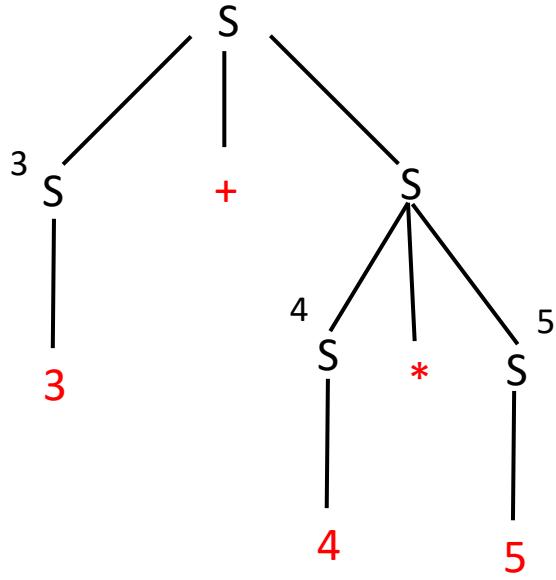
- What will be the result obtained from each of these *parsings*?

# Ambiguity

Ambiguity may not be desirable.

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

and the derivation of the string  $3 + 4 * 5$



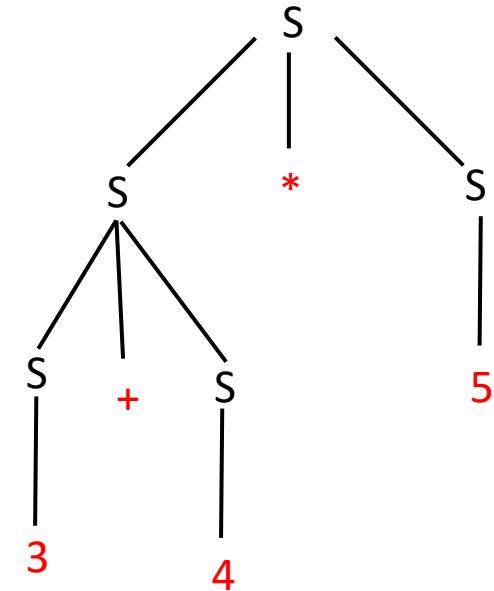
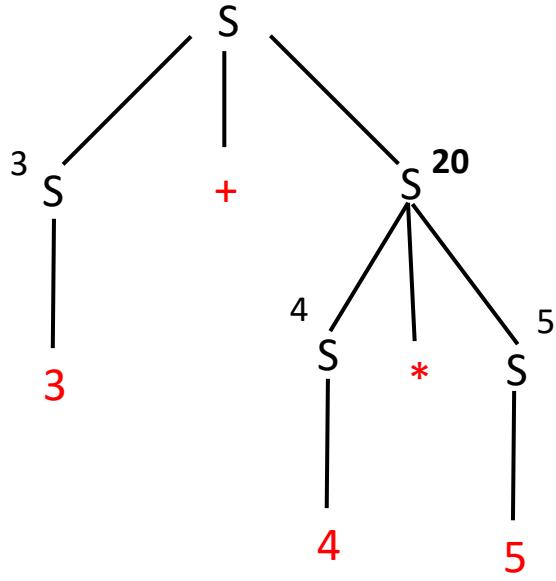
- If the compiler compiles the left parse tree

# Ambiguity

Ambiguity may not be desirable.

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

and the derivation of the string  $3 + 4 * 5$



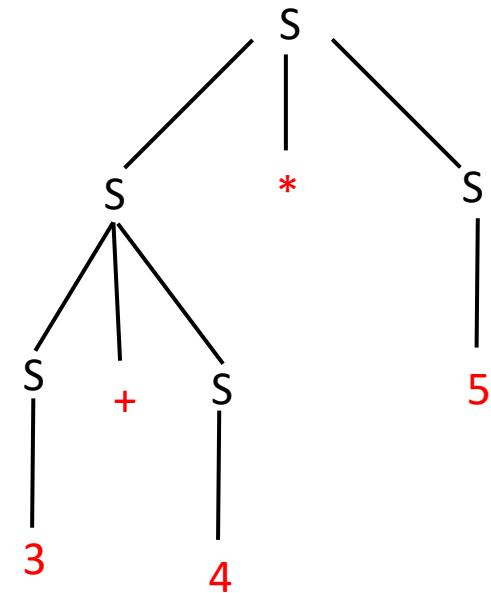
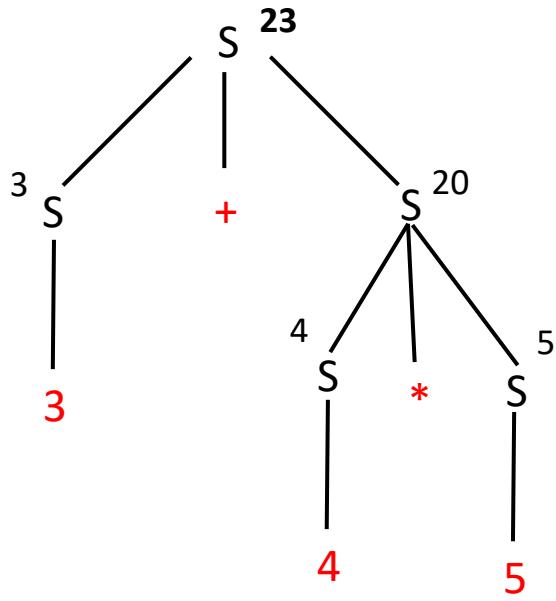
- If the compiler compiles the left parse tree

# Ambiguity

**Ambiguity may not be desirable.**

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

and the derivation of the string  $3 + 4 * 5$



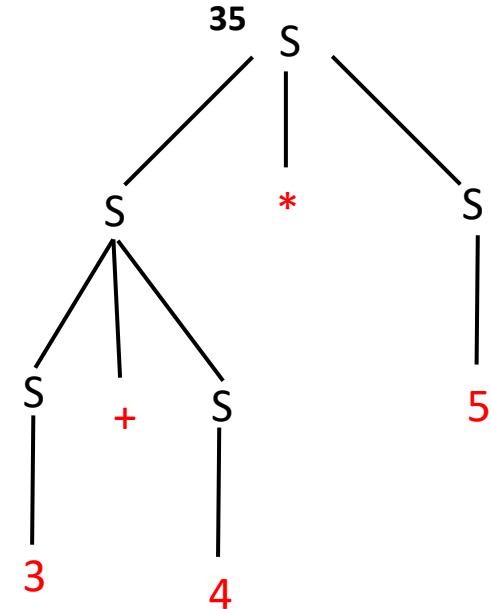
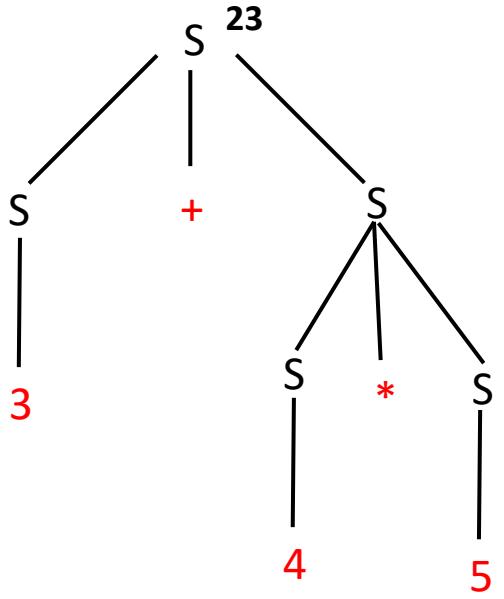
- If the compiler compiles the left parse tree. Outcome = 23

# Ambiguity

Ambiguity may not be desirable.

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

and the derivation of the string  $3 + 4 * 5$



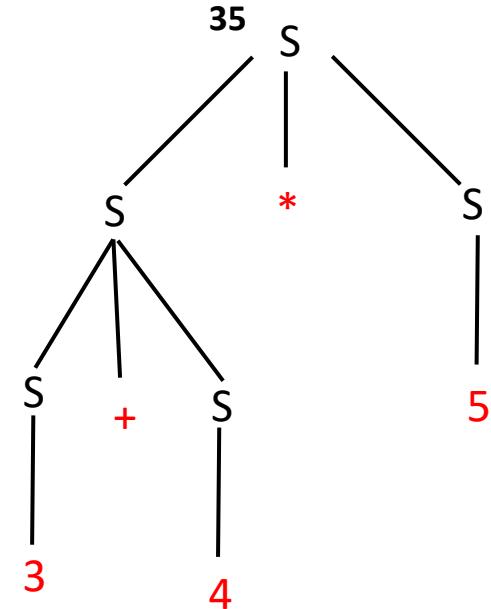
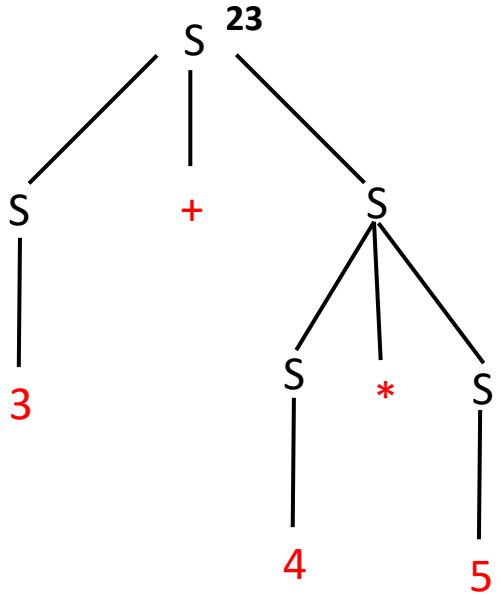
- If the compiler compiles the **right** parse tree. Outcome = **35**

# Ambiguity

Ambiguity may not be desirable.

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

and the derivation of the string  $3 + 4 * 5$



- How can we get rid of this ambiguity?

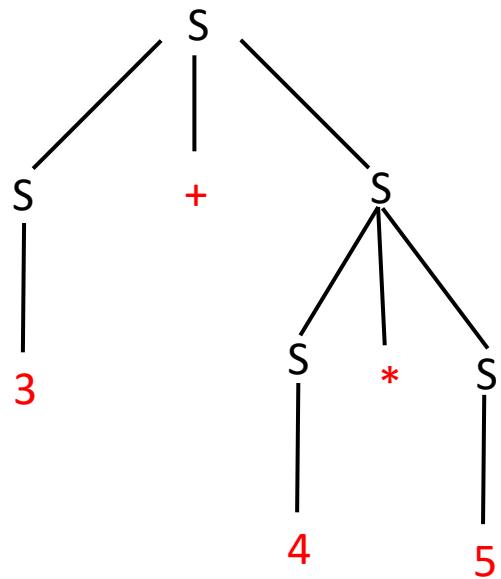
# Ambiguity

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

**How can we get rid of this ambiguity? Change the production rules**

**1) Add parenthesis**

New Grammar:  $S \rightarrow (S + S) \mid (S * S) \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$



Old Parse tree (before adding parenthesis)

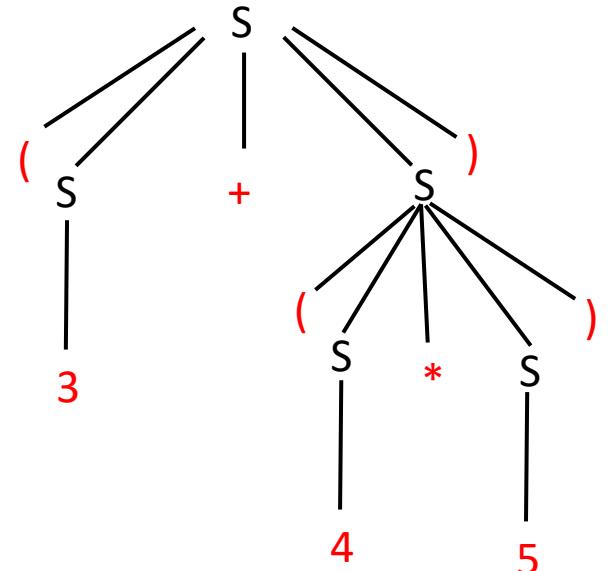
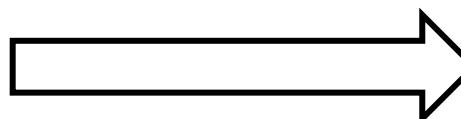
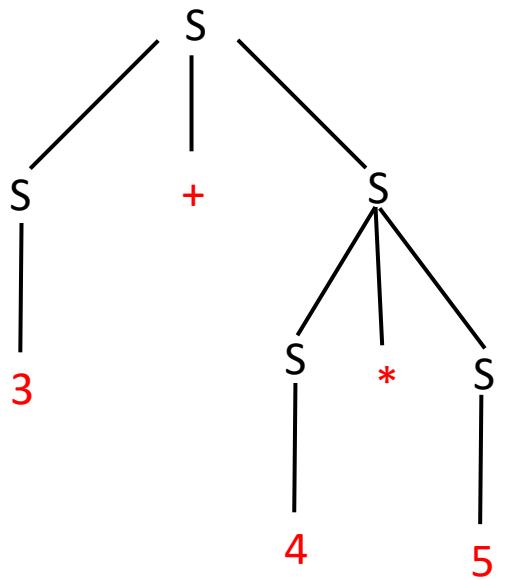
# Ambiguity

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

**How can we get rid of this ambiguity? Change the production rules**

**1) Add parenthesis**

New Grammar:  $S \rightarrow (S + S) \mid (S * S) \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$



$$(3 + (4 * 5)) = 23$$

# Ambiguity

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

**How can we get rid of this ambiguity? Change the production rules**

- 1) Add parentheses
- 2) Add new variables

# Ambiguity

Consider the grammar:  $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$

**How can we get rid of this ambiguity? Change the production rules**

- 1) Add parentheses
- 2) Add new variables

New Grammar:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9 \mid E \end{aligned}$$

# Ambiguity

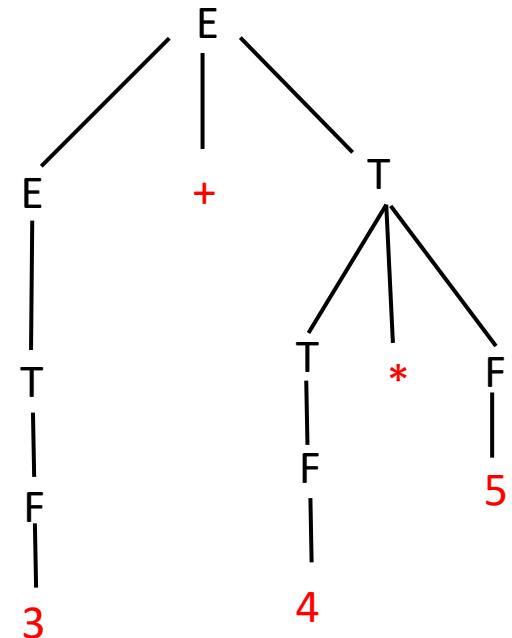
How can we get rid of this ambiguity? Change the production rules

- 1) Add parentheses
- 2) Add new variables

New Grammar:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9 \mid E \end{aligned}$$

Parse tree to derive:  $3 + (4 * 5)$



# Ambiguity

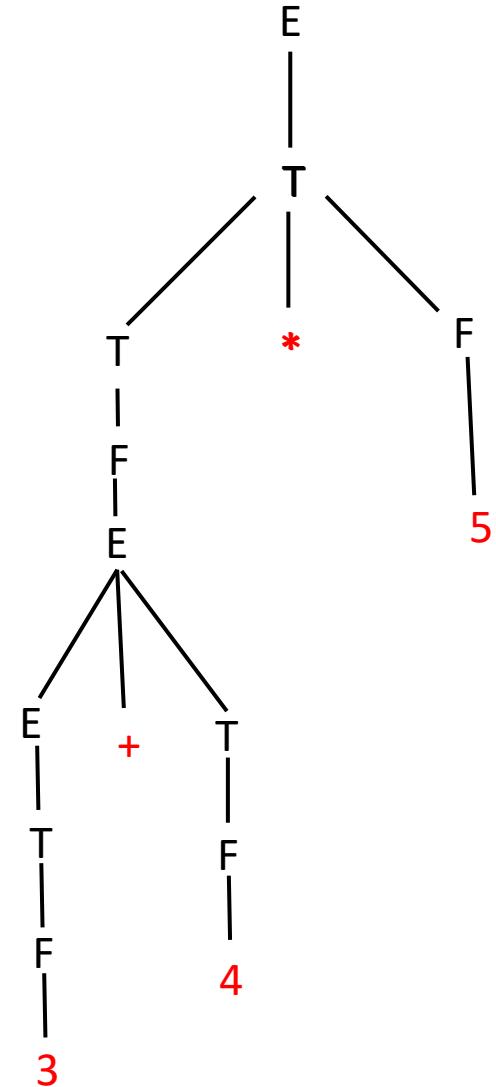
How can we get rid of this ambiguity? Change the production rules

- 1) Add parentheses
- 2) Add new variables

New Grammar:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9 \mid E \end{aligned}$$

Parse tree to derive:  $(3 + 4) * 5$



# Ambiguity

**How can we get rid of this ambiguity? Change the production rules**

- 1) Add parentheses
- 2) Add new variables

- In general, it is not possible to write an algorithm that takes as input a grammar  $G$  and outputs, YES if  $G$  is ambiguous and NO, otherwise. (**Undecidable**)
- A CFL  $L'$  is **inherently ambiguous** if all grammars  $G$  such that  $L(G) = L'$  are ambiguous.
- So removing ambiguity is impossible in general.

# Chomsky Normal Form

Often it is easier to work with CFG in a simple standardized form - the Chomsky Normal Form (CNF) is one of them.

## Chomsky Normal Form

A CFG  $G$  is in CNF if every rule of  $G$  is of the form

$$\begin{aligned}Var &\rightarrow Var\;Var \\Var &\rightarrow ter \\Start\;Var &\rightarrow \epsilon\end{aligned}$$

where  $Var$  can be any variable, including the Start Variable,  $Start\;Var$ .

# Chomsky Normal Form

Often it is easier to work with CFG in a simple standardized form - the Chomsky Normal Form (CNF) is one of them.

## Chomsky Normal Form

A CFG  $G$  is in CNF if every rule of  $G$  is of the form

$$\begin{aligned}Var &\rightarrow Var\;Var \\Var &\rightarrow ter \\Start\;Var &\rightarrow \epsilon\end{aligned}$$

where  $Var$  can be any variable, including the Start Variable,  $Start\;Var$ .

## Why are CNFs useful?

- Suppose you are given a CFG  $G$  and a string  $w$  as input and you have to write an algorithm that decides whether  $G$  generates  $w$ .
- Your algorithm outputs YES if  $G$  generates  $w$  and NO, otherwise.

# Chomsky Normal Form

A CFG  $G$  is in **CNF** if every rule of  $G$  is of the form

$$\begin{aligned}Var &\rightarrow Var\;Var \\Var &\rightarrow ter \\Start\;Var &\rightarrow \epsilon\end{aligned}$$

where  $Var$  can be any variable, including the Start Variable,  $Start\;Var$ .

## Why are CNFs useful?

- Suppose you are given a CFG  $G$  as and a string  $w$  as input and you have to **write an algorithm that decides whether  $G$  generates  $w$** .
  - The algorithm outputs YES if  $G$  generates  $w$  and *NO*, otherwise.
    - ❖ One idea is to go through ALL derivations one by one and output YES if any of them generates  $w$ .
    - ❖ However, infinitely many derivations may have to tried.
    - ❖ So if  $G$  does not generate  $w$ , the algorithm will never stop.
    - ❖ So this problem appears to be **undecidable**.

# Chomsky Normal Form

A CFG  $G$  is in **CNF** if every rule of  $G$  is of the form

$$\begin{aligned}Var &\rightarrow Var\;Var \\Var &\rightarrow ter \\Start\;Var &\rightarrow \epsilon\end{aligned}$$

where  $Var$  can be any variable, including the Start Variable,  $Start\;Var$ .

## Why are CNFs useful?

Suppose you are given a CFG  $G$  and a string  $w$  as input and you have to **write an algorithm that decides whether  $G$  generates  $w$** . This problem appears to be **undecidable**.

# Chomsky Normal Form

A CFG  $G$  is in **CNF** if every rule of  $G$  is of the form

$$\begin{aligned}Var &\rightarrow Var\;Var \\Var &\rightarrow ter \\Start\;Var &\rightarrow \epsilon\end{aligned}$$

where  $Var$  can be any variable, including the Start Variable,  $Start\;Var$ .

## Why are CNFs useful?

Suppose you are given a CFG  $G$  as and a string  $w$  as input and you have to **write an algorithm that decides whether  $G$  generates  $w$** .

- Converting  $G$  first to a CNF alleviates this and **makes the problem decidable**.
- It limits the number of steps in derivations required to generate any  $w \in L(G)$ .
- If  $w \in L(G)$ , then a CFG in Chomsky Normal Form has **derivations of  $2n - 1$  steps** for input strings  $w$  of length  $n$  (We will prove this shortly).

# Chomsky Normal Form

A CFG  $G$  is in **CNF** if every rule of  $G$  is of the form

$$\begin{aligned}Var &\rightarrow Var \; Var \\Var &\rightarrow ter \\Start \; Var &\rightarrow \epsilon\end{aligned}$$

where  $Var$  can be any variable, including the Start Variable,  $Start \; Var$ .

A CFG in Chomsky Normal Form has derivations of  $2n - 1$  steps for generating strings  $w \in L(G)$  of length  $n$ .

## Why are CNFs useful?

Suppose you are given a CFG  $G$  as and a string  $w$  as input and you have to **write an algorithm that decides whether  $G$  generates  $w$** .

1. Convert  $G$  to CNF.
2. List all derivations of  $2n - 1$  steps, where  $|w| = n$ . (There are a finite number of these)
3. If ANY of these derivations generate  $w$ , output YES, otherwise output NO.

# Chomsky Normal Form

A CFG  $G$  is in **CNF** if every rule of  $G$  is of the form

$$\begin{aligned}Var &\rightarrow Var \; Var \\Var &\rightarrow ter \\Start \; Var &\rightarrow \epsilon\end{aligned}$$

where  $Var$  can be any variable, including the Start Variable,  $Start \; Var$ .

- 1) A CFG in Chomsky Normal Form has derivations of  $2n - 1$  steps for generating strings  $w \in L(G)$  of length  $n$ .
- 2) Any CFL can be generated by a CFG written in Chomsky Normal Form.

To prove 1) use induction!

# Chomsky Normal Form

Prove that a CFG in Chomsky Normal Form has derivations of  $2n - 1$  steps for generating strings  $w \in L(G)$  of length  $n$ .

**Proof:** Note that any CFG in CNF can be written as:

$A \rightarrow BC$	[ $B, C$ are not start variables]
$A \rightarrow a$	[ $a$ is a terminal]
$S \rightarrow \epsilon$	[ $S$ is the Start Variable]

We will prove this by **induction**.

**(Basic step)** Let  $|w| = 1$ . Then **one** application of the second rule would suffice. So any derivation of  $w$  would need  $2|w| - 1 = 1$  step.

**(Inductive hypothesis)** Assume the statement of the theorem to be true for any string of length at most  $k$  where  $k \geq 1$ . Now we shall show that it holds for any  $w \in L(G)$  such that  $|w| = k + 1$ .

# Chomsky Normal Form

Prove that a CFG in Chomsky Normal Form has derivations of  $2n - 1$  steps for generating strings  $w \in L(G)$  of length  $n$ .

**Proof:** Note that any CFG in CNF can be written as:

$A \rightarrow BC$	[ $B, C$ are not start variables]
$A \rightarrow a$	[ $a$ is a terminal]
$S \rightarrow \epsilon$	[ $S$ is the Start Variable]

We will prove this by **induction**.

**(Basic step)** Let  $|w| = 1$ . Then **one** application of the second rule would suffice. So any derivation of  $w$  would need  $2|w| - 1 = 1$  step.

**(Inductive hypothesis)** Assume the statement of the theorem to be true for any string of length at most  $k$  where  $k \geq 1$ . Now we shall show that it holds for any  $w \in L(G)$  such that  $|w| = k + 1$ .

Since  $|w| > 1$ , any derivation will start from the rule  $A \rightarrow BC$ . So  $w = xy$ , where  $B \xrightarrow{*} x$ ,  $|x| > 0$  and  $C \xrightarrow{*} y$ ,  $|y| > 0$ . But since  $|x|, |y| \leq k$ , and we have that by the inductive hypothesis: (i) number of steps in the derivation  $B \xrightarrow{*} x$  is  $2|x| - 1$  and (ii) number of steps in the derivation  $C \xrightarrow{*} y$  is  $2|y| - 1$ . So the number of steps in the derivation of  $w$  is

$$1 + (2|x| - 1) + (2|y| - 1) = 2(|x| + |y|) - 1 = 2|w| - 1 = 2(k + 1) - 1.$$

# Chomsky Normal Form

A CFG  $G$  is in **CNF** if every rule of  $G$  is of the form

$$\begin{aligned}Var &\rightarrow Var \; Var \\Var &\rightarrow ter \\Start \; Var &\rightarrow \epsilon\end{aligned}$$

where  $Var$  can be any variable, including the Start Variable,  $Start \; Var$ .

- 1) A CFG in Chomsky Normal Form has derivations of  $2n - 1$  steps for generating strings  $w \in L(G)$  of length  $n$ .
- 2) **Any CFL can be generated by a CFG written in Chomsky Normal Form.**

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

1. Add a new start variable  $S' \rightarrow S$
2. Remove  $\epsilon$  rules of the form  $A \rightarrow \epsilon$ 
  - Remove nullable symbols/rules
3. Remove unit (short) rules of the form  $A \rightarrow B$ 
  - Remove useless symbols/rules
4. Remove long rules of the form  $A \rightarrow u_1 u_2 \cdots u_k$ 
  - Remove useless symbols/rules

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

1. Add a new start variable  $S' \rightarrow S$
2. Remove  $\epsilon$  rules of the form  $A \rightarrow \epsilon$

We remove the rule  $A \rightarrow \epsilon$ . For each occurrence of  $A$  in the right side of the rule, we add a new rule with the occurrence of  $A$  deleted.

E.g.: Consider any rule  $B \rightarrow uAvAw$  ( $u, v, w$  can be strings of variables and terminals)

Then new rules:  $B \rightarrow uAvAw|uvAw|uAvw|uvw$

What if you had a rule such as  $B \rightarrow A$ ? Then we would have needed to add a rule  $B \rightarrow \epsilon$  (unless this rule has been already removed) as  $B$  is a **nullable variable**.

Repeat this procedure, until all  $\epsilon$ -rules are removed.

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

1. Add a new start variable  $S' \rightarrow S$
2. Remove  $\epsilon$  rules of the form  $A \rightarrow \epsilon$

E.g.:  $S \rightarrow 0|X0|XYZ$   
 $X \rightarrow Y|\epsilon$   
 $Y \rightarrow 1|X$

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

**Remove  $\epsilon$  rules of the form  $A \rightarrow \epsilon$**  ( For each occurrence of  $A$  in the right side of the rule, add a new rule with the occurrence of  $A$  deleted ; Remove nullable variables, Repeat the procedure until all  $\epsilon$  rules are removed)

E.g.:  $S \rightarrow 0|X0|XYZ$

$X \rightarrow Y|\epsilon$

$Y \rightarrow 1|X$

To remove  $X \rightarrow \epsilon$ , we add new rules:  $S \rightarrow 0|X0|XYZ$

$X \rightarrow Y$

$Y \rightarrow 1|X|\epsilon$

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

**Remove  $\epsilon$  rules of the form  $A \rightarrow \epsilon$**  ( For each occurrence of  $A$  in the right side of the rule, add a new rule with the occurrence of  $A$  deleted ; Remove nullable variables, Repeat the procedure until all  $\epsilon$  rules are removed)

E.g.:  $S \rightarrow 0|X0|XYZ$   
 $X \rightarrow Y|\epsilon$   
 $Y \rightarrow 1|X$

To remove  $X \rightarrow \epsilon$ , we add new rules:  $S \rightarrow 0|X0|XYZ$   
 $X \rightarrow Y$   
 $Y \rightarrow 1|X|\epsilon$

To remove  $Y \rightarrow \epsilon$ , we add:

$$S \rightarrow 0|X0|XYZ|ZZ$$

$$X \rightarrow Y$$

$$Y \rightarrow 1|X$$

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

1. Add a new start variable  $S' \rightarrow S$
2. Remove  $\epsilon$  rules of the form  $A \rightarrow \epsilon$
3. Remove unit rules of the form  $A \rightarrow B$

We remove the rule  $A \rightarrow B$  and whenever a rule  $B \rightarrow u$  appears ( $u$  is a string of terminals and variables), we add a new rule  $A \rightarrow u$ , unless this rule was already removed.

Repeat these steps until all unit rules are removed.

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

**Remove unit rules of the form  $A \rightarrow B$**  (Whenever a rule  $B \rightarrow u$  appears, we add a new rule  $A \rightarrow u$ , unless this rule was already removed. Repeat these steps until all unit rules are removed.)

E.g.:

$$\begin{aligned}S &\rightarrow A|11 \\A &\rightarrow B|1 \\B &\rightarrow S|0\end{aligned}$$

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

**Remove unit rules of the form  $A \rightarrow B$**  (Whenever a rule  $B \rightarrow u$  appears, we add a new rule  $A \rightarrow u$ , unless this rule was already removed. Repeat these steps until all unit rules are removed.)

E.g.:

$$\begin{aligned} S &\rightarrow A|11 \\ A &\rightarrow B|1 \\ B &\rightarrow S|0 \end{aligned}$$

Remove $S \rightarrow A$	Remove $A \rightarrow B$	Remove $B \rightarrow S$	Remove $B \rightarrow B$	Remove $S \rightarrow B$	Remove $A \rightarrow S$
$S \rightarrow 11 B 1$	$S \rightarrow 11 B 1$	$S \rightarrow 11 B 1$	$S \rightarrow 11 B 1$	$S \rightarrow 11 \mathbf{0} 1$	$S \rightarrow 11 \mathbf{0} 1$
$A \rightarrow B 1$	$A \rightarrow 1 S \mathbf{0}$	$A \rightarrow 1 S 0$	$A \rightarrow 1 S 0$	$A \rightarrow 1 S 0$	$A \rightarrow 1 \mathbf{11} 0$
$B \rightarrow S 0$	$B \rightarrow S 0$	$B \rightarrow 0 \mathbf{11} 1 B$	$B \rightarrow 0 11 1$	$B \rightarrow 0 11 1$	$B \rightarrow 0 11 1$

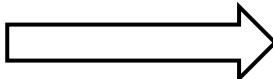
# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

**Remove unit rules of the form  $A \rightarrow B$**  (Whenever a rule  $B \rightarrow u$  appears, we add a new rule  $A \rightarrow u$ , unless this rule was already removed. Repeat these steps until all unit rules are removed.)

$$\begin{aligned} S &\rightarrow A|11 \\ A &\rightarrow B|1 \\ B &\rightarrow S|0 \end{aligned}$$



$$\begin{aligned} S &\rightarrow 11|0|1 \\ A &\rightarrow 1|11|0 \\ B &\rightarrow 0|11|1 \end{aligned}$$

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

1. Add a new start variable  $S' \rightarrow S$
2. Remove  $\epsilon$  rules of the form  $A \rightarrow \epsilon$
3. Remove unit rules of the form  $A \rightarrow B$
4. **Remove long rules of the form  $A \rightarrow u_1u_2 \cdots u_k$**

Note that each  $u_i$  could be a variable or a terminal. We do the following:

- Replace  $A \rightarrow u_1u_2 \cdots u_k$ , ( $k \geq 3$ ) with the rules  $A \rightarrow u_1A_1$ ,  $A_1 \rightarrow u_2A_2$ ,  $\dots$ ,  $A_{k-2} \rightarrow u_{k-1}u_k$
- We replace any terminal  $u_i$  in the preceding rules with the new variable  $U_i$  and add the rule  $U_i \rightarrow u_i$

# Chomsky Normal Form

**Any CFL can be generated by a CFG written in Chomsky Normal Form.**

**Proof:** The proof is constructive. Suppose we have a CFG  $G$  with a set of rules. To convert  $G$  into CNF, we do the following:

**Add a new start variable  $S' \rightarrow S$**

**Remove  $\epsilon$  rules of the form  $A \rightarrow \epsilon$**  ( For each occurrence of  $A$  in the right side of the rule, add a new rule with the occurrence of  $A$  deleted ; Remove nullable variables, Repeat the procedure until all  $\epsilon$  rules are removed).

**Remove unit rules of the form  $A \rightarrow B$**  (Whenever a rule  $B \rightarrow u$  appears, we add a new rule  $A \rightarrow u$ , unless this rule was already removed. Repeat these steps until all unit rules are removed.)

**Remove long rules of the form  $A \rightarrow u_1u_2 \cdots u_k$**  (Replace  $A \rightarrow u_1u_2 \cdots u_k$ , ( $k \geq 3$ ) with the rules  $A \rightarrow u_1A_1$ ,  $A_1 \rightarrow u_2A_2, \dots, A_{k-2} \rightarrow u_{k-1}u_k$ ; Replace any terminal  $u_i$  in the preceding rules with the new variable  $U_i$  and add the rule  $U_i \rightarrow u_i$  ).

# Chomsky Normal Form

**CNF:**

- |                          |                                   |
|--------------------------|-----------------------------------|
| $A \rightarrow BC$       | [ $B, C$ are not start variables] |
| $A \rightarrow a$        | [ $a$ is a terminal]              |
| $S \rightarrow \epsilon$ | [ $S$ is the Start Variable]      |

Convert the CFG

to CNF.

$$\begin{aligned}S &\rightarrow ASA|aB \\A &\rightarrow B|S \\B &\rightarrow b|\epsilon\end{aligned}$$

**1. Add a new start variable**

$$\begin{aligned}S' &\rightarrow S \\S &\rightarrow ASA|aB \\A &\rightarrow B|S \\B &\rightarrow b|\epsilon\end{aligned}$$

**2a. Remove  $\epsilon$  rules ( $B \rightarrow \epsilon$ )**

$$\begin{aligned}S' &\rightarrow S \\S &\rightarrow ASA|aB|a \\A &\rightarrow B|S|\epsilon \\B &\rightarrow b\end{aligned}$$

**2b. Remove  $\epsilon$  rules ( $A \rightarrow \epsilon$ )**

$$\begin{aligned}S' &\rightarrow S \\S &\rightarrow ASA|aB|a|AS|SA|S \\A &\rightarrow B|S \\B &\rightarrow b\end{aligned}$$

# Chomsky Normal Form

**CNF:**

$A \rightarrow BC$  [ $B, C$  are not start variables]

$A \rightarrow a$  [ $a$  is a terminal]

$S \rightarrow \epsilon$  [ $S$  is the Start Variable]

Convert the CFG

$$\begin{aligned} S &\rightarrow ASA|aB \\ A &\rightarrow B|S \\ B &\rightarrow b|\epsilon \end{aligned}$$

to CNF.

**3a. Remove  $S \rightarrow S$**

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow ASA|aB|a|AS|SA \\ A &\rightarrow B|S \\ B &\rightarrow b \end{aligned}$$

**3b. Remove  $S' \rightarrow S$**

$$\begin{aligned} S' &\rightarrow ASA|aB|a|AS|SA \\ S &\rightarrow ASA|aB|a|AS|SA \\ A &\rightarrow B|S \\ B &\rightarrow b \end{aligned}$$

**3c. Remove  $A \rightarrow B$**

$$\begin{aligned} S' &\rightarrow ASA|aB|a|AS|SA \\ S &\rightarrow ASA|aB|a|AS|SA \\ A &\rightarrow S|b \\ B &\rightarrow b \end{aligned}$$

**3d. Remove  $A \rightarrow S$**

$$\begin{aligned} S' &\rightarrow ASA|aB|a|AS|SA \\ S &\rightarrow ASA|aB|a|AS|SA \\ A &\rightarrow b|ASA|aB|a|AS|SA \\ B &\rightarrow b \end{aligned}$$

# Chomsky Normal Form

**CNF:**

$A \rightarrow BC$  [ $B, C$  are not start variables]

$A \rightarrow a$  [ $a$  is a terminal]

$S \rightarrow \epsilon$  [ $S$  is the Start Variable]

Convert the CFG

$$\begin{aligned} S &\rightarrow ASA|aB \\ A &\rightarrow B|S \\ B &\rightarrow b|\epsilon \end{aligned}$$

to CNF.

**3d. Remove  $A \rightarrow S$**

$$\begin{aligned} S' &\rightarrow ASA|aB|a|AS|SA \\ S &\rightarrow ASA|aB|a|AS|SA \\ A &\rightarrow b|ASA|aB|a|AS|SA \\ B &\rightarrow b \end{aligned}$$

**4a. Remove long rules**

$$\begin{aligned} S' &\rightarrow ASA|aB|a|AS|SA \\ S &\rightarrow ASA|aB|a|AS|SA \\ A &\rightarrow b|ASA|aB|a|AS|SA \\ B &\rightarrow b \end{aligned}$$

**4b. Remove long rules**

$$\begin{aligned} S' &\rightarrow AU|aB|a|AS|SA \\ S &\rightarrow AU|aB|a|AS|SA \\ A &\rightarrow b|AU|aB|a|AS|SA \\ U &\rightarrow SA \\ B &\rightarrow b \end{aligned}$$

**4c. Remove long rules**

$$\begin{aligned} S' &\rightarrow AU|VB|a|AS|SA \\ S &\rightarrow AU|VB|a|AS|SA \\ A &\rightarrow b|AU|VB|a|AS|SA \\ U &\rightarrow SA \\ V &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

There are other rules of the form: Var  $\rightarrow ASA$

# Chomsky Normal Form

**CNF:**

$A \rightarrow BC$  [ $B, C$  are not start variables]

$A \rightarrow a$  [ $a$  is a terminal]

$S \rightarrow \epsilon$  [ $S$  is the Start Variable]

Convert the CFG

$S \rightarrow ASA|aB$

$A \rightarrow B|S$

$B \rightarrow b|\epsilon$

to CNF.

$S' \rightarrow AU|VB|a|AS|SA$

$S \rightarrow AU|VB|a|AS|SA$

$A \rightarrow b|AU|VB|a|AS|SA$

$U \rightarrow SA$

$V \rightarrow a$

$B \rightarrow b$

# Pushdown Automata

- For regular languages we had
  - Designed Finite automata (DFA, NFA) that recognize the strings by the language. Helped us decide whether a given string  $\omega$  belongs to the language.

# Pushdown Automata

- For regular languages we had
  - Designed Finite automata (DFA, NFA) that recognize the strings by the language. Helped us decide whether a given string  $\omega$  belongs to the language.
  - Developed regular expressions/linear grammar that can generate all the strings in the language.

# Pushdown Automata

- For regular languages we had
  - Designed Finite automata (DFA, NFA) that recognize the strings by the language. Helped us decide whether a given string  $\omega$  belongs to the language.
  - Developed regular expressions/linear grammar that can generate all the strings in the language.
- For context free languages,
  - Context Free Grammars generate all the strings in the language

# Pushdown Automata

- For regular languages we had
  - Designed Finite automata (DFA, NFA) that recognize the strings by the language. Helped us decide whether a given string  $\omega$  belongs to the language.
  - Developed regular expressions/linear grammar that can generate all the strings in the language.
- For context free languages,
  - Context Free Grammars generate all the strings in the language
  - Can we build an automata that recognizes **exactly** context free languages?

# Pushdown Automata

- For regular languages we had
  - Designed Finite automata (DFA, NFA) that recognize the strings by the language. Helped us decide whether a given string  $\omega$  belongs to the language.
  - Developed regular expressions/linear grammar that can generate all the strings in the language.
- For context free languages,
  - Context Free Grammars generate all the strings in the language
  - Can we build an automata that recognizes **exactly** context free languages?
- **Finite Automaton model recognizes ALL regular languages**
- Any automata that recognizes **ALL** context free languages will need unbounded memory.

# Pushdown Automata

- Finite Automaton model recognizes ALL regular languages
- Any automata that recognizes **ALL** context free languages will need unbounded memory.

## Intuition to build an Automata for CFL

- It should be some **Finite State Machine** that has access to a memory device with infinite memory, i.e.

**Automata for CFL = FSM + Memory device**

- **FSM may choose to ignore the memory device** completely in which case it behaves like a DFA/NFA.
- FSM makes use of the Memory device to recognize “non-Regular” CFLs.

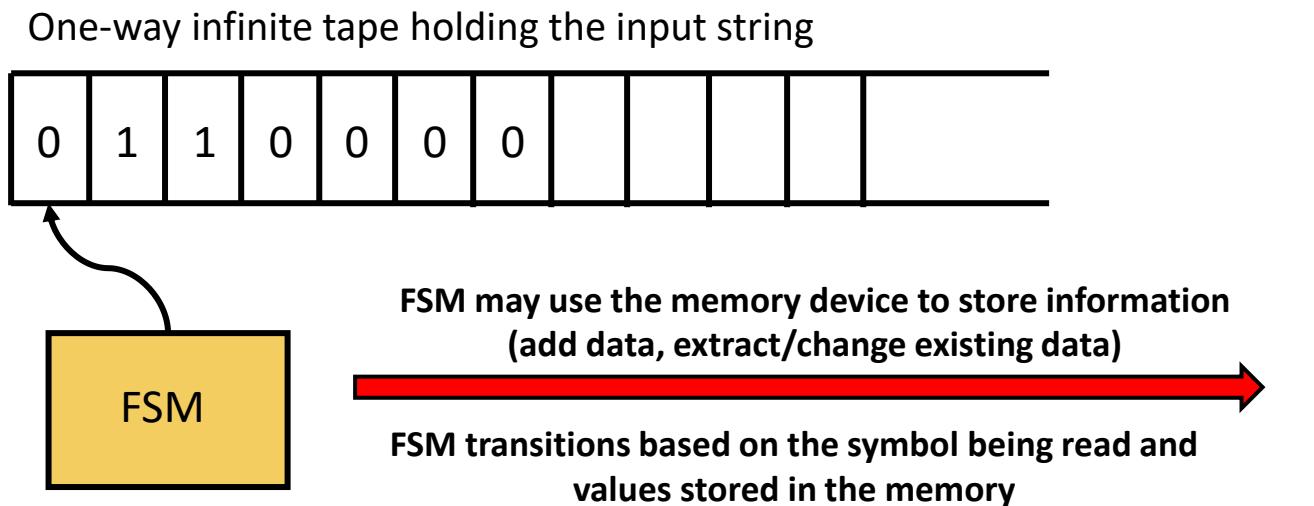
E.g.:  $\{0^n 1^n, n \in \mathbb{N}\}$

# Pushdown Automata

## Intuition to build an Automata for CFL

- **Automata for CFL = FSM + Memory device**
- **FSM may choose to ignore the memory device** completely in which case it behaves like a DFA/NFA.
- FSM makes use of the Memory device to recognize “non-Regular” CFLs.

E.g.:  $\{0^n 1^n, n \in \mathbb{N}\}$



**Thank You!**

# CS 302.1 - Automata Theory

## Lecture 06

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

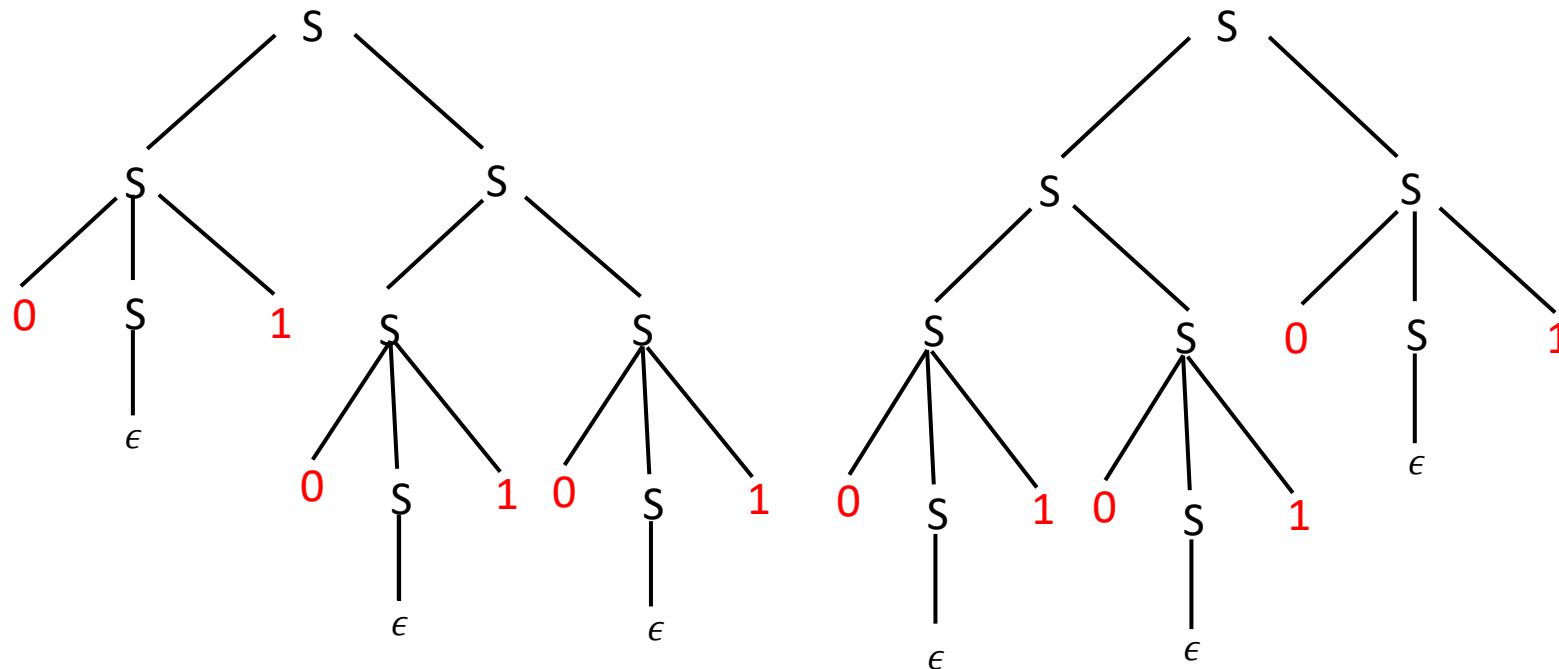
Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad

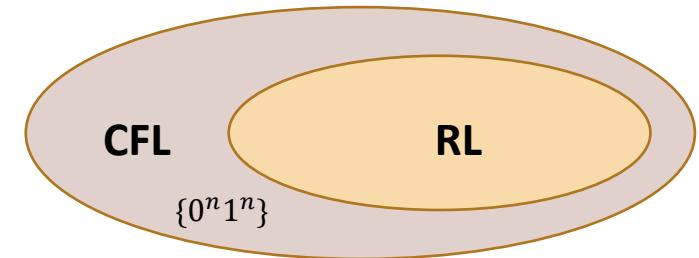
# Quick Recap

**Context-Free Grammars:** If the *rules* of the underlying grammar  $G$  are of the form  
$$V \rightarrow (VUT)^*$$

then such a grammar is called **Context-Free**.



**Parse trees:** These are ordered trees that provide alternative representations of the derivation of a grammar.



**Ambiguous grammars:** There exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** . Ambiguity may not be desirable

# Quick Recap

**Chomsky Normal Form:** If every *rule* of the CFG is of the form

$$A \rightarrow BC$$

[ $B, C$  are not start variables]

$$A \rightarrow a$$

[ $a$  is a terminal]

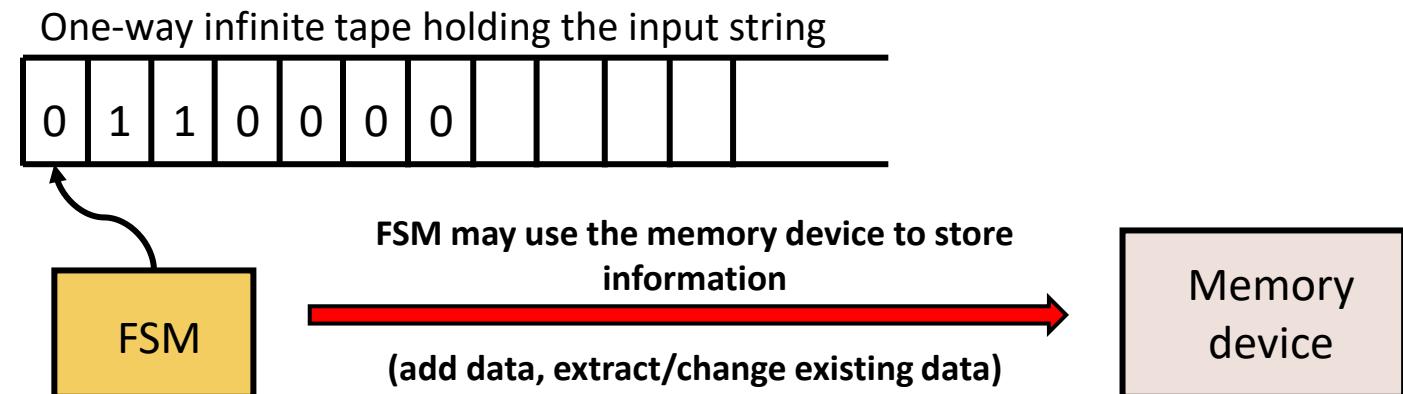
$$S \rightarrow \epsilon$$

[ $S$  is the Start Variable]

- Any CFG can be converted to a grammar in CNF that generates the same language.
- The number of steps required to derive a string  $w = 2|w| - 1$ .
- Is crucial in deciding whether  $w$  is generated by a CFG  $G$ .

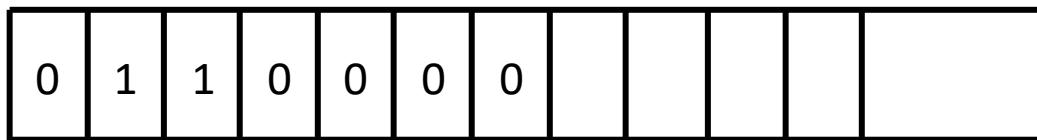
## Pushdown Automata

- Automata that recognizes CFLs
- FSM + memory device
- FSM transitions by reading an input symbol and by interacting with the device



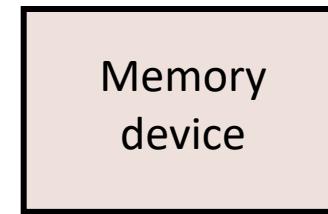
# Pushdown Automata

One-way infinite tape holding the input string



FSM may use the memory device to store information  
(add data, extract/change existing data)

FSM transitions based on the symbol being read and  
values stored in the memory



## The memory device

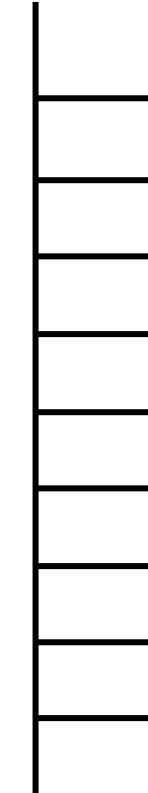
- Simple memory device with unbounded memory.

# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, new elements can be added to the Stack (**PUSH**).
- At any stage, the element at the **top** of the STACK can be read by removing it from the stack (**POP**).

Memory  
device



# Pushdown Automata

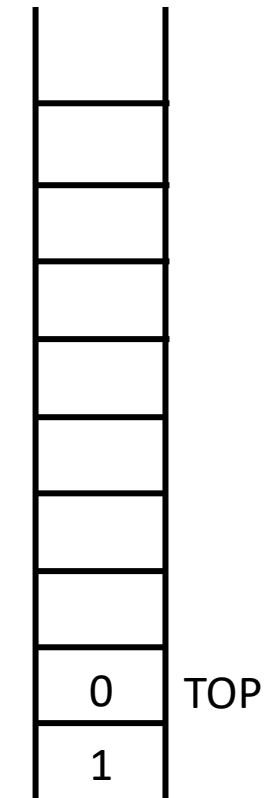
## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

### PUSH

- New symbols can be **pushed** in to the STACK.  
E.g: PUSH 1
- The Top of the STACK now covers the old stack top, i.e.  
 $\text{TOP} = \text{TOP} + 1$
- The size of the stack keeps growing.

Memory  
device



# Pushdown Automata

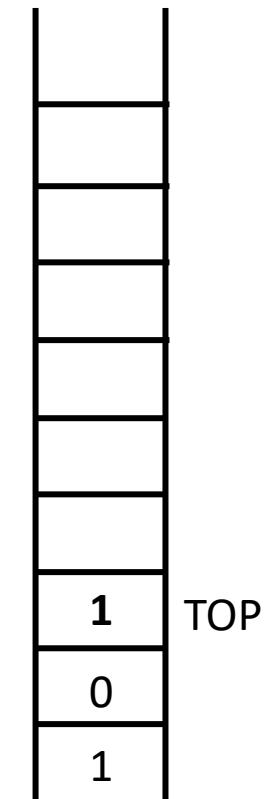
## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

### PUSH

- New symbols can be **pushed** in to the STACK.  
E.g: PUSH 1
- The Top of the STACK now covers the old stack top, i.e.  
 $\text{TOP} = \text{TOP} + 1$
- The size of the stack keeps growing.

Memory  
device



# Pushdown Automata

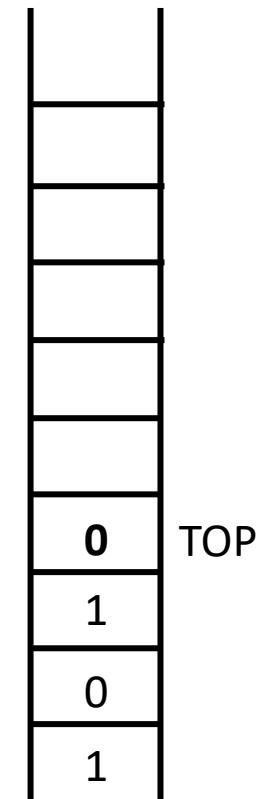
## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

### PUSH

- New symbols can be **pushed** in to the STACK.  
  
E.g: **PUSH 0**  
  
• The Top of the STACK now covers the old stack top, i.e.  
  
$$\text{TOP} = \text{TOP} + 1$$
  
  
• The size of the stack keeps growing.

Memory  
device



# Pushdown Automata

## The memory device

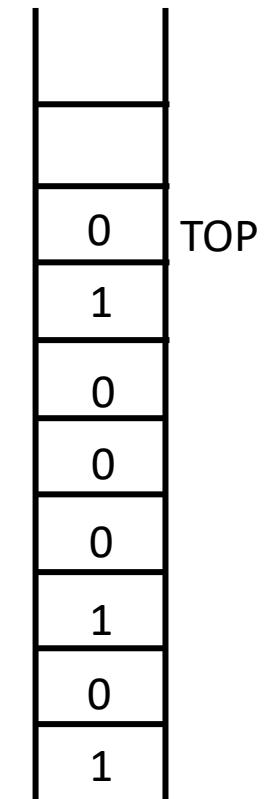
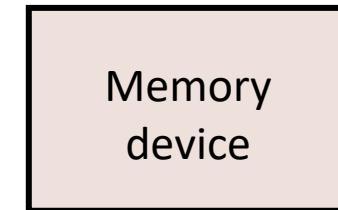
- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

### PUSH

- New symbols can be **pushed** in to the STACK.
- The Top of the STACK now covers the old stack top, i.e.

$$\text{TOP} = \text{TOP} + 1$$

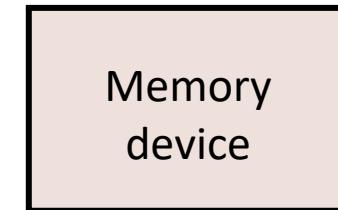
- The size of the stack keeps growing.



# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.



### POP

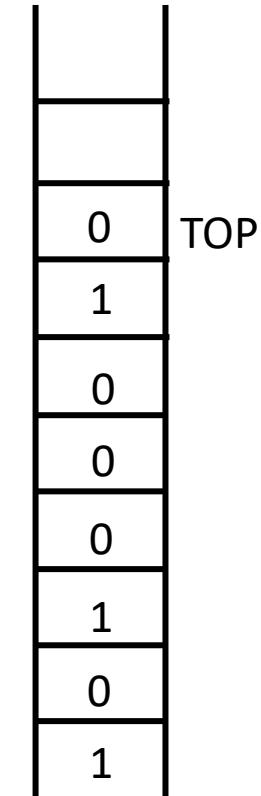
- The element from the **TOP** of the stack can be **popped** out

E.g.: **POP 0**

- The Top of the STACK moves to the element below.

$$\text{TOP} = \text{TOP} - 1$$

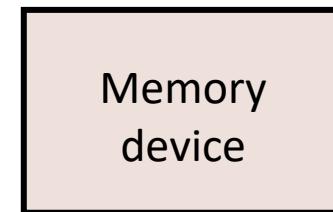
- Successive **POP** operations shrink the stack size. Elements can be popped until **EMPTY**.
- **Last In First Out (LIFO)**: The last element that was pushed is the first to be popped out



# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.



### POP

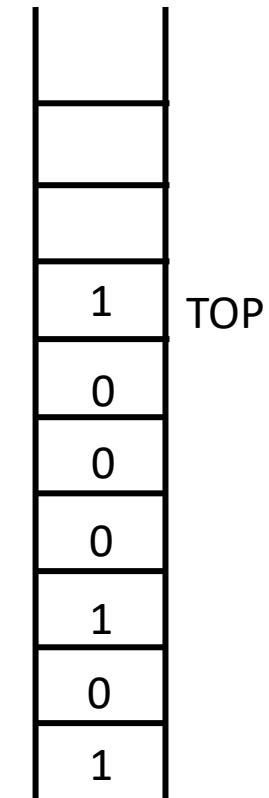
- The element from the **TOP** of the stack can be **popped** out

E.g.: **POP 0**

- The Top of the STACK moves to the element below.

$$\text{TOP} = \text{TOP} - 1$$

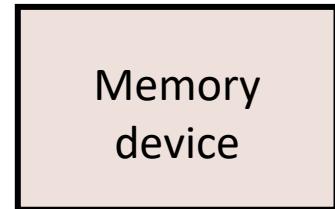
- Successive **POP** operations shrink the stack size. Elements can be popped until **EMPTY**.
- **Last In First Out (LIFO)**: The last element that was pushed is the first to be popped out



# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.



### POP

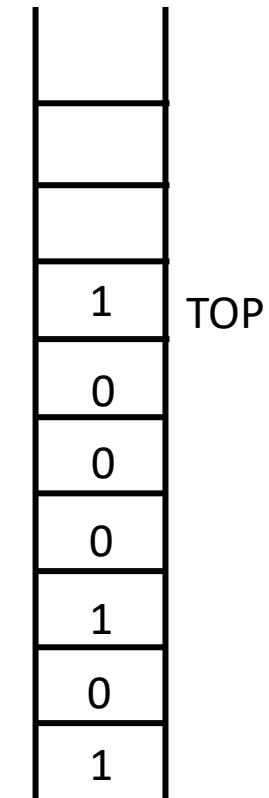
- The element from the **TOP** of the stack can be **popped** out

E.g.: **POP 1**

- The Top of the STACK moves to the element below.

$$\text{TOP} = \text{TOP} - 1$$

- Successive **POP** operations shrink the stack size. Elements can be popped until **EMPTY**.
- **Last In First Out (LIFO)**: The last element that was pushed is the first to be popped out



# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

Memory  
device

### POP

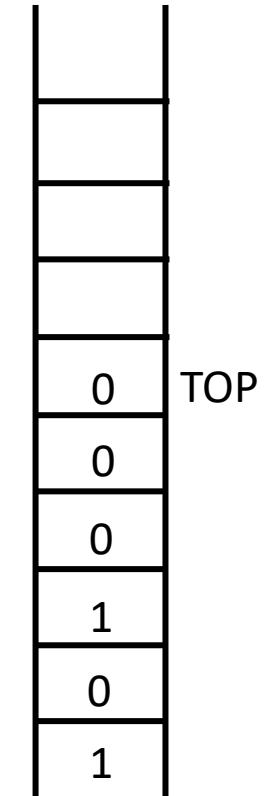
- The element from the **TOP** of the stack can be **popped** out

E.g.: **POP 1**

- The Top of the STACK moves to the element below.

$$\text{TOP} = \text{TOP} - 1$$

- Successive **POP** operations shrink the stack size. Elements can be popped until **EMPTY**.
- **Last In First Out (LIFO)**: The last element that was pushed is the first to be popped out



# Pushdown Automata

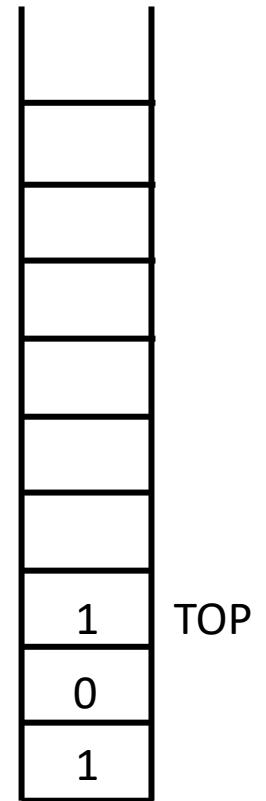
## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- Last In First Out (**LIFO**)

Memory  
device

## POP

- The element from the **TOP** of the stack can be **popped** out.
  - $\text{TOP} = \text{TOP} - 1$
  - Elements can be popped until STACK is **EMPTY**.
- 
- How would you know that the STACK is **EMPTY**?



# Pushdown Automata

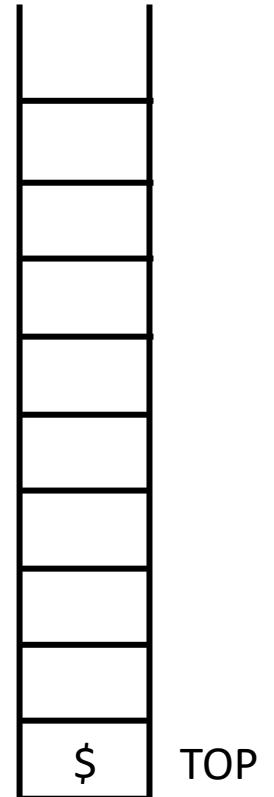
## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- Last In First Out (**LIFO**)

Memory  
device

## POP

- The element from the **TOP** of the stack can be **popped** out.
- $\text{TOP} = \text{TOP} - 1$
- Elements can be popped until STACK is **EMPTY**.
- How would you know that the STACK is **EMPTY**?
- There is generally some special symbol (say  $\$$ ) that demarcates the bottom of the STACK.
- This element is Pushed at the very beginning. Whenever  $\text{TOP} = \$$ , the STACK is **EMPTY**.



# Pushdown Automata

## Memory device of PDA: STACK

- STACK is a **LIFO** data structure of unbounded memory
- Only the TOP element can be read from the STACK.
- The bottom of the STACK contains a special symbol (\$)
- Characterized by two operations:

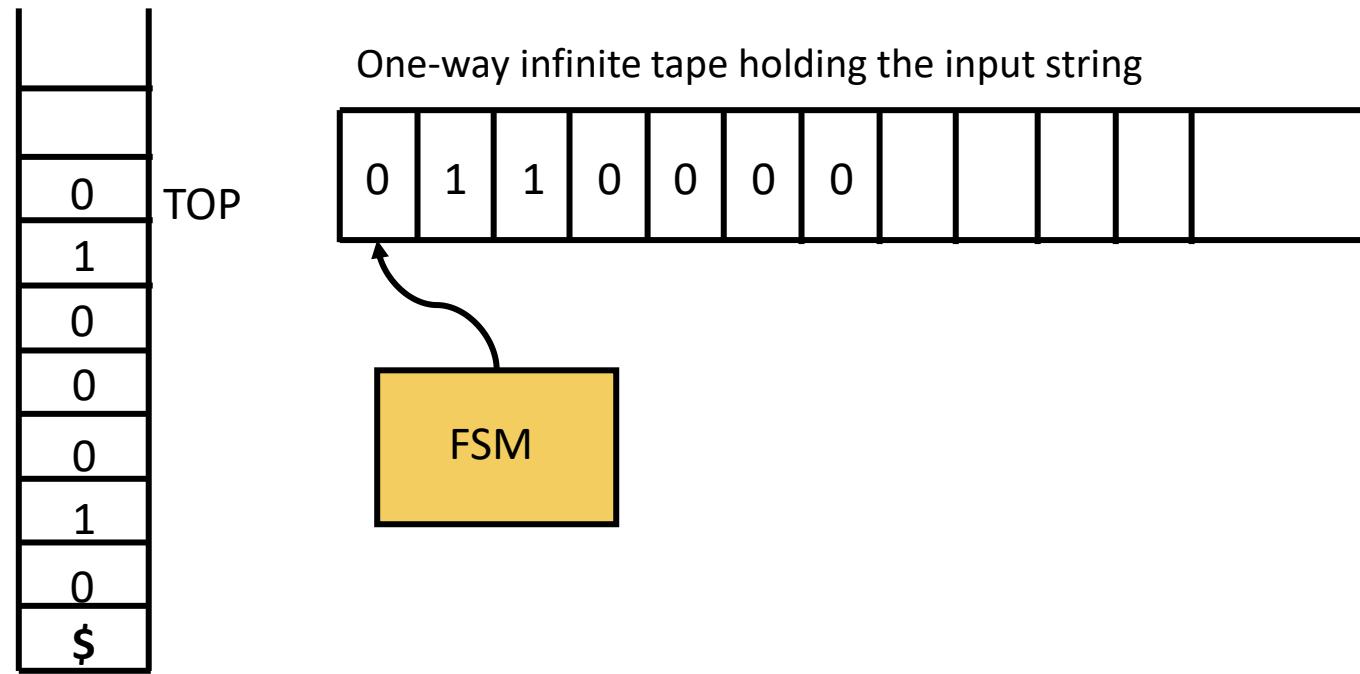
### PUSH

- New symbols can be **pushed** in to the STACK.
- $\text{TOP} = \text{TOP} + 1$

### POP

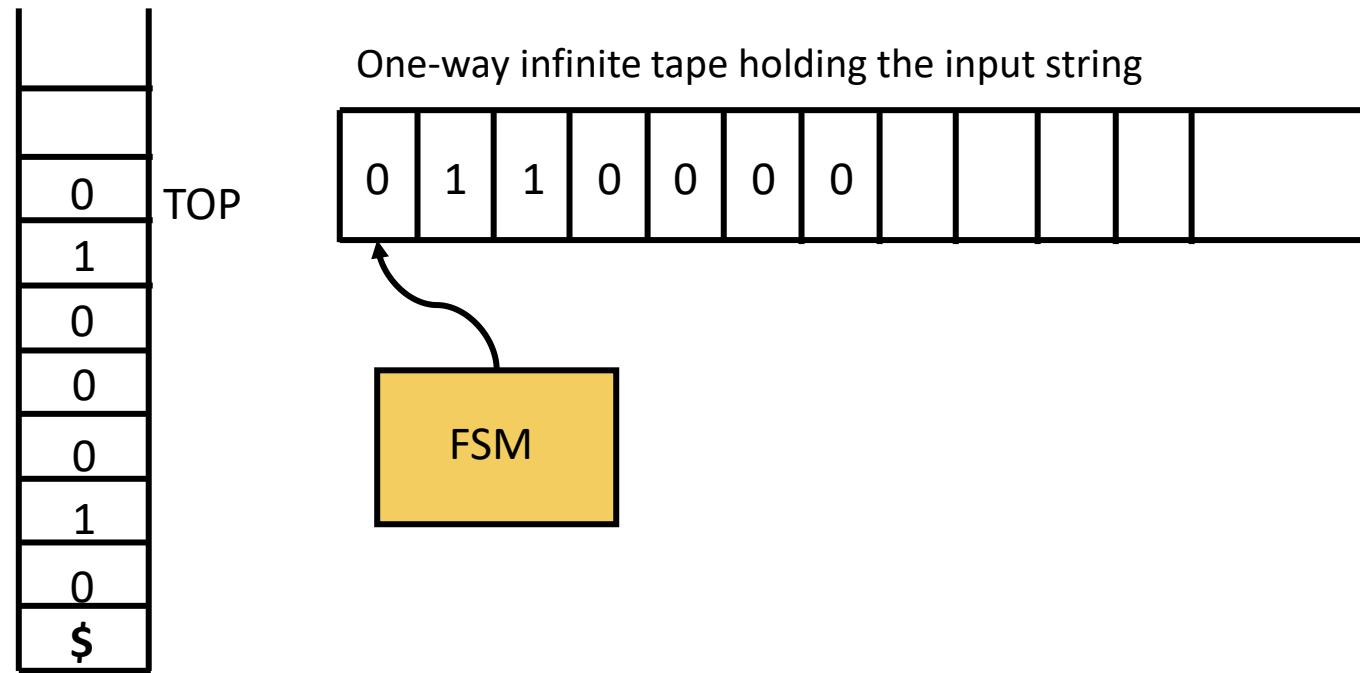
- The element from the TOP of the stack can be **popped** out.
- $\text{TOP} = \text{TOP} - 1$
- Elements can be popped until STACK is EMPTY.

# Pushdown Automata



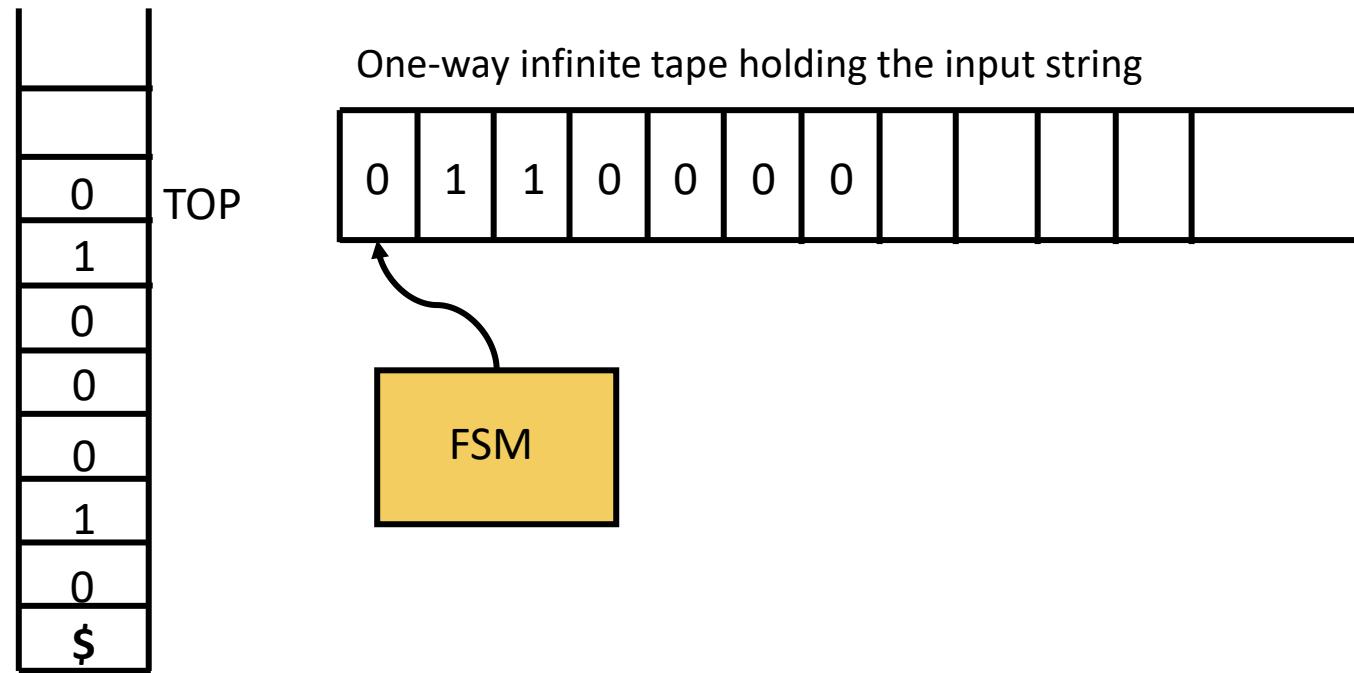
- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:

# Pushdown Automata



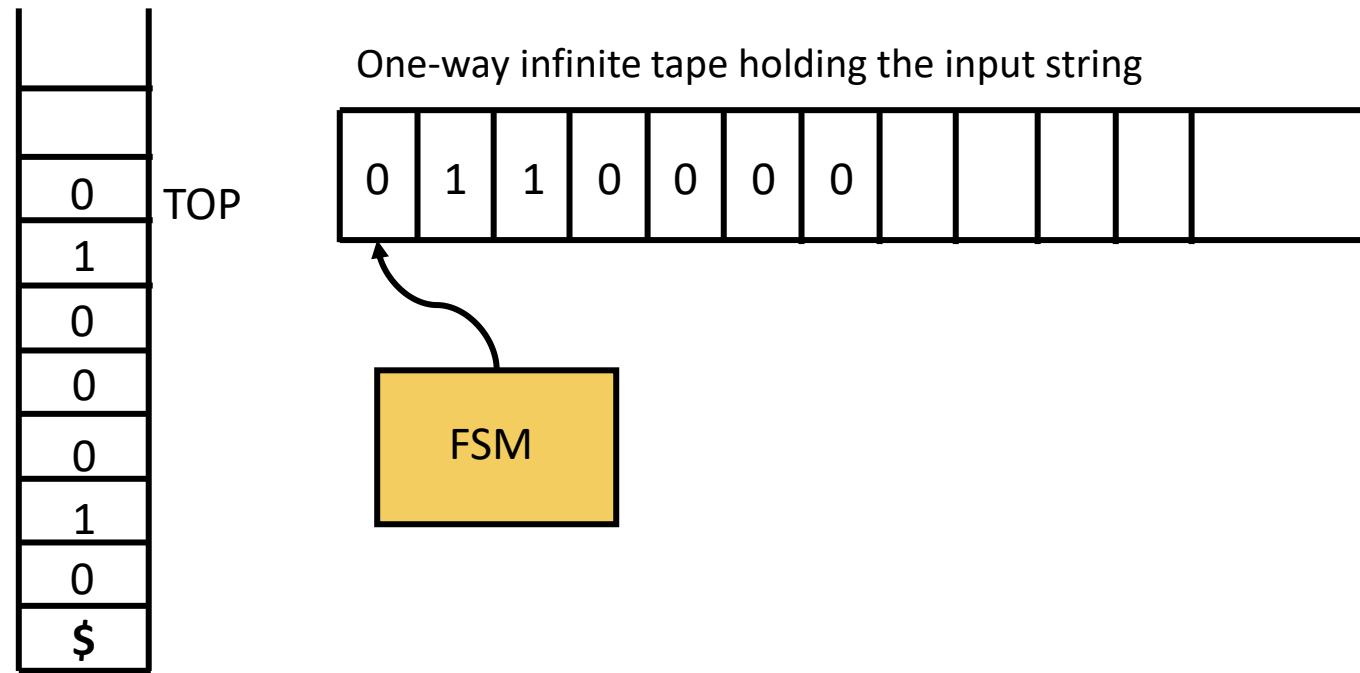
- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:
  - Transitions based on the Input symbol and the element at the top of the stack (e.g.: If I/P symbol = 0 & TOP = 0, transition from  $i$  to  $j$ )

# Pushdown Automata



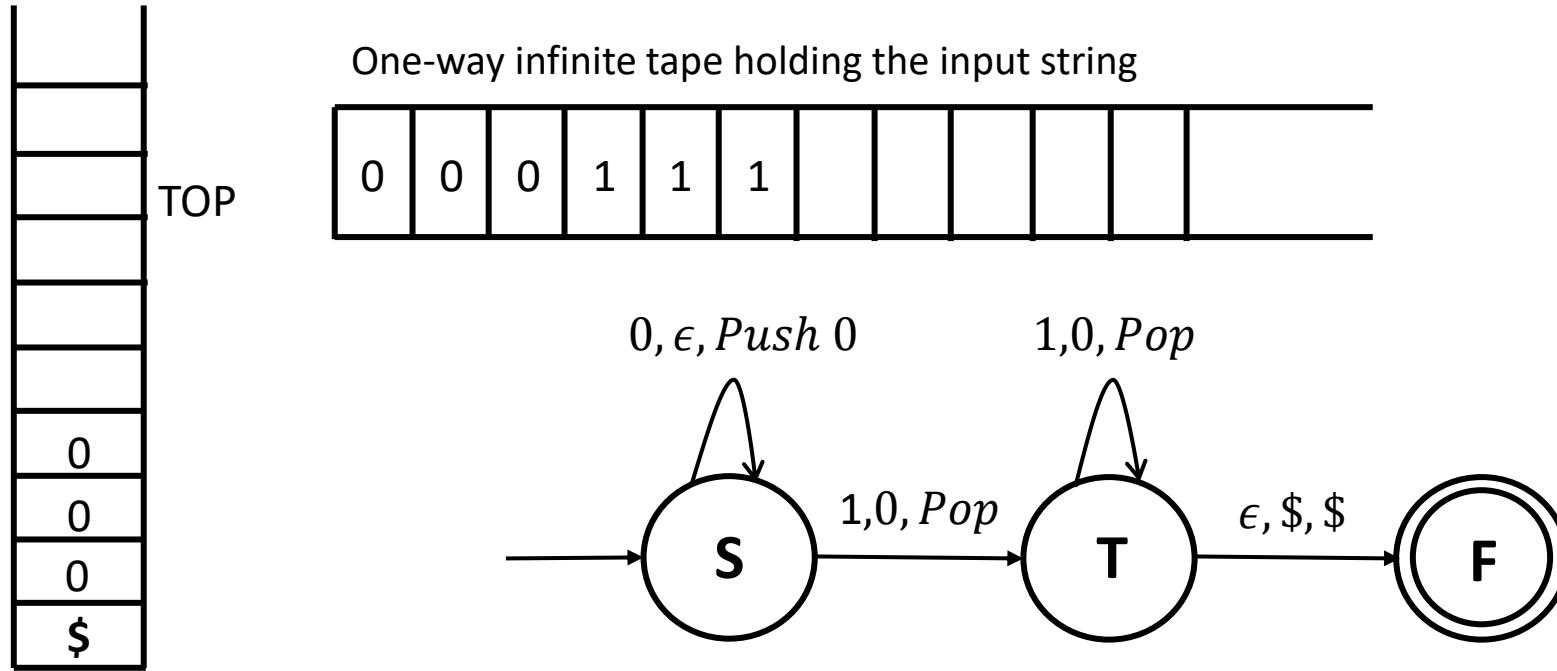
- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:
  - Transitions based on the Input symbol and the element at the top of the stack (e.g.: If I/P symbol = 0 & TOP = 0, transition from  $i$  to  $j$ ).
  - **How can we read the TOP? By popping**

# Pushdown Automata



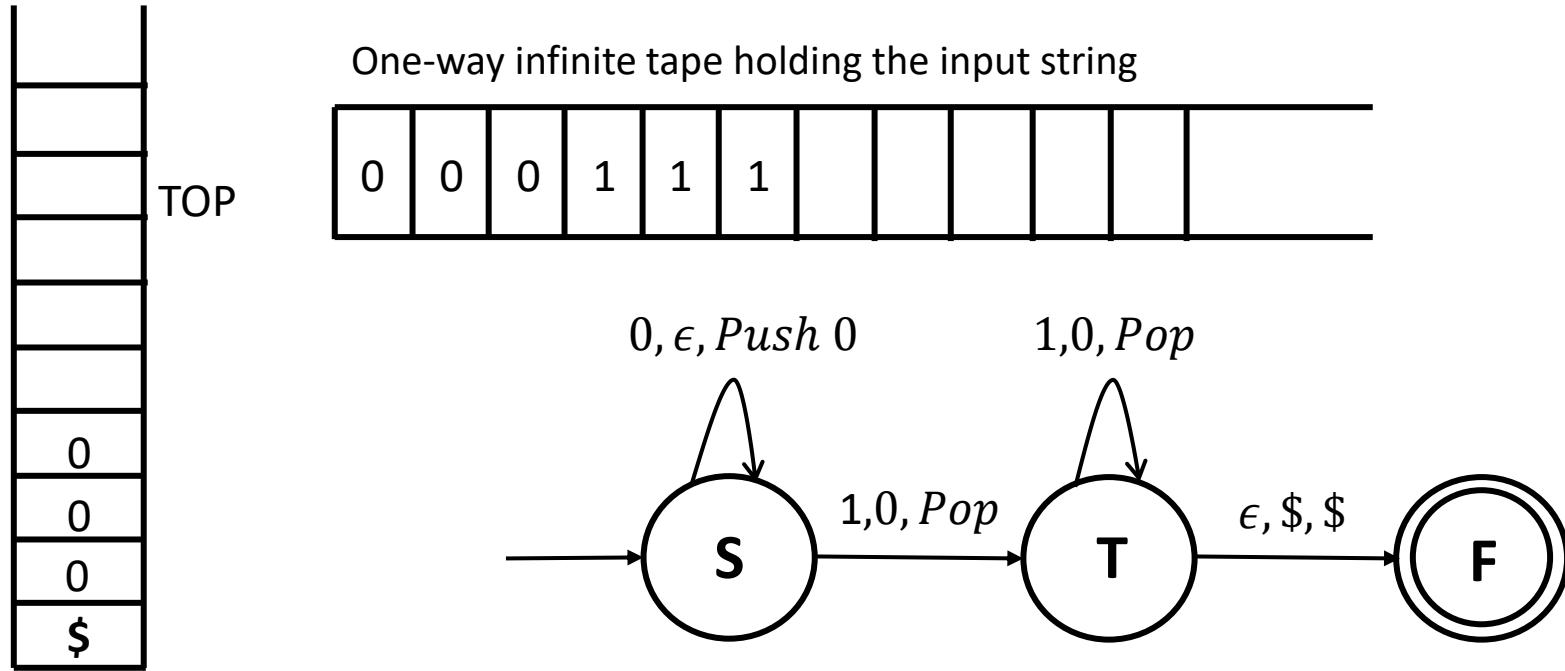
- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:
  - Transitions based on the Input symbol and the element at the top of the stack (e.g.: If I/P symbol = 0 & TOP = 0, transition from  $i$  to  $j$ ) – **Pop 0**
  - Pushes new elements into the Stack (e.g.: If I/P symbol = 0, PUSH 0, transition from  $i$  to  $j$ ).

# Pushdown Automata



- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:
  - Transitions based on the Input symbol and the element at the top of the stack
  - Pops the element at the top of the Stack.
  - Pushes new elements into the Stack.

# Pushdown Automata

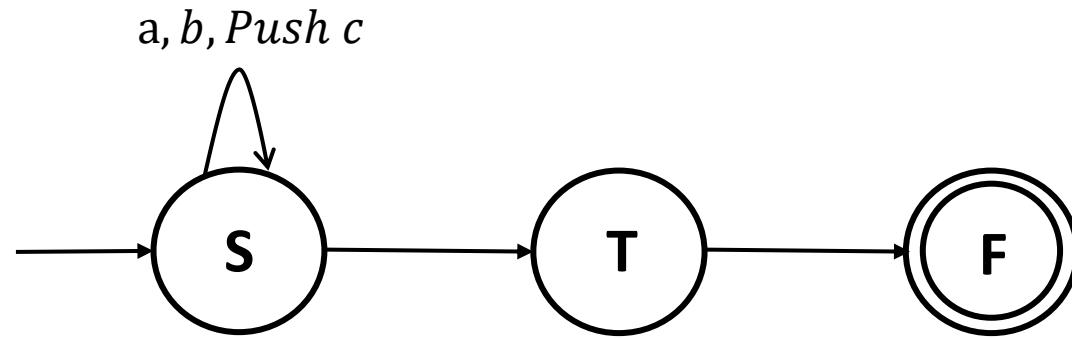


PDAs are **non-deterministic**.

- $\epsilon$ -transitions
- Multiple transitions/input symbol possible

# Pushdown Automata

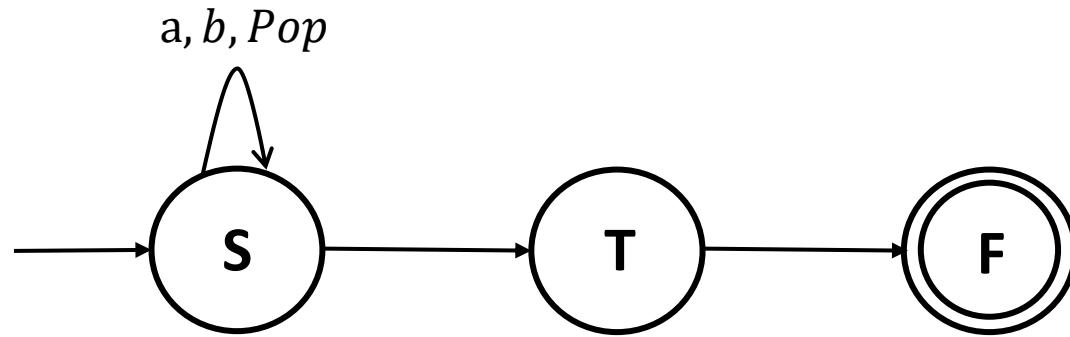
- How to represent a transition in a PDA?



If input symbol =  $a$ , Stack top =  $b$  (if  $b$  is popped) and Push  $c$  onto the Stack

# Pushdown Automata

- How to represent a transition in a PDA?

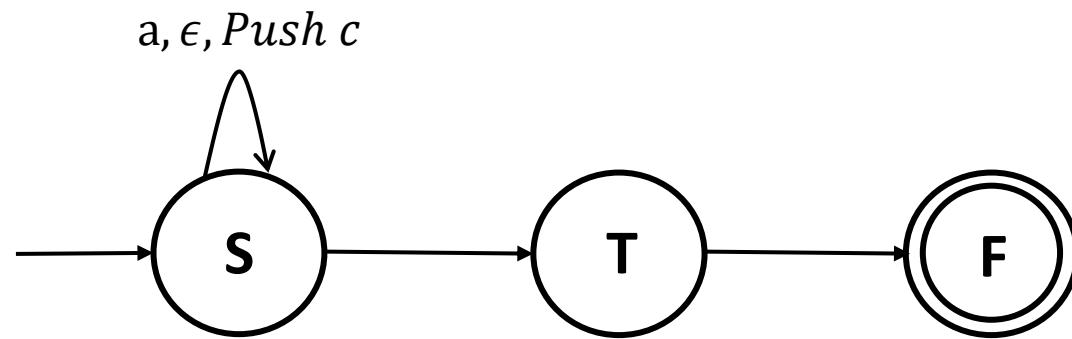


If input symbol =  $a$ , then Pop  $b$

(If the symbol read is  $a$  and the stack TOP =  $b$ , then remain in  $S$ )

# Pushdown Automata

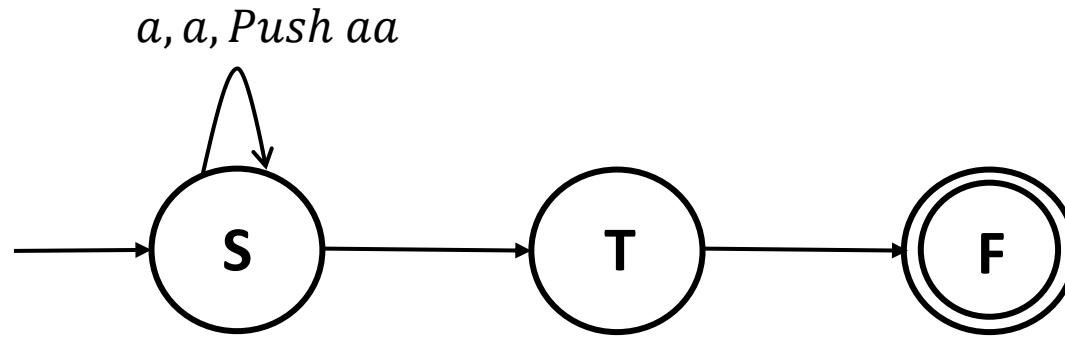
- How to represent a transition in a PDA?



If input symbol =  $a$ , then Push  $c$

# Pushdown Automata

- How to represent a transition in a PDA?



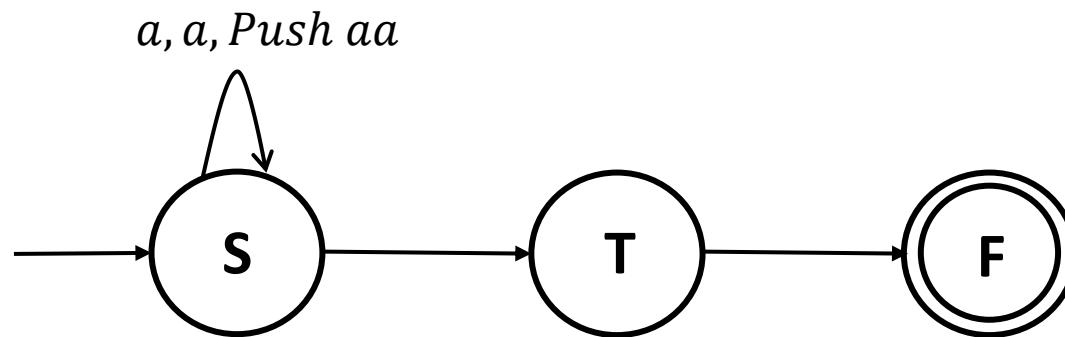
If input symbol =  $a$ , then Pop  $a$  and Push  $aa$ .

So effectively, the PDA pushes  $a$  onto the stack if it reads  $a$  on the input tape and the stack top =  $a$ .

This is a “shorthand” for describing two operations:

# Pushdown Automata

- How to represent a transition in a PDA?



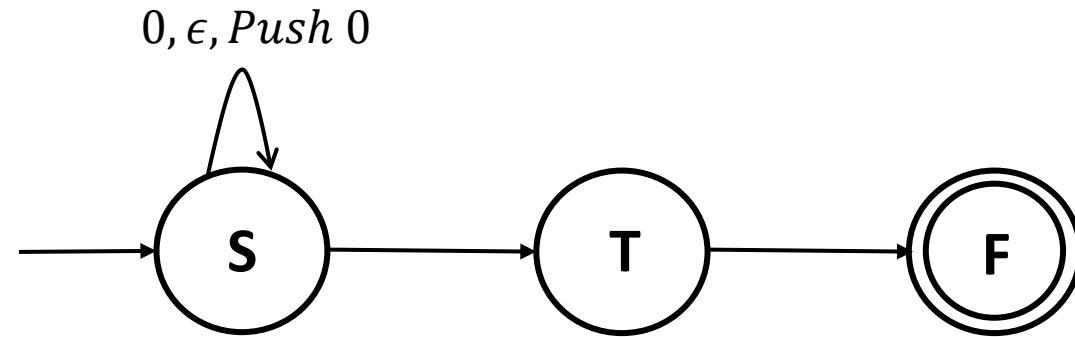
If input symbol =  $a$ , then Pop  $a$  and Push  $aa$ .

So effectively, the PDA pushes  $a$  onto the stack if it reads  $a$  on the input tape and the stack top =  $a$ .



# Pushdown Automata

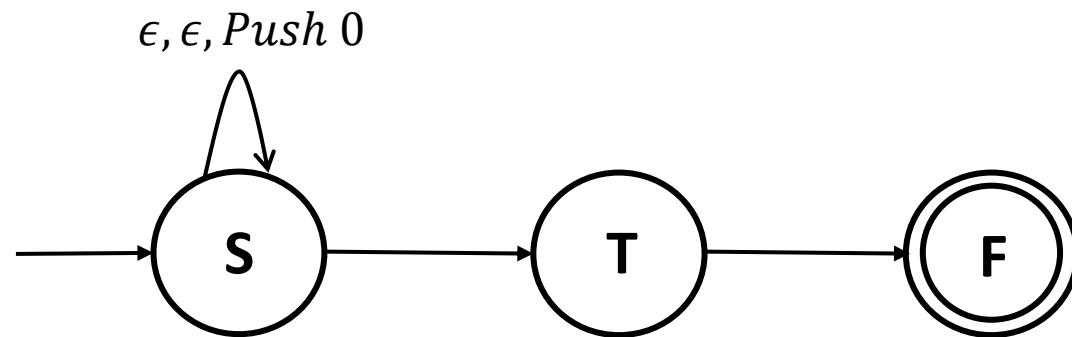
- How to represent a transition in a PDA?



If input symbol = 0, Push 0 onto the Stack irrespective of the element at the top of the stack

# Pushdown Automata

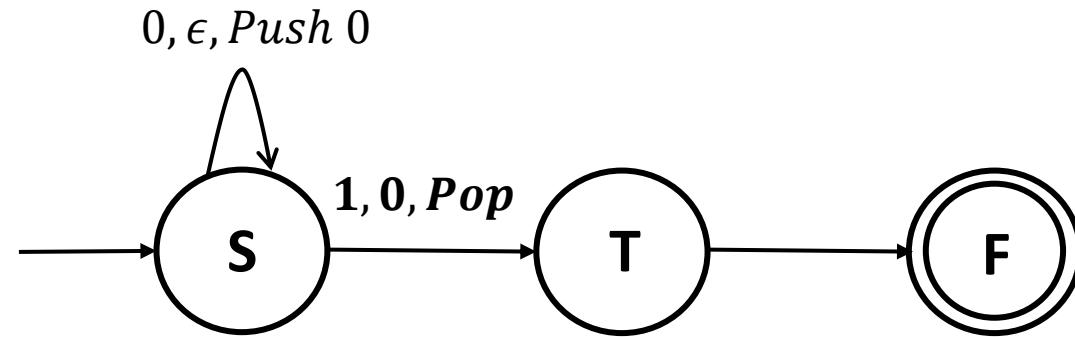
- How to represent a transition in a PDA?



Without reading the input symbol and the Stack top, Push 0 onto the Stack

# Pushdown Automata

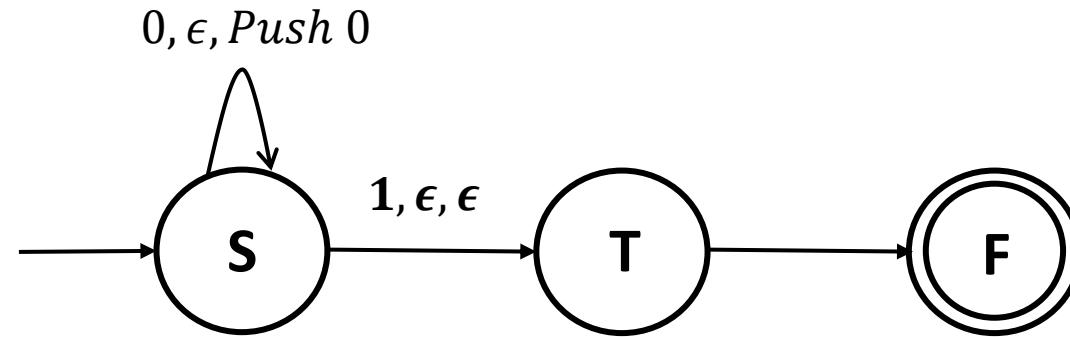
- How to represent a transition in a PDA?



If the input symbol is 1, and the element at the top of the stack is 0 (**Pop 0**), then transition from  $S$  to  $T$

# Pushdown Automata

- How to represent a transition in a PDA?

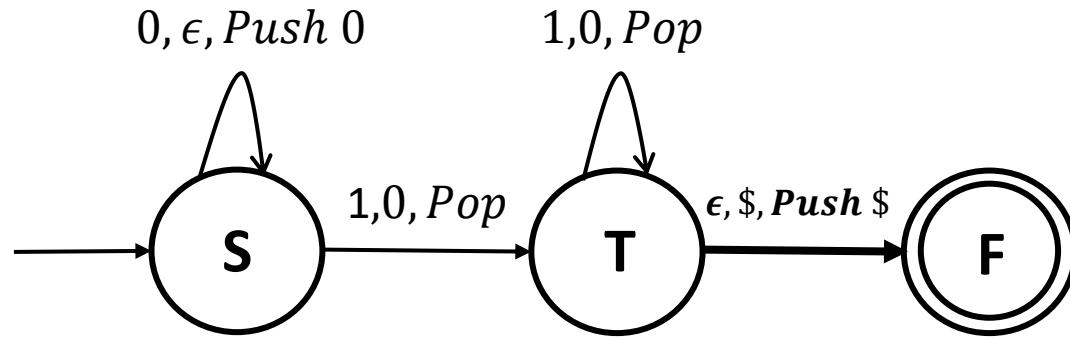


If the input symbol is 1, transition to *T* by ignoring the stack top completely.

If this happens at every step of the execution of the PDA, then it is as powerful as an NFA.

# Pushdown Automata

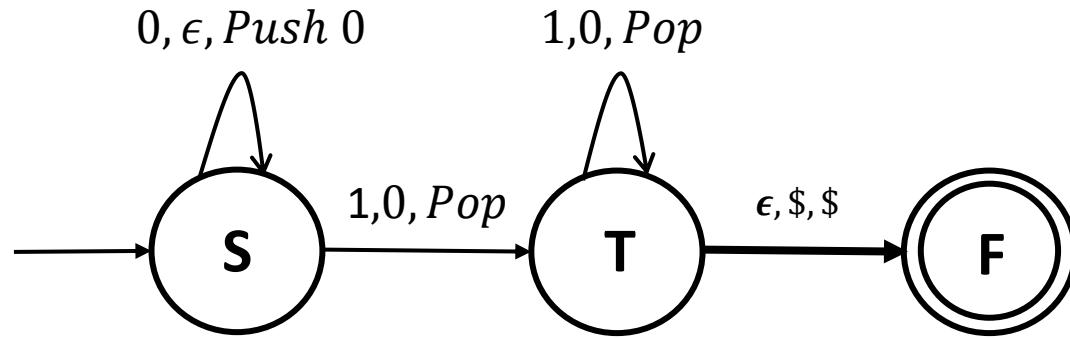
- How to represent a transition in a PDA?



If the Stack is empty, i.e.  $\text{TOP} = \$$ , transition to  $F$  from  $T$ , without reading the input

# Pushdown Automata

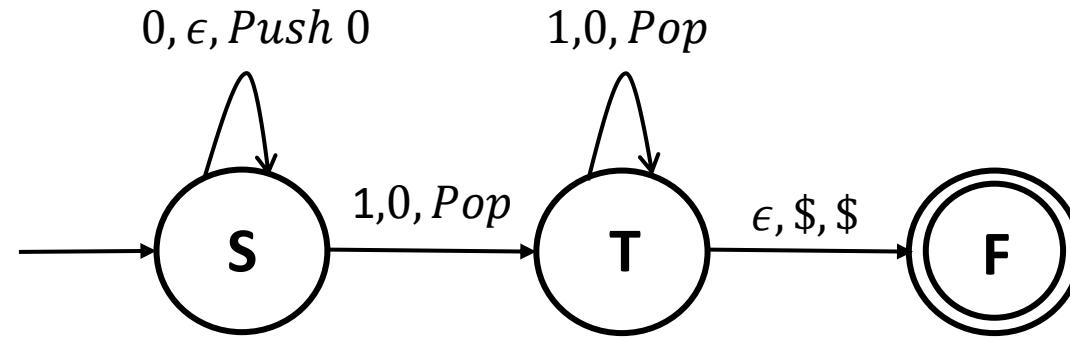
- How to represent a transition in a PDA?



If the Stack is empty, i.e.  $\text{TOP} = \$$ , transition to  $F$  from  $T$ , without reading the input

# Pushdown Automata

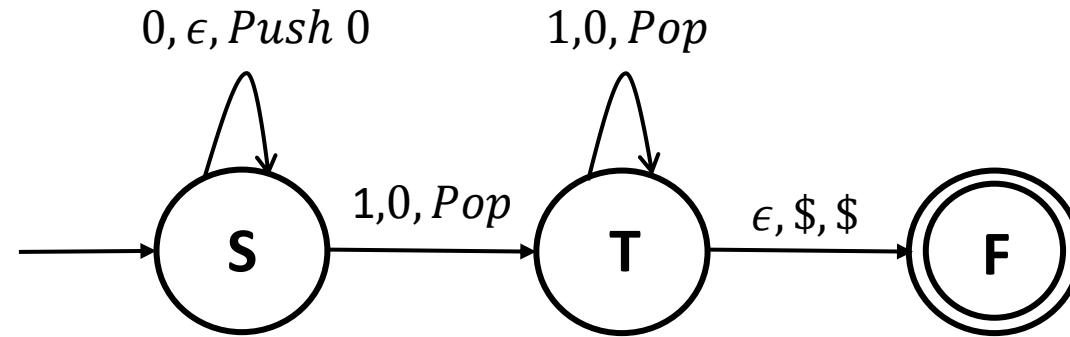
- How to represent a transition in a PDA?



What is the language accepted by this PDA?

# Pushdown Automata

- How to represent a transition in a PDA?

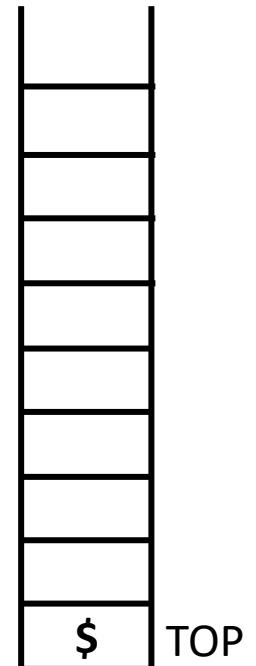
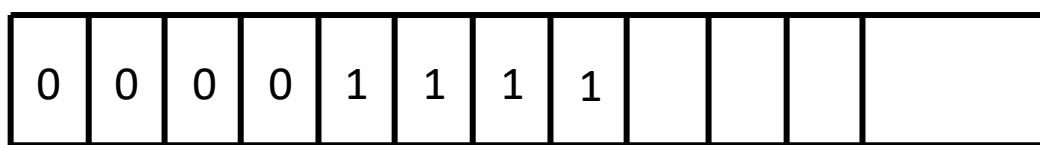
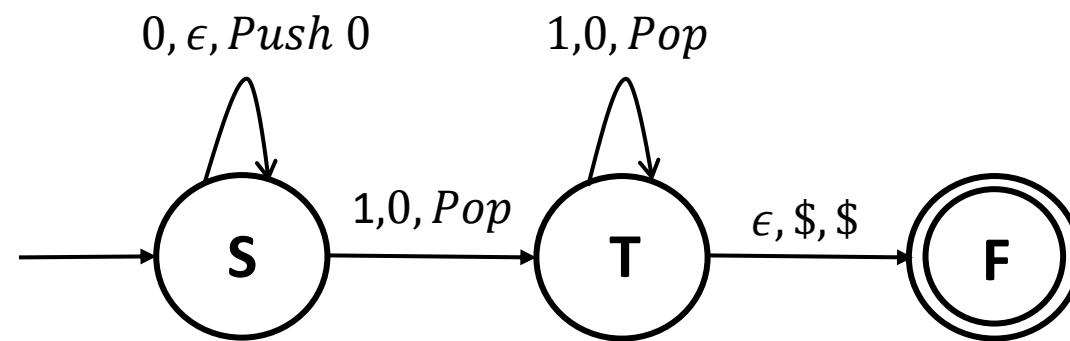


What is the language recognized by this PDA?

Verify that it is  $L = \{0^n 1^n, n \geq 1\}$

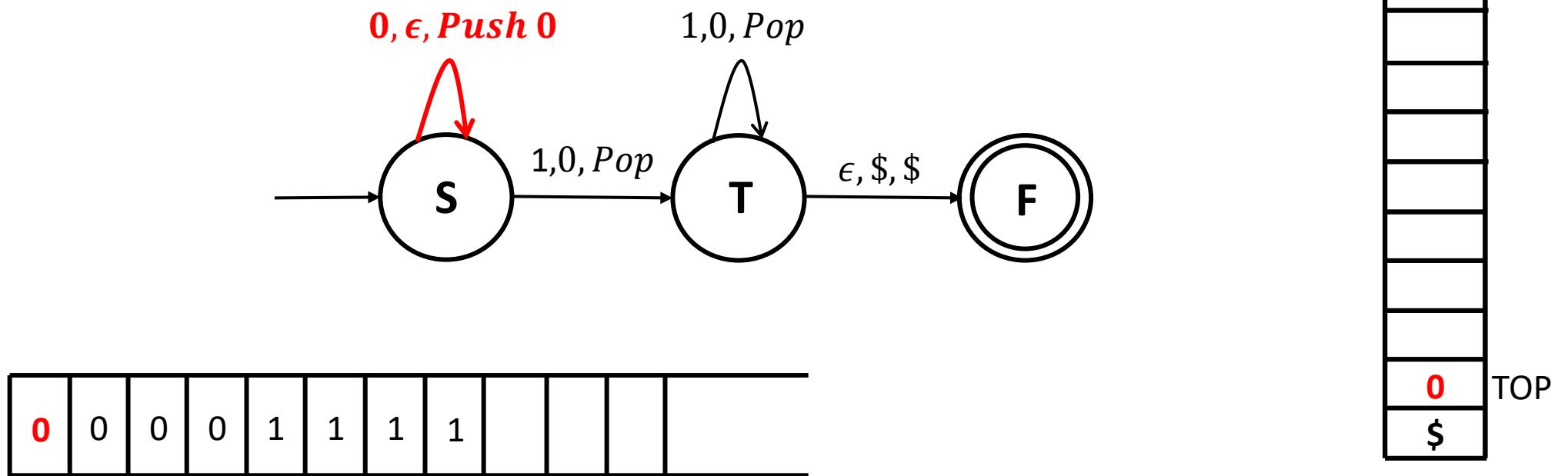
# Pushdown Automata

What is the language recognized by this PDA?



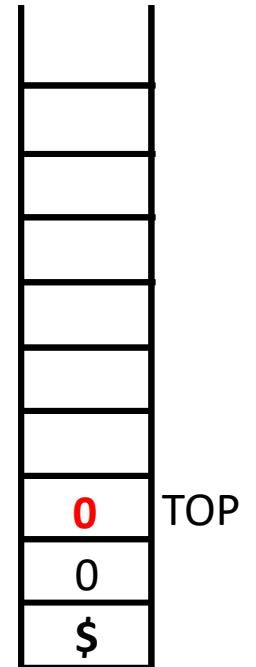
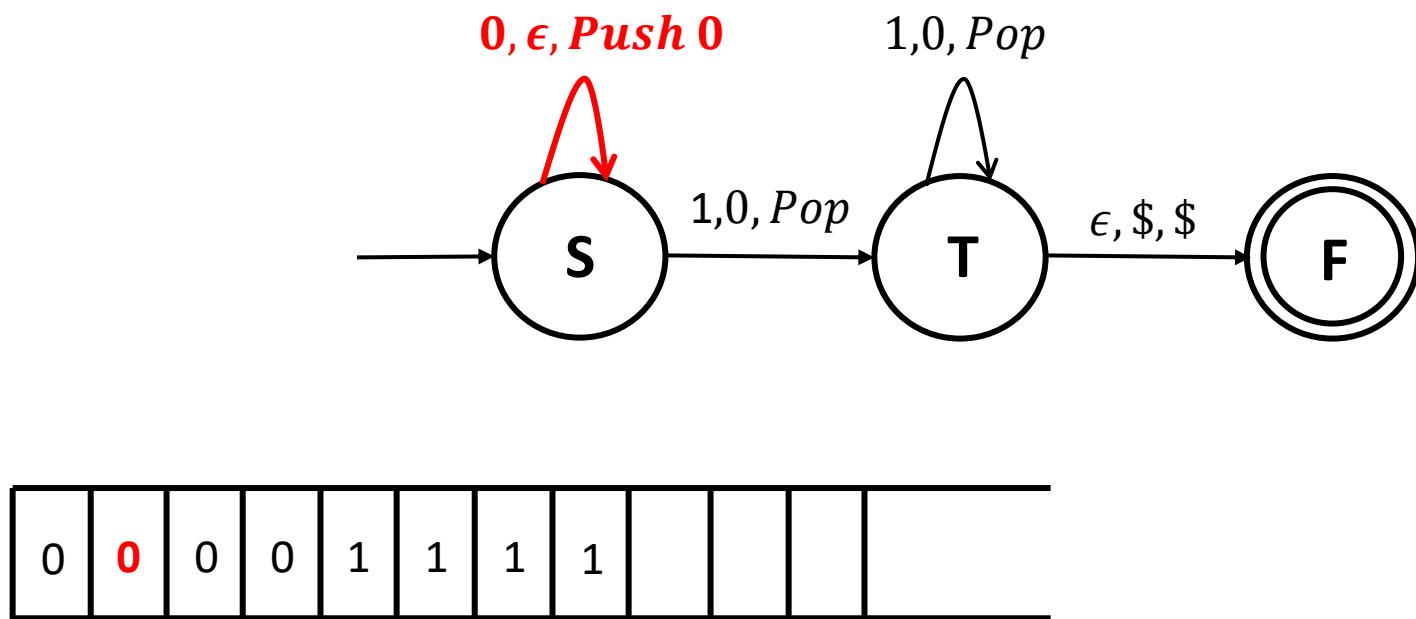
# Pushdown Automata

What is the language recognized by this PDA?



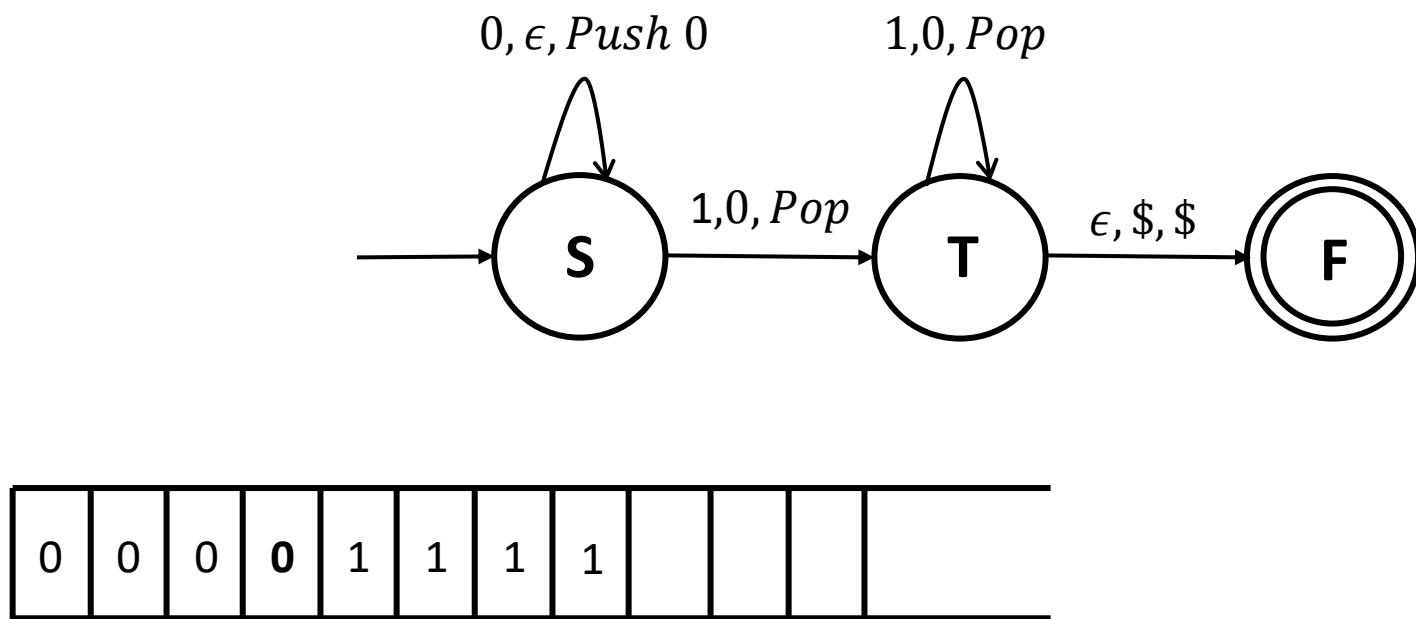
# Pushdown Automata

What is the language recognized by this PDA?



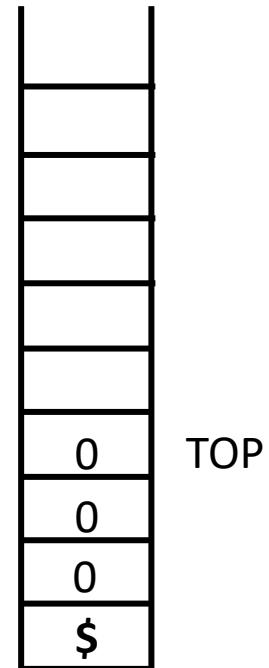
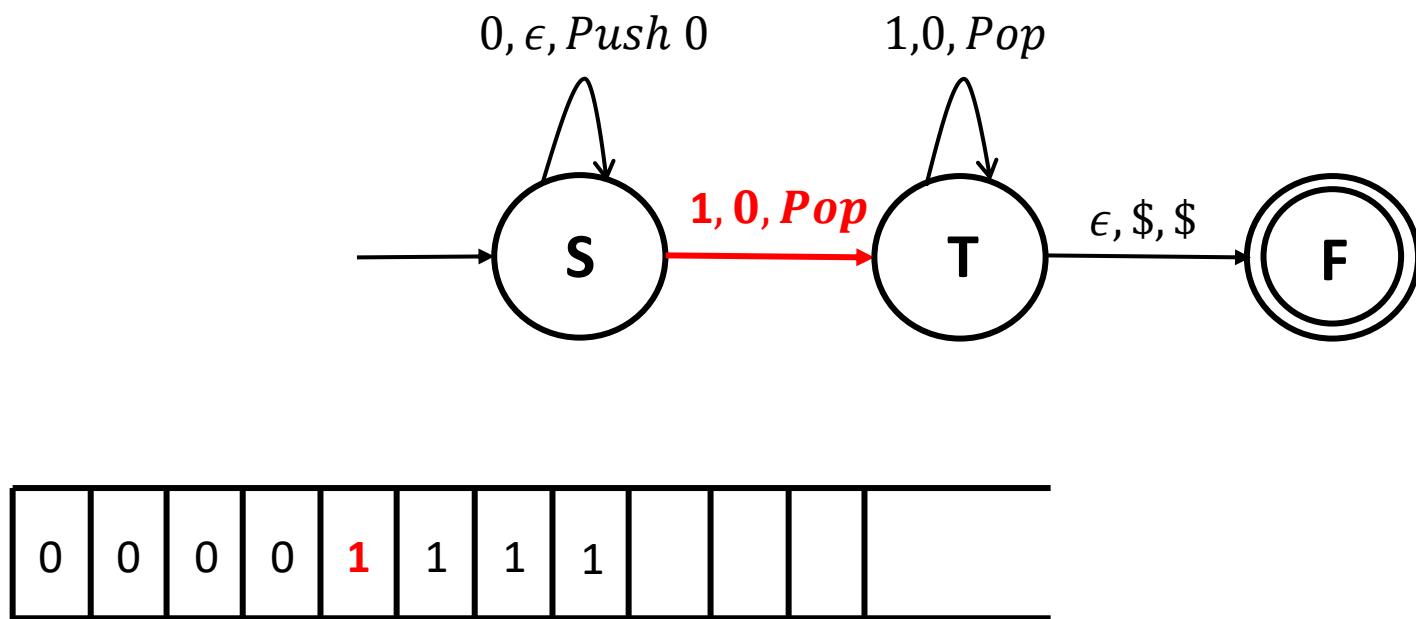
# Pushdown Automata

What is the language recognized by this PDA?



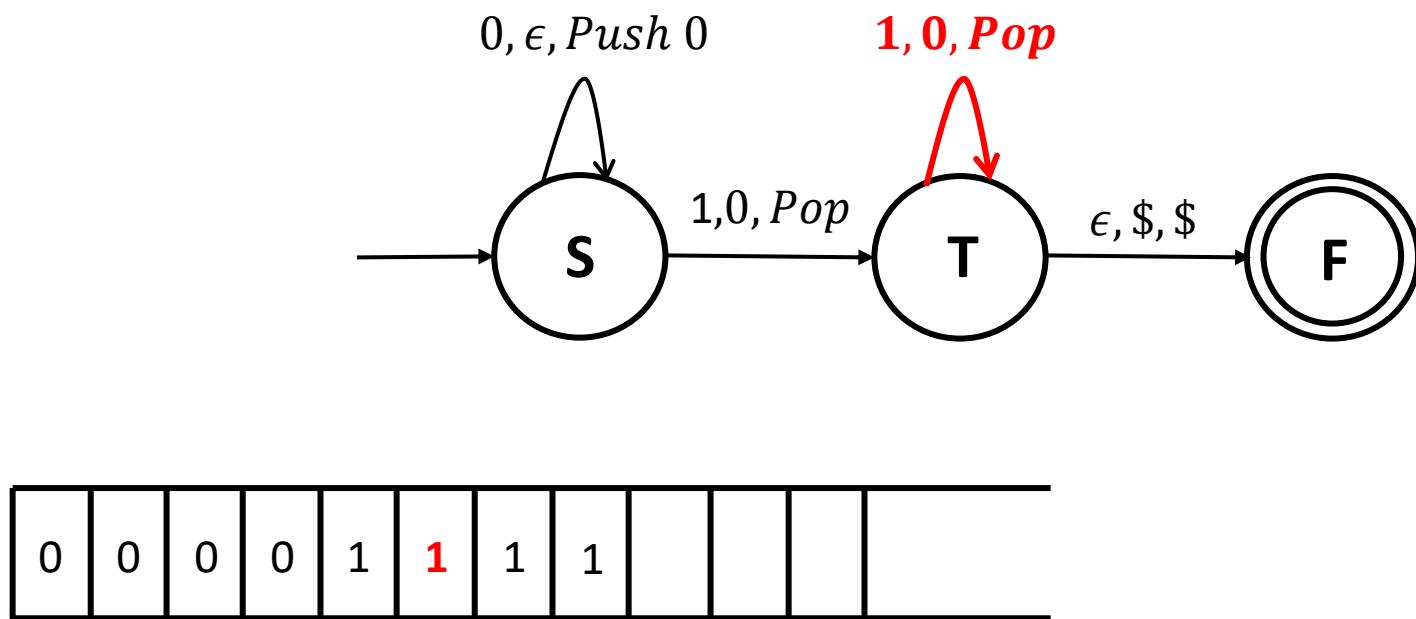
# Pushdown Automata

What is the language recognized by this PDA?



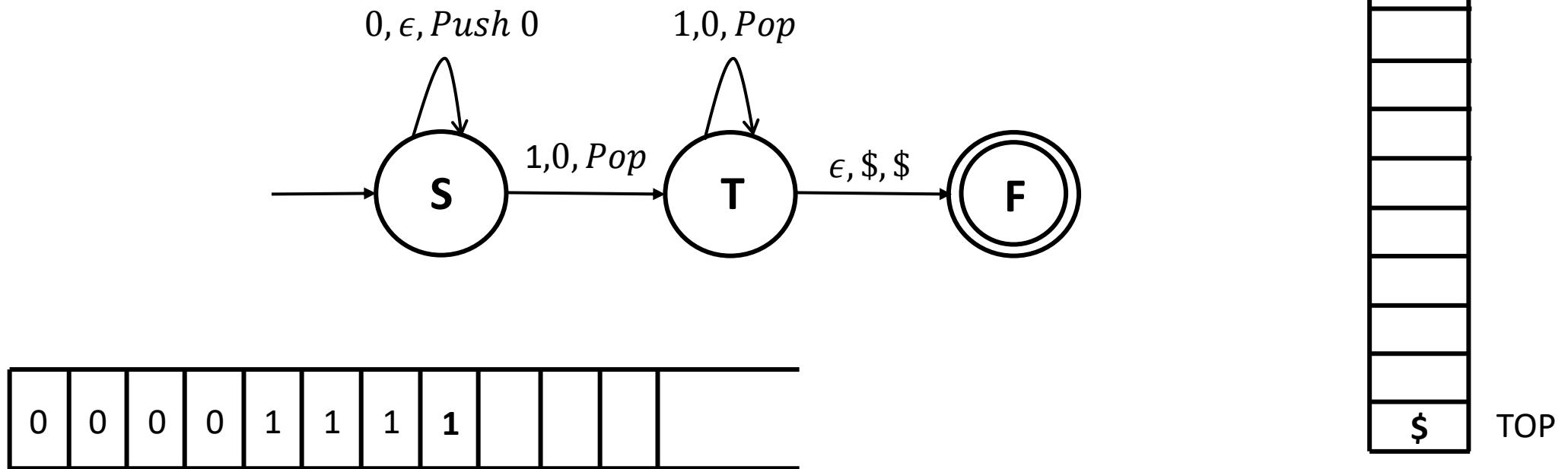
# Pushdown Automata

What is the language recognized by this PDA?



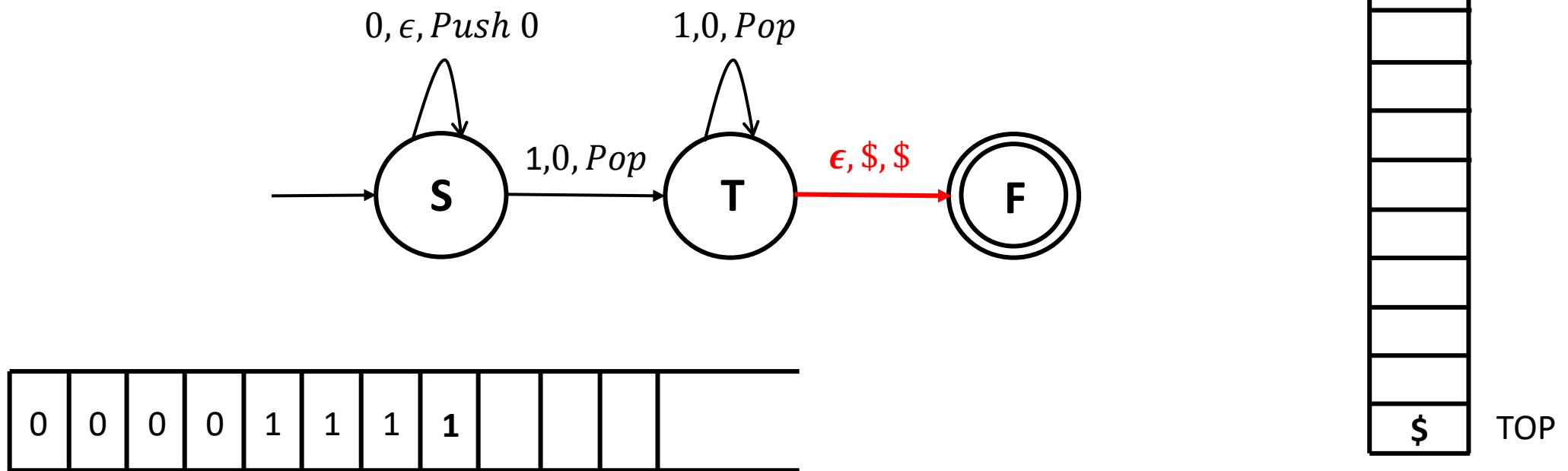
# Pushdown Automata

What is the language recognized by this PDA?



# Pushdown Automata

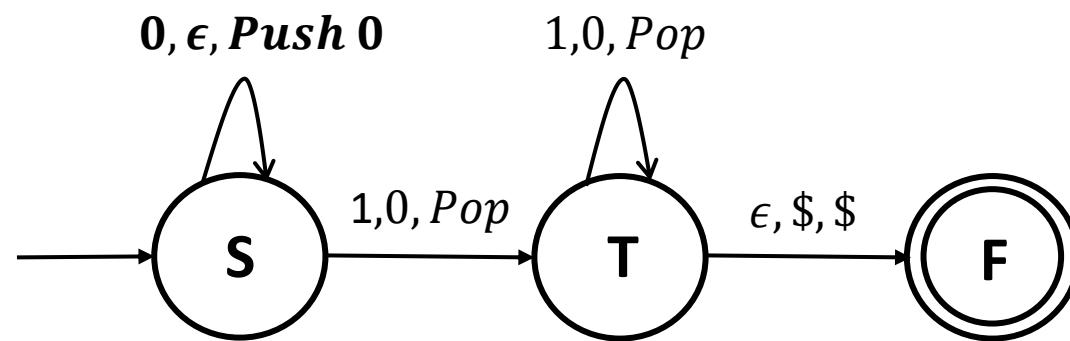
What is the language recognized by this PDA?



The language recognized by the PDA:  $L = \{0^n 1^n, n \geq 1\}$

# Pushdown Automata

What is the language recognized by this PDA?

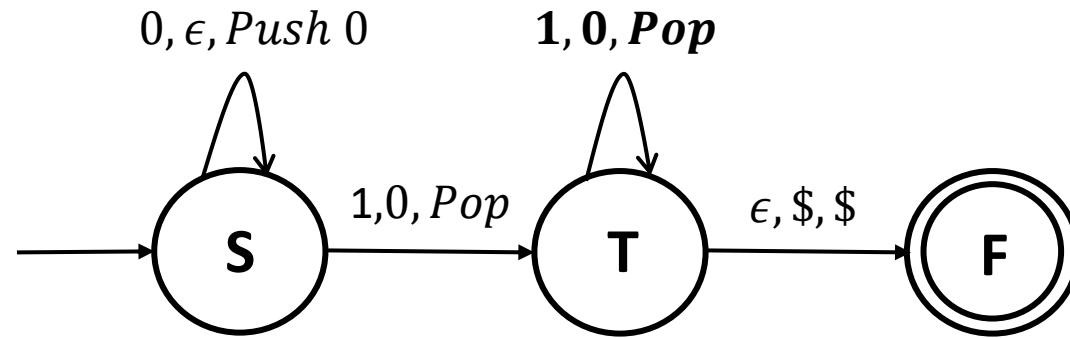


In some references (such as Sipser):

- The transitions of the PDA are labelled as " $a, b \rightarrow c$ ", implying: If the input symbol read is  $a$ , the element at the top of the stack is  $b$  (pop  $b$ ) and push  $c$  on to the Stack.

# Pushdown Automata

What is the language recognized by this PDA?

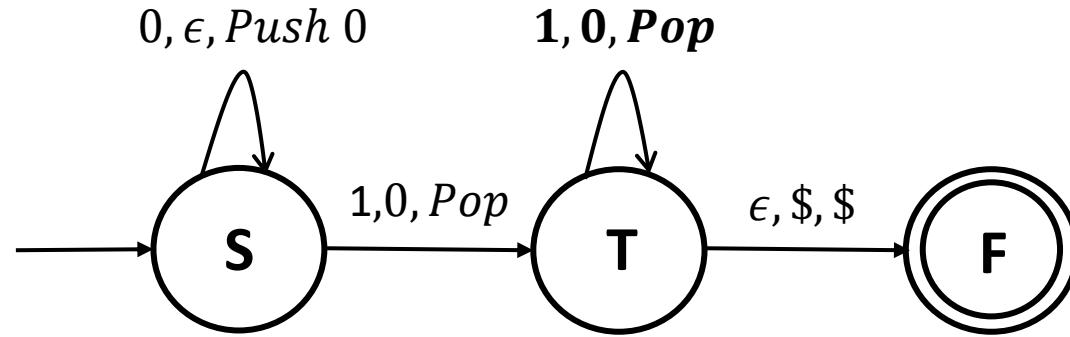


In some references (such as Sipser):

- The transitions of the PDA are labelled as " $a, b \rightarrow c$ ", implying: If the input symbol read is  $a$ , the element at the top of the stack is  $b$ , then pop  $b$  and push  $c$  on to the Stack.
- The label " $a, b \rightarrow \epsilon$ " implies that if the input symbol is  $a$  then **pop b**.

# Pushdown Automata

What is the language recognized by this PDA?



In some references (such as Sipser):

- The transitions of the PDA are labelled as " $a, b \rightarrow c$ ", implying: If the input symbol read is  $a$ , the element at the top of the stack is  $b$ , then pop  $b$  and push  $c$  on to the Stack.
- The label " $a, b \rightarrow \epsilon$ " implies that if the input symbol is  $a$  and  $b$  is popped.
- The symbol signifying the bottom of the Stack  $\$$  is pushed at the very beginning.

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

A PDA accepts a string  $w \in L$ , if there exists a run such that

- It reaches a **final state** when the entire string is read.

OR

- The **stack is empty** when the entire string is read.

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

A PDA accepts a string  $w \in L$ , if there exists a run such that

- It reaches a **final state** when the entire string is read.

OR

- The **stack is empty** when the entire string is read.

These two notions of acceptance are equivalent

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

**Transition function:**

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and  $b$  is popped, then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

**Transition function:**

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and  $b$  is popped, then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- $\delta(q_i, a, \epsilon) = (q_j, c)$ :

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

**Transition function:**

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and  $b$  is popped, then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- $\delta(q_i, a, \epsilon) = (q_j, c)$ : If the input symbol read is  $a$ , then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

**Transition function:**

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and  $b$  is popped, then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- $\delta(q_i, a, \epsilon) = (q_j, c)$ : If the input symbol read is  $a$ , then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- $\delta(q_i, a, b) = (q_j, \epsilon)$ :

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

**Transition function:**

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and  $b$  is popped, then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- $\delta(q_i, a, \epsilon) = (q_j, c)$ : If the input symbol read is  $a$ , then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- $\delta(q_i, a, b) = (q_j, \epsilon)$ : If the input symbol read is  $a$ , and the stack top =  $b$  ( $b$  is popped) then transition from  $q_i$  to  $q_j$
- $\delta(q_i, \epsilon, \$) = (q_j, \$)$ :

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

**Transition function:**

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and  $b$  is popped, then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- $\delta(q_i, a, \epsilon) = (q_j, c)$ : If the input symbol read is  $a$ , then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- $\delta(q_i, a, b) = (q_j, \epsilon)$ : If the input symbol read is  $a$ , and the stack top =  $b$ , then pop  $b$  and transition from  $q_i$  to  $q_j$
- $\delta(q_i, \epsilon, \$) = (q_j, \$)$ : Transition from  $q_i$  to  $q_j$  if the stack is empty.

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

**Transition function:**

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and  $b$  is popped, then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- If the input symbol read is  $a$  and  $a$  is popped, then Push  $a$  and remain at  $q_i$  : ?

# Pushdown Automata

Formally, a PDA M is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

**Transition function:**

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and  $b$  is popped, then push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- If the input symbol read is  $a$  and  $a$  is popped, then Push  $a$  and remain at  $q_i$  :  $\delta(q_i, a, a) = (q_i, a)$

# Pushdown Automata

Formally, a PDA  $M$  is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

- The Language of the PDA  $P$  is the set of strings the PDA accepts, i.e.

$$L = \{w \mid P \text{ accepts } w\}$$

There exists an accepting run for  $w$  on  $P$

- If  $\mathcal{L}(P) = L$ , then the PDA  $P$  recognizes  $L$

# Pushdown Automata

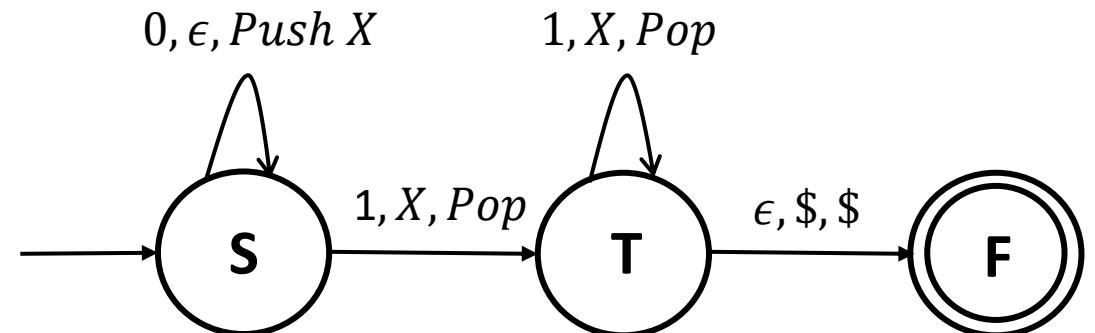
Formally, a PDA  $M$  is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

- The Language of the PDA  $P$  is the set of strings the PDA accepts, i.e.

$$L = \{w \mid P \text{ accepts } w\}$$

- If  $\mathcal{L}(P) = L$ , then the PDA  $P$  recognizes  $L$
- Stack alphabet **can be different** from the input alphabet



# Pushdown Automata

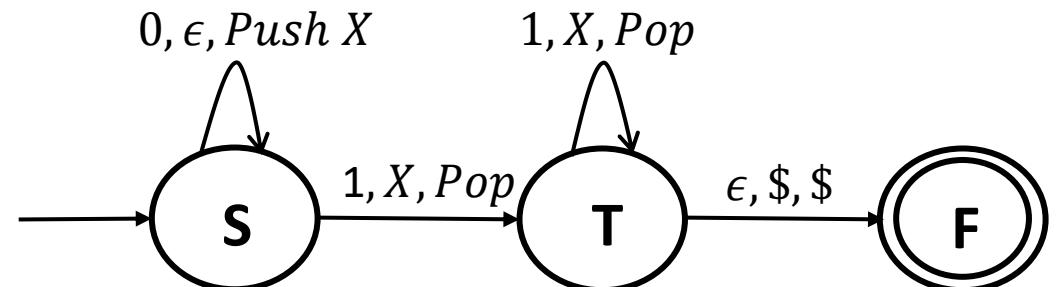
Formally, a PDA  $M$  is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

- The Language of the PDA  $P$  is the set of strings the PDA accepts, i.e.

$$L = \{w \mid P \text{ accepts } w\}$$

- If  $\mathcal{L}(P) = L$ , then the PDA  $P$  recognizes  $L$
- Stack alphabet **can be different** from the input alphabet



$$\delta(S, 0, \epsilon) = (S, X)$$

# Pushdown Automata

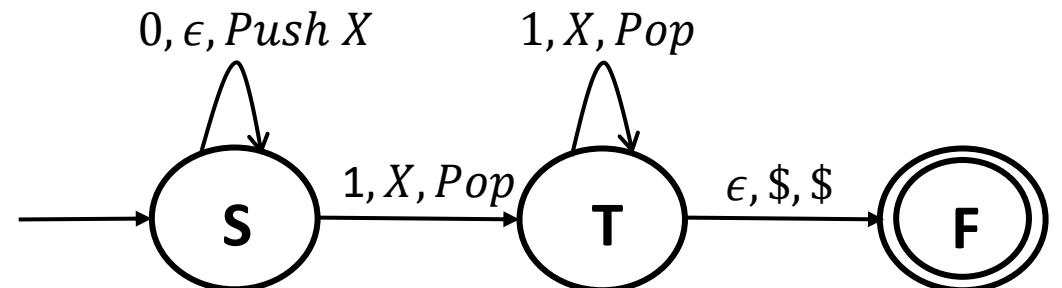
Formally, a PDA  $M$  is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

- The Language of the PDA  $P$  is the set of strings the PDA accepts, i.e.

$$L = \{w \mid P \text{ accepts } w\}$$

- If  $\mathcal{L}(P) = L$ , then the PDA  $P$  recognizes  $L$
- Stack alphabet **can be different** from the input alphabet



$$\begin{aligned}\delta(S, 0, \epsilon) &= (S, X) \\ \delta(S, 1, X) &= (T, \epsilon)\end{aligned}$$

# Pushdown Automata

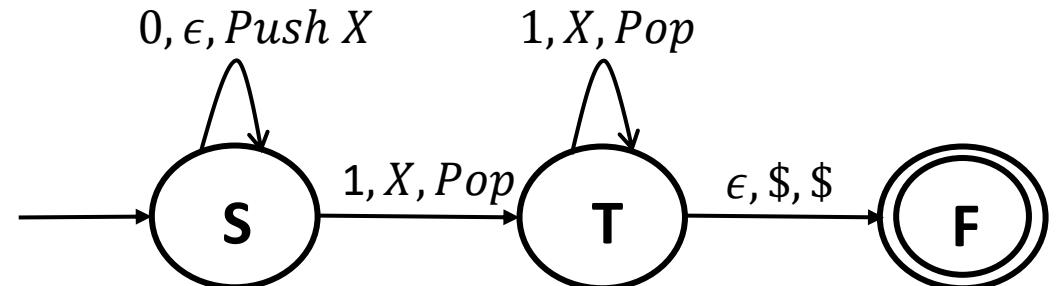
Formally, a PDA  $M$  is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the set of **Stack alphabets**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  is the **transition function** [  $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$  and  $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$  ]
- $q_0 \in Q$  is the **start state**.
- $F \subseteq Q$  is the set of **accepting states**.

- The Language of the PDA  $P$  is the set of strings the PDA accepts, i.e.

$$L = \{w \mid P \text{ accepts } w\}$$

- If  $\mathcal{L}(P) = L$ , then the PDA  $P$  recognizes  $L$
- Stack alphabet **can be different** from the input alphabet



$$\begin{aligned}\delta(S, 0, \epsilon) &= (S, X) \\ \delta(S, 1, X) &= (T, \epsilon) \\ \delta(T, 1, X) &= (T, \epsilon) \\ \delta(T, \epsilon, \$) &= (F, \$)\end{aligned}$$

# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.

# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).

# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

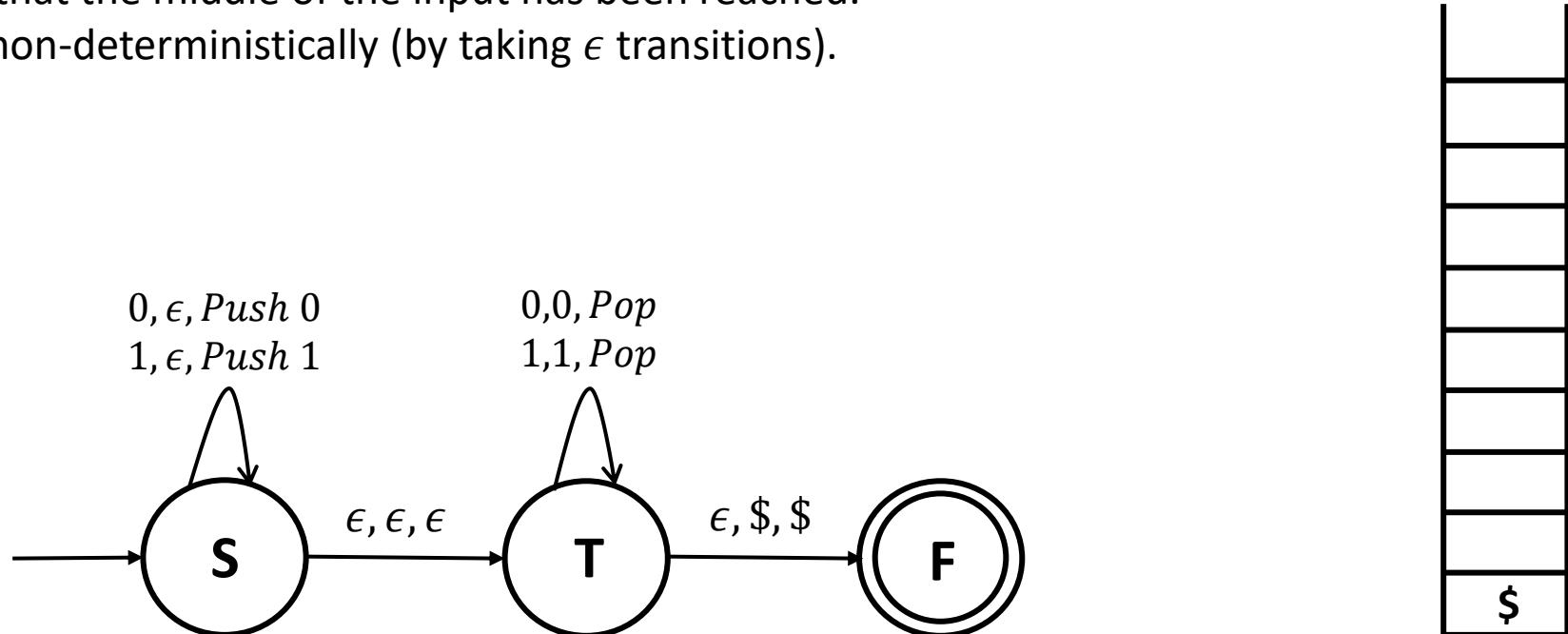
- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).
- The above intuition is applicable for even length palindromes of the form  $ww^R$ .
- What about odd length palindromes?
  - Non-determinism to the rescue once again

# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).

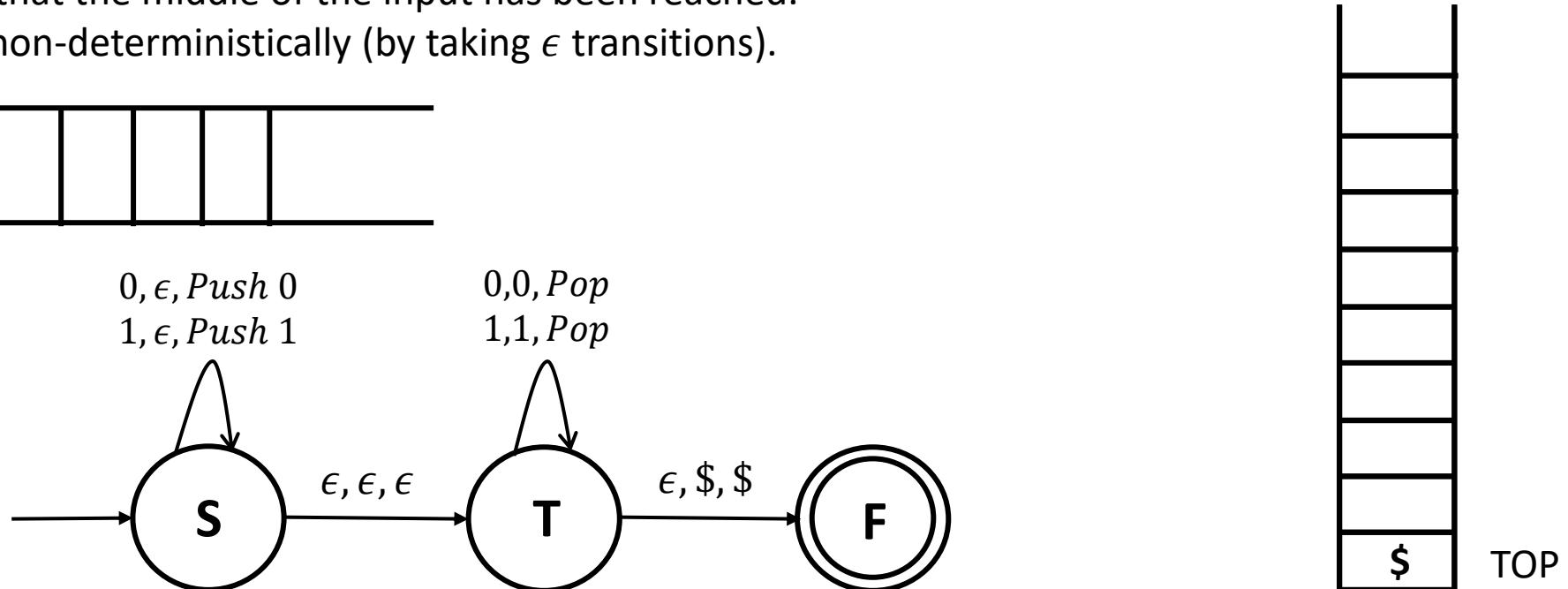
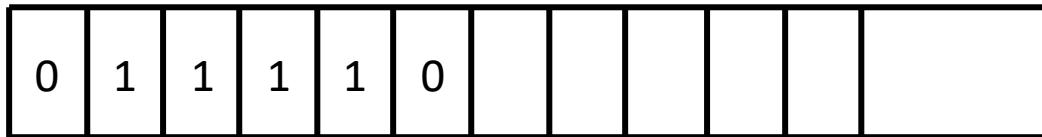


# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).

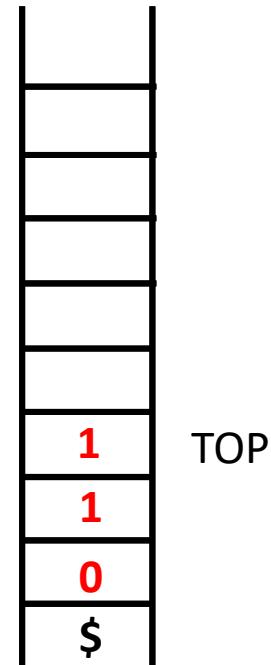
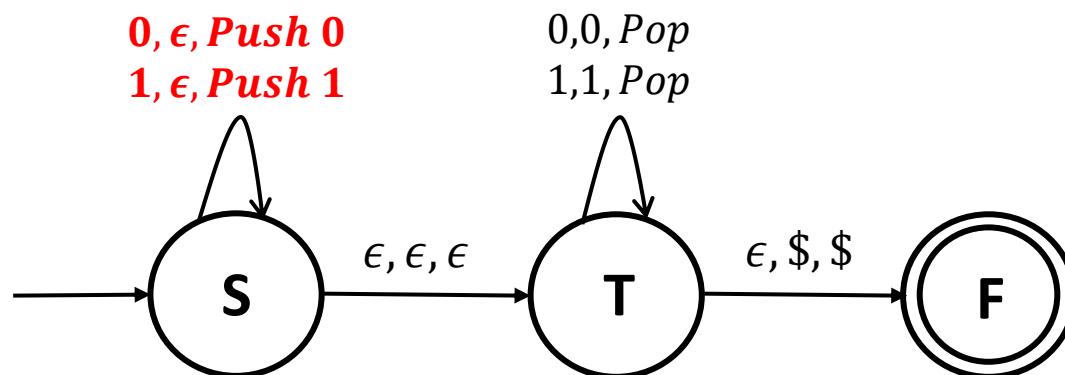
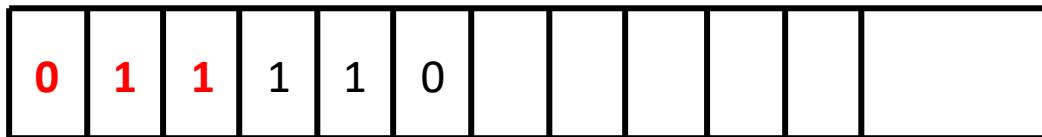


# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).

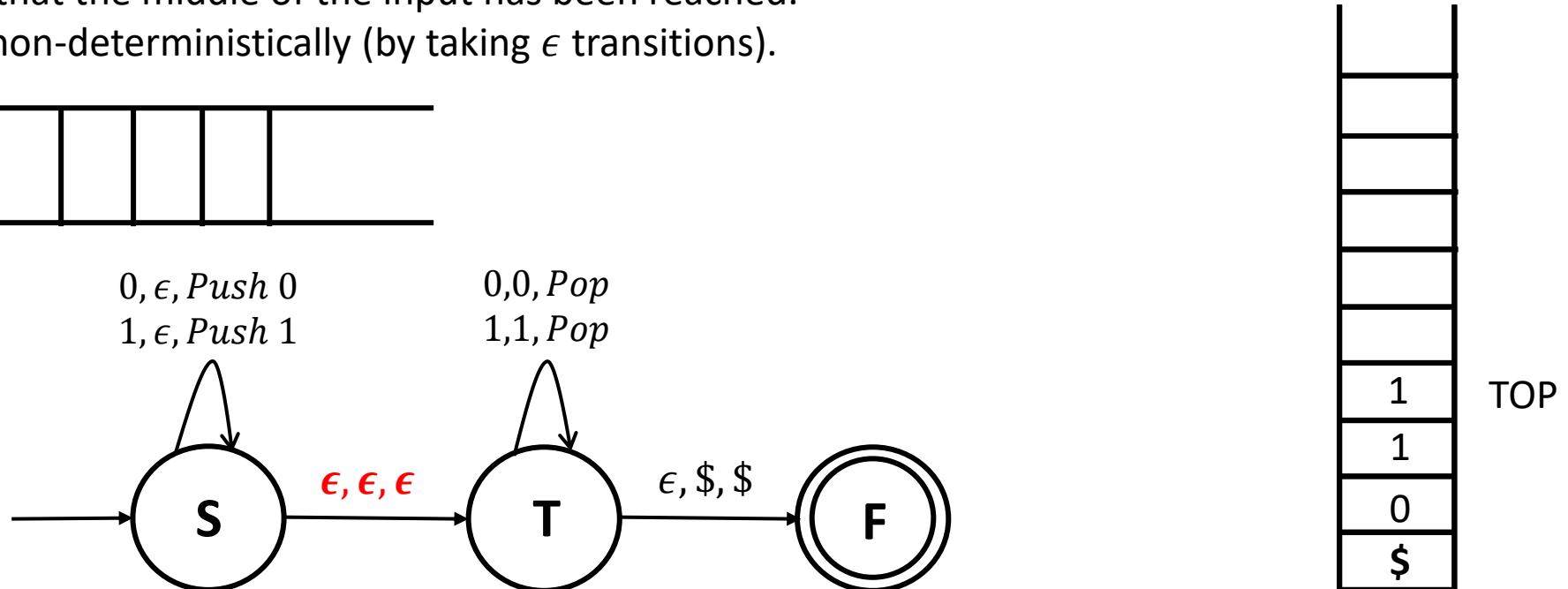
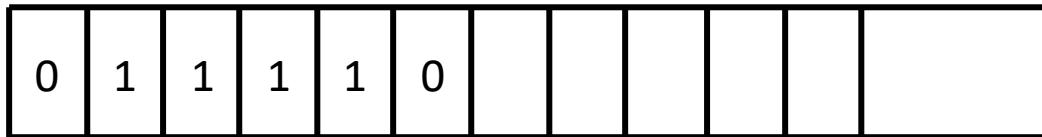


# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).

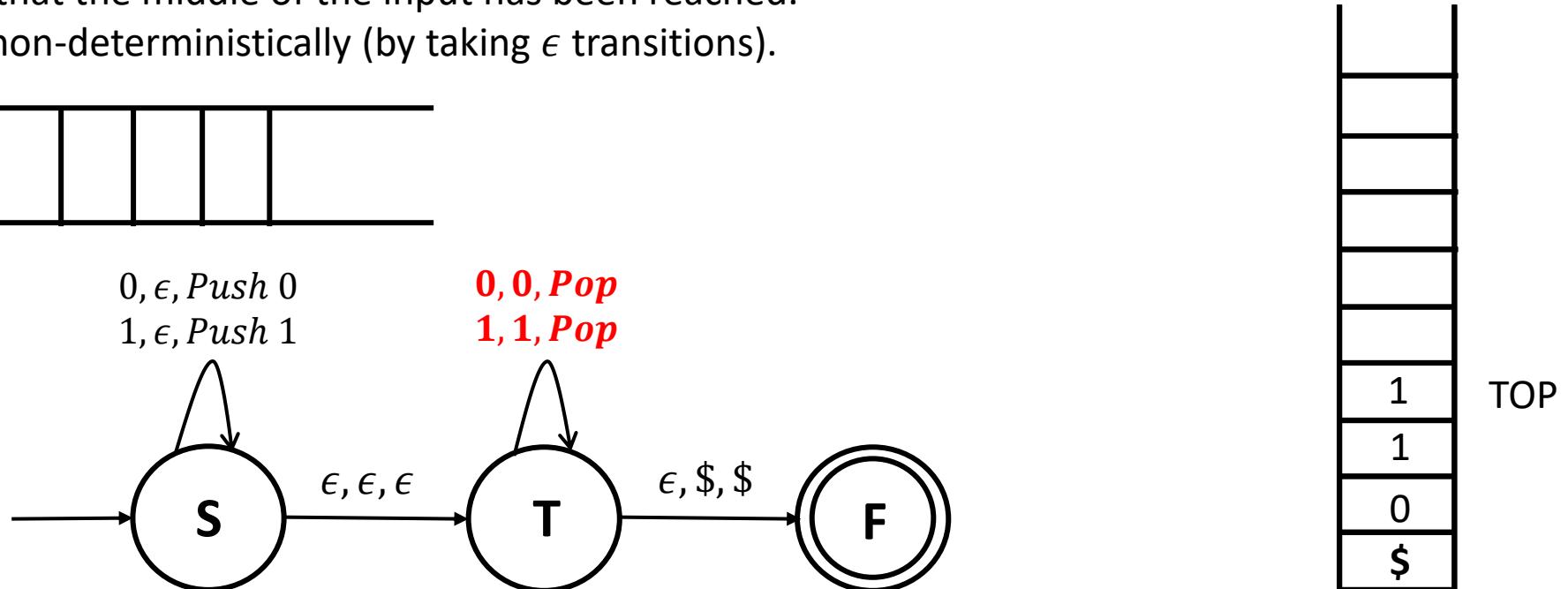
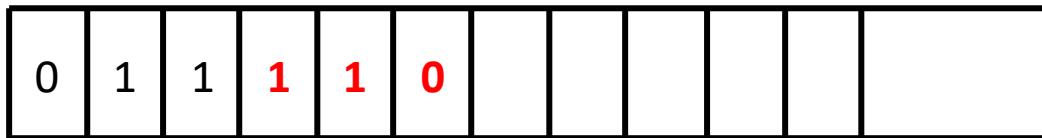


# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).

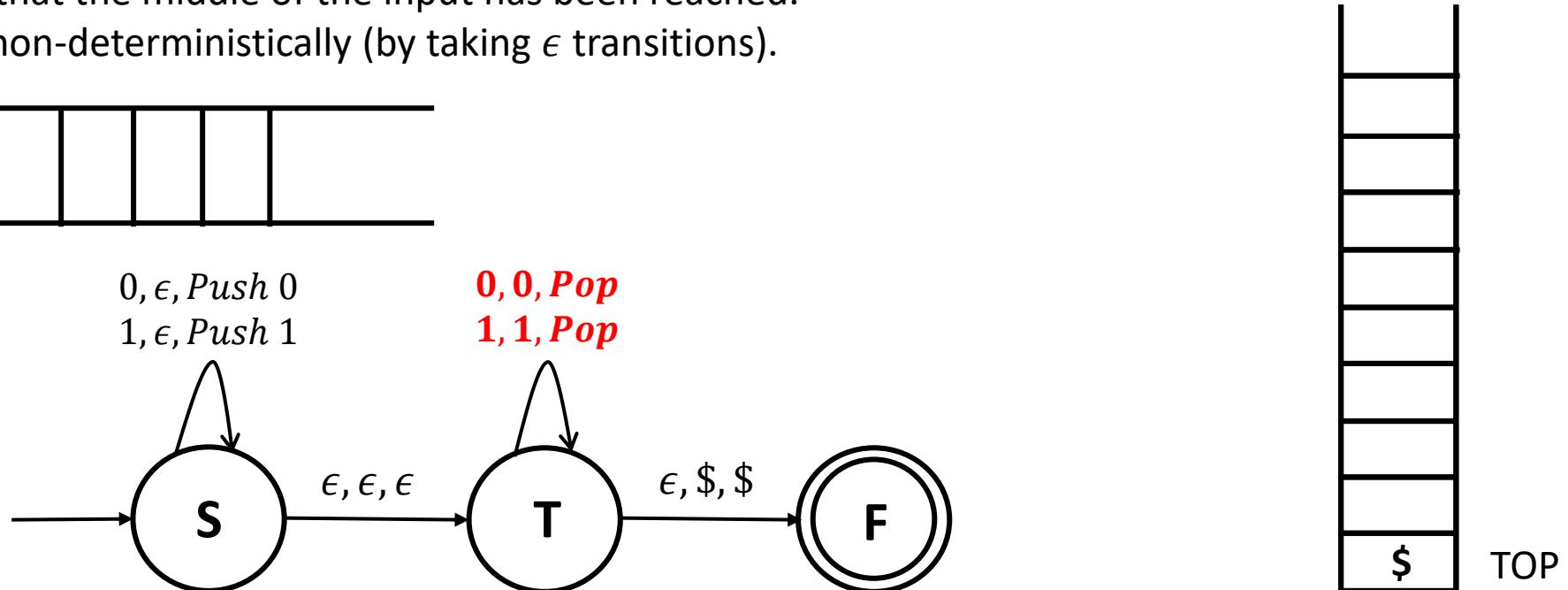
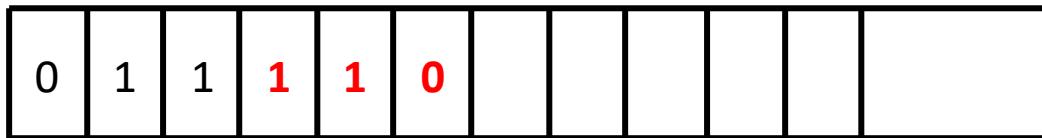


# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).

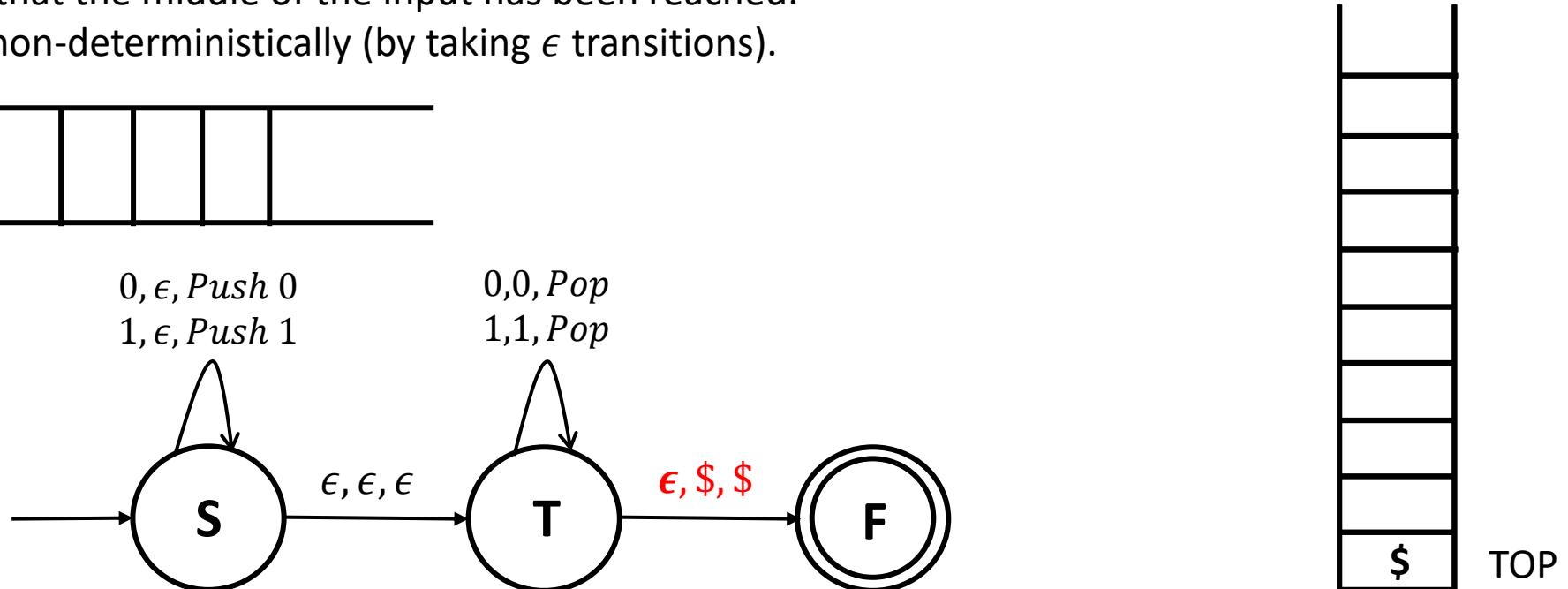
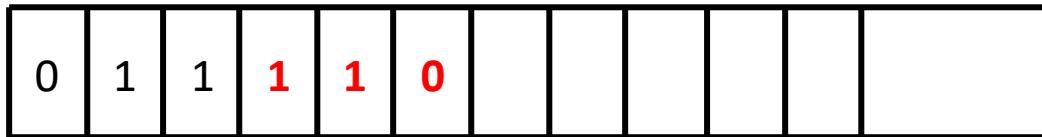


# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).

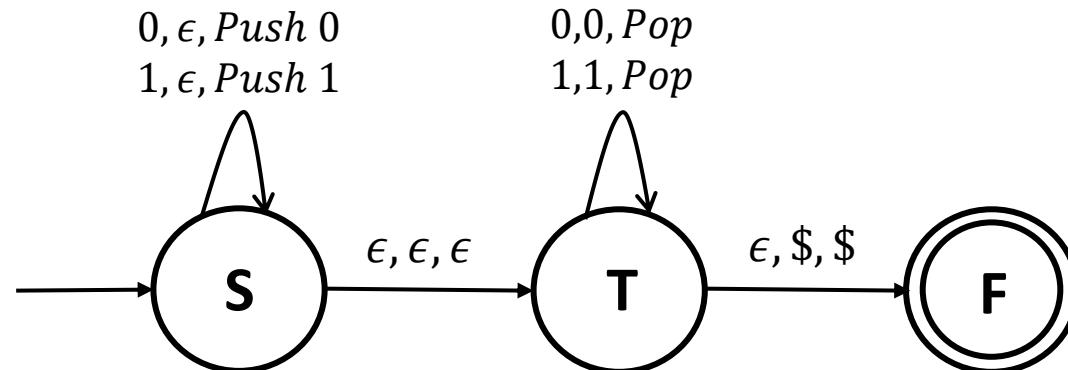


# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).
- What about odd length palindromes?



Recognizes even length  
palindromes of the  
form:  $ww^R$

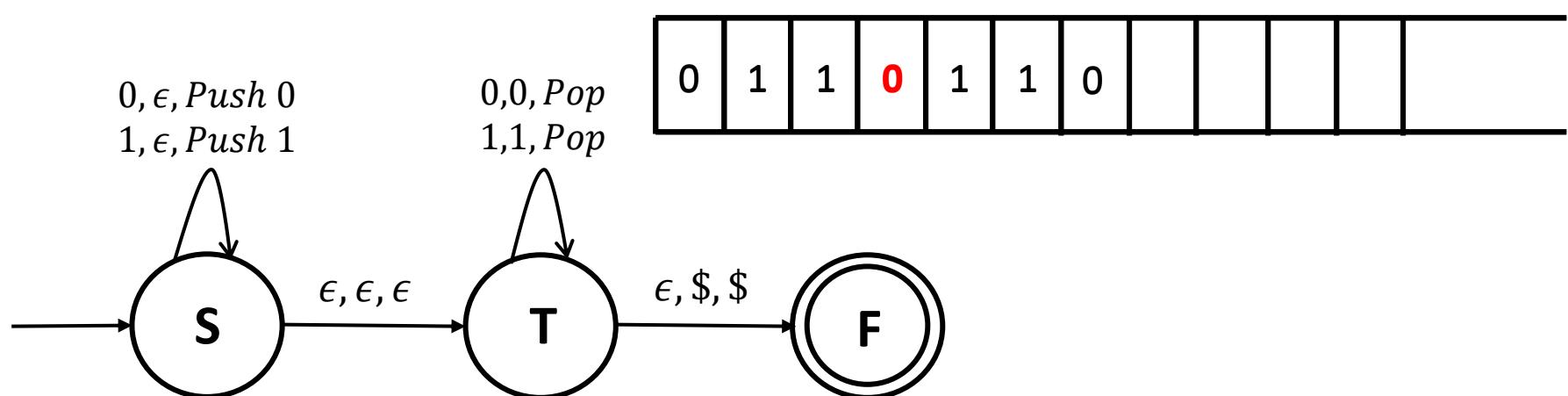
# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).
- What about odd length palindromes?

Odd length palindromes  
are of the form  $wcw^R$ ,  
such that  
 $c \in \Sigma$



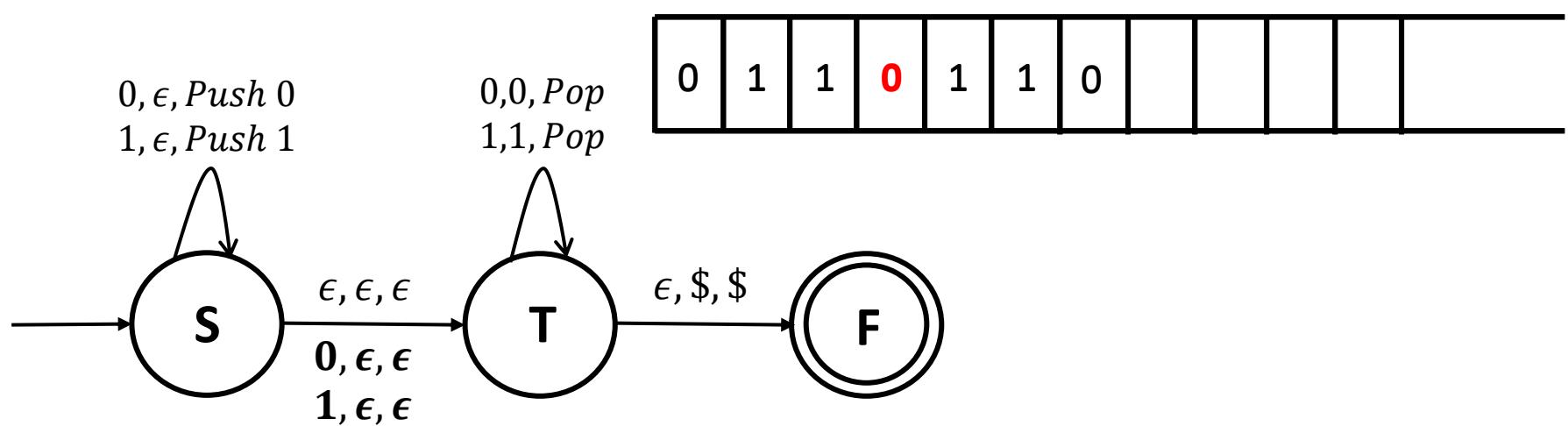
# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).
- What about odd length palindromes?

Odd length palindromes  
are of the form  $wcw^R$ ,  
such that  
 $c \in \Sigma$

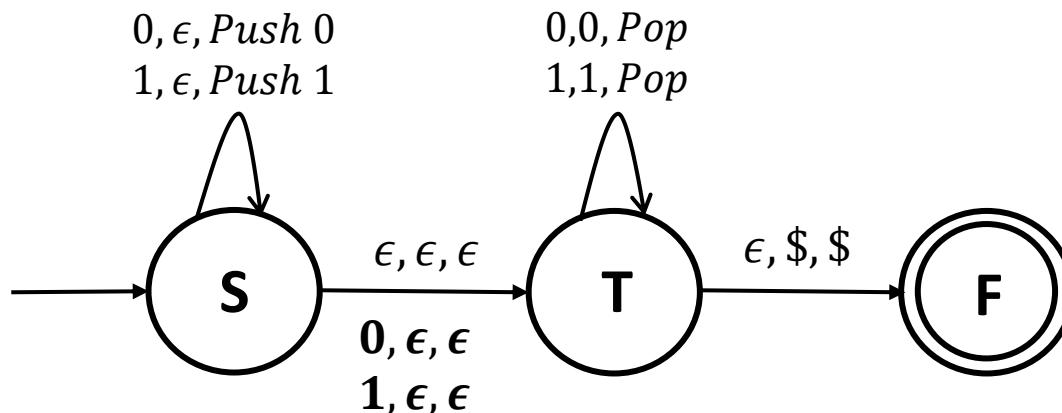


# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).
- What about odd length palindromes?



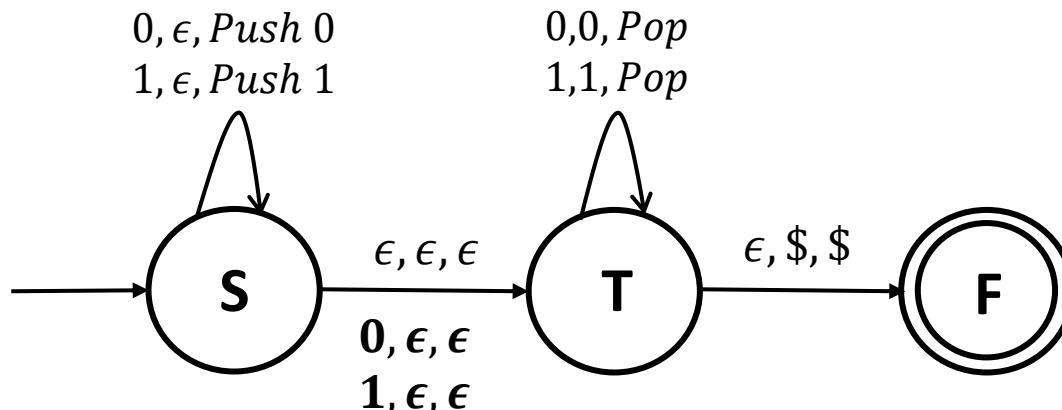
The transitions  **$0, \epsilon, \epsilon$  and  $1, \epsilon, \epsilon$**  allow the PDA to **consume one symbol** and then begin matching what it has encountered thus far.

# Pushdown Automata

Let  $\Sigma = \{0,1\}$  consider the language  $L = \{w \in \Sigma^* \mid w \text{ is a Palindrome}\}$ . Design a PDA  $P$  that recognizes  $L$ .

## Intuition

- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  transitions).
- What about odd length palindromes?



The transitions  **$0, \epsilon, \epsilon$  and  $1, \epsilon, \epsilon$**  allow the PDA to **consume one symbol** and then begin matching what it has encountered thus far.

This allows the PDA to **recognize strings of the form:  $wcw^R$** , where the aforementioned transitions non-deterministically guessed  $c \in \{0,1\}$

**Thank You!**