

# CS 302.1 - Automata Theory

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

- Obtain models of computation : simple to more powerful ones.
- 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}

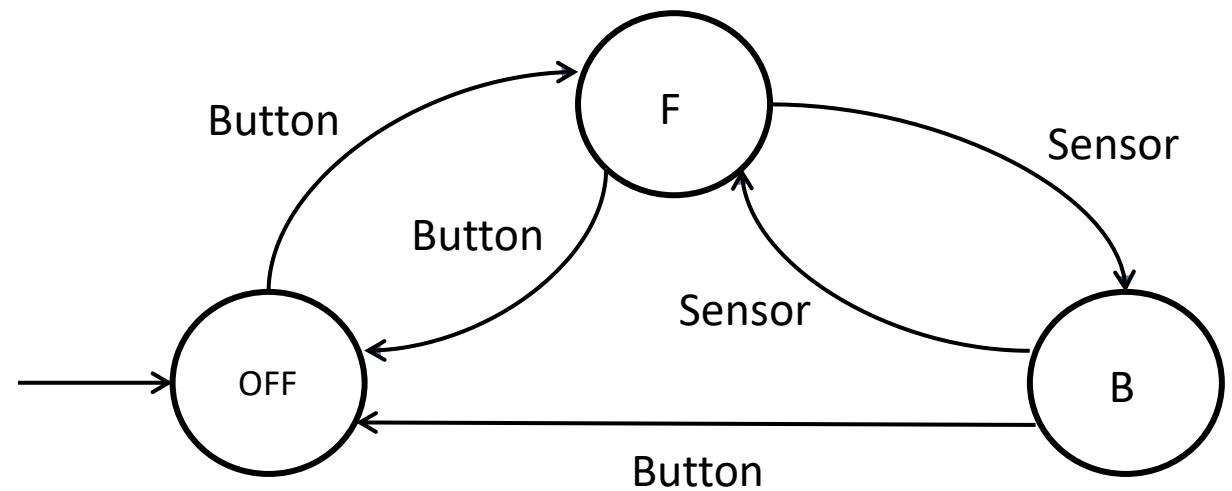
Inputs : {BUTTON, SENSOR}

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

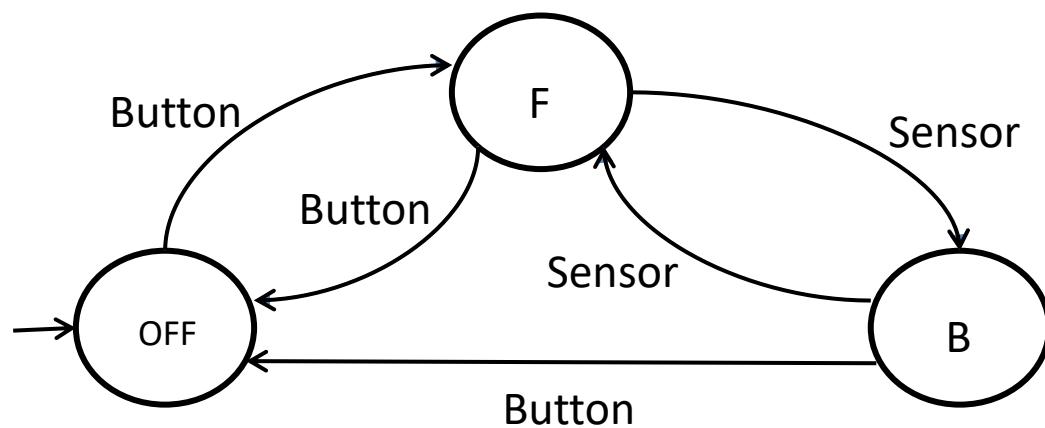


State diagram for the robot

# 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?

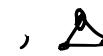
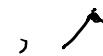
**Problem vs decision problem:** In order to answer these questions, we will always convert a given problem into a *decision (YES-NO) problem*. We will always do this!

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 : {  ,  ,  , ..... }

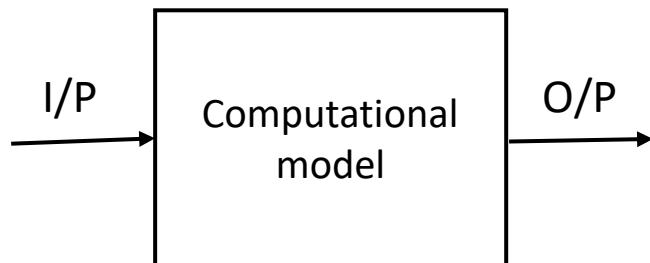
NO set : {  ,  ,  , ..... }

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**.

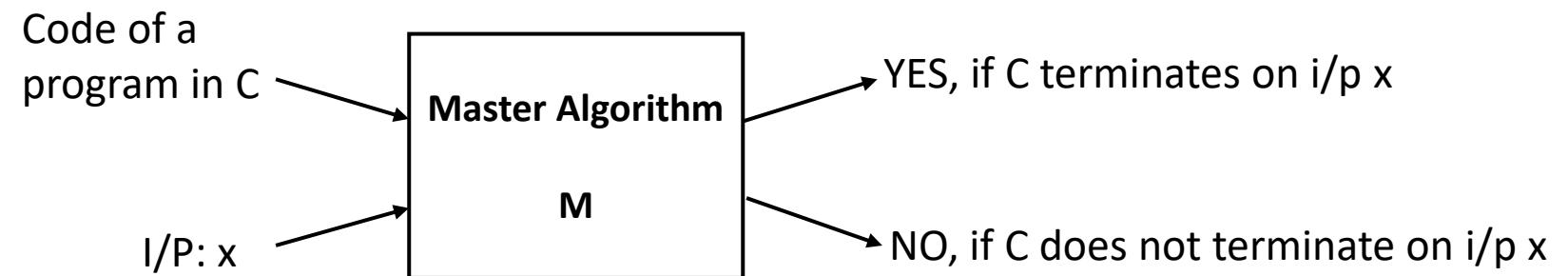
If (i) and (ii) hold, we say that the problem P is **computable** by this computational model.

# 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

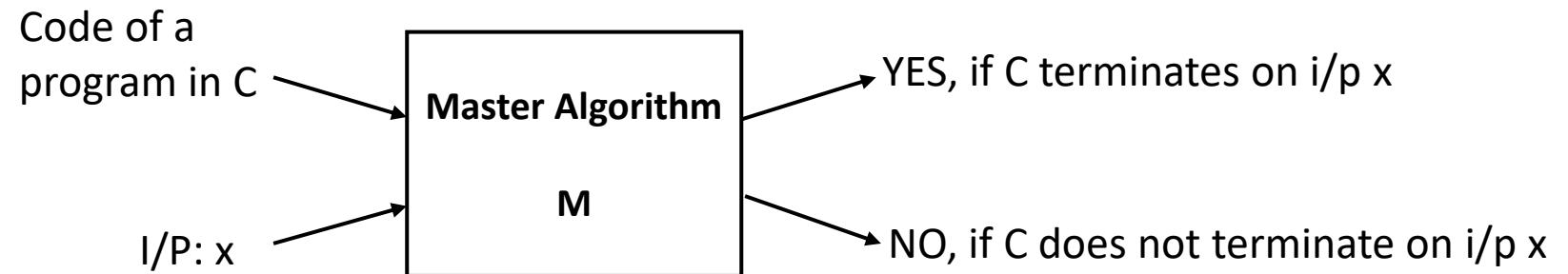


# Introduction

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

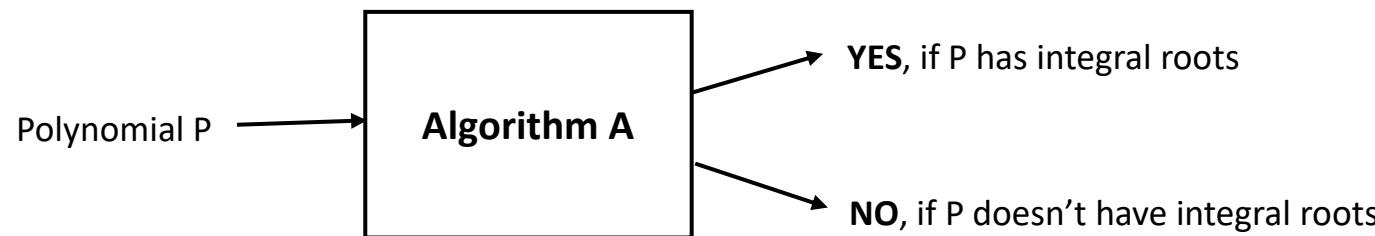


- **M terminates and outputs NO even if  $C(x)$  runs infinitely!**
- No such Algorithm M can be written. **Undecidable problem!**

**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

- ❖ 12 Lectures in all
- ❖ Final Exam at the end (**35% weightage**)
- ❖ Two theory assignments (**20% weightage**)
  - Assignment 1 – released after Lec 3 (Deadline: 10 days)
  - Assignment 2 – released after Lec 9 (Deadline: 15 days)
- ❖ Programming assignment (**25% weightage**)
  - Assignment 1 – Released after Lec 3 (Deadline: End of sem)
  - Assignment 2 – Released after Lec 5 (Deadline: 10 days)
- ❖ Quiz (**20% weightage**)

# Tutorials and TAs

- Tutorial sessions weekly: Thursday 3:30 PM – 5 PM.
- Teaching Associates:
  - **Aditya Morolia** ([aditya.morolia@research.iiit.ac.in](mailto:aditya.morolia@research.iiit.ac.in))
  - **Aakash Aanegola** ([aakash.aanegola@students.iiit.ac.in](mailto:aakash.aanegola@students.iiit.ac.in))
  - **Ashwin Mittal** ([ashwin.mittal@students.iiit.ac.in](mailto:ashwin.mittal@students.iiit.ac.in))
  - **Zeeshan Ahmed** ([zeeshan.ahmed@research.iiit.ac.in](mailto:zeeshan.ahmed@research.iiit.ac.in))
  - **Aakash Jain** ([aakash.jain@students.iiit.ac.in](mailto:aakash.jain@students.iiit.ac.in))
  - **Shaurya Dewan** ([shaurya.dewan@students.iiit.ac.in](mailto:shaurya.dewan@students.iiit.ac.in))
  - **Alapan Chaudhuri** ([alapan.chaudhuri@research.iiit.ac.in](mailto:alapan.chaudhuri@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>

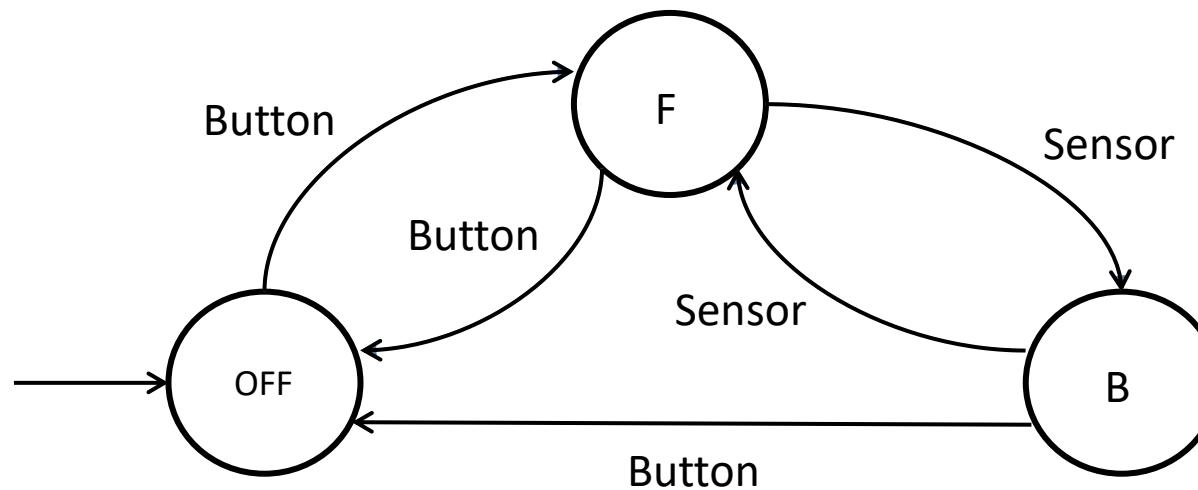
# Some terminology

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

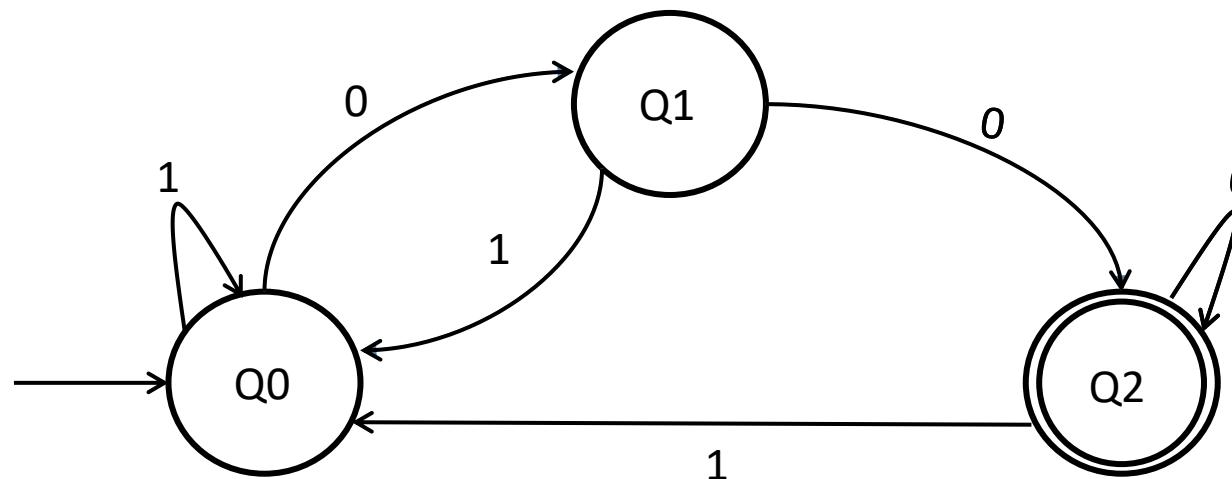


Characteristics: (i) Single start State, (ii) Unique Transitions, (iii) Zero or more final states

# Models of computation

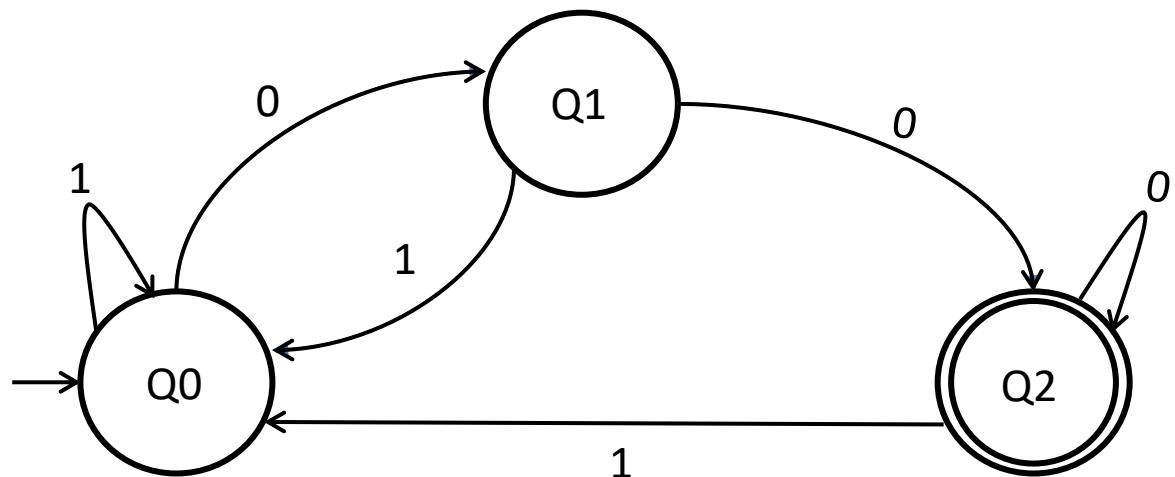
- Deterministic Finite Automata (DFA) Model

Q0: Start state, Q2: Final state



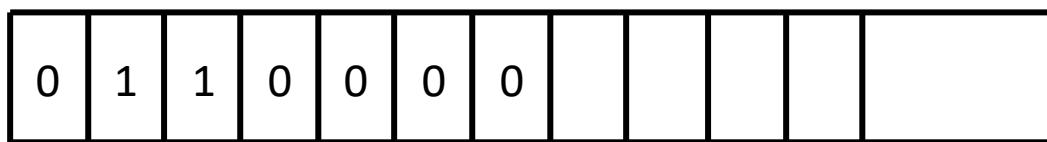
State transition diagram of the Finite State Machine

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine

One-way infinite tape



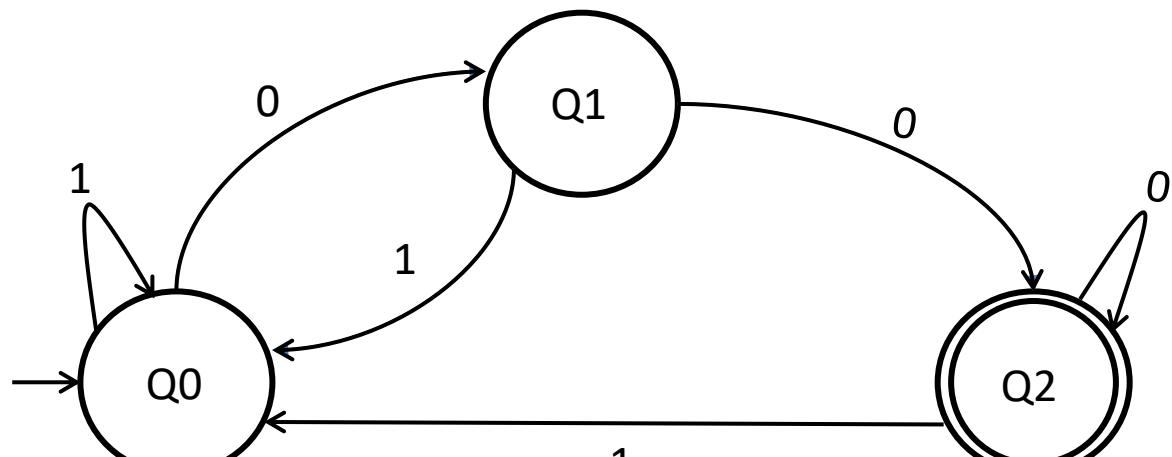
Input: Strings from alphabet  $\Sigma = \{0,1\}$

Q0: Start state, Q2: Final state

**Run:**

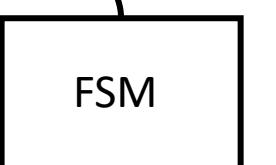
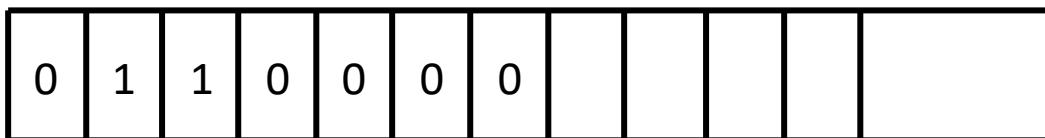
$Q0 \xrightarrow{0} Q1 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{0} Q2 \xrightarrow{0} Q2 \xrightarrow{0} Q2$

# 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

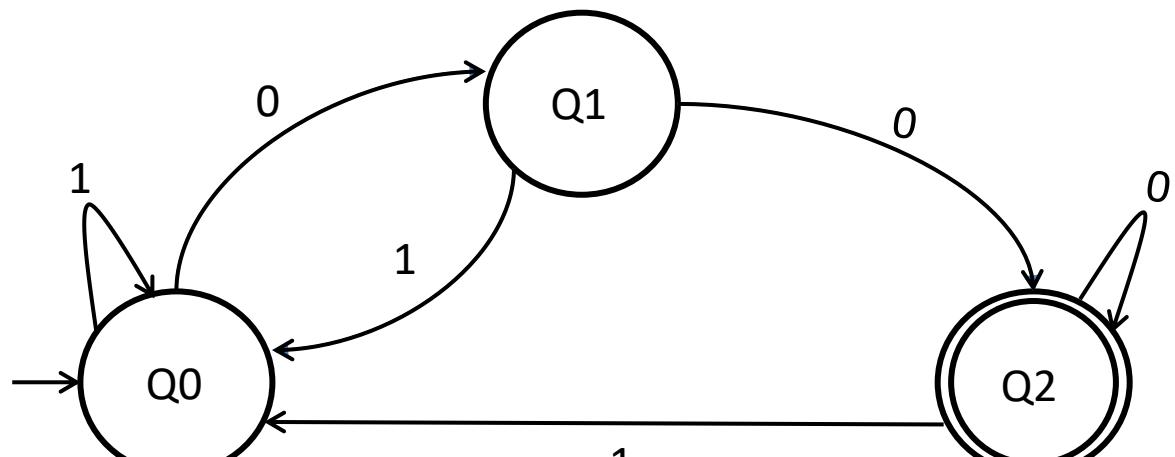
The DFA “accepts” an input string, if it corresponds to a *run* that ends up in the final state Q2. (**Accepting Run**)

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

Run:

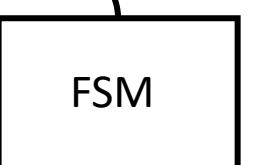
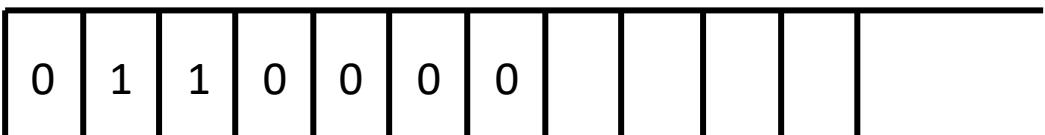
$Q0 \xrightarrow{0} Q1 \xrightarrow{1} Q0 \xrightarrow{1} Q0 \xrightarrow{0} Q1 \xrightarrow{0} Q2 \xrightarrow{0} Q2 \xrightarrow{0} Q2$

# Deterministic Finite Automata (DFA)



State transition diagram of the Finite State Machine

One-way infinite tape



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**)

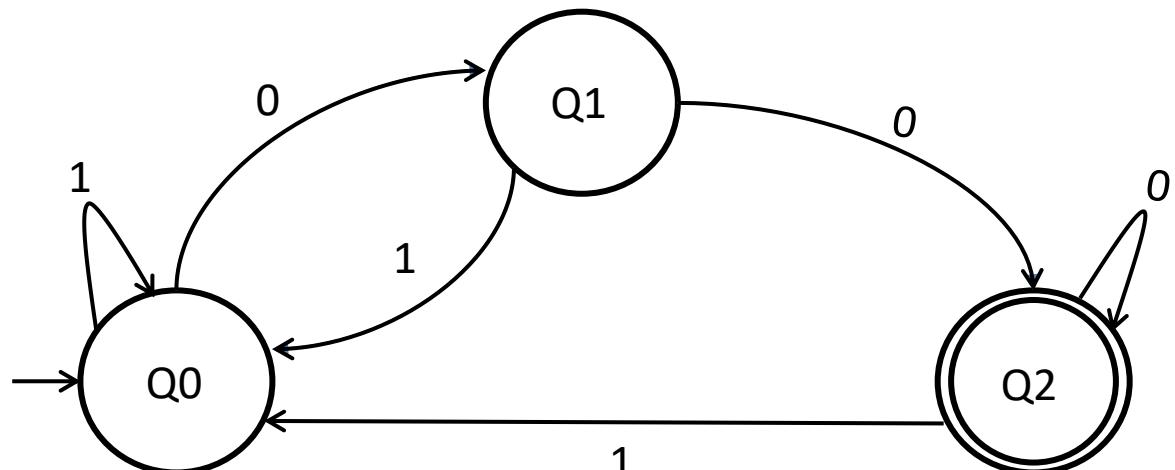
Run:

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

ACCEPT = {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**)

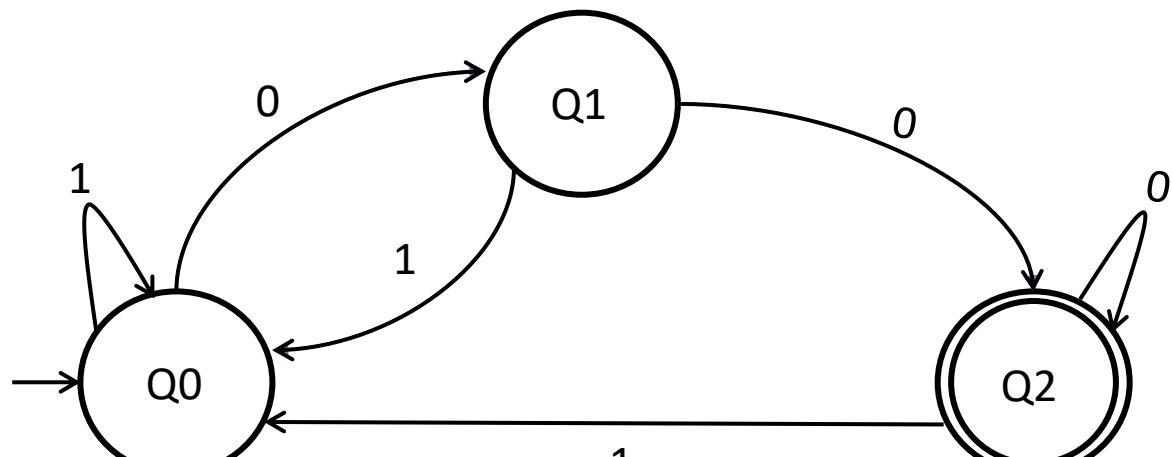
Run:

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

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**)

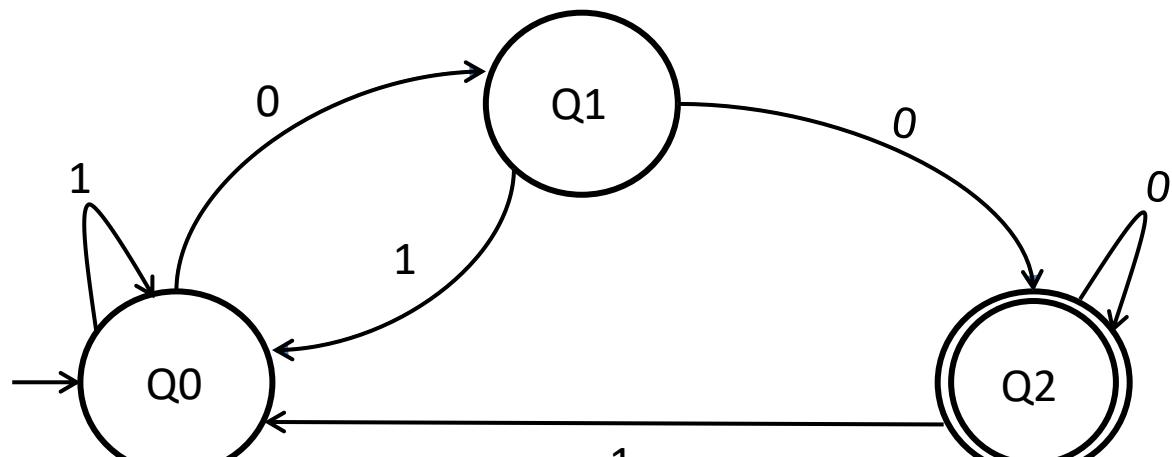
Run:

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

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**)

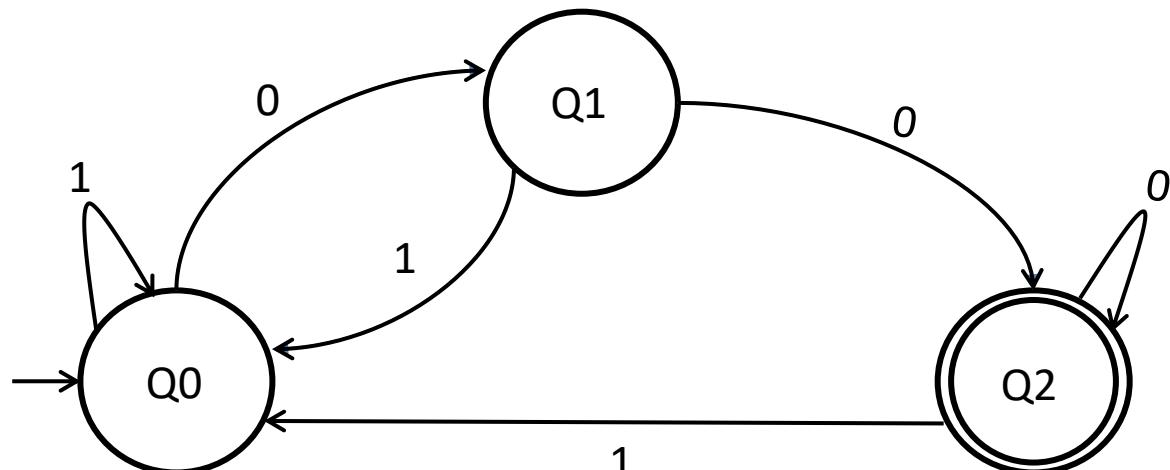
**Run:**

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

ACCEPT = {0111000, 10100, ...}

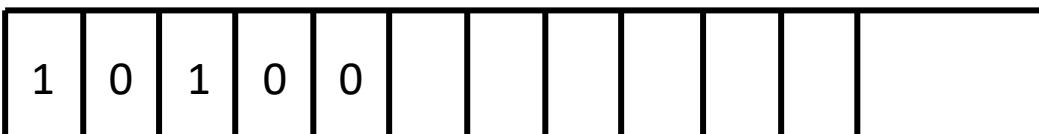
REJECT = {11101, ....}

# 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**)

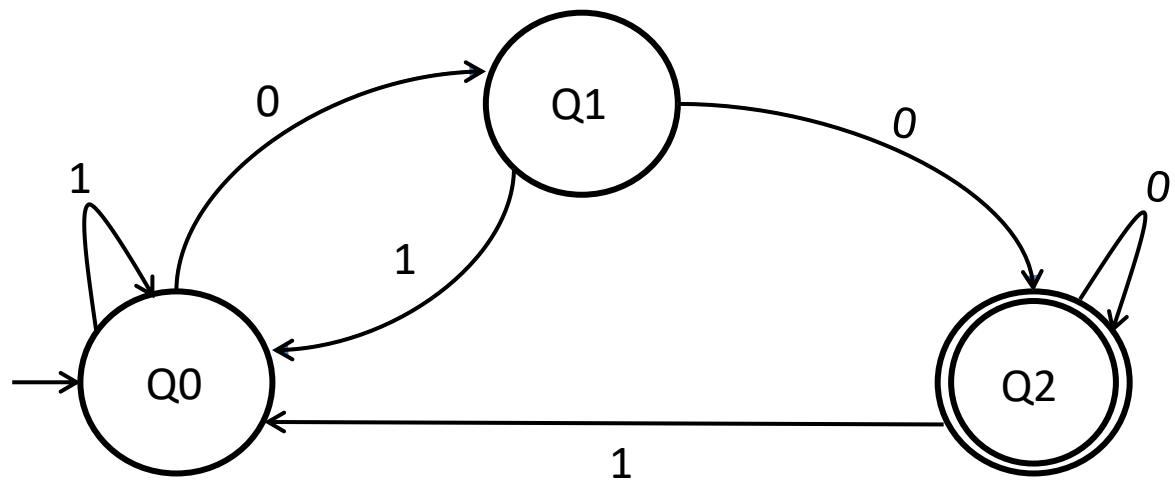
**Run:**

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

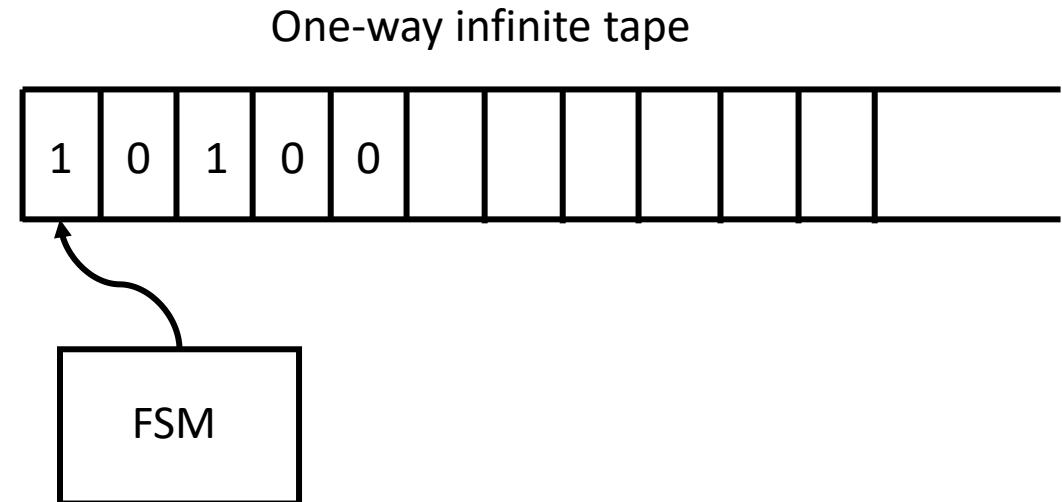
ACCEPT = {0111000, 10100, 0100, 00, 10000,...}

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,.....}

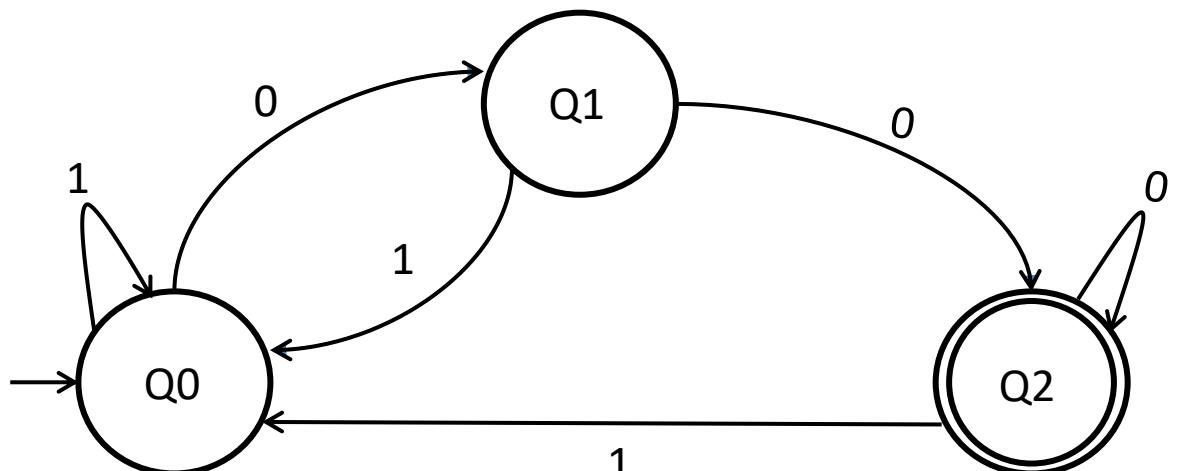
Let the DFA be M. Then, **the problem M solves/ language M accepts is**

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

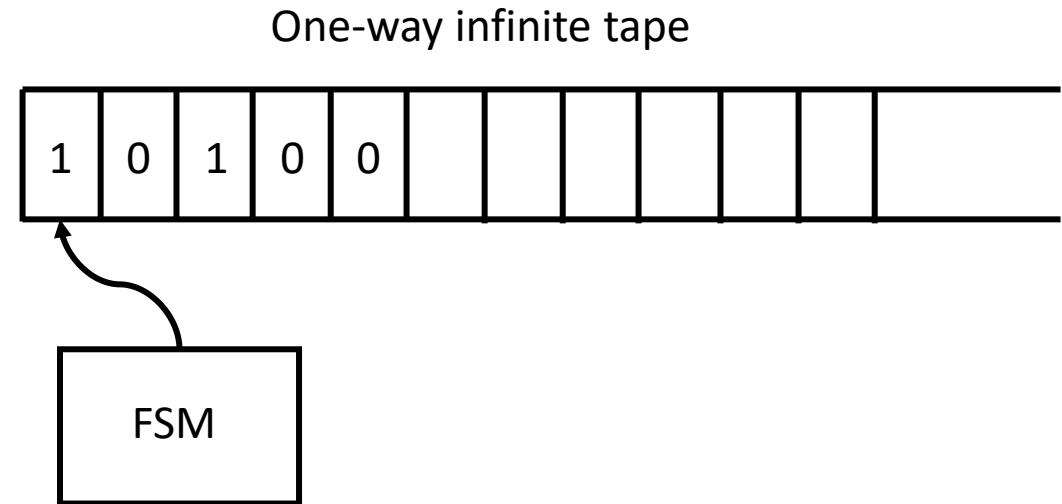
Equivalently, **M recognizes L**

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

# 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**.

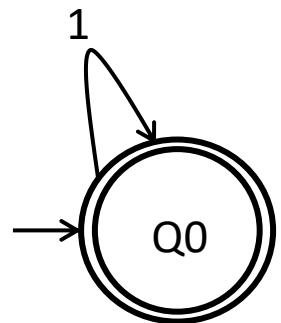
$$\begin{aligned} Q &= \{Q0, Q1, Q2\} \\ \Sigma &= \{0, 1\} \\ (Q0, 0) &\mapsto Q1; (Q0, 1) \mapsto Q0, \dots, (Q2, 1) \mapsto Q0 \\ q_0 &= Q0 \\ F &= Q2 \end{aligned}$$

# Constructing DFA for a language

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

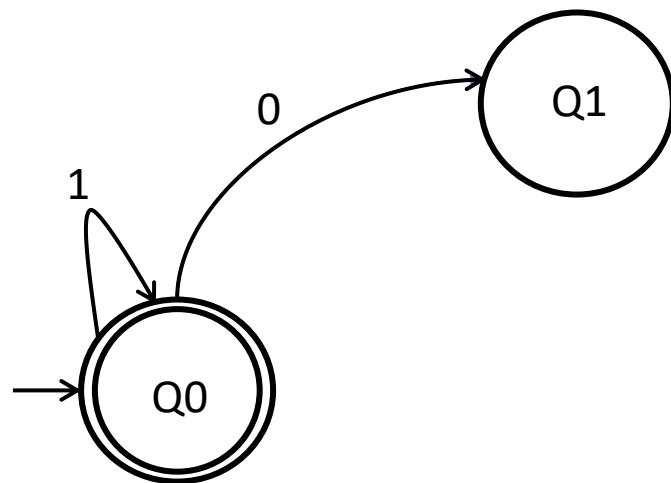
# Constructing DFA for a language

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



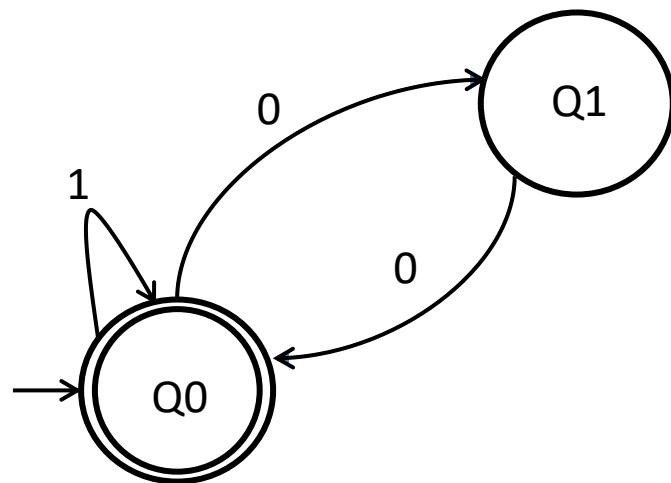
# Constructing DFA for a language

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



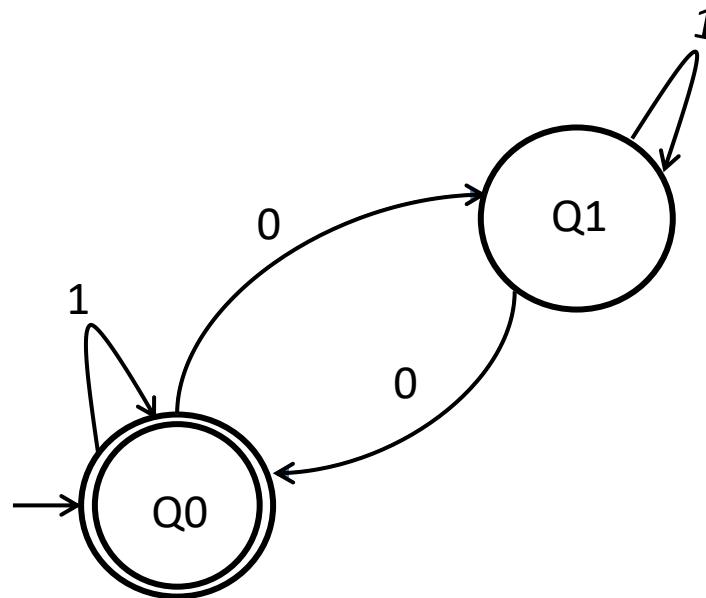
# Constructing DFA for a language

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.  $\omega = 0 \pmod{3}$

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

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

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

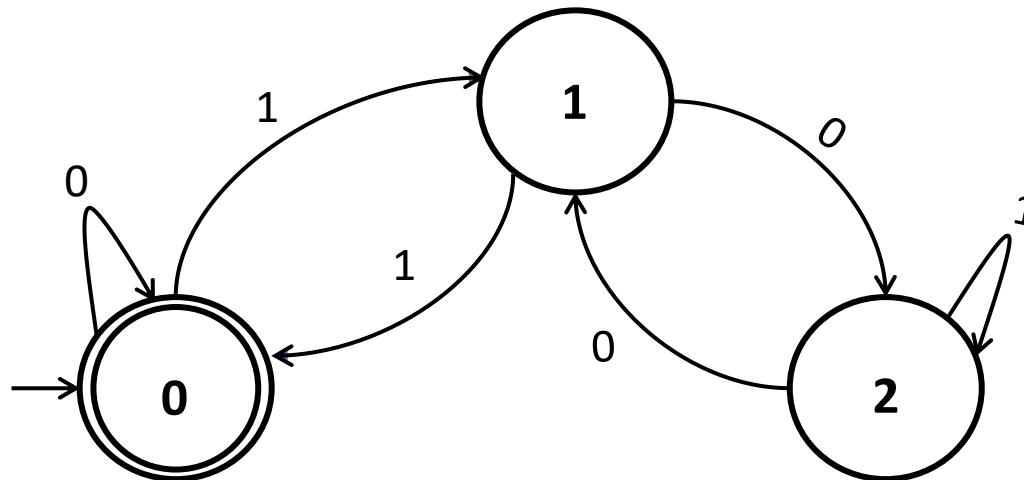
$$\omega 11 = 2 \times \text{value}(\omega 1) + 1 = 0 \pmod{3}$$

.... 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.  $\omega = 0 \pmod{3}$

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

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

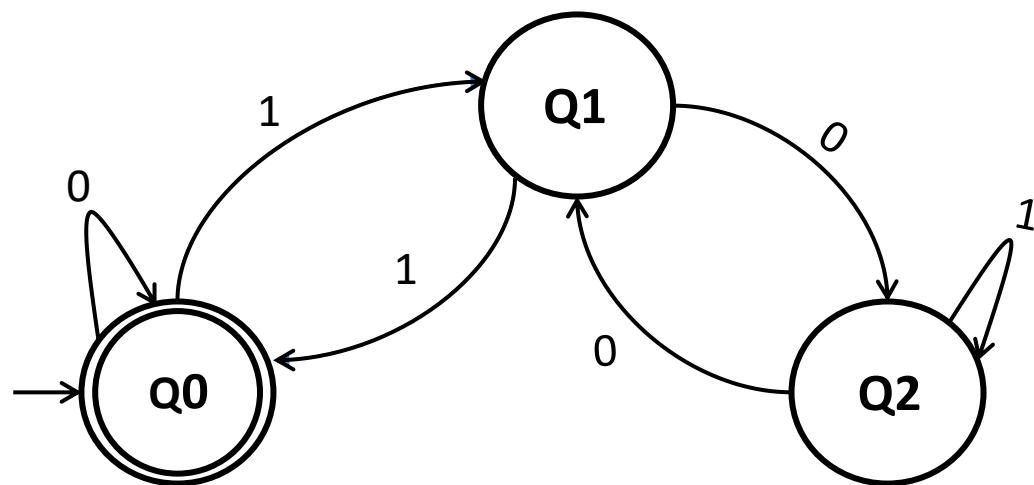
$$\omega 10 = 2 \times \text{value}(\omega 1) = 2 \pmod{3}$$

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

.... 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!

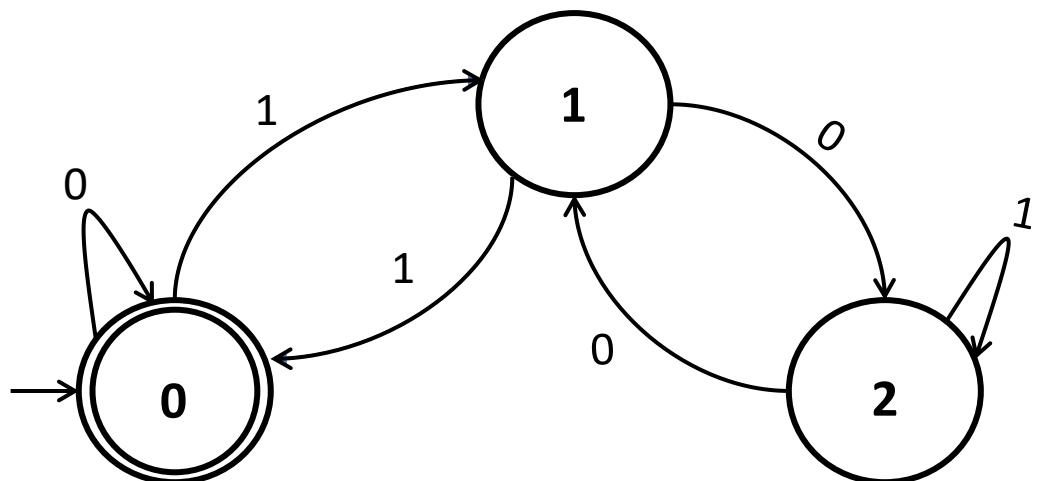
In fact if any DFA accepts  $L$ , the toggled DFA accepts  $\bar{L}$ , the complement of  $L$

# 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  $L$

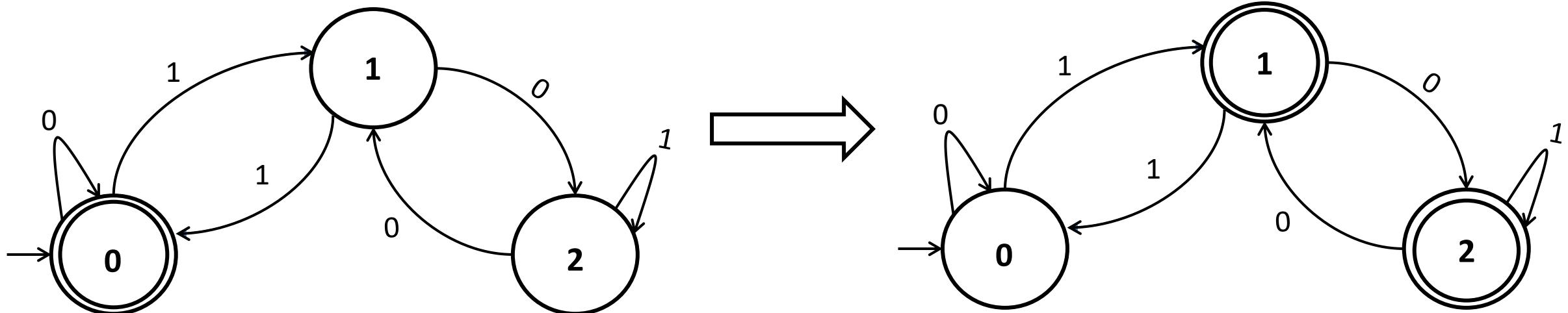


# Constructing DFA for a language

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)

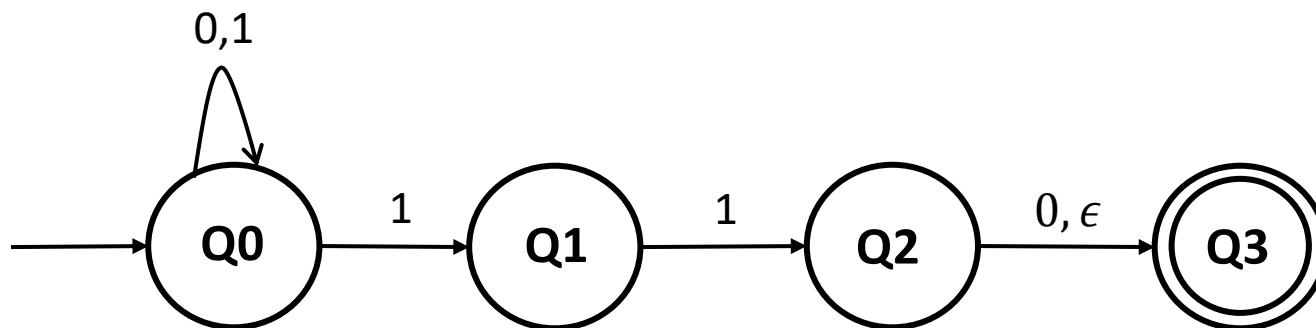
Characteristics of DFA : (i) Single start state (ii) Unique transitions (iii) Zero or more final states

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

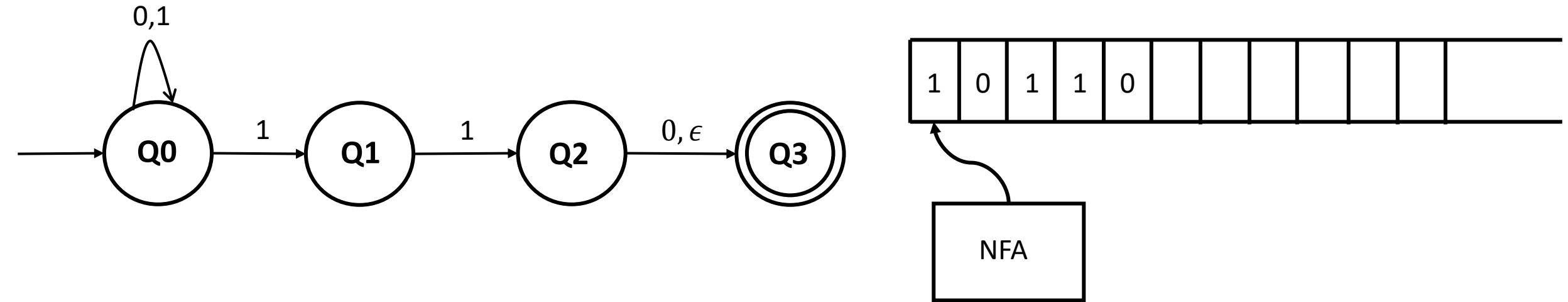
(iii) Multiple transitions are possible on the same input for a state

(iv) Some transitions might be missing

(v)  $\epsilon$  - transitions



# Non-deterministic Finite Automata (NFA)

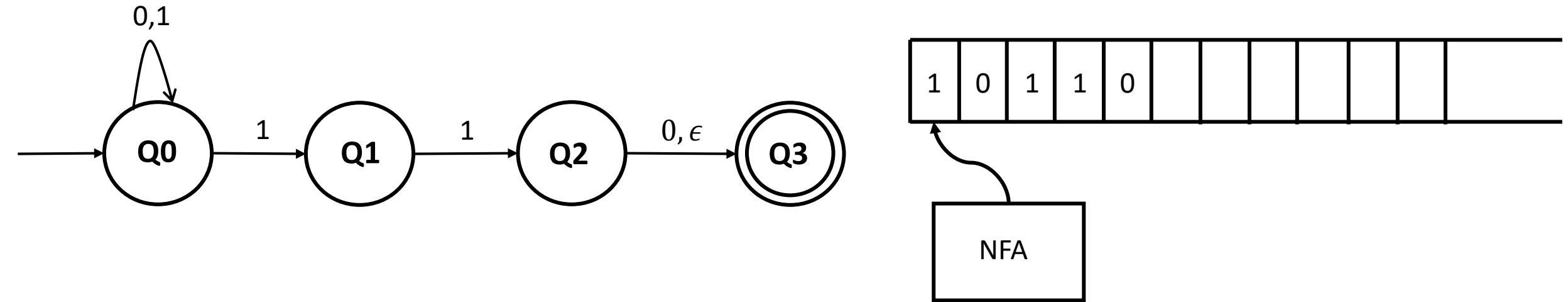


**Run 1:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0$  (**REJECT**)

**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**)

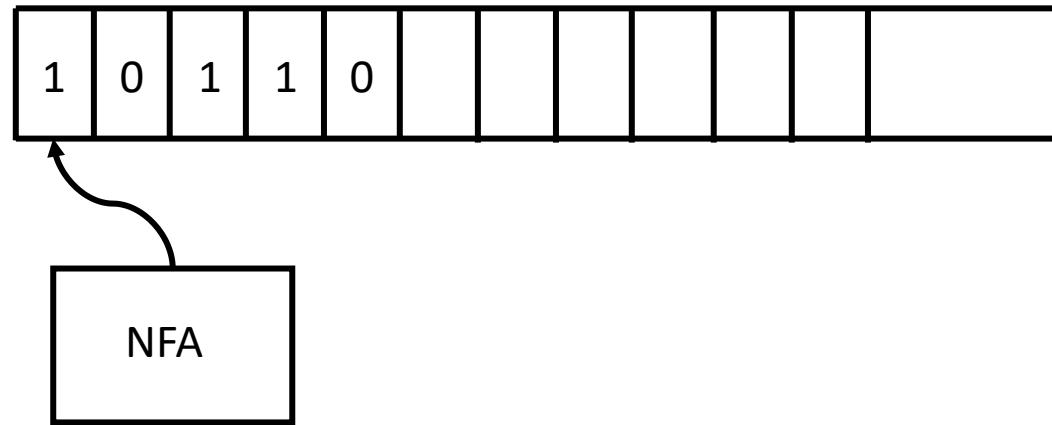
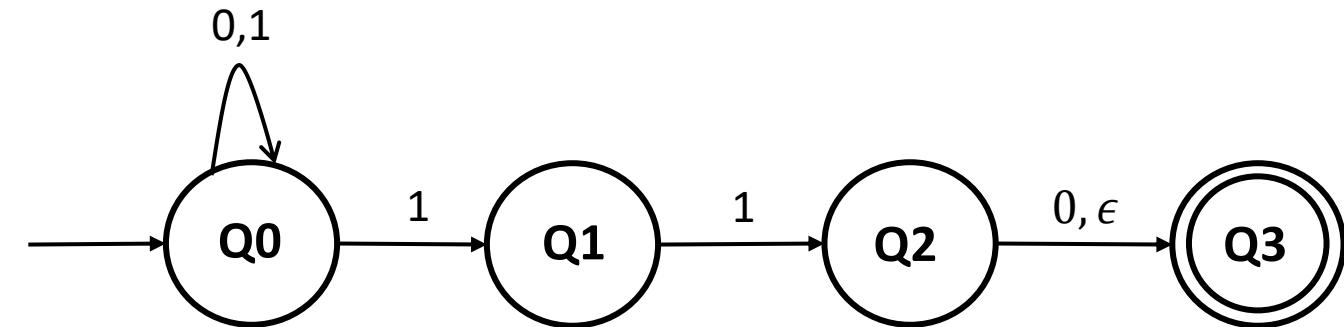
**Run 2:**  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_1 \xrightarrow{1} Q_2 \xrightarrow{0} Q_3$  (**ACCEPT**)

**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**)

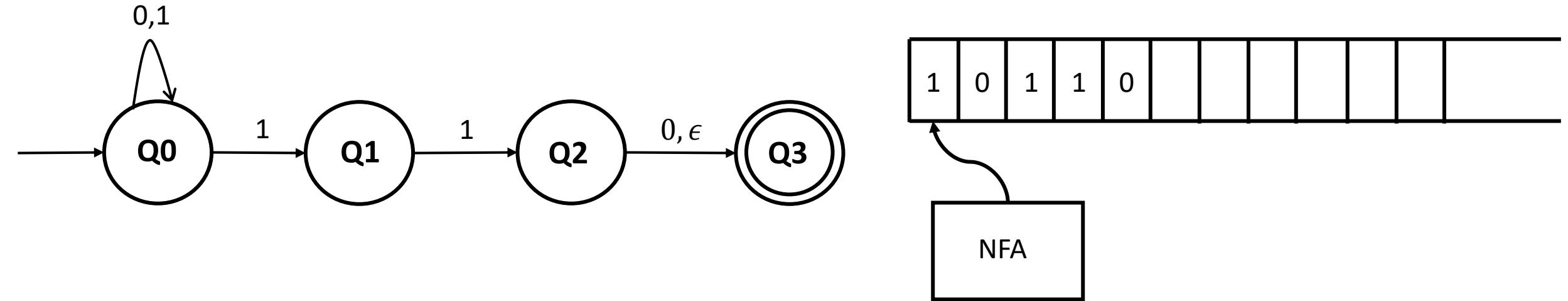
Run 3:  $Q_0 \xrightarrow{1} Q_0 \xrightarrow{0} Q_0 \xrightarrow{1} Q_0 \xrightarrow{1} Q_1 \xrightarrow{0}$  CRASH (**REJECT**)

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**)

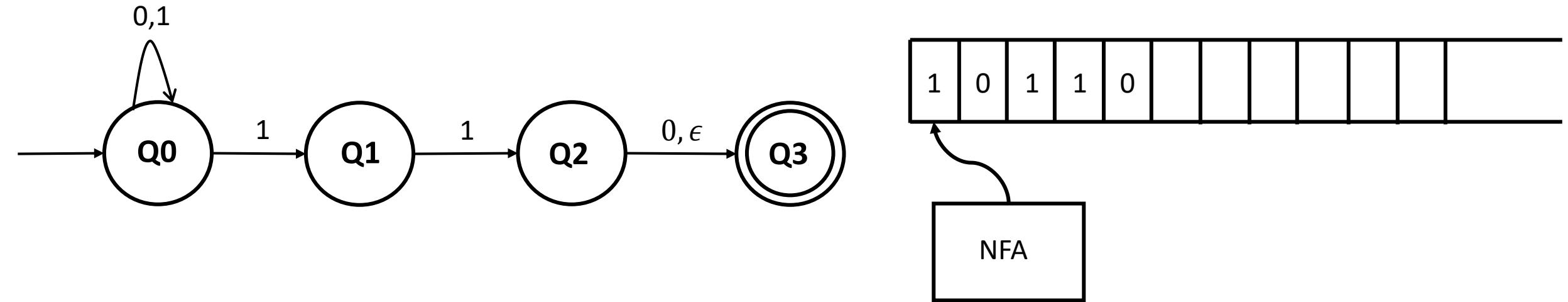
The NFA “rejects” an input string, if there are **no runs** that end up in a final state. (**Rejecting Run**)

# Non-deterministic Finite Automata (NFA)



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

# 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**”. We will learn about this in the next lecture.

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

**Thank You!**

# CS 302.1 - Automata Theory

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad

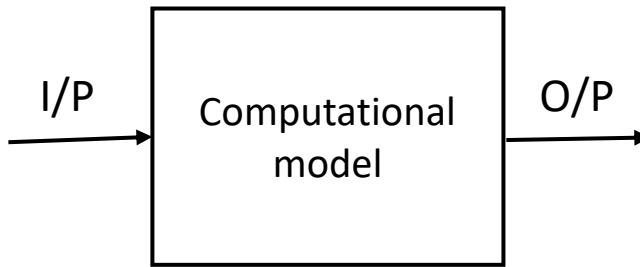


INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

H Y D E R A B A D

# A quick recap

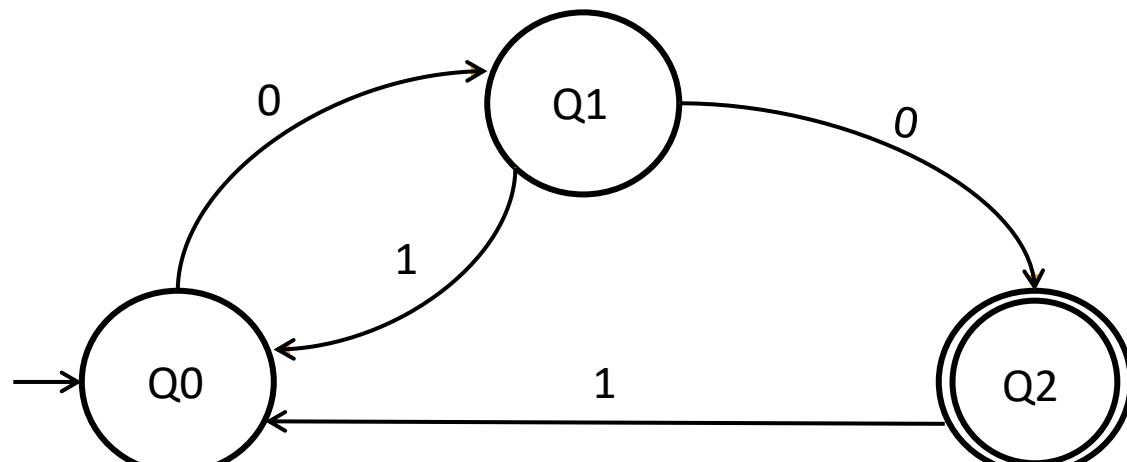
- Can a given problem be computed by a particular computational model?



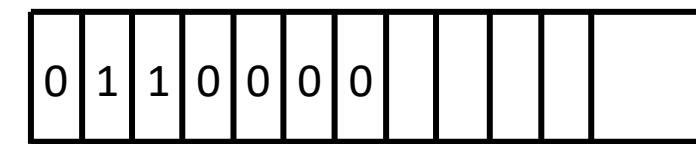
A computational model solves a problem P if,

- (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)



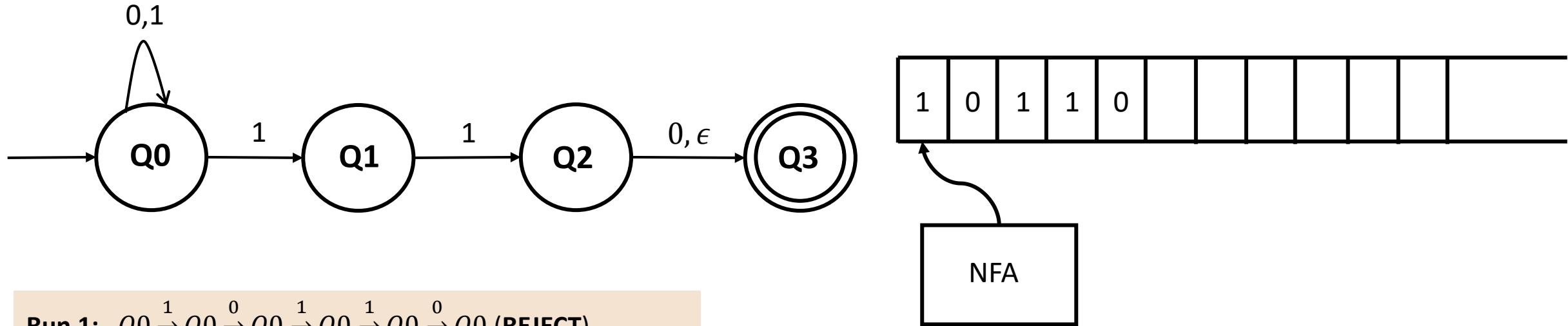
**Run:**

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

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

# A quick recap

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**)

- Multiple runs per input possible.
- The NFA “accepts” an input string, if there exists at **least one accepting run**

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

	0	1	ε
Q0	Q0	Q0, Q1	
Q1		Q2	
Q2	Q3		Q3
Q3			

# 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?

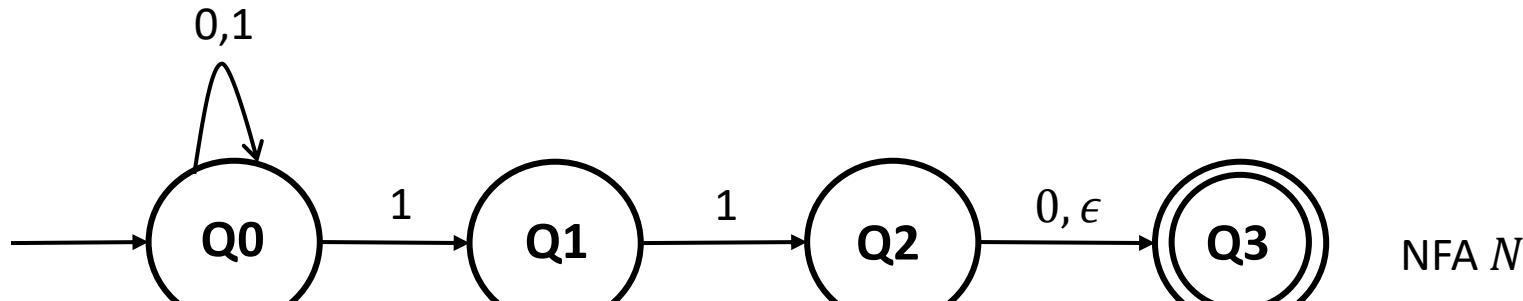
# Converting an NFA to a DFA

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

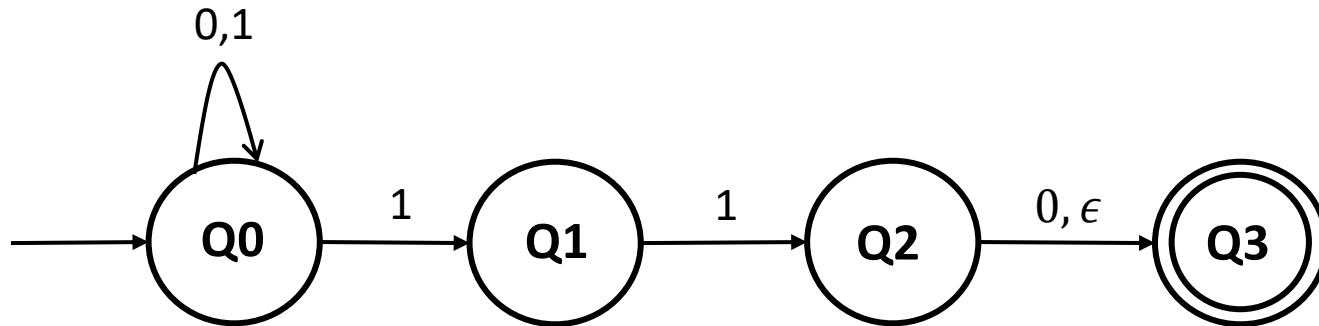
- $R$  on an input enters a state that is labelled by all possible states that  $N$  can enter on that input.



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

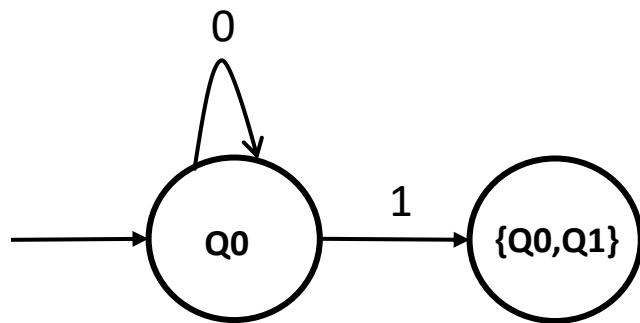
# Converting an NFA to a DFA

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



NFA  $N$

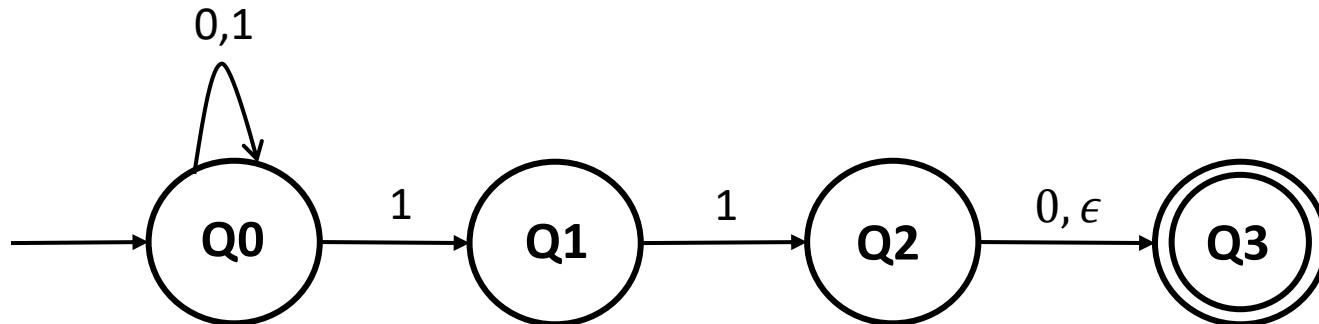
	0	1	$\epsilon$
$Q_0$	$Q_0$	$Q_0, Q_1$	
$Q_1$		$Q_2$	
$Q_2$	$Q_3$		$Q_3$
$Q_3$			



Remembering DFA  $R$

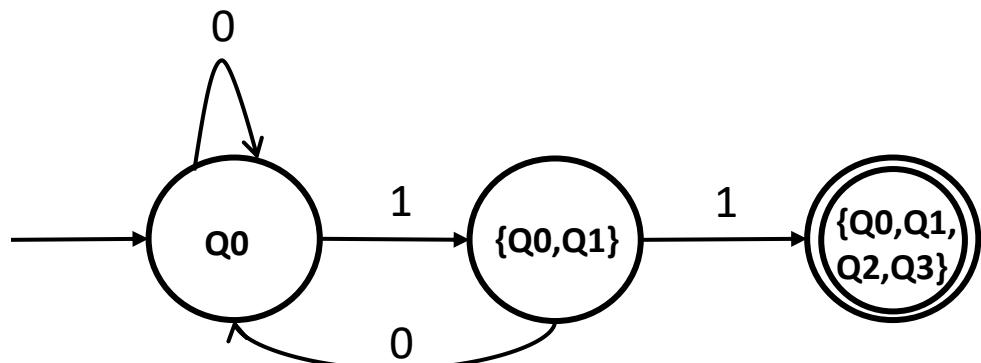
# Converting an NFA to a DFA

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



NFA  $N$

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

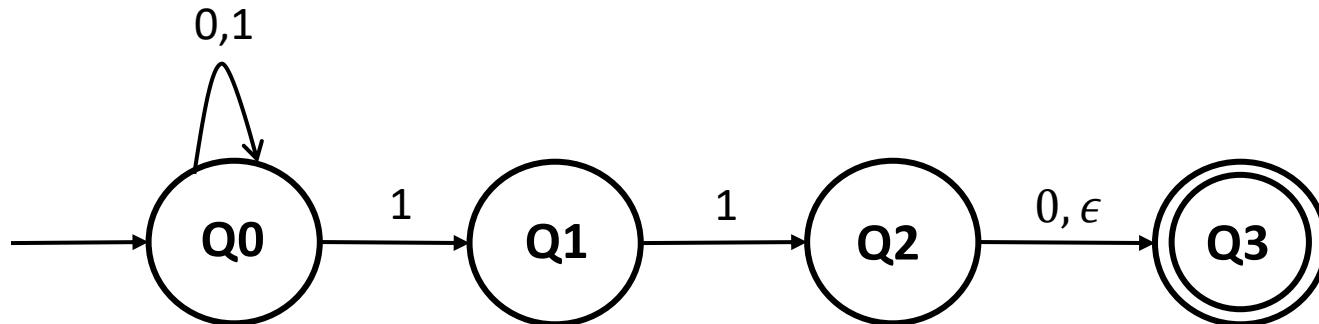


Remembering DFA  $R$

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

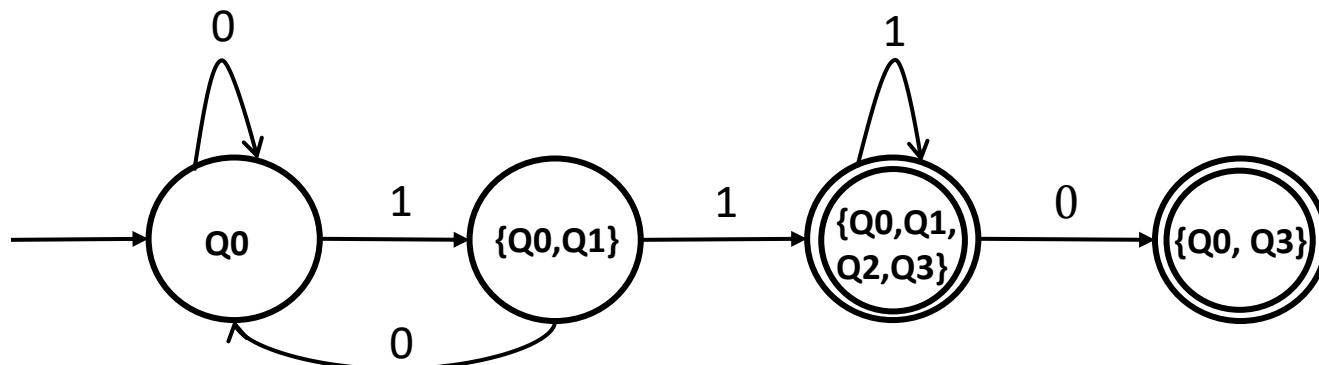
# Converting an NFA to a DFA

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



NFA  $N$

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

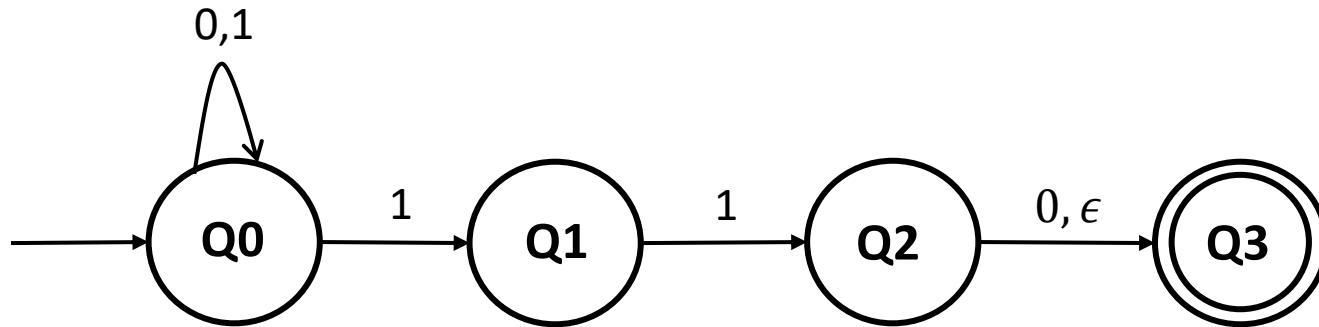


Remembering DFA  $R$

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

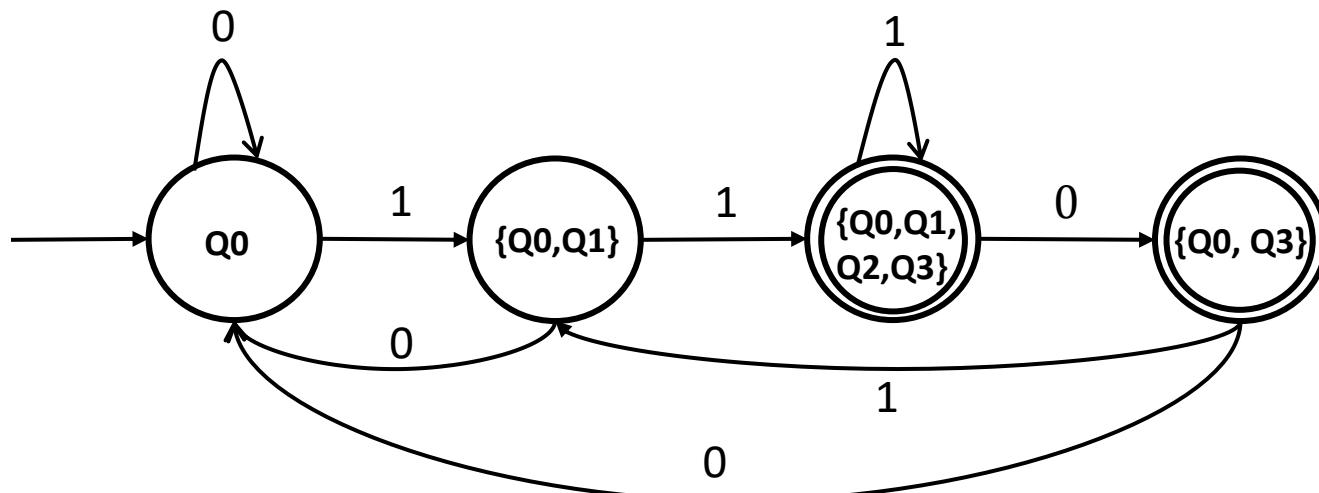
# Converting an NFA to a DFA

- $M_2$  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	$\epsilon$
$Q_0$	$Q_0$	$Q_0, Q_1$	
$Q_1$		$Q_2$	
$Q_2$	$Q_3$		$Q_3$
$Q_3$			

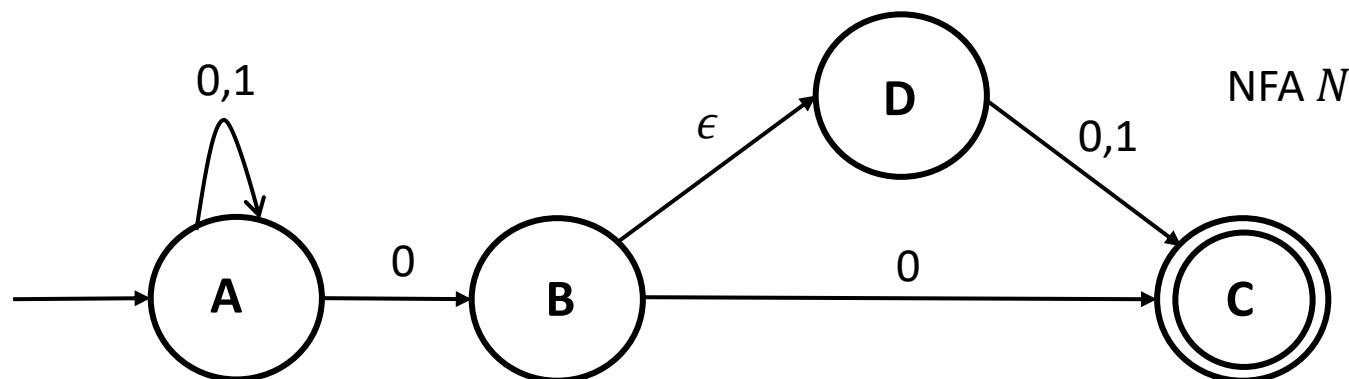


Remembering DFA  $R$

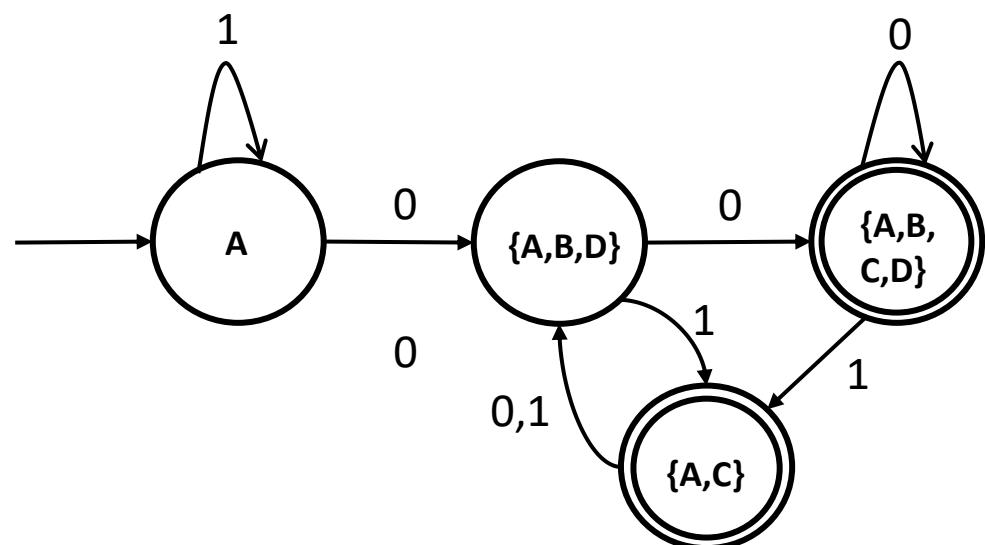
**Any state of  $R$  that contains in its label, an accepting state of  $N$  is an accepting state of  $R$ .**

# 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.



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



Remembering DFA  $R$

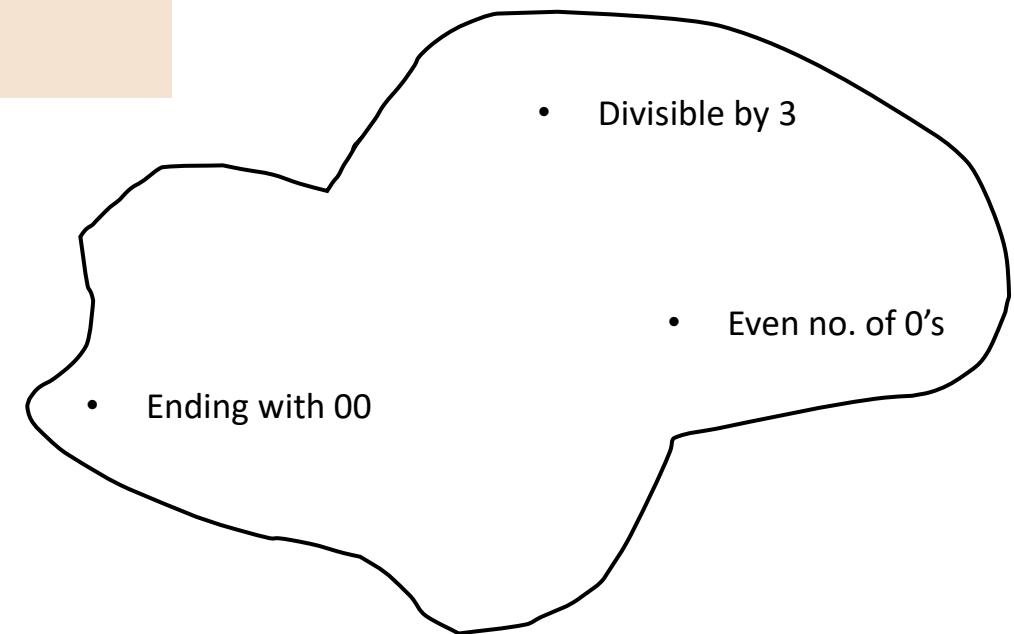
# Regular Languages

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

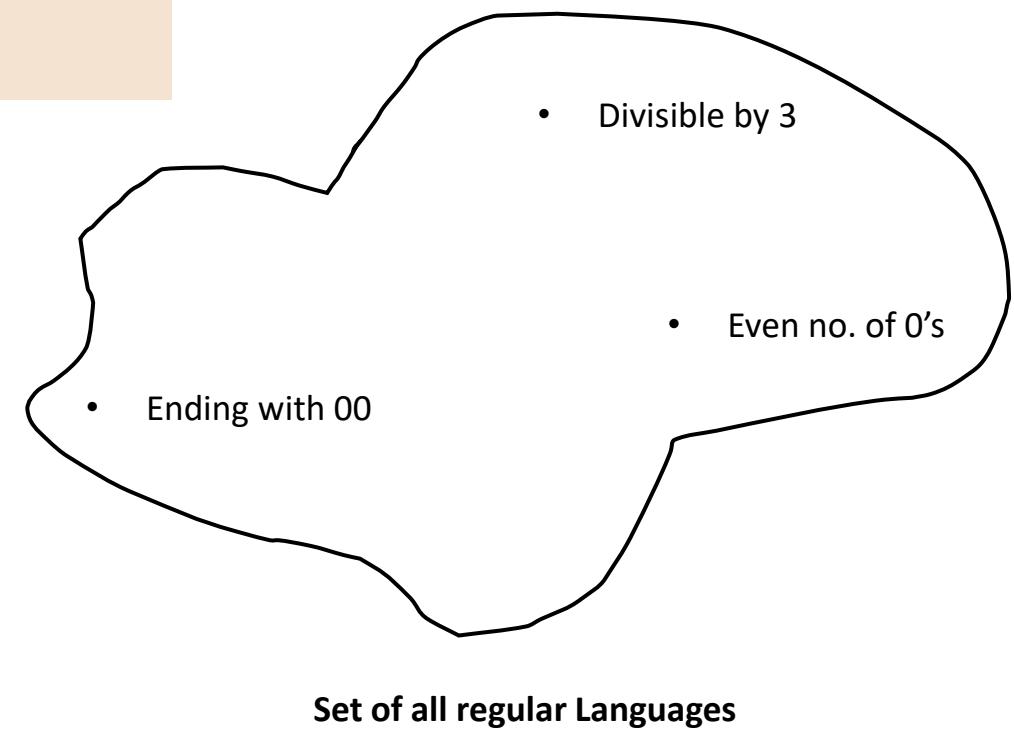
A language is called a **Regular Language** if there exists some finite automata recognizing it.

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

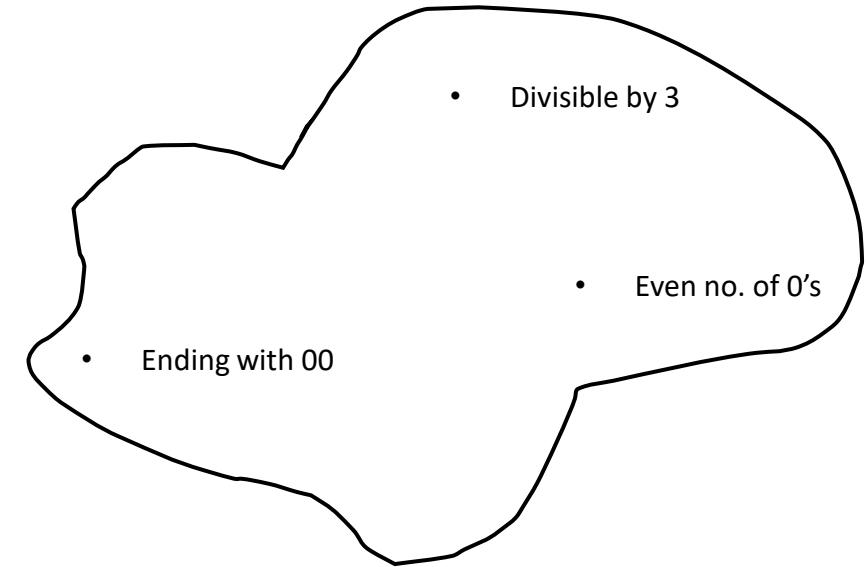


# Regular Languages

## 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:**  $L_1^* = \{x_1x_2 \cdots x_k | k \geq 0 \text{ and each } x_i \in L_1\}$



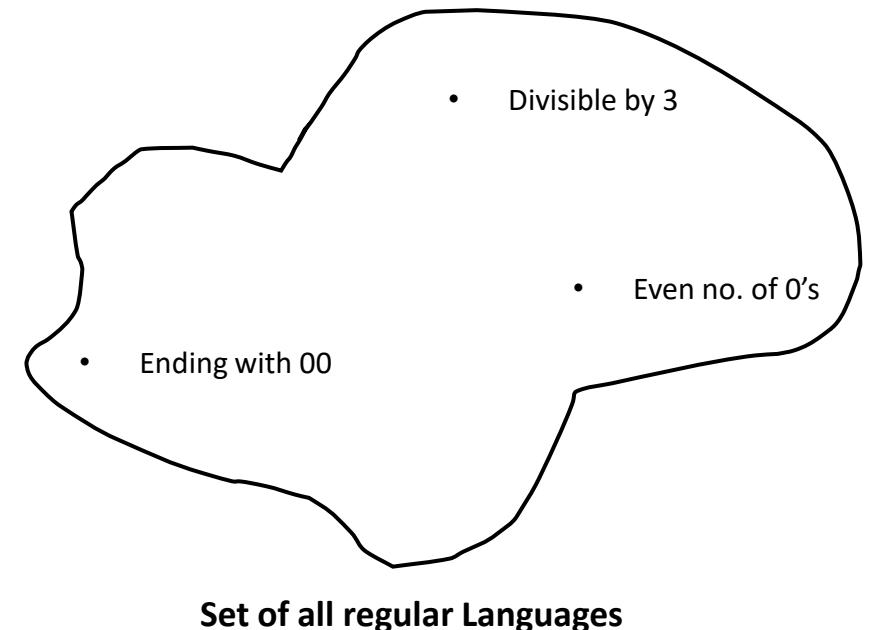
Set of all regular Languages

# 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  $L_1^*$ .

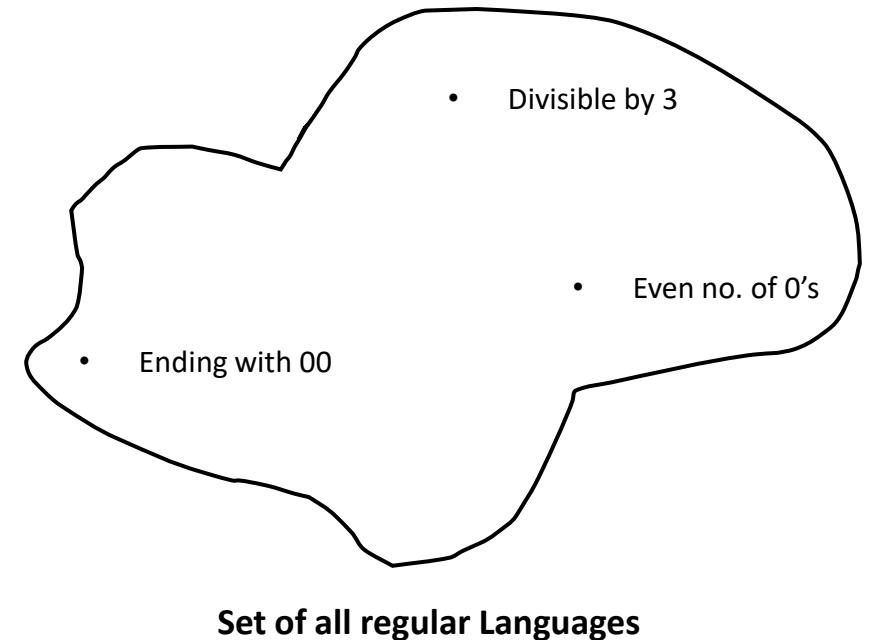
$$\text{If } \Sigma = \{a\}, \Sigma^* = \{\epsilon, a, aa, aaa, \dots \dots \} ; \text{ If } \Sigma = \{\Phi\}, \Sigma^* = \{\}$$

# 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:**  $L_1^* = \{x_1x_2 \cdots x_k | k \geq 0 \text{ and each } x_i \in L\}$

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

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

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

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

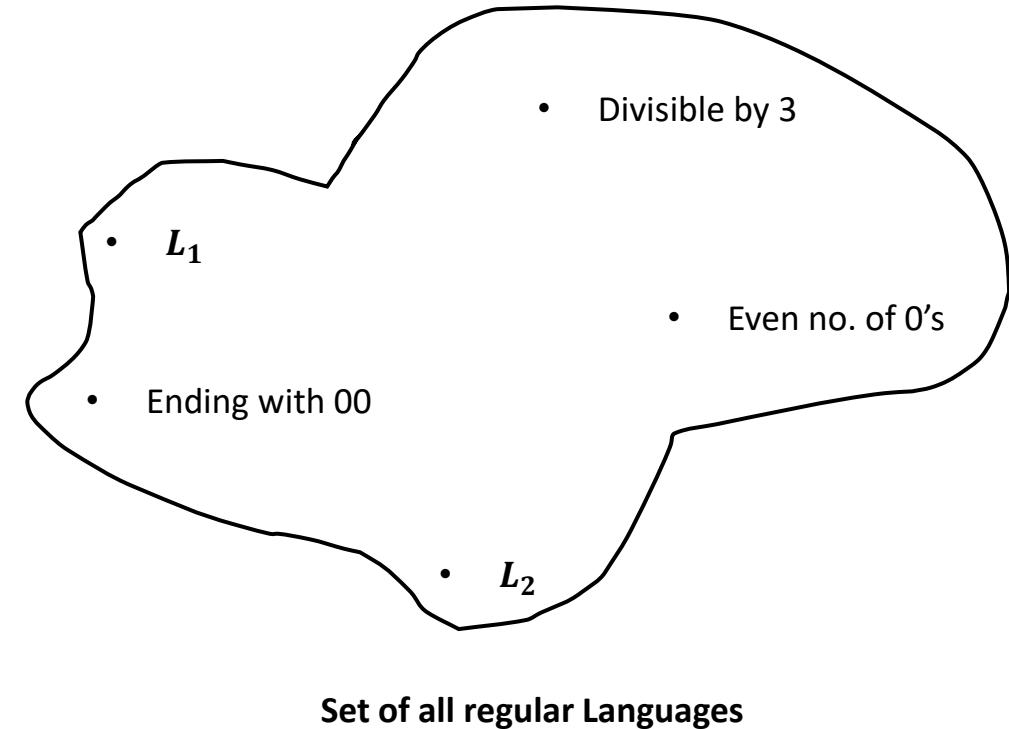
- $L_1 \cup L_2 = \{\text{social, economic, justice, reform}\}$
- $L_1 \cdot L_2 = \{\text{socialjustice, socialreform, economicjustice, economicreform}\}$
- $L_1^* = \{\epsilon, \text{social, economic, socialsocial, socialeconomic, economicsocial, economiceconomic, socialsocialsocial, socialsocialeconomic,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

# 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, compliment 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.

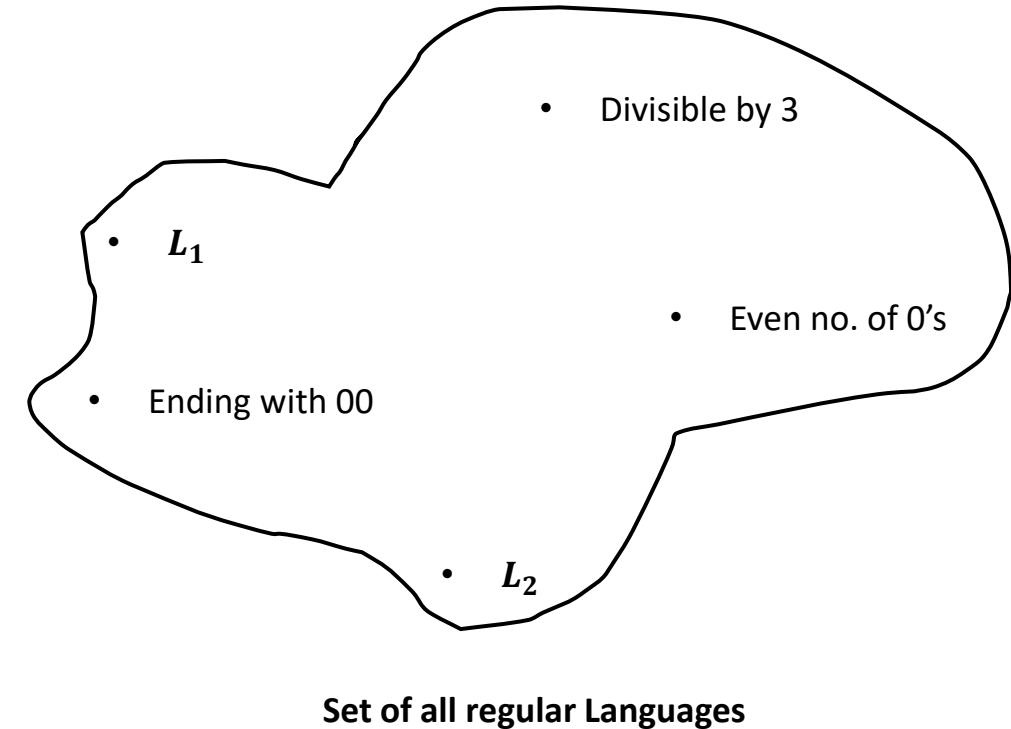


# Closure of Regular Languages

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, compliment 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.

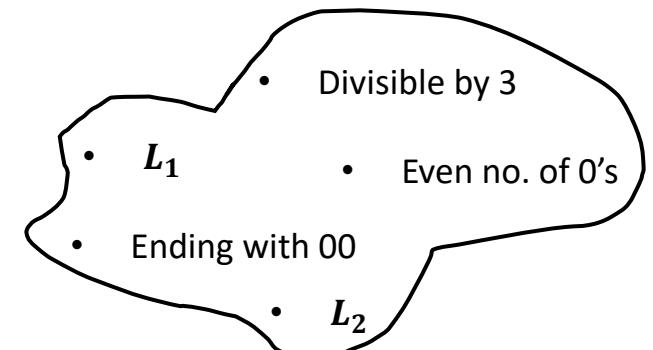


For example, the **natural numbers** are closed under addition/multiplication and not under subtraction/division.

# 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

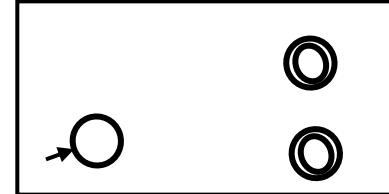
**Q:** Is the set of all regular languages **closed under union?**

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

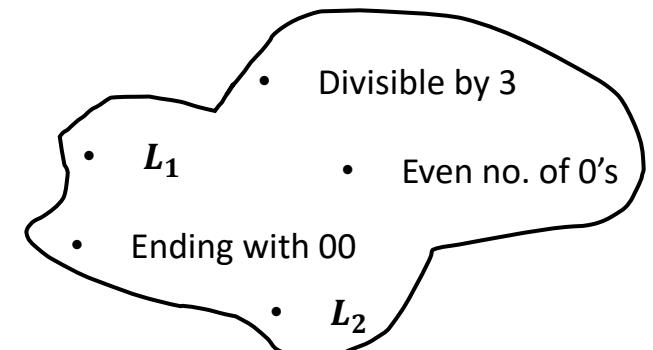
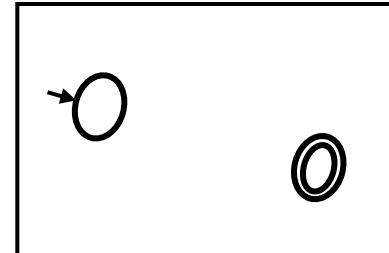
**Proof:** Since  $L_1$  and  $L_2$  are regular, there must be a DFA  $M_1$  that accepts  $L_1$ , i.e.  $L(M_1) = L_1$  and a DFA  $M_2$  that accepts  $L_2$ , i.e.  $L(M_2) = L_2$ .

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

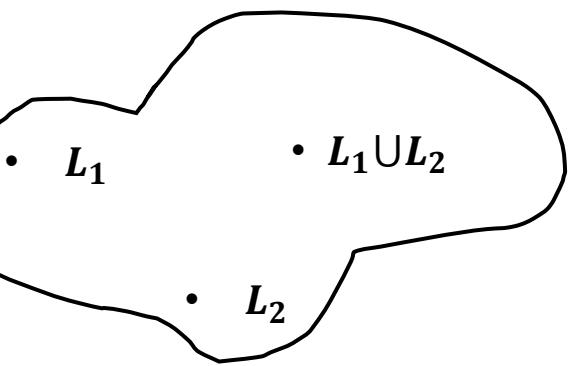
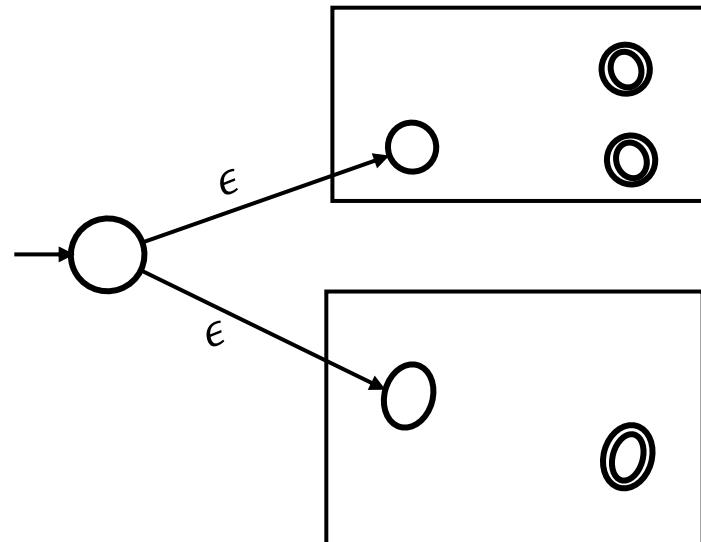
**Q:** Is the set of all regular languages **closed under union?**

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

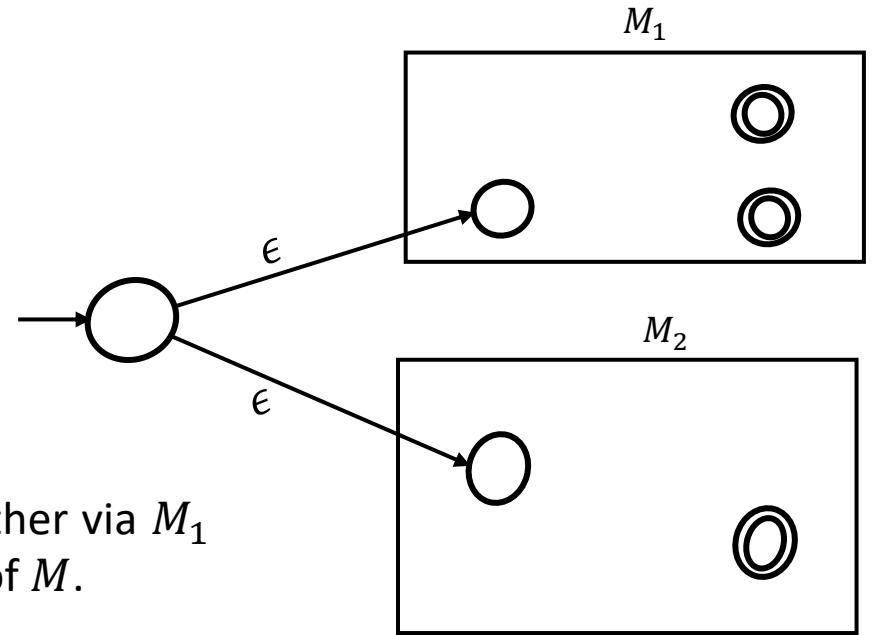
**Q:** Is the set of all regular languages **closed under union?**

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:

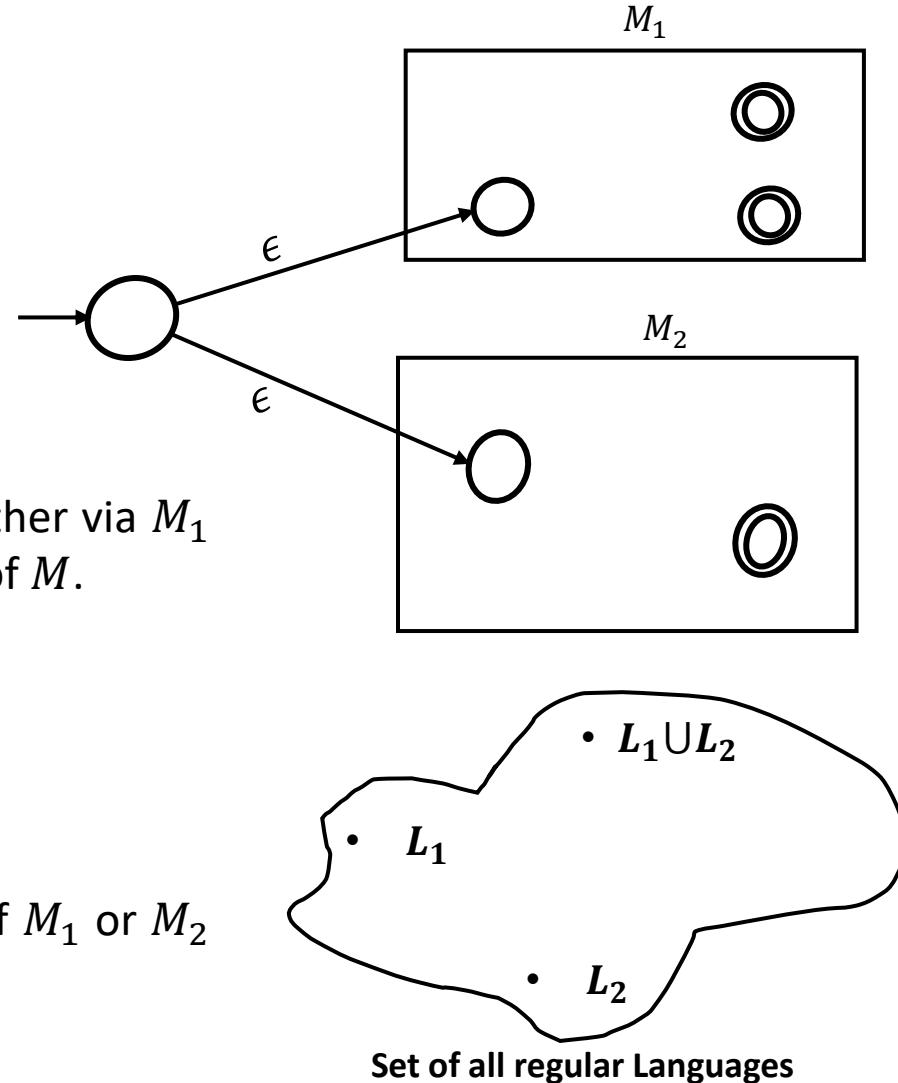
(i)  $L \subseteq L_1 \cup L_2$

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,  $\omega \in L$ .

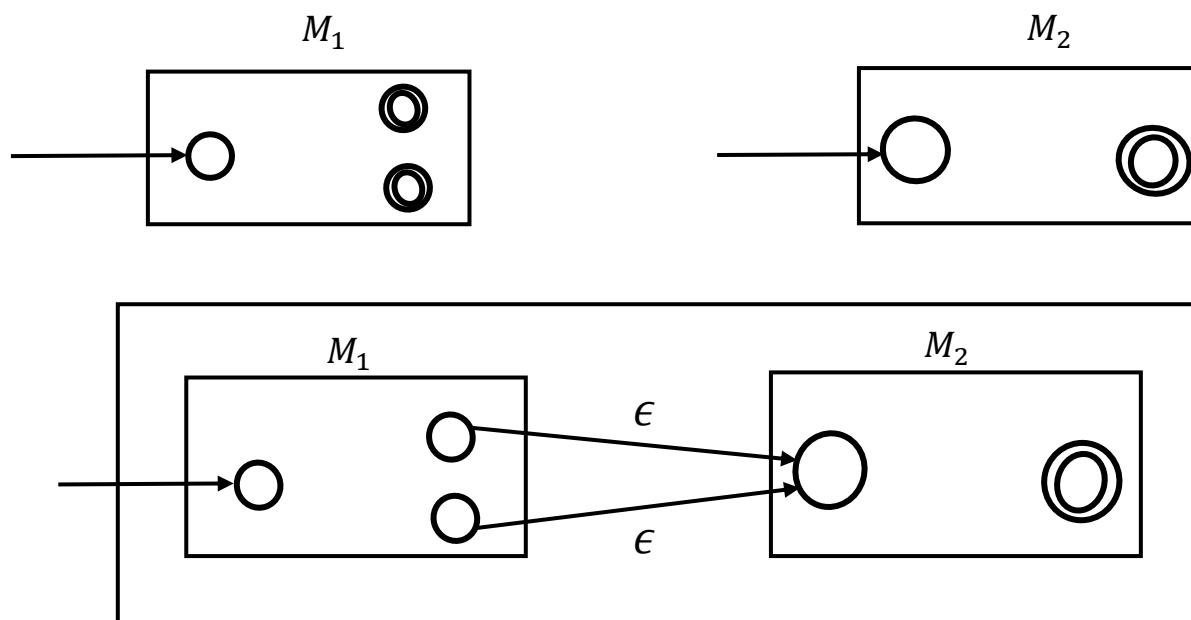


# Closure of Regular Languages

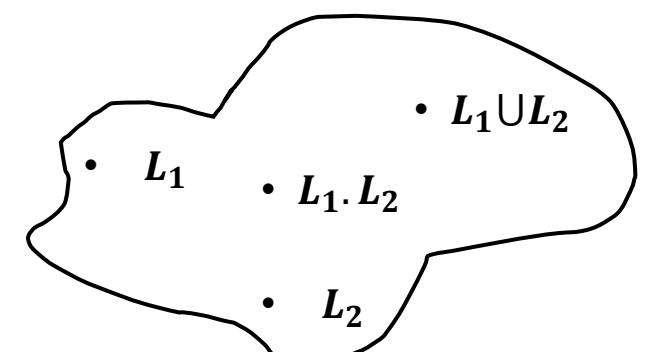
**Q:** Is the set of all regular languages **closed under concatenation**? Suppose  $L_1$  and  $L_2$  are regular languages. Is  $L = L_1 \cdot L_2$  also regular?

**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  $L = L_1 \cdot L_2$



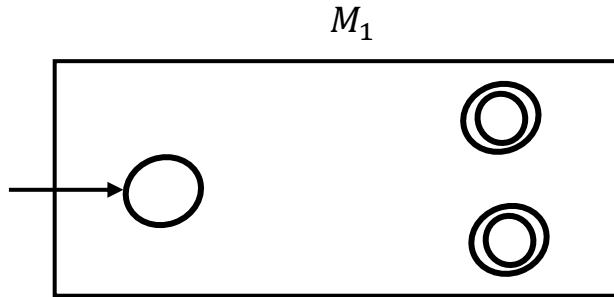
Set of all regular Languages

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

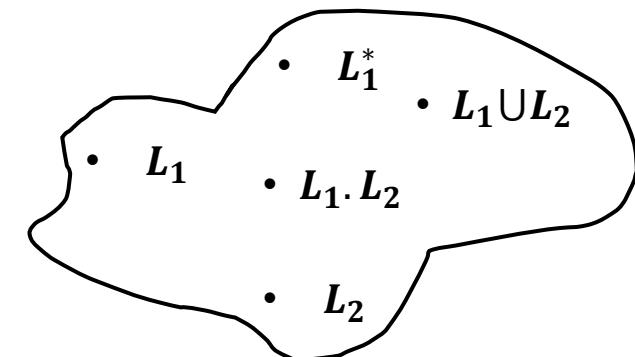
# Closure of Regular Languages

**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  $L = L_1^*$ .



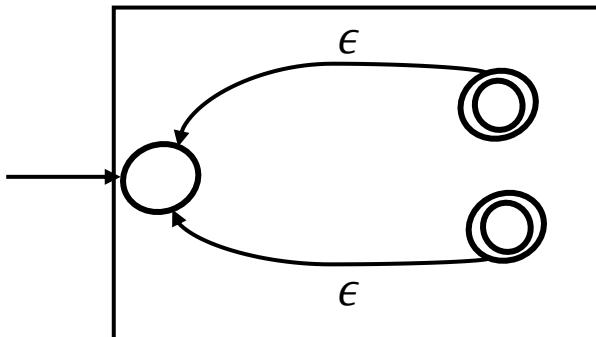
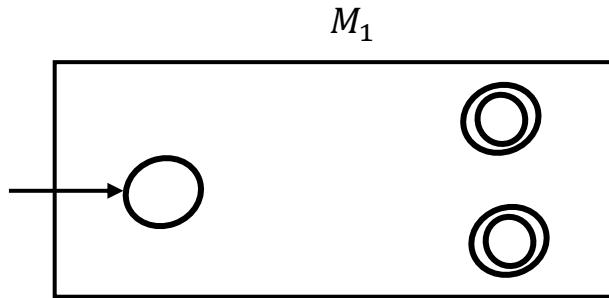
$$L_1^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$$



# 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  $L = L_1^*$ .

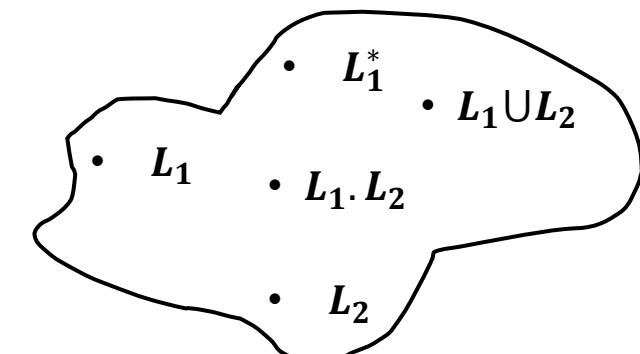


NFA accepting  $L = L_1^*$

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

**Steps:**

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

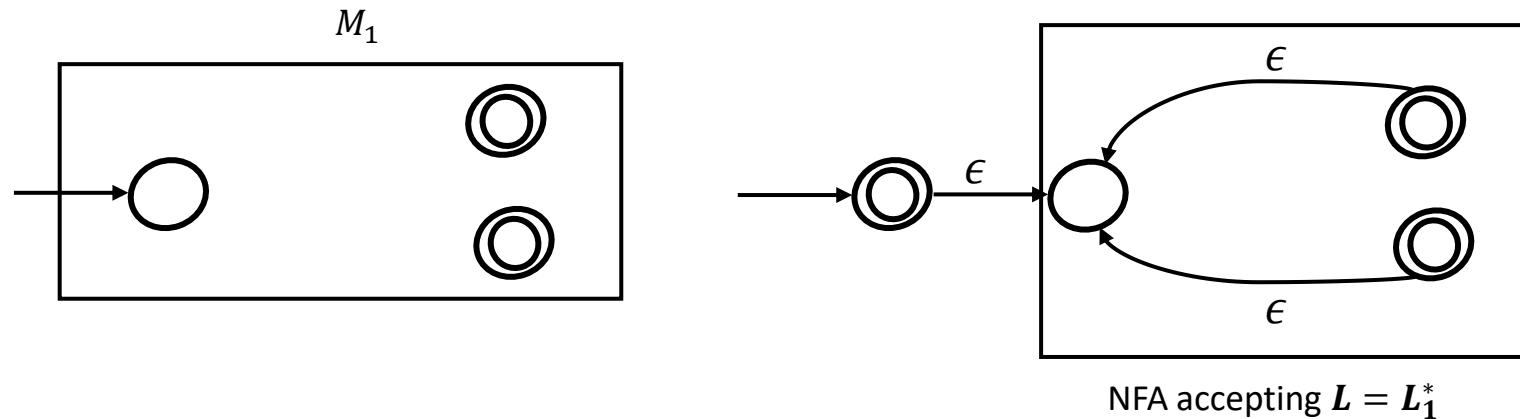


Set of all regular Languages

# 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?

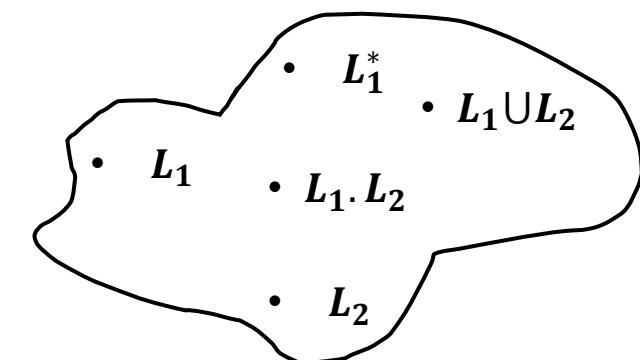
**Proof:** Since  $L_1$  is regular, there must be a DFA  $M_1$  that accepts  $L_1$ , i.e.  $L(M_1) = L_1$ . Using  $M_1$ , we will show how to construct an NFA  $M$  that accepts  $L = L_1^*$ .



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

**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

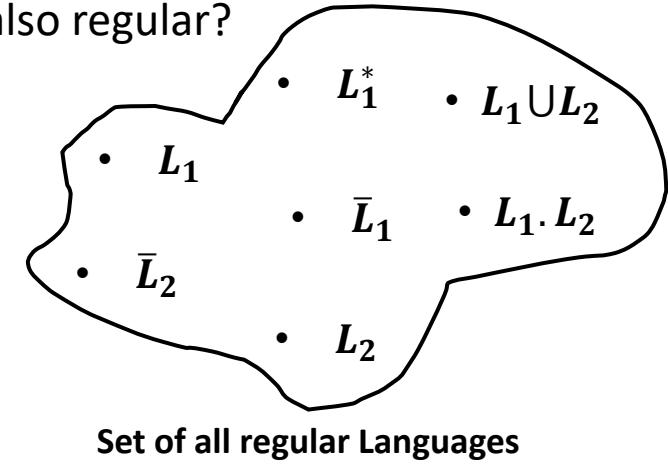
# Closure of Regular Languages

**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  $M'$ .

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



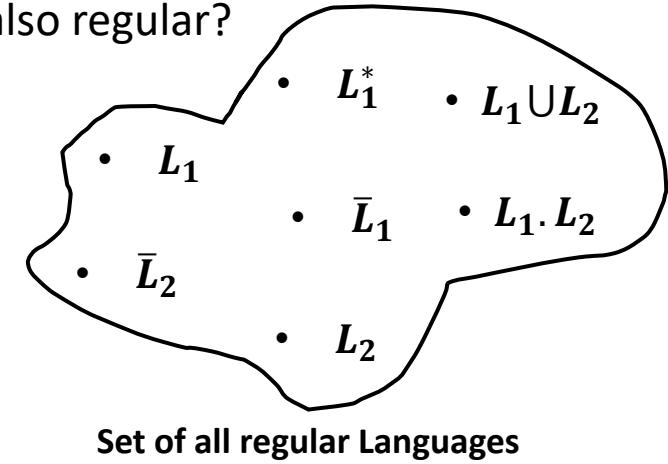
# Closure of Regular Languages

**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  $\bar{L}$ ?

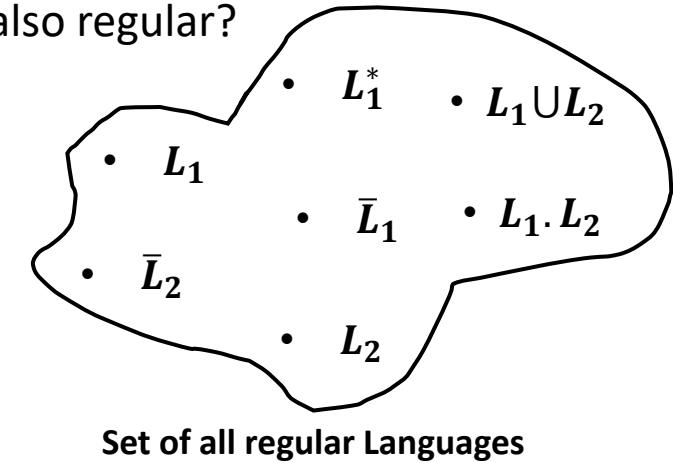
# Closure of Regular Languages

**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	

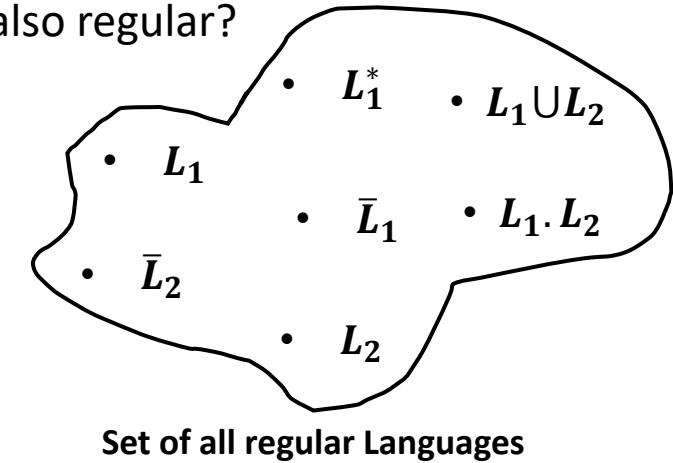
# Closure of Regular Languages

**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

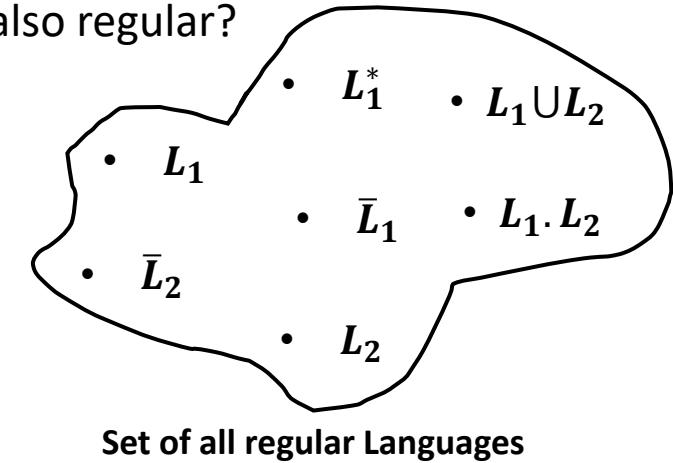
# Closure of Regular Languages

**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

# Closure of Regular Languages

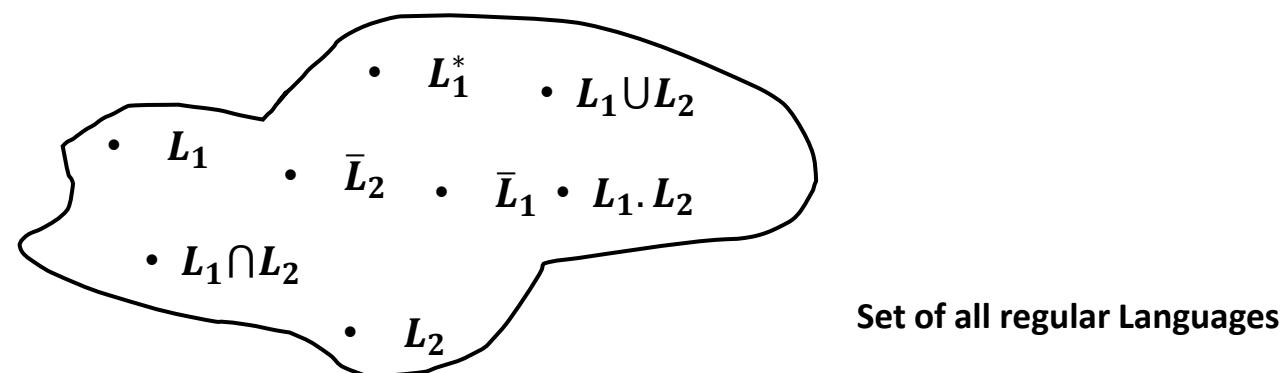
**Q:** Is the set of all regular languages **closed under intersection**? If  $L_1$  and  $L_2$  are regular, then is  $L = L_1 \cap L_2$  also regular?

**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  $L = L_1 \cap L_2$ .

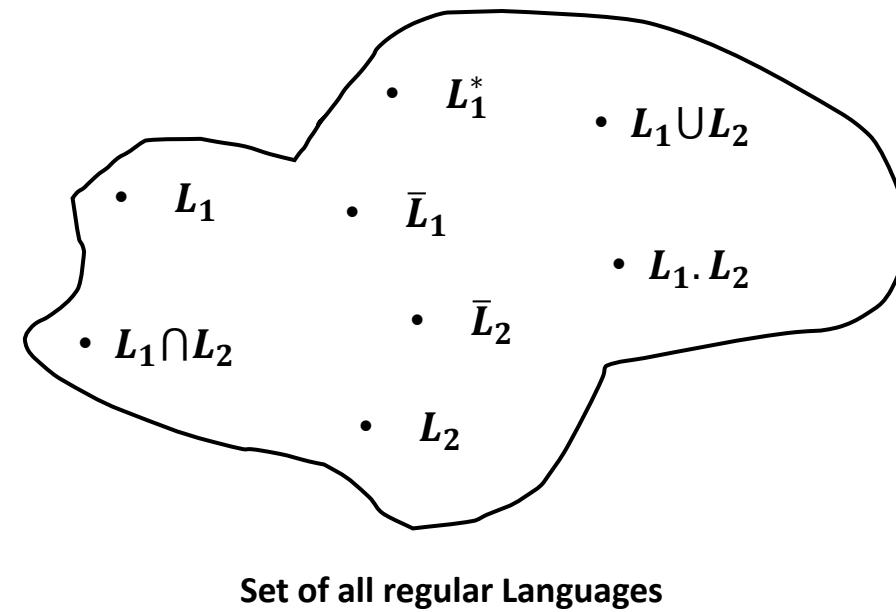


# Closure of Regular Languages

## Summary:

Regular Languages are closed under:

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



# Regular Languages

If  $\Sigma$  is an alphabet, then

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

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

# Regular Languages

If  $\Sigma$  is an alphabet, then

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

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

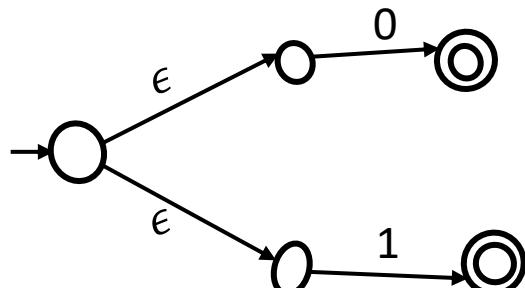
Syntax for regular expressions:

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, ... }

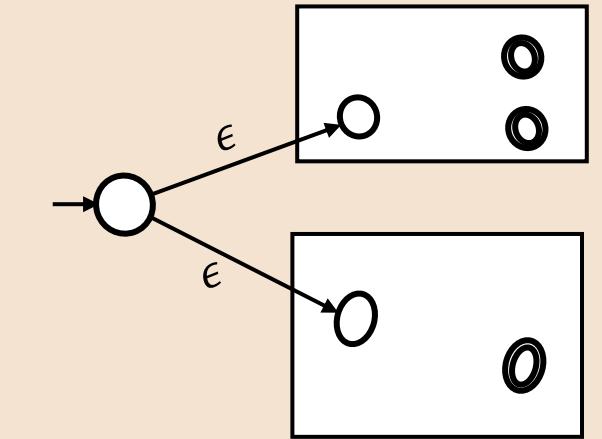
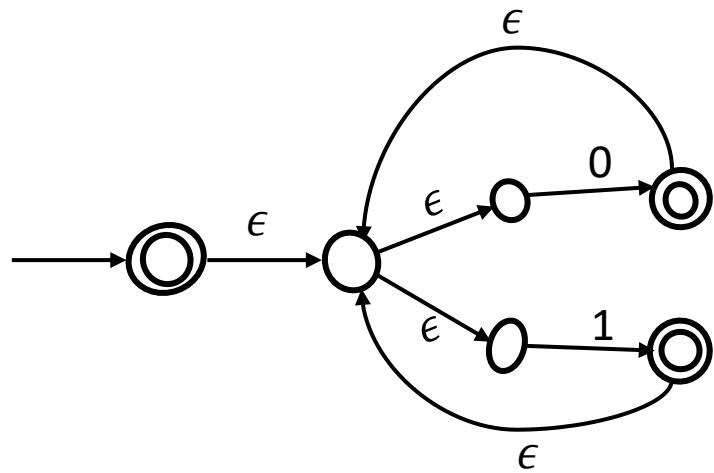
# Regular Expressions

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

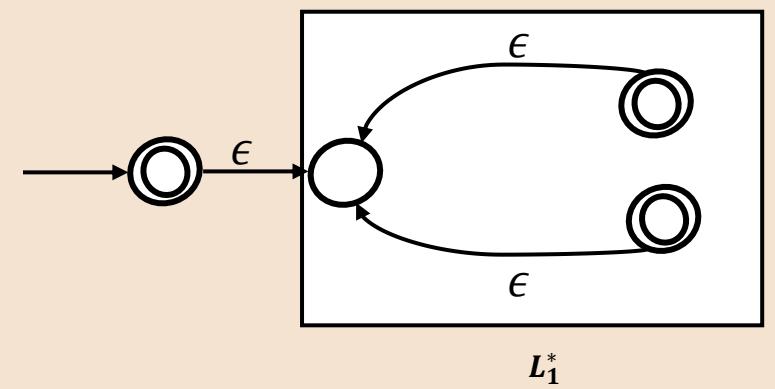
(i) NFA for  $(0 + 1)$



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



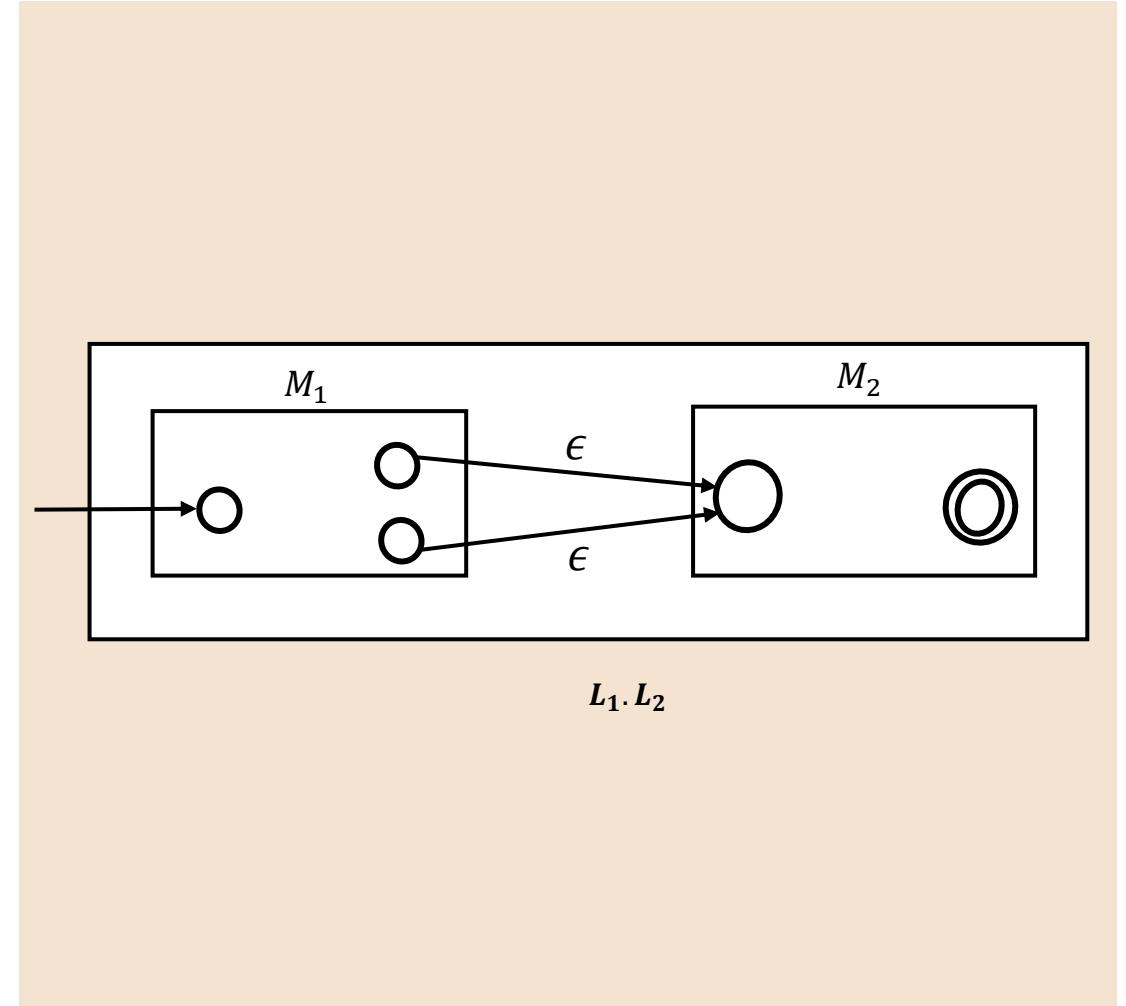
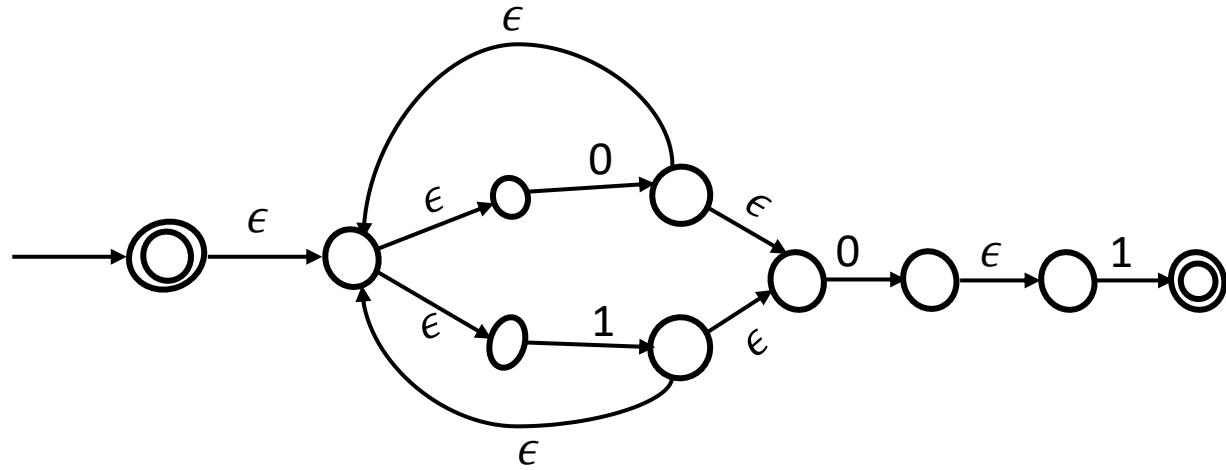
$L_1 \cup L_2$



$L_1^*$

# 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:

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

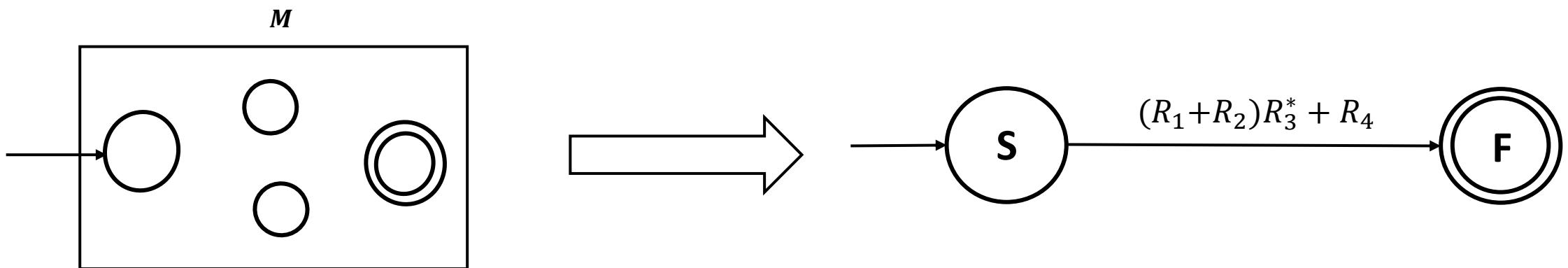
# DFA to Regular Expressions

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  $M$ .



**Thank You!**

# CS 302.1 - Automata Theory

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad

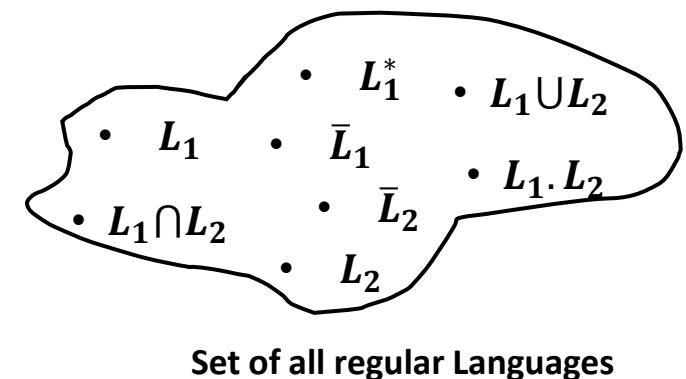


INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

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.
- RL can also be derived from first principles.
- Regular Languages are closed under: Union, Star, Complement, Intersection...
- Regular expressions provide an elegant algebraic framework to represent regular languages.
- We can construct NFAs given a Regular Expression.



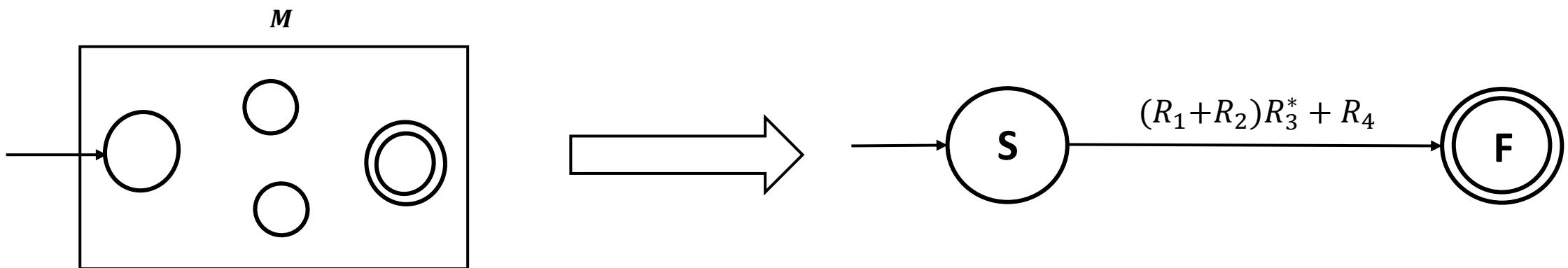
# DFA to Regular Expressions

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

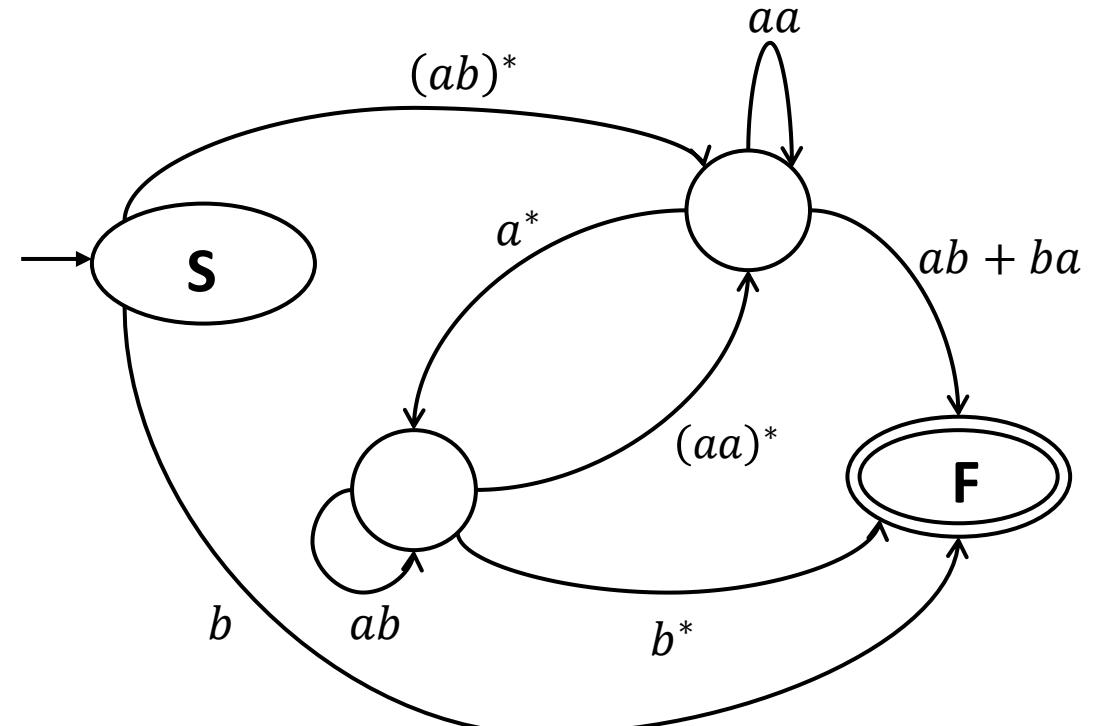
- A start state and a final state
- A single arrow goes from the start state to the final state
- The label of this arrow is the regular expression corresponding to the language accepted by the DFA  $M$ .



# DFA to Regular Expressions: GNFA

What are GNFs? They are simply NFAs such that

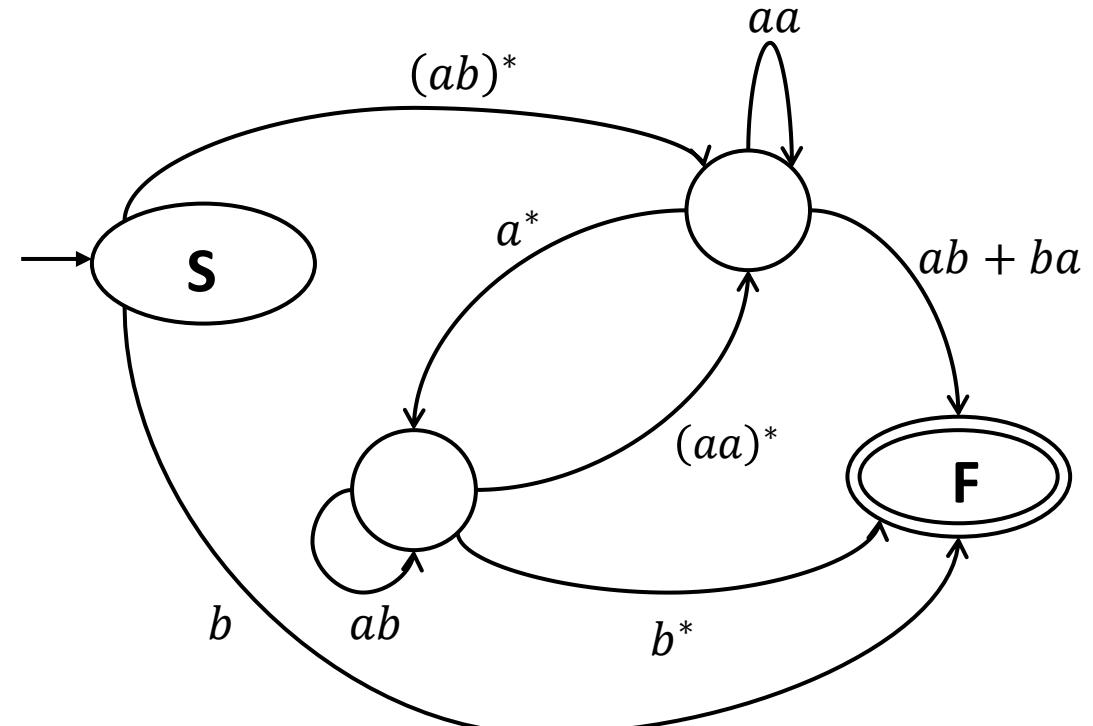
- The transitions may have regular expressions
- A unique start state that has arrows going to other states, but has no incoming arrows
- A unique final state that has arrows incoming from other states, but has no outgoing arrows
- For an input string, **runs** on a GNFA are similar to that of an NFA, 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

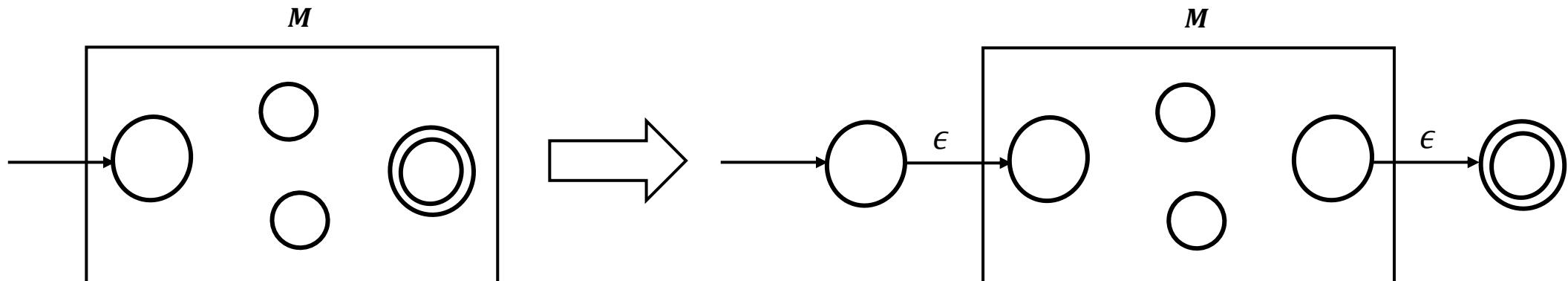


Starting from a DFA we will begin by constructing a GNFA with  $k$  states. We then outline a recursive procedure by which at each step, we will construct a GNFA with one less state. This step will be repeated until we obtain the **2-state GNFA**.

# DFA to Regular Expressions: GNFA

Starting from the DFA  $M$ ,

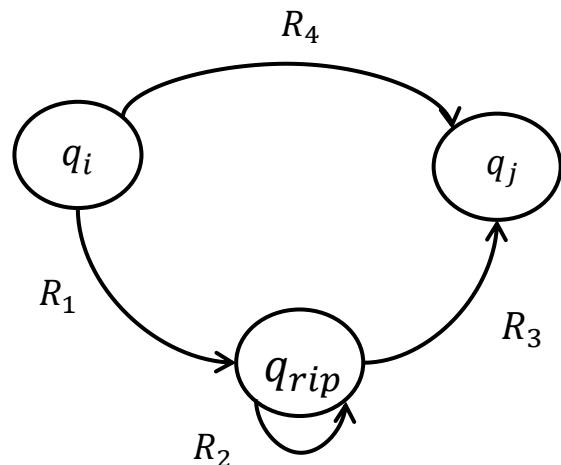
- Add a new start state with an  $\epsilon$  arrow to the old start state.
- Add a new final state by with an  $\epsilon$  arrow to the old final state.



# 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.

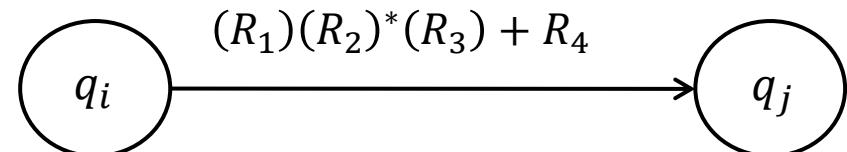
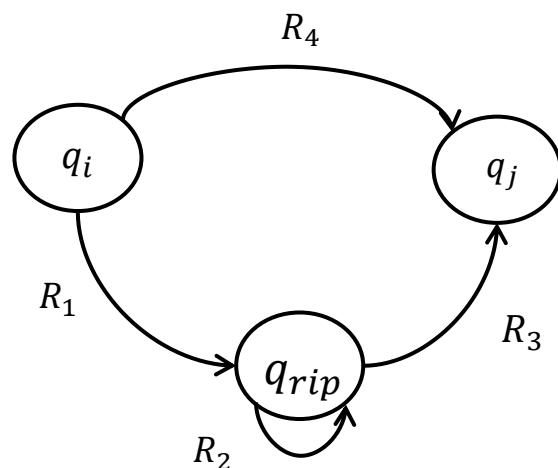
- Start by picking any state of the GNFA (except the new start and final states)
- Let us call this state  $q_{rip}$ . We “rip”  $q_{rip}$  out of the machine and create a GNFA with  $k - 1$  states.
- Of course, we need to “repair” the machine by altering the regular expressions that label each of the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

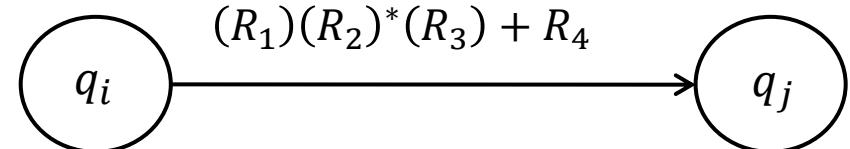
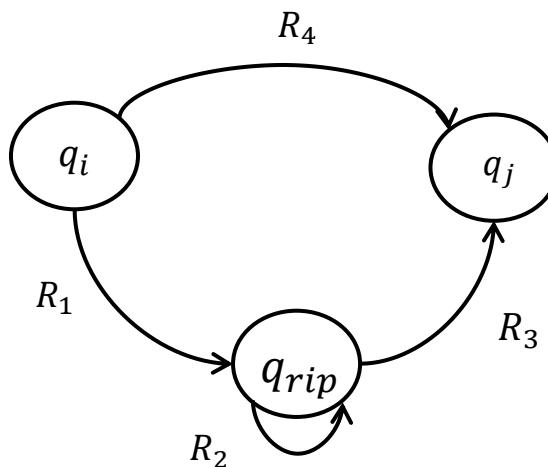
The crucial step is to convert a GNFA with  $k (>2)$  states to a GNFA with  $k - 1$  states.

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  $k = 2$**

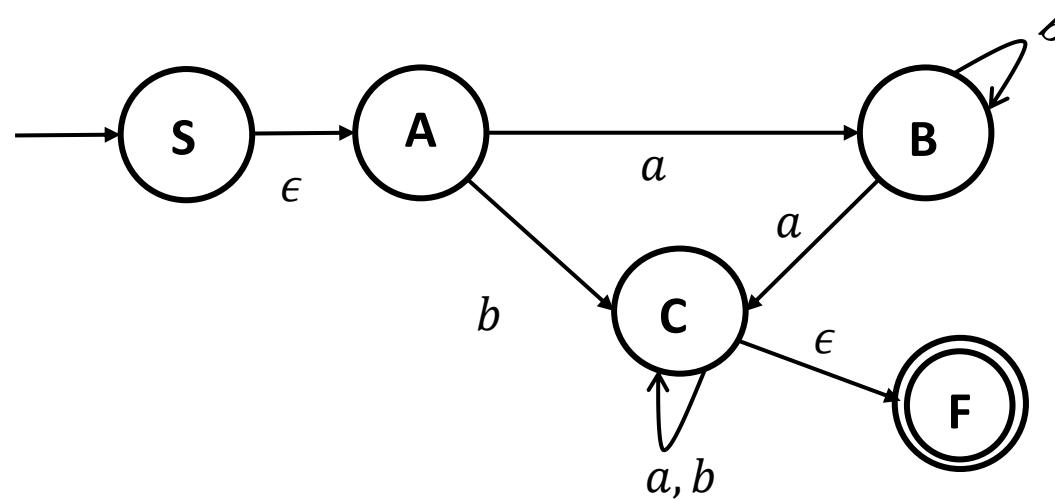
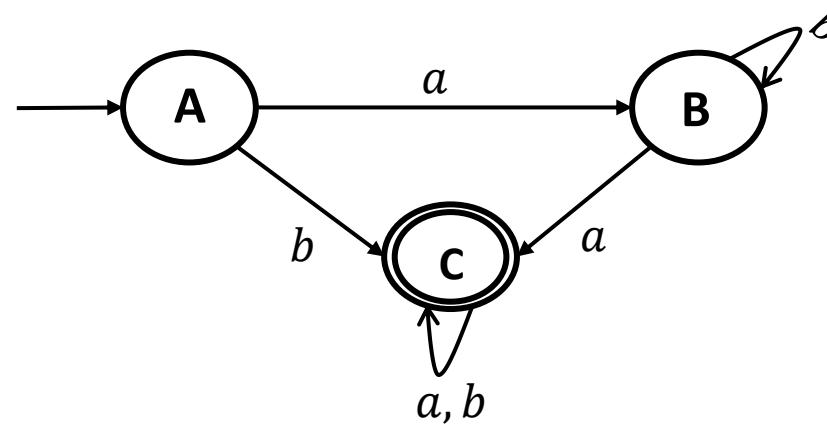
then in the new machine, the arrow from  $q_i$  to  $q_j$  has the label  $(R_1)(R_2)^*(R_3) + R_4$



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

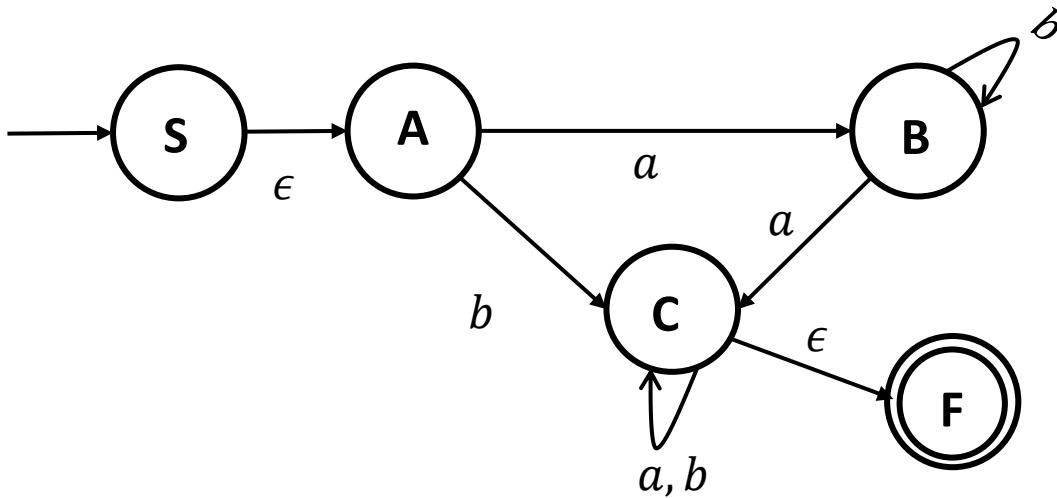
# DFA to Regular Expressions: GNFA

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

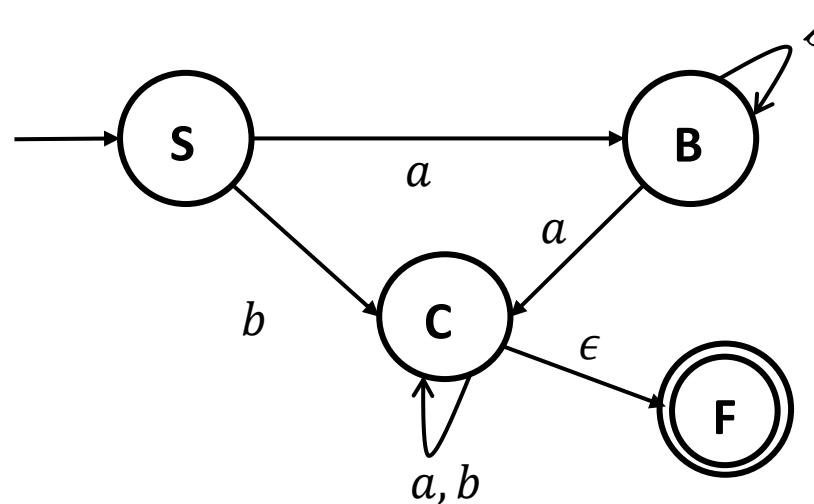


**Step 1: Add new start and final states**

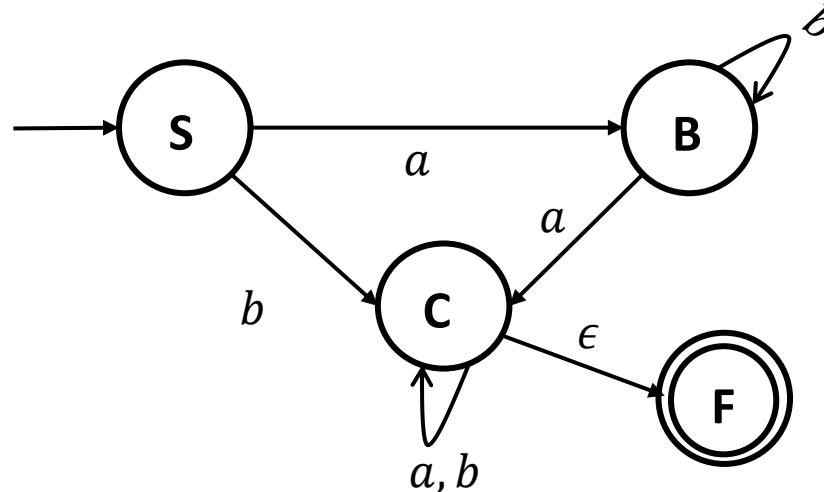
# DFA to Regular Expressions: GNFA



**Step 2: Eliminate A**

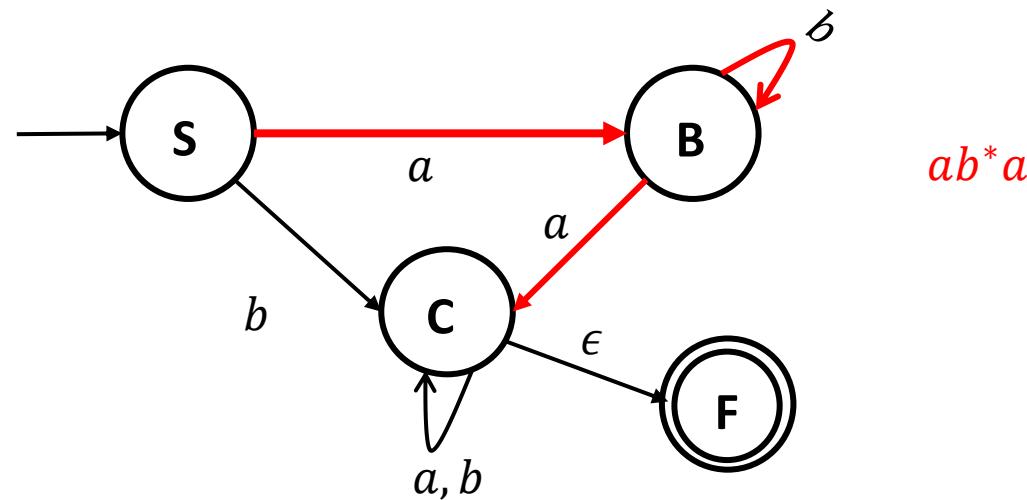


# DFA to Regular Expressions: GNFA

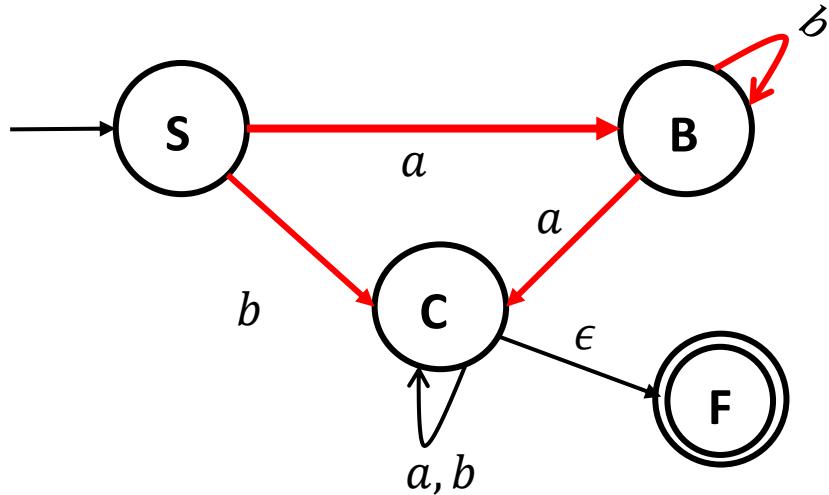


**Step 2: Eliminate B**

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



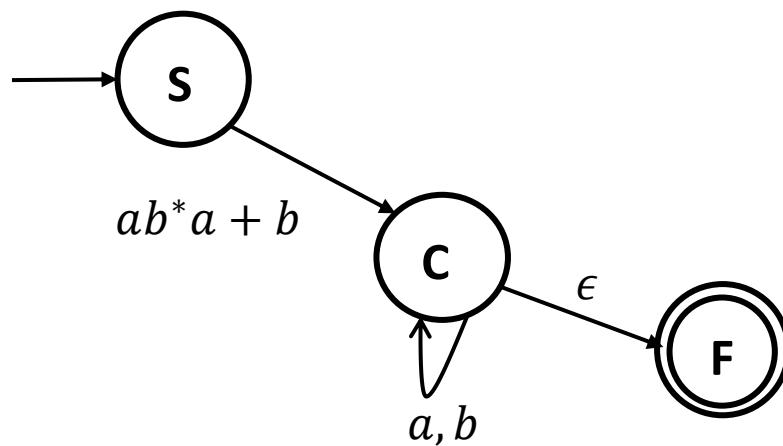
# DFA to Regular Expressions: GNFA



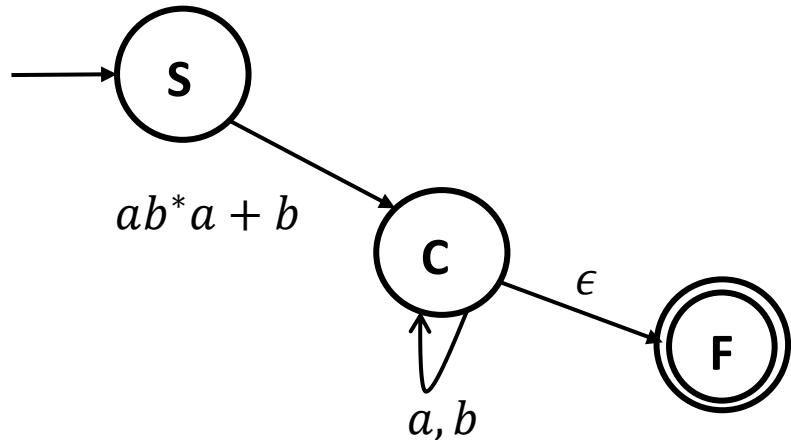
**Step 2: Eliminate  $B$**

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

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

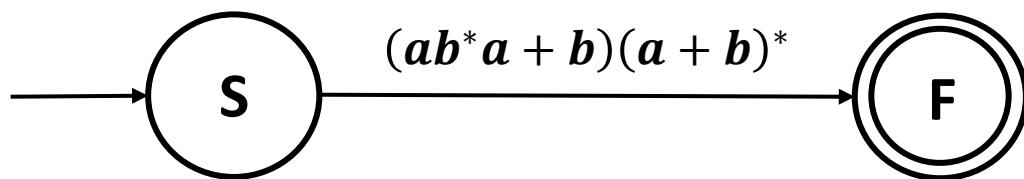


# DFA to Regular Expressions: GNFA

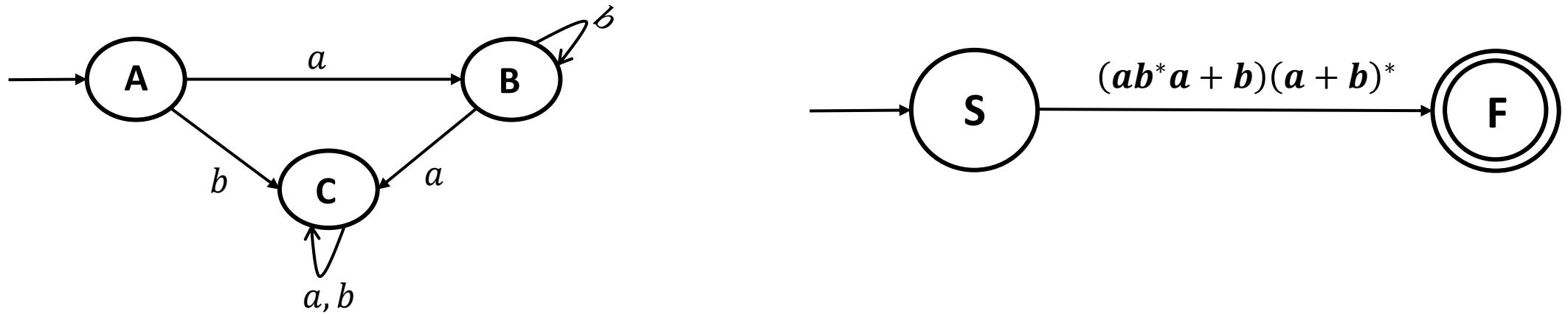


**Step 2: Eliminate C**

$S \rightarrow F$  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  $L(M)$ .

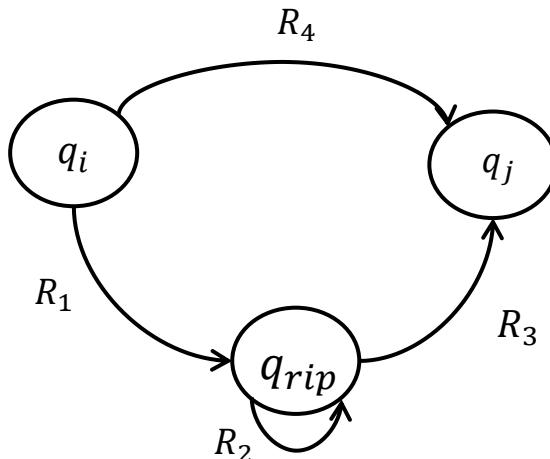
# DFA to Regular Expressions: GNFA

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  $(Q' \cup \{q_{rip}\}, \Sigma, \delta', q_0, F)$ , 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.

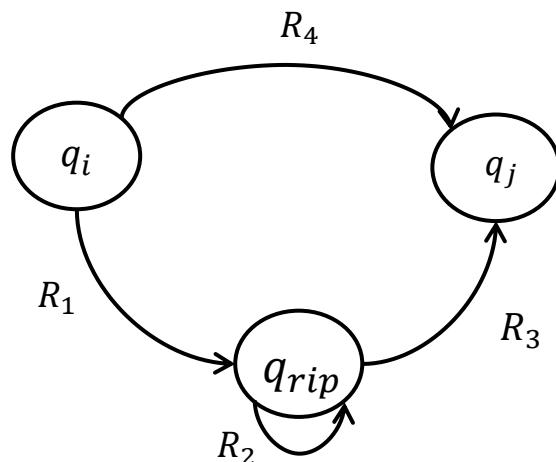
# DFA to Regular Expressions: GNFA

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:

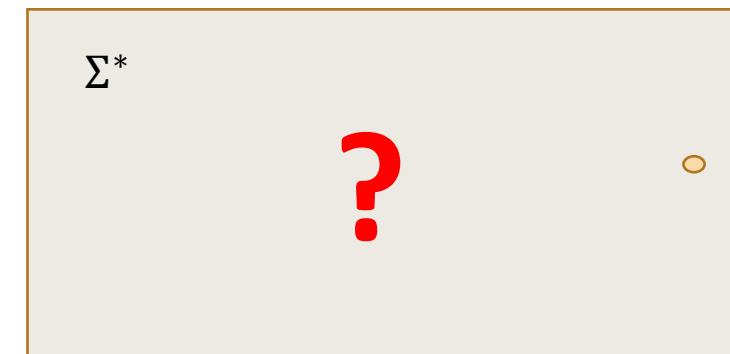
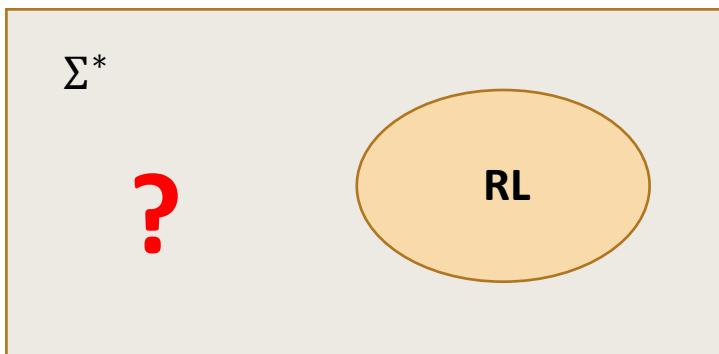
- $L$  is a regular language.
- There is a DFA  $D$  such that  $\mathcal{L}(D) = L$ .
- There is an NFA  $N$  such that  $\mathcal{L}(N) = L$ .
- There is a regular expression  $R$  such that  $\mathcal{L}(R) = L$ .
- Not all languages are regular.



# 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  $L$ .

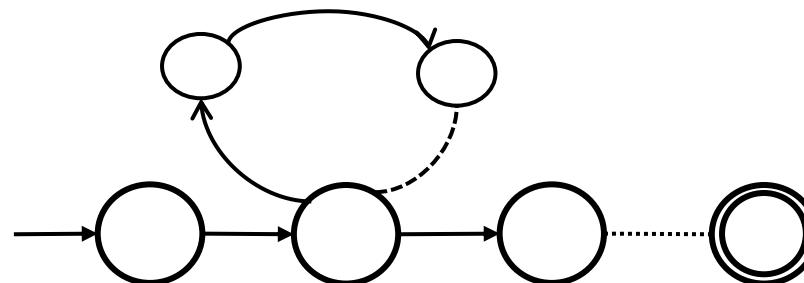
**Karl:** How many states are there?

**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  $t = 3$ ).

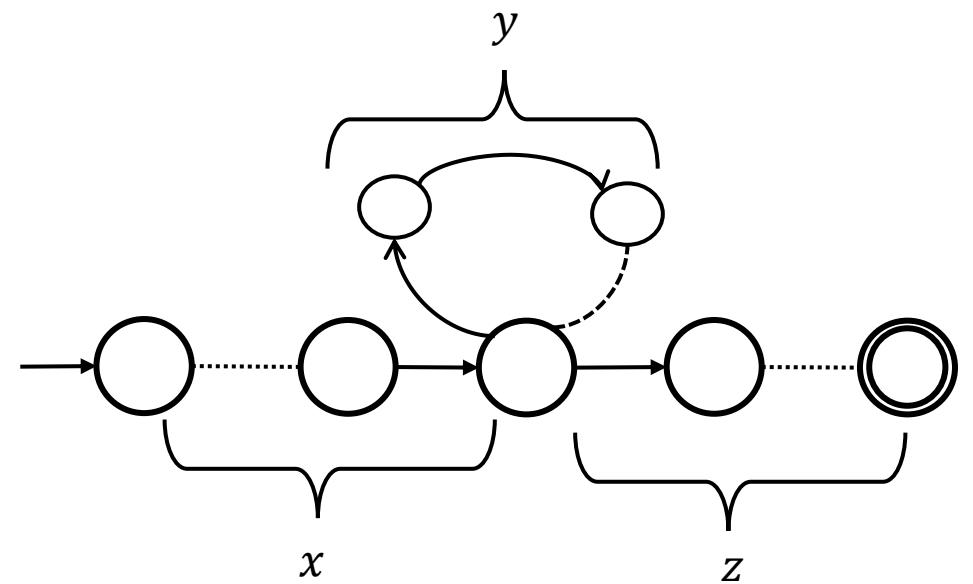
**Karl:** If your DFA accepts  $0^n 1^n$ , it must also accept  $0^{n+t} 1^n$ . This is because, if we take the loop one extra time, we read  $t$  more 0's.



**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  $\in L$ .

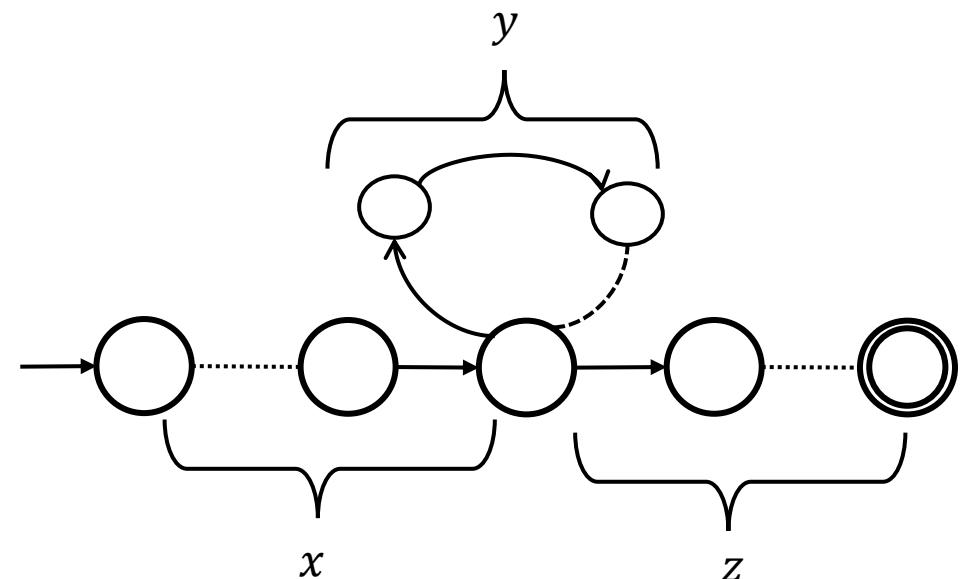


# Pumping Lemma

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

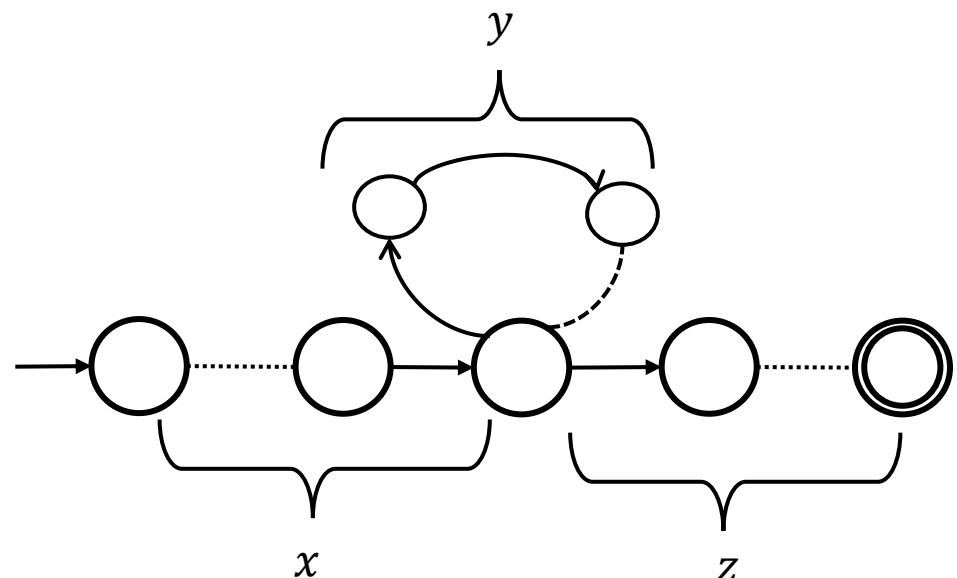
1.  $|xy| \leq p$ .
2.  $|y| \geq 1$
3.  $\forall i \geq 0, xy^i z \in L$ .

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

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

$\equiv$

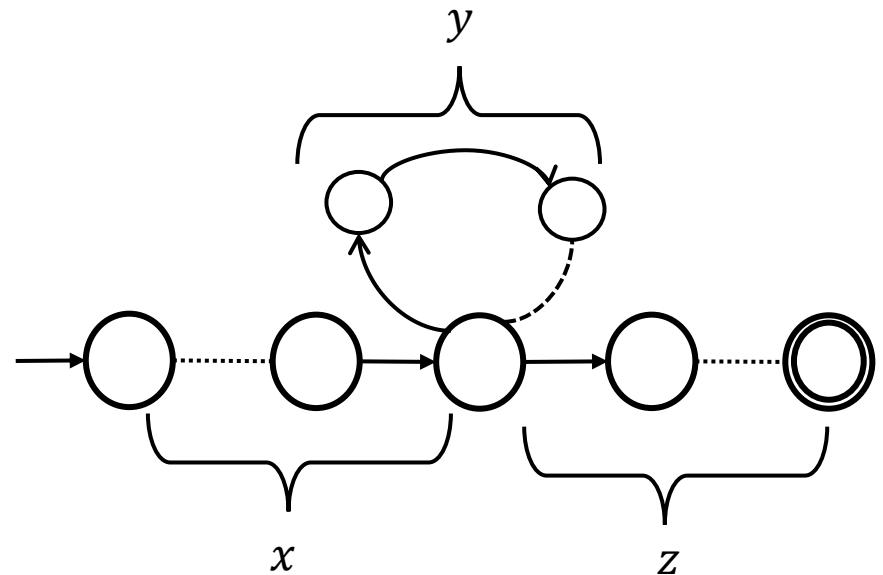
If pumping property is NOT satisfied, then  $L$  is NOT regular.



# 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.



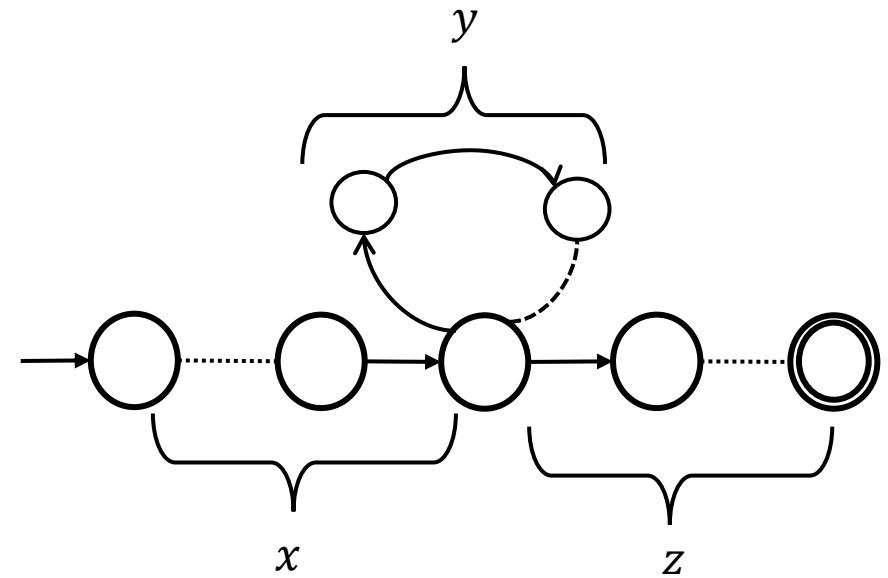
# Pumping Lemma

**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.

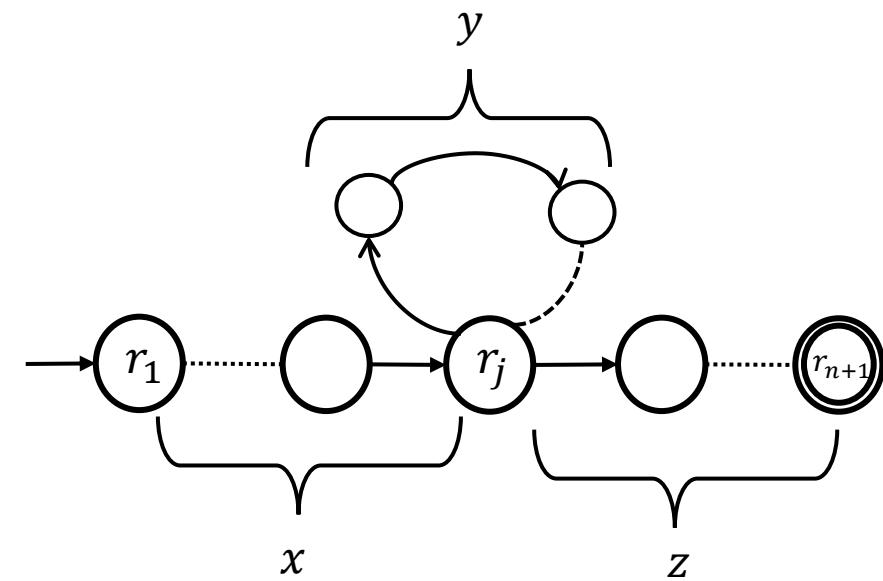
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

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.

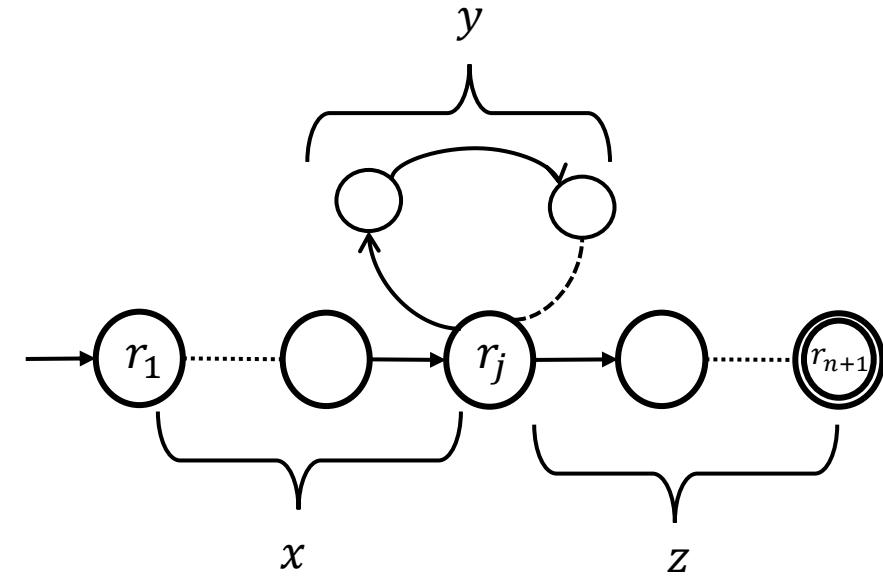
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

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.

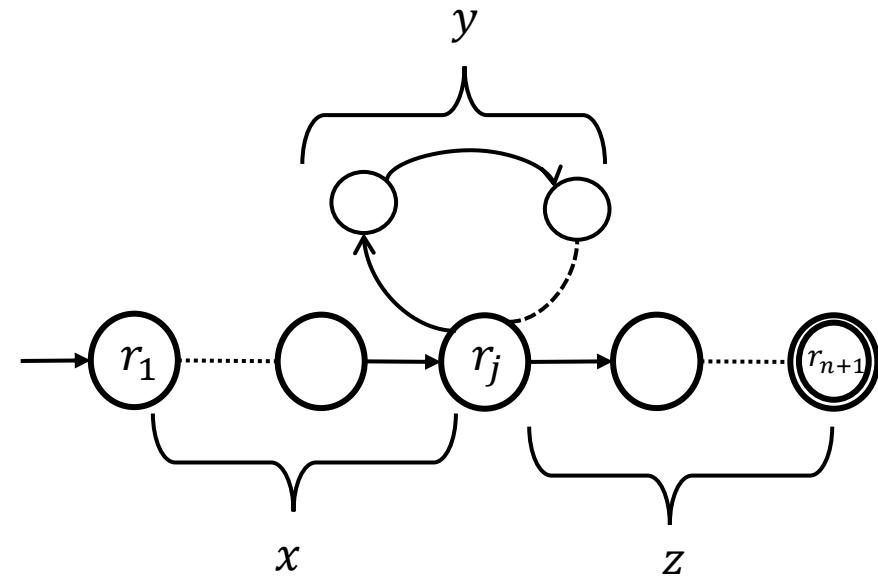
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

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.

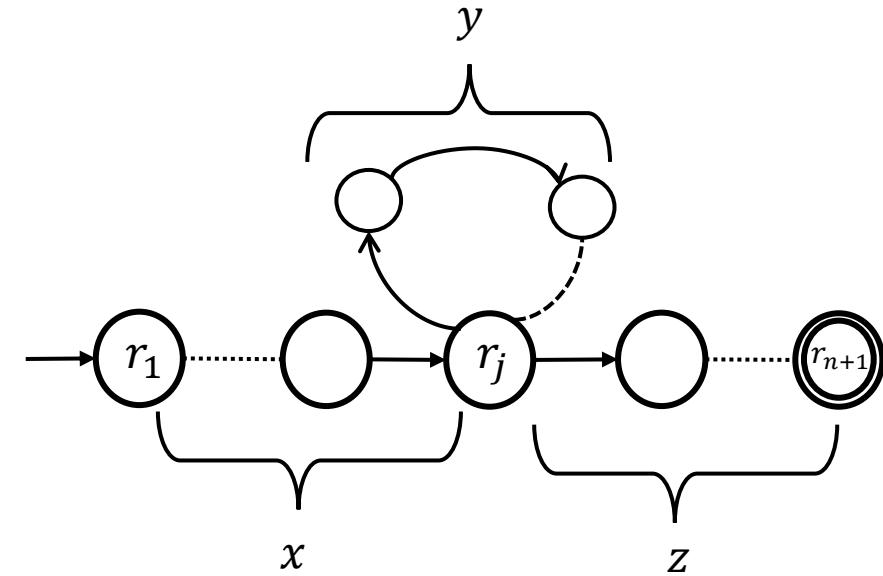
This is because of the **pigeonhole principle**: any such run would encounter  $p + 1$  states, but there are  $p$  distinct states in the DFA.

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 *recognize* whether a string belongs to a language
- 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 ( $\Sigma^*$ )

**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

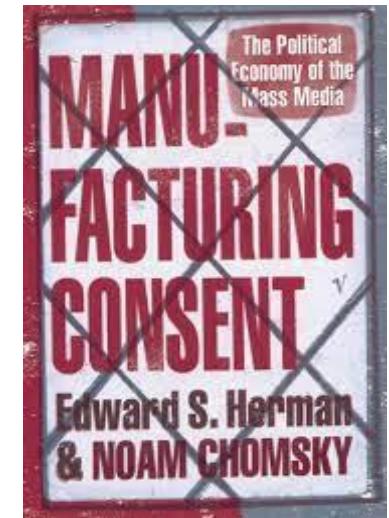
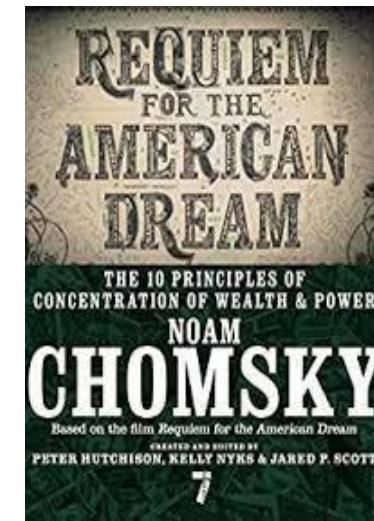
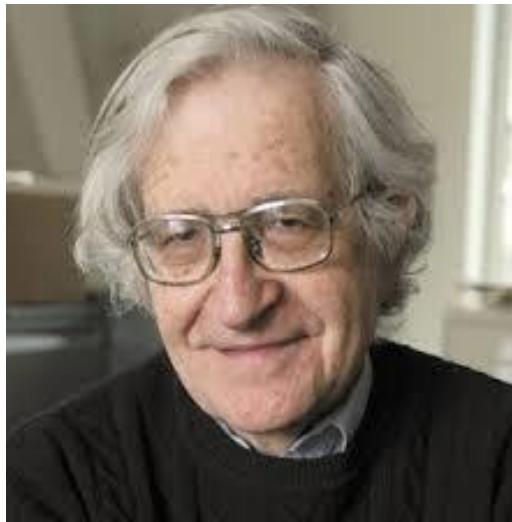
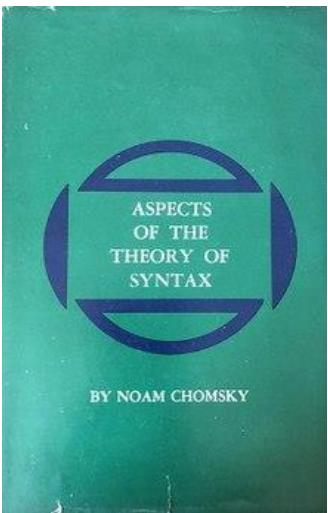
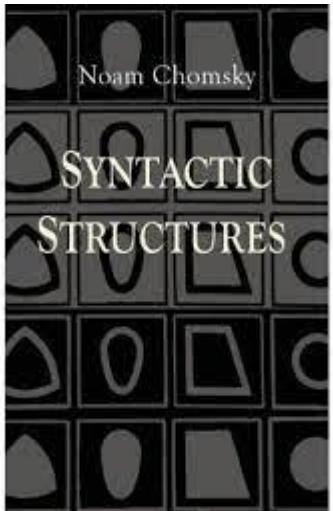
- **Grammars** provide a way to generate strings belonging to a language. The set of all strings generated by the grammar is the *language* of the grammar.
- **Grammars generate languages:** Grammars consist of a set of **rules** that allow you to construct strings of the language.
- For some classes 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 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  $\{w \in \Sigma^* | S \xrightarrow{*} w\}$

# 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  $G$

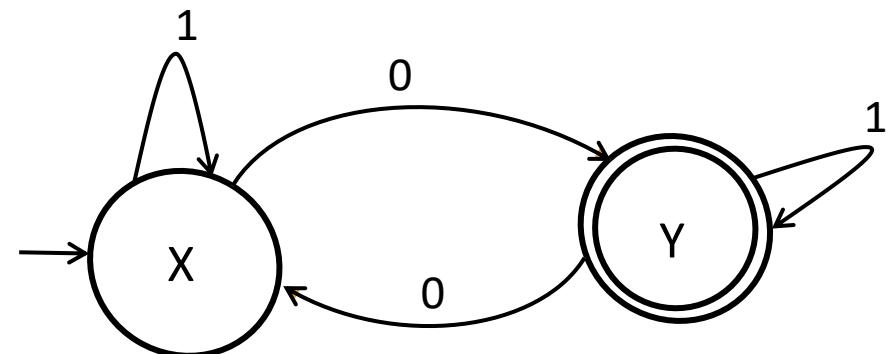
$$X \rightarrow 1X$$

$$X \rightarrow 0Y$$

$$Y \rightarrow 1Y$$

$$Y \rightarrow 0X$$

$Y \rightarrow \epsilon$  (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  $G$

$$X \rightarrow 1X$$

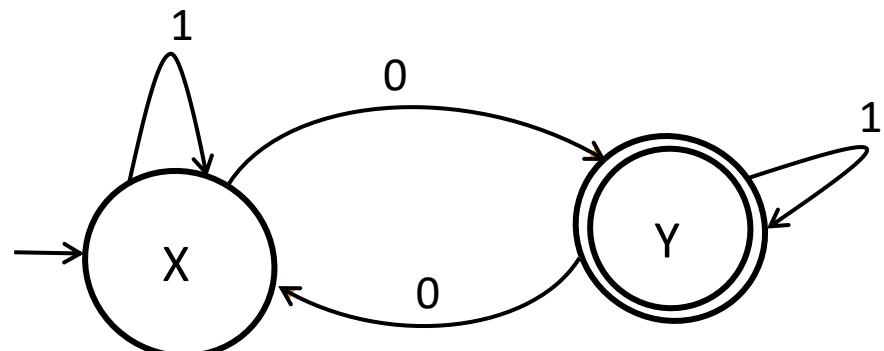
$$X \rightarrow 0Y$$

$$Y \rightarrow 1Y$$

$$Y \rightarrow 0X$$

$Y \rightarrow \epsilon$  (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

$$Var \rightarrow Ter\ Var$$

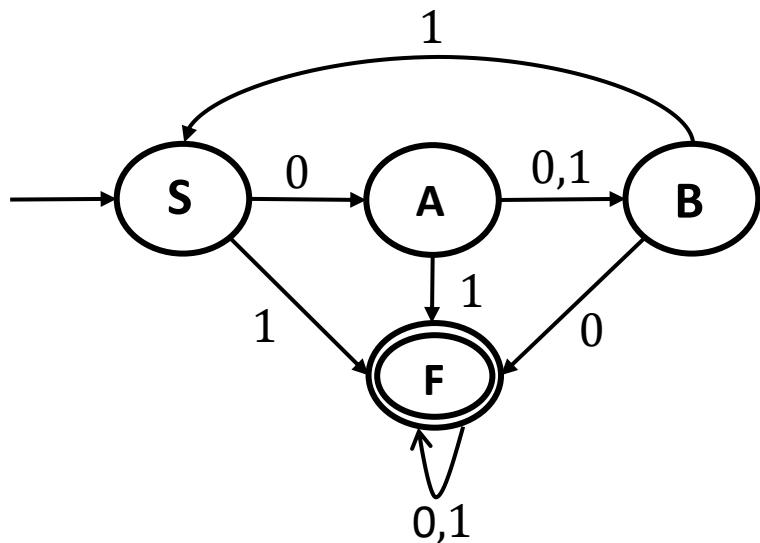
$$Var \rightarrow Ter$$

$$Var \rightarrow \epsilon$$

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 – See Assignment 1)

# 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**.

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

**Thank You!**

# CS 302.1 - Automata Theory

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad



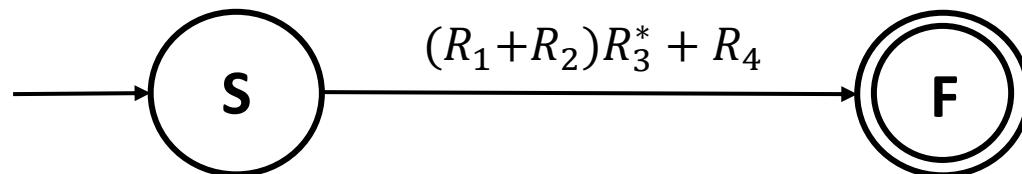
INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

H Y D E R A B A D

# Quick Recap

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)** 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.  $\forall i \geq 0, xy^i z \in L$ .

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

$\equiv$

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

Examples of languages that are NOT regular:

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

# Quick Recap

**(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  $w$**  ( $S \xrightarrow{*} w$ ).
- 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**

# Context free 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 ]

**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**.

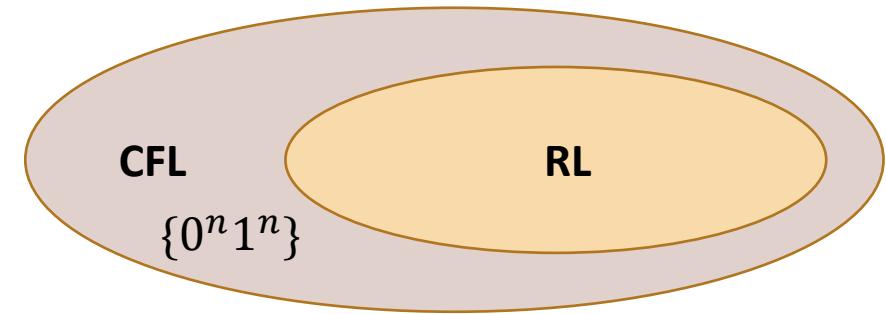
Any language generated by a context-free grammars is called a ***context-free language***.

- 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  $G$ :

$$S \rightarrow SS \rightarrow 00S1S1 \rightarrow 00S10S11 \rightarrow 001011$$

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

# 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  $G$ :

$$S \rightarrow SS \rightarrow 00S1S1 \rightarrow 00S10S11 \rightarrow 001011$$

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

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.

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

# Parse trees for CFG

Consider the Grammar  $G$  with the following rules:

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

One derivation:

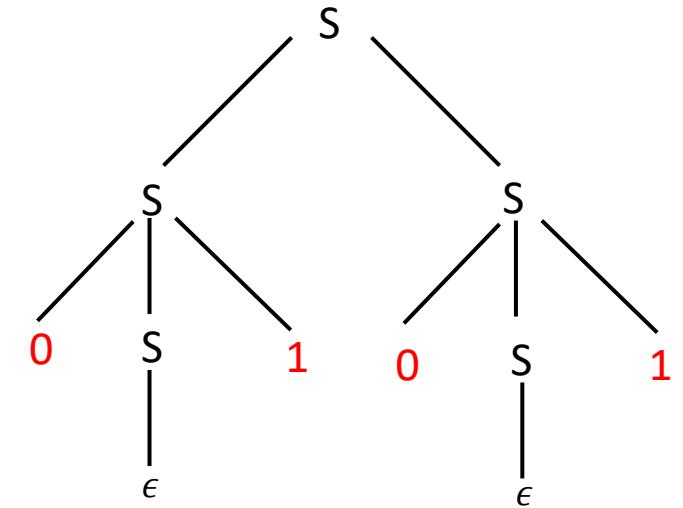
$$S \rightarrow SS \rightarrow 0S1S \rightarrow 0S10S1 \rightarrow 0101$$

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

**Parsing** is a useful technique for compilers.

**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  $w \in L(G)$



# Parse trees for CFG

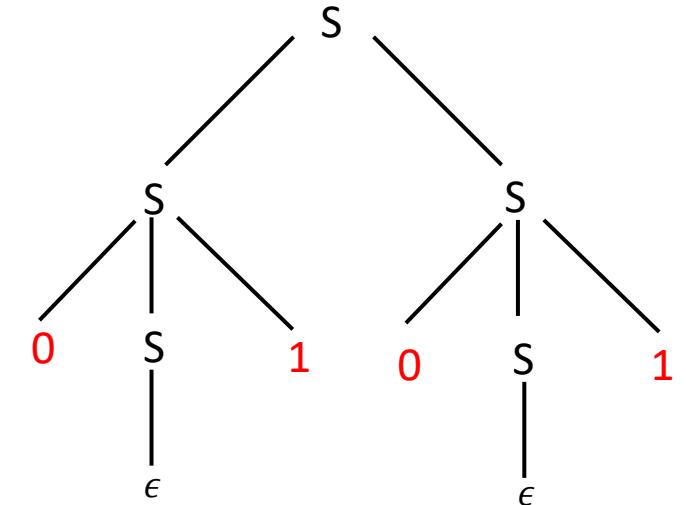
Consider the Grammar  $G$  with the following rules:

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

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

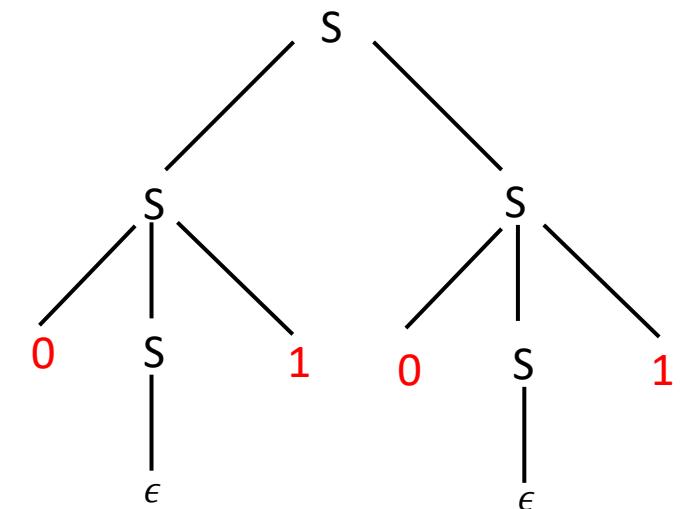
Consider the Grammar  $G$  with the following rules:

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

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

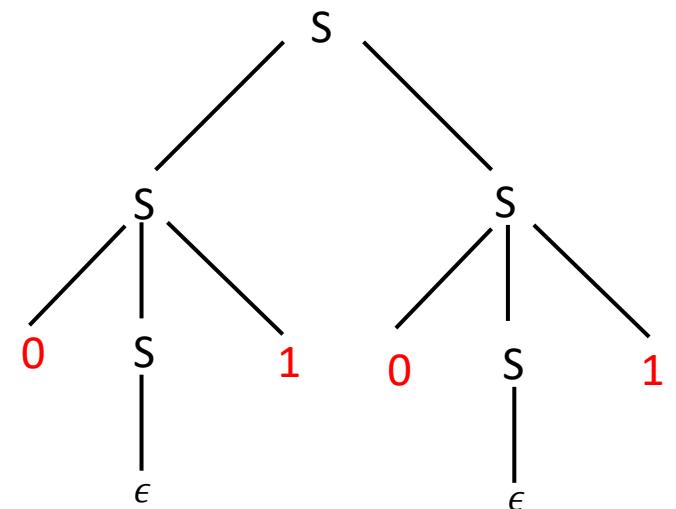
Consider the Grammar  $G$  with the following rules:

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

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

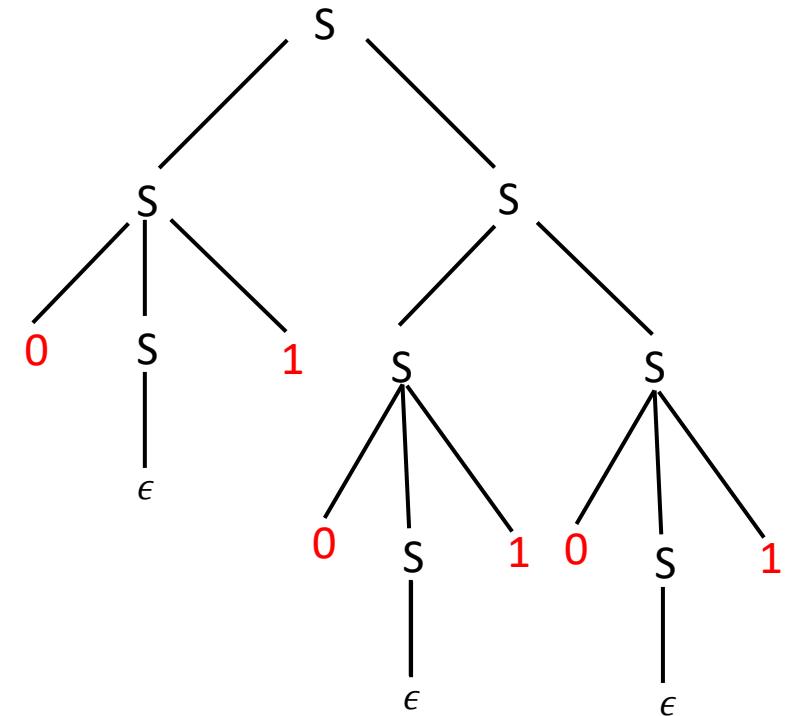
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

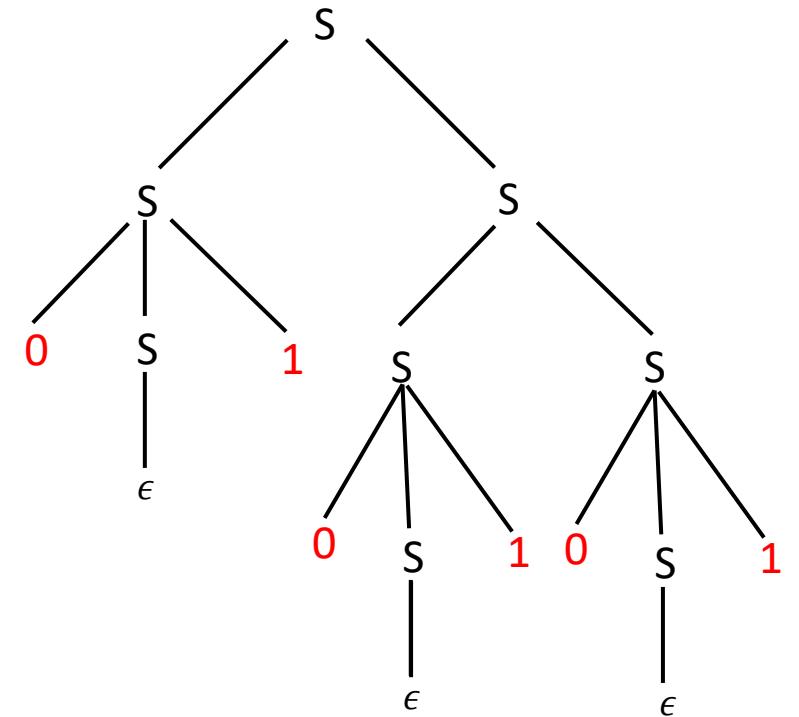
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

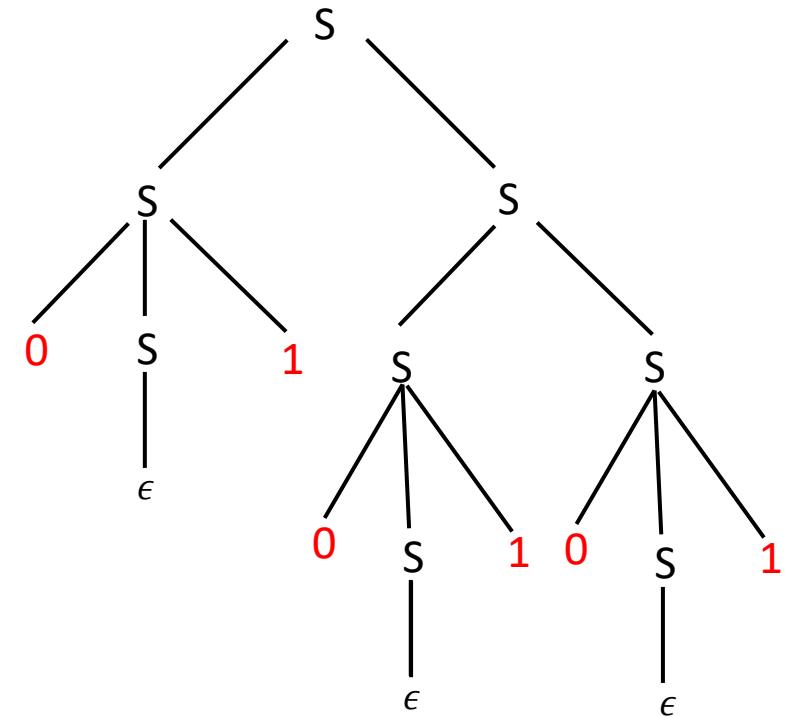
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

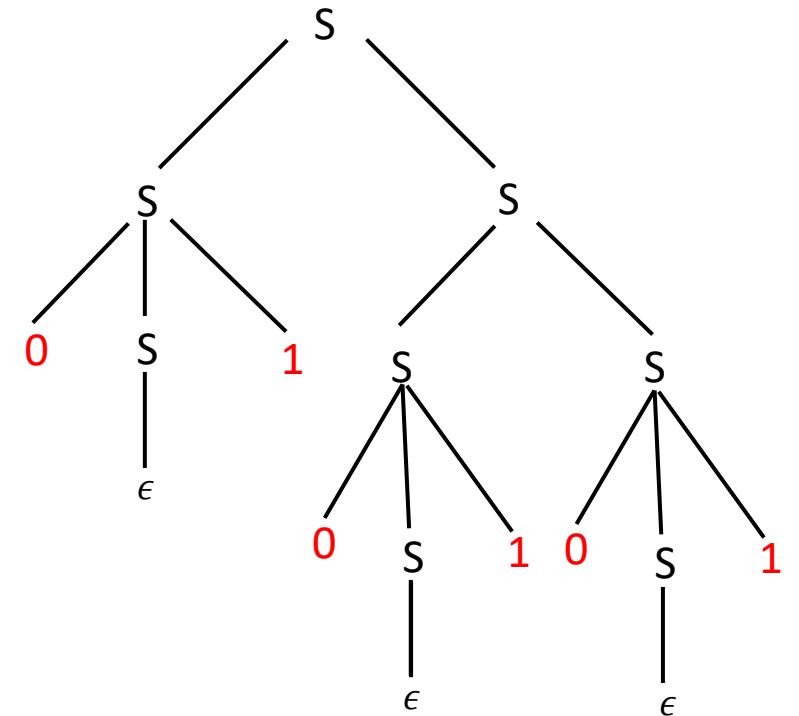
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

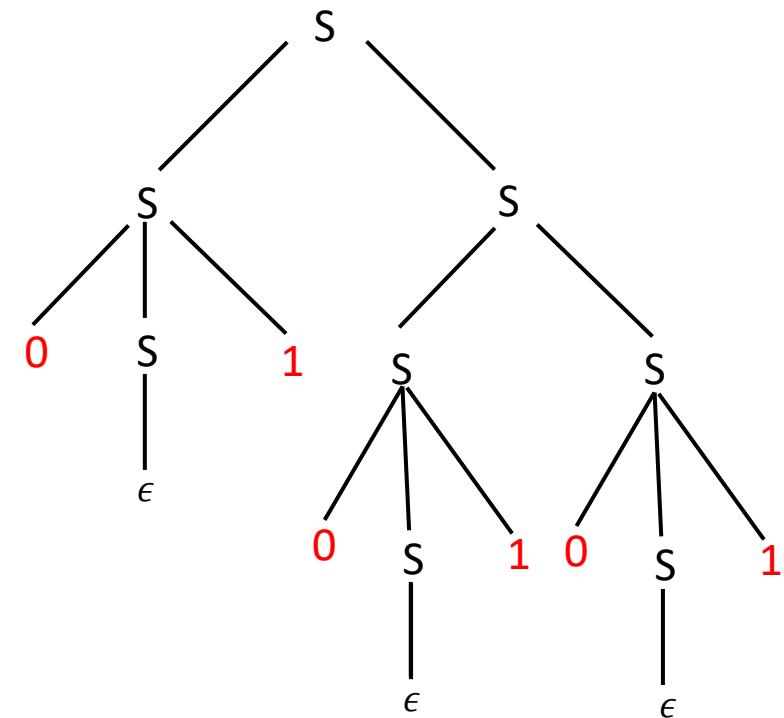
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

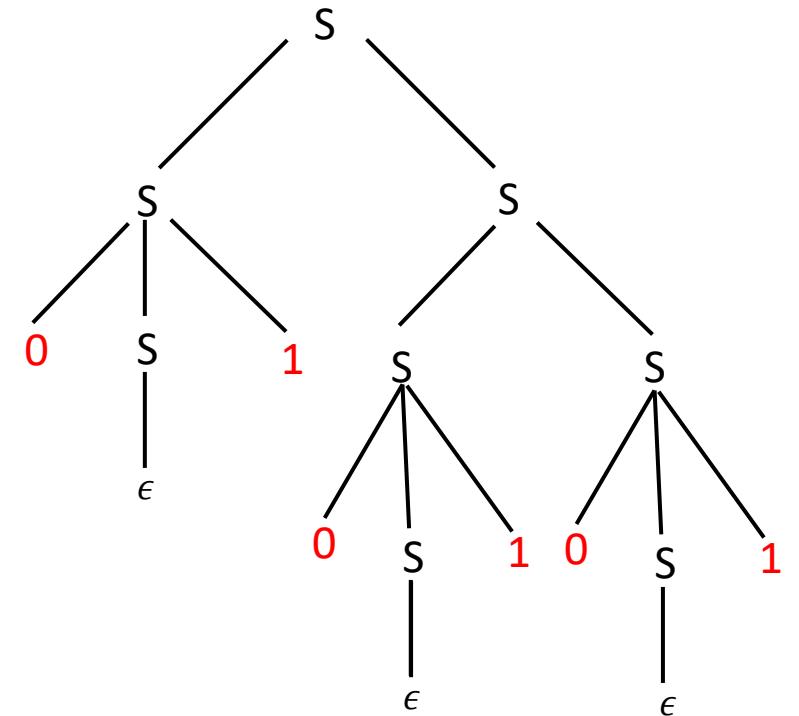
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

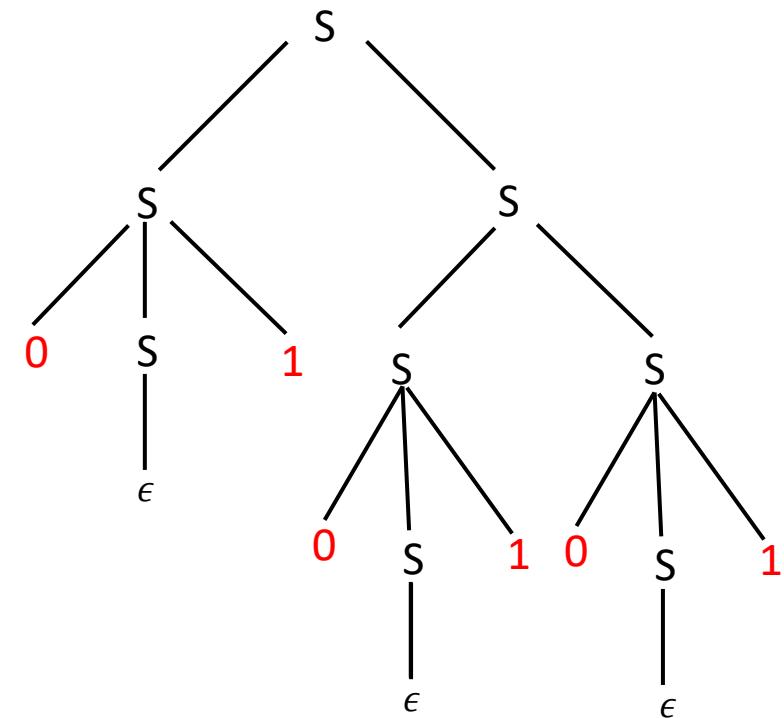
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

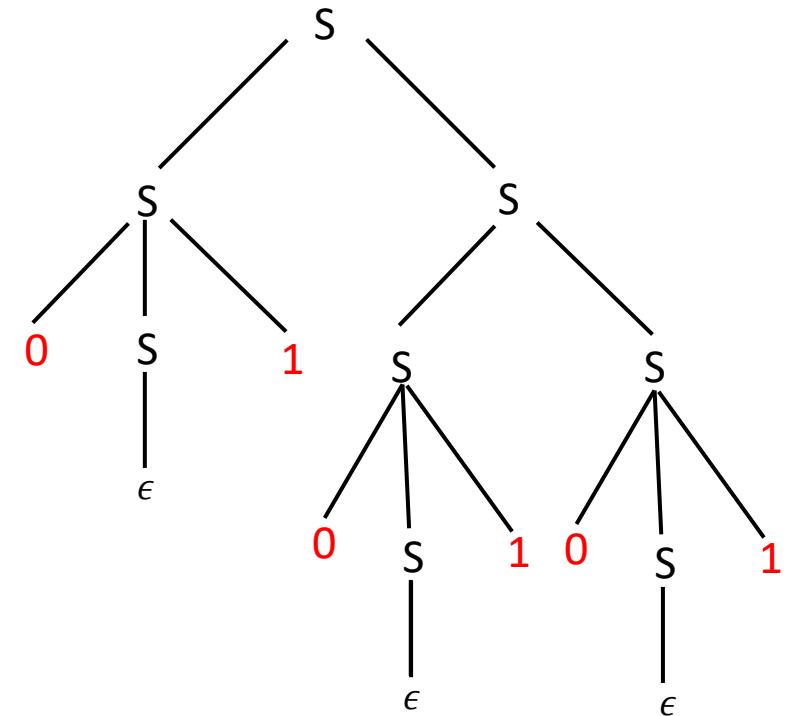
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

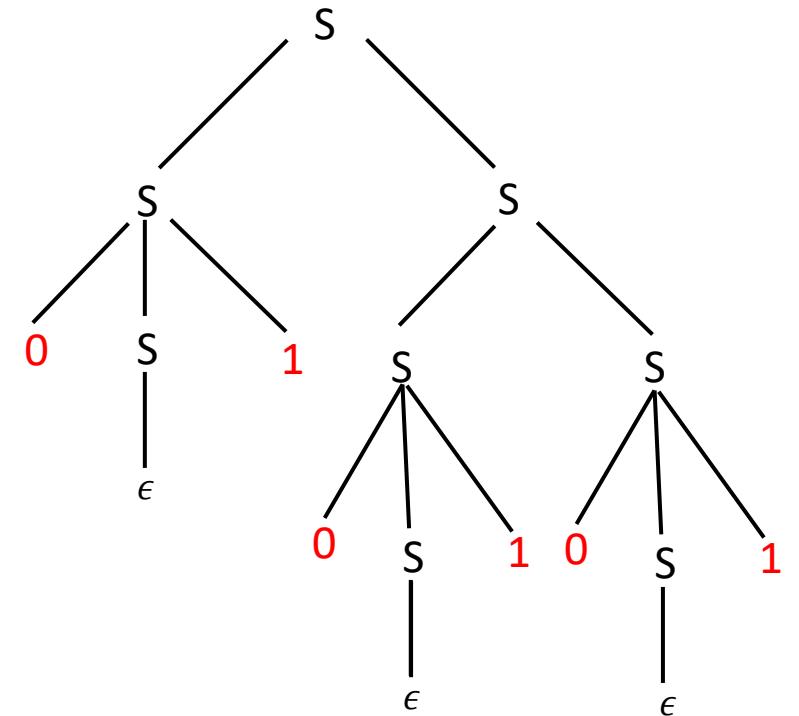
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

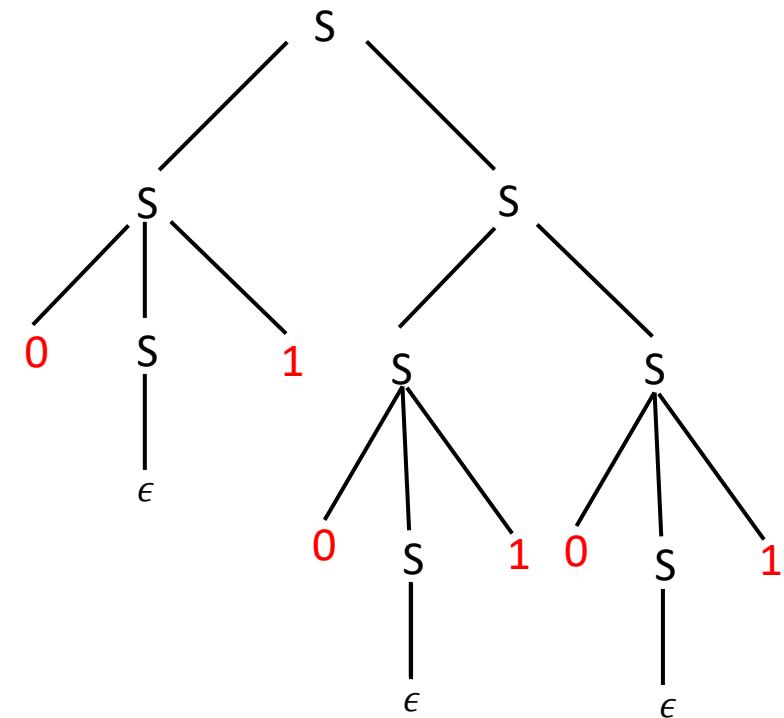
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

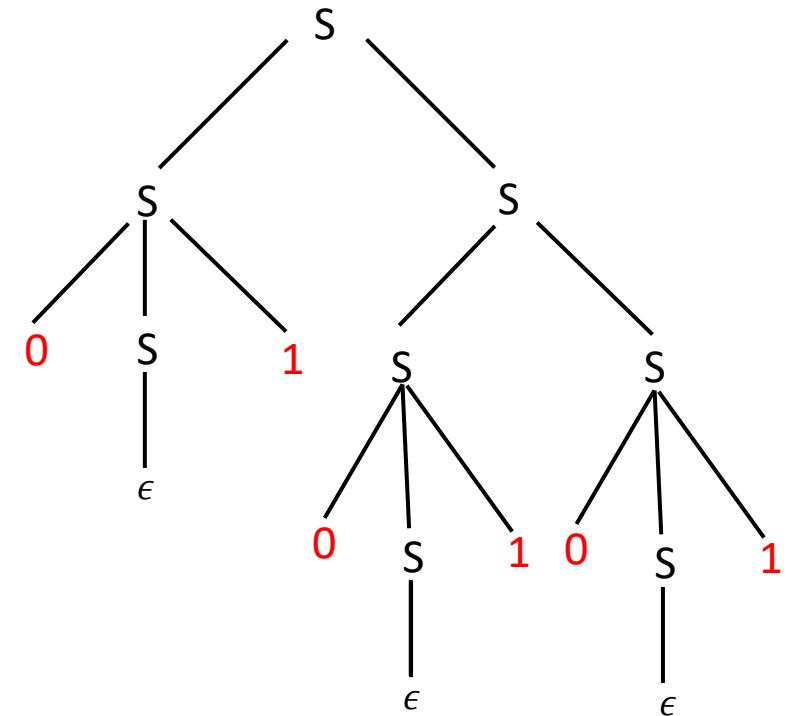
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

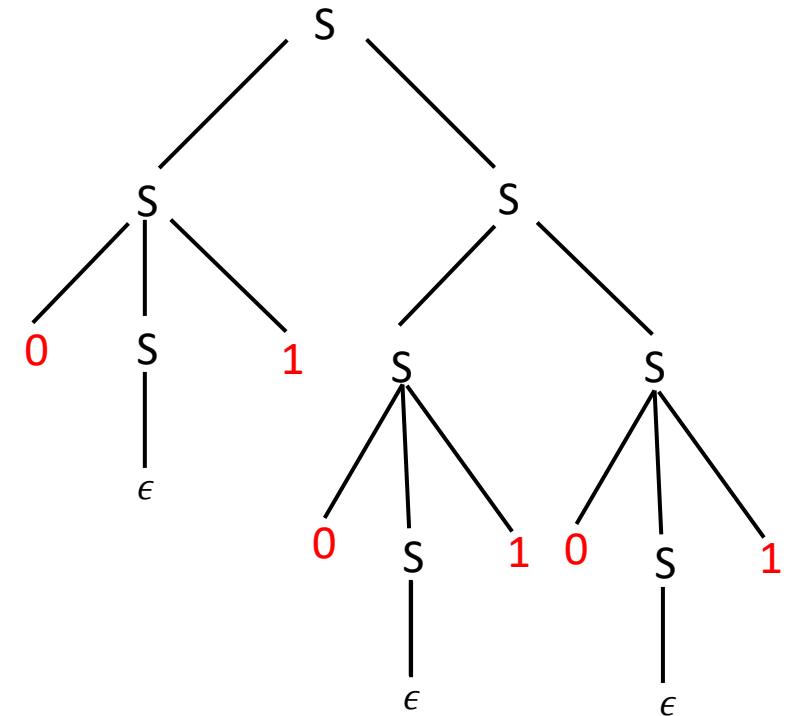
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

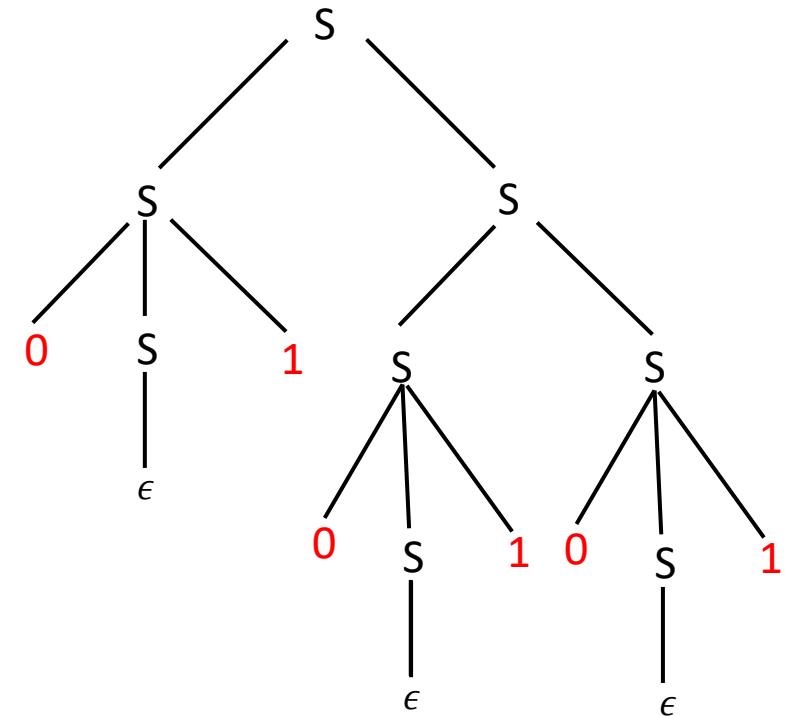
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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

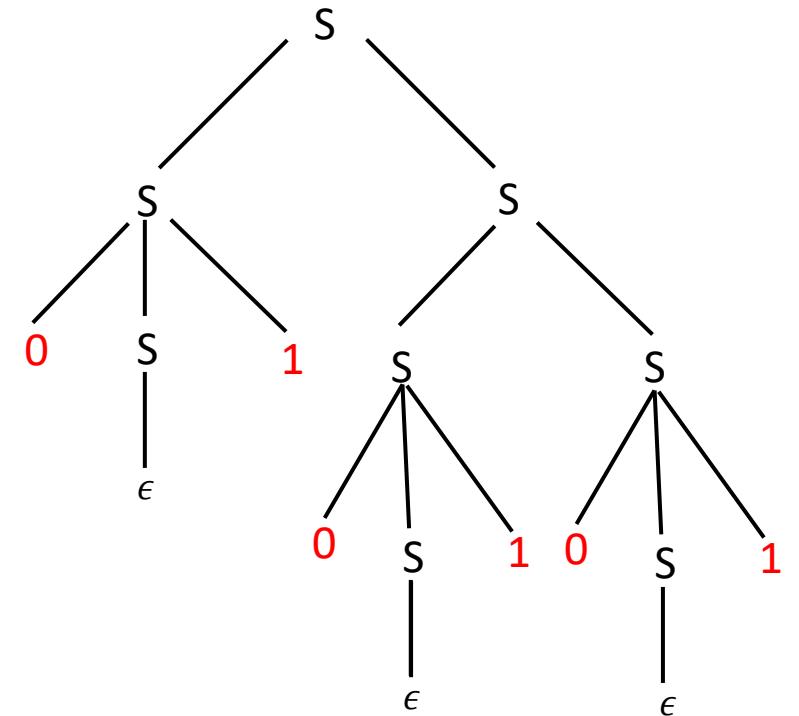
**Ambiguous grammars:** A CFG  $G$  is said to be **ambiguous** if there exists  $\omega \in L(G)$ , such that there are **two or more leftmost derivations for  $\omega$**  (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for  $\omega$** .

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:  $S \rightarrow SS \rightarrow 0S1S \rightarrow 01S \rightarrow 01SS \rightarrow 010S1S \rightarrow 0101S \rightarrow 01010S1 \rightarrow 010101$

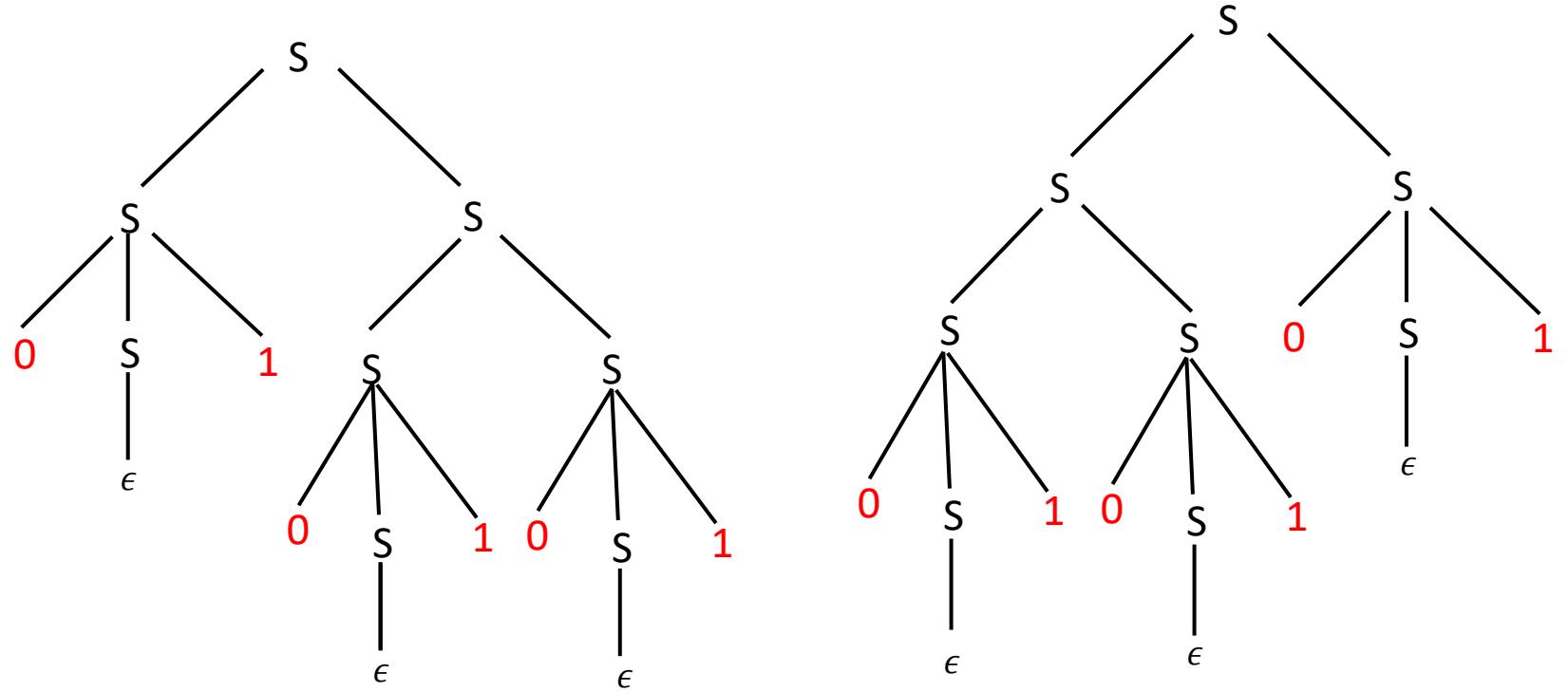
# Parse trees for CFG

**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**.

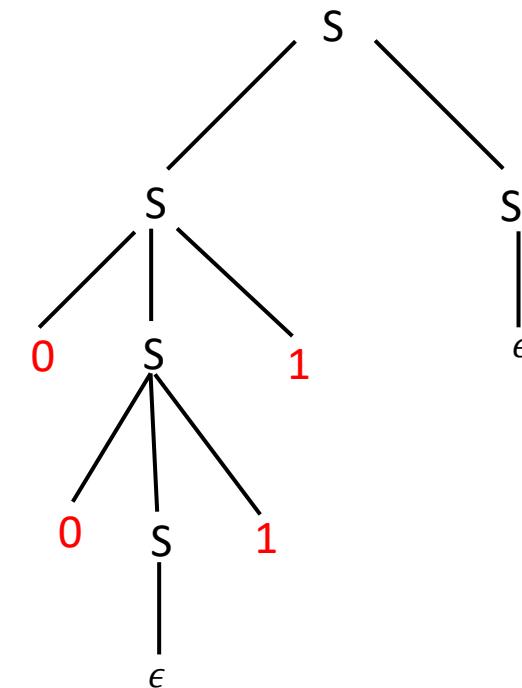
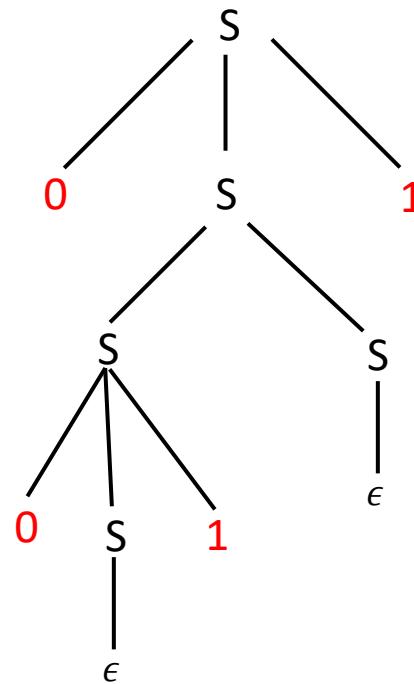
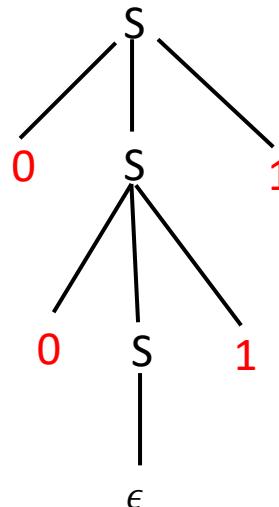


Leftmost Derivation:  $S \rightarrow SS \rightarrow SSS \rightarrow 0S1SS \rightarrow 01SS \rightarrow 010S1S \rightarrow 0101S \rightarrow 01010S1 \rightarrow 010101$

# 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.**

Consider the grammar:

$$S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid \dots \mid 9$$

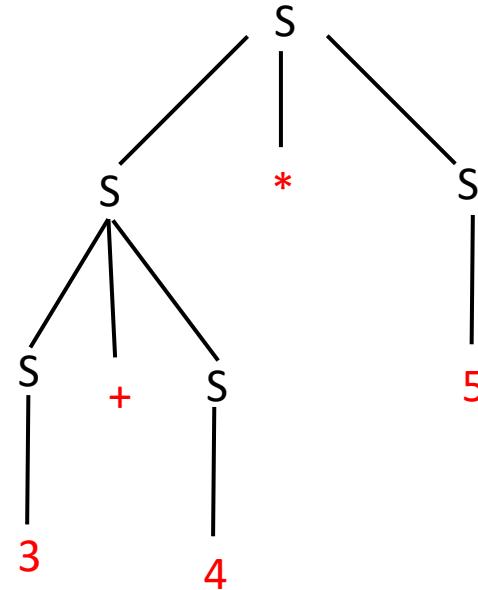
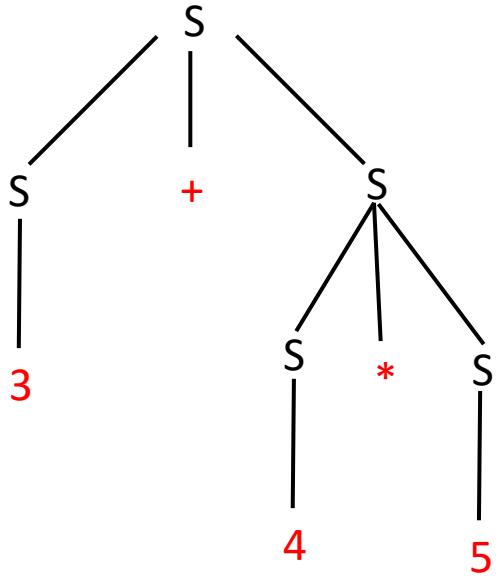
and the derivation of the string **3 + 4 \* 5**

# 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  $3 + 4 * 5$



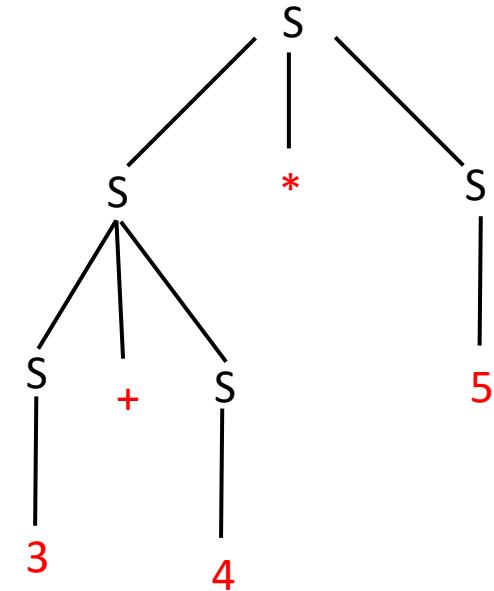
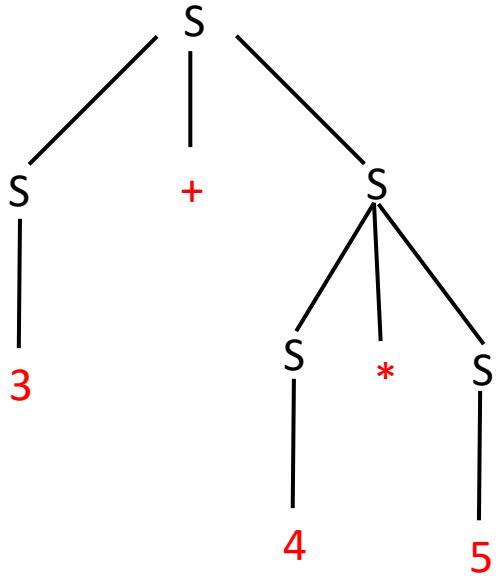
- The grammar contains no information on the precedence relations of the various arithmetic operations.
- The grammar may group  $+$  before  $*$

# 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  $3 + 4 * 5$



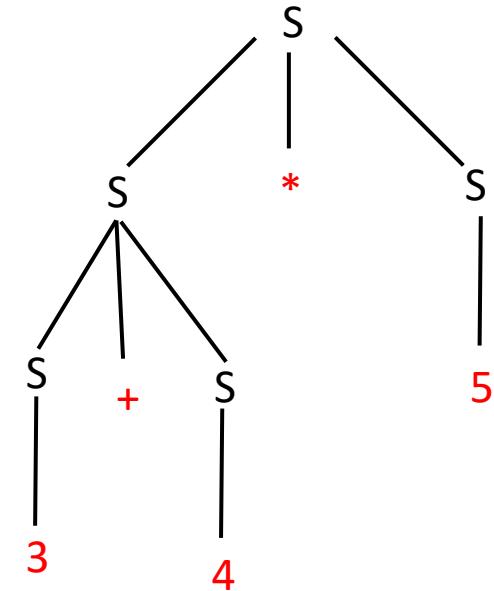
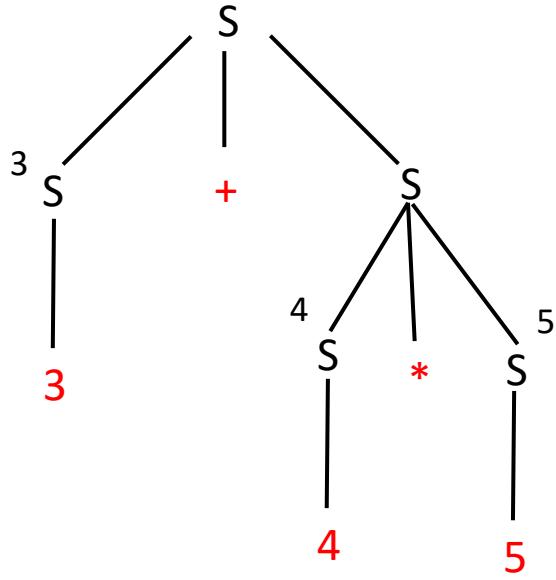
- What will be the result obtained from each of these *parsings*?

# 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  $3 + 4 * 5$



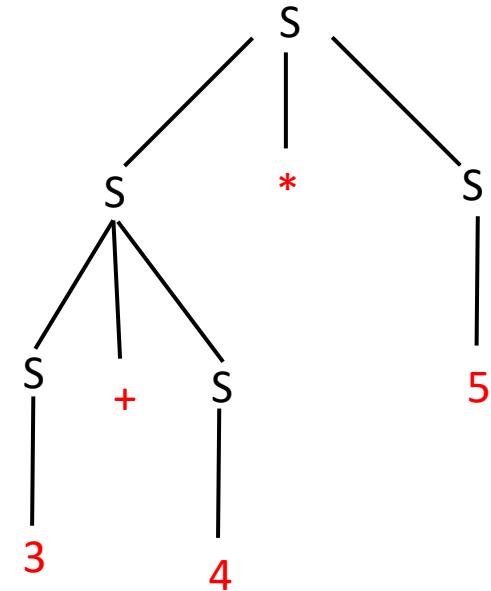
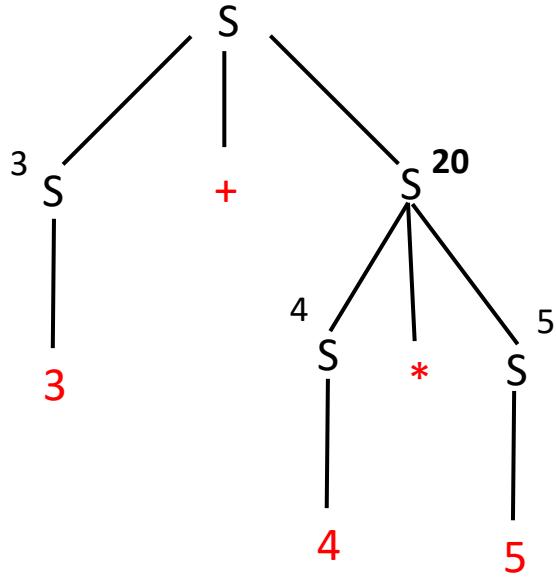
- If the compiler compiles the left parse tree

# 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  $3 + 4 * 5$



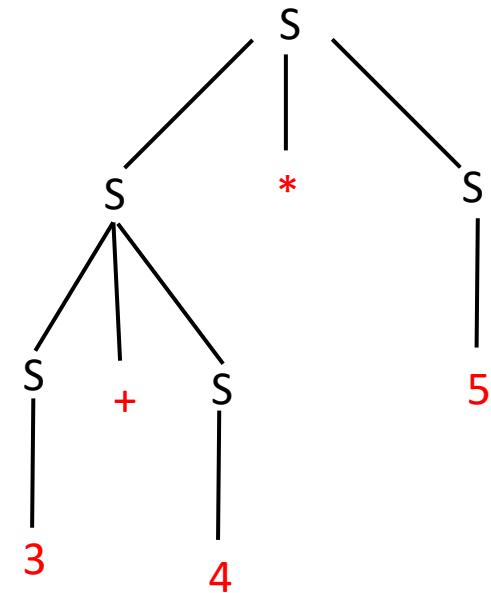
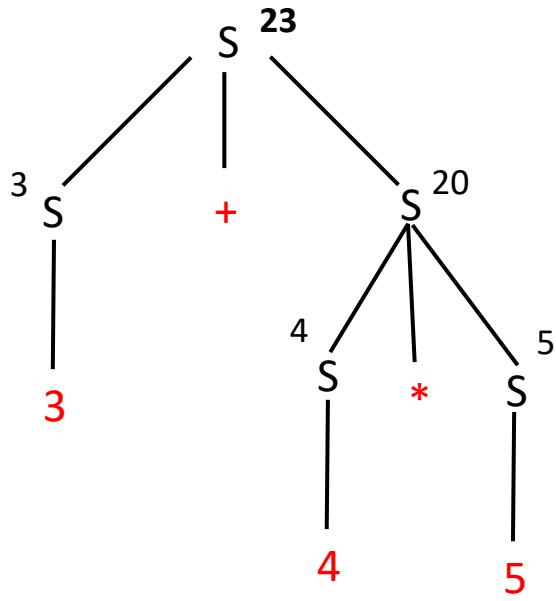
- If the compiler compiles the left parse tree

# 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  $3 + 4 * 5$



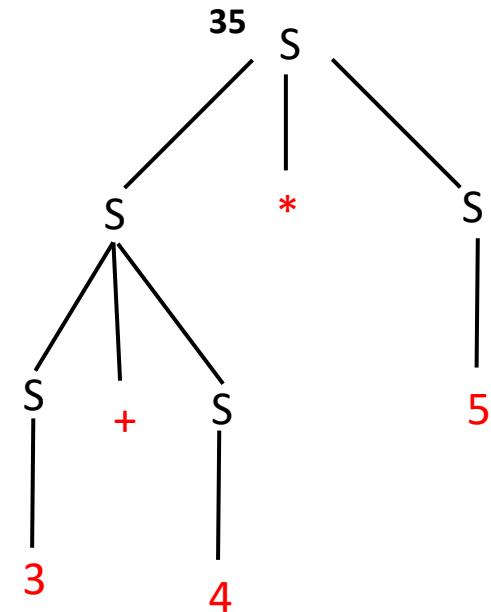
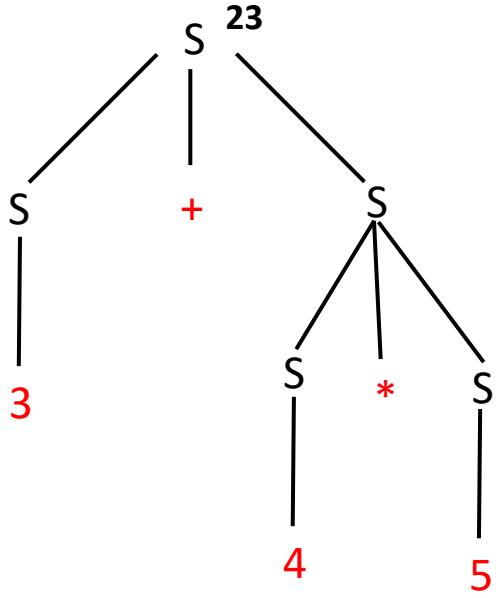
- If the compiler compiles the left parse tree. Outcome = **23**

# 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  $3 + 4 * 5$



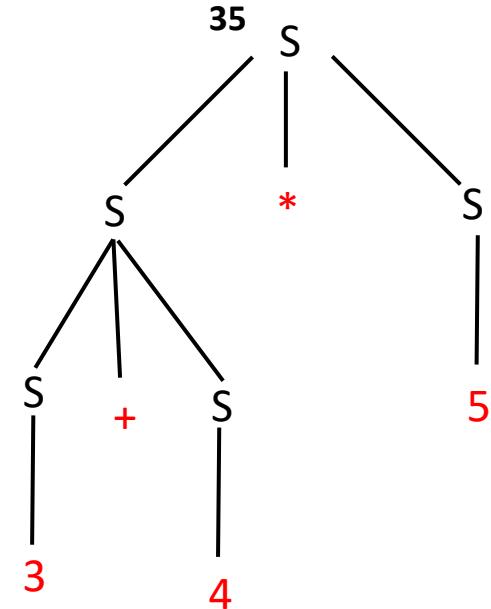
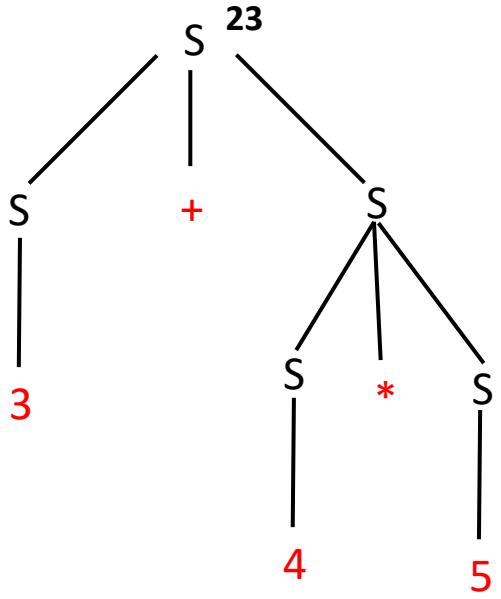
- If the compiler compiles the **right** parse tree. Outcome = **35**

# 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?

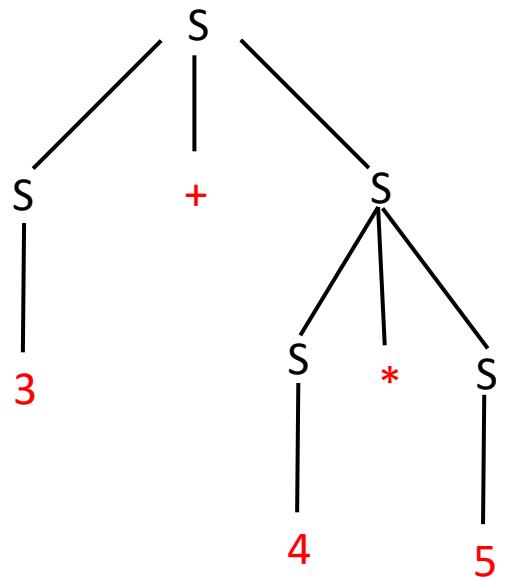
# Ambiguity

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)

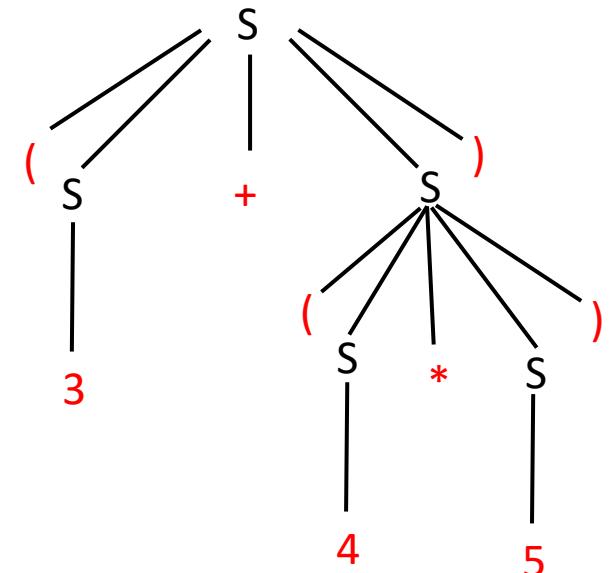
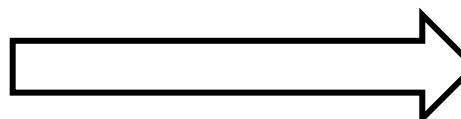
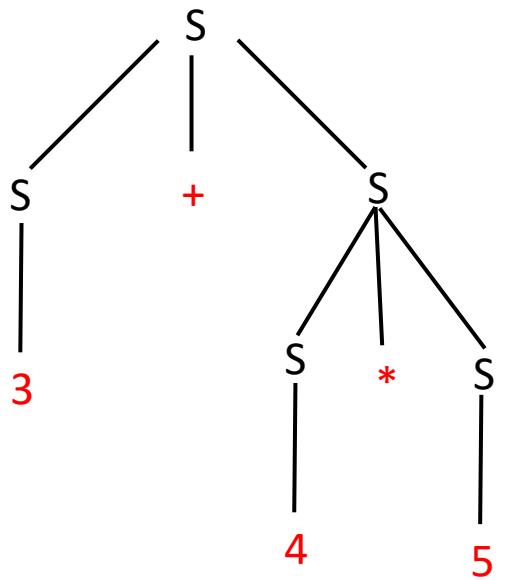
# Ambiguity

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

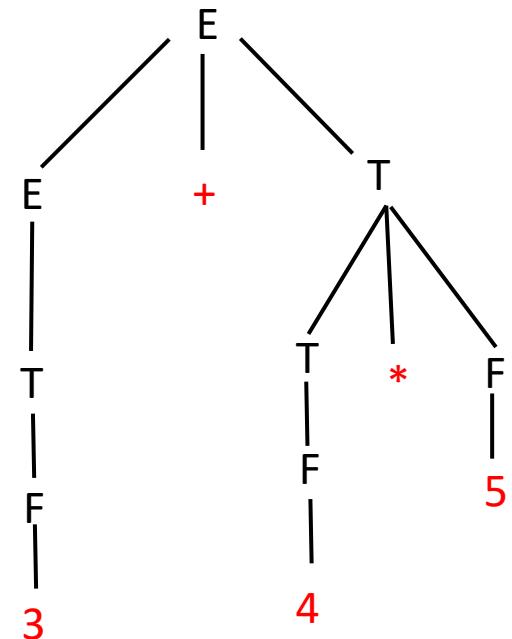
How can we get rid of this ambiguity? Change the production rules

- 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

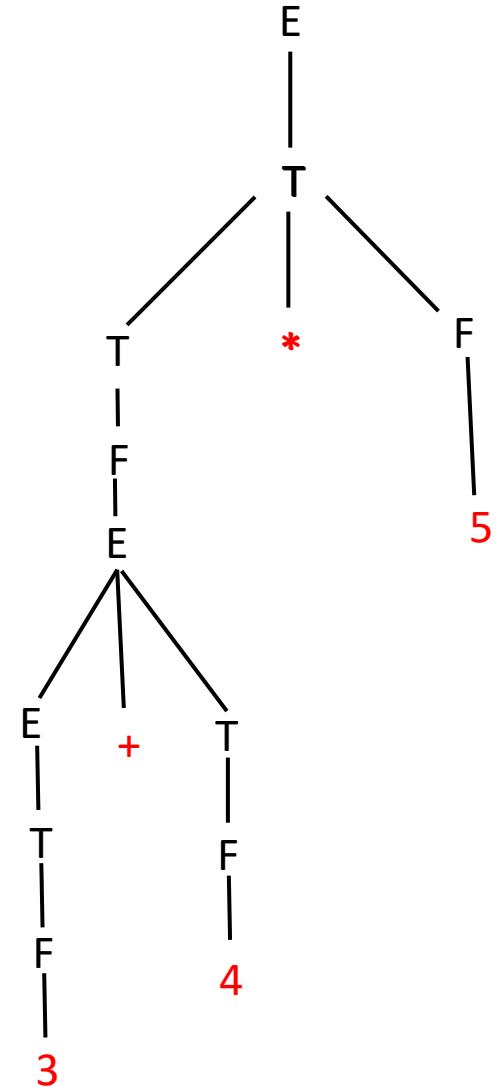
How can we get rid of this ambiguity? Change the production rules

- 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.

**Thank You!**

# CS 302.1 - Automata Theory

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad



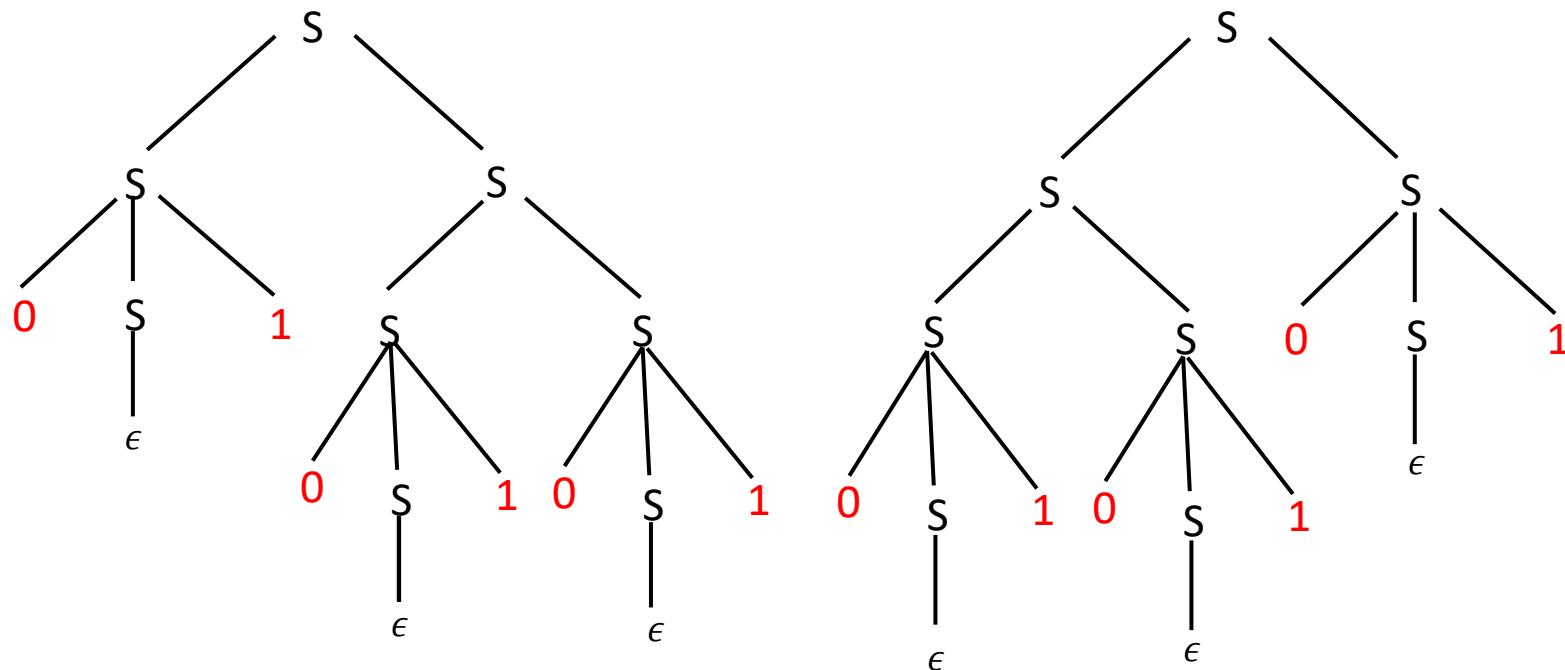
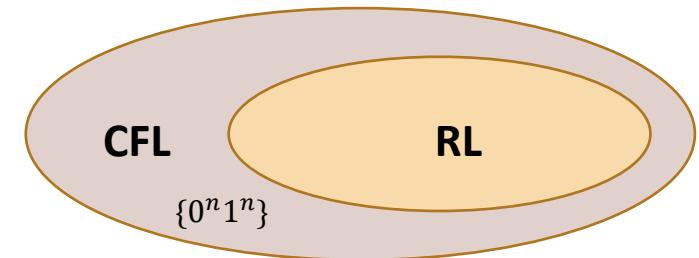
INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

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

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

# 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  $G$  as 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  $G$  as 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:

$$\begin{array}{ll} A \rightarrow BC & [B, C \text{ are not start variables}] \\ A \rightarrow a & [a \text{ is a terminal}] \\ S \rightarrow \epsilon & [S \text{ is the Start Variable}] \end{array}$$

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 $A \rightarrow S$
$S \rightarrow 11 B 1$	$S \rightarrow 11 B 1$	$S \rightarrow 11 B 1$	$S \rightarrow 11 B 1$	$S \rightarrow 11 \mathbf{0} 1$	$S \rightarrow 11 \mathbf{0} 1$
$A \rightarrow B 1$	$A \rightarrow 1 S \mathbf{0}$	$A \rightarrow 1 S 0$	$A \rightarrow 1 S 0$	$A \rightarrow 1 S 0$	$A \rightarrow 1 \mathbf{11} 0$
$B \rightarrow S 0$	$B \rightarrow S 0$	$B \rightarrow 0 \mathbf{11} 1 B$	$B \rightarrow 0 11 1$	$B \rightarrow 0 11 1$	$B \rightarrow 0 11 1$

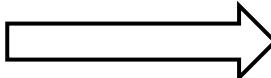
# Chomsky Normal Form

**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.

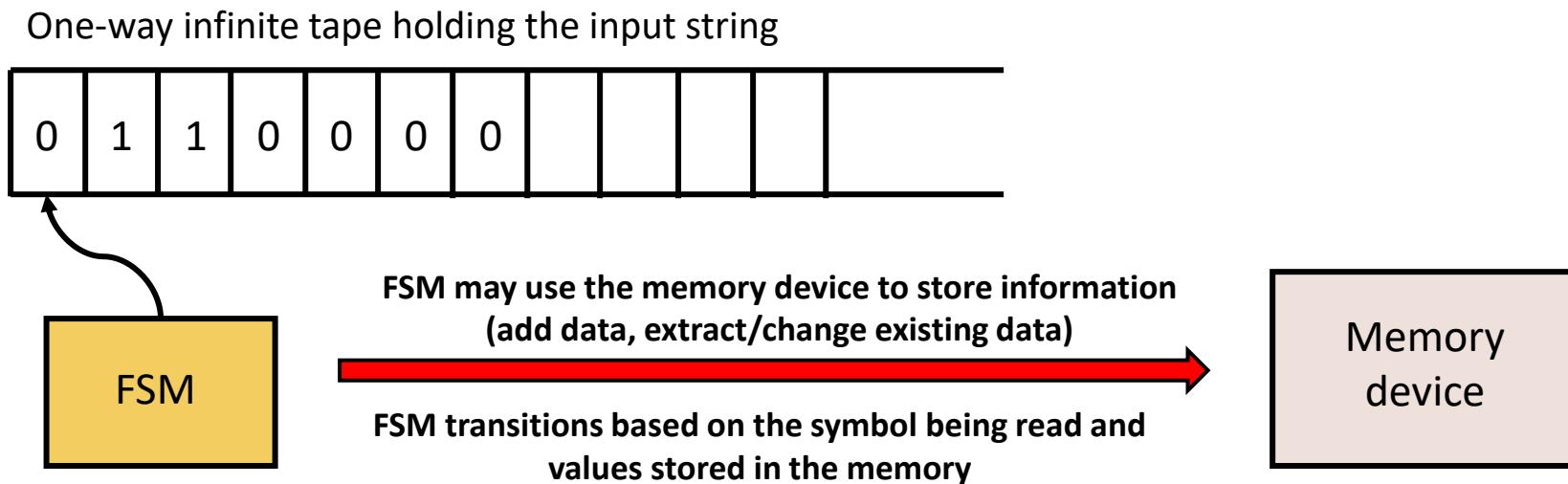
E.g.:  $\{0^n 1^n, n \in \mathbb{N}\}$

# Pushdown Automata

## Intuition to build an Automata for CFL

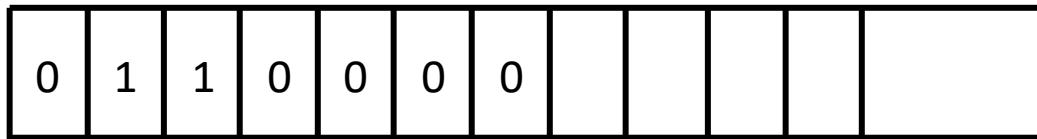
- **Automata for CFL = FSM + Memory device**
- **FSM may choose to ignore the memory device** completely in which case it behaves like a DFA/NFA.
- FSM makes use of the Memory device to recognize “non-Regular” CFLs.

E.g.:  $\{0^n 1^n, n \in \mathbb{N}\}$



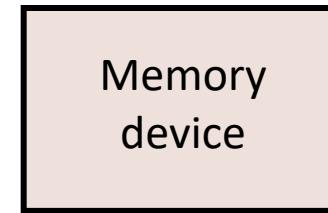
# Pushdown Automata

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

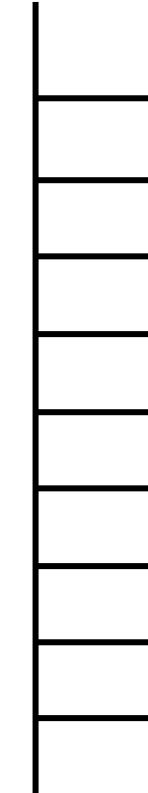
- Simple memory device with unbounded memory.

# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, the **top** of the STACK can be read.

Memory  
device



# Pushdown Automata

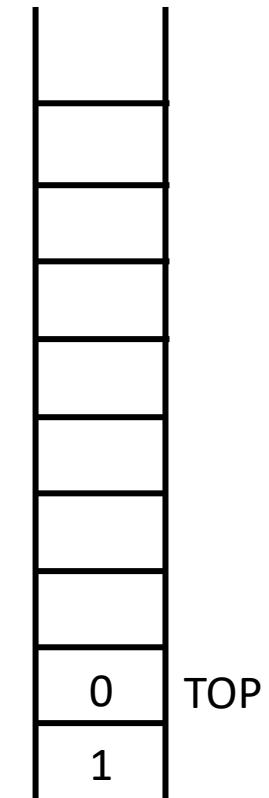
## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, the **top** of the STACK can be read.

### PUSH

- New symbols can be **pushed** in to the STACK.  
E.g: If TOP of STACK = 0, 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

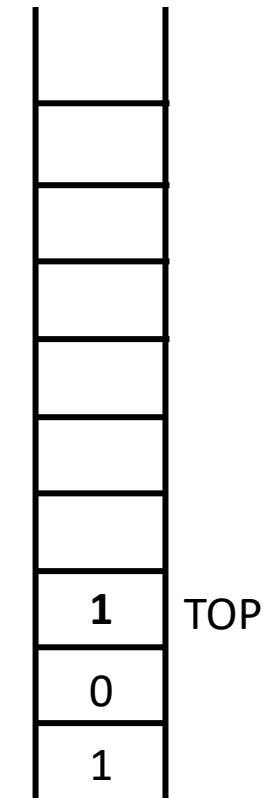
## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, the **top** of the STACK can be read.

## PUSH

- New symbols can be **pushed** in to the STACK.  
E.g: If TOP of STACK = 0, 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

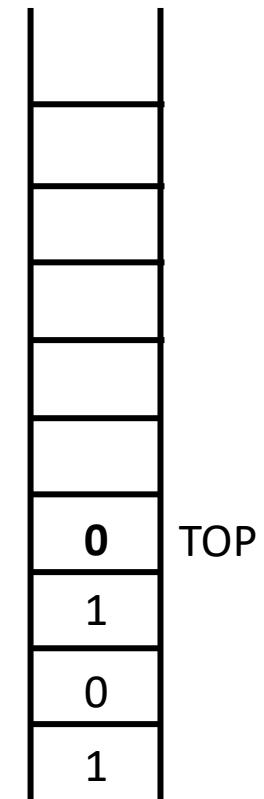
## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, the **top** of the STACK can be read.

### PUSH

- New symbols can be **pushed** in to the STACK.  
**E.g: PUSH 0 (irrespective of the value of Top)**
- 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

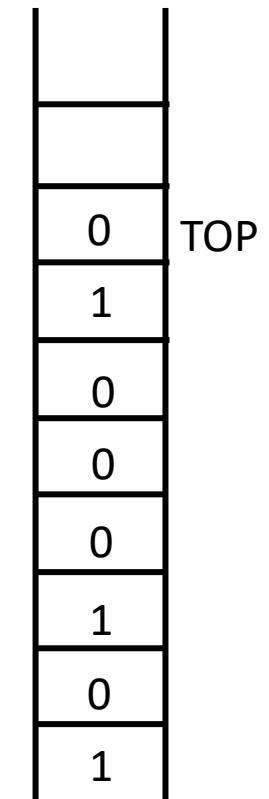
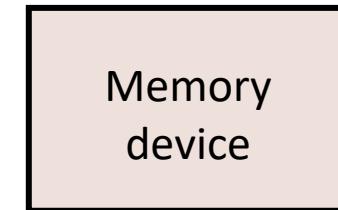
- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, the **top** of the STACK can be read.

## PUSH

- New symbols can be **pushed** in to the STACK.
- The Top of the STACK now covers the old stack top, i.e.

$$\text{TOP} = \text{TOP} + 1$$

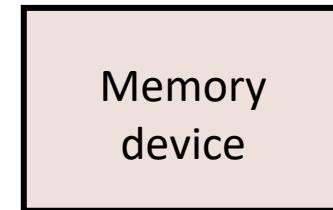
- The size of the stack keeps growing.



# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, the **top** of the STACK can be read.



### POP

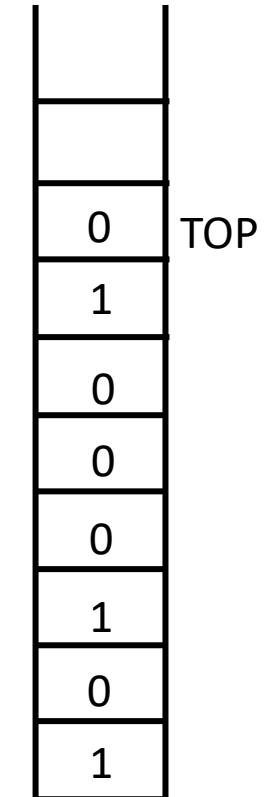
- The element from the **TOP** of the stack can be **popped** out

E.g.: **POP 0**

- The Top of the STACK moves to the element below.

$$\text{TOP} = \text{TOP} - 1$$

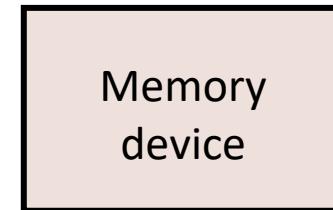
- Successive **POP** operations shrink the stack size. Elements can be popped until **EMPTY**.
- **Last In First Out (LIFO)**: The last element that was pushed is the first to be popped out



# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, the **top** of the STACK can be read.



### POP

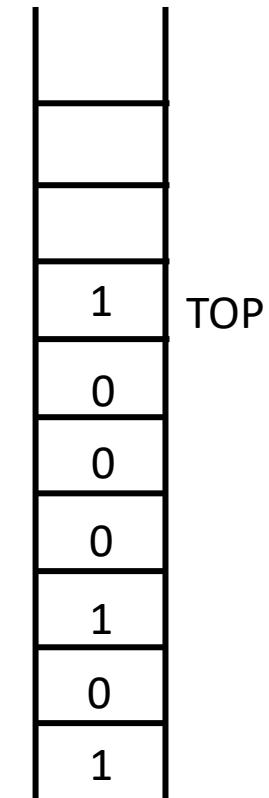
- The element from the **TOP** of the stack can be **popped** out

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, the **top** of the STACK can be read.



### POP

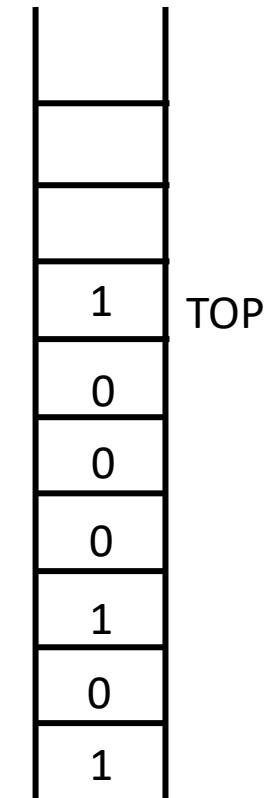
- The element from the **TOP** of the stack can be **popped** out

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, the **top** of the STACK can be read.

Memory  
device

### POP

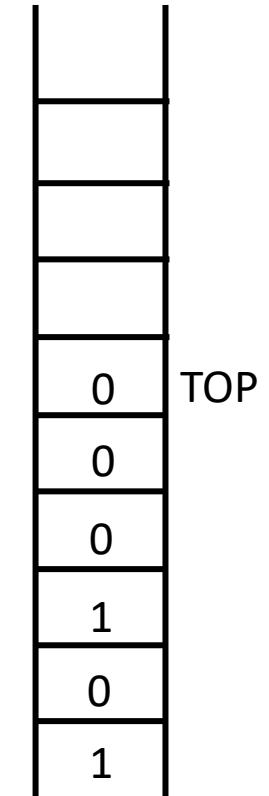
- The element from the **TOP** of the stack can be **popped** out

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



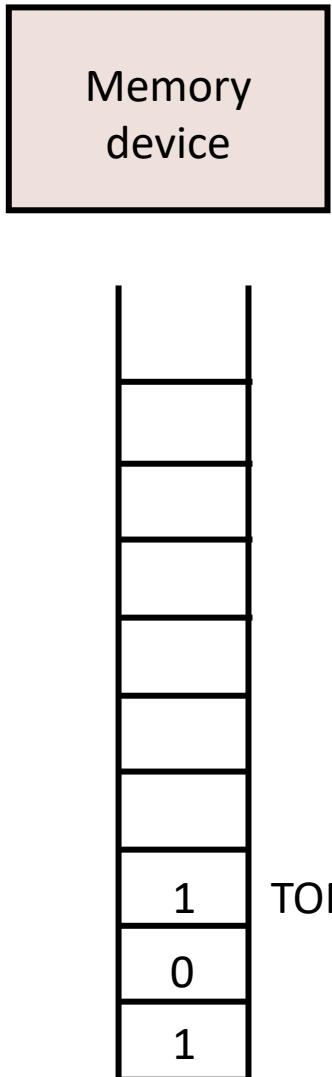
# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, the **top** of the STACK can be read.
- **LIFO**

### 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?



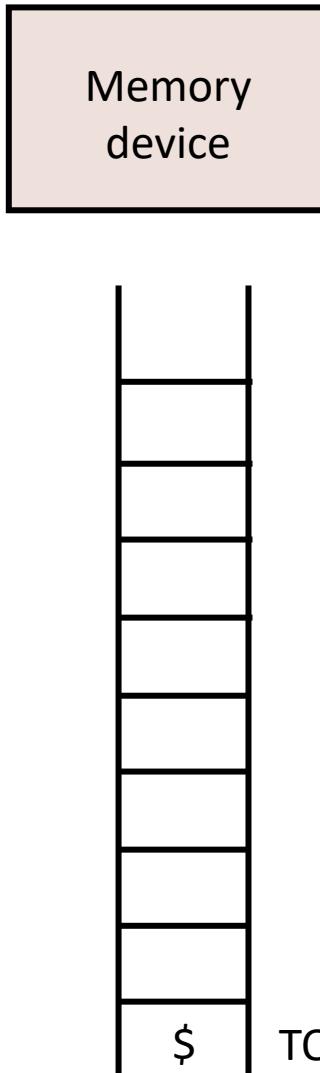
# Pushdown Automata

## The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, the **top** of the STACK can be read.
- **LIFO**

### 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:

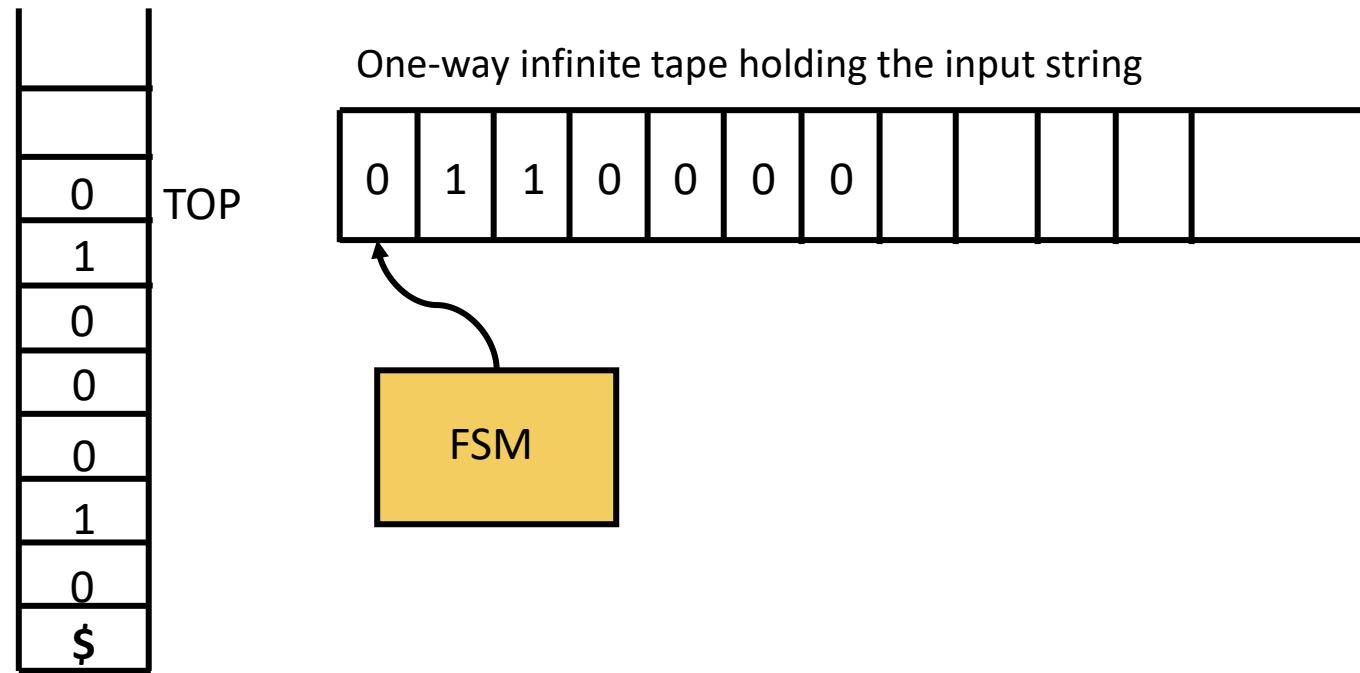
### PUSH

- New symbols can be **pushed** in to the STACK.
- $\text{TOP} = \text{TOP} + 1$

### POP

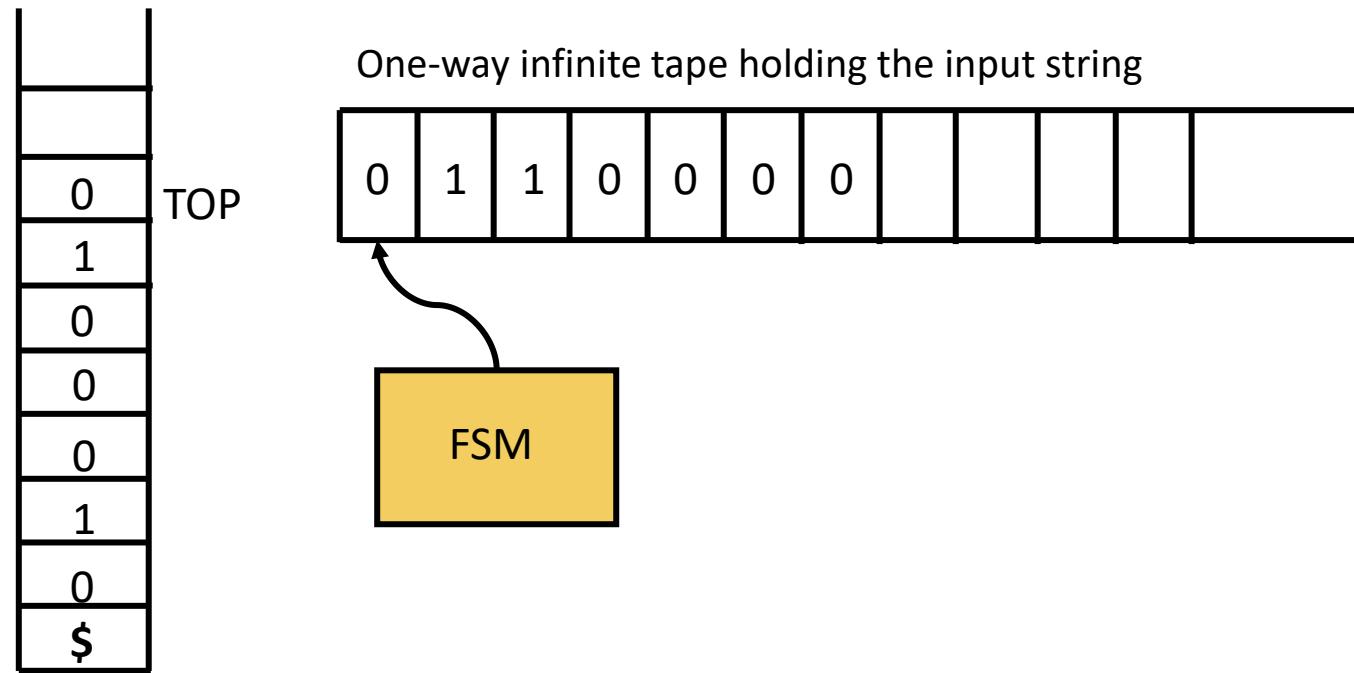
- The element from the TOP of the stack can be **popped** out.
- $\text{TOP} = \text{TOP} - 1$
- Elements can be popped until STACK is EMPTY.

# Pushdown Automata



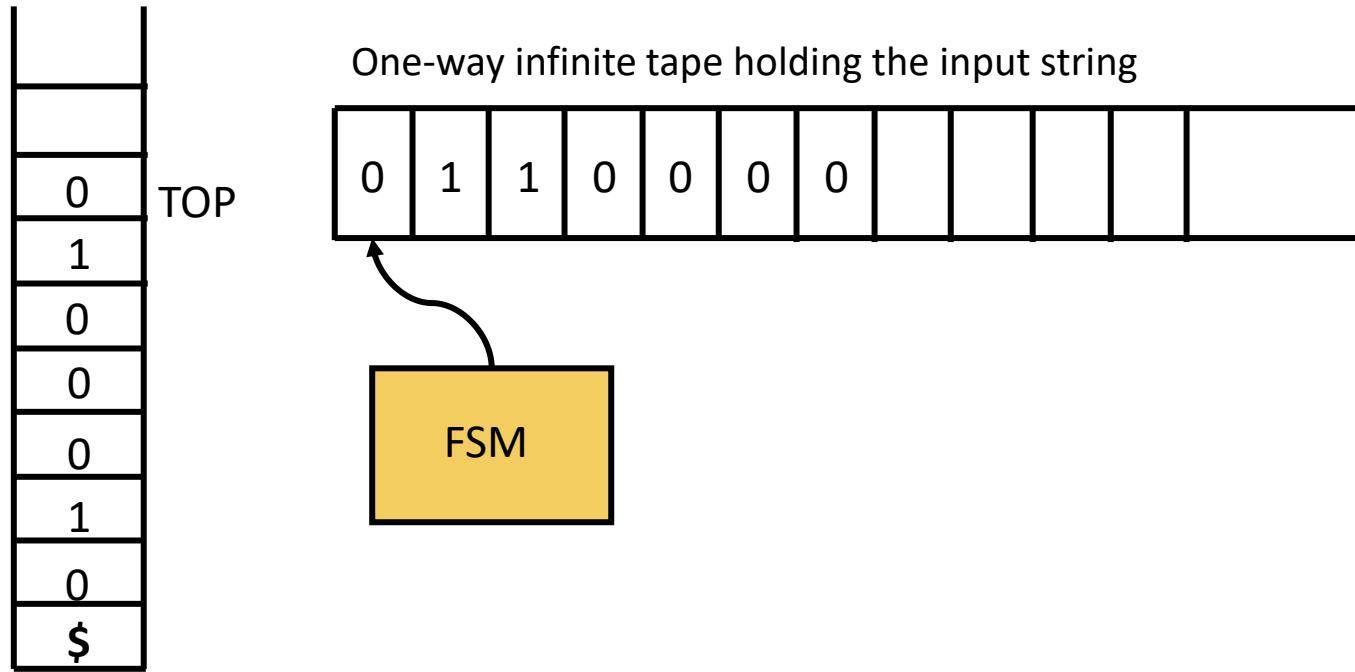
- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:

# Pushdown Automata



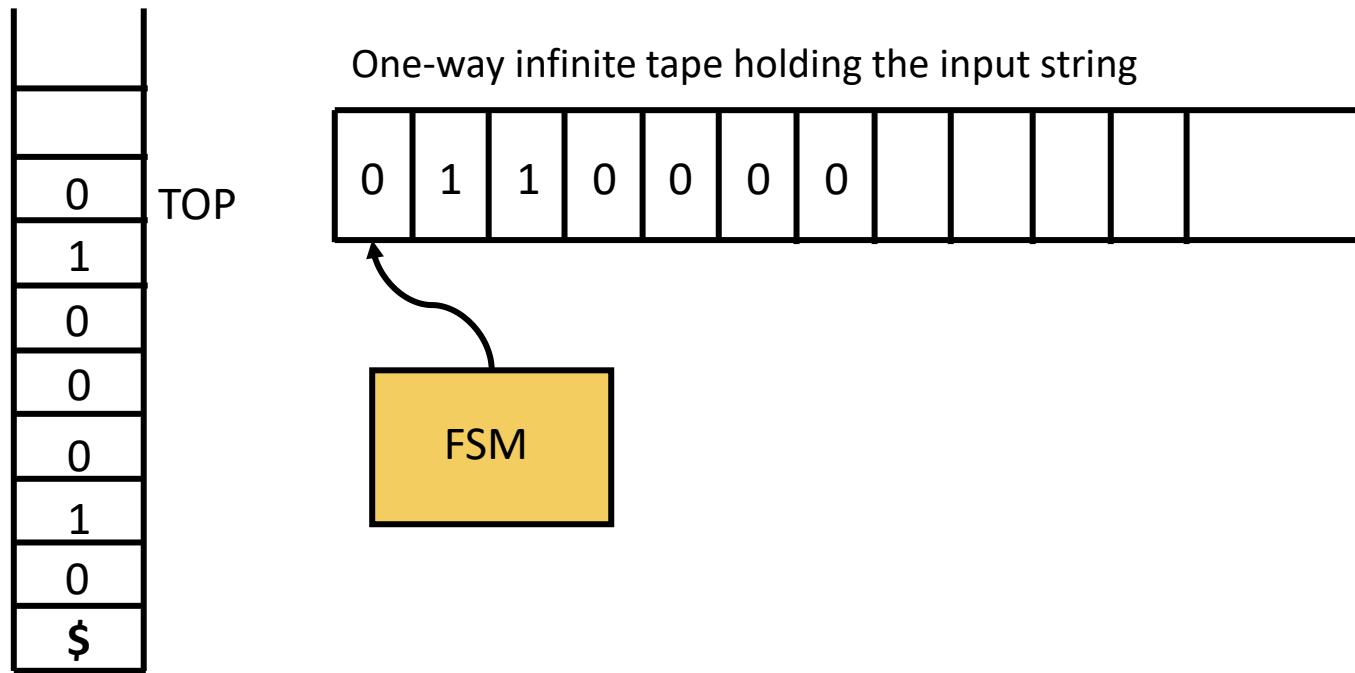
- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:
  - Transitions based on the Input symbol and the element at the top of the stack (e.g.: If I/P symbol = 0 & TOP = 0, transition from  $i$  to  $j$ )

# Pushdown Automata



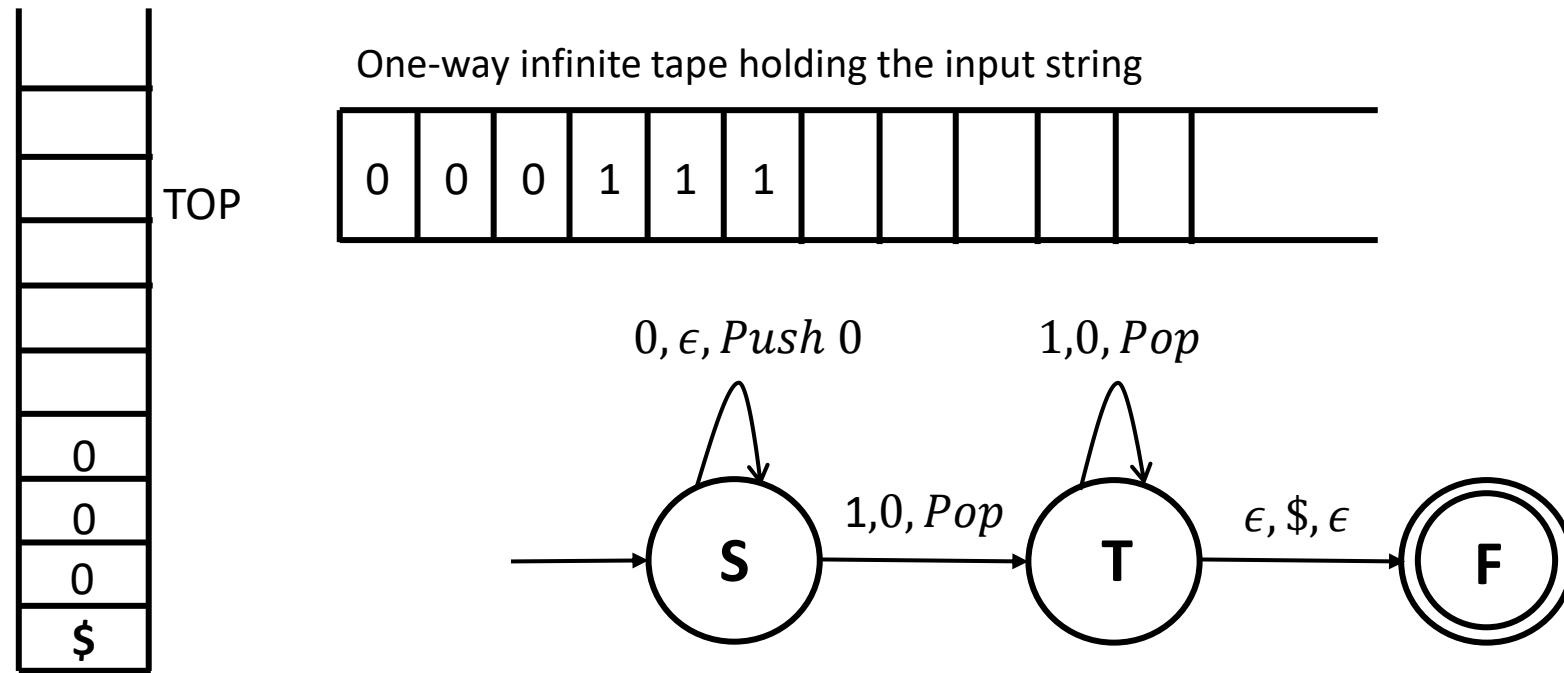
- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:
  - Transitions based on the Input symbol and the element at the top of the stack (e.g.: If I/P symbol = 0 & TOP = 0, transition from  $i$  to  $j$ )
  - Pops the element at the top of the Stack (e.g.: If I/P symbol = 0, Pop and remain at  $i$ ).

# 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  $j$ )
  - Pops the element at the top of the Stack (e.g.: If I/P symbol = 0, Pop and remain at  $i$ ).
  - 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.

**Thank You!**

# CS 302.1 - Automata Theory

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

**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]

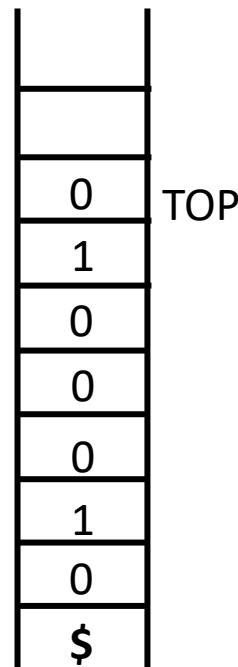
$$S \rightarrow \epsilon$$

[ $S$  is the Start Variable]

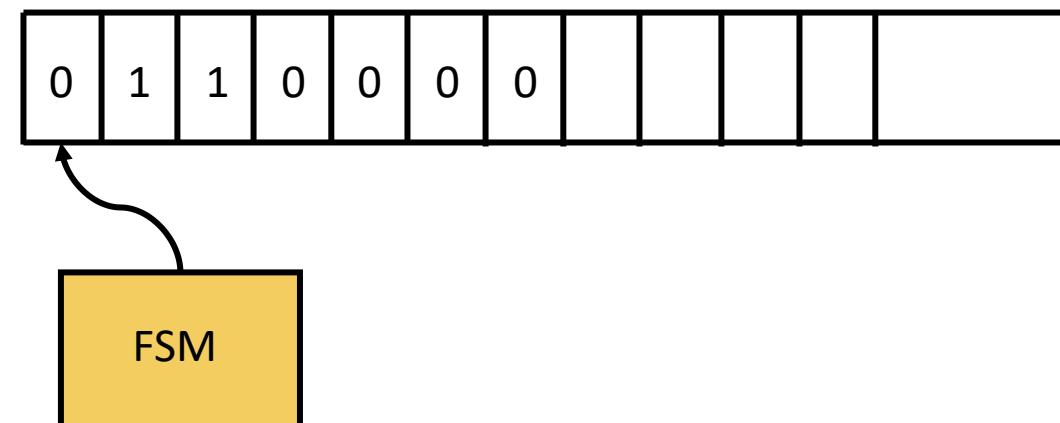
- Any CFG can be converted to a grammar in CNF that generates the same language.
- The number of steps required to derive a string  $w = 2|w| - 1$ .
- Is crucial in deciding whether  $w$  is generated by a CFG  $G$ .

## Pushdown Automata

- Automata that recognizes CFLs
- FSM + stack
- FSM transitions by reading an input symbol and by interacting with the stack



One-way infinite tape holding the input string



# Pushdown Automata

PDAs are **non-deterministic**. (Multiple transitions/input symbol possible)

# Pushdown Automata

PDAs are **non-deterministic**. (Multiple transitions/input symbol possible)

**Informally**, the PDA for some language may work as follows:

- Read symbols from the input.

# Pushdown Automata

PDAs are **non-deterministic**. (Multiple transitions/input symbol possible)

**Informally**, the PDA for some language may work as follows:

- Read symbols from the input.
- As each 0 is read, push 0 on to the stack and remain in the state  $Q_0$ .

# Pushdown Automata

PDAs are **non-deterministic**. (Multiple transitions/input symbol possible)

**Informally**, the PDA for some language may work as follows:

- Read symbols from the input.
- As each 0 is read, push 0 on to the stack and remain in the state  $Q_0$ .
- If FSM is at  $Q_0$ , and a 1 is read, pop a 0 off the Stack and transition to  $Q_1$ .

# Pushdown Automata

PDAs are **non-deterministic**. (Multiple transitions/input symbol possible)

**Informally**, the PDA for some language may work as follows:

- Read symbols from the input.
- As each 0 is read, push 0 on to the stack and remain in the state  $Q_0$ .
- If FSM is at  $Q_0$ , and a 1 is read, pop a 0 off the Stack and transition to  $Q_1$ .
- If FSM is at  $Q_1$ , and a 1 is read, pop a 0 off the Stack and remain at  $Q_1$ .

# Pushdown Automata

PDAs are **non-deterministic**. (Multiple transitions/input symbol possible)

**Informally**, the PDA for some language may work as follows:

- Read symbols from the input.
- As each 0 is read, push 0 on to the stack and remain in the state  $Q_0$ .
- If FSM is at  $Q_0$ , and a 1 is read, pop a 0 off the Stack and transition to  $Q_1$ .
- If FSM is at  $Q_1$ , and a 1 is read, pop a 0 off the Stack and remain at  $Q_1$ .
- If FSM is at  $Q_1$ , and a 1 is read, pop a 0 off the Stack, push 1 on to the stack and transition to  $Q_2$

# Pushdown Automata

PDAs are **non-deterministic**. (Multiple transitions/input symbol possible)

**Informally**, the PDA for some language may work as follows:

- Read symbols from the input.
- As each 0 is read, push 0 on to the stack and remain in the state  $Q_0$ .
- If FSM is at  $Q_0$ , and a 1 is read, pop a 0 off the Stack and transition to  $Q_1$ .
- If FSM is at  $Q_1$ , and a 1 is read, pop a 0 off the Stack and remain at  $Q_1$ .
- If FSM is at  $Q_1$ , and a 1 is read, pop a 0 off the Stack, push 1 on to the stack and transition to  $Q_2$
- If the input is finished exactly when the stack is empty ( $\text{TOP} = \$$ ), ACCEPT the input.

# Pushdown Automata

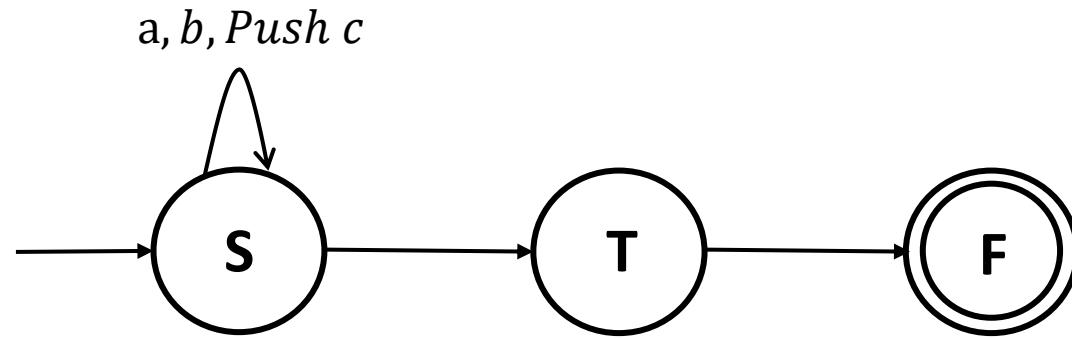
PDAs are **non-deterministic**. (Multiple transitions/input symbol possible)

**Informally**, the PDA for some language may work as follows:

- Read symbols from the input.
- As each 0 is read, push 0 on to the stack and remain in the state  $Q_0$ .
- If FSM is at  $Q_0$ , and a 1 is read, pop a 0 off the Stack and transition to  $Q_1$ .
- If FSM is at  $Q_1$ , and a 1 is read, pop a 0 off the Stack and remain at  $Q_1$ .
- If FSM is at  $Q_1$ , and a 1 is read, pop a 0 off the Stack, push 1 on to the stack and transition to  $Q_2$ .
- If the input is finished exactly when the stack is empty ( $\text{TOP} = \$$ ), ACCEPT the input.
- REJECT otherwise (Stack becomes empty before all the inputs are read/non-empty after the entire input is read)

# Pushdown Automata

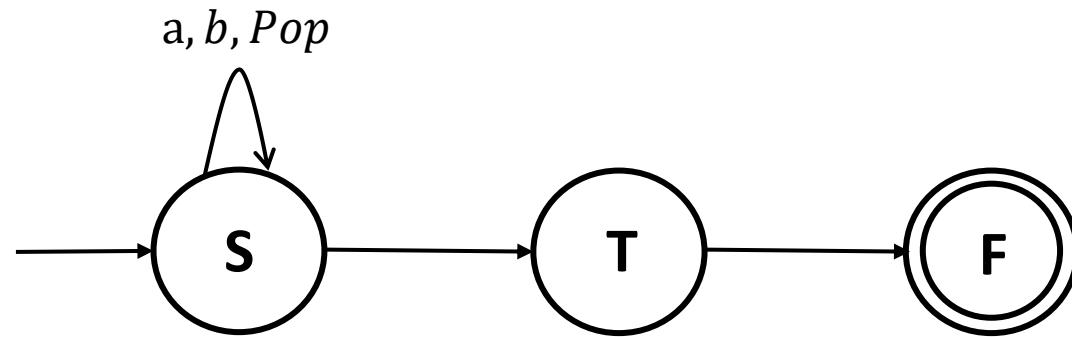
- How to represent a transition in a PDA?



If input symbol =  $a$ , Stack top =  $b$ , then Pop  $b$  and Push  $c$  onto the Stack

# Pushdown Automata

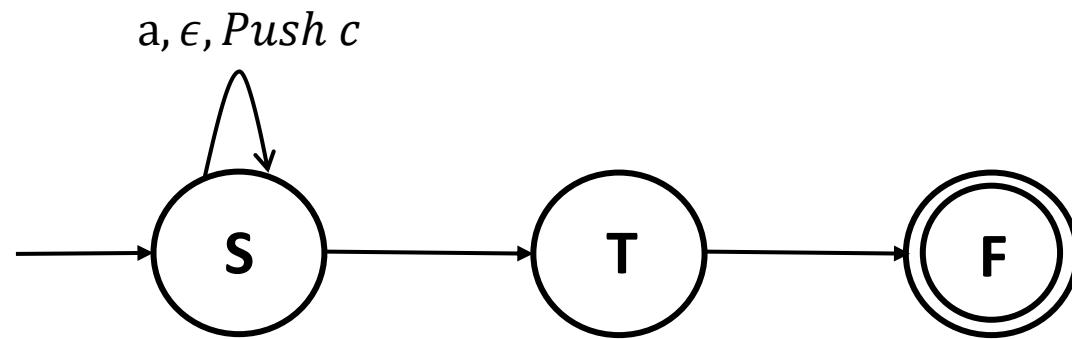
- How to represent a transition in a PDA?



If input symbol =  $a$  and Stack top =  $b$ , then Pop  $b$

# Pushdown Automata

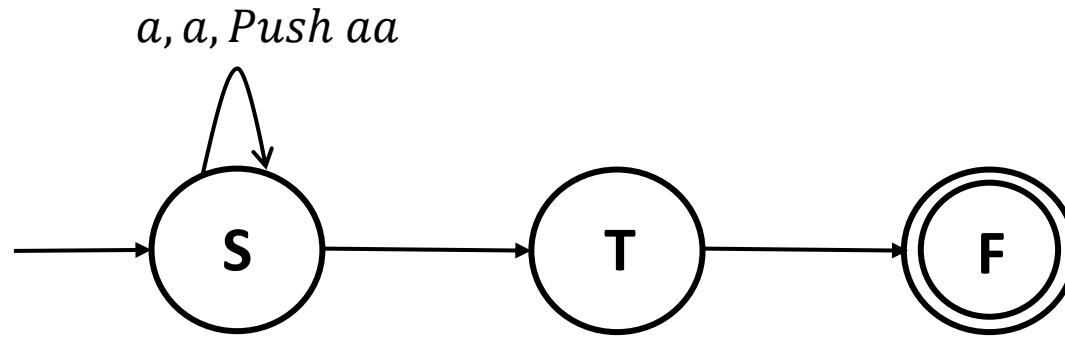
- How to represent a transition in a PDA?



If input symbol =  $a$ , then Push  $c$

# Pushdown Automata

- How to represent a transition in a PDA?

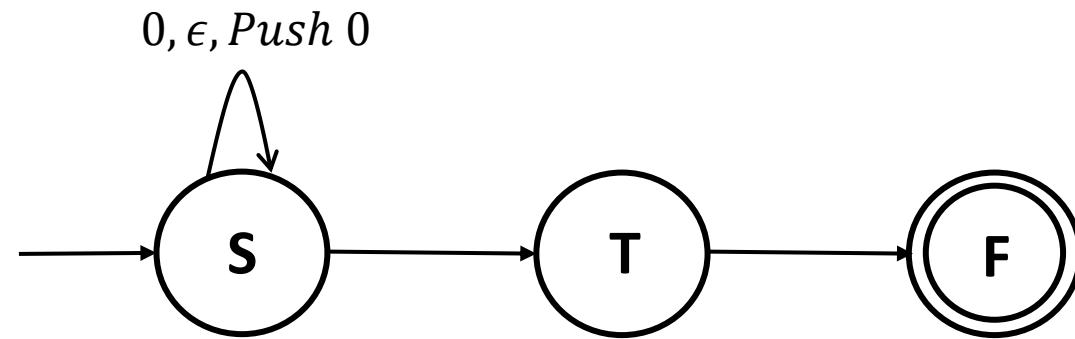


If input symbol =  $a$ , then Pop  $a$  and Push  $aa$ .

So effectively, the PDA pushes  $a$  onto the stack if it reads  $a$  on the input tape and the stack top =  $a$ .

# Pushdown Automata

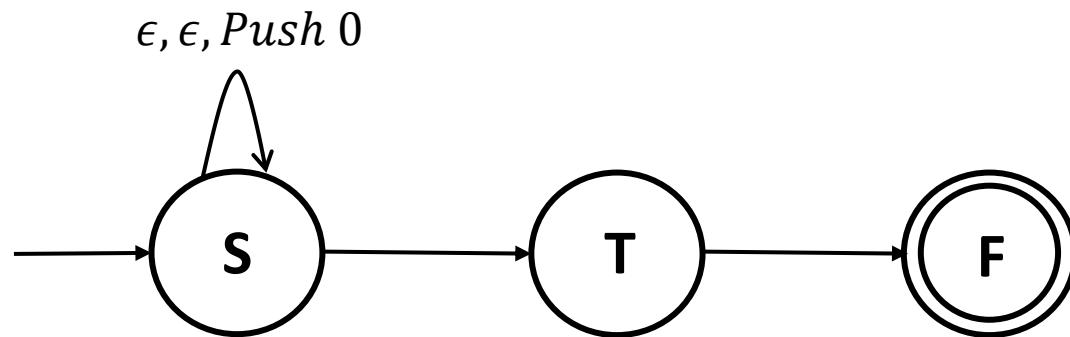
- How to represent a transition in a PDA?



If input symbol = 0, Push 0 onto the Stack irrespective of the element at the top of the stack

# Pushdown Automata

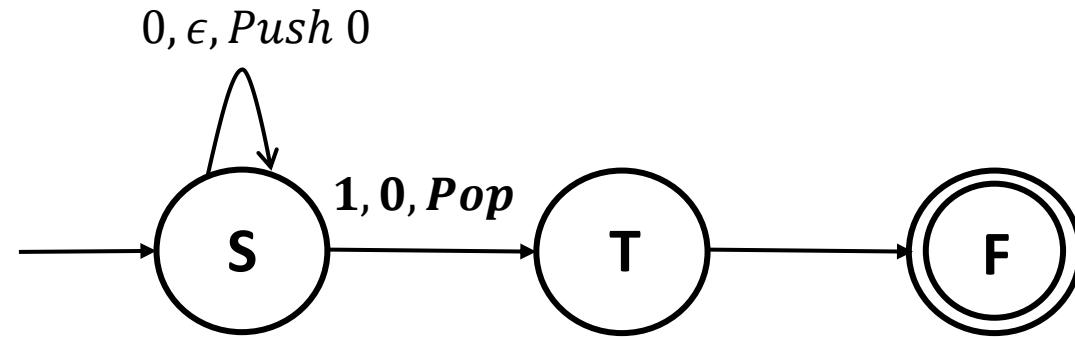
- How to represent a transition in a PDA?



Without reading the input symbol and the Stack top, Push 0 onto the Stack

# Pushdown Automata

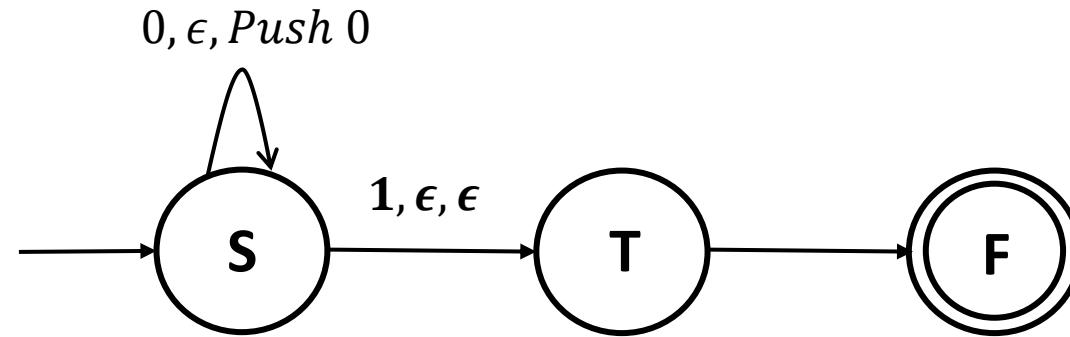
- How to represent a transition in a PDA?



If the input symbol is 1, and the element at the top of the stack is 0, pop it (**Pop 0**).

# Pushdown Automata

- How to represent a transition in a PDA?

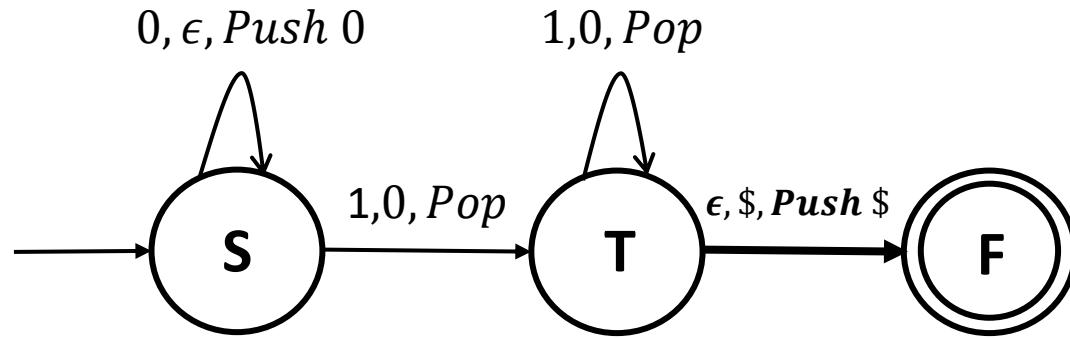


If the input symbol is 1, transition to  $T$  by ignoring the stack top completely.

If this happens at every step of the execution of the PDA, then it is as powerful as an NFA.

# Pushdown Automata

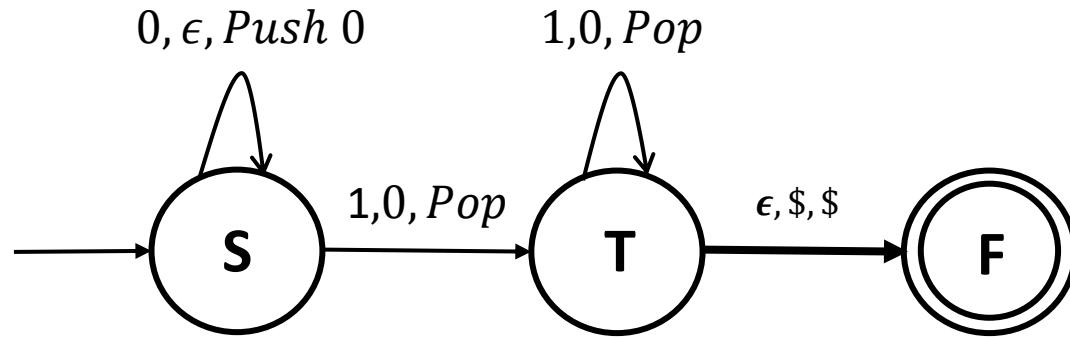
- How to represent a transition in a PDA?



If the Stack is empty, i.e.  $\text{TOP} = \$$ , transition to  $F$  from  $T$ , without reading the input

# Pushdown Automata

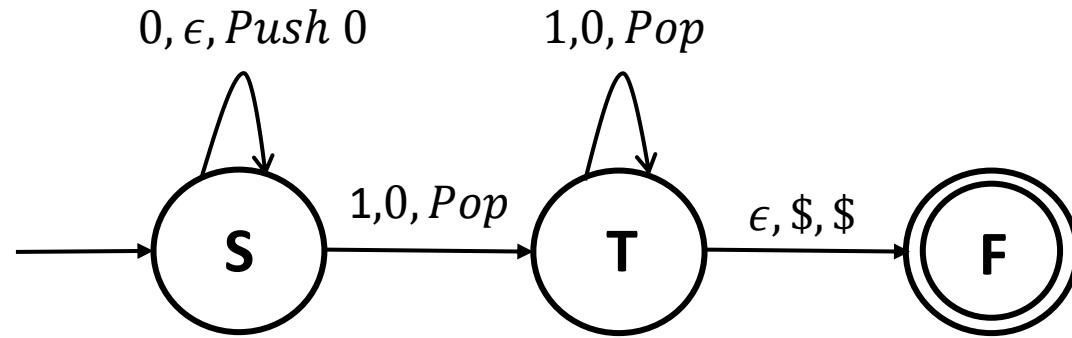
- How to represent a transition in a PDA?



If the Stack is empty, i.e.  $\text{TOP} = \$$ , transition to  $F$  from  $T$ , without reading the input

# Pushdown Automata

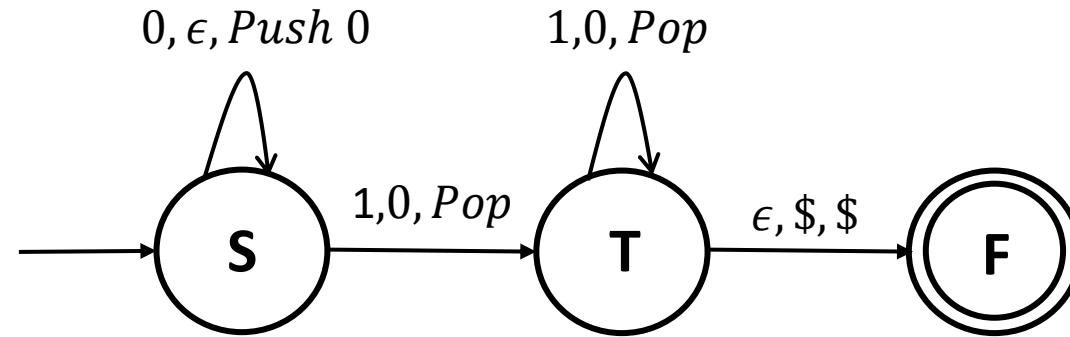
- How to represent a transition in a PDA?



What is the language accepted by this PDA?

# Pushdown Automata

- How to represent a transition in a PDA?

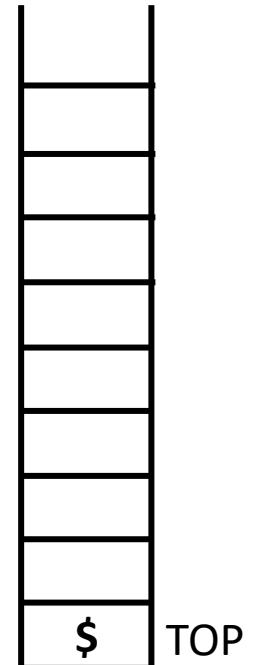
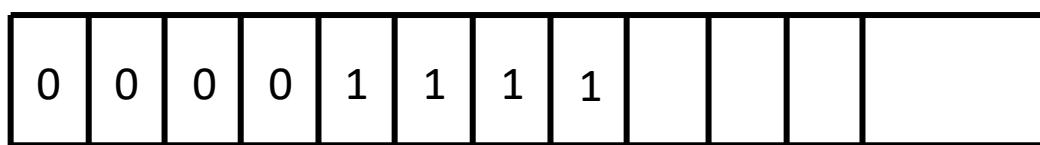
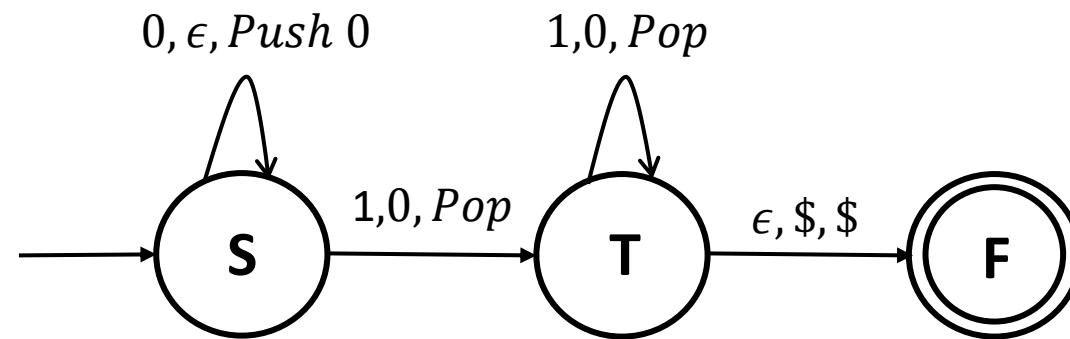


What is the language recognized by this PDA?

Verify that it is  $L = \{0^n 1^n, n \geq 1\}$

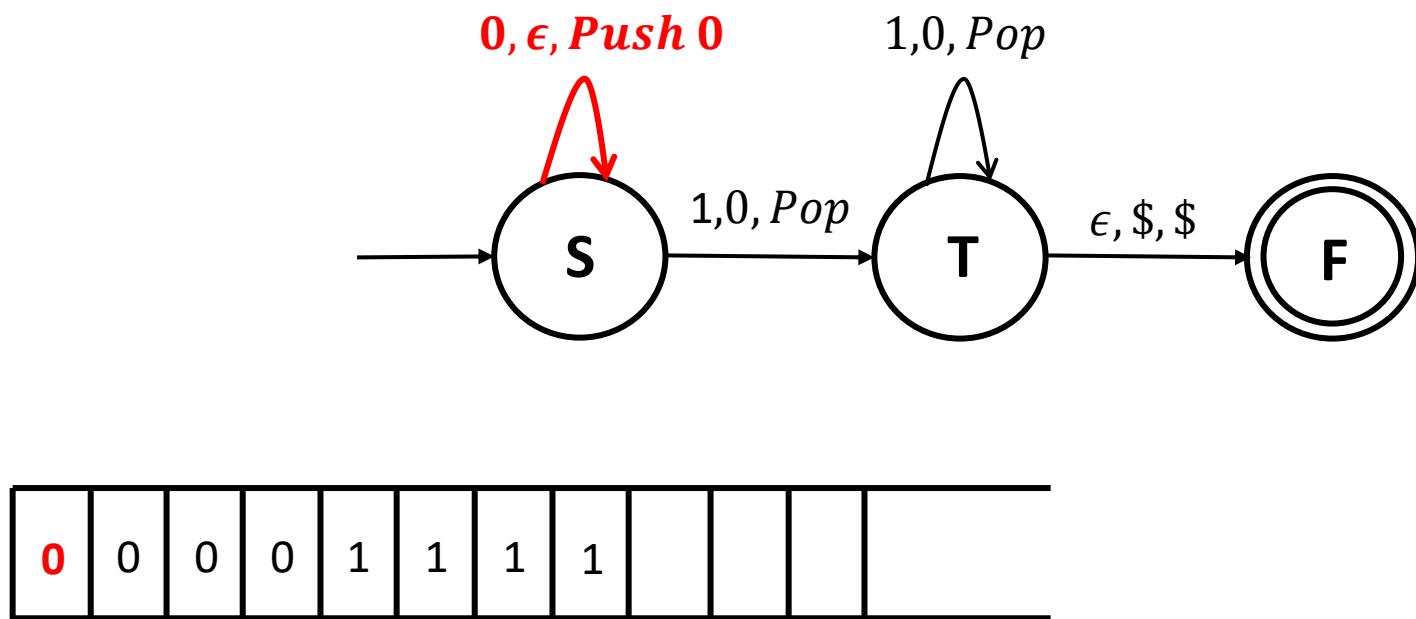
# Pushdown Automata

What is the language recognized by this PDA?



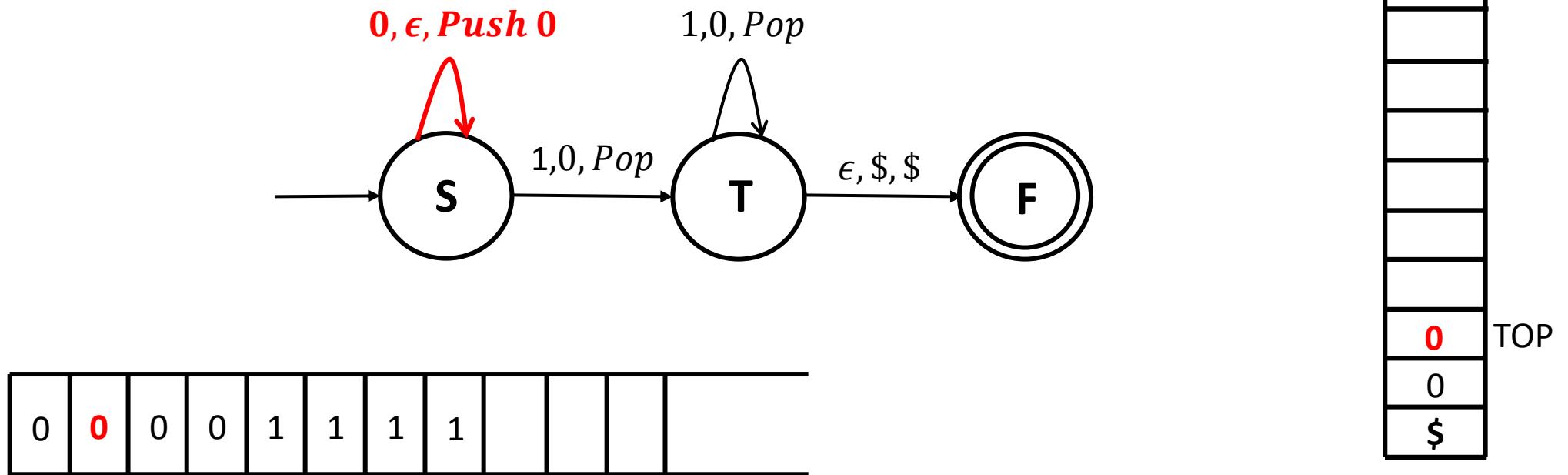
# Pushdown Automata

## What is the language recognized by this PDA?



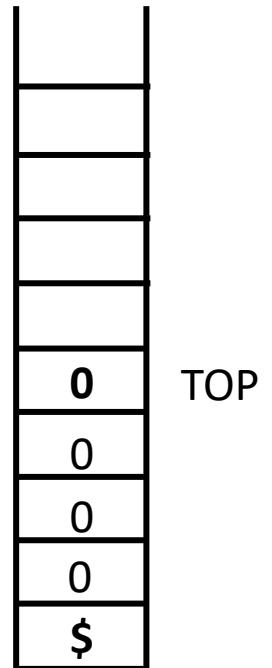
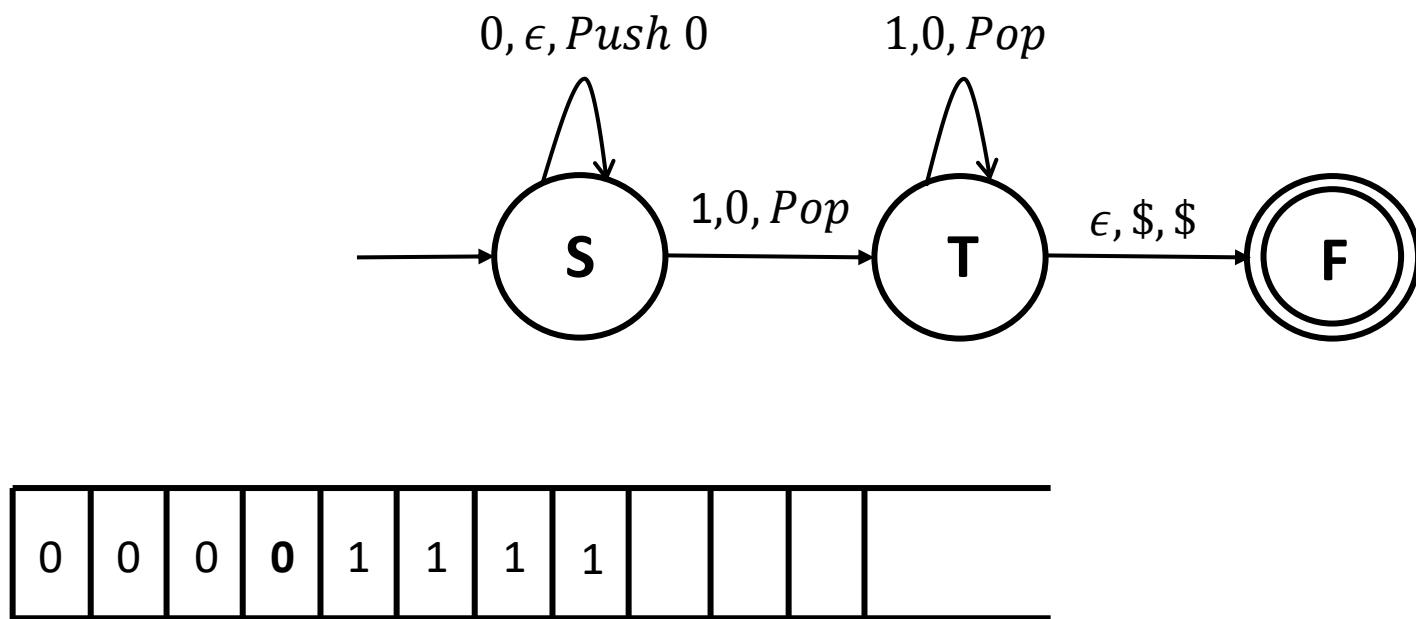
# Pushdown Automata

What is the language recognized by this PDA?



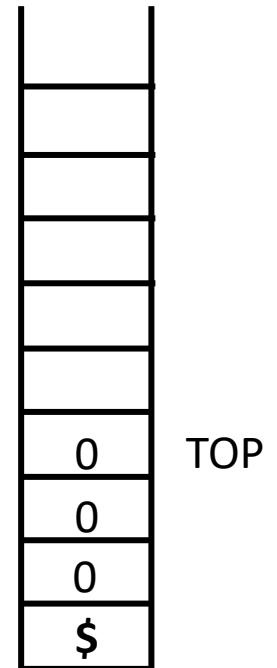
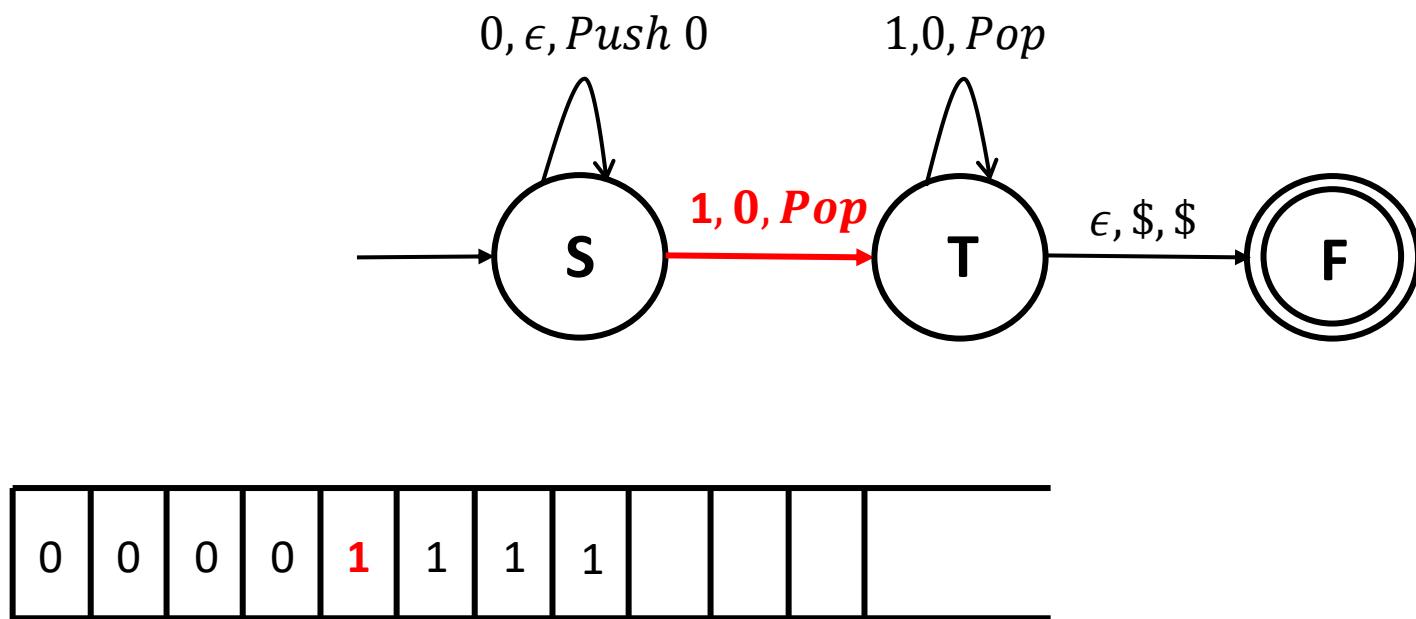
# Pushdown Automata

What is the language recognized by this PDA?



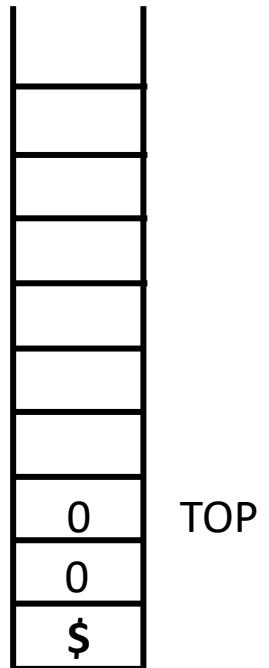
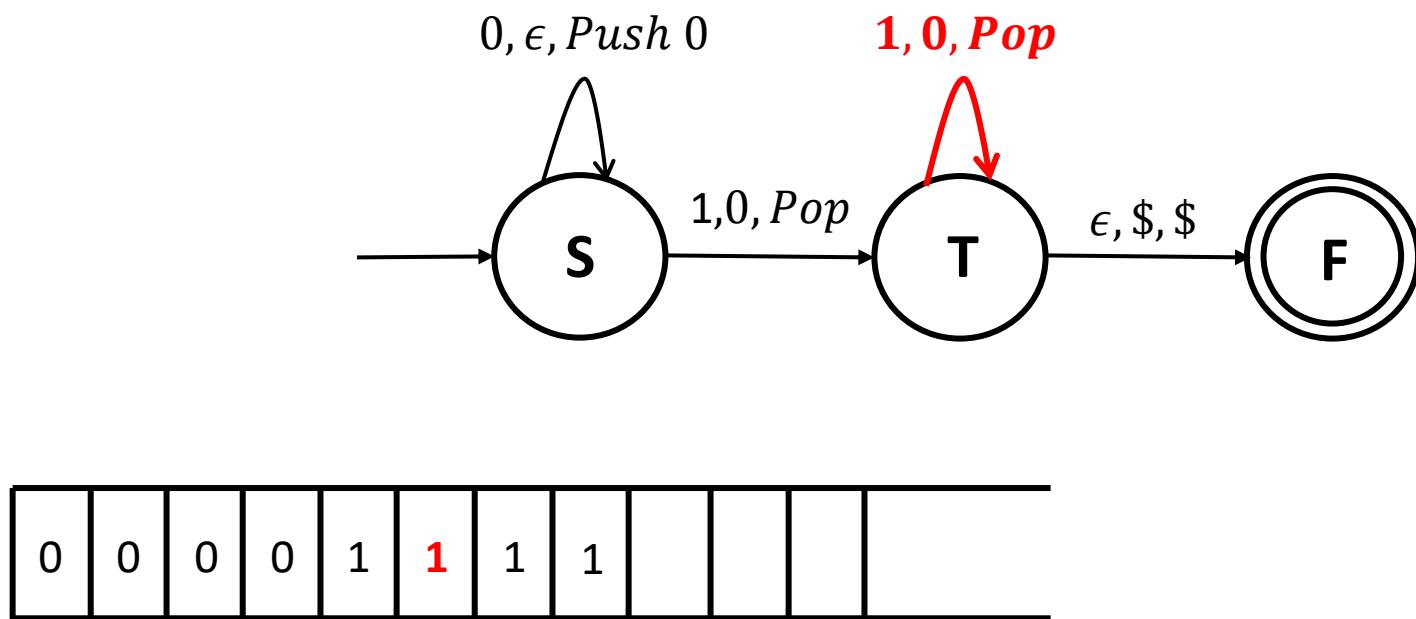
# Pushdown Automata

What is the language recognized by this PDA?



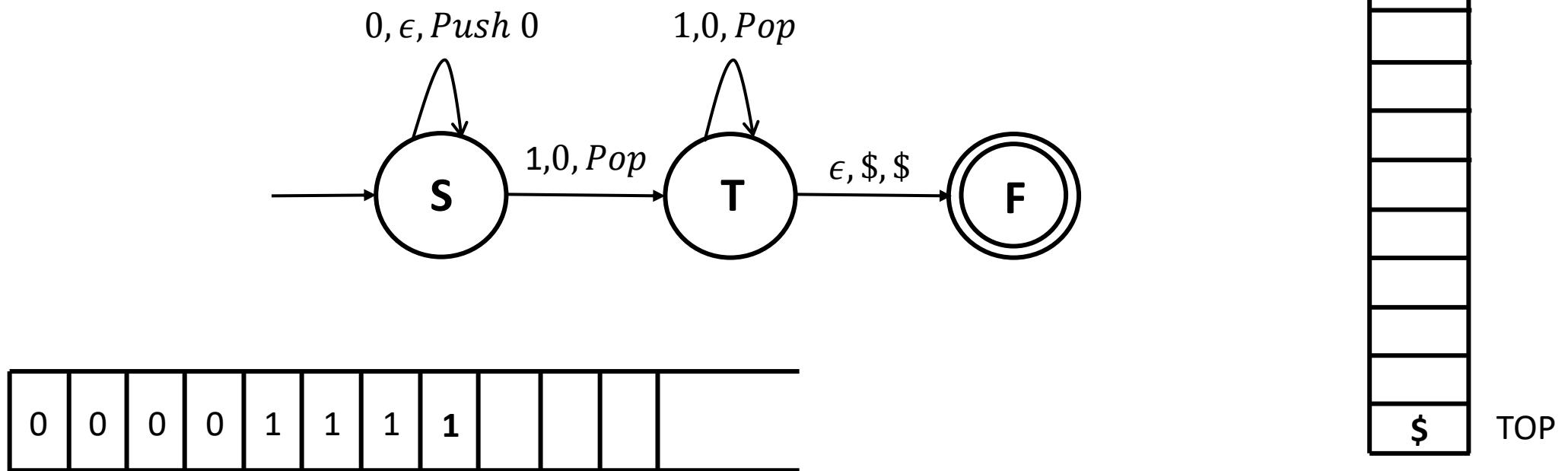
# Pushdown Automata

What is the language recognized by this PDA?



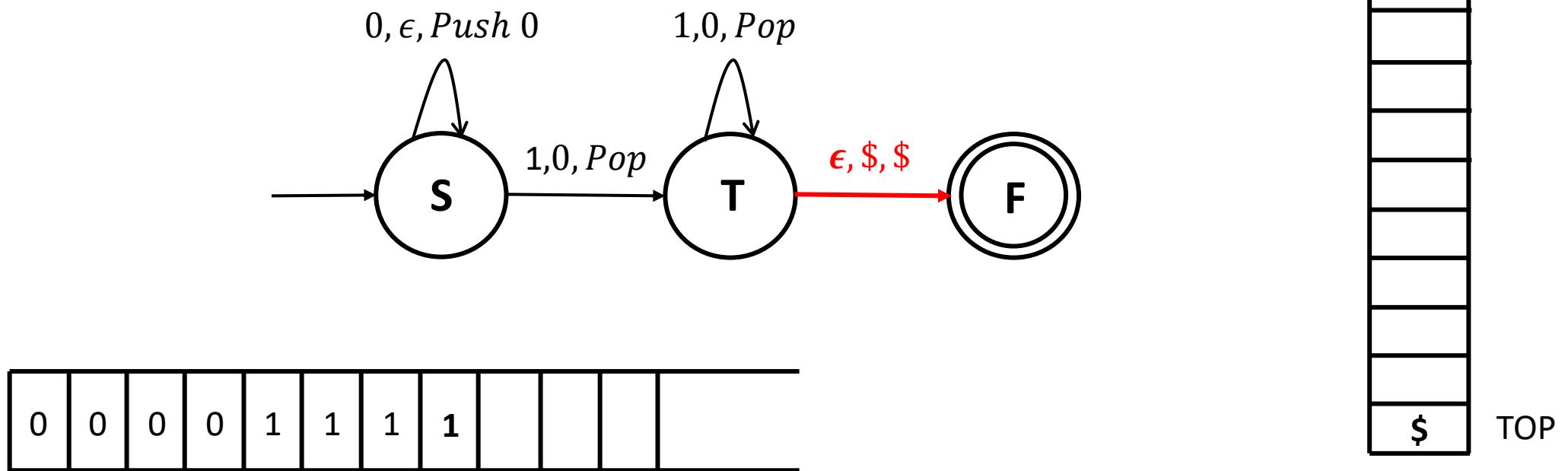
# Pushdown Automata

## What is the language recognized by this PDA?



# Pushdown Automata

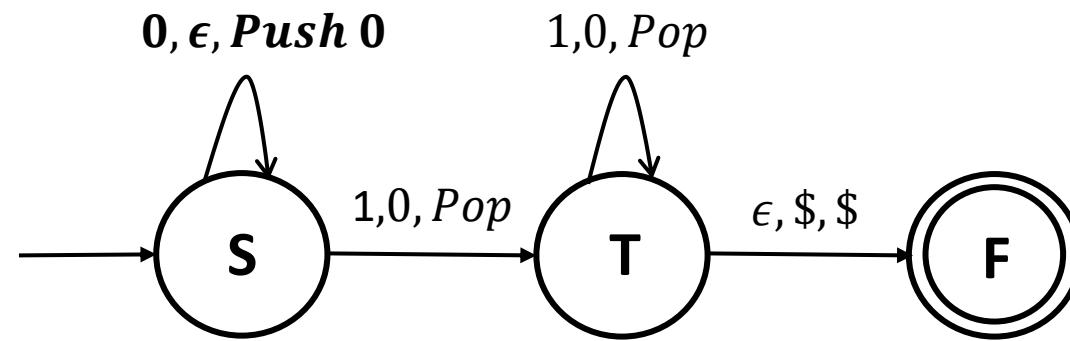
What is the language recognized by this PDA?



The language recognized by the PDA:  $L = \{0^n 1^n, n \geq 1\}$

# Pushdown Automata

What is the language recognized by this PDA?

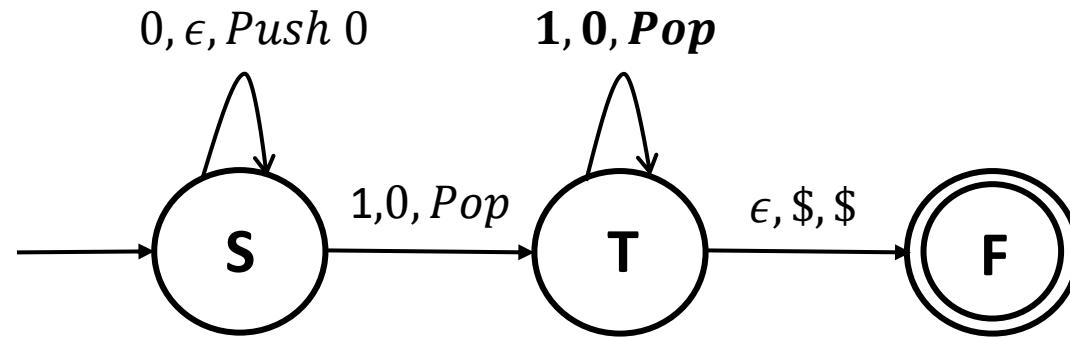


In some references (such as Sipser):

- 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  $b$ , then pop  $b$  and push  $c$  on to the Stack.

# Pushdown Automata

What is the language recognized by this PDA?

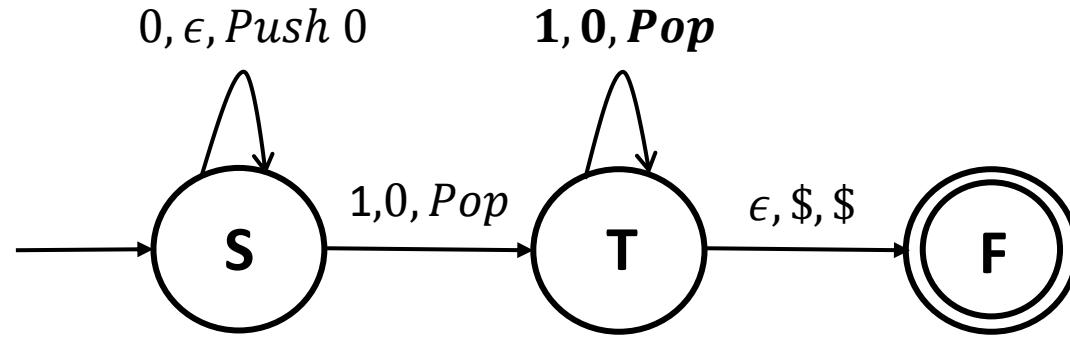


In some references (such as Sipser):

- 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  $a$  and the element at the top of the stack is  $b$ , then **pop**.

# 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 the element at the top of the stack is  $b$ , then **pop**.
- 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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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 the stack top =  $b$ , then pop  $b$ , 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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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 the stack top =  $b$ , then pop  $b$ , 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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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 the stack top =  $b$ , then pop  $b$ , 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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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 the stack top =  $b$ , then pop  $b$ , 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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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 the stack top =  $b$ , then pop  $b$ , 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  $q_j$
- $\delta(q_i, \epsilon, \$) = (q_j, \$)$ :

# 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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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 the stack top =  $b$ , then pop  $b$ , 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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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 the stack top =  $b$ , then pop  $b$ , push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- If the input symbol read is  $a$  and the stack top =  $a$ , 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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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 the stack top =  $b$ , then pop  $b$ , push  $c$  onto the stack and transition from  $q_i$  to  $q_j$
- If the input symbol read is  $a$  and the stack top =  $a$ , then Push  $a$  and remain at  $q_i$  :  $\delta(q_i, a, a) = (q_i, aa)$

# 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 **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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  $L$

# Pushdown Automata

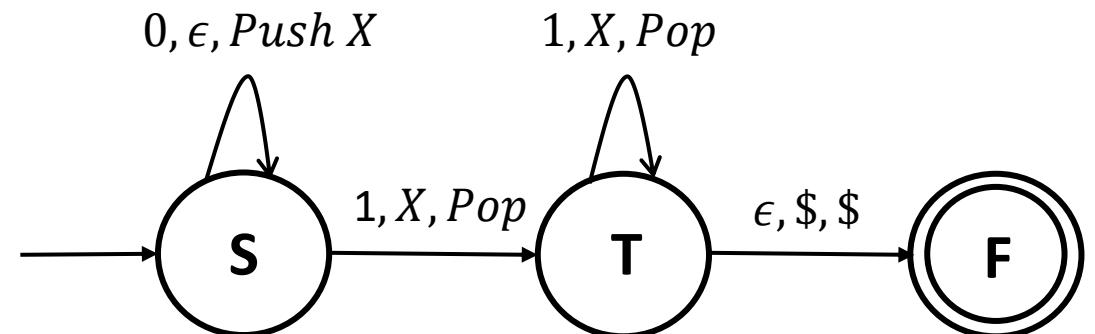
Formally, a PDA  $M$  is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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

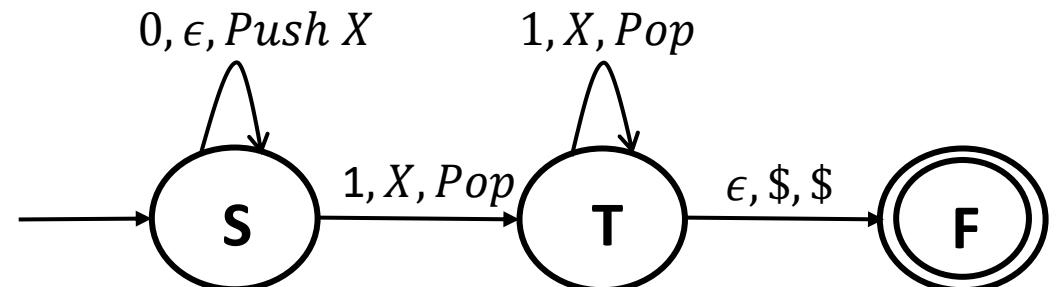
Formally, a PDA  $M$  is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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

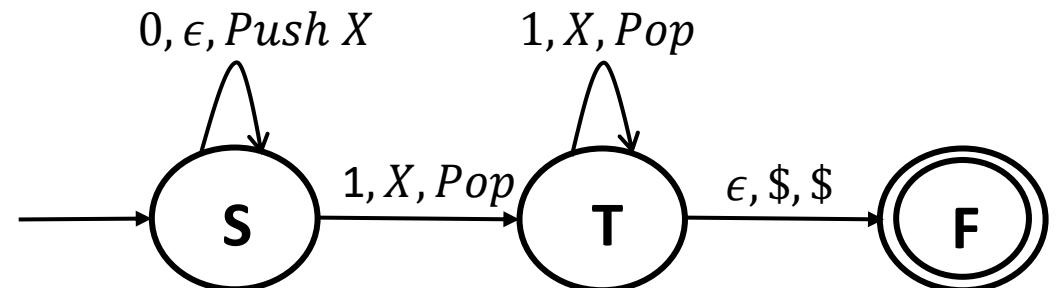
Formally, a PDA  $M$  is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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

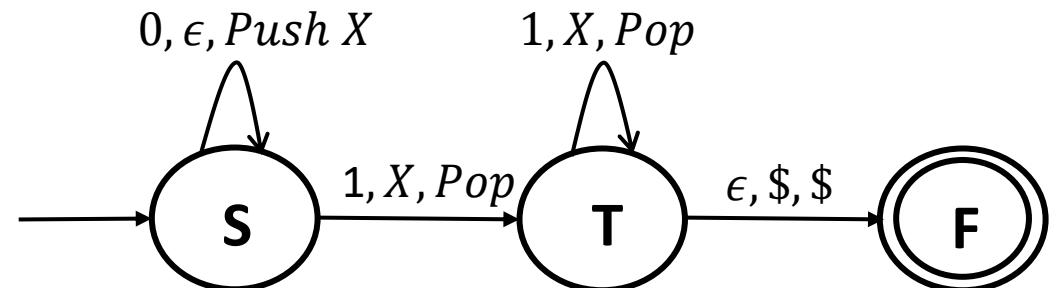
Formally, a PDA  $M$  is a 6-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets**.
- $\Gamma$  is the **Stack alphabet**
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$  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

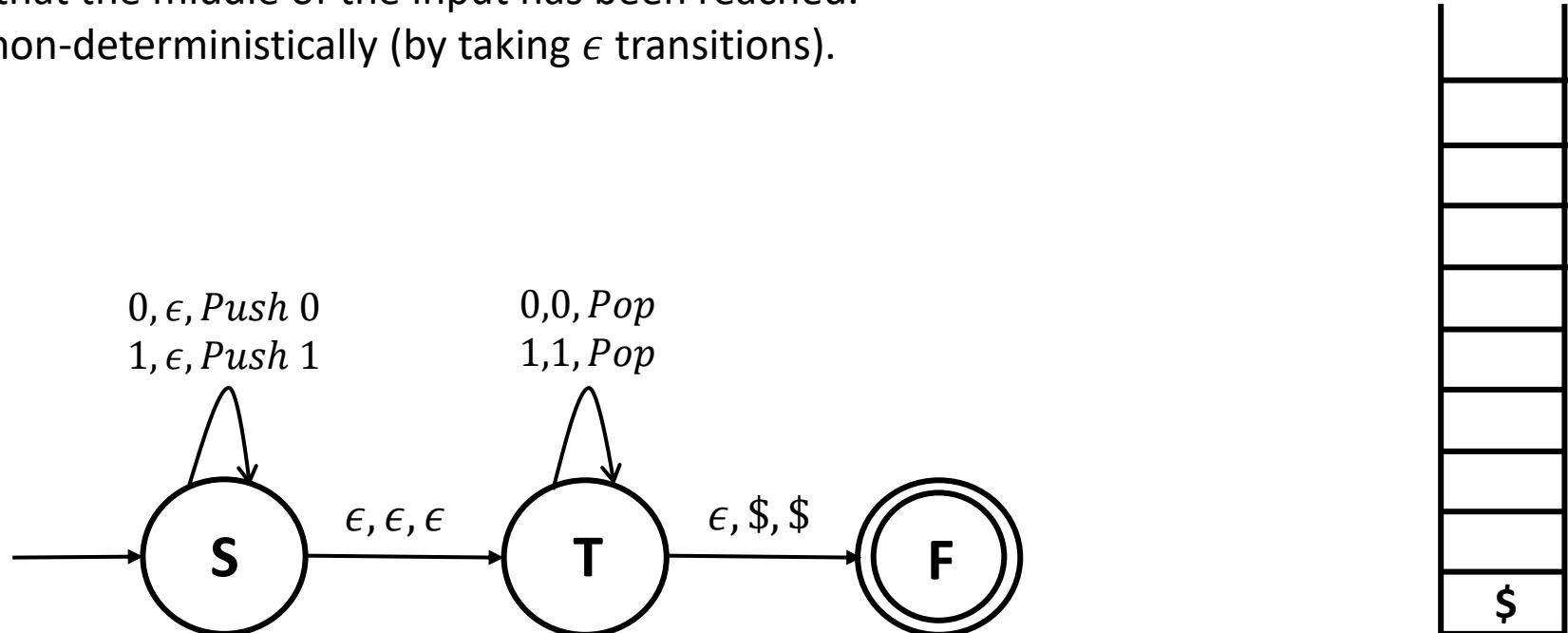
- Push first half of the input string onto the stack.
- Verify that the second half of the symbols match the first half: Keep Popping the stack until the end of the input.
- How can the PDA know that the middle of the input has been reached.
  - The PDA does this non-deterministically (by taking  $\epsilon$  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).

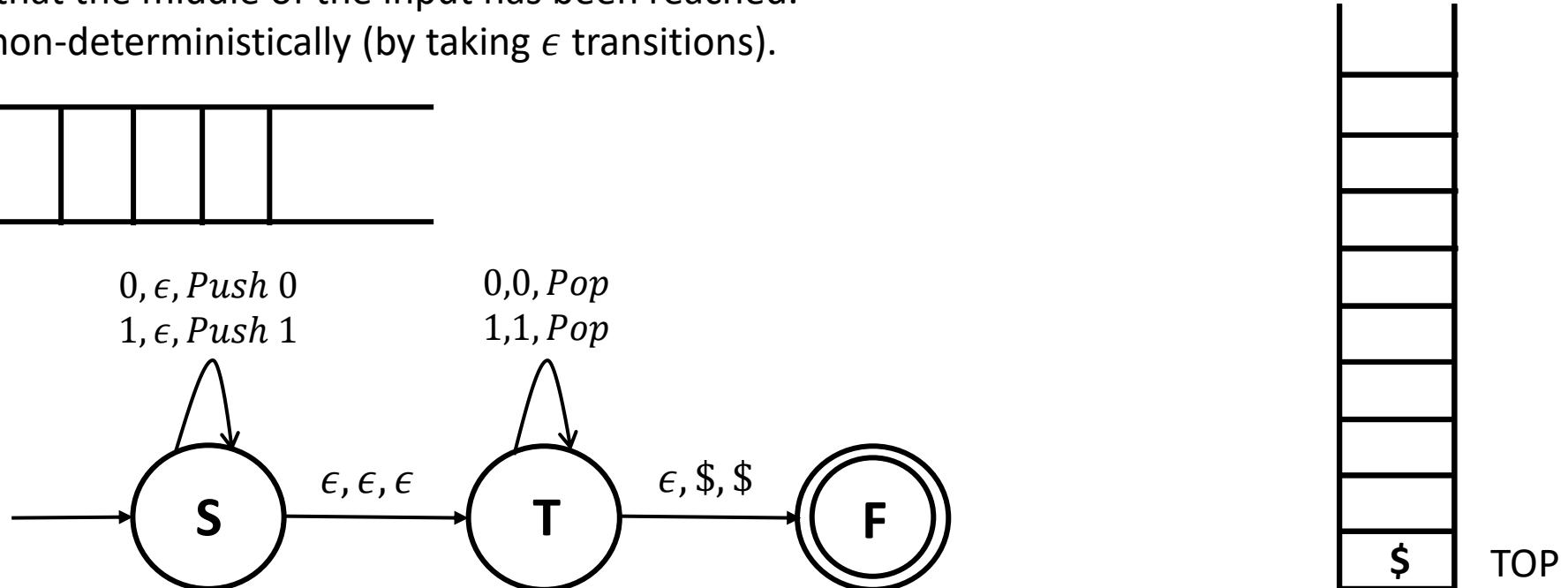
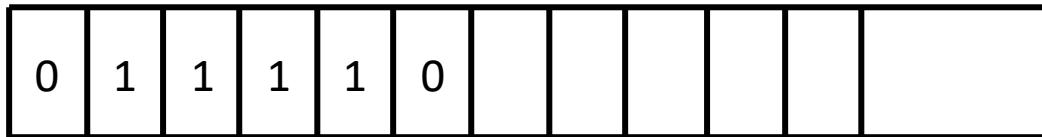


# Pushdown Automata

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).

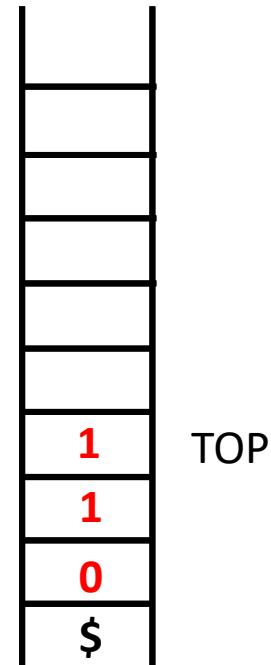
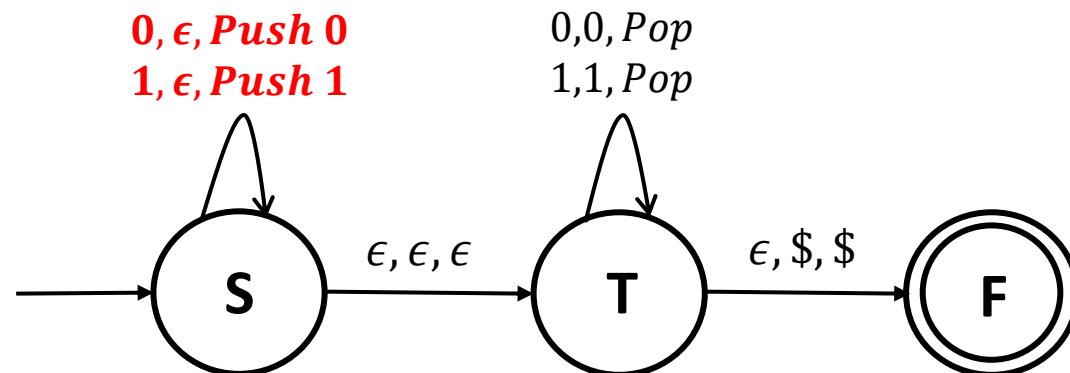
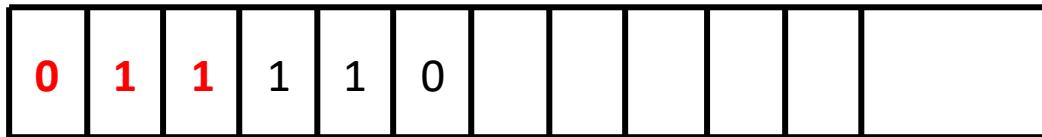


# Pushdown Automata

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).

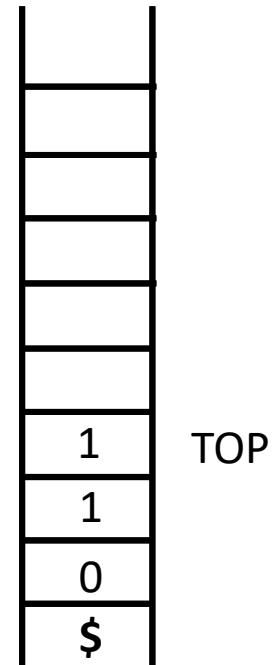
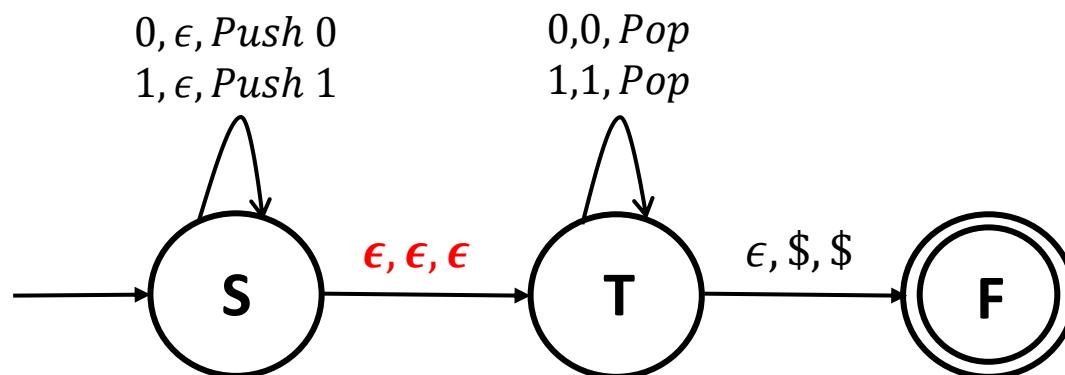
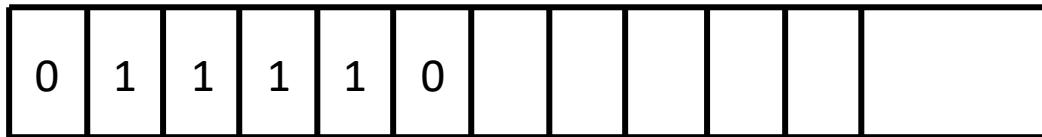


# Pushdown Automata

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).

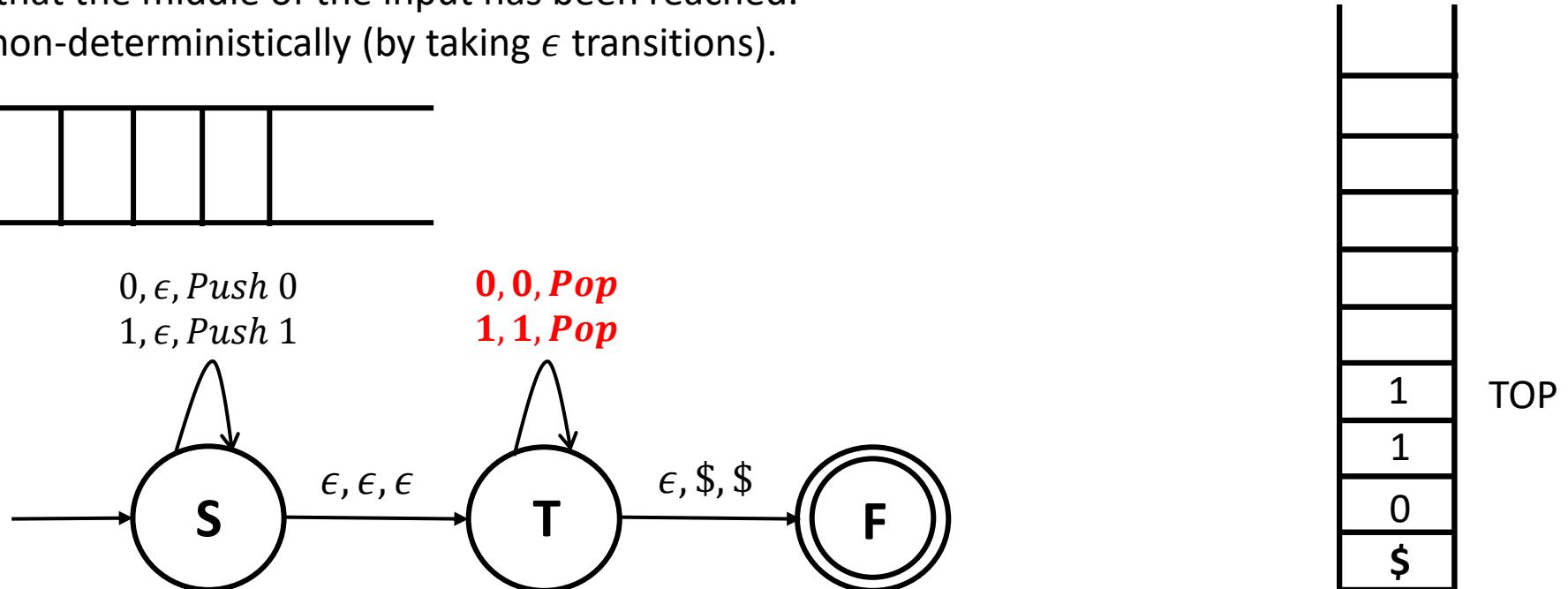


# Pushdown Automata

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).

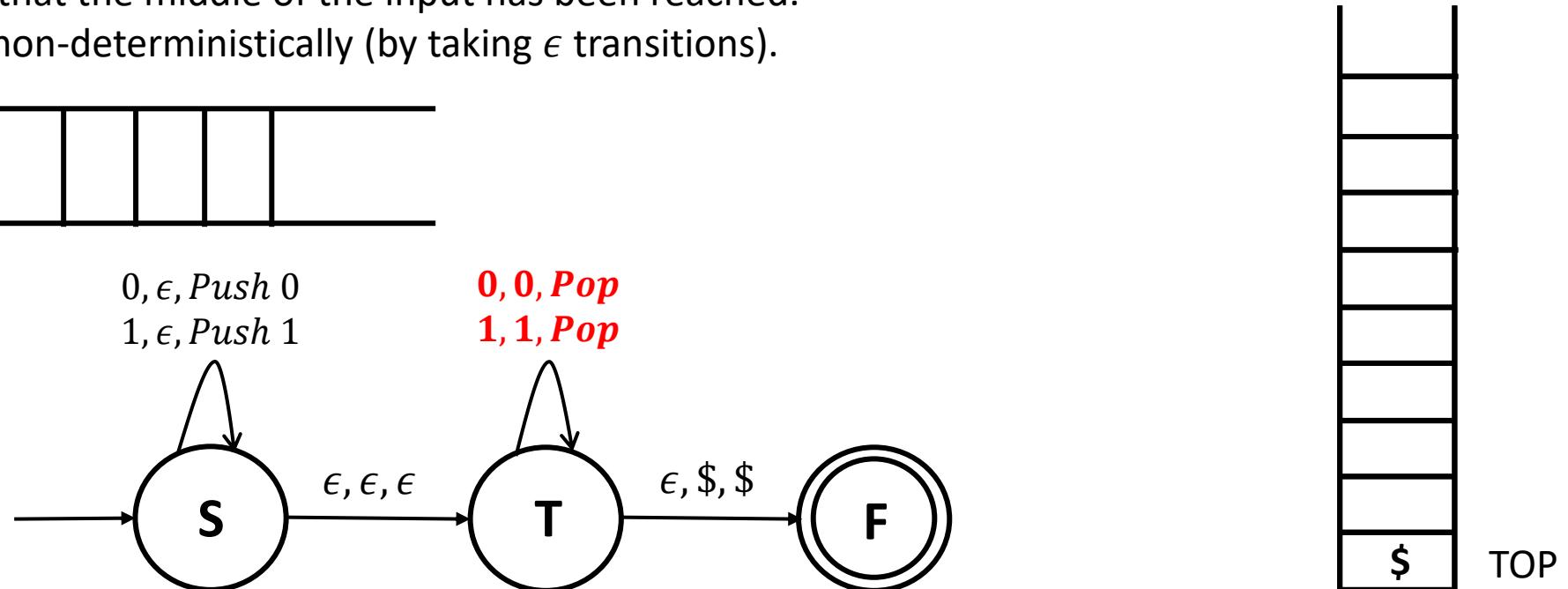
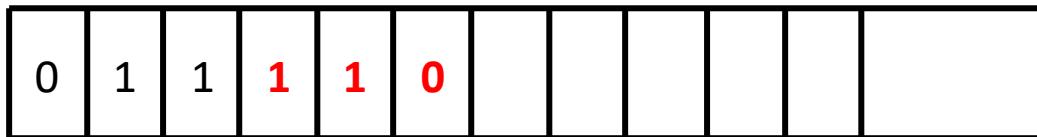


# Pushdown Automata

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).

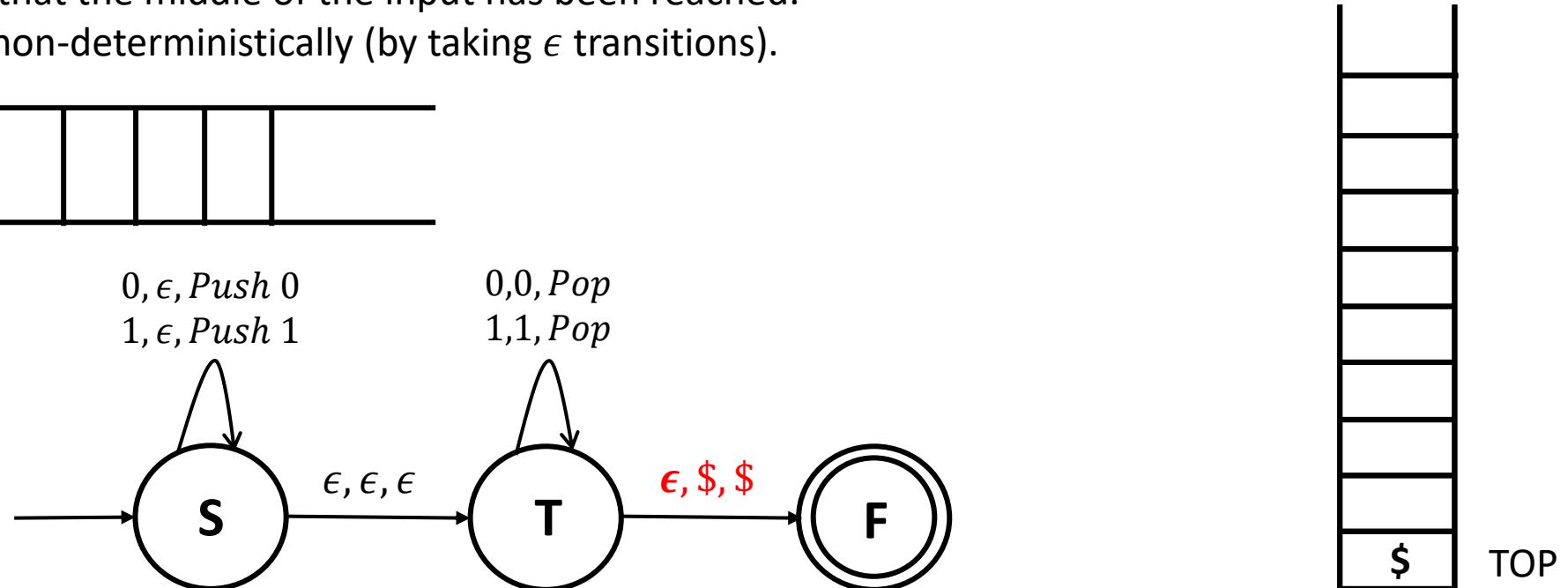
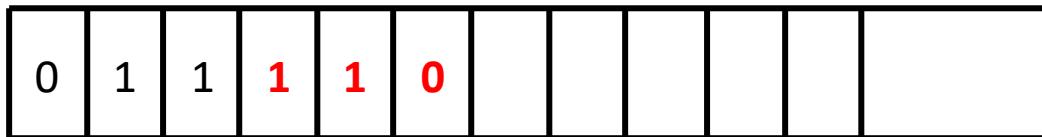


# Pushdown Automata

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).

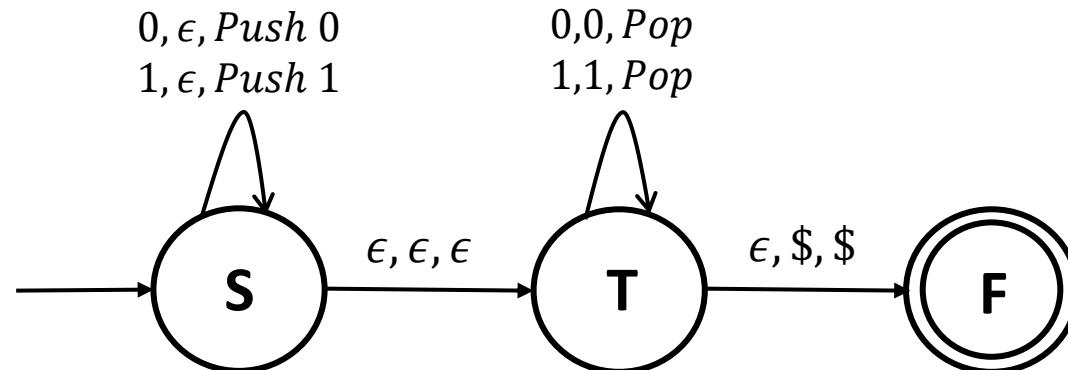


# Pushdown Automata

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:  $ww^R$

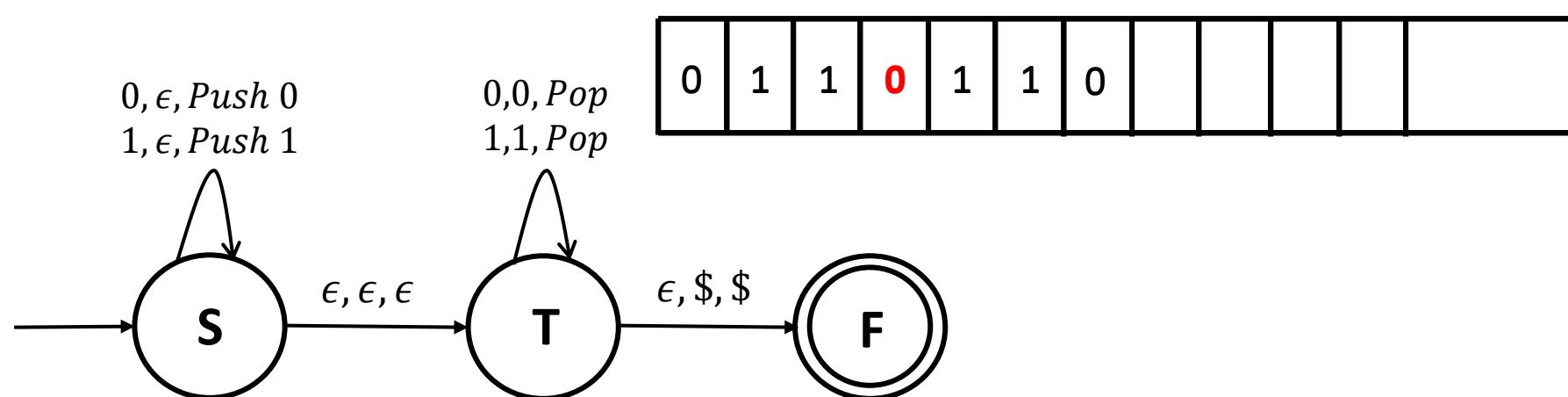
# Pushdown Automata

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  
 $c \in \Sigma$



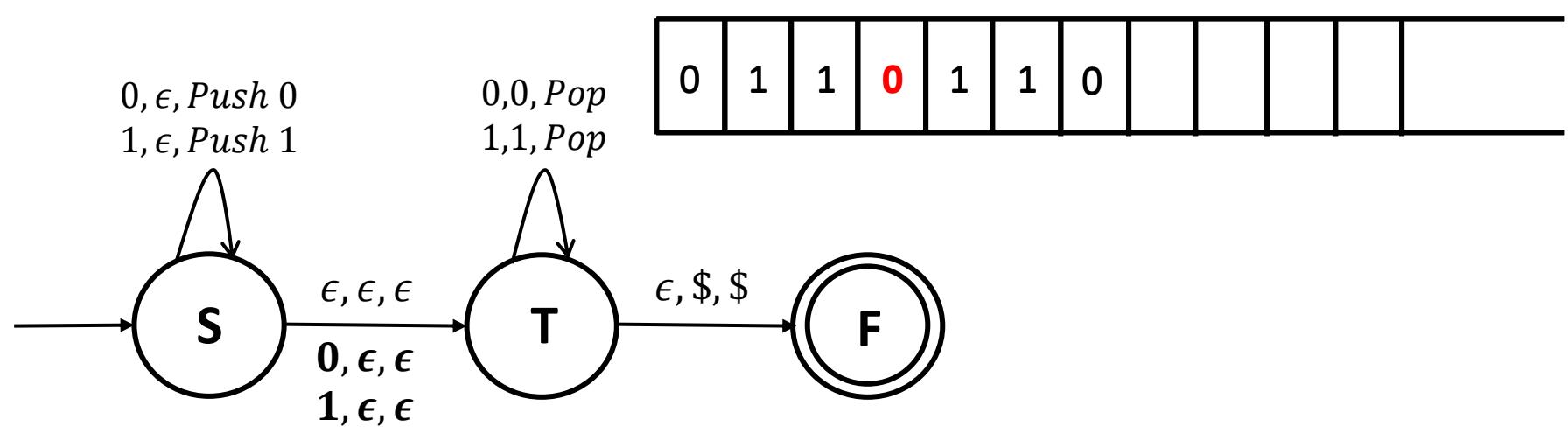
# Pushdown Automata

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  
 $c \in \Sigma$

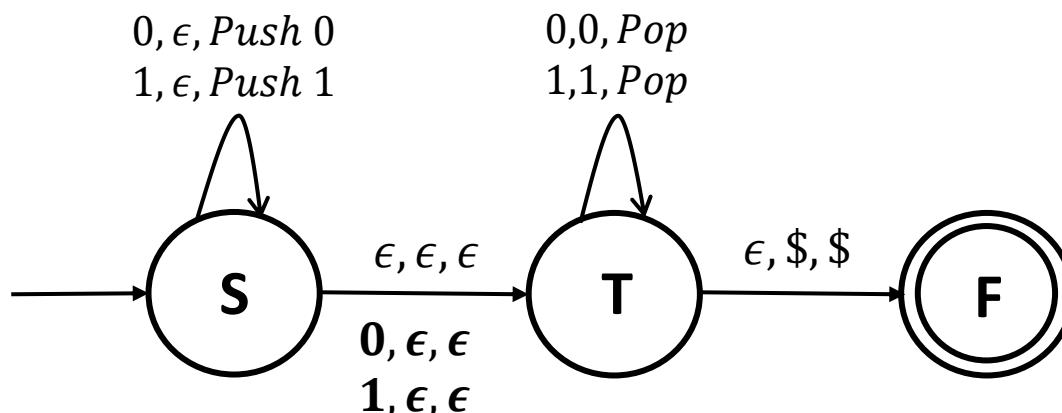


# Pushdown Automata

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?



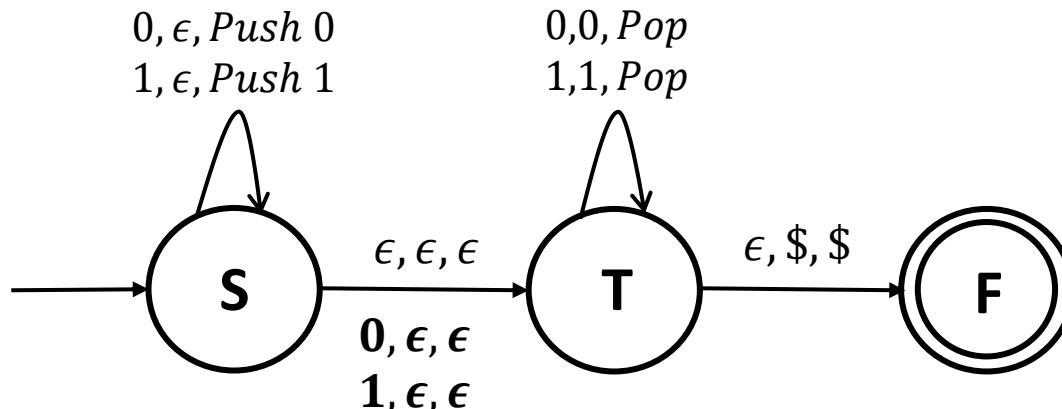
The transitions  **$0, \epsilon, \epsilon$  and  $1, \epsilon, \epsilon$**  allow the PDA to **consume one symbol** and then begin matching what it has encountered thus far.

# 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!**

# CS 302.1 - Automata Theory

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad



INTERNATIONAL INSTITUTE OF  
INFORMATION TECHNOLOGY

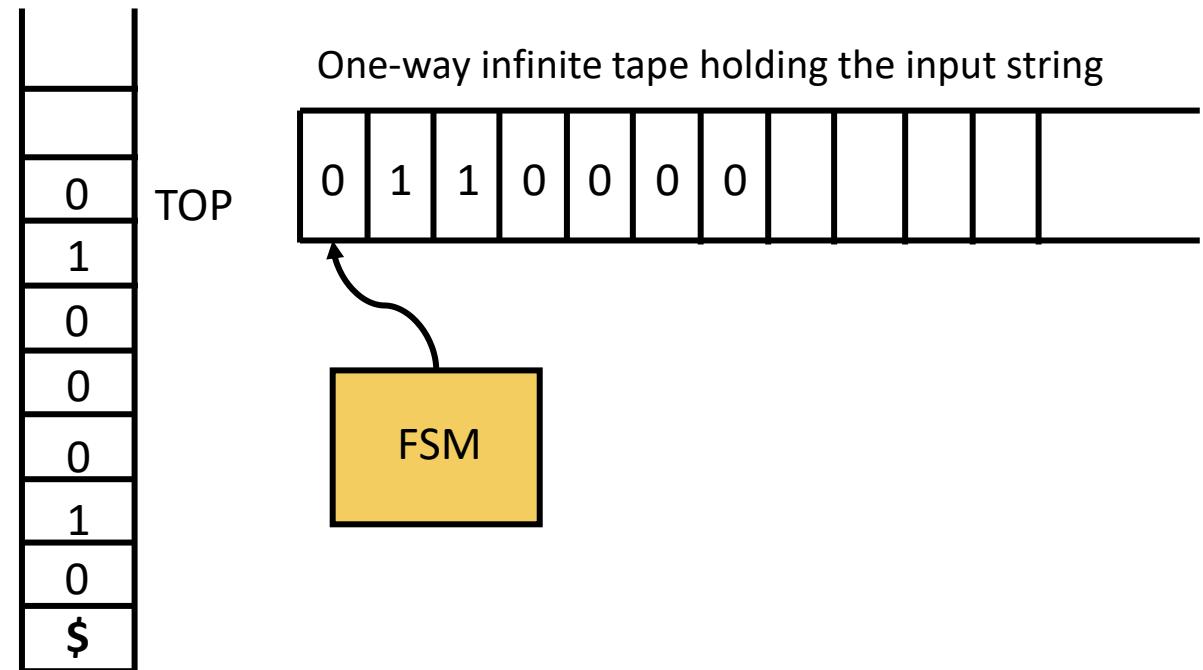
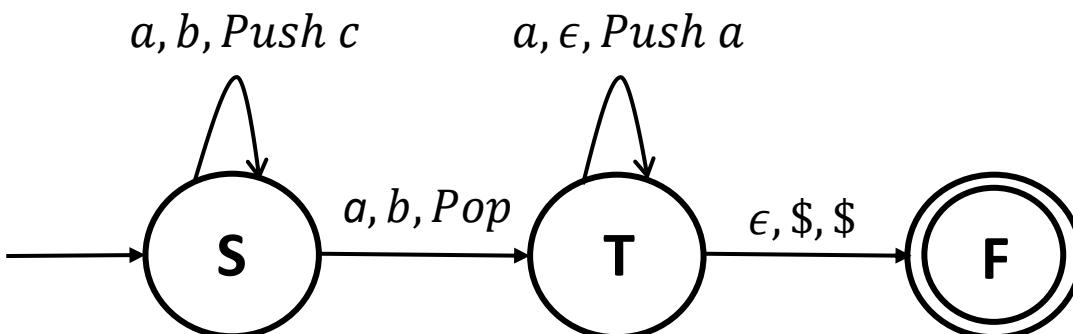
H Y D E R A B A D

# Quick Recap

## Pushdown Automata

- Automata that recognizes CFLs
- FSM + stack
- FSM transitions by reading an input symbol and by interacting with the stack

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and the stack top =  $b$ , then pop  $b$ , 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.

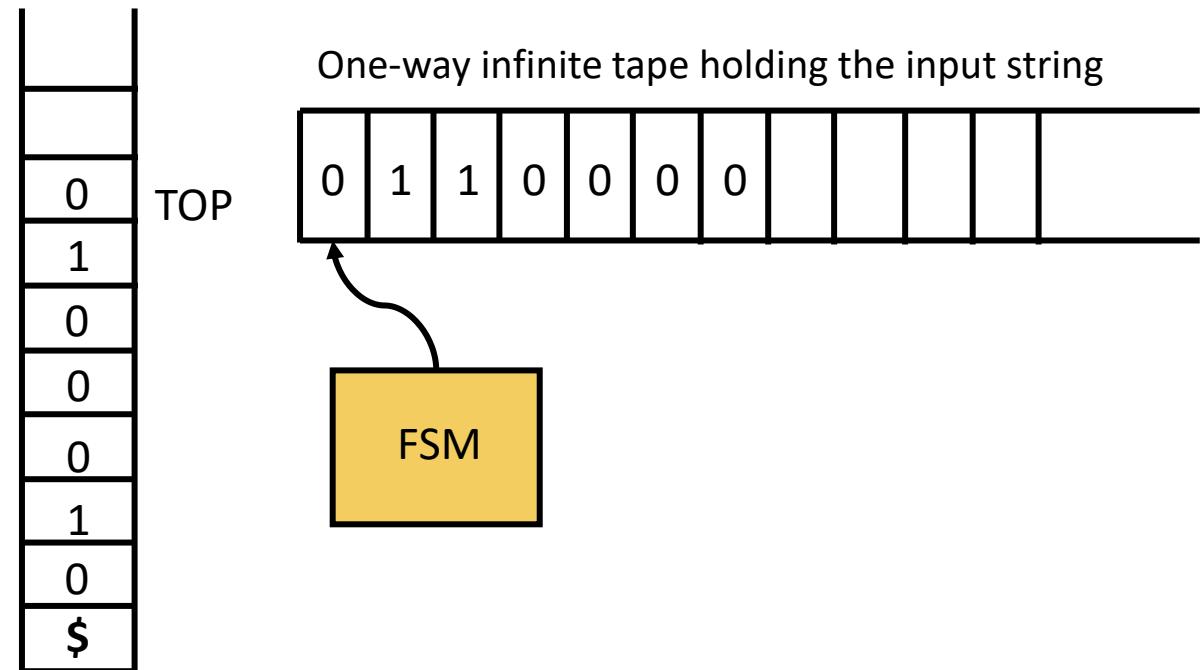
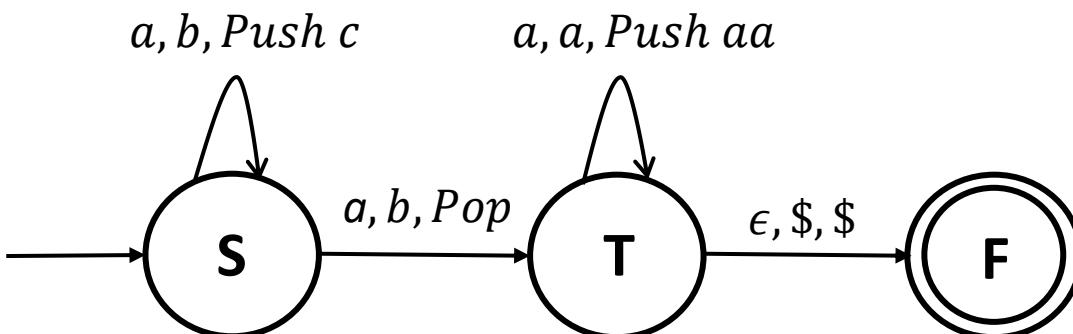


# Quick Recap

## Pushdown Automata

- Automata that recognizes CFLs
- FSM + stack
- FSM transitions by reading an input symbol and by interacting with the stack

- $\delta(q_i, a, b) = (q_j, c)$ : If the input symbol read is  $a$  and the stack top =  $b$ , then pop  $b$ , 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.



# Equivalence between PDA and CFL

- We already know that a language is Context-Free if and only if there exists a CFG that generates all the strings belonging to the CFL.
- It can be shown that a language is context free if and only if a PDA recognizes it.
  - If  $L$  is context free then there exists a PDA that recognizes  $L$ . (We'll prove this next)
  - If there exists a PDA for  $L$ , then  $L$  is context-free. (Won't prove this in class. Look up a standard text book)

# Pushdown Automata and CFL

**Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .**

- Before formally proving this, we will use some examples in order to provide some intuition.
- For any  $L$ , we can write a context free grammar that can generate all strings that are in  $L$ .
- Any string  $w$  is generated by the CFG if there exists a derivation  $S \xrightarrow{*} w$ .

# Pushdown Automata and CFL

**Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .**

- Before formally proving this, we will use some examples in order to provide some intuition.
- For any  $L$ , we can write a context free grammar that can generate all strings that are in  $L$ .
- Any string  $w$  is generated by the CFG if there exists a derivation  $S \xrightarrow{*} w$ .
- The proof consists of using the rules of the CFG to build a PDA so that it can simulate any derivation  $S \xrightarrow{*} w$ .
  - The PDA accepts an input  $w$  if the CFG  $G$  generates  $w$
  - It determines whether  $\exists$  a derivation for  $w$ .
  - Takes advantage of non determinism

# Pushdown Automata and CFL

**Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .**

## Intuitions

- The PDA begins by pushing the start variable  $S$  onto the stack.

# Pushdown Automata and CFL

Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .

## Intuitions

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**

# Pushdown Automata and CFL

Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .

## Intuitions

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

# Pushdown Automata and CFL

**Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .**

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

**Example:** Consider the grammar  $G$  with the rules:  $S \rightarrow aTb|b$   
 $T \rightarrow Ta|\epsilon$

The string  $w = aaab$  can be generated by  $G$ . Derivation:

$$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$$

# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

**Example:**  $S \rightarrow aTb|b$

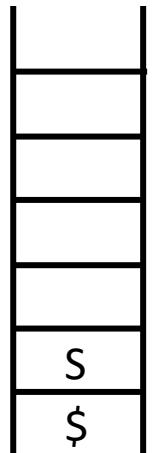
$T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

**Example:**  $S \rightarrow aTb|b$

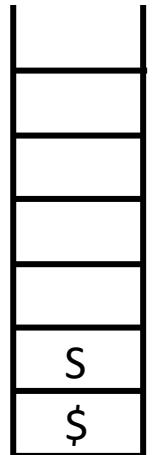
$T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and
  - a. Push b,
  - b. Push T
  - c. Push a



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

**Example:**  $S \rightarrow aTb|b$

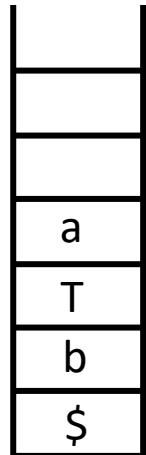
$T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and
  - a. Push b
  - b. Push T
  - c. Push a



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

**Example:**  $S \rightarrow aTb|b$

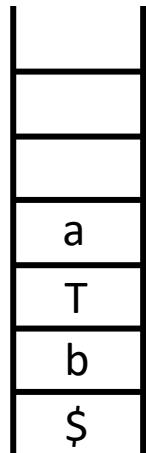
$T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and **push aTb** (Shorthand).
3. As the top of the stack is  $a$ , read the input ( $a$ ). Since they match, pop  $a$ .



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

**Example:**  $S \rightarrow aTb|b$

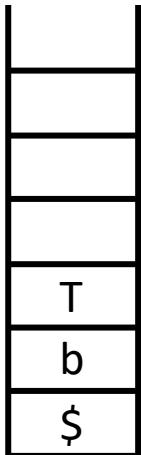
$T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and push  $aTb$  (Shorthand).
3. As the top of the stack is  $a$ , read the input ( $a$ ). Since they match, pop  $a$ .
4. The top of stack is  $T$ , so pop  $T$  and push  $Ta$



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

**Example:**  $S \rightarrow aTb|b$

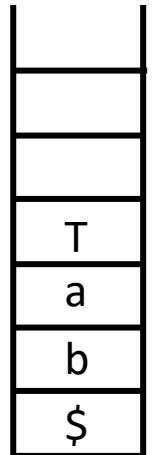
$T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and push  $aTb$  (Shorthand).
3. As the top of the stack is  $a$ , read the input ( $a$ ). Since they match, pop  $a$ .
4. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
5. The top of stack is  $T$ , so pop  $T$  and push  $Ta$



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

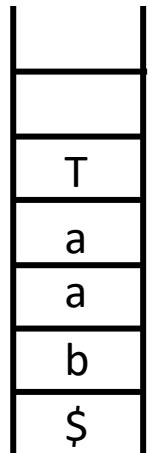
**Example:**  $S \rightarrow aTb|b$

$T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$



1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and push  $aTb$  (Shorthand).
3. As the top of the stack is  $a$ , read the input ( $a$ ). Since they match, pop  $a$ .
4. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
5. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
6. The top of stack is  $T$ , so pop  $T$  (for the rule  $T \rightarrow \epsilon$ )

# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

**Example:**  $S \rightarrow aTb|b$

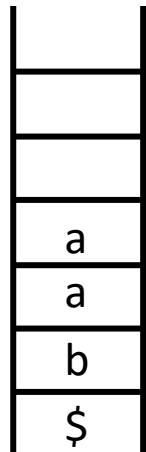
$T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and push  $aTb$  (Shorthand).
3. As the top of the stack is  $a$ , read the input ( $a$ ). Since they match, pop  $a$ .
4. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
5. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
6. The top of stack is  $T$ , so pop  $T$  (for the rule  $T \rightarrow \epsilon$ )
7. The top of the stack is  $a$ , so read the input ( $a$ ). Since they match pop  $a$ .



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

**Example:**  $S \rightarrow aTb|b$

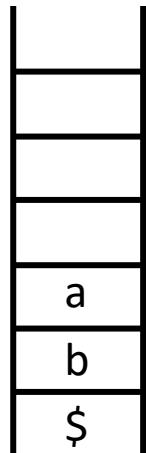
$T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and push  $aTb$  (Shorthand).
3. As the top of the stack is  $a$ , read the input ( $a$ ). Since they match, pop  $a$ .
4. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
5. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
6. The top of stack is  $T$ , so pop  $T$  (for the rule  $T \rightarrow \epsilon$ )
7. The top of the stack is  $a$ , so read the input ( $a$ ). Since they match pop  $a$ .



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

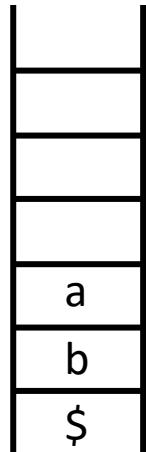
**Example:**  $S \rightarrow aTb|b$   
 $T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and push  $aTb$  (Shorthand).
3. As the top of the stack is  $a$ , read the input ( $a$ ). Since they match, pop  $a$ .
4. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
5. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
6. The top of stack is  $T$ , so pop  $T$  (for the rule  $T \rightarrow \epsilon$ )
7. The top of the stack is  $a$ , so read the input ( $a$ ). Since they match pop  $a$ .
8. The top of the stack is  $a$ , so read the input ( $a$ ). Since they match pop  $a$ .



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

Example:  $S \rightarrow aTb|b$

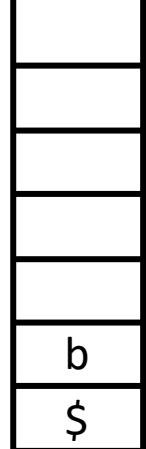
$T \rightarrow Ta|\epsilon$

Input to PDA:  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and push  $aTb$  (Shorthand).
3. As the top of the stack is  $a$ , read the input ( $a$ ). Since they match, pop  $a$ .
4. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
5. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
6. The top of stack is  $T$ , so pop  $T$  (for the rule  $T \rightarrow \epsilon$ )
7. The top of the stack is  $a$ , so read the input ( $a$ ). Since they match pop  $a$ .
8. The top of the stack is  $a$ , so read the input ( $a$ ). Since they match pop  $a$ .
9. The top of the stack is  $b$ , so read the input ( $b$ ). Since they match pop  $b$ .



# Pushdown Automata and CFL

- The PDA begins by pushing the start variable  $S$  onto the stack.
- If the top of the stack any variable  $A$ , then non-deterministically select one of the rules  $A \rightarrow x$  ( $x$  can be a sequence of variables and terminals) pop  $A$  and push  $x$  on to the stack. **[Non deterministically chooses a rule as an intermediate derivation step]**
- If the top of the stack is some terminal  $a$ , **read the input symbol**. If they match, pop  $a$ . **[This tries to match part of the input string  $w$ ]**

Example:  $S \rightarrow aTb|b$

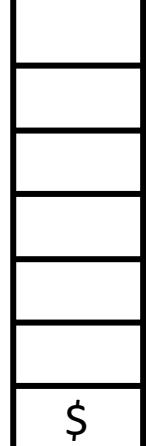
$T \rightarrow Ta|\epsilon$

Input to PDA:  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$

1. Push  $S$  onto the Stack.
2. As top of the stack is  $S$ , pop  $S$  and push  $aTb$  (Shorthand).
3. As the top of the stack is  $a$ , read the input ( $a$ ). Since they match, pop  $a$ .
4. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
5. The top of stack is  $T$ , so pop  $T$  and push  $Ta$
6. The top of stack is  $T$ , so pop  $T$  (for the rule  $T \rightarrow \epsilon$ )
7. The top of the stack is  $a$ , so read the input ( $a$ ). Since they match pop  $a$ .
8. The top of the stack is  $a$ , so read the input ( $a$ ). Since they match pop  $a$ .
9. The top of the stack is  $b$ , so read the input ( $b$ ). Since they match pop  $b$ .
10. Since the stack is empty exactly when the input has been read, accept  $w$ .



# Pushdown Automata and CFL

**Example:**  $S \rightarrow aTb|b$   
 $T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$$

$\epsilon, S, \text{Push } aTb$

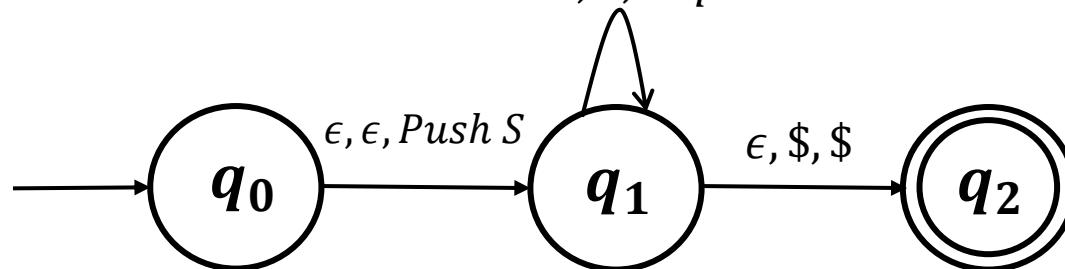
$\epsilon, T, \text{Push } Ta$

$\epsilon, S, \text{Push } b$

$\epsilon, T, \text{Pop}$

$a, a, \text{Pop}$

$b, b, \text{Pop}$



# Pushdown Automata and CFL

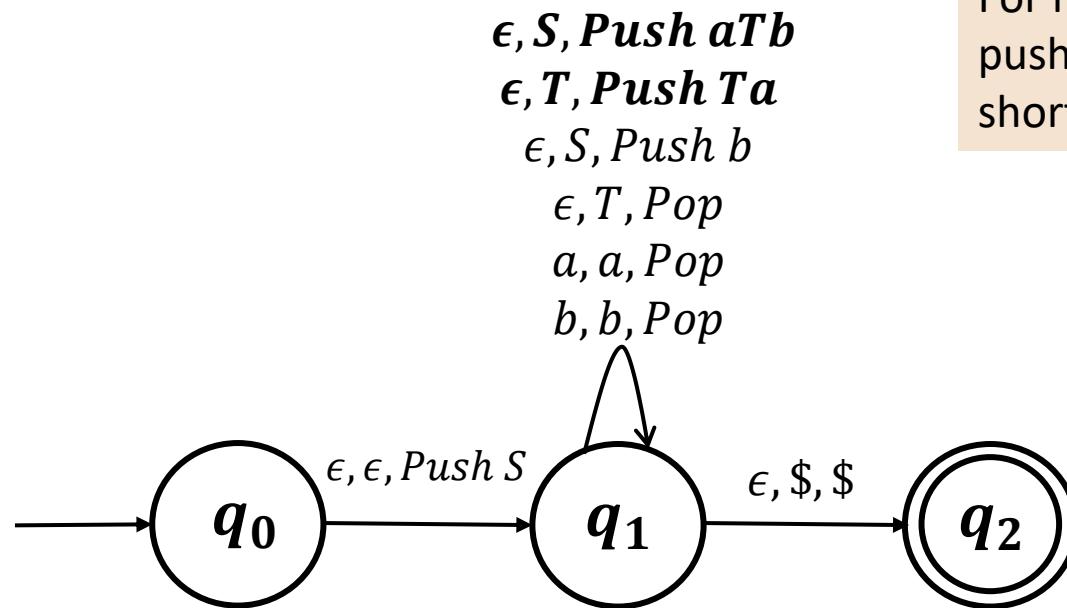
**Example:**  $S \rightarrow aTb|b$   
 $T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = \mathbf{aaab}$  can be generated by  $G$ :

$$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$$

For rules where several elements need to be pushed, new states are introduced. This is only a shorthand for that.



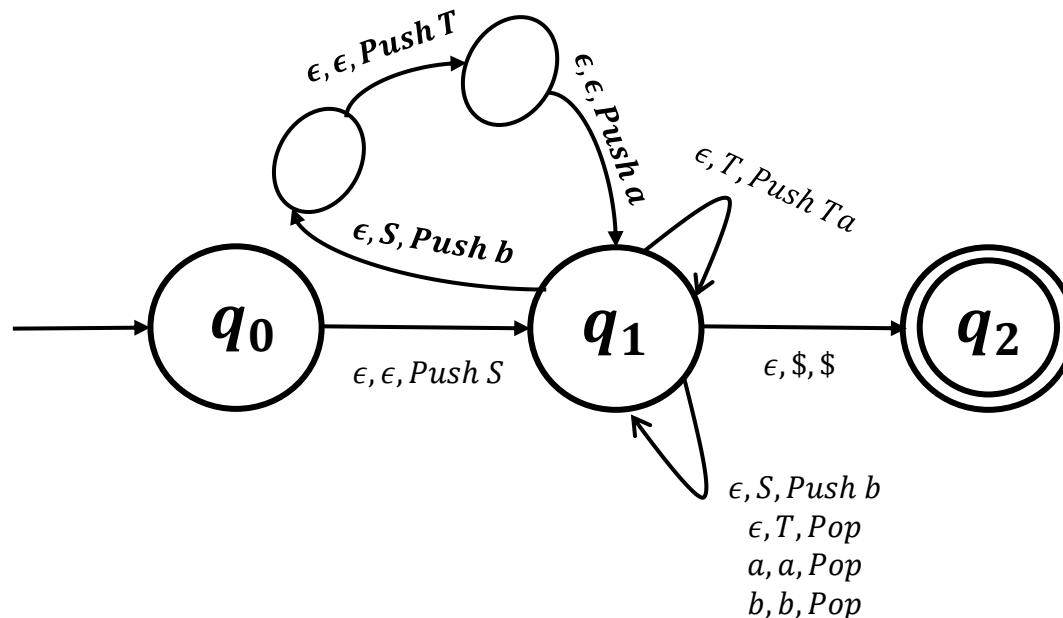
# Pushdown Automata and CFL

**Example:**  $S \rightarrow aTb|b$   
 $T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$$



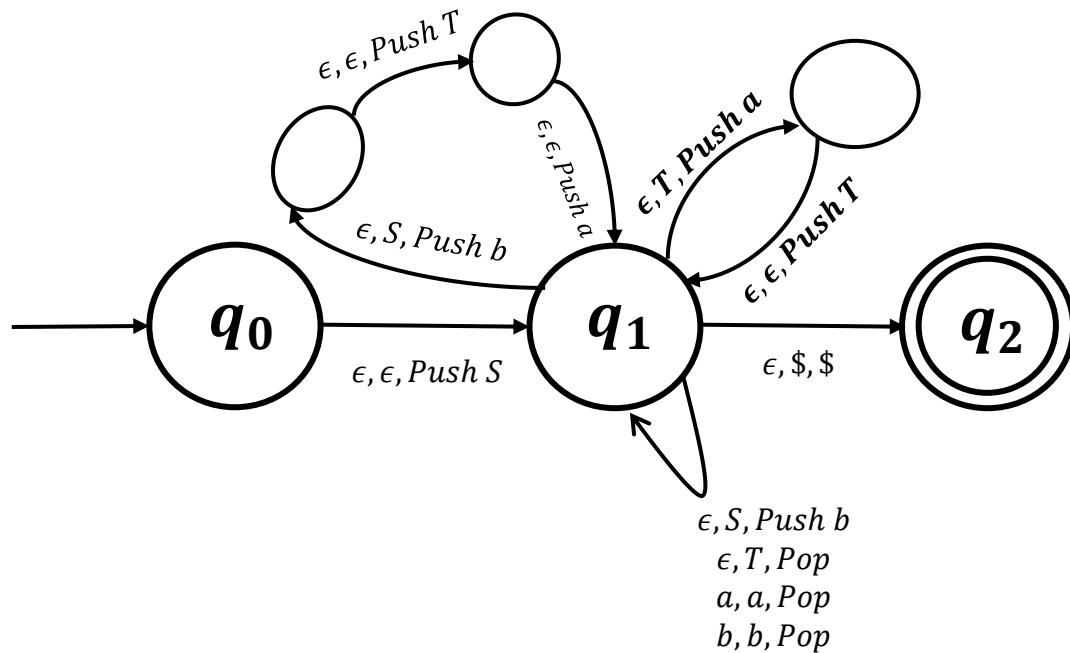
# Pushdown Automata and CFL

**Example:**  $S \rightarrow aTb|b$   
 $T \rightarrow Ta|\epsilon$

**Input to PDA:**  $w = aaab$

Derivation for input string  $w = aaab$  can be generated by  $G$ :

$$S \rightarrow aTb \rightarrow aTab \rightarrow aTaab \rightarrow aaab$$



## Summary

Given the rules of a CFG  $G$ , the equivalent PDA either non deterministically chooses which rule to use or matches part of the input symbol.

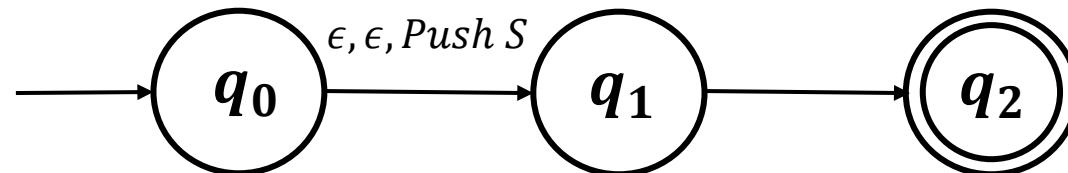
# Pushdown Automata and CFL

**Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .**

**Proof:** For convenience, we shall be using the shorthand notation.

Let  $G$  be a CFG with a set of rules  $R$ , then the equivalent PDA  $P$  will have three states  $\{q_0, q_1, q_2\}$ .

The PDA  $P$  first pushes the start symbol  $S$  into the stack, irrespective of the input symbol and transitions from the initial state  $q_0$  to  $q_1$ , i.e. let  $\delta(q_0, \epsilon, \epsilon) = (q_1, S)$ .



# Pushdown Automata and CFL

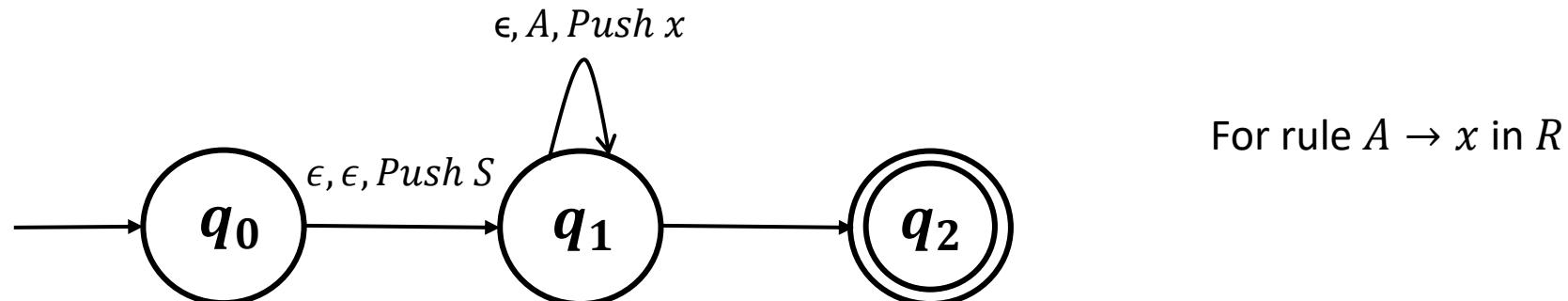
**Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .**

**Proof:** Let  $G$  be a CFG with a set of rules  $R$ , then the equivalent PDA  $P$  will have three states  $\{q_0, q_1, q_2\}$ .

The PDA  $P$  first pushes the start symbol  $S$  into the stack, irrespective of the input symbol and transitions from the initial state  $q_0$  to  $q_1$ , i.e. let  $\delta(q_0, \epsilon, \epsilon) = (q_1, S)$ .

At  $q_1$ , the PDA  $P$  implements the rules  $R$  of  $G$ .

- If the top of the stack is a variable  $A$ , pop  $A$  and push  $x$  onto the stack, where  $A \rightarrow x$  is a rule in  $R$  and return back to  $q_1$ , i.e. let  $\delta(q_1, \epsilon, A) = (q_1, x)$ .



# Pushdown Automata and CFL

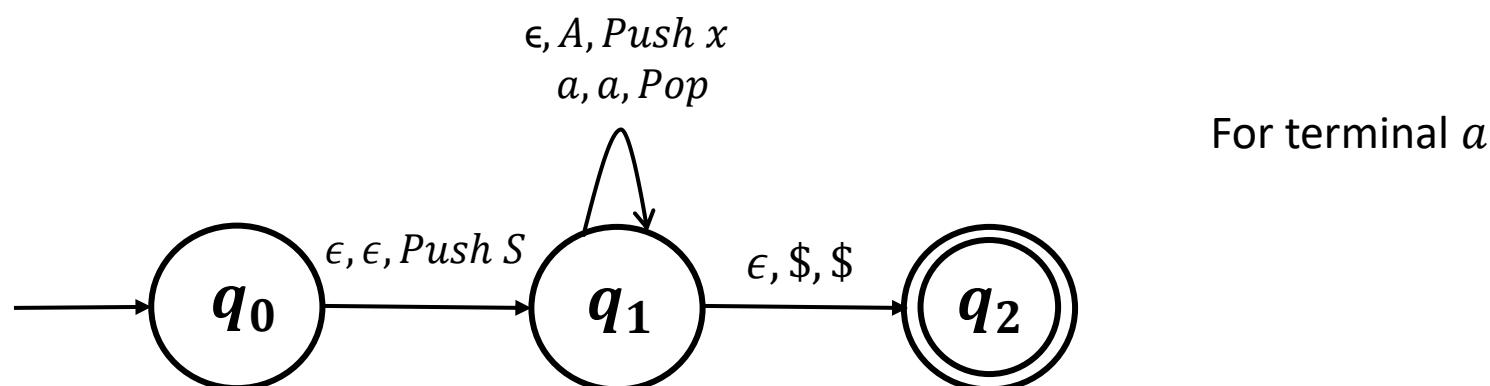
**Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .**

**Proof:** Let  $G$  be a CFG with a set of rules  $R$ , then the equivalent PDA  $P$  will have three states  $\{q_0, q_1, q_2\}$ .

The PDA  $P$  first pushes the start symbol  $S$  into the stack, irrespective of the input symbol and transitions from the initial state  $q_0$  to  $q_1$ , i.e. let  $\delta(q_0, \epsilon, \epsilon) = (q_1, S)$ .

At  $q_1$ , the PDA  $P$  implements the rules  $R$  of  $G$ .

- If the top of the stack is a variable  $A$ , pop  $A$  and push  $x$  onto the stack, where  $A \rightarrow x$  is a rule in  $R$  and return back to  $q_1$ , i.e. let  $\delta(q_1, \epsilon, A) = (q_1, x)$ .
- If the top of the stack is a terminal  $a$ , read the input symbol and if they match, pop  $a$ , i.e. let  $\delta(q_1, a, a) = (q_1, \epsilon)$



# Pushdown Automata and CFL

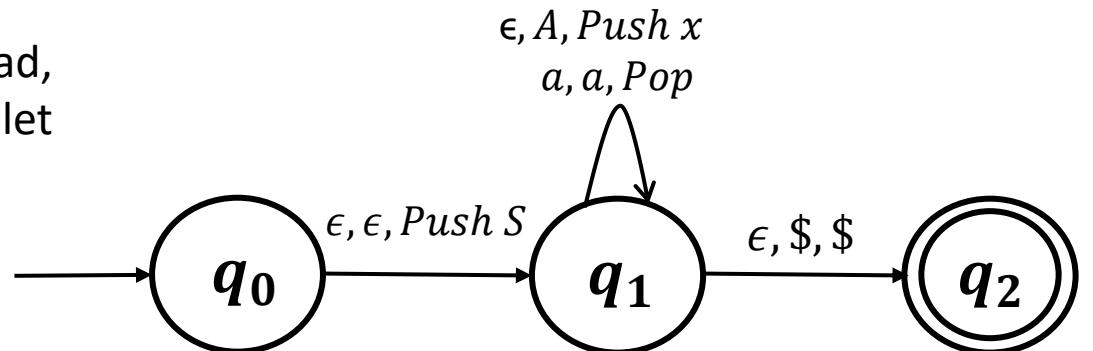
**Prove that if  $L$  is context free then there exists an equivalent PDA that recognizes  $L$ .**

**Proof:** Let  $G$  be a CFG with a set of rules  $R$ , then the equivalent PDA  $P$  will have three states  $\{q_0, q_1, q_2\}$ .

The PDA  $P$  first pushes the start symbol  $S$  into the stack, irrespective of the input symbol and transitions from the initial state  $q_0$  to  $q_1$ , i.e. let  $\delta(q_0, \epsilon, \epsilon) = (q_1, S)$ .

At  $q_1$ , the PDA  $P$  implements the rules  $R$  of  $G$ .

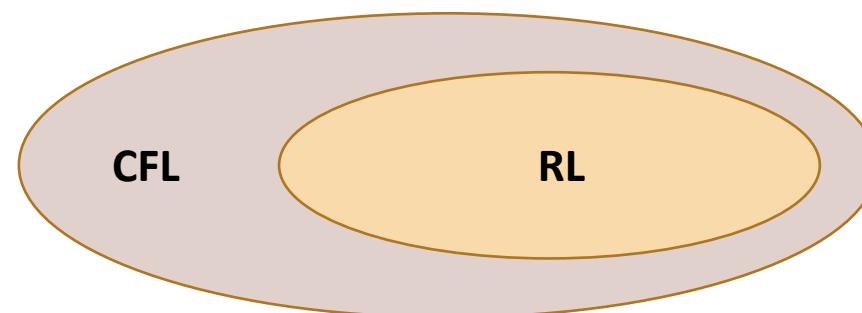
- If the top of the stack is a variable  $A$ , pop  $A$  and push  $x$  onto the stack, where  $A \rightarrow x$  is a rule in  $R$  and return back to  $q_1$ , i.e. let  $\delta(q_1, \epsilon, A) = (q_1, x)$ .
- If the top of the stack is a terminal  $a$ , read the input symbol and if they match, pop  $a$ , i.e. let  $\delta(q_1, a, a) = (q_1, \epsilon)$
- If the stack is empty, when all the input symbols are read, transition from  $q_1$  to the accepting state  $q_2$ , i.e. let  $\delta(q_1, \epsilon, \$) = (q_2, \$)$



# Equivalence between PDA and CFL

- It can be shown that a language is context free **if and only if** a PDA recognizes it.
  - If  $L$  is context free then there exists a PDA that recognizes  $L$ . (We proved this)
  - The proof for the other direction (Constructing a CFG that generates  $L$  given a PDA that recognizes  $L$ ) is quite elaborate
  - We won't be covering it in class. But the proof itself is quite easy to understand.
  - Refer to a standard text book (e.g. Sipser)

$(RL \equiv \text{Regular Grammar} \equiv \text{Regular Expressions} \equiv NFA \equiv DFA) \subseteq (CFL \equiv CFG \equiv PDA)$



# Deterministic Pushdown Automata

- So far we have considered Non-deterministic PDAs (which are referred to as just PDAs)
- Multiple transitions per input symbol/stack symbol is allowed
- Recall that for regular languages, introducing non-determinism added no extra power to finite automata: NFAs and DFAs were equivalent
- What about PDAs and CFLs?

# Deterministic Pushdown Automata

- So far we have considered Non-deterministic PDAs (which are referred to as just PDAs)
- Multiple transitions per input symbol/stack symbol is allowed
- Recall that for regular languages, introducing non-determinism added no extra power to finite automata: NFAs and DFAs were equivalent
- What about PDAs and CFLs?

## Deterministic Pushdown Automata (DPDA)

DPDAs can be defined in a similar manner to PDAs with the following restriction:

- For each state in the PDA, for any combination of the current input symbol and the current stack symbol **at most one transition is possible.** (At most one valid transition is allowed for any input)

# Deterministic Pushdown Automata

- So far we have considered Non-deterministic PDAs (which are referred to as just PDAs)
- Multiple transitions per input symbol/stack symbol is allowed
- Recall that for regular languages, introducing non-determinism added no extra power to finite automata: NFAs and DFAs were equivalent
- What about PDAs and CFLs?

## Deterministic Pushdown Automata (DPDA)

DPDAs can be defined in a similar manner to PDAs with the following restriction:

- For each state in the PDA, for any combination of the current input symbol and the current stack symbol **at most one transition is possible.** (At most one valid transition is allowed for any input)
- So determinism is defined slightly differently here. For example:  $\epsilon$  transitions are allowed, some transitions may be absent (leading to CRASH runs) unlike in the case of DFAs.

# Deterministic Pushdown Automata

DPDAs can be defined in a similar manner to PDAs with the following restriction:

- For each state in the PDA, for any combination of the current input symbol and the current stack symbol **at most one transition is possible**. (At most one valid transition is allowed for any input)
- So determinism is defined slightly differently here. For example:  $\epsilon$ -transitions are allowed, some transitions may be absent (leading to CRASH runs) unlike in the case of DFAs.
- In fact, for a DPDA, the following condition must be satisfied:
  - For every  $q \in Q, a \in \Sigma_\epsilon$  and  $x \in \Gamma_\epsilon$ , the set  $\delta(q, a, x)$  **has at most one element**

# Deterministic Pushdown Automata

DPDAs can be defined in a similar manner to PDAs with the following restriction:

- For each state in the PDA, for any combination of the current input symbol and the current stack symbol **at most one transition is possible**. (At most one valid transition is allowed for any input)
- So determinism is defined slightly differently here. For example:  $\epsilon$ -transitions are allowed, some transitions may be absent (leading to CRASH runs) unlike in the case of DFAs.
- In fact, for a DPDA, the following condition must be satisfied:
  - For every  $q \in Q, a \in \Sigma_\epsilon$  and  $x \in \Gamma_\epsilon$ , the set  $\delta(q, a, x)$  **has at most one element**

This implies that if  $\delta(q, \epsilon, x) \neq \Phi$ , then  $\delta(q, a, x) = \Phi$  for any  $a \in \Sigma$ : If there is an  $\epsilon$ -transition for some configuration, no other input consuming move is possible.

# Deterministic Pushdown Automata

DPDAs can be defined in a similar manner to PDAs with the following restriction:

- For each state in the PDA, for any combination of the current input symbol and the current stack symbol **at most one transition is possible**. (At most one valid transition is allowed for any input)
- So determinism is defined slightly differently here. For example:  $\epsilon$  transitions are allowed, some transitions may be absent (leading to CRASH runs) unlike in the case of DFAs.
- In fact, for a DPDA, the transition rule  $\delta$  must satisfy the following condition:

For every  $q \in Q, a \in \Sigma$  and  $x \in \Gamma$ , **exactly one** of the following values is non-empty

$$\delta(q, a, x), \delta(q, a, \epsilon), \delta(q, \epsilon, x) \text{ and } \delta(q, \epsilon, \epsilon)$$

# Deterministic Pushdown Automata

DPDAs can be defined in a similar manner to PDAs with the following restriction:

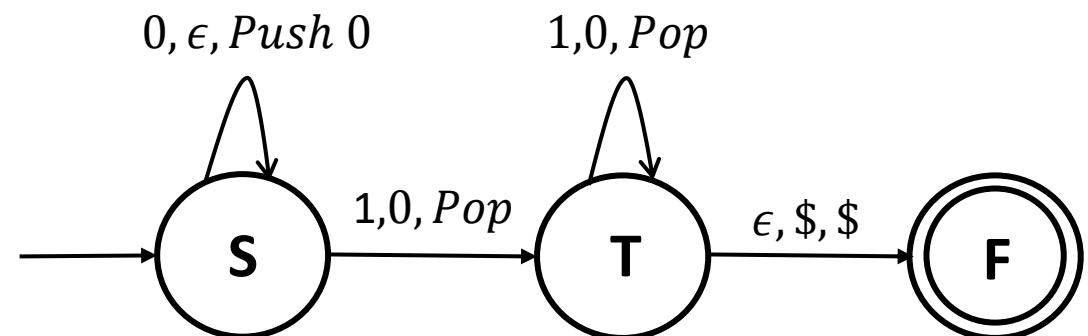
- For each state in the PDA, for any combination of the current input symbol and the current stack symbol **at most one transition is possible**. (At most one valid transition is allowed for any input)
- So determinism is defined slightly differently here. For example:  $\epsilon$  transitions are allowed, some transitions may be absent (leading to CRASH runs) unlike in the case of DFAs.
- In fact, for a DPDA, the transition rule  $\delta$  must satisfy the following condition:

For every  $q \in Q$ ,  $a \in \Sigma$  and  $x \in \Gamma$ , **exactly one** of the following values is non-empty

$$\delta(q, a, x), \delta(q, a, \epsilon), \delta(q, \epsilon, x) \text{ and } \delta(q, \epsilon, \epsilon)$$

Is this a DPDA?

YES!



# Deterministic Pushdown Automata

DPDAs can be defined in a similar manner to PDAs with the following restriction:

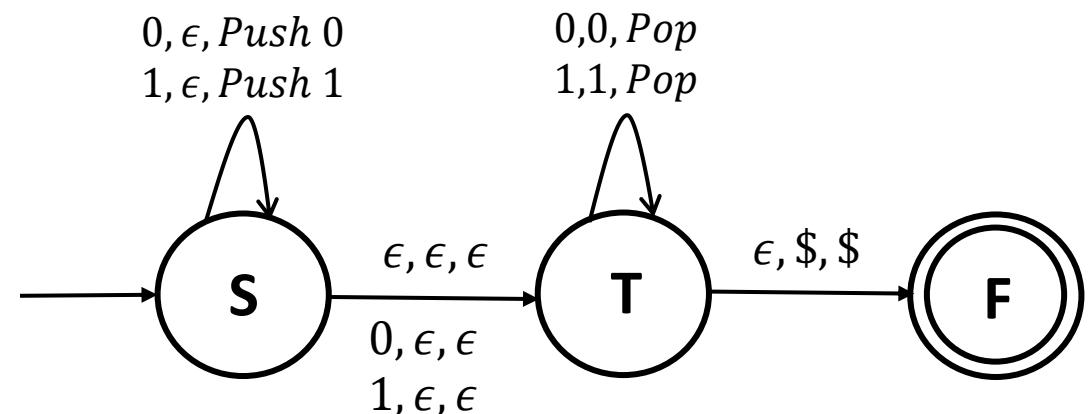
- For each state in the PDA, for any combination of the current input symbol and the current stack symbol **at most one transition is possible**. (At most one valid transition is allowed for any input)
- So determinism is defined slightly differently here. For example:  $\epsilon$  transitions are allowed, some transitions may be absent (leading to CRASH runs) unlike in the case of DFAs.
- In fact, for a DPDA, the transition rule  $\delta$  must satisfy the following condition:

For every  $q \in Q$ ,  $a \in \Sigma$  and  $x \in \Gamma$ , **exactly one** of the following values is non-empty

$$\delta(q, a, x), \delta(q, a, \epsilon), \delta(q, \epsilon, x) \text{ and } \delta(q, \epsilon, \epsilon)$$

Is this a DPDA?

NO!



# Deterministic Pushdown Automata

DPDAs can be defined in a similar manner to PDAs with the following restriction:

- For each state in the PDA, for any combination of the current input symbol and the current stack symbol **at most one transition is possible**. (At most one valid transition is allowed for any input)
- So determinism is defined slightly differently here. For example:  $\epsilon$  transitions are allowed, some transitions may be absent (leading to CRASH runs) unlike in the case of DFAs.
- In fact, for a DPDA, the transition rule  $\delta$  must satisfy the following condition:

For every  $q \in Q$ ,  $a \in \Sigma$  and  $x \in \Gamma$ , **exactly one** of the following values is non-empty

$$\delta(q, a, x), \delta(q, a, \epsilon), \delta(q, \epsilon, x) \text{ and } \delta(q, \epsilon, \epsilon)$$

- Interestingly, the power of DPDAs and PDAs are not the same. It turns out that PDAs are more powerful.
- The language recognized by DPDAs known as **Deterministic CFLs (DCFLs)** are a proper subset of CFLs, i.e.

$$DCFL \subseteq CFL$$

# Deterministic Pushdown Automata

- For a DPDA, the transition rule  $\delta$  must satisfy the following condition:

For every  $q \in Q, a \in \Sigma$  and  $x \in \Gamma$ , **exactly one** of the following values is non-empty  
 $\delta(q, a, x), \delta(q, a, \epsilon), \delta(q, \epsilon, x)$  and  $\delta(q, \epsilon, \epsilon)$

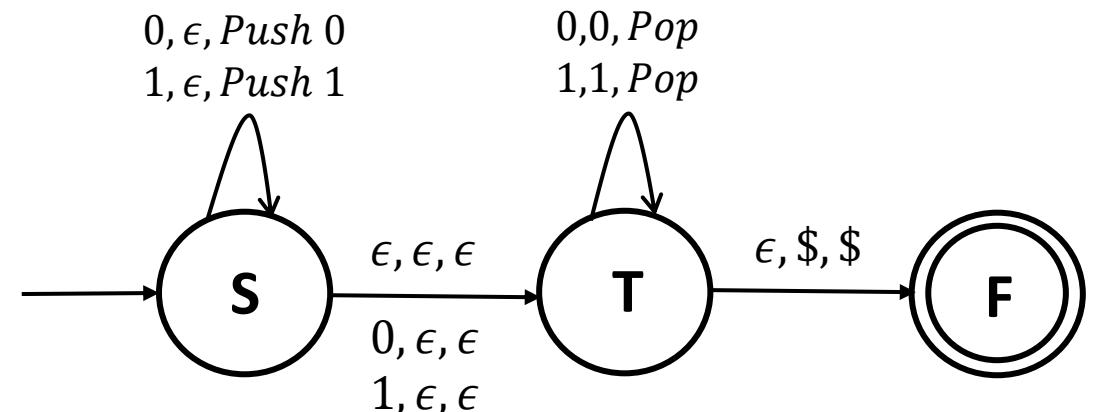
- Interestingly, the power of DPDAs and PDAs are not the same. It turns out that PDAs are more powerful.
- The language recognized by DPDAs, known as **Deterministic CFLs (DCFLs)** are a **proper subset of CFLs**, i.e.

$$DCFL \subseteq CFL$$

**Example:**  $L = \{w \mid w \text{ is a Palindrome}\}$

The PDA had to non deterministically guess when half the string has been read and make a transition.

So although  $L \in CFL, L \notin DCFL$ .

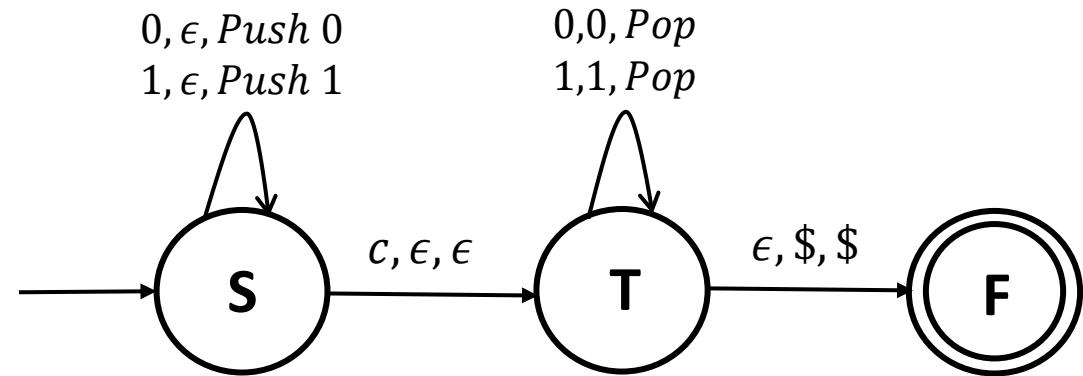


# Deterministic Pushdown Automata

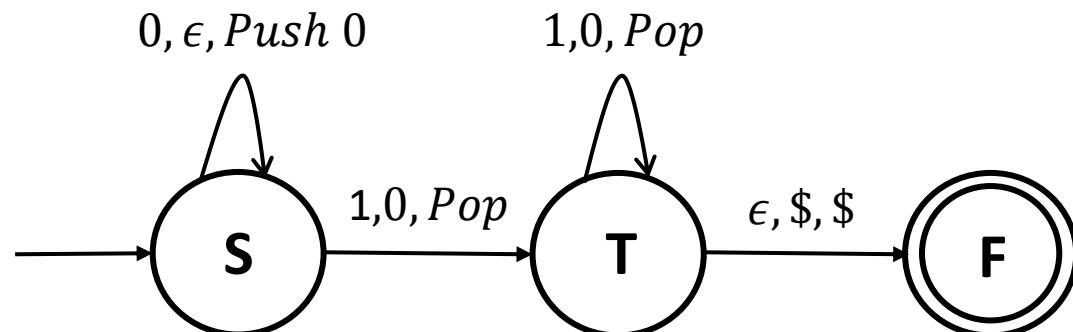
- The language recognized by DPDA known as **Deterministic CFLs (DCFLs)** are a proper subset of CFLs, i.e.

$$DCFL \subseteq CFL$$

- $\Sigma = \{0, 1, c\}$
- $L_1 = \{wcw^R \mid w \in \{0, 1\}^+\}$
- $L_1 \in DCFL$ .



- $L_2 = \{0^n 1^n, n \geq 1\}$
- $L_2 \in DCFL$ .



# Deterministic Pushdown Automata

- The language recognized by DPDAs known as **Deterministic CFLs (DCFLs)** are a proper subset of CFLs, i.e.

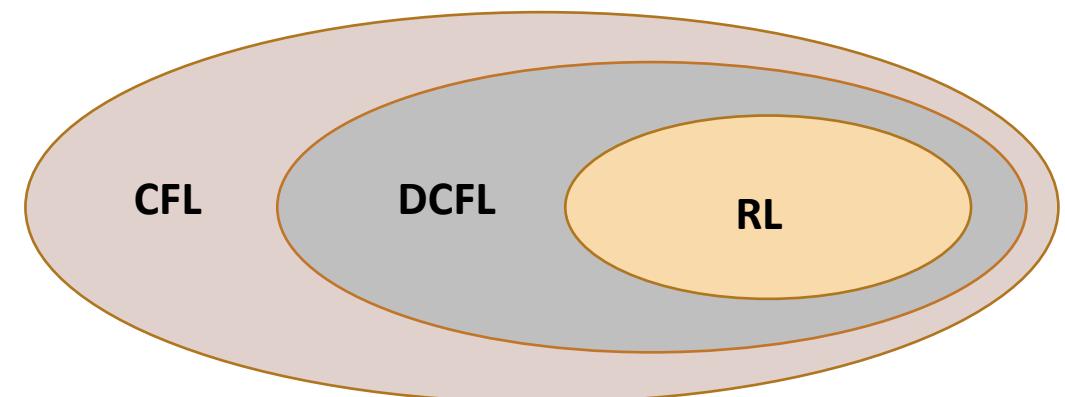
$$DCFL \subseteq CFL$$

*(RL  $\equiv$  Regular Grammar  $\equiv$  Regular Expressions  $\equiv$  NFA  $\equiv$  DFA) \subseteq (DCFL \equiv DPDA) \subseteq (CFL \equiv CFG \equiv PDA)*

Parsers corresponding to DPDAs are efficient!

**Next lecture:**

- Pumping lemma for CFLs
- Closure properties of CFLs

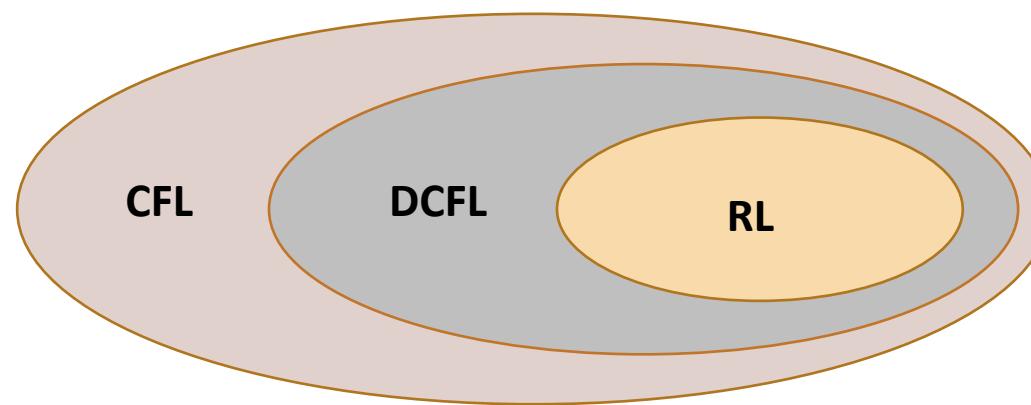


# Deterministic Pushdown Automata

- The language recognized by DPDAs known as **Deterministic CFLs (DCFLs)** are a proper subset of CFLs, i.e.

$$DCFL \subseteq CFL$$

*(RL  $\equiv$  Regular Grammar  $\equiv$  Regular Expressions  $\equiv$  NFA  $\equiv$  DFA) \subseteq (DCFL \equiv DPDA) \subseteq (CFL \equiv CFG \equiv PDA)*



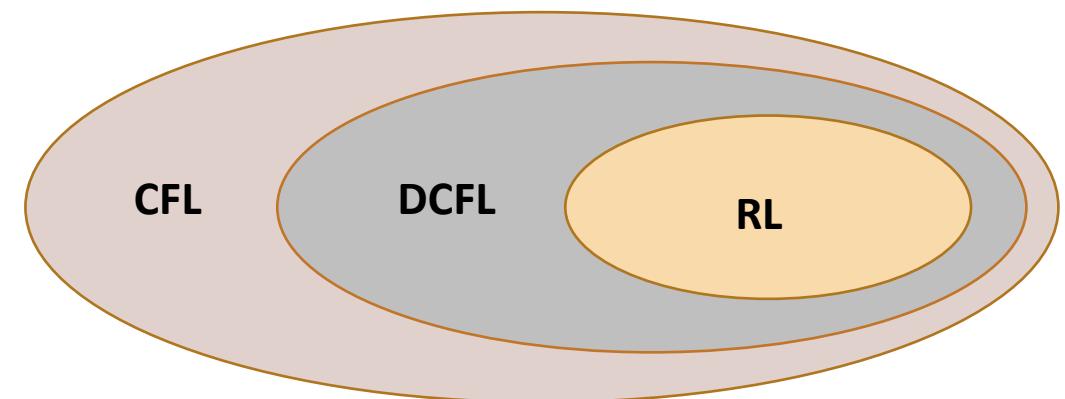
# Deterministic Pushdown Automata

- The language recognized by DPDAs known as **Deterministic CFLs (DCFLs)** are a proper subset of CFLs, i.e.

$$DCFL \subseteq CFL$$

*(RL  $\equiv$  Regular Grammar  $\equiv$  Regular Expressions  $\equiv$  NFA  $\equiv$  DFA) \subseteq (DCFL \equiv DPDA) \subseteq (CFL \equiv CFG \equiv PDA)*

Parsers corresponding to DPDAs are efficient!



**Thank You!**

# CS 302.1 - Automata Theory

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

Pushdown Automata and CFLs are equivalent

CFLs  $\Rightarrow$  Pushdown Automata

## Deterministic Pushdown Automata (DPDA)

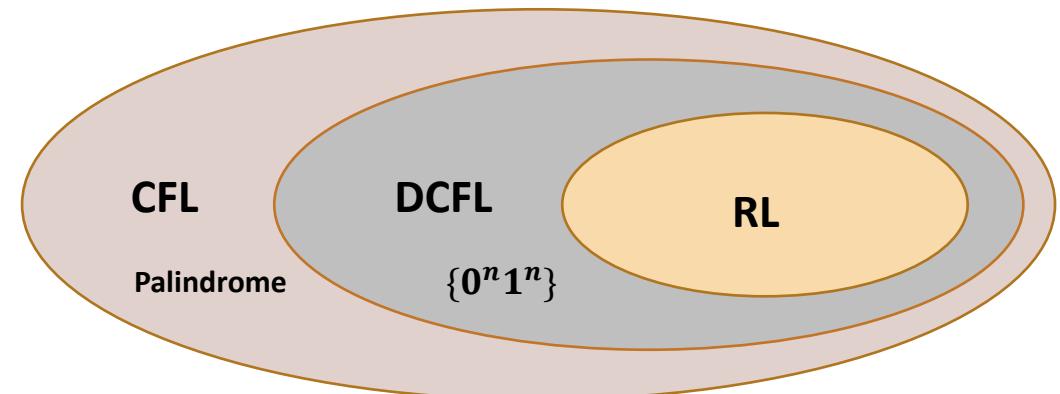
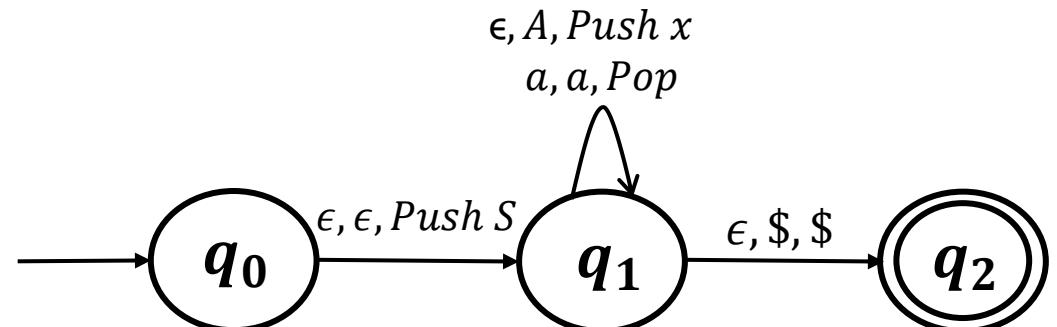
For every  $q \in Q, a \in \Sigma$  and  $x \in \Gamma$ , **exactly one** of the following values is non-empty

$\delta(q, a, x), \delta(q, a, \epsilon), \delta(q, \epsilon, x)$  and  $\delta(q, \epsilon, \epsilon)$

- PDAs are more powerful than DPDAs
- Language recognized by DPDA : DCFL

$L = \{w | w \text{ is a Palindrome}\}$   $L \subseteq \text{CFL}$  but not  $\text{DCFL}$

- $\text{DCFL} \subseteq \text{CFL}$



$(\text{RL} \equiv \text{Regular Grammar} \equiv \text{Regular Expressions} \equiv \text{NFA} \equiv \text{DFA}) \subseteq (\text{DCFL} \equiv \text{DPDA}) \subseteq (\text{CFL} \equiv \text{CFG} \equiv \text{PDA})$

# Pumping Lemma for CFLs

Recall that so far, we have seen the following:

- $L$  is a context-free language.
- $L$  is generated by a Context Free Grammar (CFG) from which any  $w \in L$  can be **derived**.
- The derivation of any CFG can be represented by **parse trees**.
- Any CFG can be expressed in Chomsky Normal Form (CNF): the number of steps required to derive any  $w \in L$ :  $2|w| - 1$
- There exists a Pushdown Automata  $P$  such that  $\mathcal{L}(P) = L$ .

# Pumping Lemma for CFLs

Recall that so far, we have seen the following:

- $L$  is a context-free language.
  - $L$  is generated by a Context Free Grammar (CFG) from which any  $w \in L$  can be **derived**.
  - The derivation of any CFG can be represented by **parse trees**.
  - Any CFG can be expressed in Chomsky Normal Form (CNF): the number of steps required to derive any  $w \in L$ :  $2|w| - 1$
  - There exists a Pushdown Automata  $P$  such that  $\mathcal{L}(P) = L$ .
- 
- Not all languages are context free.
  - Just like in the case of Regular languages, the pumping lemma helps us identify non-CFLs.
  - **All CFLs satisfy the conditions of the pumping lemma:** If any language  $L$  fails to do so, it is not Context-Free.
  - The principle of the Pumping Lemma for CFLs is similar to that of Regular Languages

# Pumping Lemma for CFLs

Recall that so far, we have seen the following:

- $L$  is a context-free language.
  - $L$  is generated by a Context Free Grammar (CFG) from which any  $w \in L$  can be **derived**.
  - The derivation of any CFG can be represented by **parse trees**.
  - Any CFG can be expressed in Chomsky Normal Form (CNF): the number of steps required to derive any  $w \in L$ :  $2|w| - 1$
  - There exists a Pushdown Automata  $P$  such that  $\mathcal{L}(P) = L$ .
- 
- Not all languages are context free.
  - Just like in the case of Regular languages, the pumping lemma helps us identify non-CFLs.
  - **All CFLs satisfy the conditions of the pumping lemma:** If any language  $L$  fails to do so, it is not Context-Free.
  - The principle of the Pumping Lemma for CFLs is similar to that of Regular Languages
  - In order to recognize very long strings in a given CFL  $L$ , the **model of computation (CFGs/parse-trees) must repeat some steps of the computation**

# Pumping Lemma for CFLs

Recall that so far, we have seen the following:

- $L$  is a context-free language.
  - $L$  is generated by a Context Free Grammar (CFG) from which any  $w \in L$  can be **derived**.
  - The derivation of any CFG can be represented by **parse trees**.
  - Any CFG can be expressed in Chomsky Normal Form (CNF): the number of steps required to derive any  $w \in L$ :  $2|w| - 1$
  - There exists a Pushdown Automata  $P$  such that  $\mathcal{L}(P) = L$ .
- 
- Not all languages are context free.
  - Just like in the case of Regular languages, the pumping lemma helps us identify non-CFLs.
  - **All CFLs satisfy the conditions of the pumping lemma:** If any language  $L$  fails to do so, it is not Context-Free.
  - The principle of the Pumping Lemma for CFLs is similar to that of Regular Languages
  - In order to recognize very long strings in a given CFL  $L$ , the **model of computation (CFGs/parse-trees) must repeat some steps of the computation**
  - These steps can be **repeated any number of times (pumped)** to produce longer and longer strings all of which **belong to  $L$** .
  - Conversely if  $L$  is not CF, **there exists a pumping length ( $p$ ) such that some string  $s$  (with  $|s| \geq p$ )  $\notin L$** .

# Pumping Lemma for CFLs

**Example:**

$$A \rightarrow BC|0$$

$$B \rightarrow BA|1|CC$$

$$C \rightarrow AB|0$$

No of variables  $|V| = 3$ .

Consider a derivation of  $w = 11100001$

# Pumping Lemma for CFLs

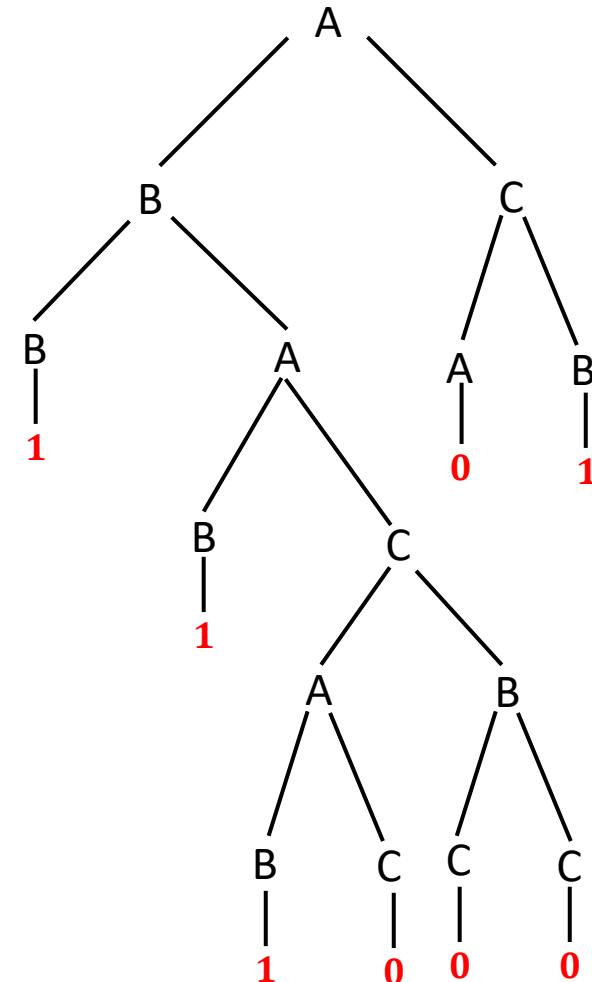
Example:

$$\begin{aligned}A &\rightarrow BC|0 \\B &\rightarrow BA|1|CC \\C &\rightarrow AB|0\end{aligned}$$

No of variables  $|V| = 3$ .

Consider a derivation of  $w = 11100001$

Consider the longest path in the parse tree.

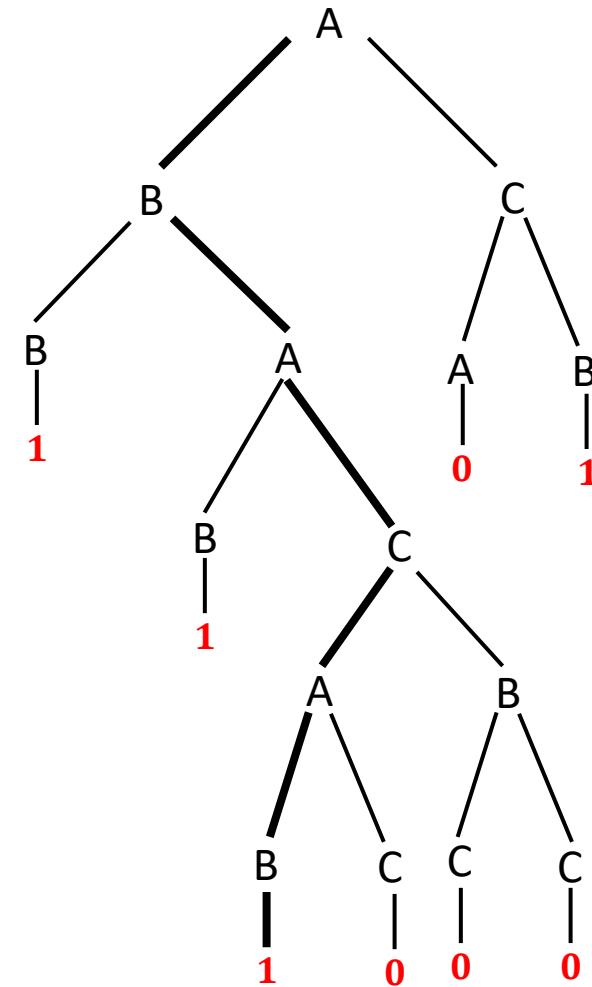


# Pumping Lemma for CFLs

**Example:**

$$\begin{aligned}A &\rightarrow BC|0 \\B &\rightarrow BA|1|CC \\C &\rightarrow AB|0\end{aligned}$$

- No of variables  $|V| = 3$
- Consider a derivation of  $w = 11100001$
- Consider the longest path in the parse tree.
- Longest path length = 6, which is larger than  $|V|$ .
- There exists at least one variable that is repeated.
- For example:  $A$  – mark it.

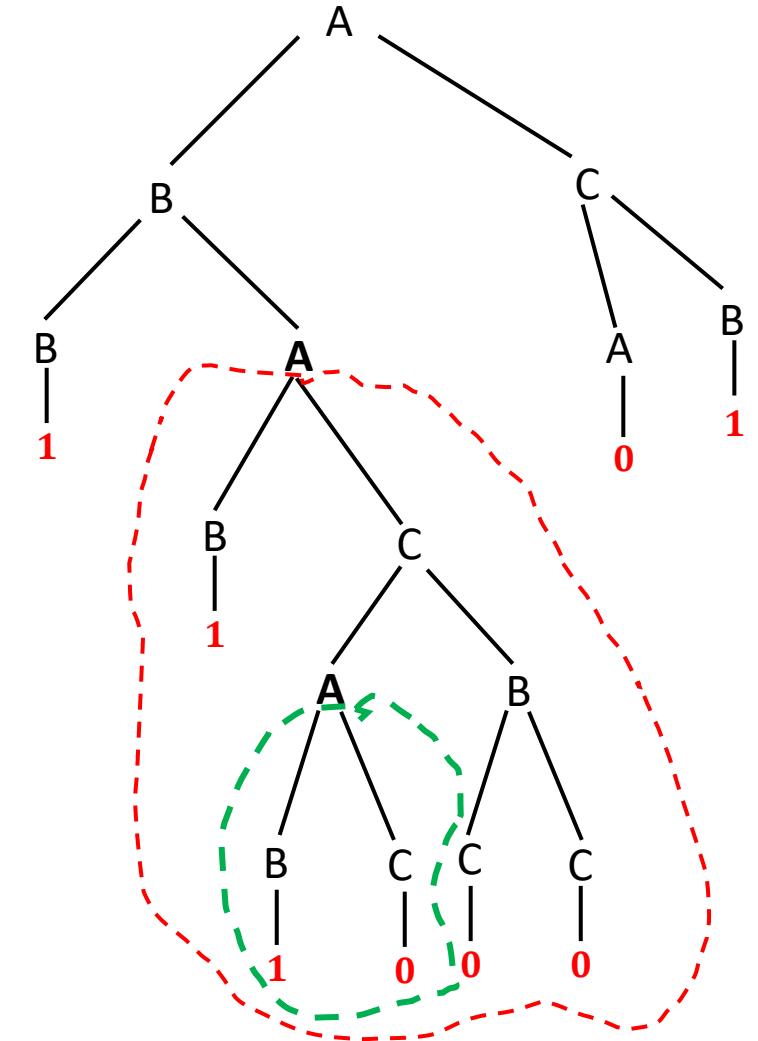


# Pumping Lemma for CFLs

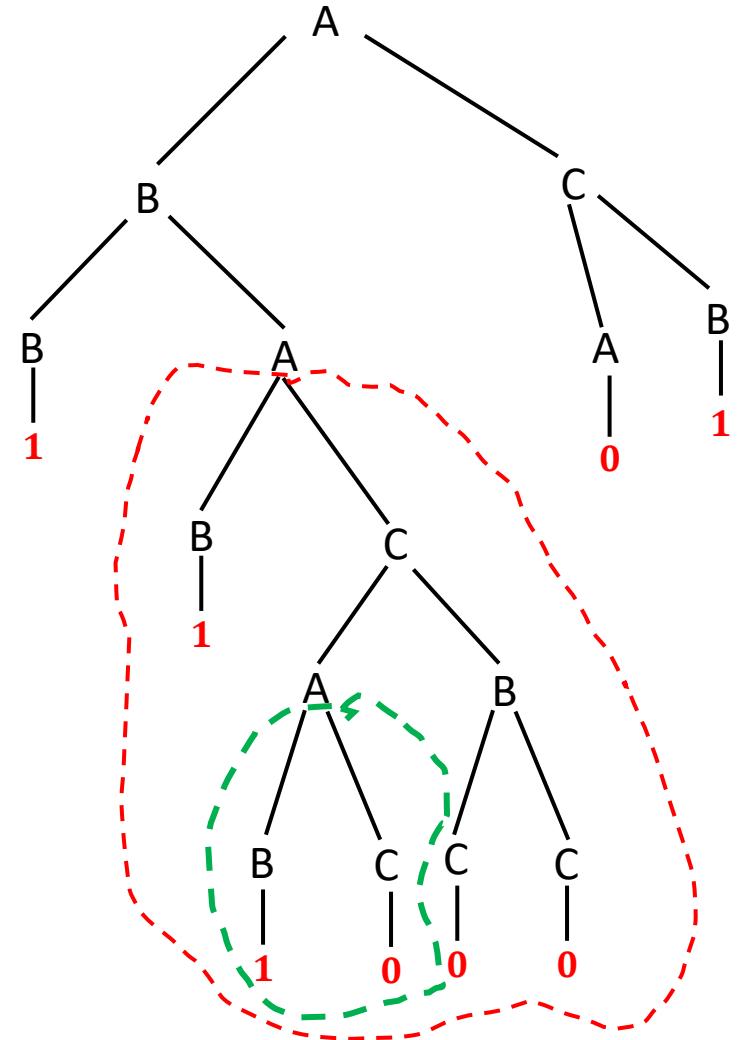
## Example:

$$\begin{aligned}A &\rightarrow BC|0 \\B &\rightarrow BA|1|CC \\C &\rightarrow AB|0\end{aligned}$$

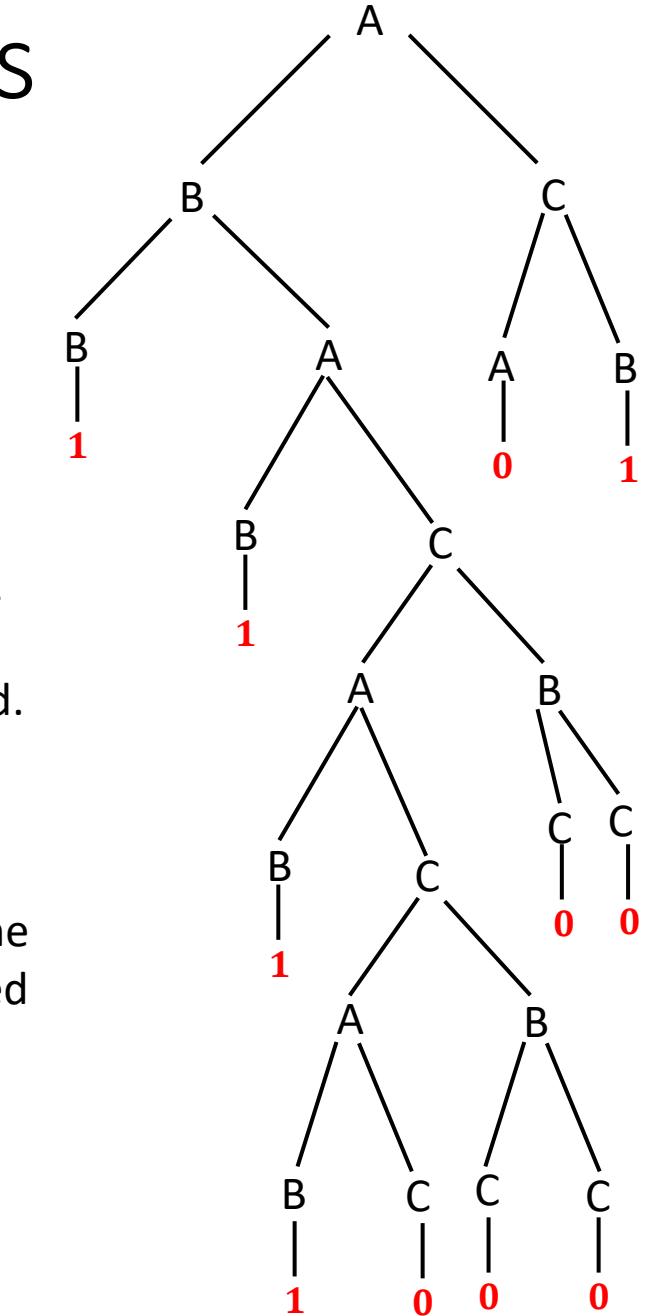
- No of variables  $|V| = 3$
- Consider a derivation of  $w = 11100001$
- Consider the longest path in the parse tree.
- Longest path length = 6, which is larger than  $|V|$ .
- There exists at least one variable that is repeated.
- For example:  $A$  – mark it.
- Replace the bottom most appearance of the variable  $A$  with the subtree in the middle rooted at  $A$  to obtain a new tree.



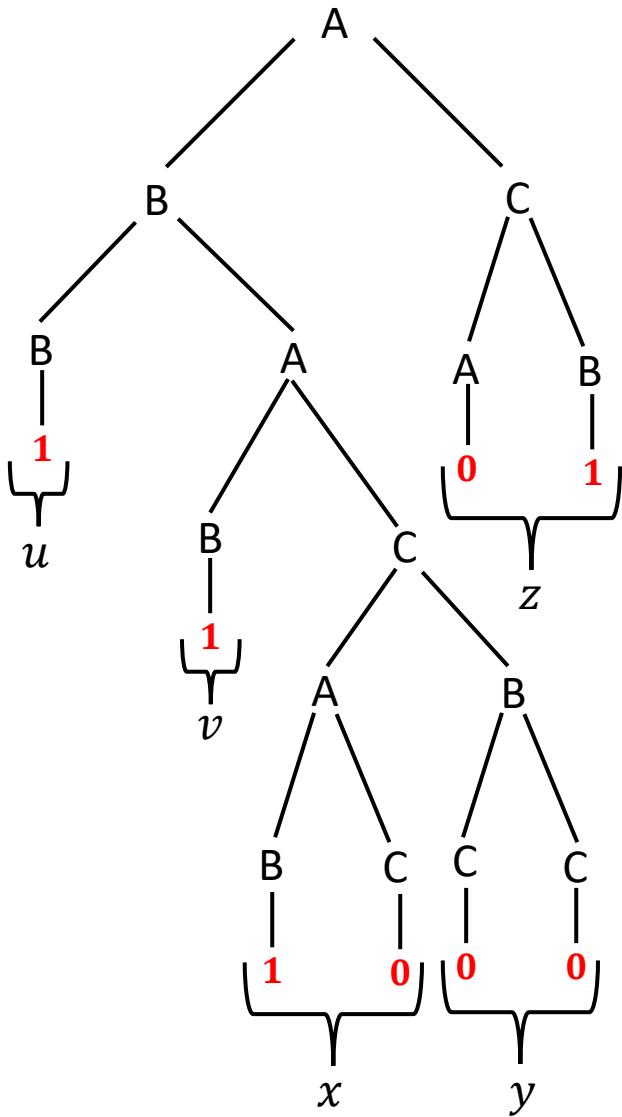
# Pumping Lemma for CFLs



- No of variables  $|V| = 3$
- Consider the longest path in the parse tree.
- Longest path length = 6, which is larger than  $|V|$ .
- There exists at least one variable that is repeated.
- For example:  $A$  – mark it.
- Replace the bottom most appearance of the variable  $A$  with the subtree in the middle rooted at  $A$  to obtain a new tree.



# Pumping Lemma for CFLs



For the tree in the left, the input string  $w$  can be split into five parts:  $w = uvxyz$

	<b>u</b>	<b>v</b>	<b>x</b>	<b>y</b>	<b>z</b>
L Tree	1	1	10	00	01

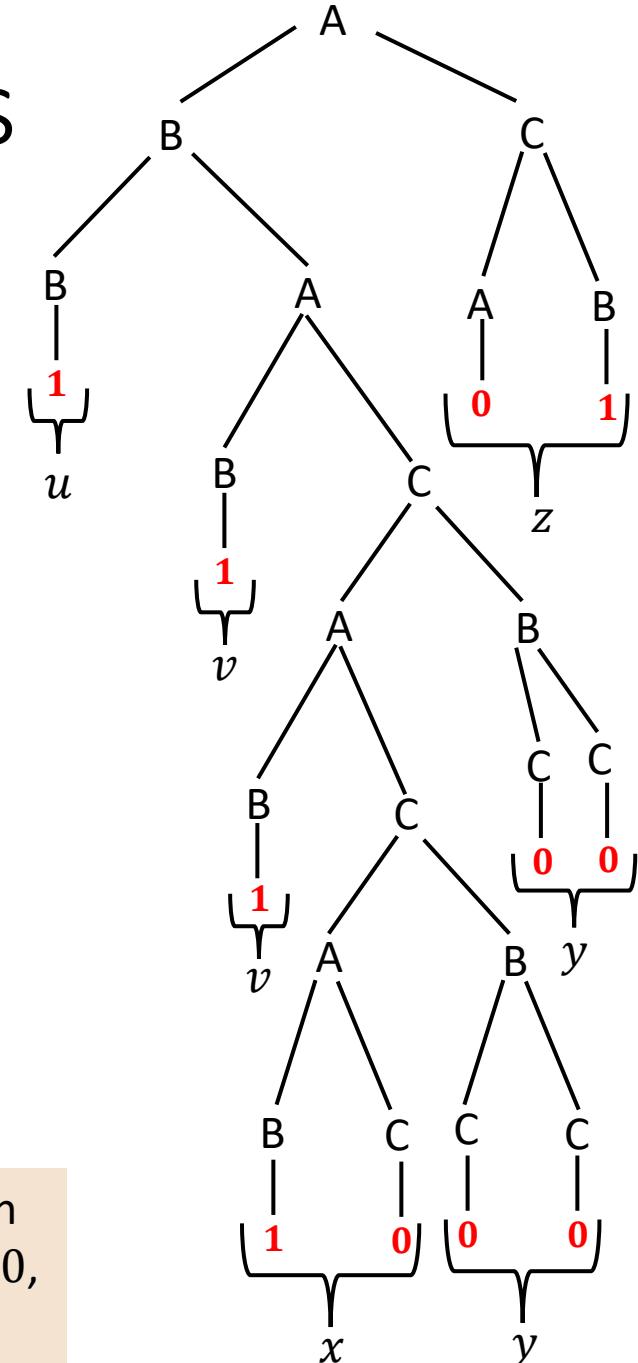
	<b>u</b>	<b>vv</b>	<b>x</b>	<b>yy</b>	<b>z</b>
R Tree	1	11	10	0000	01

By the substitution mentioned in the previous slide, we can keep pumping in  $v$  and  $y$  to get new strings of the form  $w = uv^i xy^i z$  ( $i \geq 0$ ), and any such  $w \in L$  as it is a valid derivation.

**Other conditions:**

$|vy| \geq 1$ ,  $v, y$  cannot be both  $\epsilon$   
 $|vxy| \leq p$

In fact if  $L$  is a CFL,  $\exists p$  such that  $\forall w \in L$  of length  $|w| \geq p$ , we can split  $w = uvxyz$ , such that  $\forall i \geq 0$ ,  $w = uv^i xy^i z \in L$

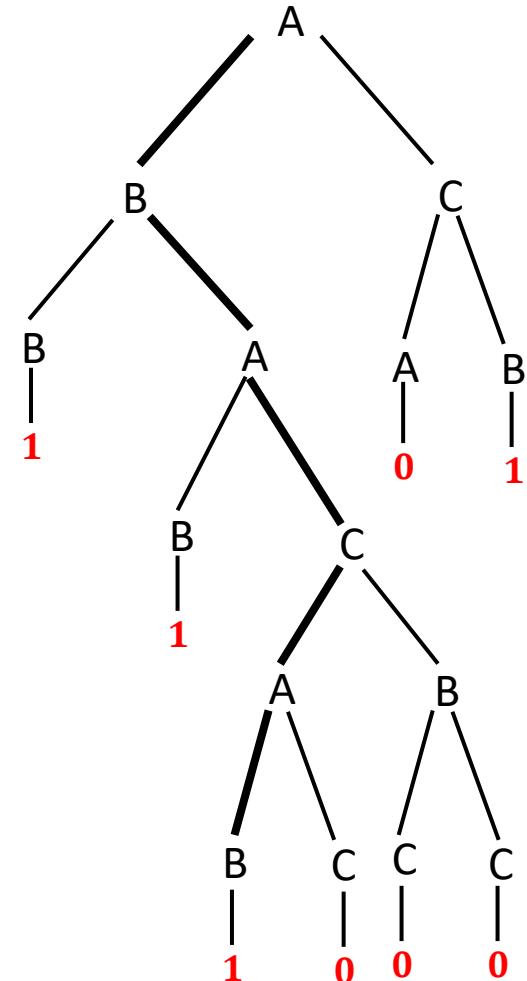


# Pumping Lemma for CFLs

## Properties of parse trees:

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ . Then:

- A path from the root to a leaf is sequence of variables and ends in a terminal or  $\epsilon$ .

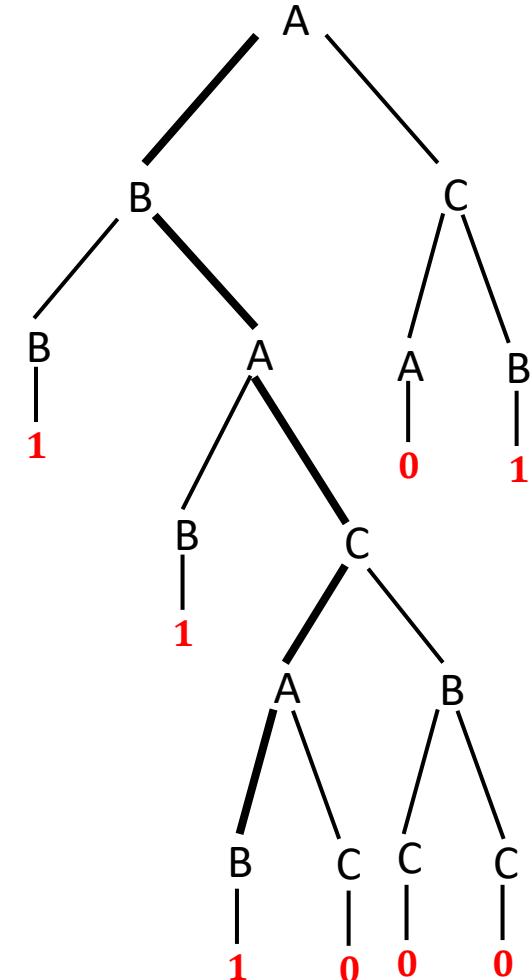


# Pumping Lemma for CFLs

## Properties of parse trees:

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ . Then:

- A path from the root to a leaf is sequence of variables and ends in a terminal or  $\epsilon$ .
- Let  $d$  be the maximum number of variables in the RHS of any rule of  $G$ .
- For example: If  $G$  is in CNF,  $d = 2$ .



# Pumping Lemma for CFLs

## Properties of parse trees:

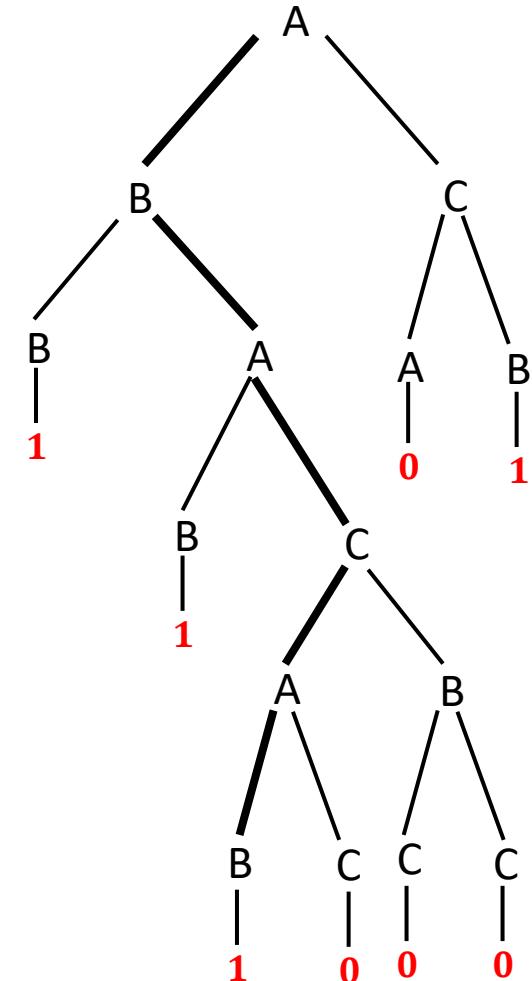
Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ . Then:

- A path from the root to a leaf is sequence of variables and ends in a terminal or  $\epsilon$ .
- Let  $d$  be the maximum number of variables in the RHS of any rule of  $G$ .
- For example: If  $G$  is in CNF,  $d = 2$ .

## Example:

$$\begin{aligned}A &\rightarrow BCDE|0 \\B &\rightarrow BA|1|CC \\C &\rightarrow AB|0 \\D &\rightarrow 01|EA\end{aligned}$$

$$d = ??$$



# Pumping Lemma for CFLs

## Properties of parse trees:

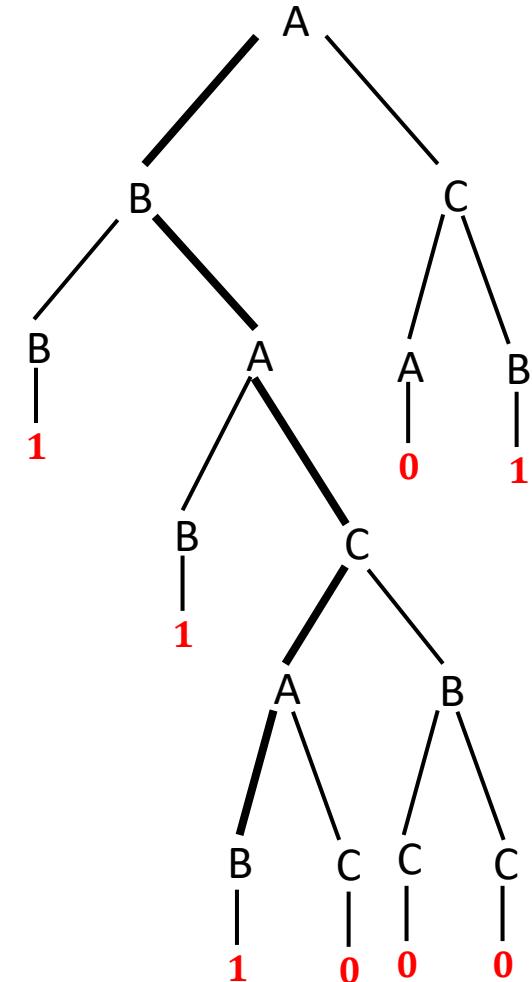
Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ . Then:

- A path from the root to a leaf is sequence of variables and ends in a terminal or  $\epsilon$ .
- Let  $d$  be the maximum number of variables in the RHS of any rule of  $G$ .
- For example: If  $G$  is in CNF,  $d = 2$ .

## Example:

$$\begin{aligned}A &\rightarrow BCDE|0 \\B &\rightarrow BA|1|CC \\C &\rightarrow AB|0 \\D &\rightarrow 01|EA\end{aligned}$$

$$d = 4$$

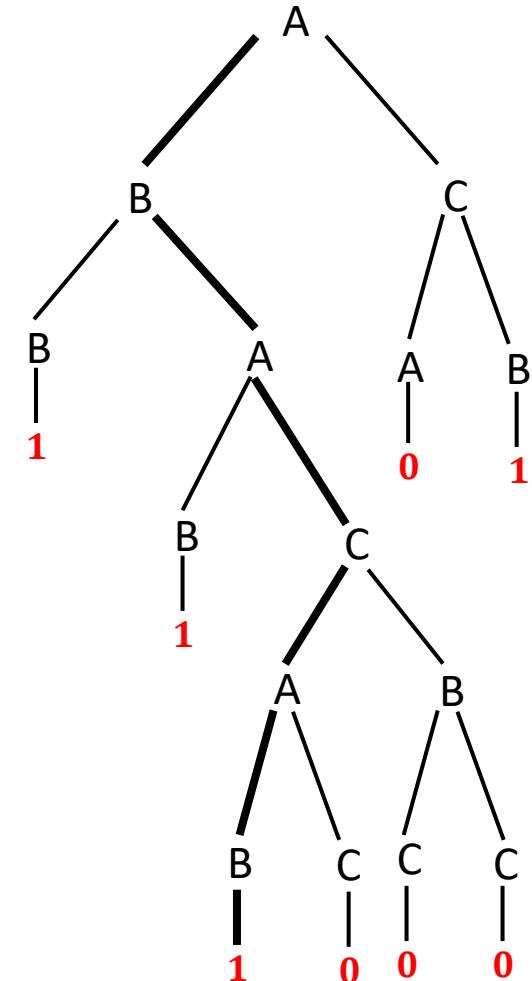


# Pumping Lemma for CFLs

## Properties of parse trees:

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ . Then:

- A path from the root to a leaf is sequence of variables and ends in a terminal or  $\epsilon$ .
- Let  $d$  be the maximum number of variables in the RHS of any rule of  $G$ .
- For example: If  $G$  is in CNF,  $d = 2$ .
- This results in a  $d$ -ary parse tree.

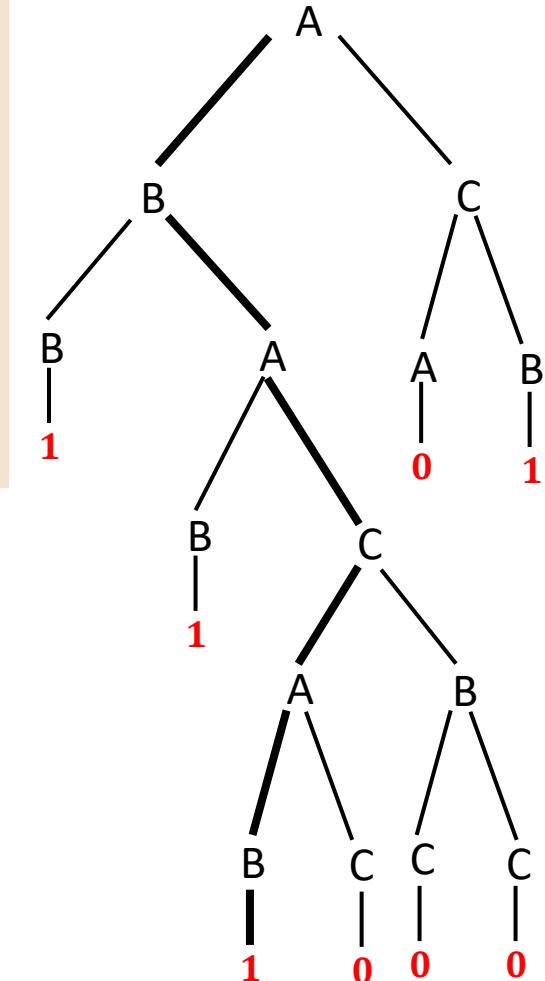


# Pumping Lemma for CFLs

## Properties of parse trees:

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ . Then:

- A path from the root to a leaf is sequence of variables and ends in a terminal or  $\epsilon$ .
  - Let  $d$  be the maximum number of variables in the RHS of any rule of  $G$ .
  - For example: If  $G$  is in CNF,  $d = 2$ .
  - This results in a  $d$ -ary parse tree.
  - Any  $T_w^G$  has at **level  $l$** , at most  $d^l$  **nodes**. Thus, any  $T_w^G$  of height  $h$  has at most  $d^h$  terminals.

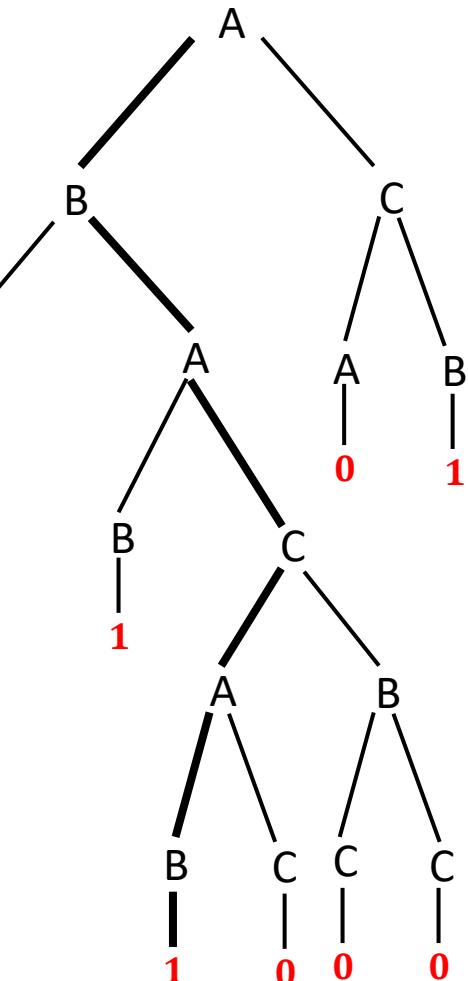


# Pumping Lemma for CFLs

## Properties of parse trees:

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ . Then:

- A path from the root to a leaf is sequence of variables and ends in a terminal or  $\epsilon$ .
- Let  $d$  be the maximum number of variables in the RHS of any rule of  $G$ .
- For example: If  $G$  is in CNF,  $d = 2$ .
- This results in a  $d$ -ary parse tree.
- Any  $T_w^G$  has at **level  $l$** , at most  **$d^l$  nodes**. Thus, any  $T_w^G$  of height  $h$  has at most  $d^h$  terminals.



- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V|+1$ .

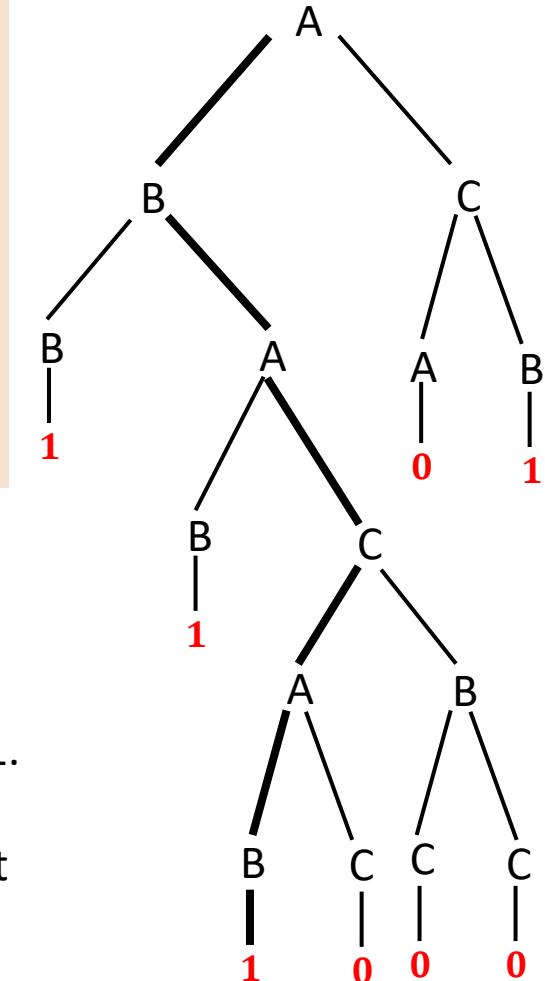
# Pumping Lemma for CFLs

## Properties of parse trees:

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ . Then:

- A path from the root to a leaf is sequence of variables and ends in a terminal or  $\epsilon$ .
- Let  $d$  be the maximum number of variables in the RHS of any rule of  $G$ .
- For example: If  $G$  is in CNF,  $d = 2$ .
- This results in a  $d$ -ary parse tree.
- Any  $T_w^G$  has at **level  $l$** , at most  **$d^l$  nodes**. Thus, any  $T_w^G$  of height  $h$  has at most  $d^h$  terminals.

- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V|+1$ .
- The longest path from the Start Variable  $S$  to a terminal is at least  $|V| + 1$  (containing at least  $|V| + 1$  variables and 1 terminal).



# Pumping Lemma for CFLs

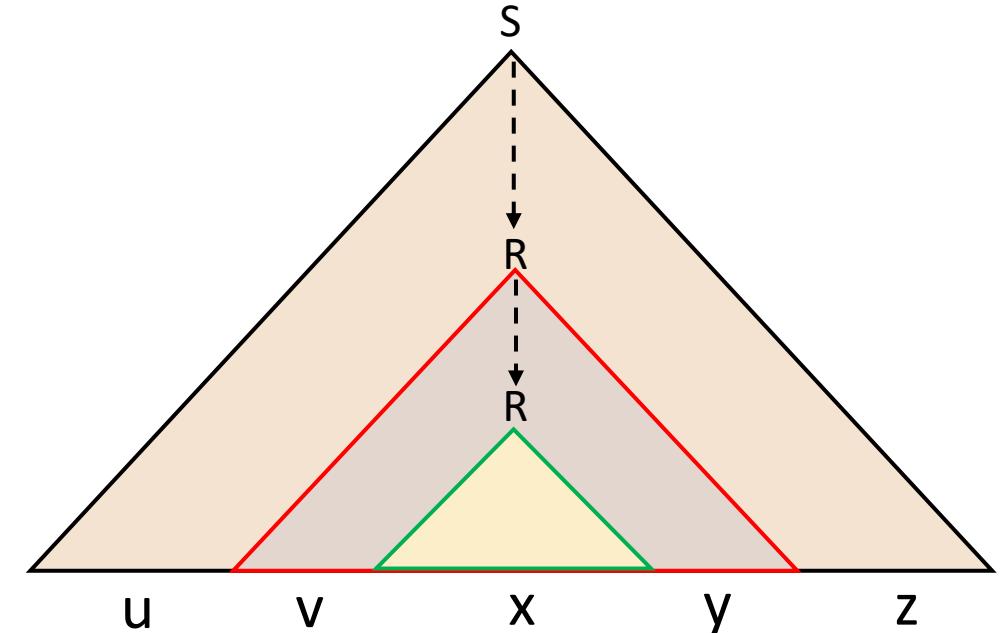
Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V|+1$ .
- The longest path from the Start Variable  $S$  to a terminal is  $\geq |V| + 1$ .
- Consider the lowest  $|V| + 1$  variables in that path.
- By the pigeonhole principle, within the lowest  $|V| + 1$  variables, at least one variable is repeated.

# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V| + 1$ .
- The longest path from the Start Variable  $S$  to a terminal is at least  $|V| + 1$ .
- Consider the lowest  $|V| + 1$  variables in that path.
- By the pigeonhole principle, within the lowest  $|V| + 1$  variables, at least one variable is repeated.

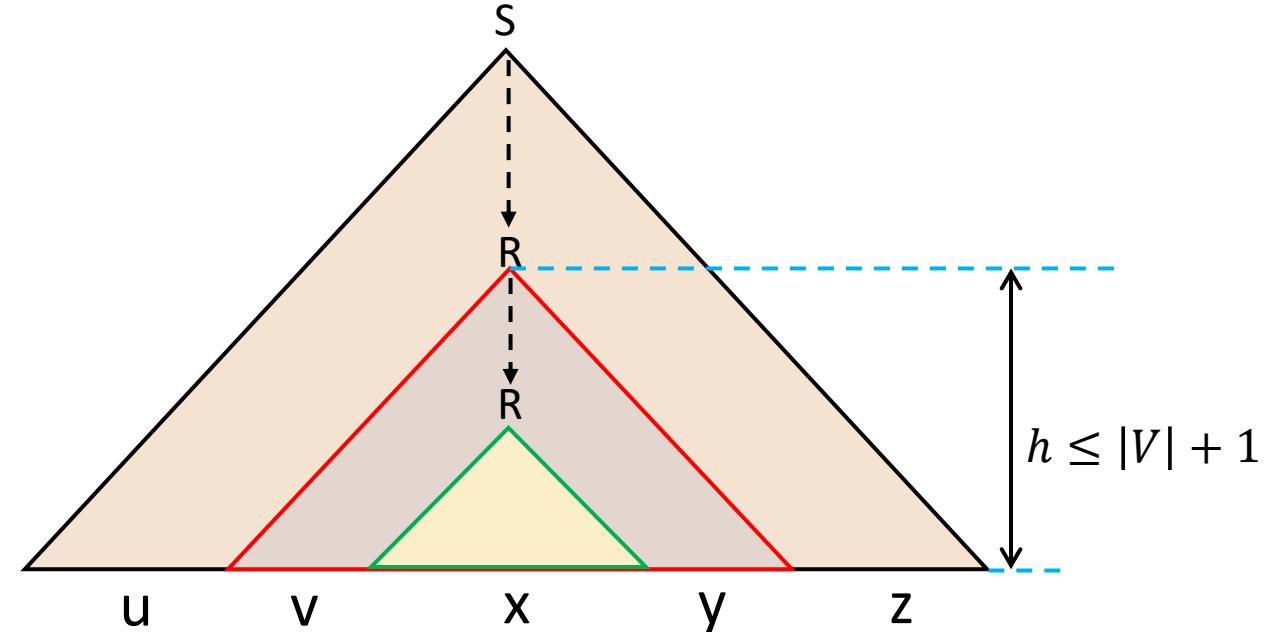


Then any string  $w$  such that  $|w| \geq p$  can be partitioned as  $w = uvxyz$

# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V| + 1$ .
- Consider the longest path in the parse tree.
- By the pigeonhole principle, within the lowest  $|V| + 1$  variables, at least one variable is repeated



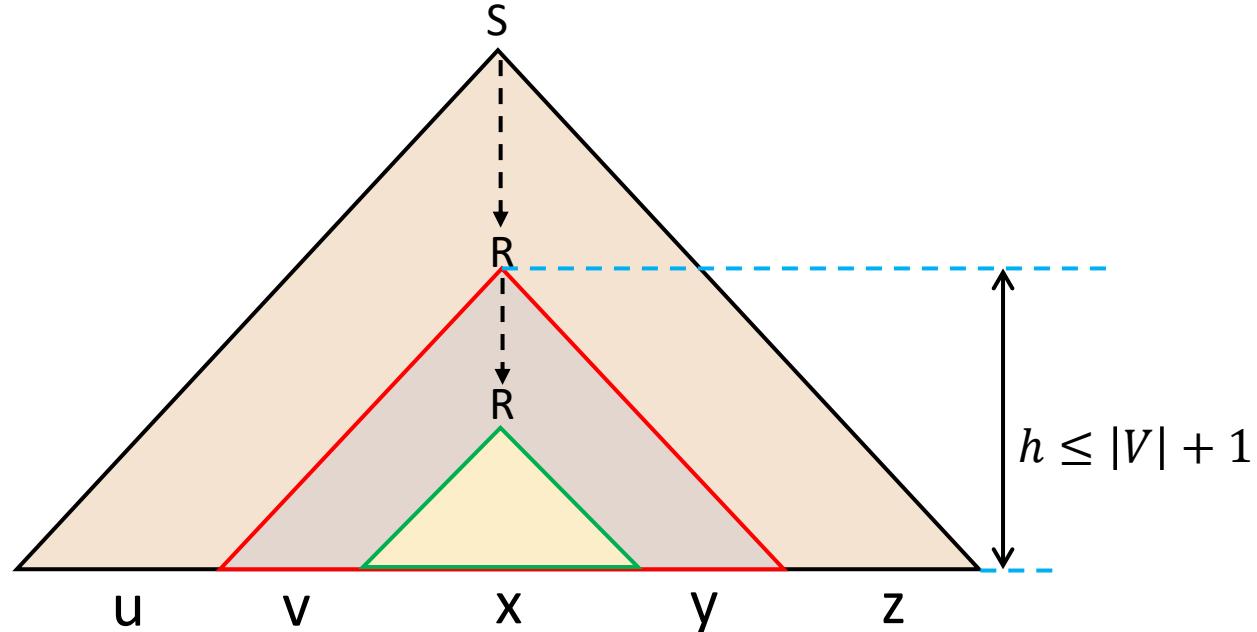
Then any string  $w$  such that  $|w| \geq p$ , can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$

# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V| + 1$ .
- Consider the lowest  $|V| + 1$  variables in the longest path of length  $\geq |V| + 1$ .



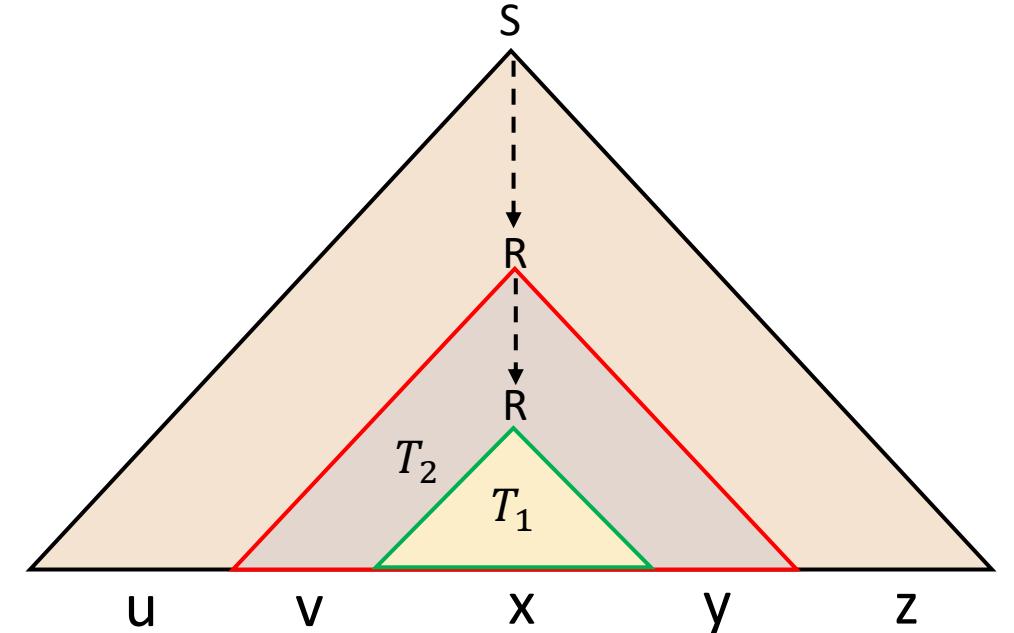
Then any string  $w$  such that  $|w| \geq p$ , can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$  - the uppermost  $R$  falls within the bottom  $|V| + 1$  variables in the longest path and so the length of the string it can generate is  $l \leq d^{|V|+1} = p$ .

# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V|+1$ .
- Consider the lowest  $|V| + 1$  variables in the longest path of  $h \geq |V| + 1$ .

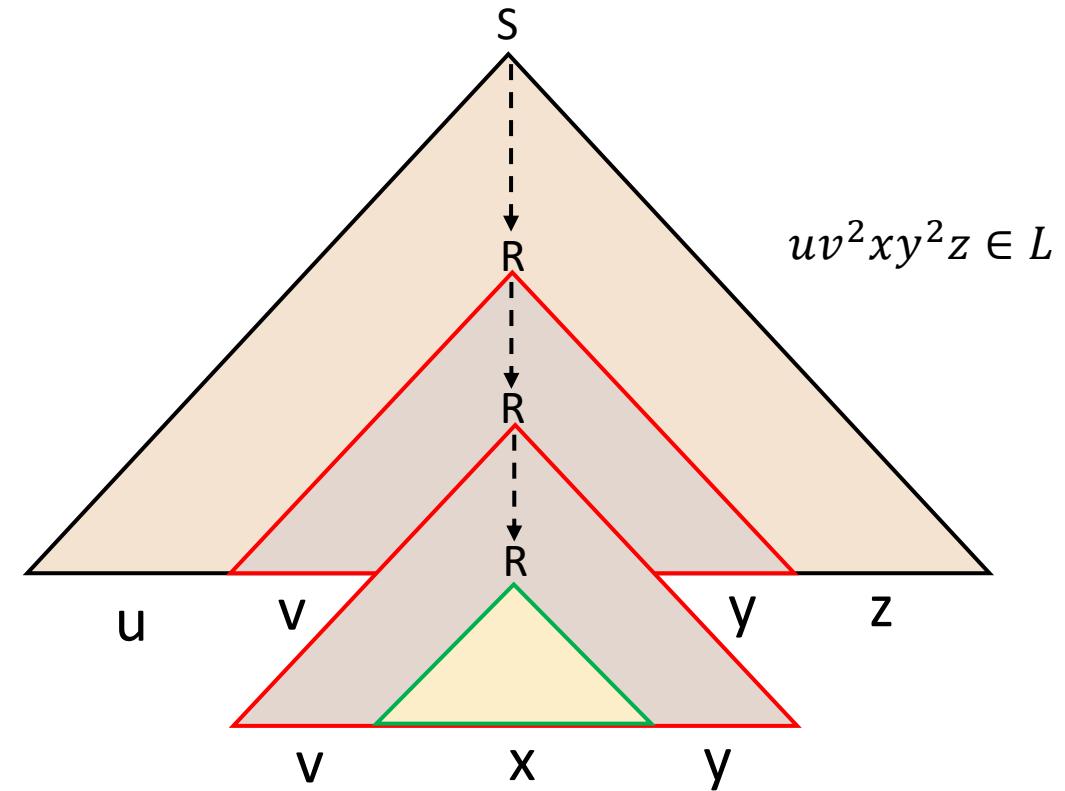
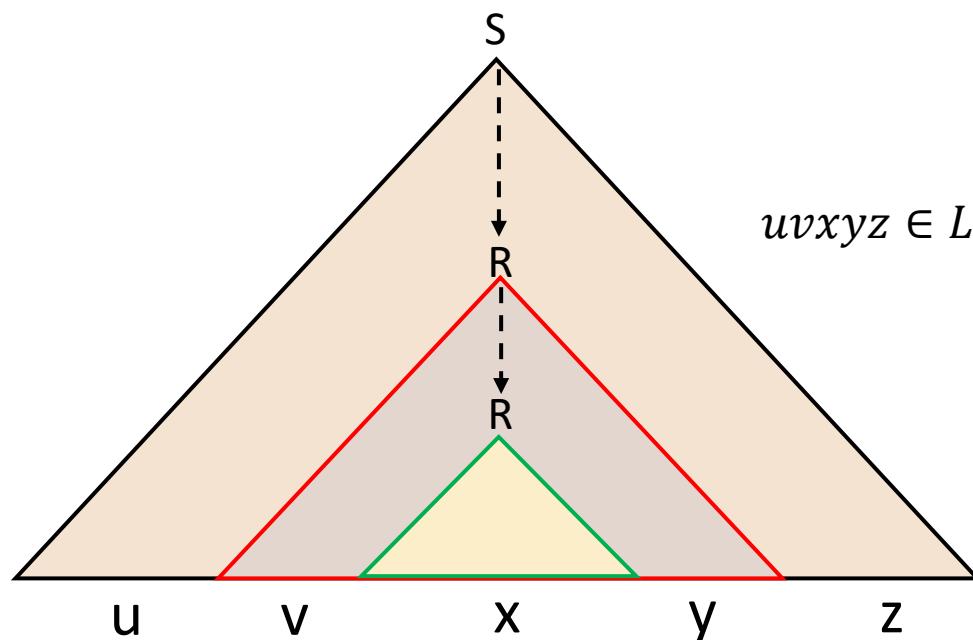


Then any string  $w$  such that  $|w| \geq p$  can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i > 0$  – Replace the subtree rooted at  $T_1$  with the subtree rooted at  $T_2$ .

# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

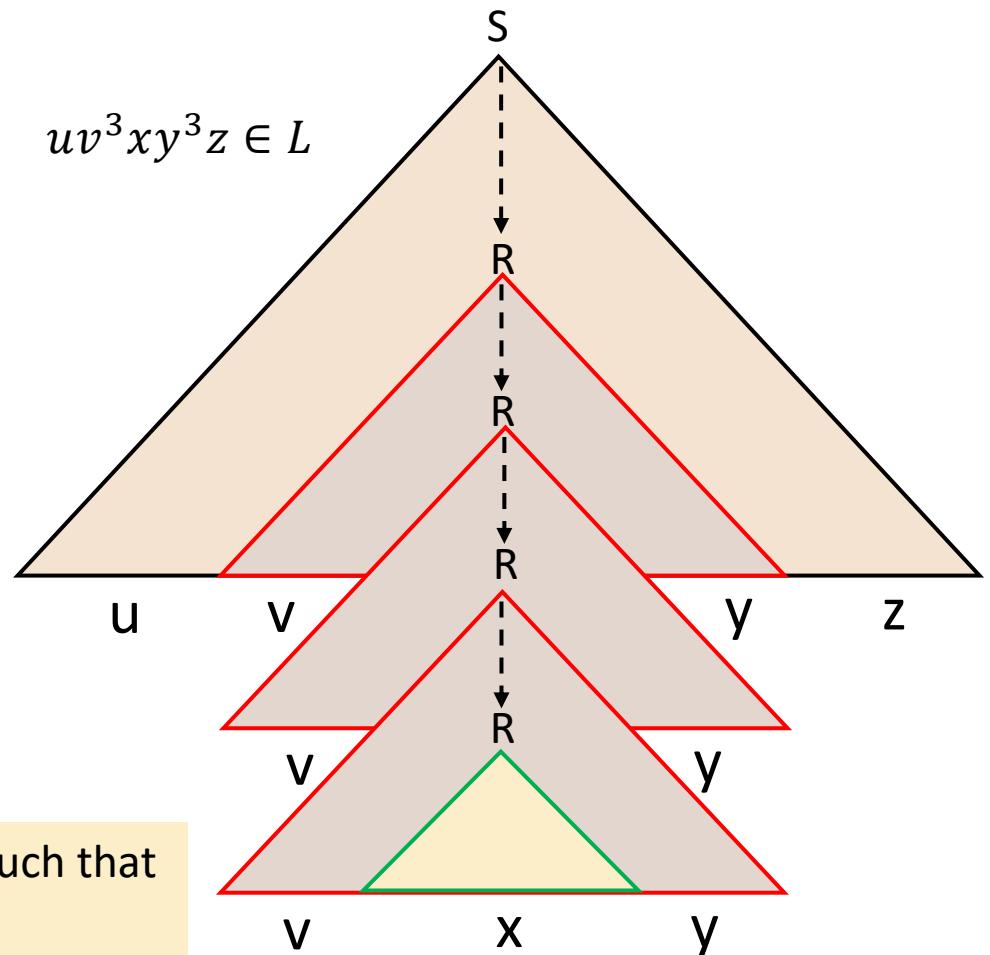
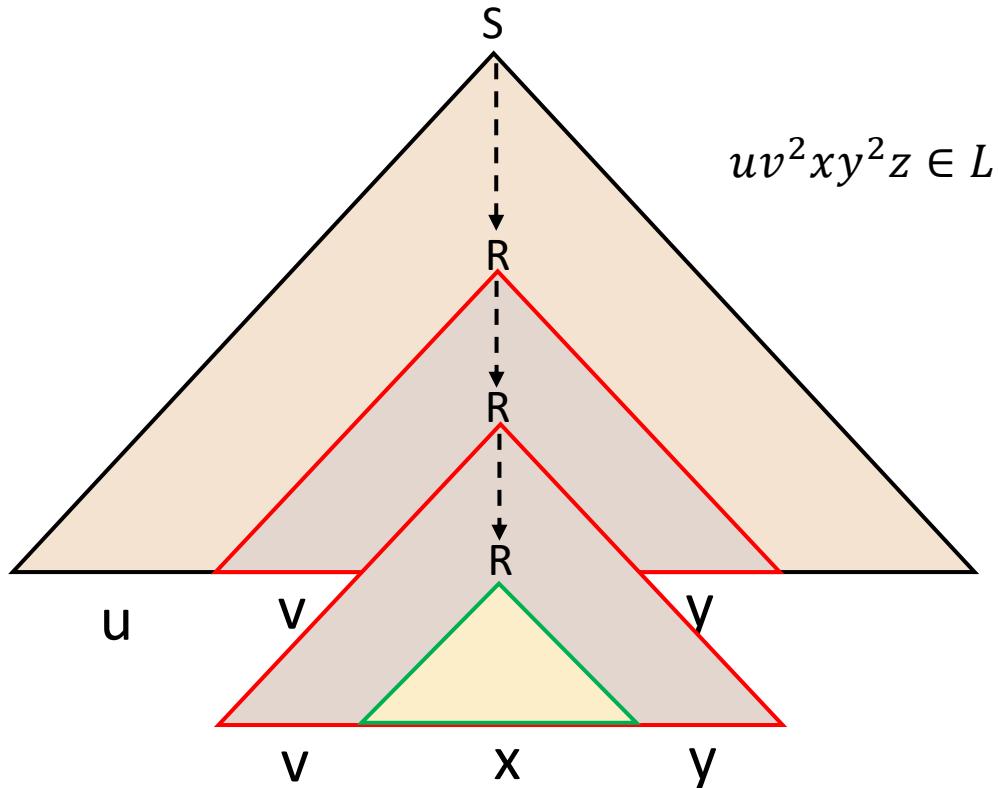


Then any string  $w$ , such that  $|w| \geq p$  can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i > 0$  – Replace the subtree rooted at  $T_1$  with the subtree rooted at  $T_2$ .

# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

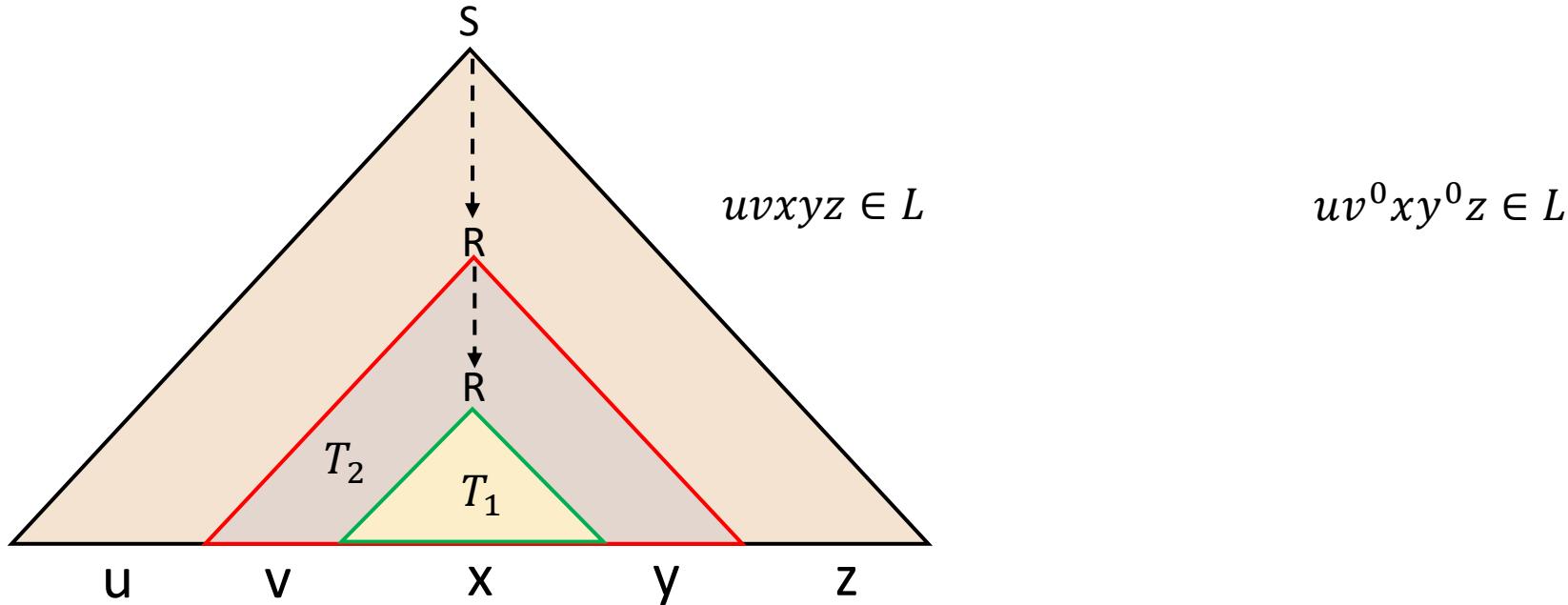


Then any string  $w$  such that  $|w| \geq p$  can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i > 0$

# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

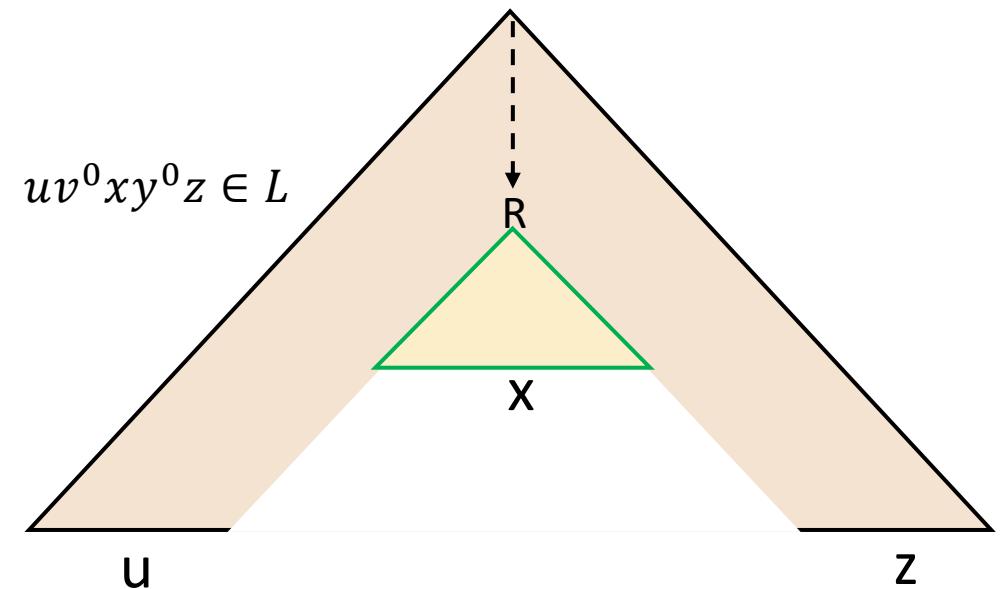
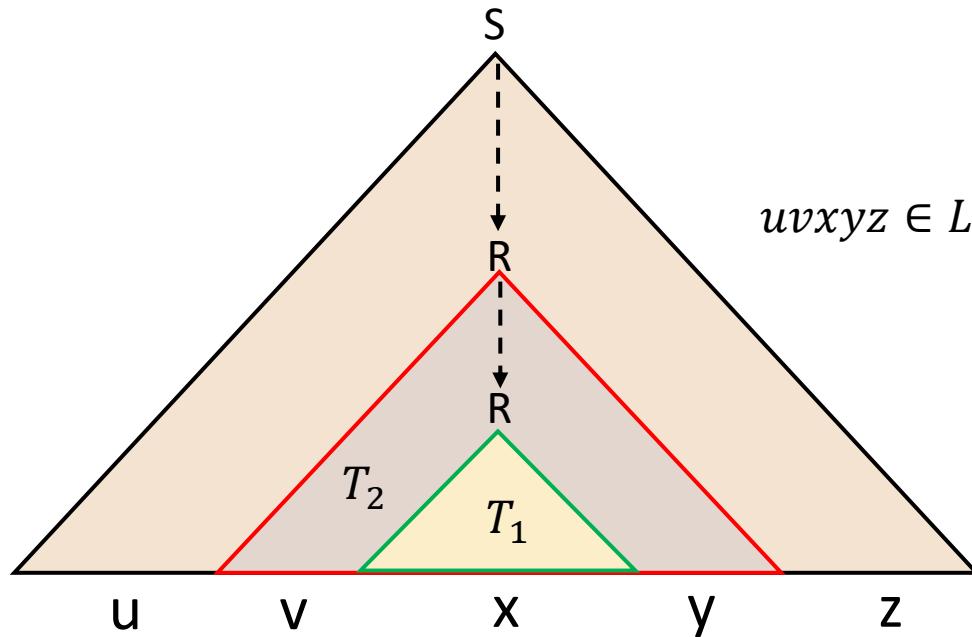


Then any string  $w$  such that  $|w| \geq p$  can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$  – for the  $i = 0$  case, **replace the subtree rooted at  $T_2$  with the subtree rooted at  $T_1$**

# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .



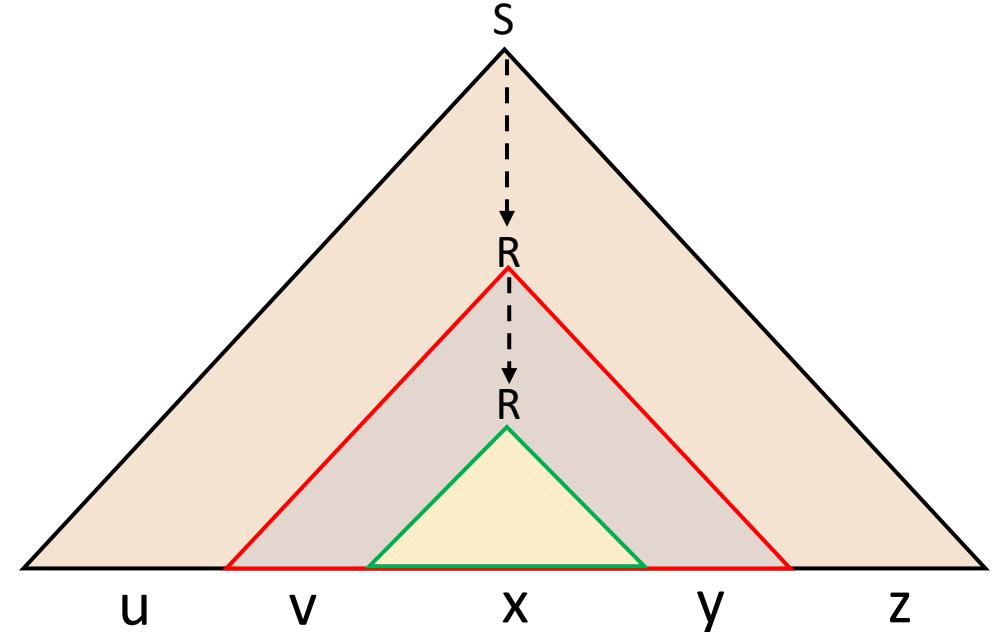
Then any string  $w$  such that  $|w| \geq p$  can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$  – for the  $i = 0$  case, replace the subtree rooted at  $T_2$  with the subtree rooted at  $T_1$

# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V| + 1$ .
- The longest path from the Start Variable  $S$  to a terminal is at least  $|V| + 1$ .
- Consider the lowest  $|V| + 1$  variables in that path.
- By the pigeonhole principle, within the lowest  $|V| + 1$  variables, at least one variable is repeated



Then any string  $w$  such that  $|w| \geq p$  can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

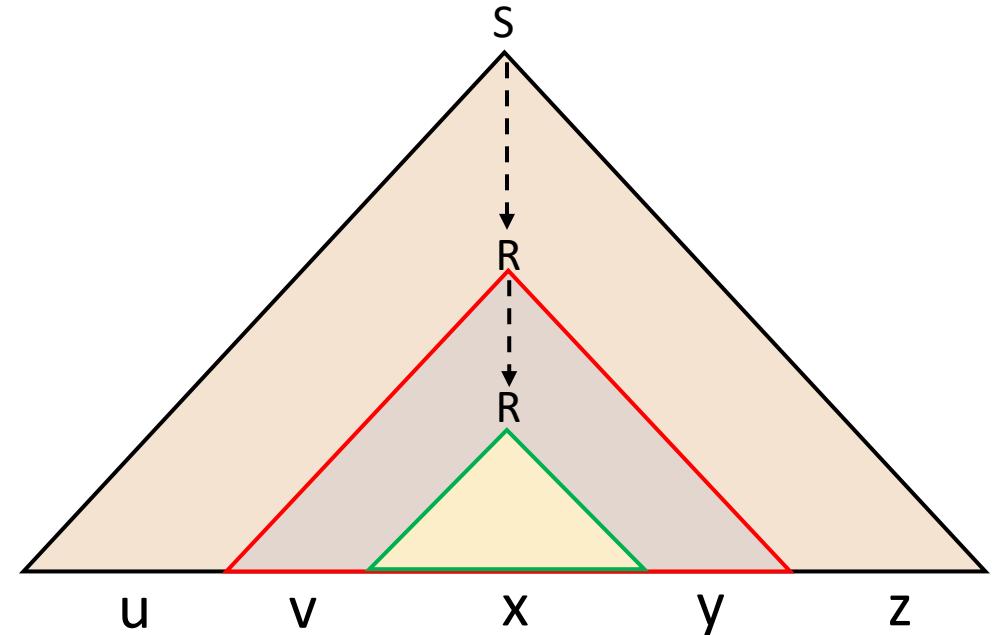
# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider a parse tree  $T_w^G$  of  $G$  that yields  $w$ .

- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V| + 1$ .
- The longest path from the Start Variable  $S$  to a terminal is at least  $|V| + 1$
- Consider the lowest  $|V| + 1$  variables in that path.
- By the pigeonhole principle, within the lowest  $|V| + 1$  variables, at least one variable is repeated

Then any string  $w$  such that  $|w| \geq p$  can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$



What if  $G$  is ambiguous? More than one parse tree that generates  $w$ .

Pick the one with the smallest number of nodes. So  $T_w^G$  is the smallest parse tree generating  $w$  (smallest height)

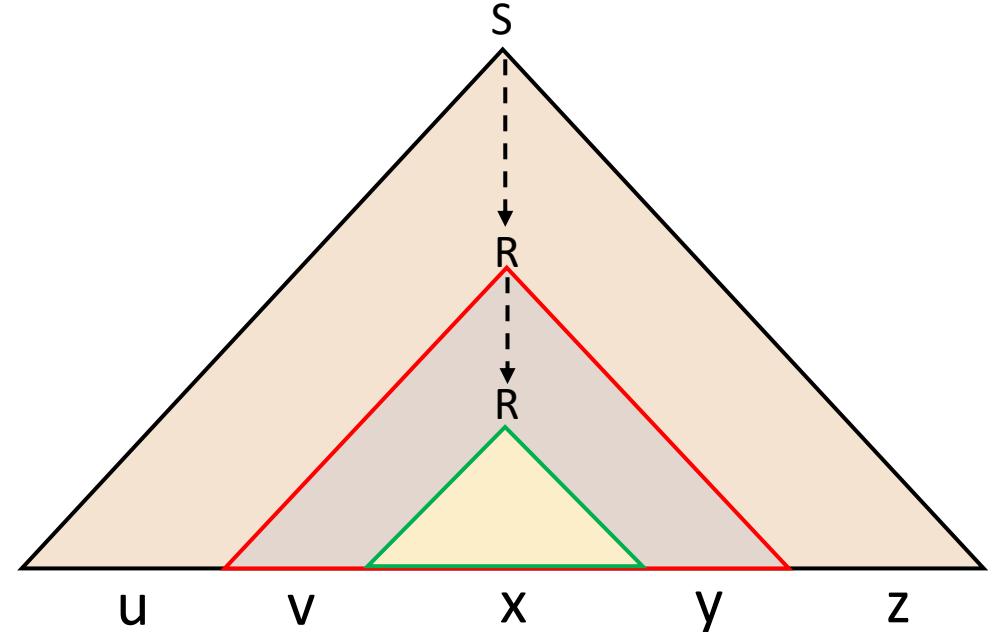
# Pumping Lemma for CFLs

Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider the smallest parse tree  $T_w^G$  of  $G$  that yields  $w$ .

- Let  $|V|$  be the total number of variables in the Grammar  $G$ .
- If  $w \in L$  such that  $|w| = p = d^{|V|+1}$ , the underlying parse tree would have a height  $\geq |V| + 1$ .
- The longest path from the Start Variable  $S$  to a terminal is at least  $|V| + 1$ .
- Consider the lowest  $|V| + 1$  variables in that path.
- By the pigeonhole principle, within the lowest  $|V| + 1$  variables, at least one variable is repeated

Then any string  $w$  such that  $|w| \geq p$  can be partitioned as  $w = uvxyz$  such that

- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$



$T_w^G$  is the smallest parse tree generating  $w$ .

This leads to an additional condition!

$v, y$  cannot be both empty, i.e.  $|vy| \geq 1$

# Pumping Lemma for CFLs

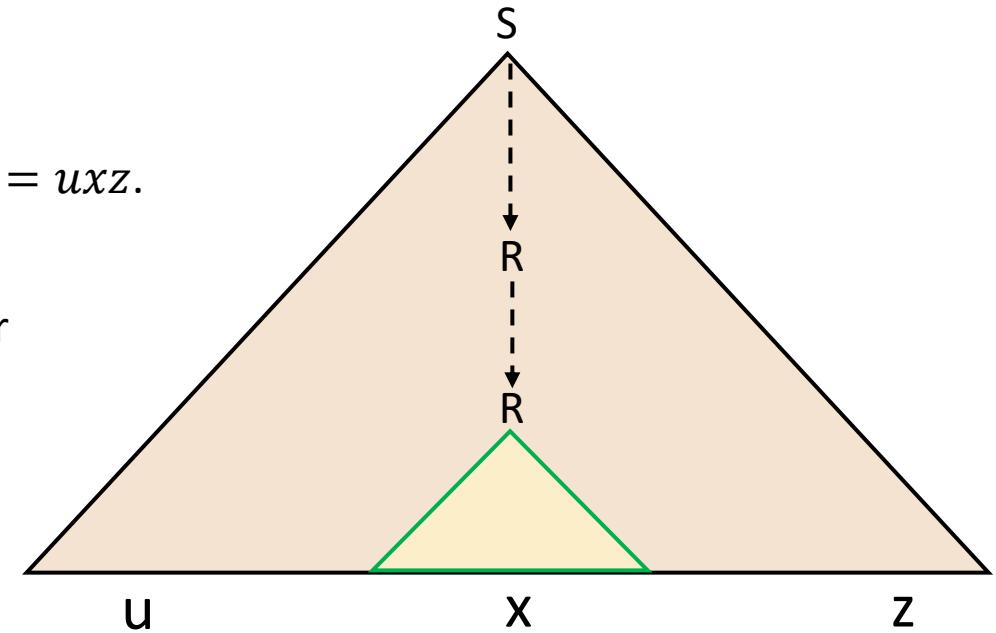
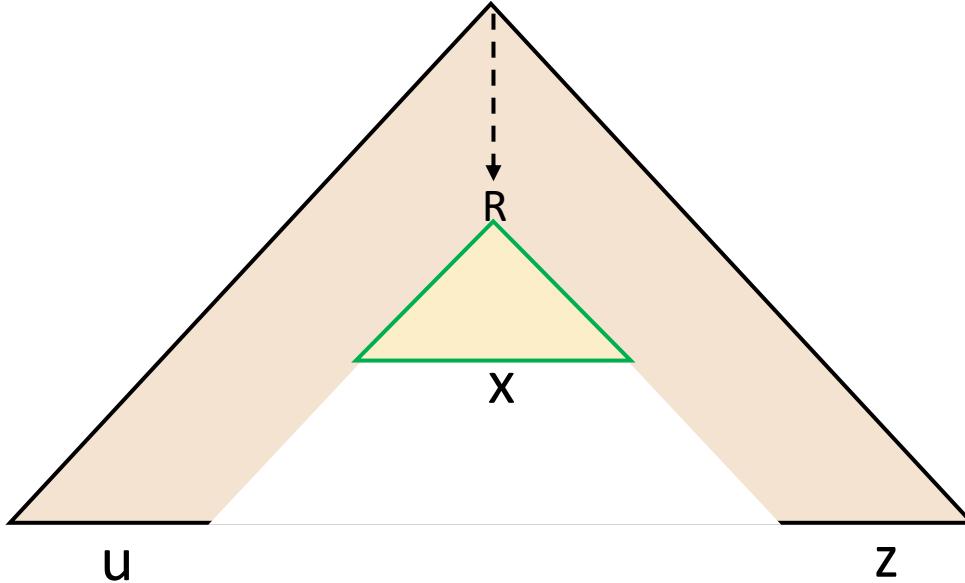
Let  $L$  be a CFL and  $G$  be such that  $L = \mathcal{L}(G)$  and  $w \in L$ . Consider the smallest parse tree  $T_w^G$  of  $G$  that yields  $w$ .

**$v, y$  cannot be both empty, i.e.  $|vy| \geq 1$**

**Proof by contradiction:** Let us assume that they were both empty, i.e.  $w = uxz$ .

Then  $T_w^G$  would look like this.

However, if we substitute the smaller subtree rooted at  $R$  with the higher subtree, we obtain



The parse tree to the left generates  $w$  and has fewer nodes which is a **contradiction!!**

# Pumping Lemma for CFLs

**Putting things together:**

**Pumping Lemma for CFL:** If  $L$  is Context Free, then there exists  $p > 0$  (pumping length), such that, for any  $w \in L$  of length  $|w| \geq p$ ,  $w$  can be split into five parts, i.e.

$$w = uvxyz$$

satisfying the following conditions:

- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

We have proved this in the previous slides.

# Pumping Lemma for CFLs

**Pumping Lemma for CFL:** IF  $L$  is Context Free, THEN there exists  $p > 0$  (pumping length), such that, for any  $w \in L$  of length  $|w| \geq p$ ,  $w$  can be split into five parts, i.e.

$$w = uvxyz$$

satisfying the following conditions:

- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

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

IF  $L$  is Context Free, THEN conditions of Pumping Lemma are Satisfied

$\equiv$

IF conditions of Pumping Lemma are NOT satisfied THEN  $L$  is NOT Context Free

In order to prove that a language is not Context Free, assume that it is Context Free and obtain a contradiction.

# Non Context Free Languages

$L = \{0^n 1^n 2^n | n \geq 0\}$  is not Context-Free.

**Proof:** We shall prove this by contradiction. Let  $L$  be a CFL and so it must satisfy the conditions of the Pumping Lemma. Let  $p$  be the pumping length and so  $w = 0^p 1^p 2^p \in L$ .

# Non Context Free Languages

$L = \{0^n 1^n 2^n | n \geq 0\}$  is not Context-Free.

**Proof:** We shall prove this by contradiction. Let  $L$  be a CFL and so it must satisfy the conditions of the Pumping Lemma. Let  $p$  be the pumping length and so  $w = 0^p 1^p 2^p \in L$ .

Note that  $|w| = 3p$  ( $\geq p$ ). The pumping lemma states that  $w$  can be split into  $w = uvxyz$  such that

- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

# Non Context Free Languages

$L = \{0^n 1^n 2^n | n \geq 0\}$  is not Context-Free.

**Proof:** We shall prove this by contradiction. Let  $L$  be a CFL and so it must satisfy the conditions of the Pumping Lemma. Let  $p$  be the pumping length and so  $w = 0^p 1^p 2^p \in L$ .

Note that  $|w| = 3p$  ( $\geq p$ ). The pumping lemma states that  $w$  can be split into  $w = uvxyz$  such that

- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

First observe that in  $w = 0^p 1^p 2^p$ , the last 0 and the first 2 is separated by  $p$  1's. As  $|vxy| \leq p$ , the substring  $vxy$  has at most two distinct symbols. Now consider the string  $w' = uv^2 xy^2 z$ . What are the possibilities of  $vxy$ ?

# Non Context Free Languages

$L = \{0^n 1^n 2^n | n \geq 0\}$  is not Context-Free.

**Proof:** We shall prove this by contradiction. Let  $L$  be a CFL and so it must satisfy the conditions of the Pumping Lemma. Let  $p$  be the pumping length and so  $w = 0^p 1^p 2^p \in L$ .

Note that  $|w| = 3p$  ( $\geq p$ ). The pumping lemma states that  $w$  can be split into  $w = uvxyz$  such that

- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

First observe that in  $w = 0^p 1^p 2^p$ , the last 0 and the first 2 is separated by  $p$  1's. As  $|vxy| \leq p$ , the substring  $vxy$  has at most two distinct symbols. Now consider the string  $w' = uv^2 xy^2 z$ . What are the possibilities of  $vxy$ ?

- $vxy = 0^k$  or  $1^k$  or  $2^k, k \leq p$ : Then  $w' \notin L$ . (E.g: If  $vxy = 0^k$ , then,  $w'$  will have more 0's than 1's and 2's.)

# Non Context Free Languages

$L = \{0^n 1^n 2^n | n \geq 0\}$  is not Context-Free.

**Proof:** We shall prove this by contradiction. Let  $L$  be a CFL and so it must satisfy the conditions of the Pumping Lemma. Let  $p$  be the pumping length and so  $w = 0^p 1^p 2^p \in L$ .

Note that  $|w| = 3p$  ( $\geq p$ ). The pumping lemma states that  $w$  can be split into  $w = uvxyz$  such that

- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

First observe that in  $w = 0^p 1^p 2^p$ , the last 0 and the first 2 is separated by  $p$  1's. As  $|vxy| \leq p$ , the substring  $vxy$  has at most two distinct symbols. Now consider the string  $w' = uv^2 xy^2 z$ . What are the possibilities of  $vxy$ ?

- $vxy = 0^k$  or  $1^k$  or  $2^k, k \leq p$ : Then  $w' \notin L$ . (E.g: If  $vxy = 0^k$ , then,  $w'$  will have more 0's than 1's and 2's.)
- $vxy = 0^m 1^n$  or  $1^m 2^n, m + n \leq p$ : Again,  $w' \notin L$ .

# Non Context Free Languages

$L = \{0^n 1^n 2^n | n \geq 0\}$  is not Context-Free.

**Proof:** We shall prove this by contradiction. Let  $L$  be a CFL and so it must satisfy the conditions of the Pumping Lemma. Let  $p$  be the pumping length and so  $w = 0^p 1^p 2^p \in L$ .

Note that  $|w| = 3p$  ( $\geq p$ ). The pumping lemma states that  $w$  can be split into  $w = uvxyz$  such that

- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

First observe that in  $w = 0^p 1^p 2^p$ , the last 0 and the first 2 is separated by  $p$  1's. As  $|vxy| \leq p$ , the substring  $vxy$  has at most two distinct symbols. Now consider the string  $w' = uv^2 xy^2 z$ . What are the possibilities of  $vxy$ ?

- $vxy = 0^k$  or  $1^k$  or  $2^k, k \leq p$ : Then  $w' \notin L$ . (E.g: If  $vxy = 0^k$ , then,  $w'$  will have more 0's than 1's and 2's.)
- $vxy = 0^m 1^n$  or  $1^m 2^n, m + n \leq p$ : Again,  $w' \notin L$ . (E.g: If  $vxy = 0^m 1^n$ , then  $w'$  will have less 2's than the other two symbols)

# Non Context Free Languages

$L = \{0^n 1^n 2^n | n \geq 0\}$  is not Context-Free.

**Proof:** We shall prove this by contradiction. Let  $L$  be a CFL and so it must satisfy the conditions of the Pumping Lemma. Let  $p$  be the pumping length and so  $w = 0^p 1^p 2^p \in L$ .

Note that  $|w| = 3p$  ( $\geq p$ ). The pumping lemma states that  $w$  can be split into  $w = uvxyz$  such that

- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

First observe that in  $w = 0^p 1^p 2^p$ , the last 0 and the first 2 is separated by  $p$  1's. As  $|vxy| \leq p$ , the substring  $vxy$  has at most two distinct symbols. Now consider the string  $w' = uv^2 xy^2 z$ . What are the possibilities of  $vxy$ ?

- $vxy = 0^k$  or  $1^k$  or  $2^k, k \leq p$ : Then  $w' \notin L$ . (E.g: If  $vxy = 0^k$ , then,  $w'$  will have more 0's than 1's and 2's.)
- $vxy = 0^m 1^n$  or  $1^m 2^n, m + n \leq p$ : Again,  $w' \notin L$ . (E.g: If  $vxy = 0^m 1^n$ , then  $w'$  will have less 2's than the other two symbols)

Both cases lead to a contradiction. Hence,  $L \notin CFL$ .

# Non Context Free Languages

$L = \{0^n 1^n 2^n | n \geq 0\}$  is not Context-Free.

Other examples:

- $L = \{ww | w \in \{0, 1\}^*\}$
- $L = \{a^p | p \text{ is prime}\}$
- $L = \{0^n 1^{n^2} | n \geq 0\}$

.....

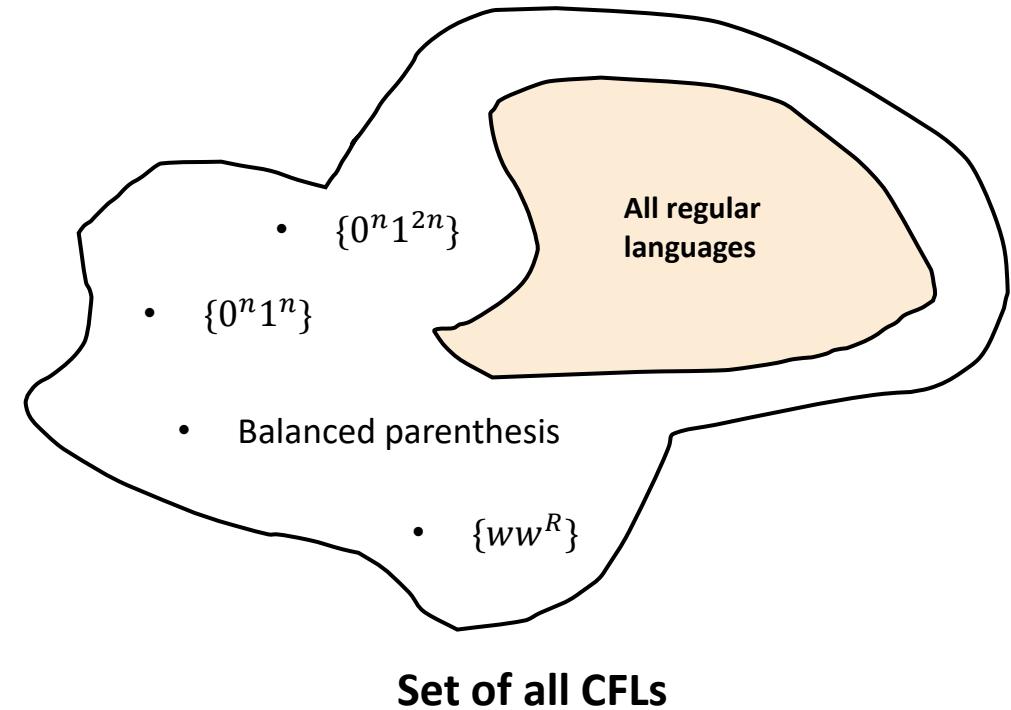
Recommend you to use Pumping Lemma and check that they are indeed not Context Free

# Closure properties of CFL

Now that we know that there are languages that are not Context Free – let us investigate the closure properties of CFLs.

**Recall** what we mean by the statement “**CFLs are closed under some operation**”

- We pick up points within the set of all CFLs (say  $L_1$  and  $L_2$ )
- Perform *set operations* such as Union, concatenation, Star, intersection, compliment etc on them.
- Observe whether the resulting language still belongs to the set of all CFLs.
- If so, we say, CFLs are **closed** under that operation otherwise we say CFLs are not closed under than operation.



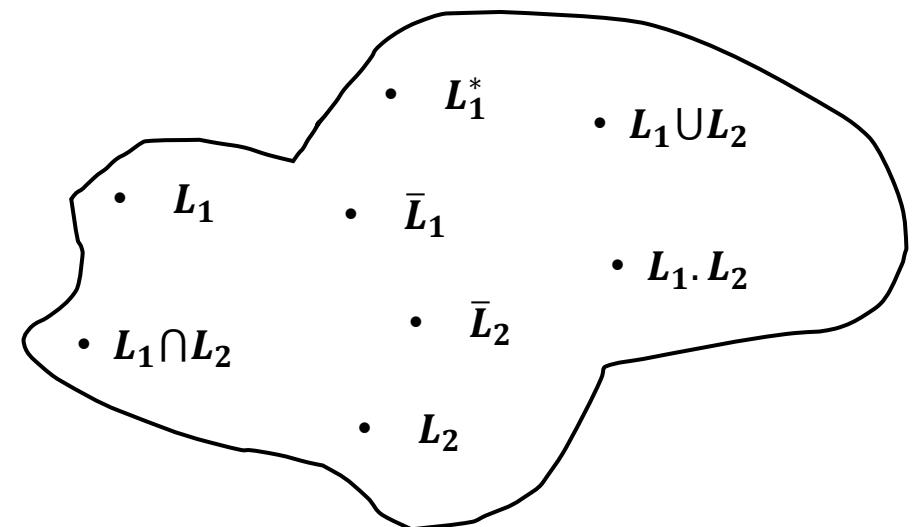
# Closure properties of CFL

**Some 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:**  $L_1 \cdot L_2 = \{xy|x \in L_1 \text{ and } y \in L_2\}$
- **Intersection:**  $L_1 \cap L_2 = \{x|x \in L_1 \text{ and } x \in L_2\}$
- **Star:**  $L_1^* = \{x_1x_2 \cdots x_k | k \geq 0 \text{ and each } x_i \in L\}$
- **Complementation:**  $\bar{L} = \{x|x \notin L\}$

**Recall that for Regular languages:** RL are closed under

- Union
- Intersection
- Star
- Complement
- Concatenation



Set of all regular Languages

What about CFLs?

# Closure properties of CFL

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

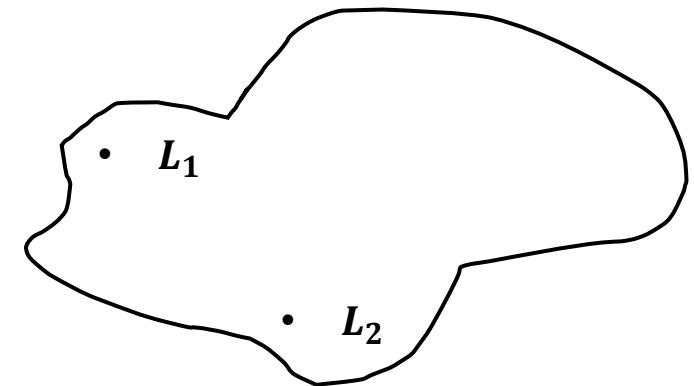
Suppose  $L_1$  and  $L_2$  are CFLs. Is  $L = L_1 \cup L_2$  also a CFL?

**Proof:** Suppose  $G_1$  and  $G_2$  be grammars such that  $L(G_1) = L_1$  and  $L(G_2) = L_2$ .

Suppose:

Rules of  $G_1$ :  $S_1 \rightarrow \dots$

Rules of  $G_2$ :  $S_2 \rightarrow \dots$



Set of all CFLs

# Closure properties of CFL

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

Suppose  $L_1$  and  $L_2$  are CFLs. Is  $L = L_1 \cup L_2$  also a CFL?

**Proof:** Suppose  $G_1$  and  $G_2$  be grammars such that  $L(G_1) = L_1$  and  $L(G_2) = L_2$ .

Suppose:

Rules of  $G_1$ :  $S_1 \rightarrow \dots$

Rules of  $G_2$ :  $S_2 \rightarrow \dots$

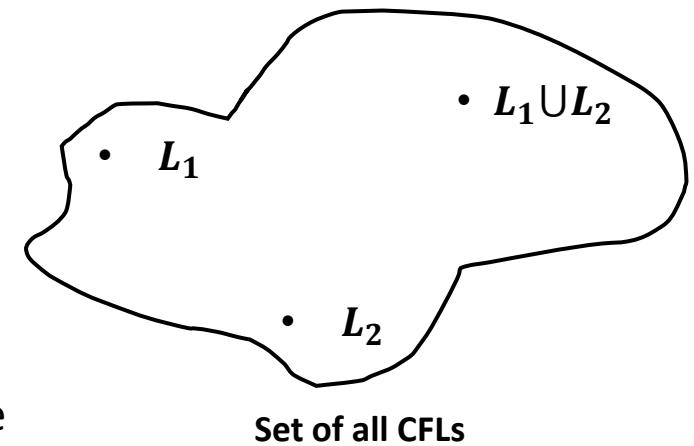
Also suppose that the rules of  $G_1$  and  $G_2$  have different variables.

Then the grammar for  $L_1 \cup L_2$  contains all the variables of  $G_1$  and  $G_2$ , all the terminals of  $G_1$  and  $G_2$ . Additionally,

Add a new start symbol  $S$  and the rules of  $G_1 \cup G_2$ :

$$S \rightarrow S_1 | S_2$$

followed by rules of  $G_1$  and rules of  $G_2$ . So CFLs are closed under union.



# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under Concatenation?**

Suppose  $L_1$  and  $L_2$  are CFLs. Is  $L = L_1 \cdot L_2$  also a CFL?

**Proof:** Suppose  $G_1$  and  $G_2$  be grammars such that  $L(G_1) = L_1$  and  $L(G_2) = L_2$ .

Suppose:

Rules of  $G_1$ :  $S_1 \rightarrow \dots$

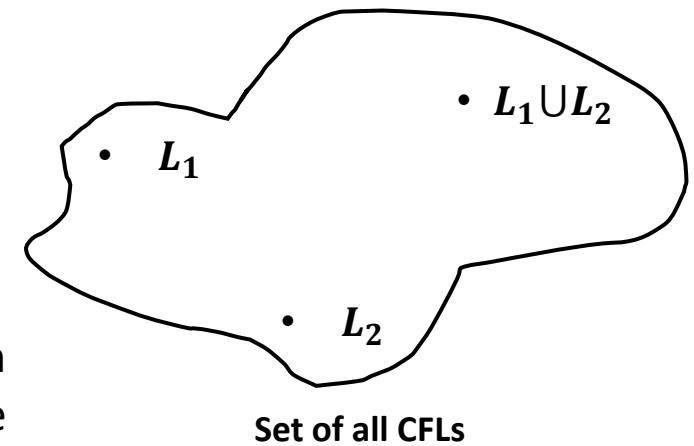
Rules of  $G_2$ :  $S_2 \rightarrow \dots$

Also suppose that the rules of  $G_1$  and  $G_2$  have different variables. Then define  $G'$  such that  $L(G') = L_1 \cdot L_2$ , as the grammar containing all the variables of  $G_1$  and  $G_2$ , all the terminals of  $G_1$  and  $G_2$ , with a new start symbol  $S$ . The new rules :

$$S \rightarrow S_1 \cdot S_2$$

followed by rules of  $G_1$  and rules of  $G_2$ .

So CFLs are closed under concatenation.



# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under Concatenation?**

Suppose  $L_1$  and  $L_2$  are CFLs. Is  $L = L_1 \cdot L_2$  also a CFL?

**Proof:** Suppose  $G_1$  and  $G_2$  be grammars such that  $L(G_1) = L_1$  and  $L(G_2) = L_2$ .

Suppose:

Rules of  $G_1$ :  $S_1 \rightarrow \dots$

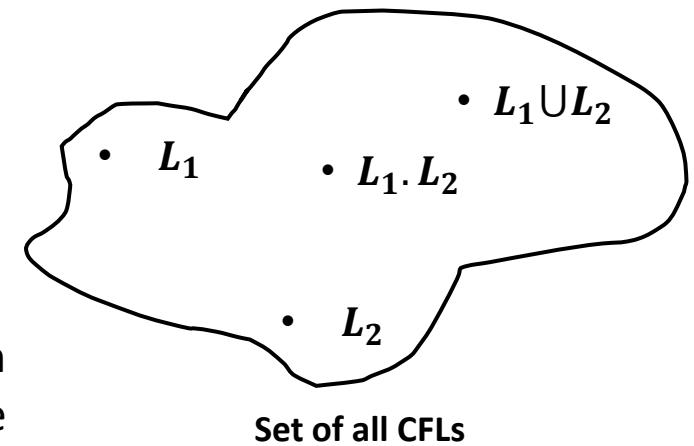
Rules of  $G_2$ :  $S_2 \rightarrow \dots$

Also suppose that the rules of  $G_1$  and  $G_2$  have different variables. Then define  $G'$  such that  $L(G') = L_1 \cdot L_2$ , as the grammar containing all the variables of  $G_1$  and  $G_2$ , all the terminals of  $G_1$  and  $G_2$ , with a new start symbol  $S$ . The new rules:

$$S \rightarrow S_1 \cdot S_2$$

followed by rules of  $G_1$  and rules of  $G_2$ .

So CFLs are closed under concatenation.



# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under Star?**

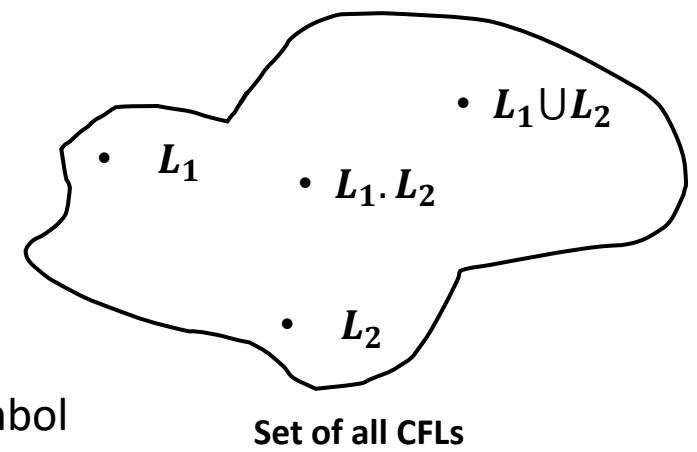
Suppose  $L$  is a CFL. Is  $L^*$  also a CFL?

**Proof:** Suppose  $G$  be a grammar such that  $L(G) = L_1$

Suppose:

Rules of  $G$ :  $S_1 \rightarrow \dots$

Then the grammar  $G'$  such that  $L(G') = L^*$  is the same as  $G$  with a new start symbol and the additional rules



# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under Star?**

Suppose  $L$  is a CFL. Is  $L^*$  also a CFL?

**Proof:** Suppose  $G$  be a grammar such that  $L(G) = L_1$

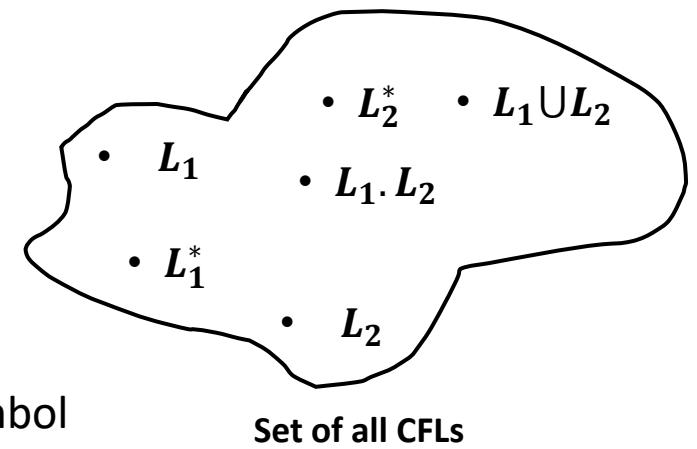
Suppose:

Rules of  $G$ :  $S_1 \rightarrow \dots$

Then the grammar  $G'$  such that  $L(G') = L^*$  is the same as  $G$  with a new start symbol and the additional rules

$$S \rightarrow S_1 S | \epsilon$$

So CFLs are closed under Star.



# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under intersection?**

Suppose  $L_1$  and  $L_2$  are CFLs. Is  $L = L_1 \cap L_2$  also a CFL?

**Proof:** We will prove that CFLs are NOT closed under intersection by using this simple counterexample. Let

# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under intersection?**

Suppose  $L_1$  and  $L_2$  are CFLs. Is  $L = L_1 \cap L_2$  also a CFL?

**Proof:** We will prove that CFLs are NOT closed under intersection by using this simple counterexample. Let

$$L_1 = \{0^n 1^n 2^m | m, n \geq 1\} \text{ and } L_2 = \{0^m 1^n 2^n | l, n \geq 1\}$$

# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under intersection?**

Suppose  $L_1$  and  $L_2$  are CFLs. Is  $L = L_1 \cap L_2$  also a CFL?

**Proof:** We will prove that CFLs are NOT closed under intersection by using this simple counterexample. Let

$$L_1 = \{0^n 1^n 2^m | m, n \geq 1\} \text{ and } L_2 = \{0^m 1^n 2^n | l, n \geq 1\}$$

Note that  $L_1, L_2 \in \text{CFL}$  – each of them are concatenation of two CFLs.

# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under intersection?**

Suppose  $L_1$  and  $L_2$  are CFLs. Is  $L = L_1 \cap L_2$  also a CFL?

**Proof:** We will prove that CFLs are NOT closed under intersection by using this simple counterexample. Let

$$L_1 = \{0^n 1^n 2^m | m, n \geq 1\} \text{ and } L_2 = \{0^m 1^n 2^n | l, n \geq 1\}$$

Note that  $L_1, L_2 \in \text{CFL}$  – each of them are concatenation of two CFLs.

E.g:  $L_1$  is a concatenation of  $\{0^n 1^n | n \geq 1\}$  and  $\{2^m | m \geq 1\}$  and the rules of the corresponding grammar are

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow 0A1|01 \\ B &\rightarrow 2B|2 \end{aligned}$$

What is  $L_1 \cap L_2$ ?

# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under intersection?**

Suppose  $L_1$  and  $L_2$  are CFLs. Is  $L = L_1 \cap L_2$  also a CFL?

**Proof:** We will prove that CFLs are NOT closed under intersection by using this simple counterexample. Let

$$L_1 = \{0^n 1^n 2^m | m, n \geq 1\} \text{ and } L_2 = \{0^m 1^n 2^n | m, n \geq 1\}$$

Note that  $L_1, L_2 \in \text{CFL}$  – each of them are concatenation of two CFLs. E.g:  $L_1$  is a concatenation of  $\{0^n 1^n | n \geq 1\}$  and  $\{2^m | m \geq 1\}$  and the rules of the corresponding grammar are

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow 0A1|01 \\ B &\rightarrow 2B|2 \end{aligned}$$

$$L_1 \cap L_2 = \{0^n 1^n 2^n | n \geq 1\} \text{ which is not a CFL.}$$

Hence **CFLs are NOT closed under intersection!**

# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under complementation?**

Suppose  $L$  is a CFL. Is  $\bar{L}$  also a CFL?

**Proof:** ??????????

# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under complementation?**

Suppose  $L$  is a CFL. Is  $\bar{L}$  also a CFL?

**Proof:** Let us assume that CFLs are closed under complementation. Then if  $L_1$  and  $L_2$  are context free, then  $\bar{L}_1$  and  $\bar{L}_2$  are also context free. This would imply that

$$\bar{L}_1 \cup \bar{L}_2 \in \text{CFL}$$

# Closure properties of CFL

**Q:** Is the set of all CFLs **closed under complementation?**

Suppose  $L$  is a CFL. Is  $\bar{L}$  also a CFL?

**Proof:** Let us assume that CFLs are closed under complementation. Then if  $L_1$  and  $L_2$  are context free, then  $\bar{L}_1$  and  $\bar{L}_2$  are also context free. This would imply that

$$\bar{L}_1 \cup \bar{L}_2 \in \mathbf{CFL}$$

Finally, this would imply  $\overline{\bar{L}_1 \cup \bar{L}_2} \in \mathbf{CFL}$ . However,

$$L_1 \cap L_2 = \overline{\bar{L}_1 \cup \bar{L}_2}$$

But this would imply  $L_1 \cap L_2 \in \mathbf{CFL}$ , which is a contradiction.

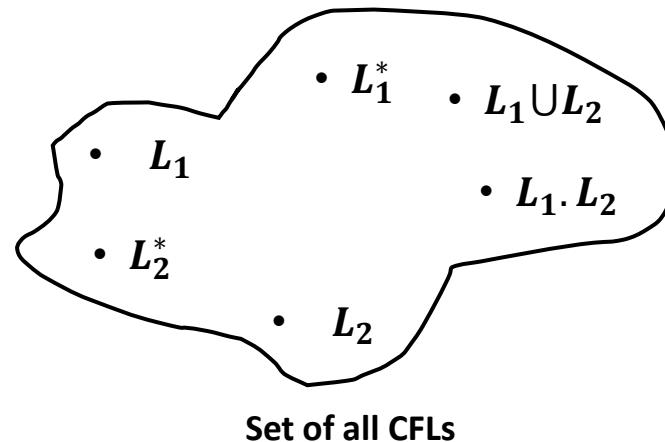
Thus CFLs are NOT closed under complementation.

# Closure properties of CFL

Recall that for Regular languages:

RLs are closed under

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



For CFLs:

CFLs are closed under

- **Union**
- **Star**
- **Concatenation**

CFLs are NOT closed under

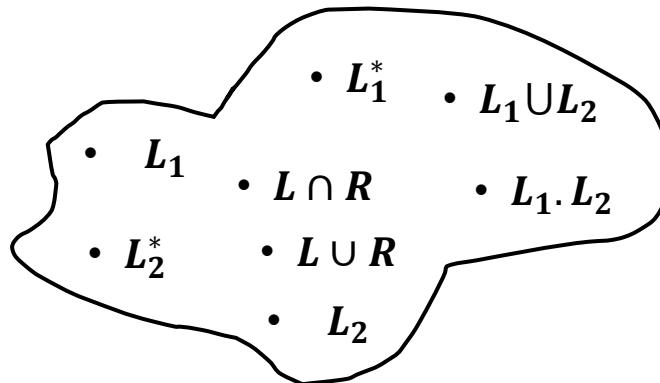
- **Complementation**
- **Intersection**

# Closure properties of CFL

Recall that for Regular languages:

RLs are closed under

- Union
- Intersection
- Star
- Complement
- Concatenation



Set of all CFLs

If  $L$  is a CFL and  $R$  is a regular language then  
 $L \cap R$  is a CFL.  
 $L \cup R$  is a CFL.

For CFLs:

CFLs are closed under

- Union
- Star
- Concatenation

CFLs are NOT closed under

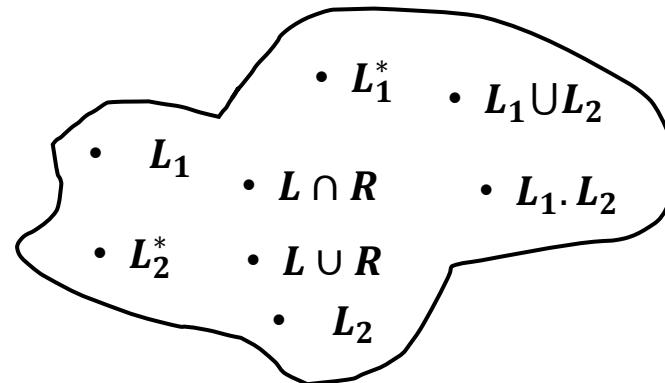
- Complementation
- Intersection

# Closure properties of CFL

Recall that for Regular languages:

RLs are closed under

- Union
- Intersection
- Star
- Complement
- Concatenation



Set of all CFLs

If  $L$  is a CFL and  $R$  is a regular language then  
 $L \cap R$  is a CFL.  
 $L \cup R$  is a CFL.

For CFLs:

CFLs are closed under

- Union
- Star
- Concatenation

CFLs are NOT closed under

- Complementation
- Intersection

Proof intuition: Construct a **Product PDA**. If states of PDA  $P: (q_1, q_2, \dots, q_m)$  and DFA  $D: (d_1, d_2, \dots, d_n)$ , then states of **Product PDA X**:

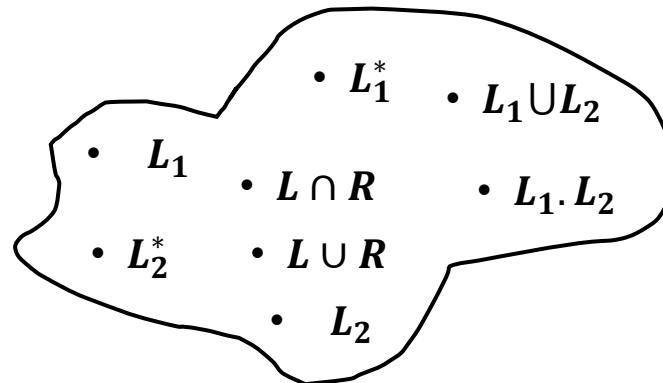
$$Q = \{(q_1, d_1), (q_1, d_2), \dots, (q_m, d_n)\}$$

# Closure properties of CFL

Recall that for Regular languages:

RL are closed under

- Union
- Intersection
- Star
- Complement
- Concatenation



Set of all CFLs

If  $L$  is a CFL and  $R$  is a regular language then  
 $L \cap R$  is a CFL.  
 $L \cup R$  is a CFL.

For CFLs:

CFLs are closed under

- Union
- Star
- Concatenation

CFLs are NOT closed under

- Complementation
- Intersection

Proof intuition: Construct a **Product PDA**. If states of PDA  $P: (q_1, q_2, \dots, q_m)$  and DFA  $D: (d_1, d_2, \dots, d_n)$ , then states of **Product PDA X**:

$$Q = \{(q_1, d_1), (q_1, d_2), \dots, (q_m, d_n)\}$$

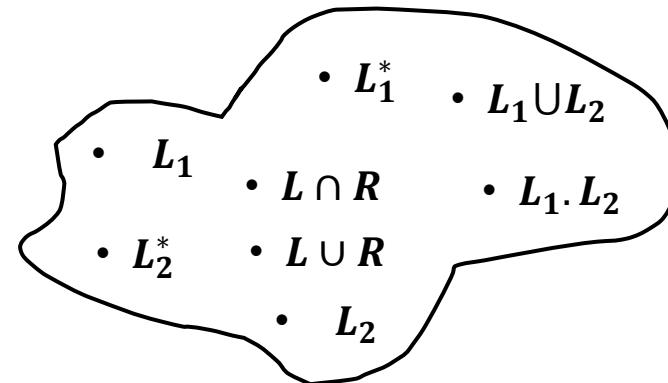
If  $\delta(q_i, a, b) = (q_j, c)$  and  $\delta(d_k, a) = d_l$ , then for  $X$ :  $\delta((q_i, d_k), a, b) = ((q_j, d_l), c)$ . So  $X$  is a PDA.

# Closure properties of CFL

Recall that for Regular languages:

RL are closed under

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



Set of all CFLs

If  $L$  is a CFL and  $R$  is a regular language then  
 $L \cap R$  is a CFL.  
 $L \cup R$  is a CFL.

For CFLs:

CFLs are closed under

- **Union**
- **Star**
- **Concatenation**

CFLs are NOT closed under

- **Complementation**
- **Intersection**

Proof intuition: Construct a **Product PDA**. If states of PDA  $P: (q_1, q_2, \dots, q_m)$  and DFA  $D: (d_1, d_2, \dots, d_n)$ , then states of **Product PDA X**:

$$Q = \{(q_1, d_1), (q_1, d_2), \dots, (q_m, d_n)\}$$

If  $\delta(q_i, a, b) = (q_j, c)$  and  $\delta(d_k, a) = d_l$ , then for  $X$ :  $\delta((q_i, d_k), a, b) = ((q_j, d_l), c)$ . So  $X$  is a PDA.

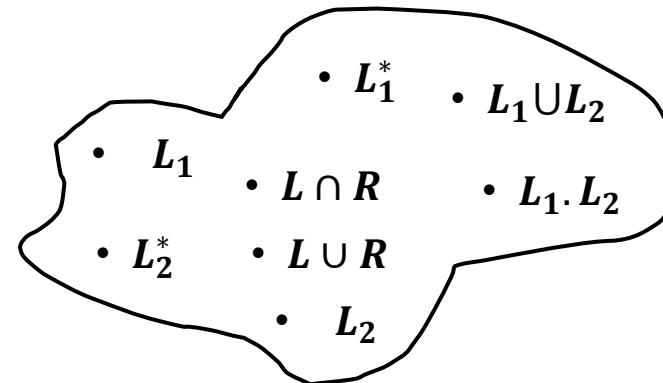
$L(X) = L(P) \cap L(R)$  if the final state, say  $(q_r, d_s)$  is such that  $q_r$  and  $d_s$  are both final states of  $P$  AND  $D$  respectively

# Closure properties of CFL

Recall that for Regular languages:

RL are closed under

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



Set of all CFLs

If  $L$  is a CFL and  $R$  is a regular language then  
 $L \cap R$  is a CFL.  
 $L \cup R$  is a CFL.

For CFLs:

CFLs are closed under

- **Union**
- **Star**
- **Concatenation**

CFLs are NOT closed under

- **Complementation**
- **Intersection**

Proof intuition: Construct a **Product PDA**. If states of PDA  $P: (q_1, q_2, \dots, q_m)$  and DFA  $D: (d_1, d_2, \dots, d_n)$ , then states of **Product PDA X**:

$$Q = \{(q_1, d_1), (q_1, d_2), \dots, (q_m, d_n)\}$$

If  $\delta(q_i, a, b) = (q_j, c)$  and  $\delta(d_k, a) = d_l$ , then for  $X$ :  $\delta((q_i, d_k), a, b) = ((q_j, d_l), c)$ . So  $X$  is a PDA.

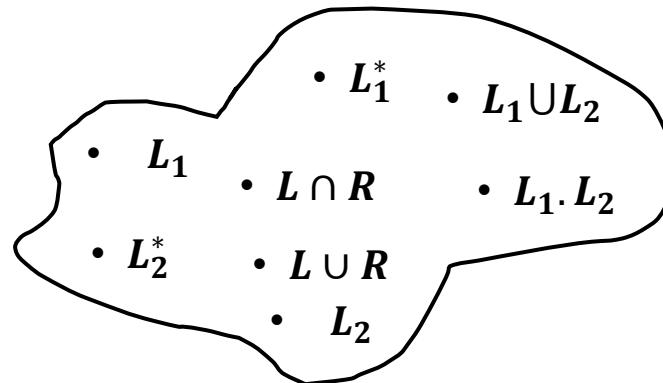
$L(X) = L(P) \cup L(R)$  if the final state, say  $(q_r, d_s)$  is such that **EITHER**  $q_r$  or  $d_s$  are final states of  $P$  OR  $D$  respectively

# Closure properties of CFL

Recall that for Regular languages:

RL are closed under

- Union
- Intersection
- Star
- Complement
- Concatenation



If  $L$  is a CFL and  $R$  is a regular language then  
 $L \cap R$  is a CFL.  
 $L \cup R$  is a CFL.

For CFLs:

CFLs are closed under

- Union
- Star
- Concatenation

CFLs are NOT closed under

- Complementation
- Intersection

For DCFLs

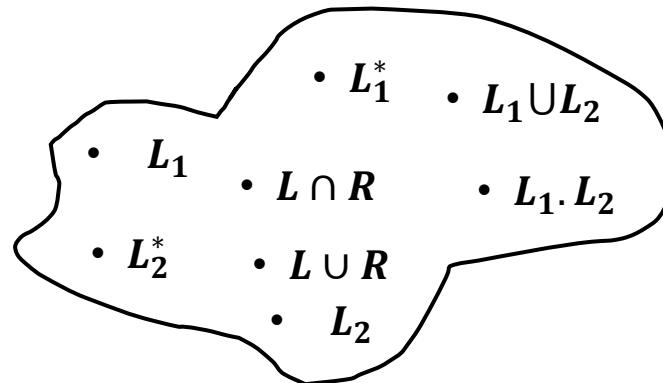
- NOT closed Union - construct a counter example
- Closed under complementation - construct a “toggled” DPDA (a bit of extra work is needed to take care of the dead states)
- NOT closed under intersection - use the first two to prove this

# Closure properties of CFL

Recall that for Regular languages:

RL are closed under

- Union
- Intersection
- Star
- Complement
- Concatenation



If  $L$  is a CFL and  $R$  is a regular language then  
 $L \cap R$  is a CFL.  
 $L \cup R$  is a CFL.

For CFLs:

CFLs are closed under

- Union
- Star
- Concatenation

CFLs are NOT closed under

- Complementation
- Intersection

For DCFLs

- NOT closed Union
- Closed under complementation
- NOT closed under intersection

Next lecture:

- Turing Machine

**Thank You!**

# CS 302.1 - Automata Theory

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

**Pumping Lemma for CFL:** If  $L$  is Context Free, then there exists  $p > 0$  (pumping length), such that, for any  $w \in L$  of length  $|w| \geq p$ ,  $w$  can be split into five parts, i.e.  $w = uvxyz$  satisfying the following conditions:

- $|vy| \geq 1$
- $|vxy| \leq p$
- $uv^i xy^i z \in L, \forall i \geq 0$

## Closure properties of CFLs

CFLs are closed under

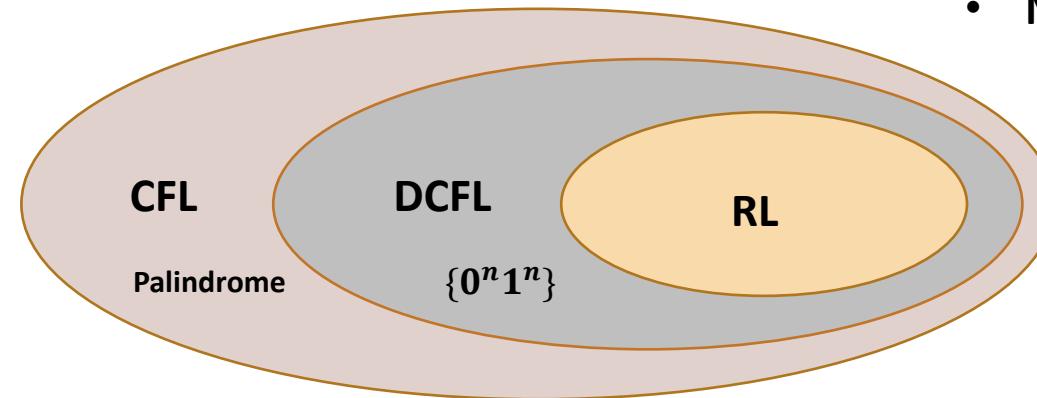
- Union
- Star
- Concatenation

CFLs are NOT closed under

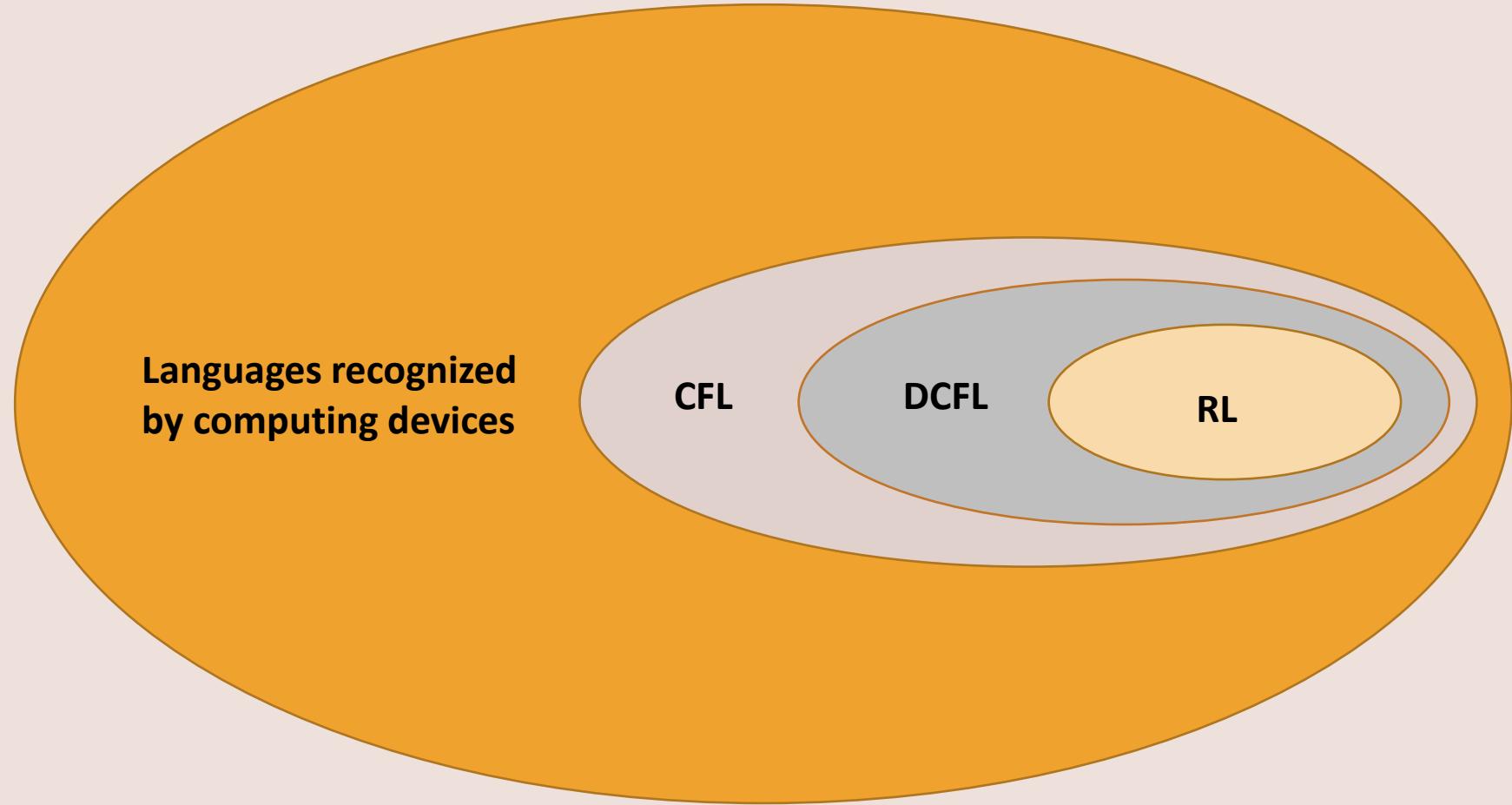
- Complementation
- Intersection

## For DCFLs

- NOT closed Union
- Closed under complementation
- NOT closed under intersection

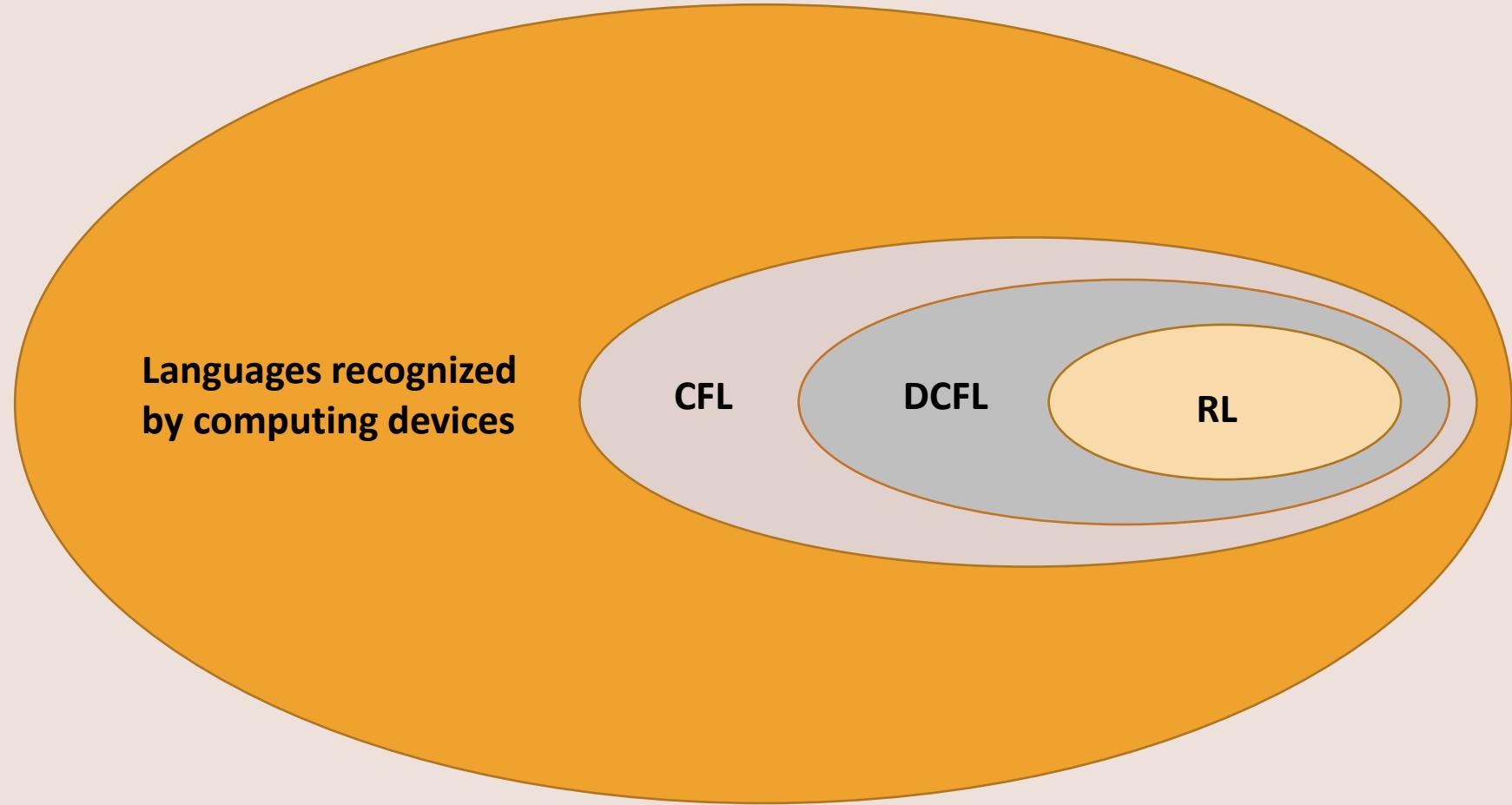


The set of all languages



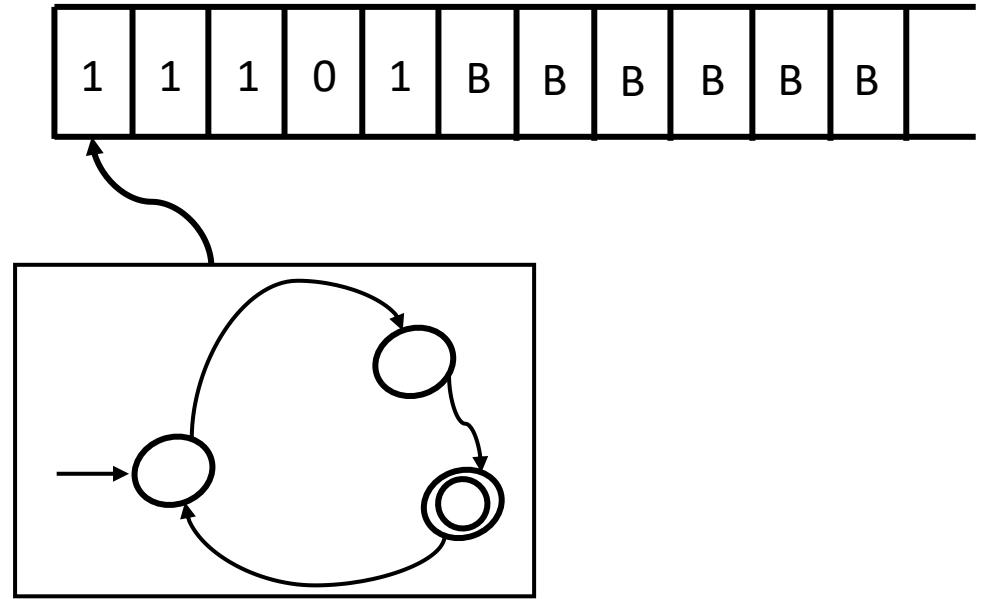
**Which languages lie here?**

**The set of all languages**



# Turing Machines

- A Turing machine is a FSM that has access to a infinite tape as its memory.
- The infinite tape contains in it, the input string followed by Blanks (indicated by B)
- The Turing machine can both read from the tape and write in it – one cell at a time, using a Read/Write head.
- The Read/Write head can move to the Left or to the Right – again one cell at a time.

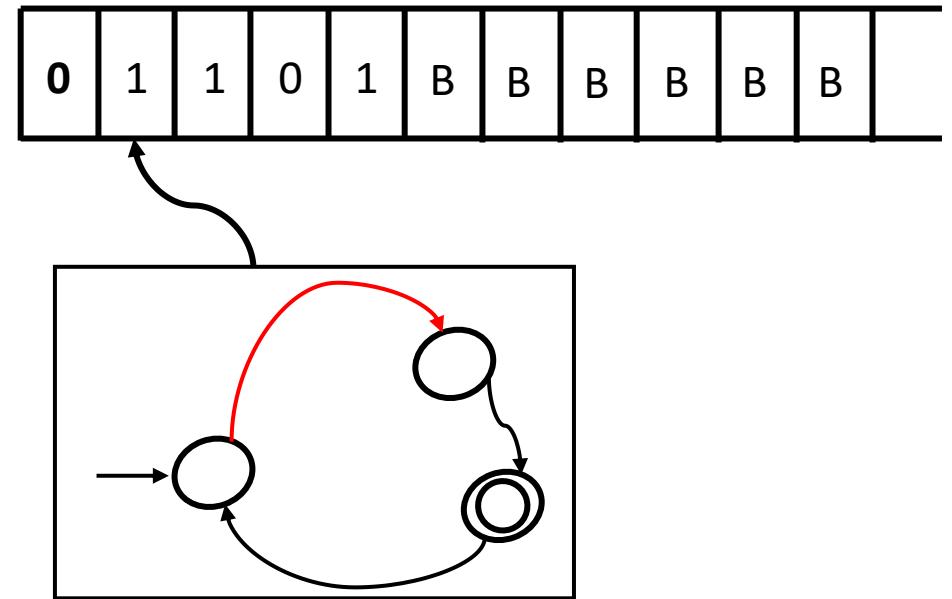


## At each step, the TM:

- Replaces the symbol of the current cell in the tape with a new one
- Transitions from one state to another (or remains where it was)
- The Read/Write head moves either to the Left or to the Right

# Turing Machines

- A Turing machine is an FSM that has access to a infinite tape as its memory.
- The infinite tape contains in it, the input string followed by Blanks (indicated by B)
- The Turing machine can both read from the tape and write in it – one cell at a time, using a Read/Write head.
- The Read/Write head can move to the Left or to the Right – again one cell at a time.

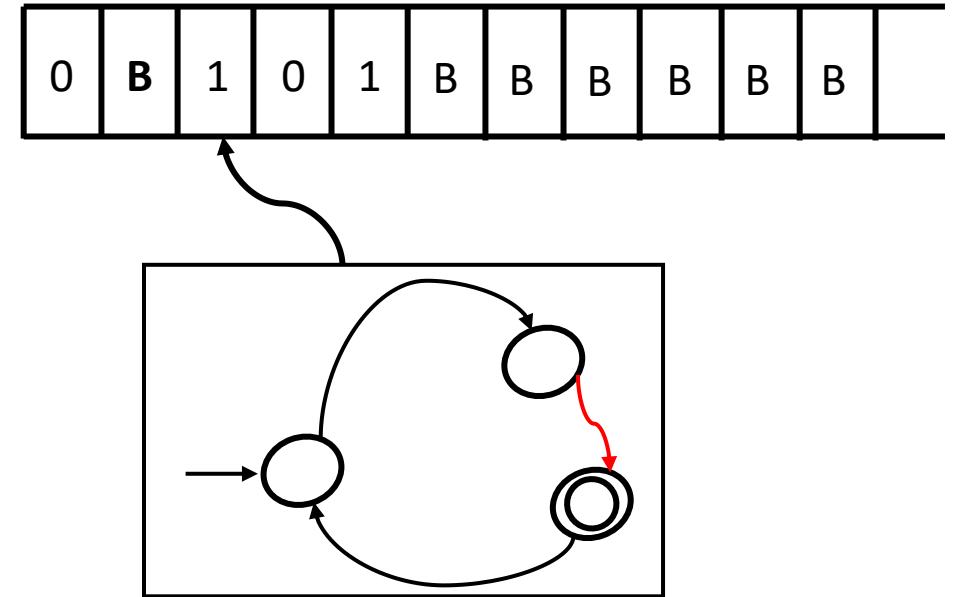


## At each step, the TM:

- Replaces the symbol of the current cell in the tape with a new one
- Transitions from one state to another (or remains where it was)
- The Read/Write head moves either to the Left or to the Right

# Turing Machines

- A Turing machine is an FSM that has access to a infinite tape as its memory.
- The infinite tape contains in it, the input string followed by Blanks (indicated by B)
- The Turing machine can both read from the tape and write in it – one cell at a time, using a Read/Write head.
- The Read/Write head can move to the Left or to the Right – again one cell at a time.

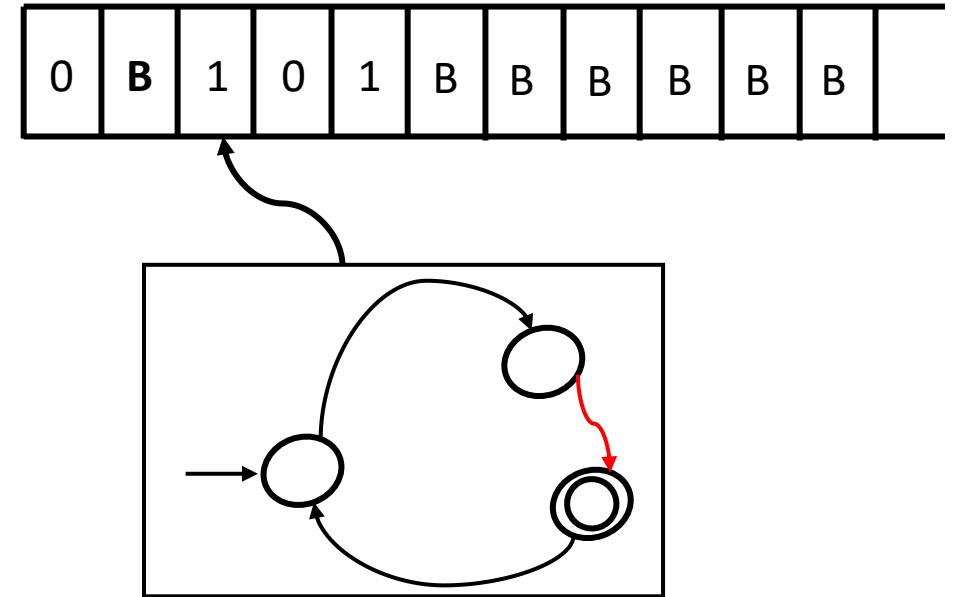


## At each step, the TM:

- Replaces the symbol of the current cell in the tape with a new one
  - Transitions from one state to another (or remains where it was)
  - The Read/Write head moves either to the Left or to the Right
- 
- Whenever the computation reaches an accept/reject state – the TM accepts or rejects the input string (halts)

# Turing Machines

- A Turing machine is an FSM that has access to a infinite tape as its memory.
- The infinite tape contains in it, the input string followed by Blanks (indicated by B)
- The Turing machine can both read from the tape and write in it – one cell at a time, using a Read/Write head.
- The Read/Write head can move to the Left or to the Right – again one cell at a time.

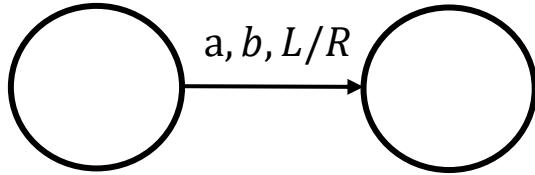


## At each step, the TM:

- Replaces the symbol of the current cell in the tape with a new one
- Transitions from one state to another (or remains where it was)
- The Read/Write head moves either to the Left or to the Right
- Whenever the computation reaches an accept/reject state – the TM accepts or rejects the input string (halts)

- In a way these “added features” give TMs their power. (eg: ability to write on the tape)
- Notice: acceptance/rejection of a run is not tied to the input.
- Auxiliary computation can be performed – as much as needed, even when the input string has been scanned

# Turing Machines



Transition  $a, b, L/R$ : Read  $a$  from the tape, replace with  $b$  and move  $L/R$



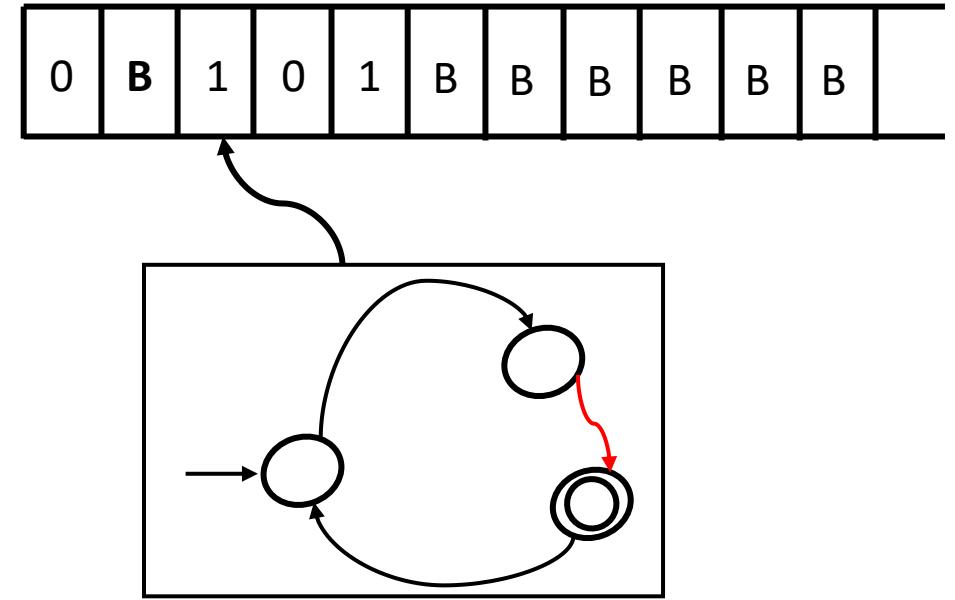
Accept state



Reject state

**TM may never halt – it may loop forever**

TM halts and **accepts/rejects** on reaching these states

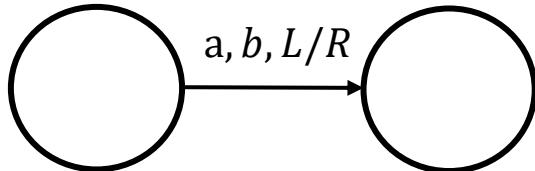


## At each step, the TM:

- Replaces the symbol of the current cell in the tape with a new one
- Transitions from one state to another (or remains where it was)
- The Read/Write head moves either to the Left or to the Right
- Whenever the computation reaches an accept/reject state – the TM accepts or rejects the input string (halts)

- In a way these “added features” give TMs their power. (eg: ability to write on the tape)
- Notice: acceptance/rejection of a run is not tied to the input.
- Auxiliary computation can be performed – as much as needed, even when the input string has been scanned

# Turing Machines



Transition  $a, b, L/R$ : Read  $a$  from the tape, replace with  $b$  and move  $L/R$

So, given a TM  $M$  and an input  $\omega$ ,

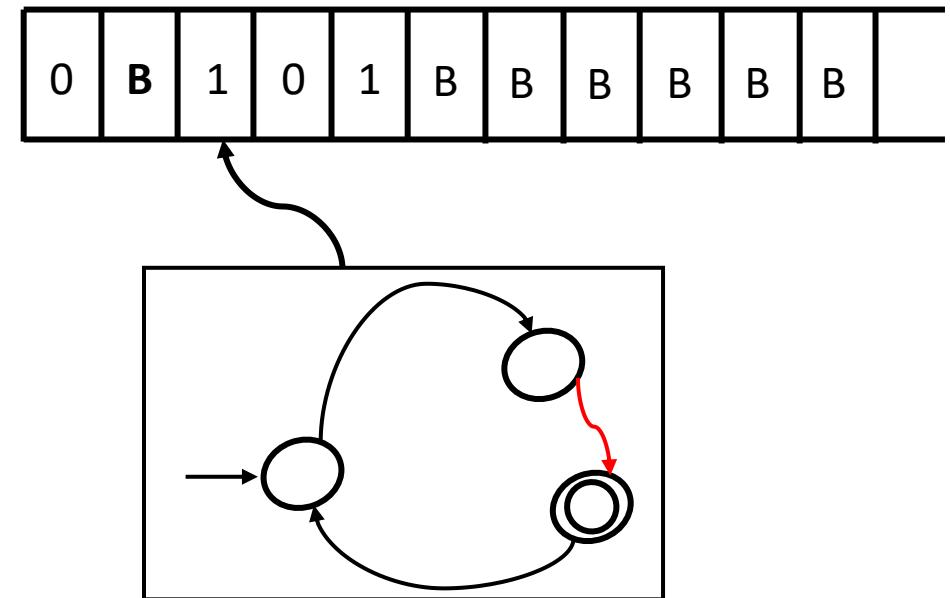
$M(\omega)$  accepts if  $\omega \in L(M)$

$M(\omega)$  rejects if  $\omega \notin L(M)$

$M(\omega)$  runs infinitely if  $\omega \notin L(M)$



} TM halts and **accepts/rejects** on reaching these states



## At each step, the TM:

- Replaces the symbol of the current cell in the tape with a new one
- Transitions from one state to another (or remains where it was)
- The Read/Write head moves either to the Left or to the Right
- Whenever the computation reaches an accept/reject state – the TM accepts or rejects the input string (halts)

- In a way these “added features” give TMs their power. (eg: ability to write on the tape)
- Notice: acceptance/rejection of a run is not tied to the input.
- Auxiliary computation can be performed – as much as needed, even when the input string has been scanned

# Turing Machines

Turing machines are named after **Alan Turing**. In 1936, gave a negative answer to Hilbert's *Entscheidungsproblem* (Decision problem) – *Are all decision problems decidable?*

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO  
THE ENTSCHEIDUNGSPROBLEM

*By A. M. TURING.*

[Received 28 May, 1936.—Read 12 November, 1936.]

The “computable” numbers may be described briefly as the real  
numbers whose expansions as a decimal are calculable by finite means



Turing claimed that anything “humanly computable” can be simulated by his machines

# Turing Machines

Turing machines are named after **Alan Turing**. In 1936, gave a negative answer to Hilbert's *Entscheidungsproblem* (Decision problem) – *Are all decision problems decidable?*

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO  
THE ENTSCHEIDUNGSPROBLEM

*By A. M. TURING.*

[Received 28 May, 1936.—Read 12 November, 1936.]

The “computable” numbers may be described briefly as the real

numbers whose expansions as a decimal are calculable by finite means



Turing claimed that anything “humanly computable” can be simulated by his machines

- Turing assumed that the human brain to be a finite state machine with a finite number of states
- Consider such a human being working on a problem with a notebook, pencil and an eraser.
- The pages of the notebook are laid out on the tape – each cell consists of one page, with a finite amount of information.
- Whatever the human being does with the notebook, can be simulated on the TM: reading, writing, erasing (writing a blank), moving left or right to a new page etc.

# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.

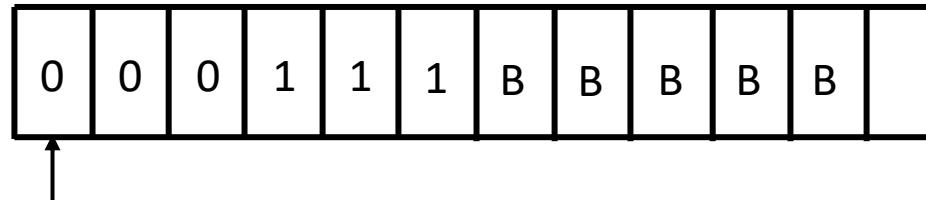
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



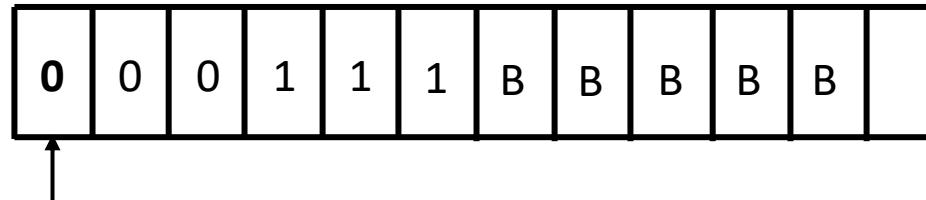
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



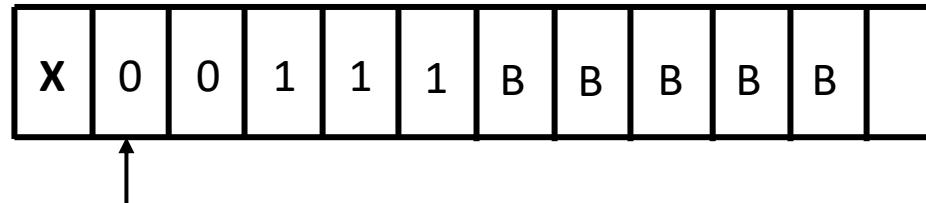
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



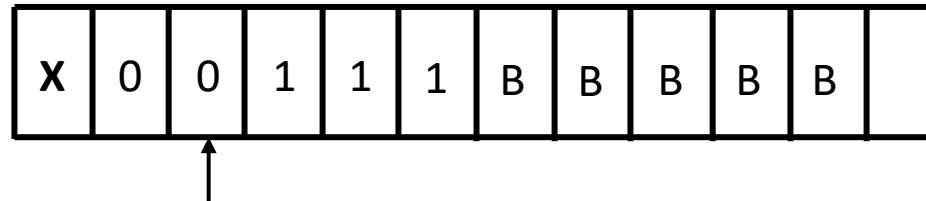
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



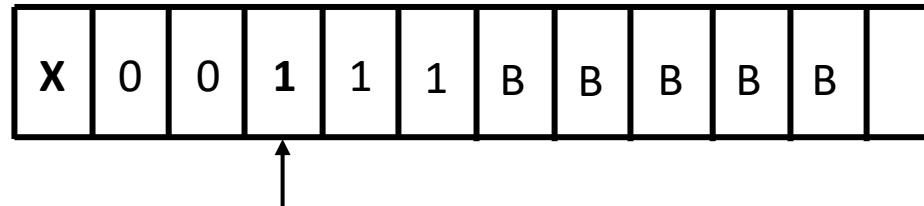
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



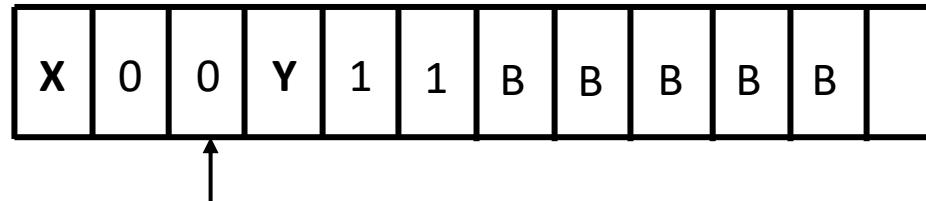
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



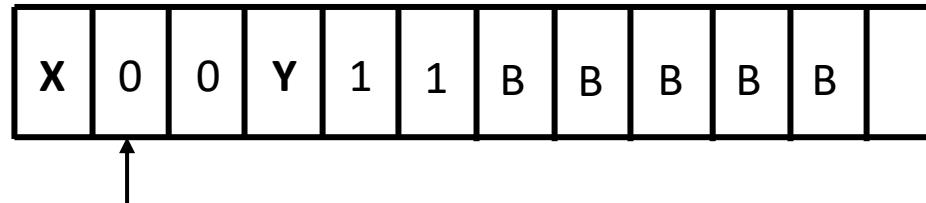
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



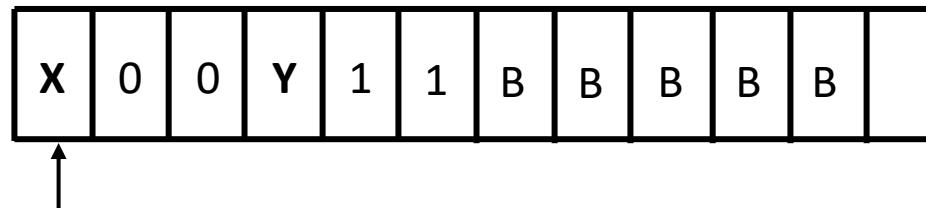
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



While moving left, when an  $X$  is encountered, the head should move right until the next 0 to be marked is encountered  $\Rightarrow$  We need rules like  $(X, X, R)$

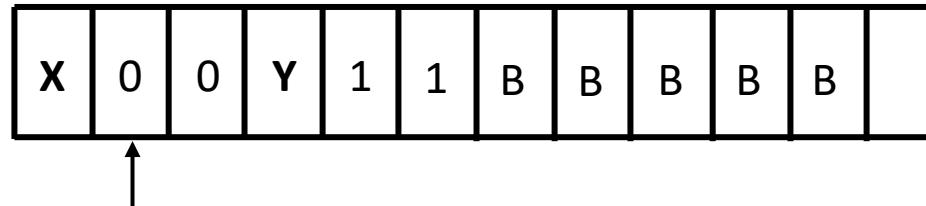
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



While moving left, when an  $X$  is encountered, the head should move right until the next 0 to be marked is encountered  $\Rightarrow$  We need rules like  $(X, X, R)$

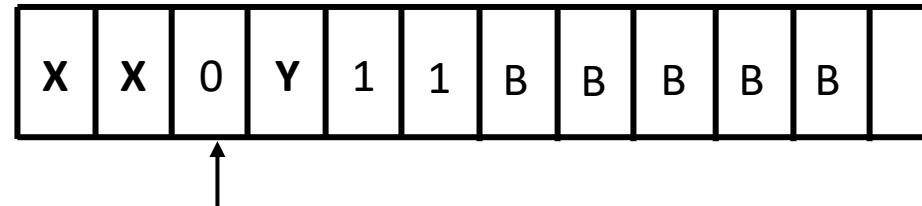
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



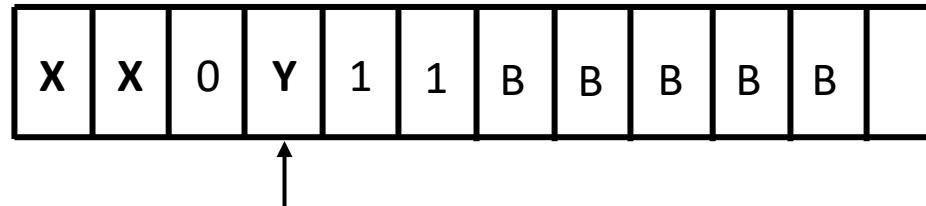
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



While moving right, when a  $Y$  is encountered, the head should move right as that's where the next 1 to be marked is  
⇒ We need rules like  $(Y, Y, R)$

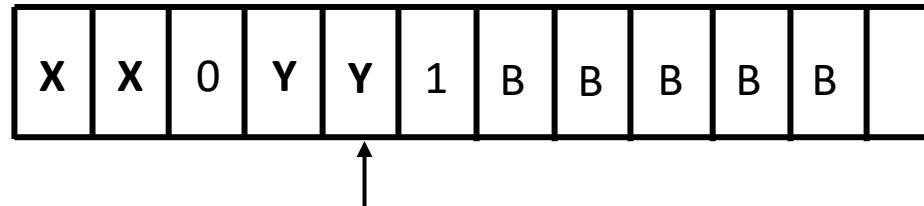
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



While moving left, when a  $Y$  is encountered, the head should keep moving left as those 1's have been marked already  $\Rightarrow$  We need rules like  $(Y, Y, L)$

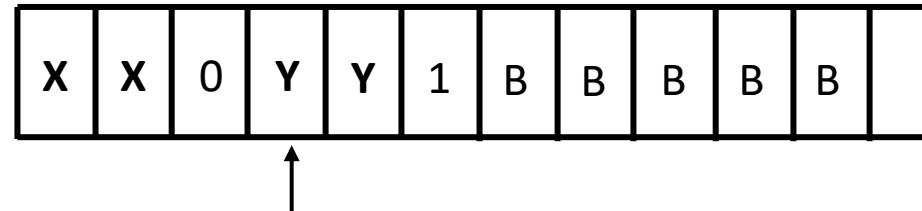
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



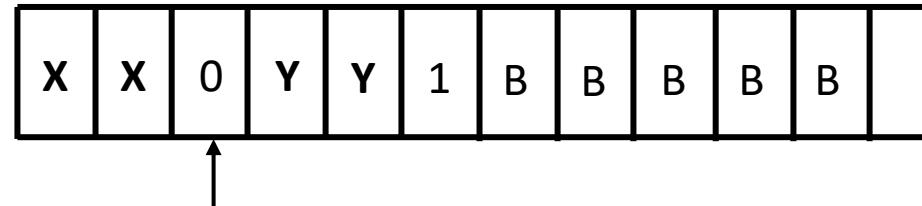
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



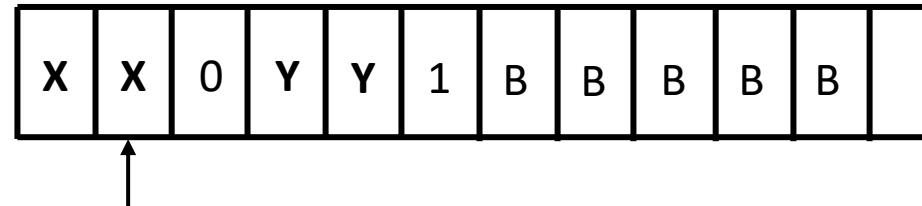
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



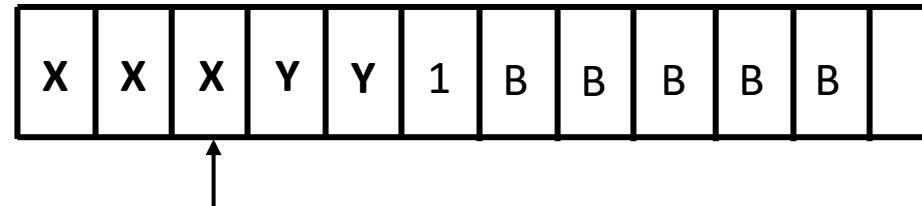
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



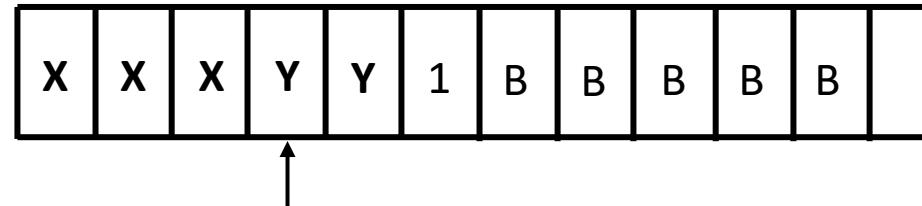
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



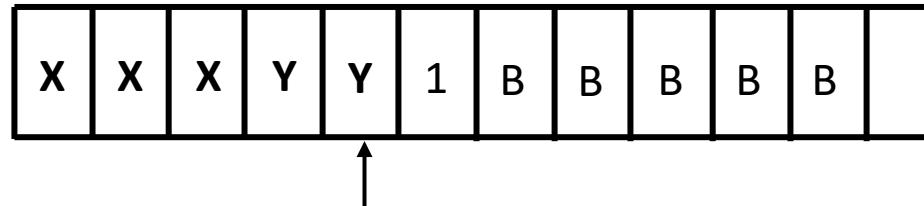
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



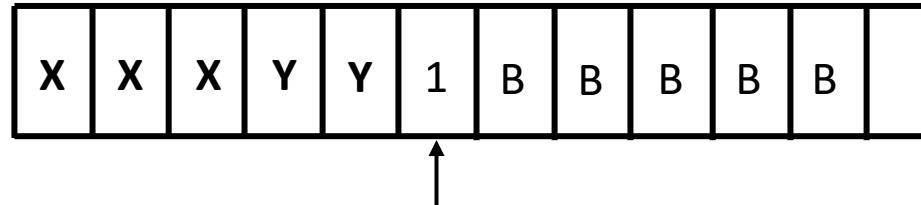
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



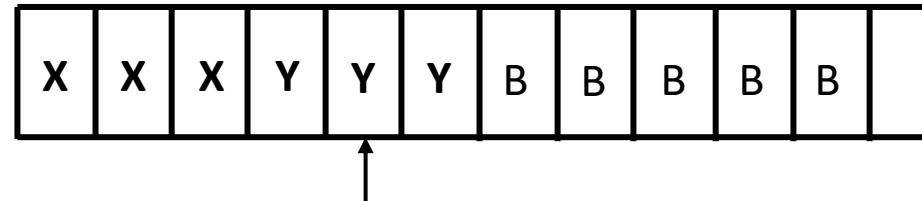
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



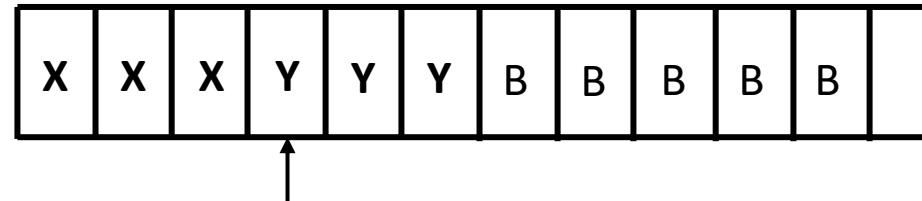
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



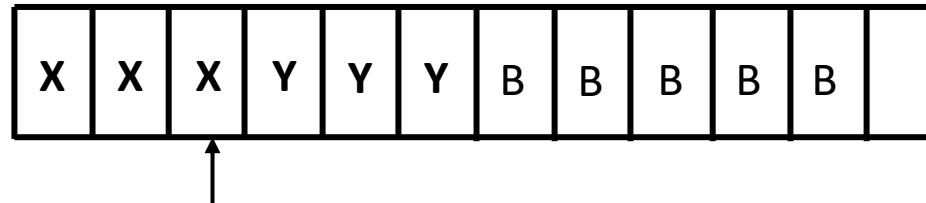
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



At this stage the head should move right to look for the next 0 to mark, but finds Y

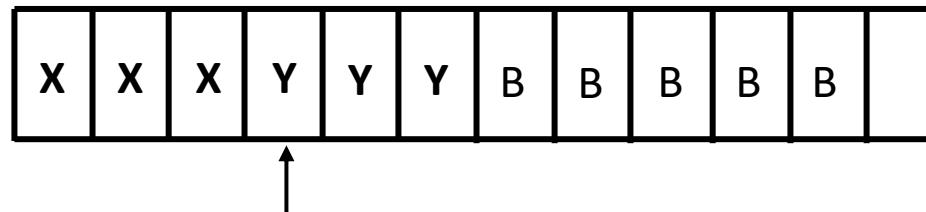
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



At this stage the head should move right and looks for the next 0 to mark, but finds  $Y$

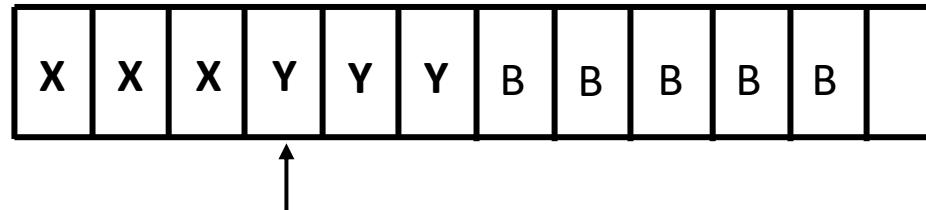
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



The head keeps moving right until it finds a  $B$

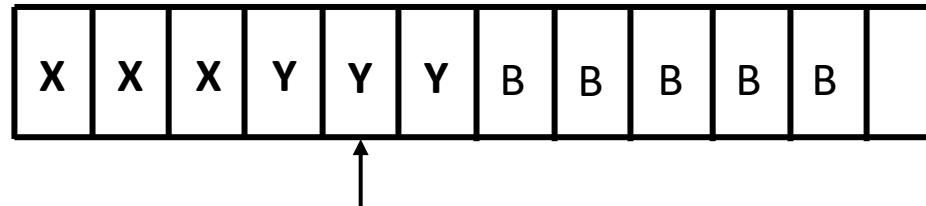
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



The head keeps moving right until it finds a  $B$

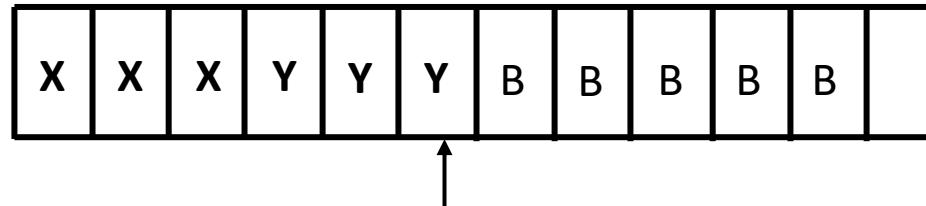
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



The head keeps moving right until it finds a  $B$

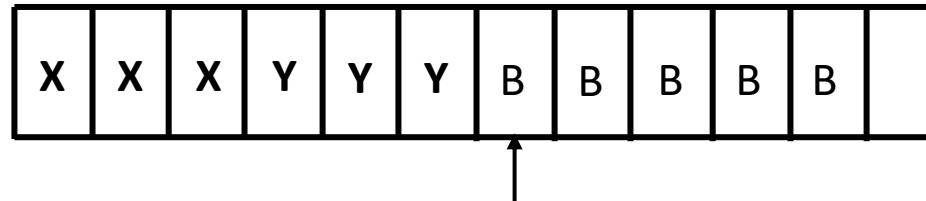
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if there are no 0's or 1's left in the tape.



The head keeps moving right until it finds a  $B$

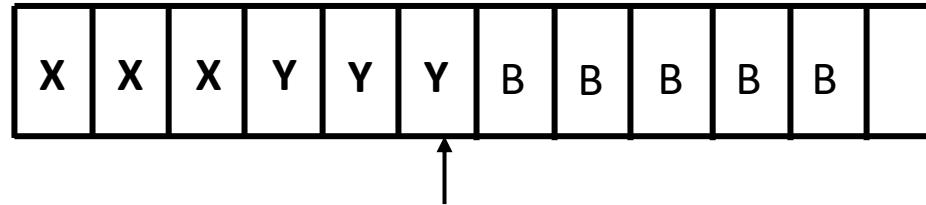
# Turing Machines

**Example:** Let  $L = \{0^n 1^n \mid n \geq 1\}$

We will try to develop the basic idea in designing the Turing Machine for this language. Note that  $L = CFL$ .

**Idea:** An accepting run of a TM for  $L$  could look something like this:

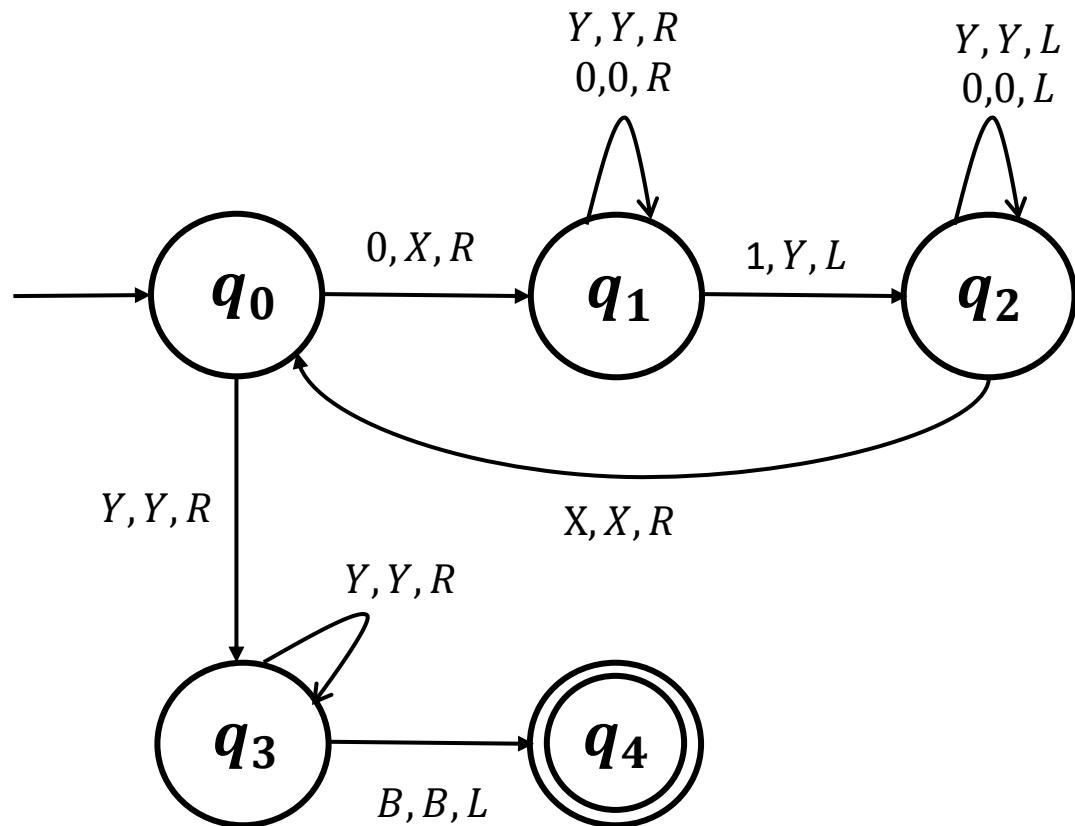
- Mark the first 0 (by replacing it with some special symbol say  $X$ )
- Continue to move to the right until the first 1 is encountered.
- Mark the first 1 (by replacing it with some special symbol say  $Y$ )
- Continue to move left until you encounter the second 0 (its to the right of the  $X$  you had marked before)
- Continue to move right until you encounter a second 1 (its to the right of the  $Y$  you had marked before)
- Go on repeating this, until there all the 0's and 1's have been marked (with  $X$  and  $Y$ ).
- Accept if are no 0's or 1's left in the tape.



This is when the TM decides to accept the input string.

# Turing Machines

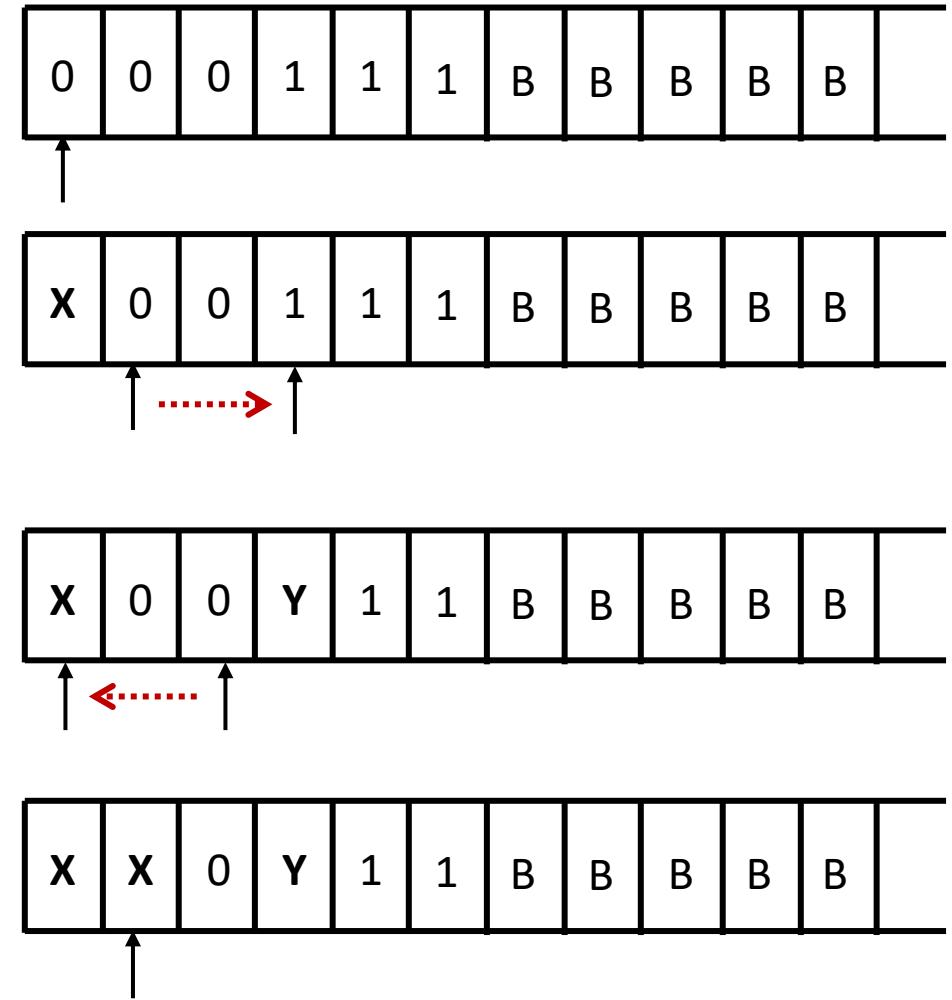
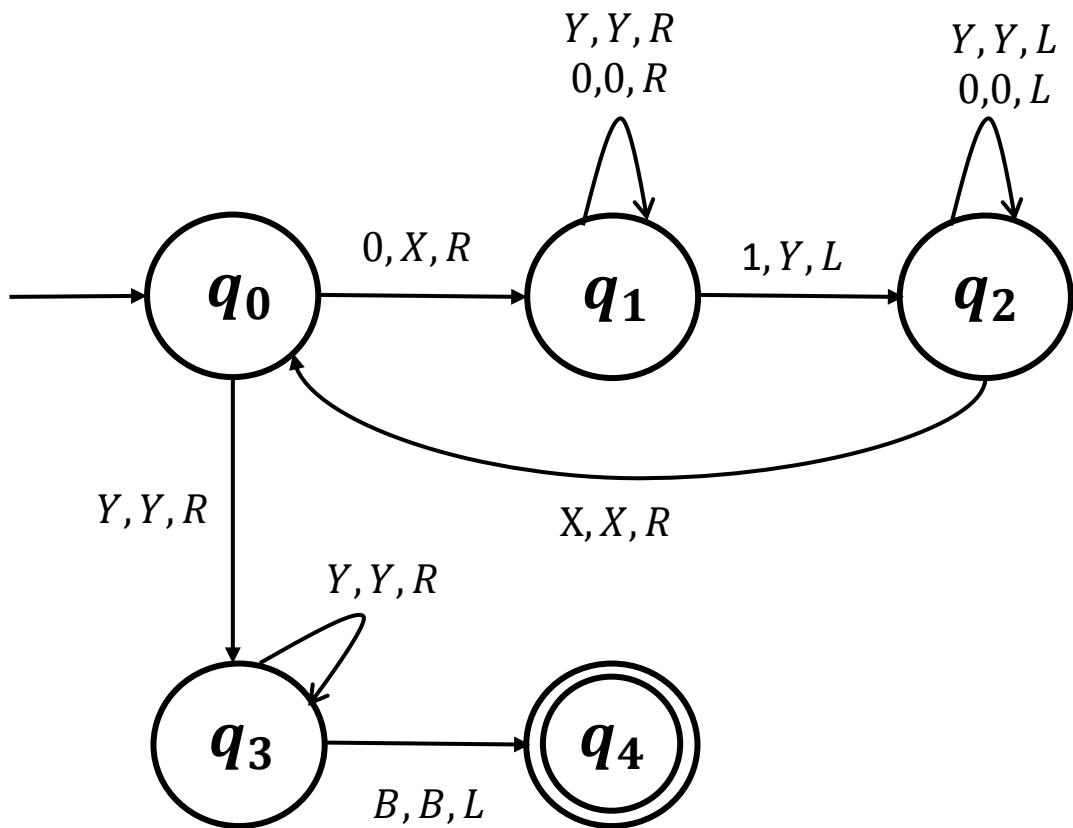
Example: Let  $L = \{0^n 1^n \mid n \geq 1\}$



All missing transitions lead to the reject state and the input is rejected when this state is reached.

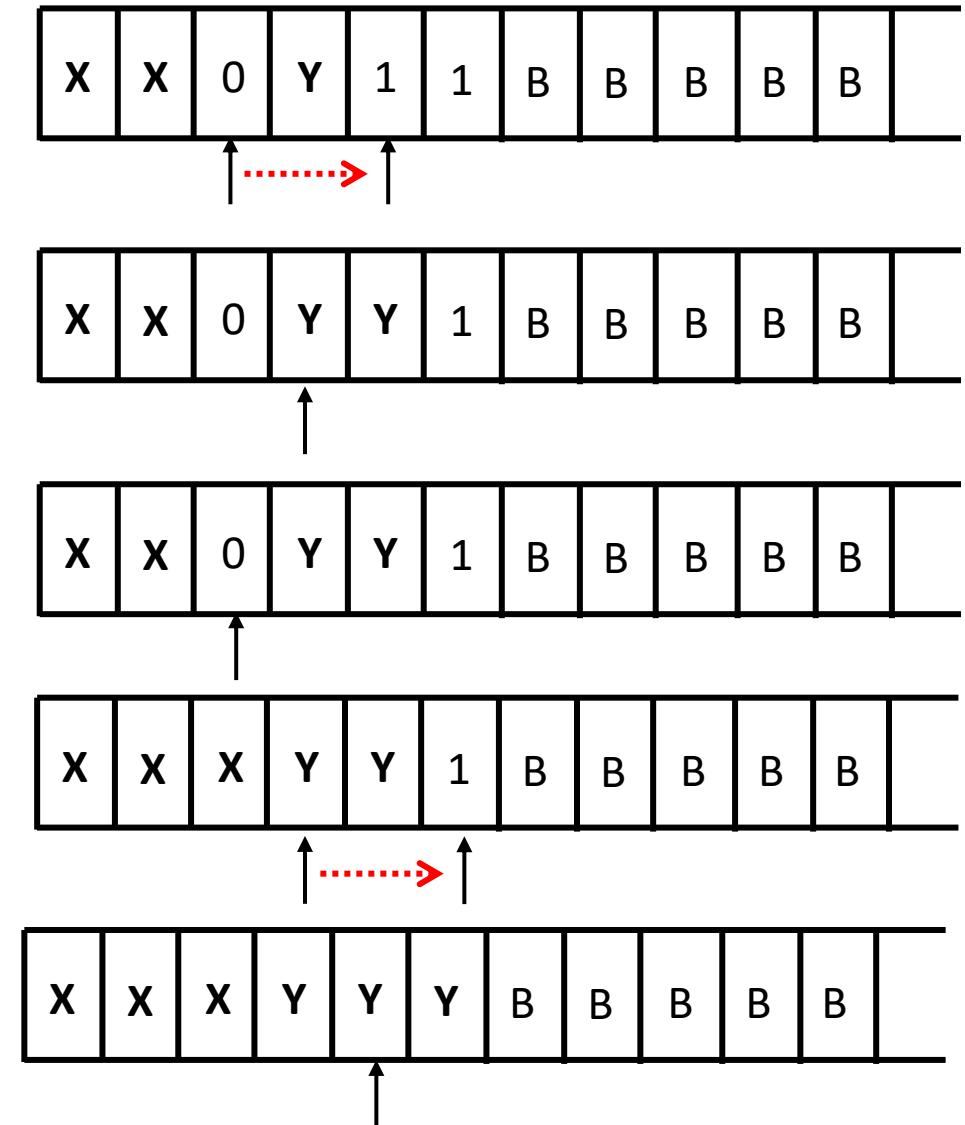
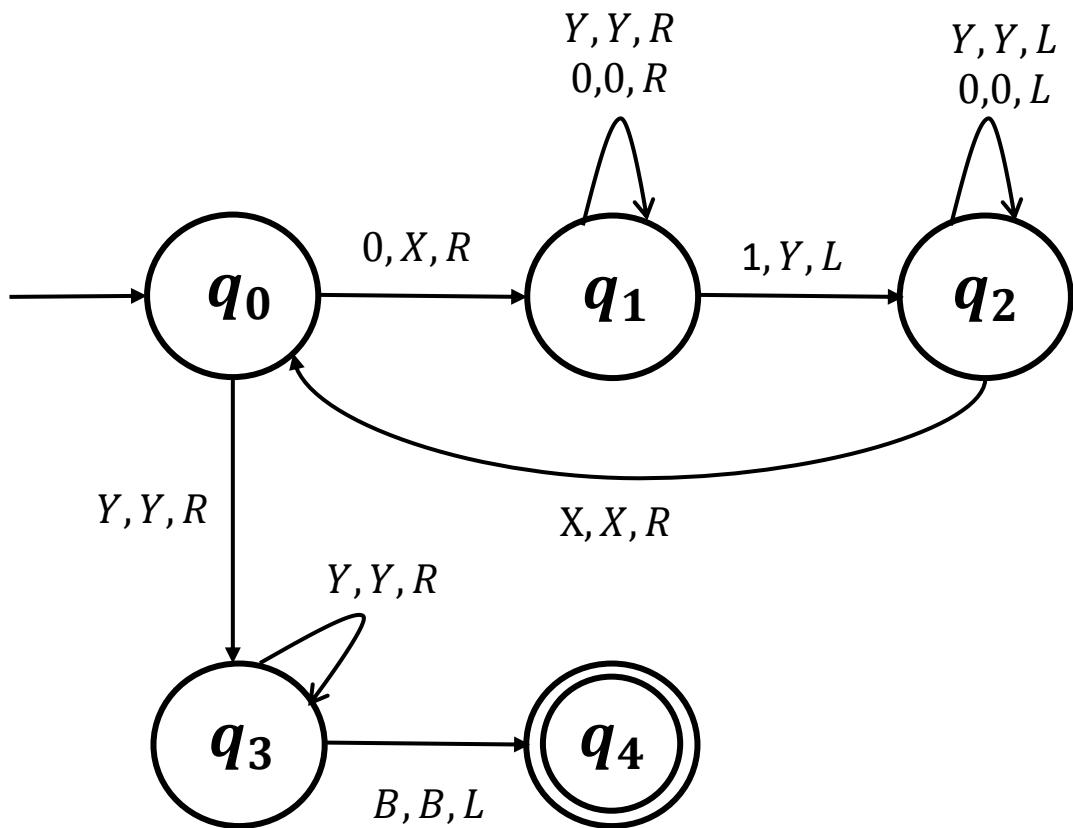
# Turing Machines

Example: Let  $L = \{0^n 1^n \mid n \geq 1\}$



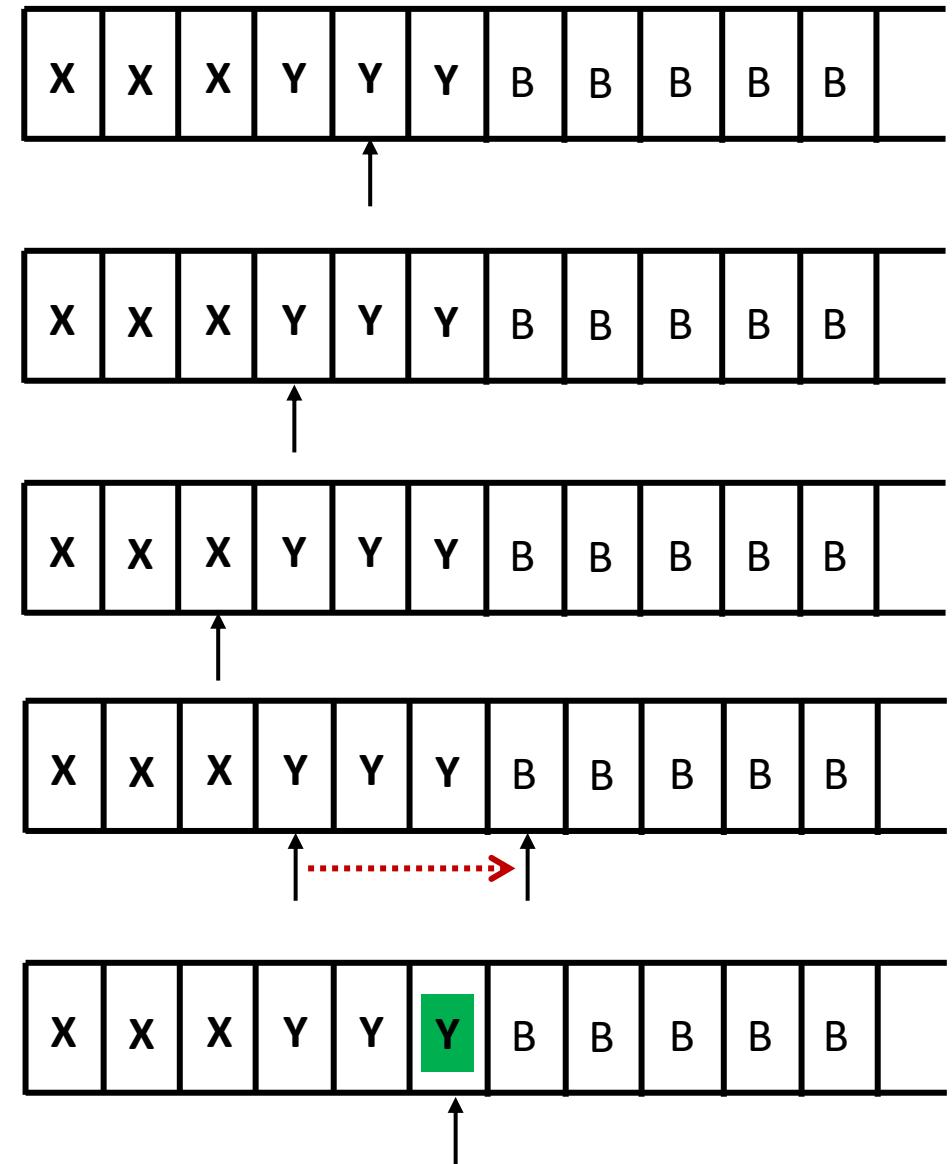
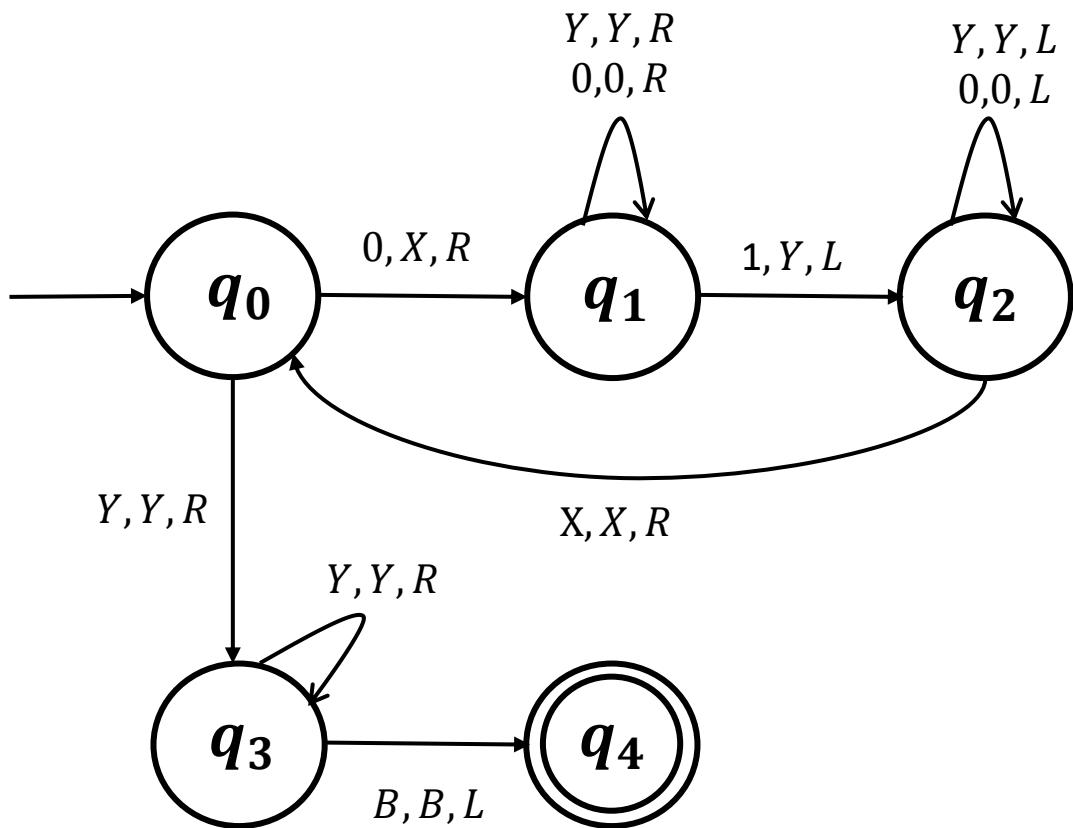
# Turing Machines

Example: Let  $L = \{0^n 1^n \mid n \geq 1\}$



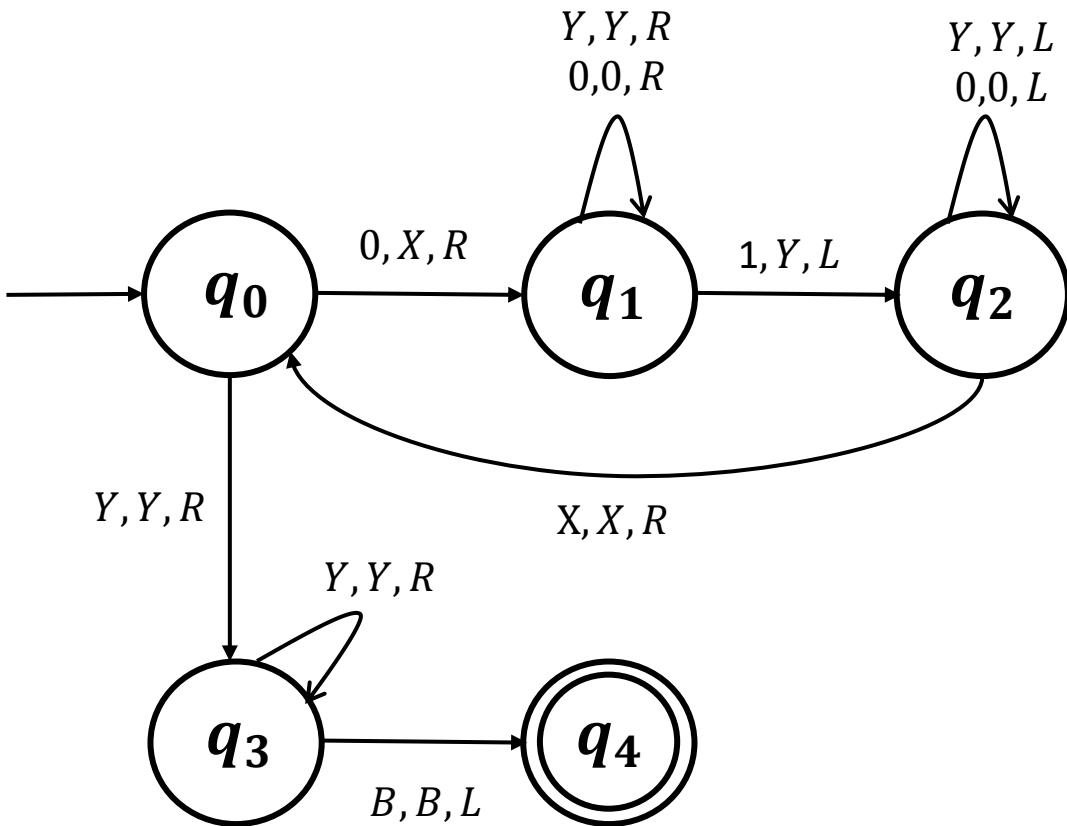
# Turing Machines

Example: Let  $L = \{0^n 1^n \mid n \geq 1\}$

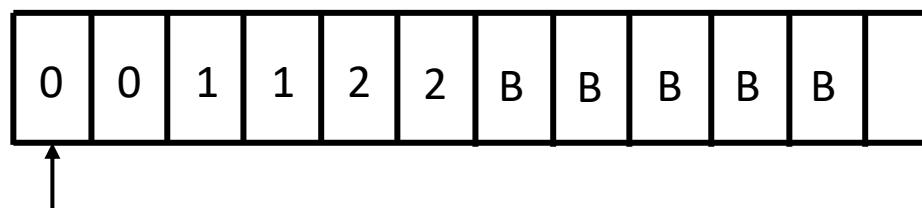


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$

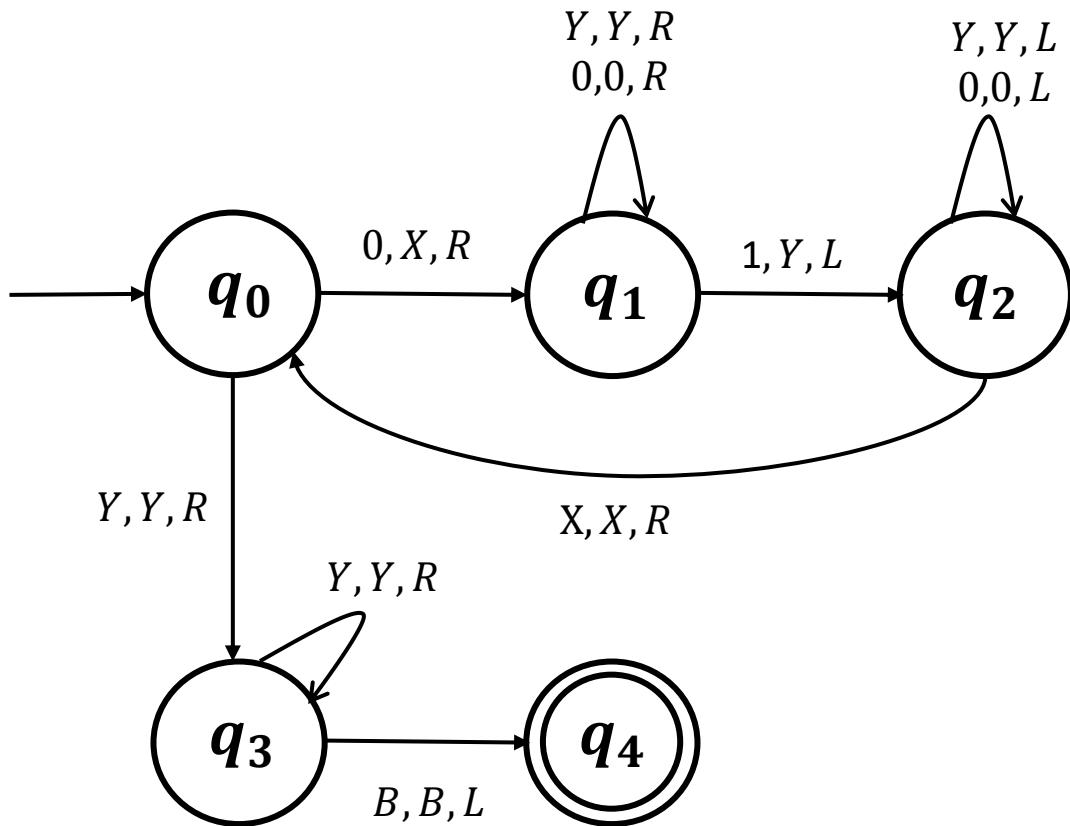


- We will start off with the TM for  $\{0^n 1^n\}$  and construct the TM for  $\{0^n 1^n 2^n\}$
- Very similar to the TM for  $\{0^n 1^n\}$ ,** except now the FSM would count the number of 2's as well. So it marks the 2's with another symbol (say Z)

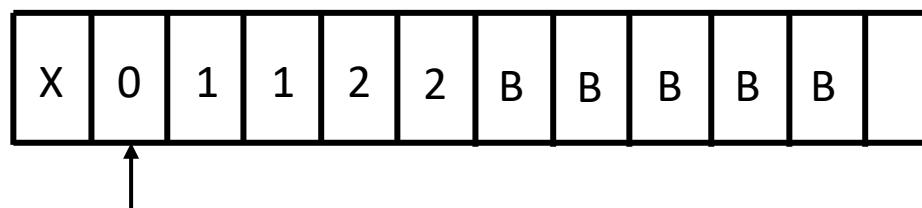


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$

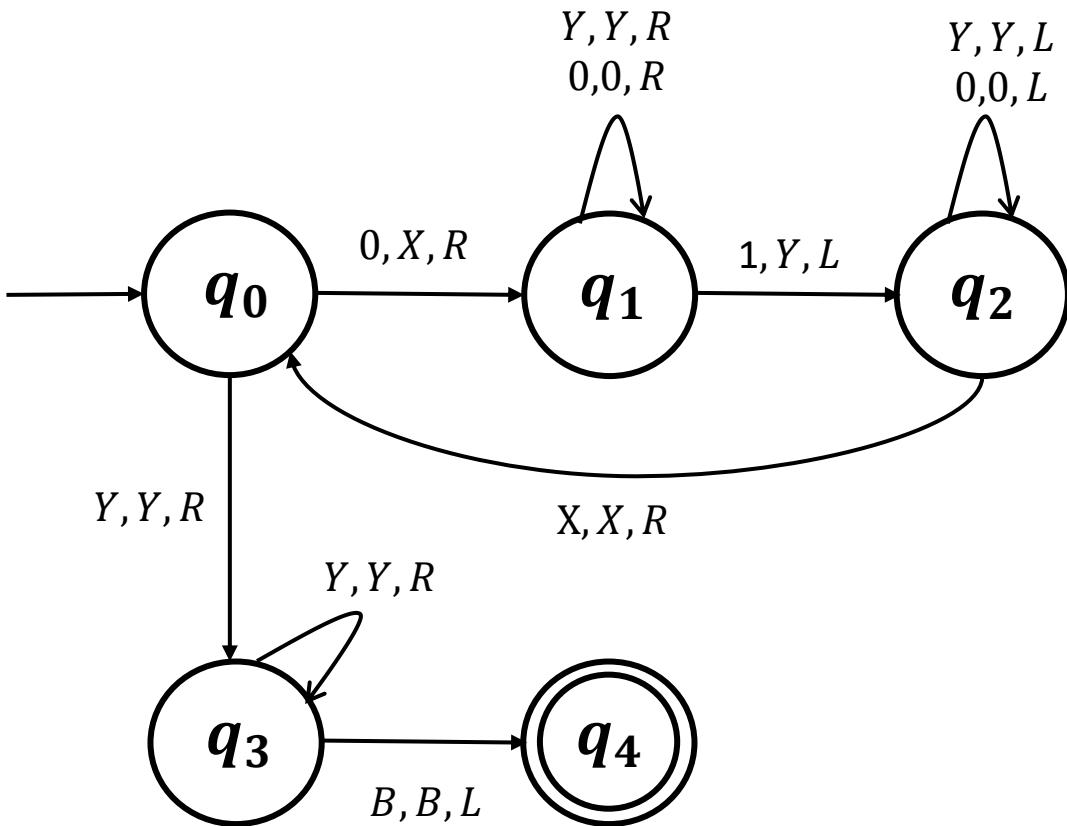


- We will start off with the TM for  $\{0^n 1^n\}$  and construct the TM for  $\{0^n 1^n 2^n\}$
- **Very similar to the TM for  $\{0^n 1^n\}$ ,** except now the FSM would count the number of 2's as well. So it marks the 2's with another symbol (say Z)

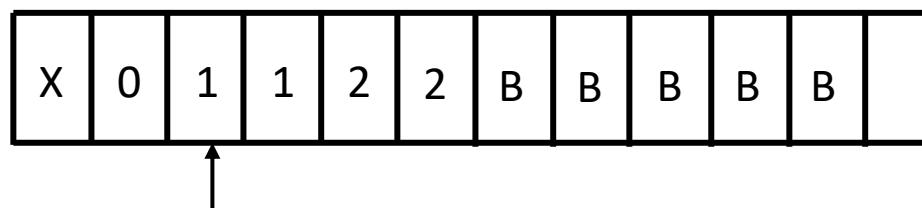


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$

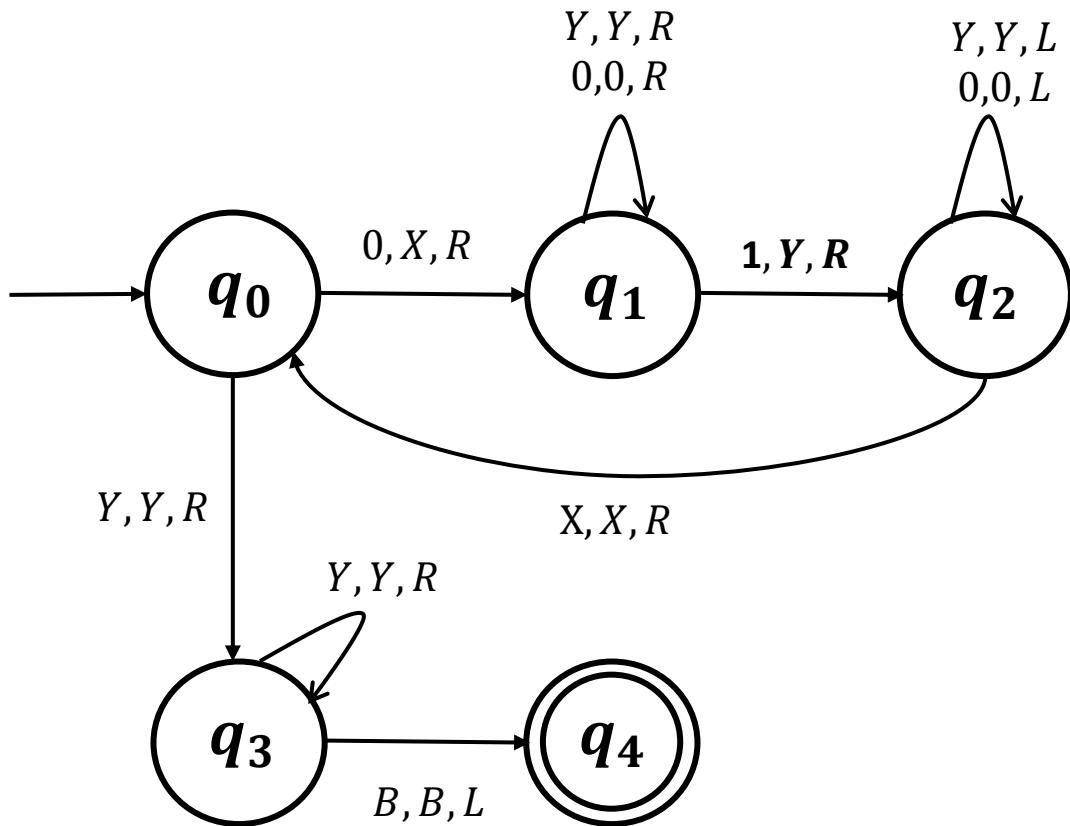


- We will start off with the TM for  $\{0^n 1^n\}$  and construct the TM for  $\{0^n 1^n 2^n\}$
- **Very similar to the TM for  $\{0^n 1^n\}$ ,** except now the FSM would count the number of 2's as well. So it marks the 2's with another symbol (say Z)

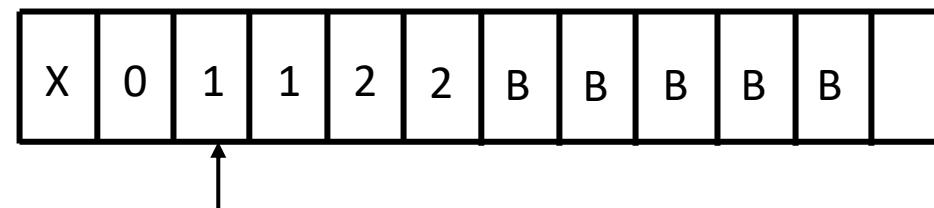


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$

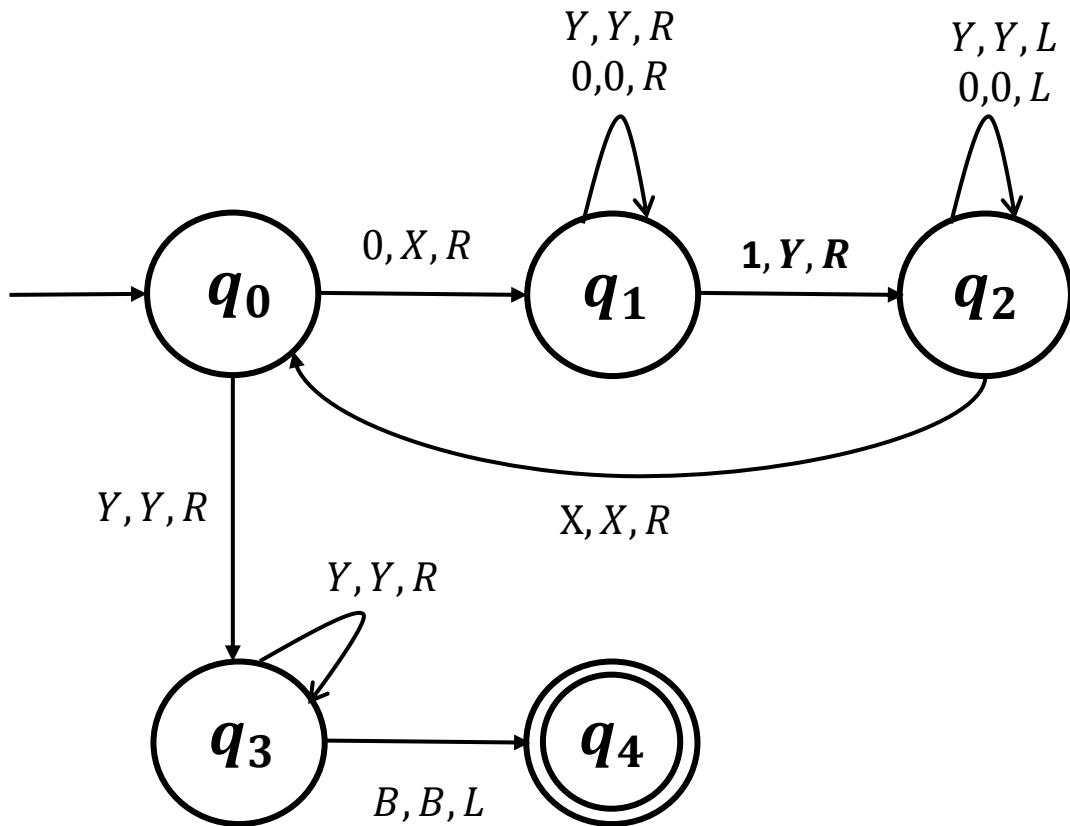


- Continue to go right to mark the next 2 with a Z.

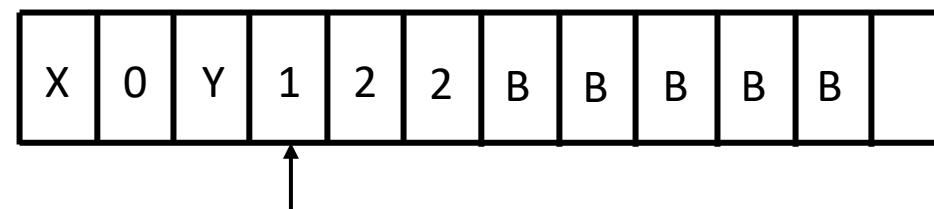


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$

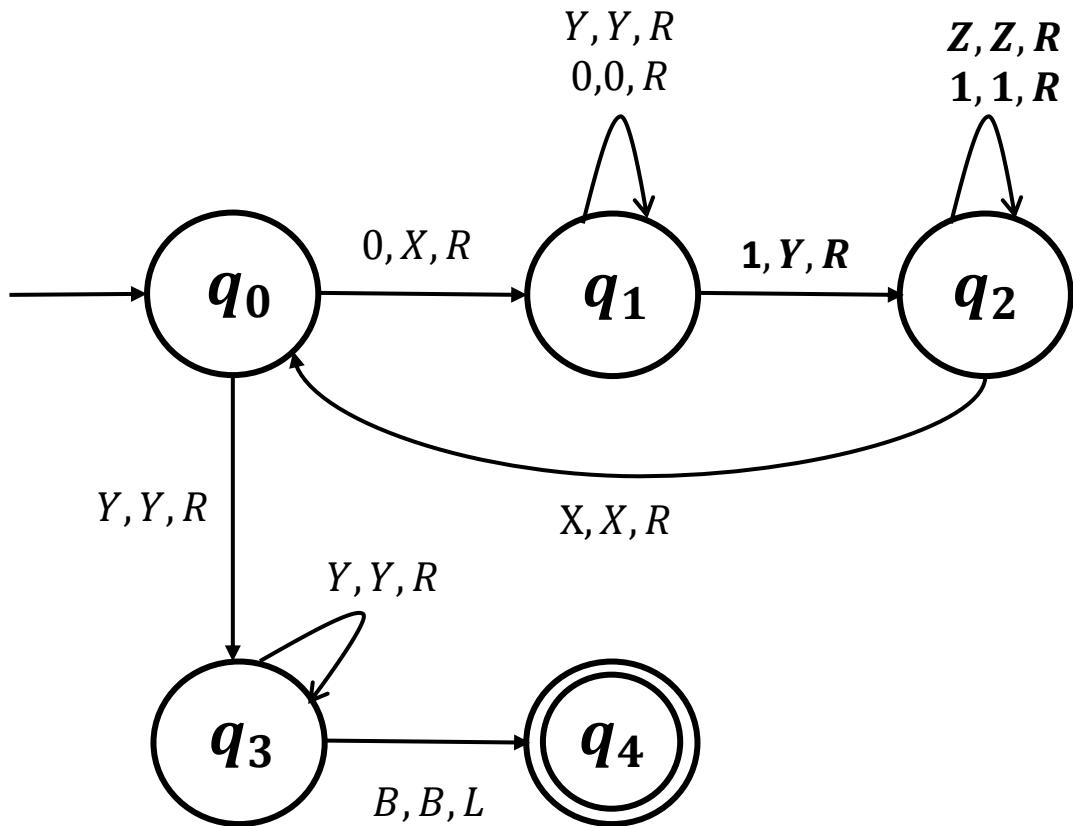


- Continue to go right to mark the next 2 with a Z.

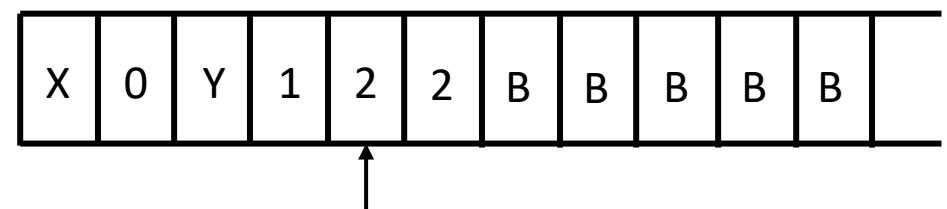


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$

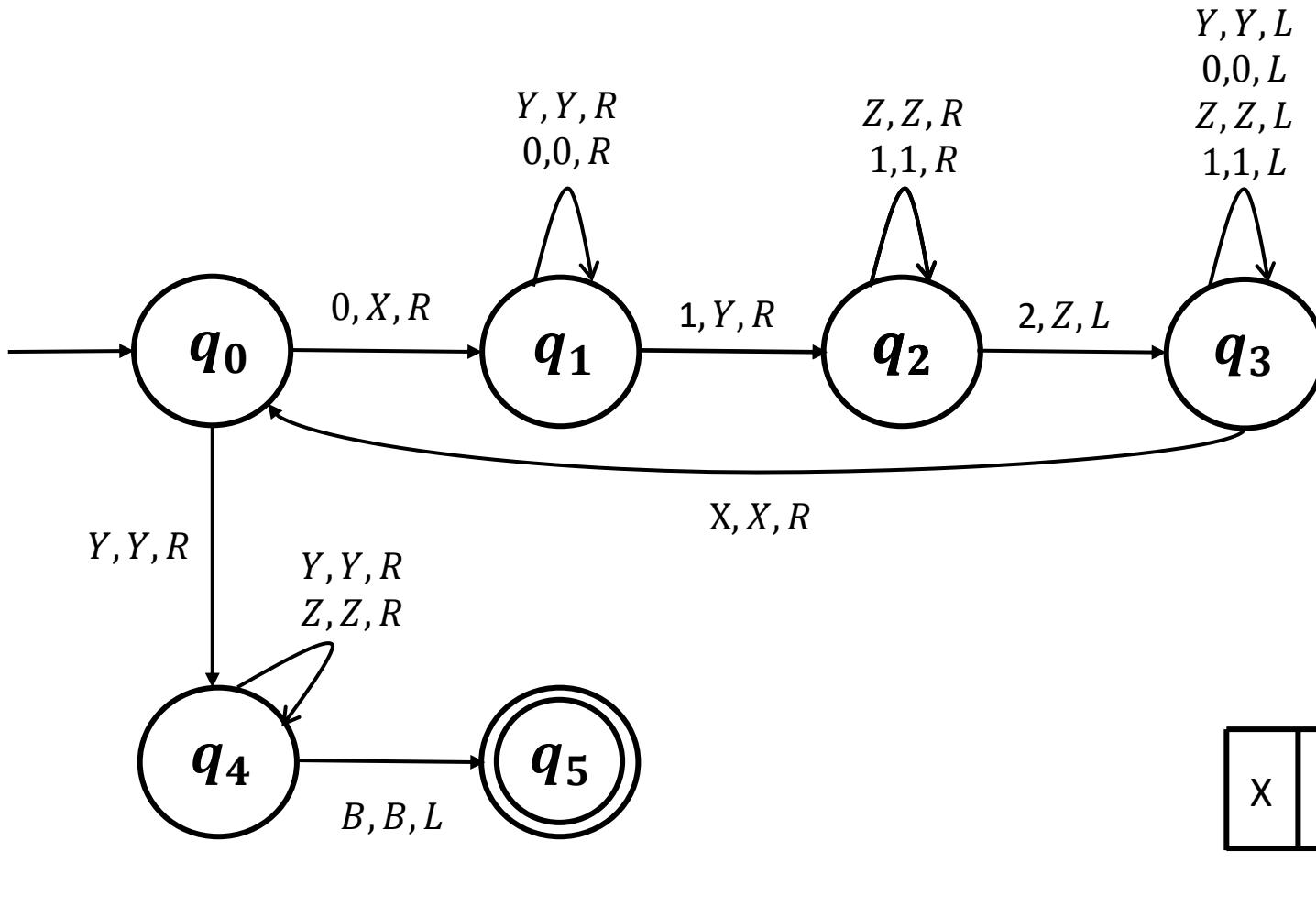


- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.

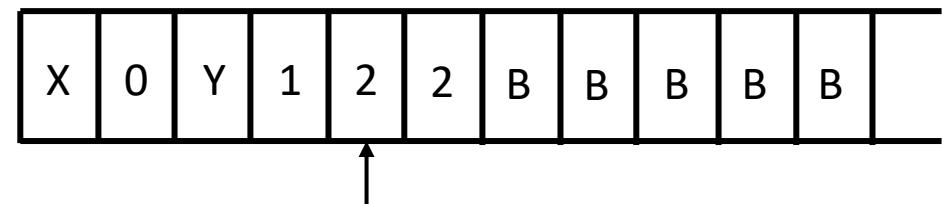


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$

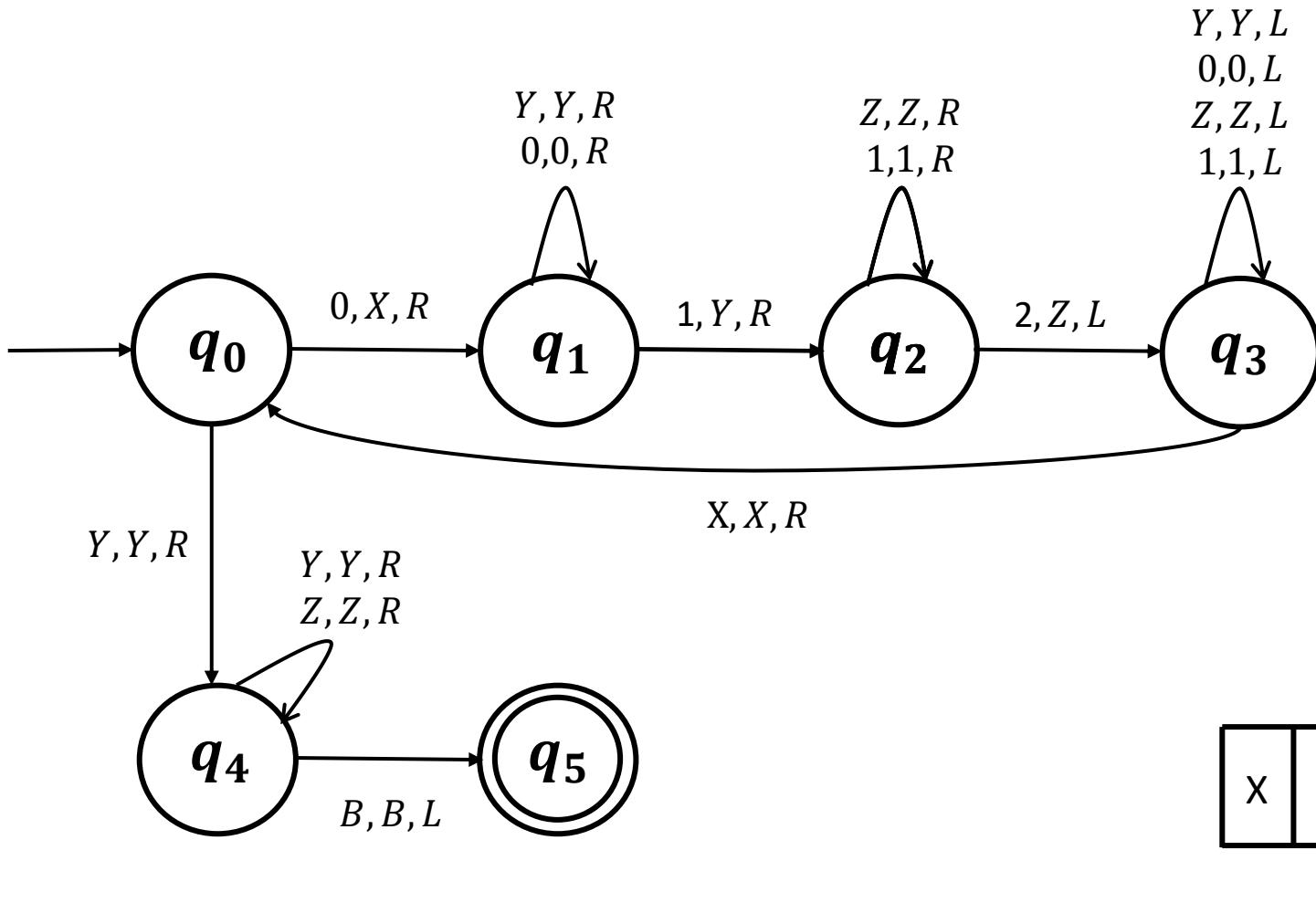


- Continue to go right to mark the next 2 with a  $Z$ .
- Skip all the 1's and move right/ Also, move across (to the right) the  $Z$ 's already marked.
- Mark a new 2 with a  $Z$  and start moving left.
- Keep moving left until an  $X$  is encountered.

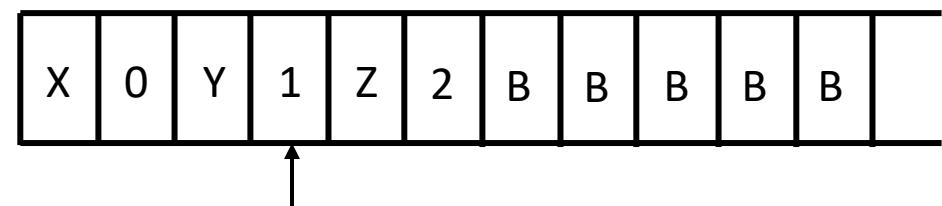


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$

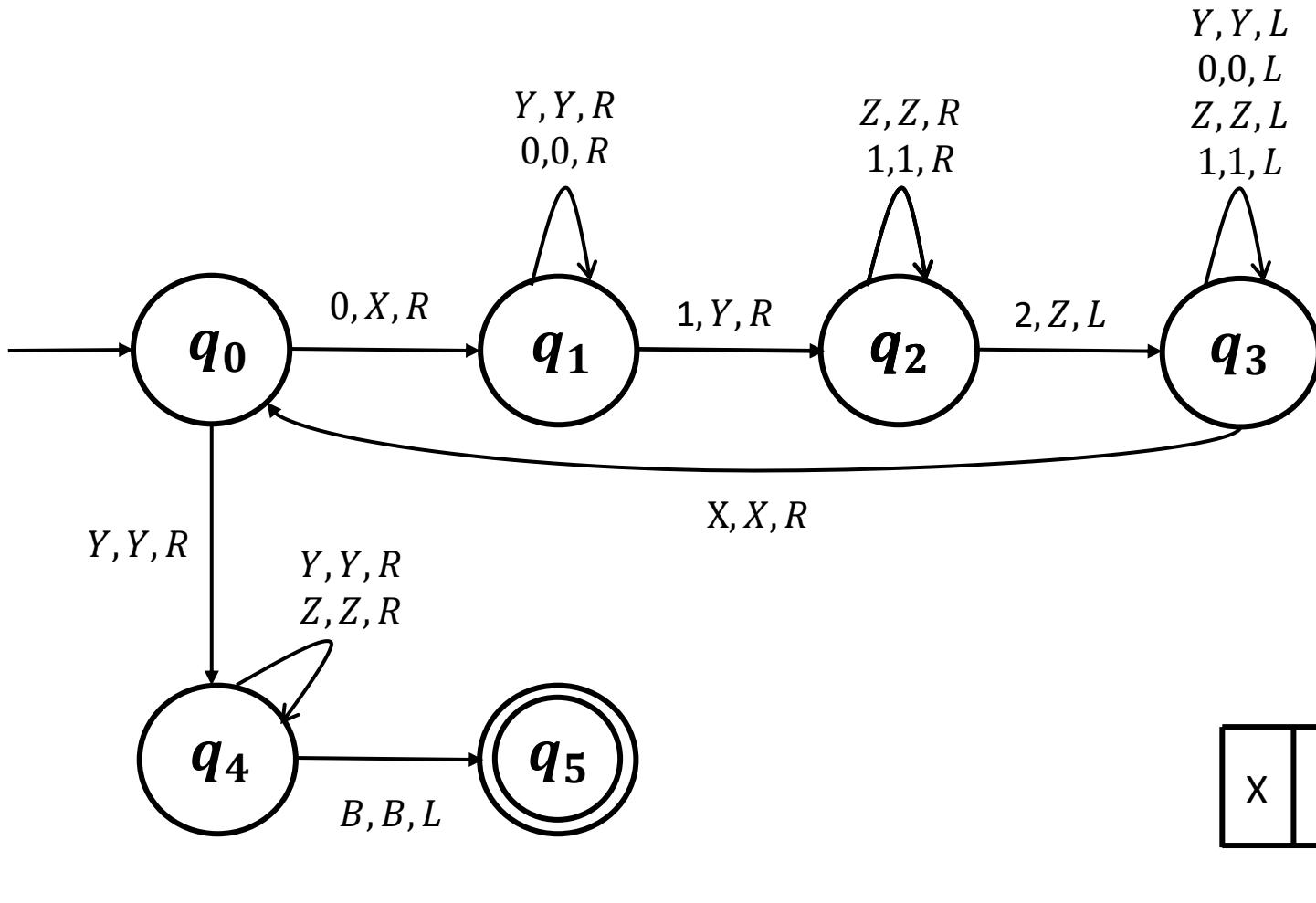


- Continue to go right to mark the next 2 with a  $Z$ .
- Skip all the 1's and move right/ Also, move across (to the right) the  $Z$ 's already marked.
- Mark a new 2 with a  $Z$  and start moving left.
- Keep moving left until an  $X$  is encountered.

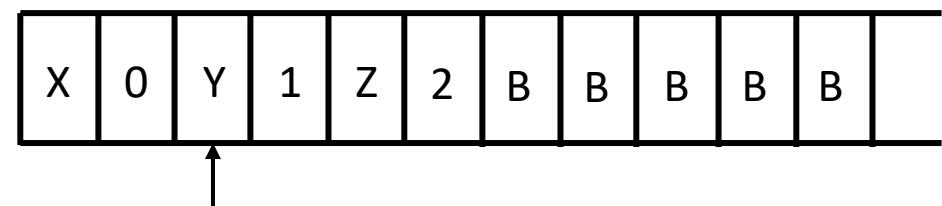


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$

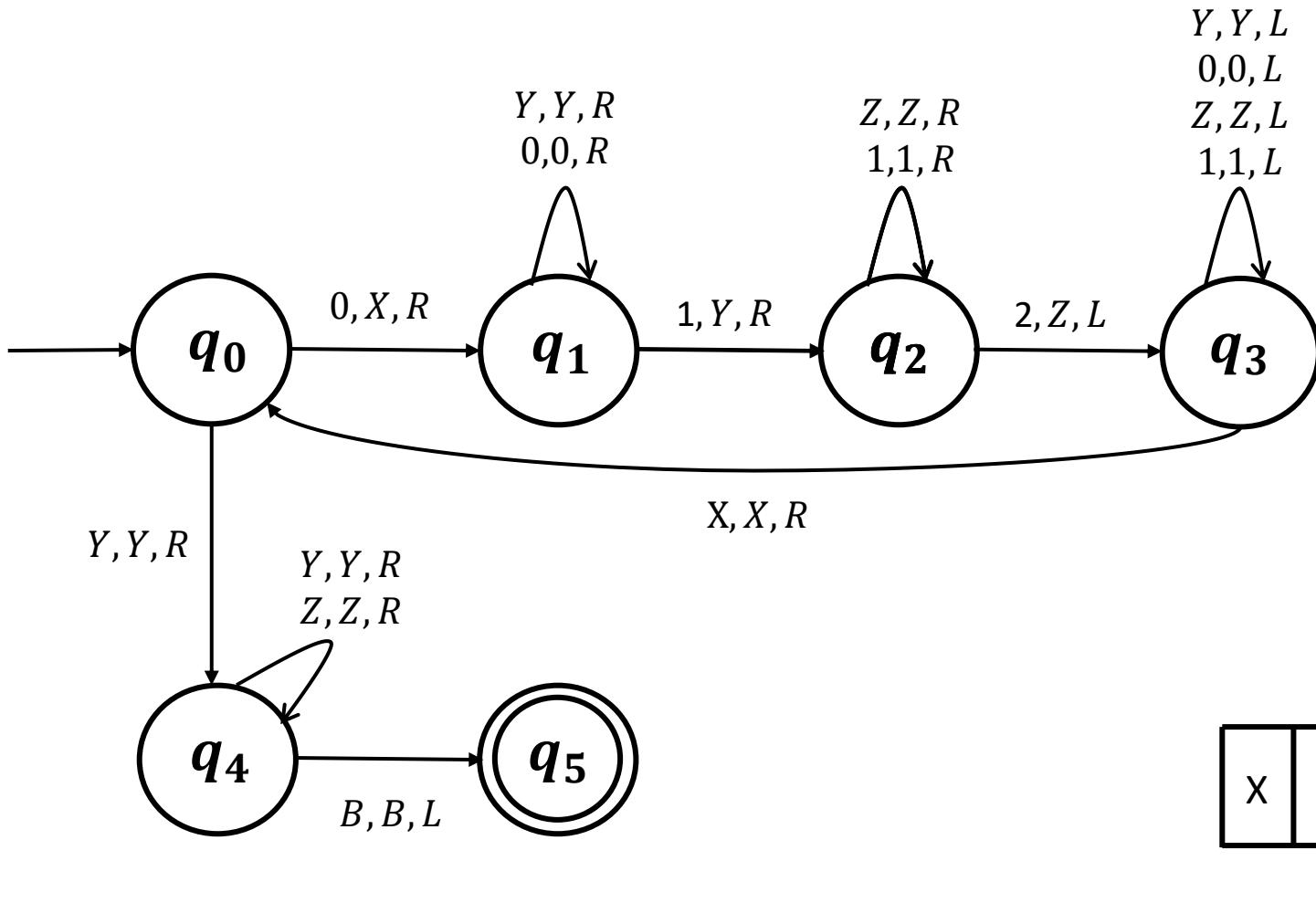


- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.

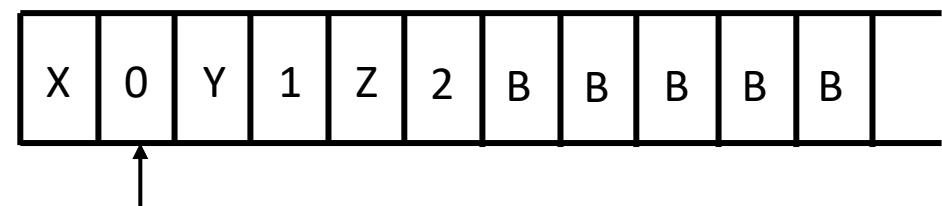


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$

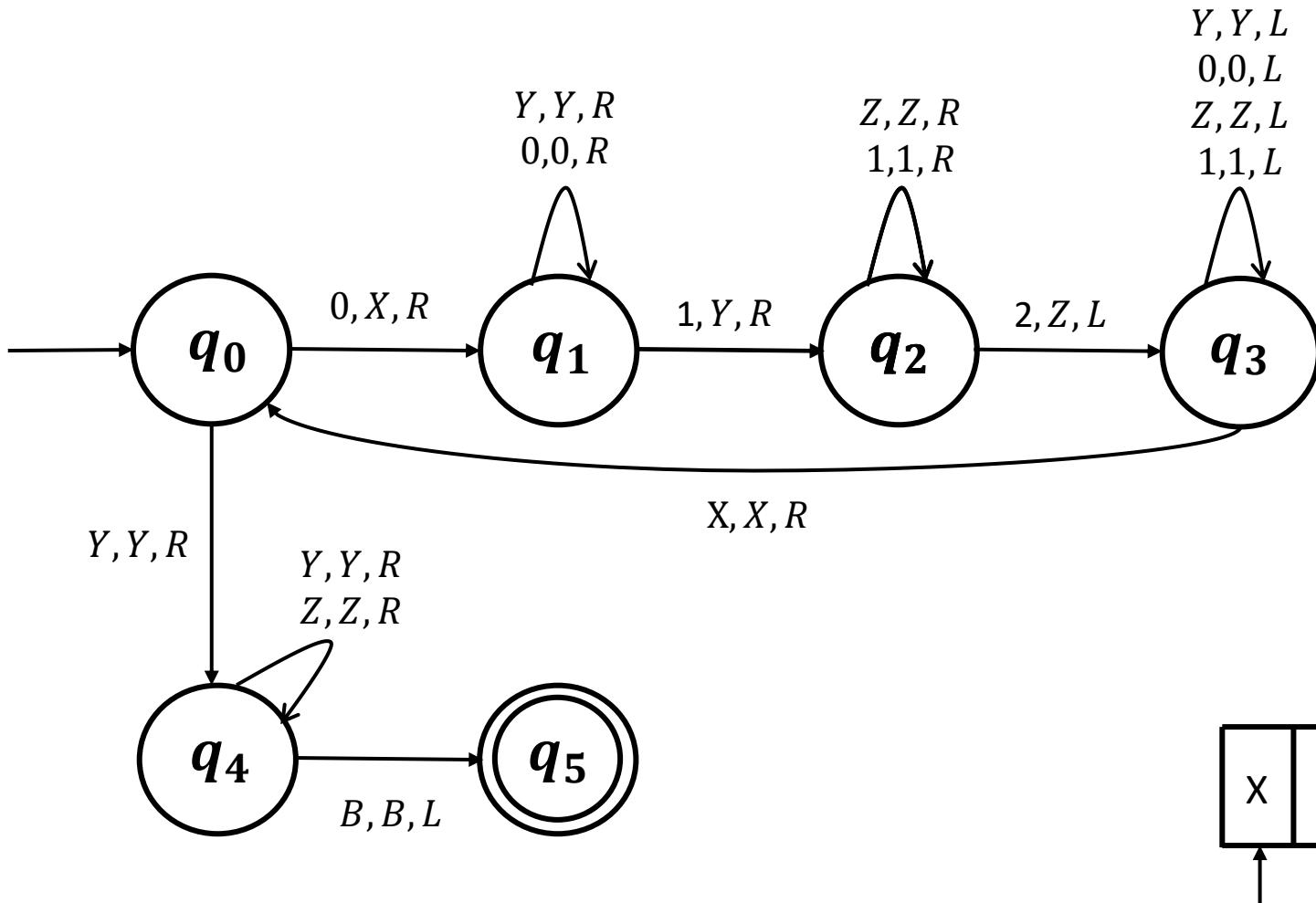


- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.

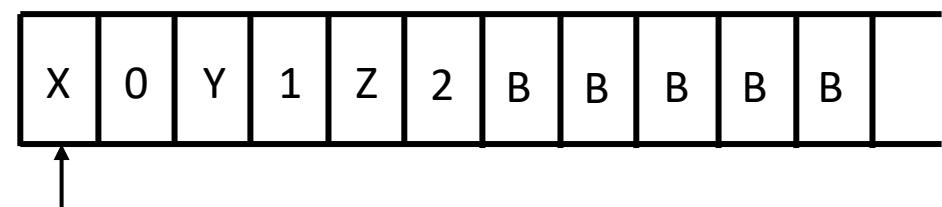


# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$

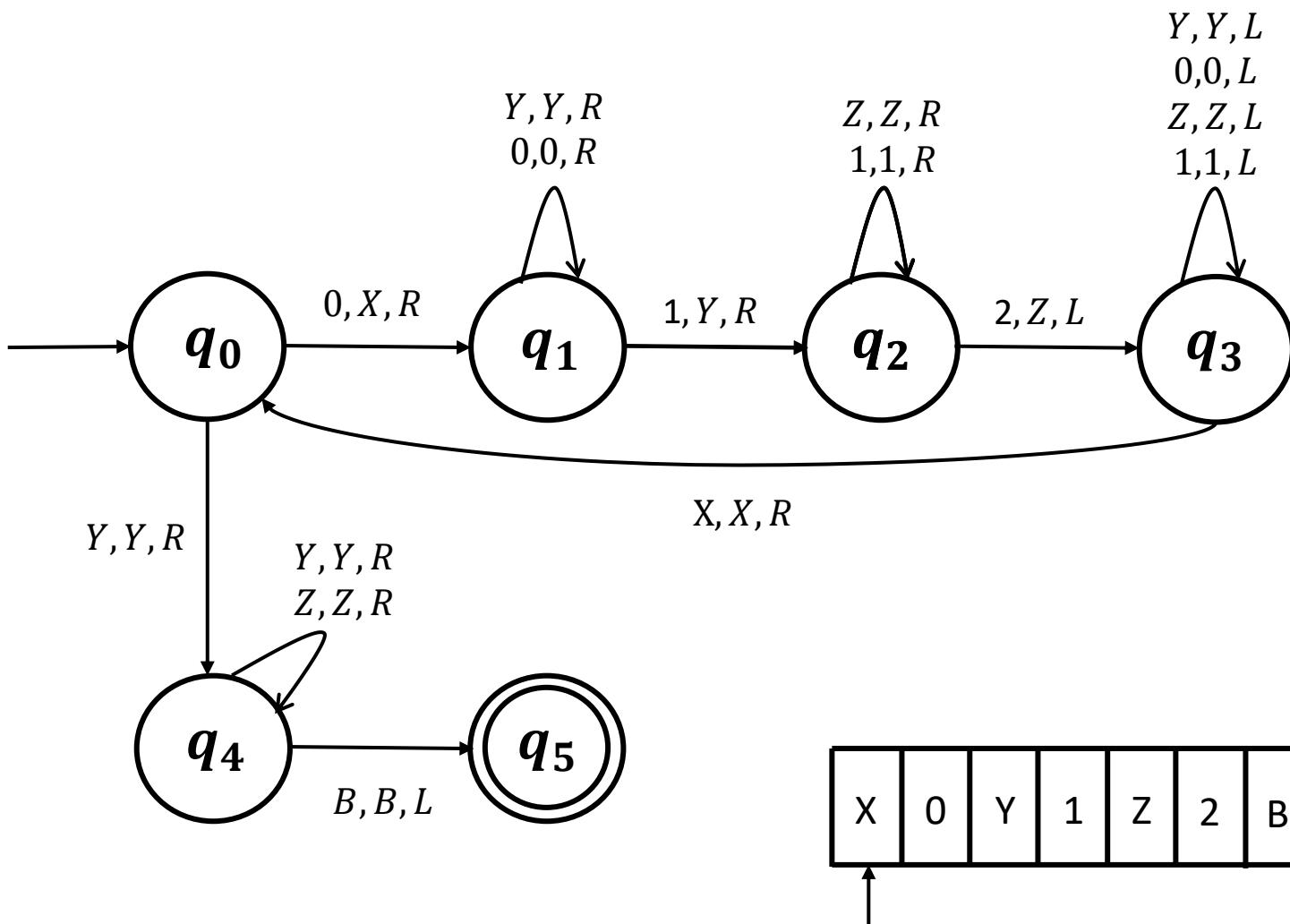


- Continue to go right to mark the next 2 with a  $Z$ .
- Skip all the 1's and move right/ Also, move across (to the right) the  $Z$ 's already marked.
- Mark a new 2 with a  $Z$  and start moving left.
- Keep moving left until an  $X$  is encountered.



# Turing Machines

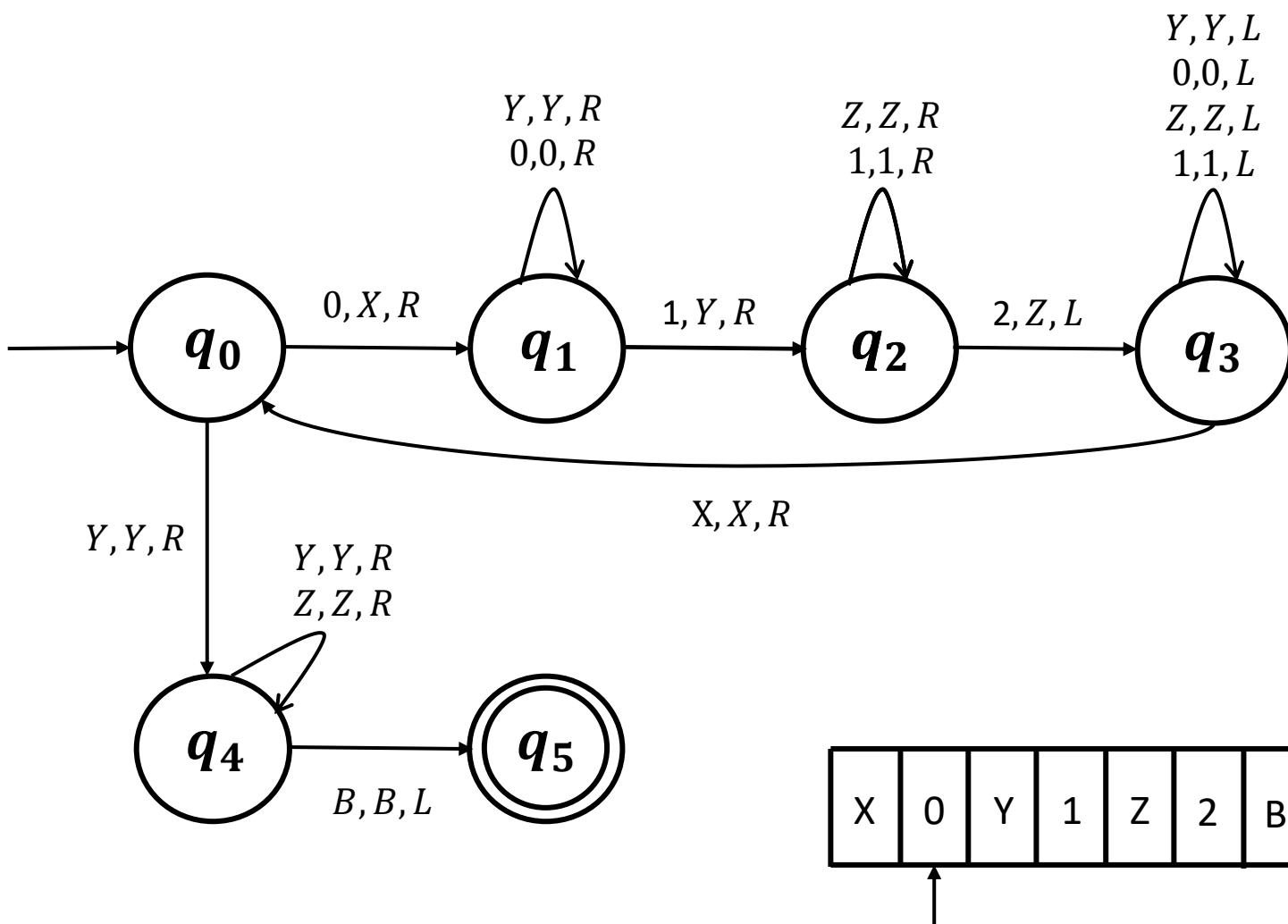
**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**



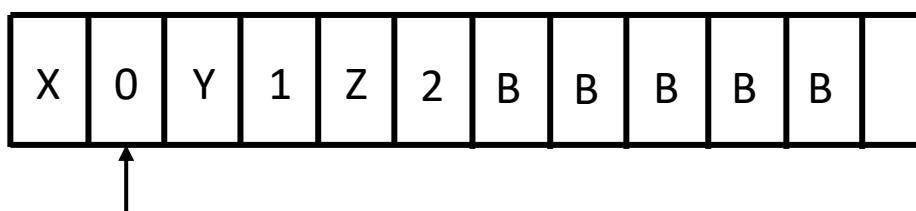
- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**

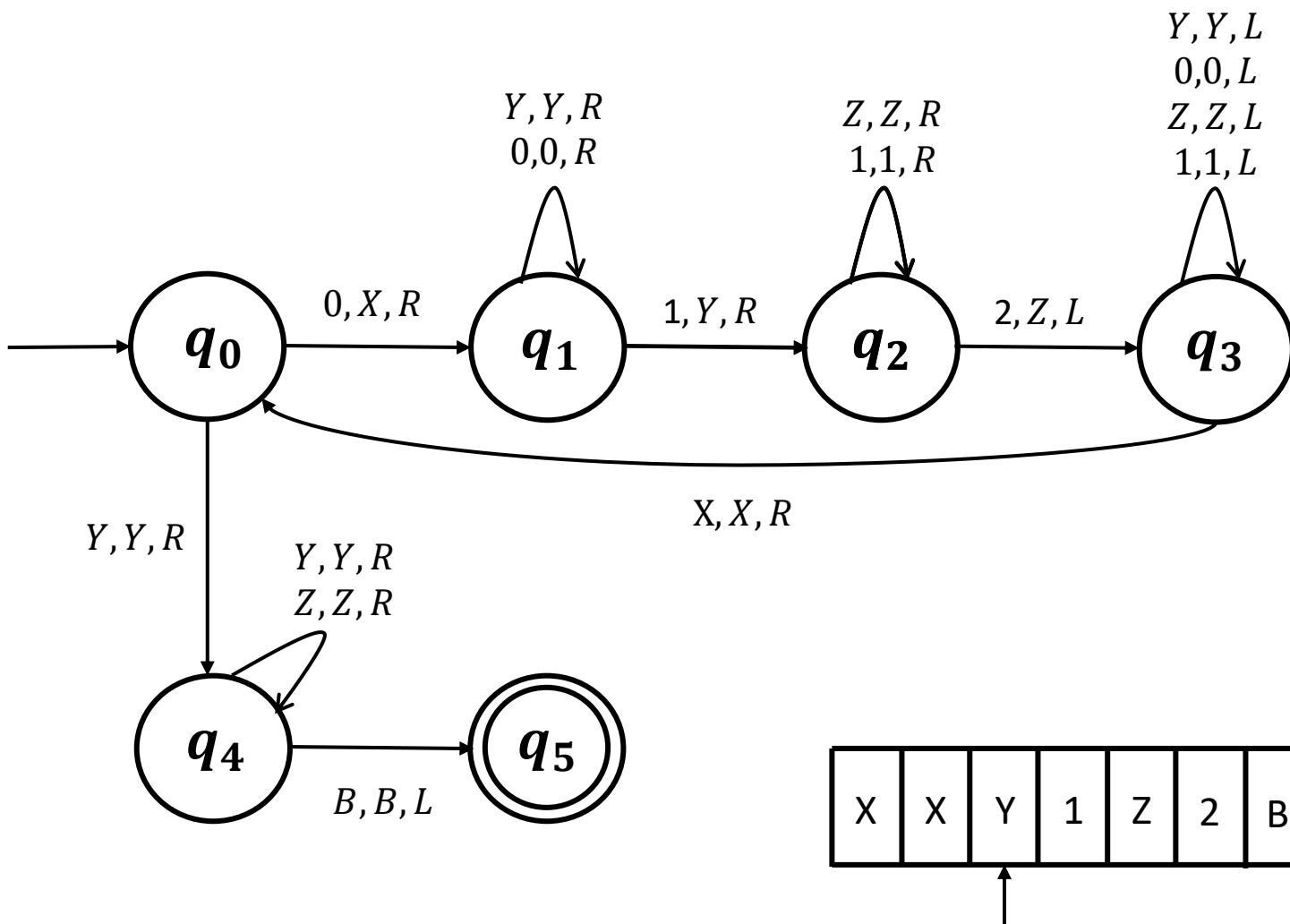


- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.



# Turing Machines

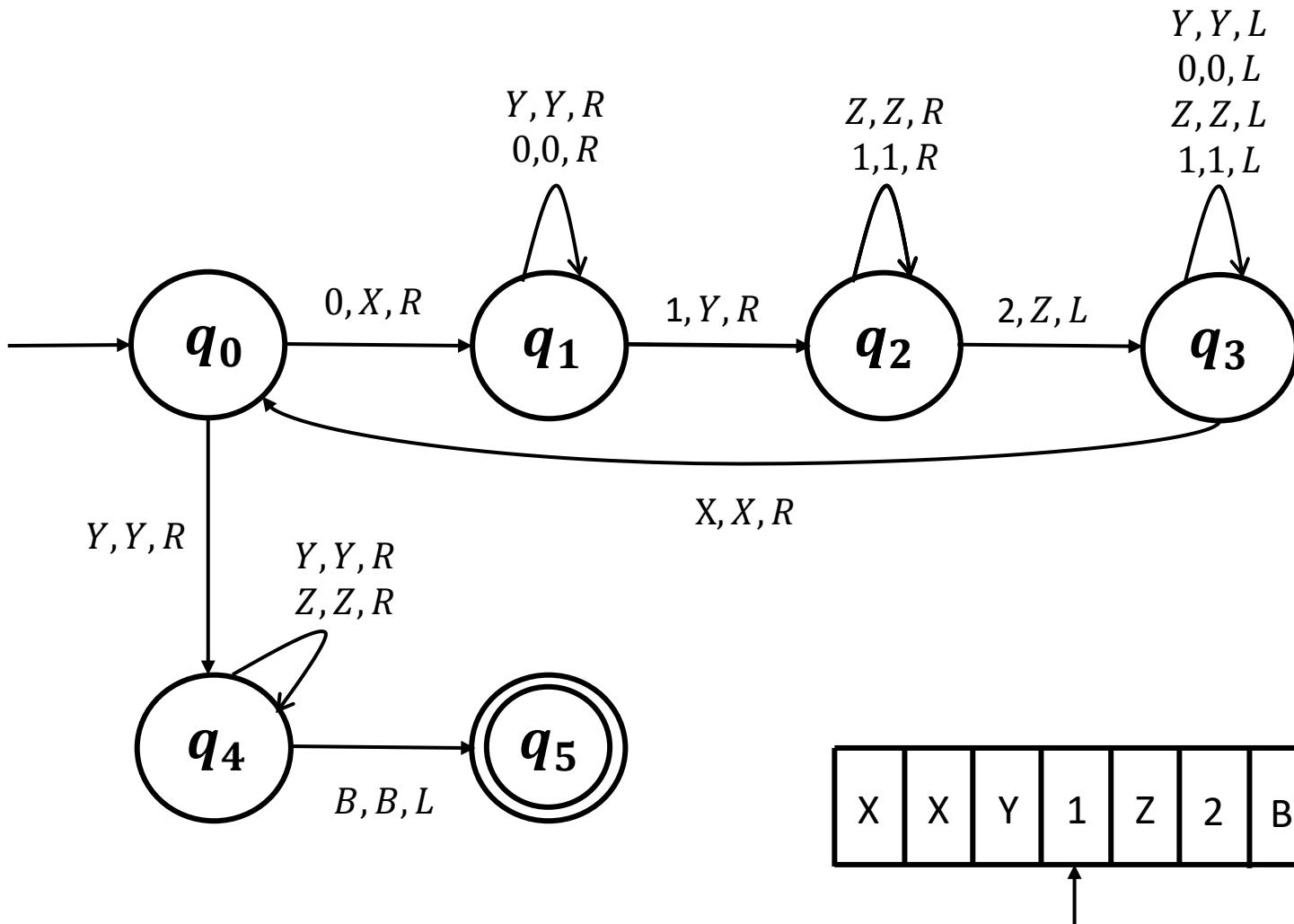
**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**



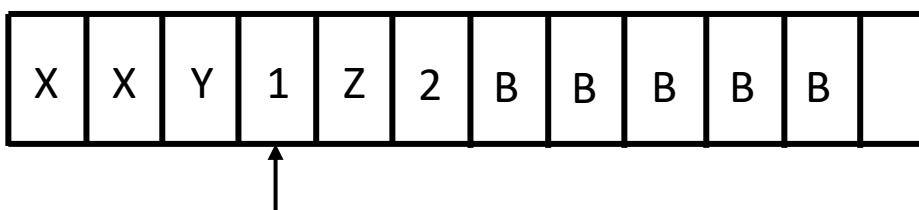
- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**

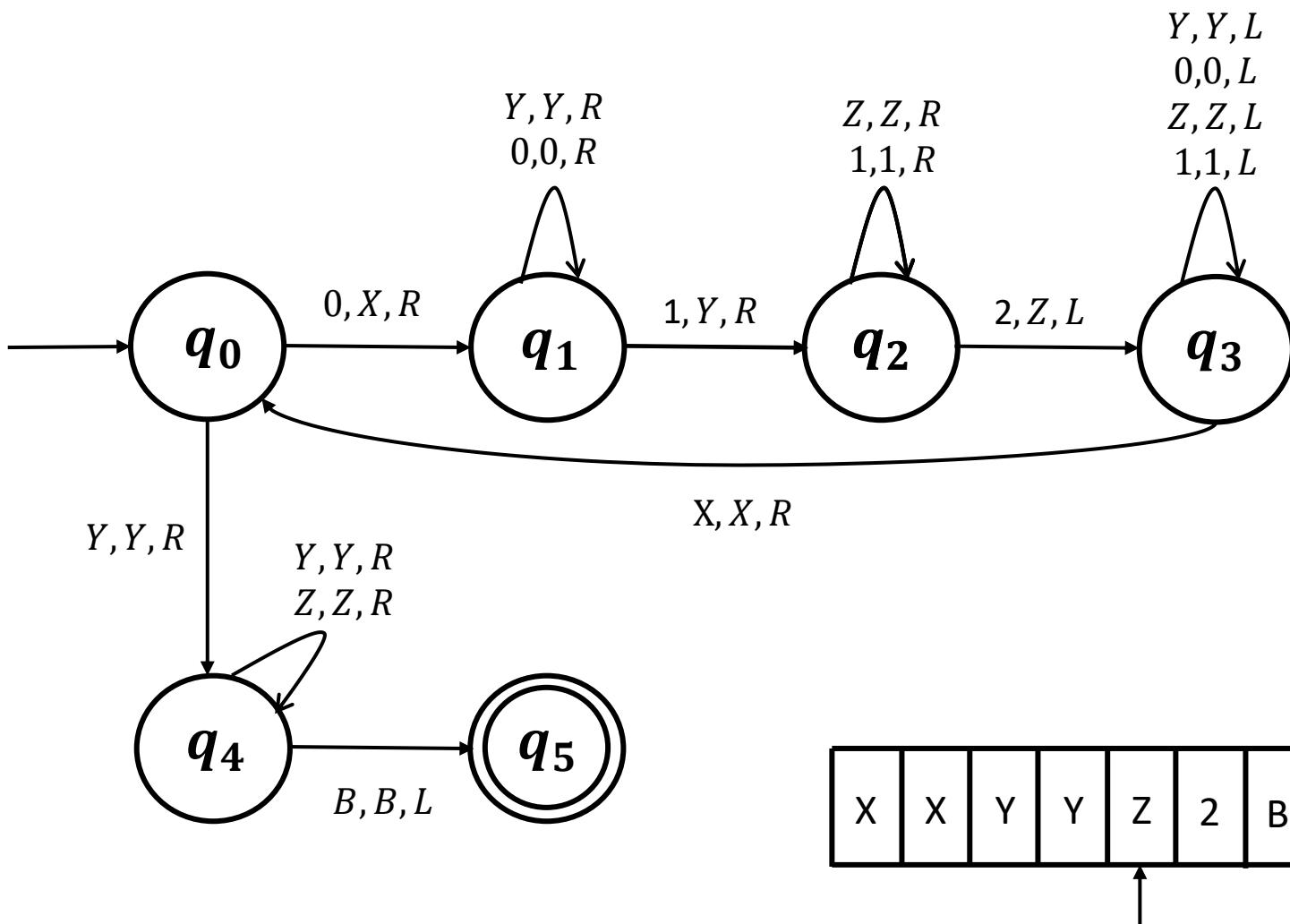


- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

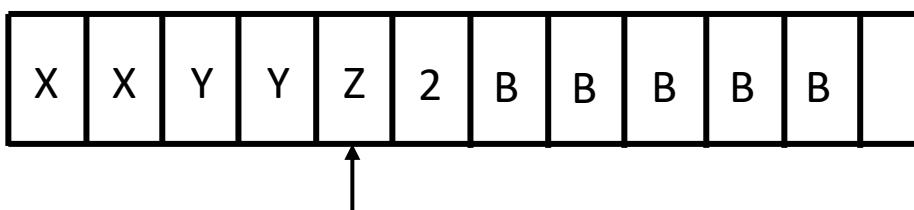


# Turing Machines

**Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$**

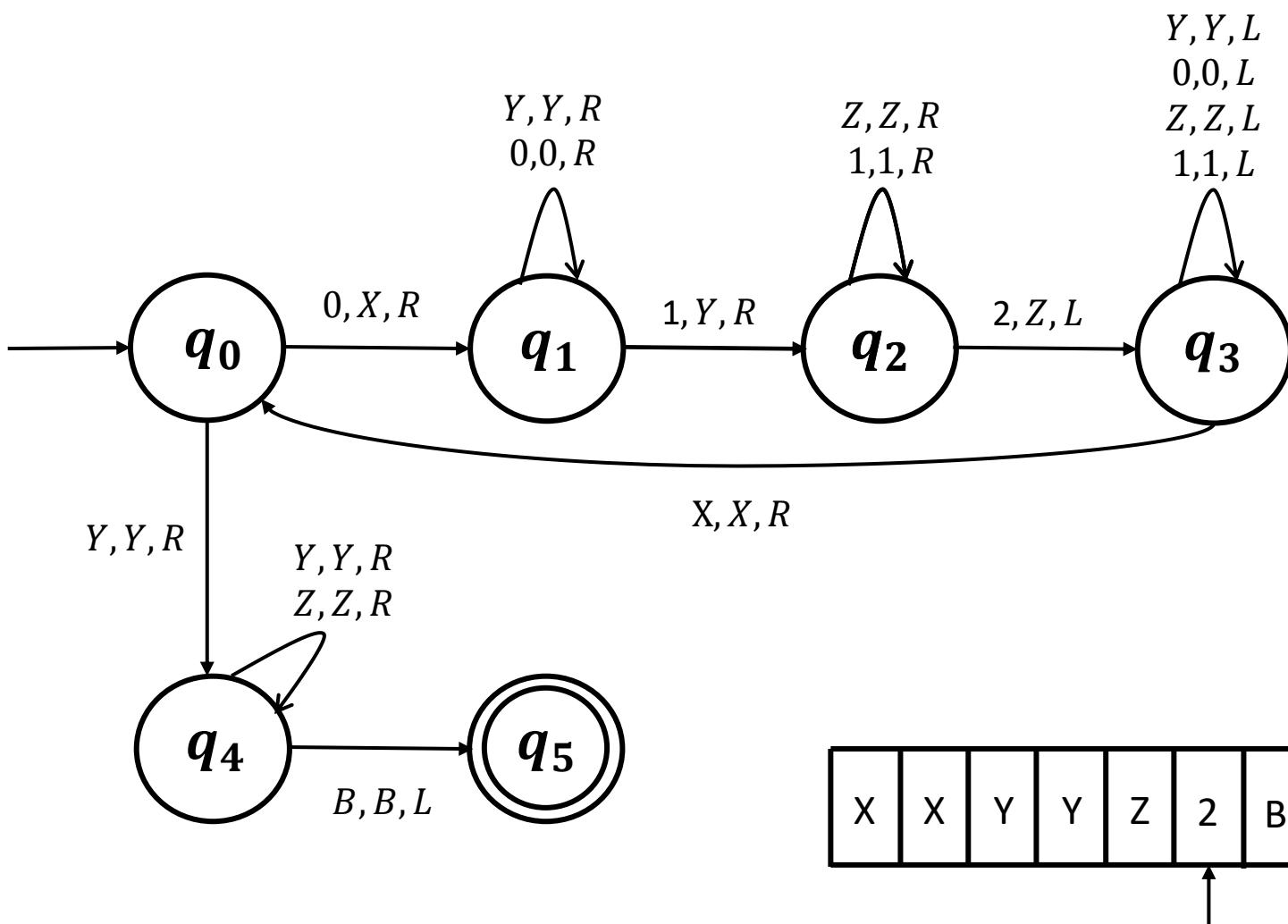


- Continue to go right to mark the next 2 with a  $Z$ .
- Skip all the 1's and move right/ Also, move across (to the right) the  $Z$ 's already marked.
- Mark a new 2 with a  $Z$  and start moving left.
- Keep moving left until an  $X$  is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the  $Y$ 's and  $Z$ 's to the right end of the tape until a blank is found.
- Move left and accept the input string.



# Turing Machines

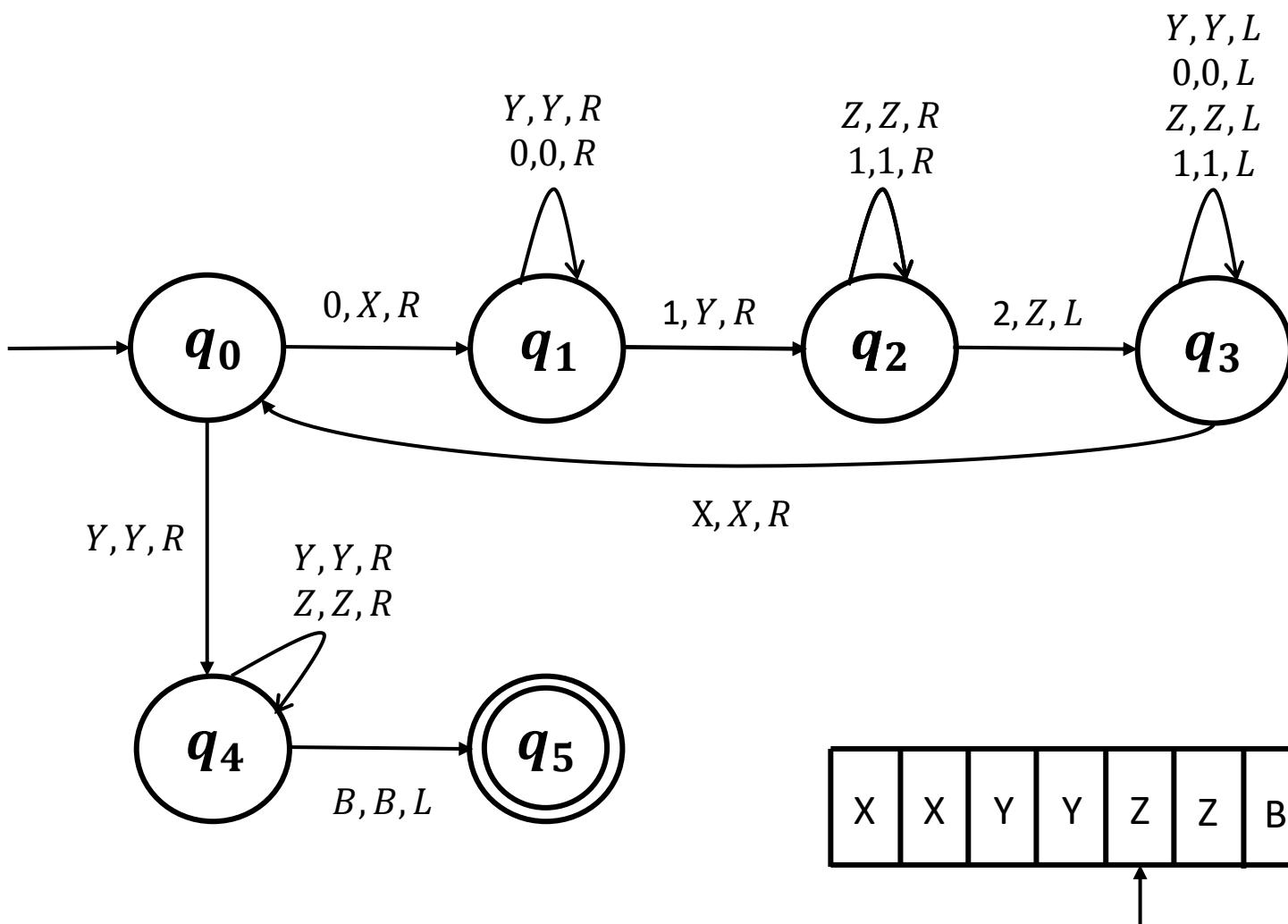
**Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$**



- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark
  - or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

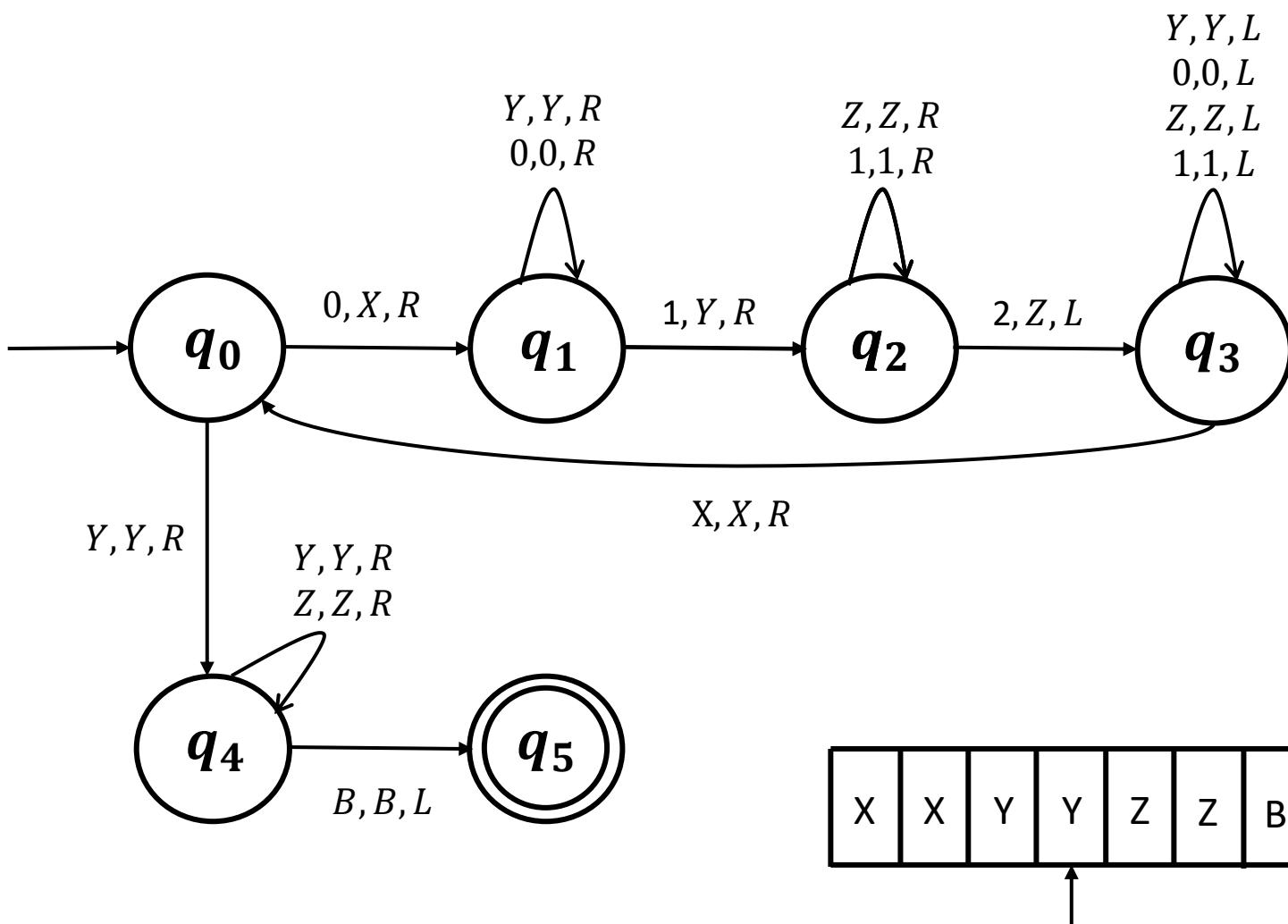
Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$



- Continue to go right to mark the next 2 with a  $Z$ .
- Skip all the  $1$ 's and move right/ Also, move across (to the right) the  $Z$ 's already marked.
- Mark a new  $2$  with a  $Z$  and start moving left.
- Keep moving left until an  $X$  is encountered.
- Either repeat this process if there are  $0$ 's,  $1$ 's or  $2$ 's left to mark  
or
- Skip across the  $Y$ 's and  $Z$ 's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

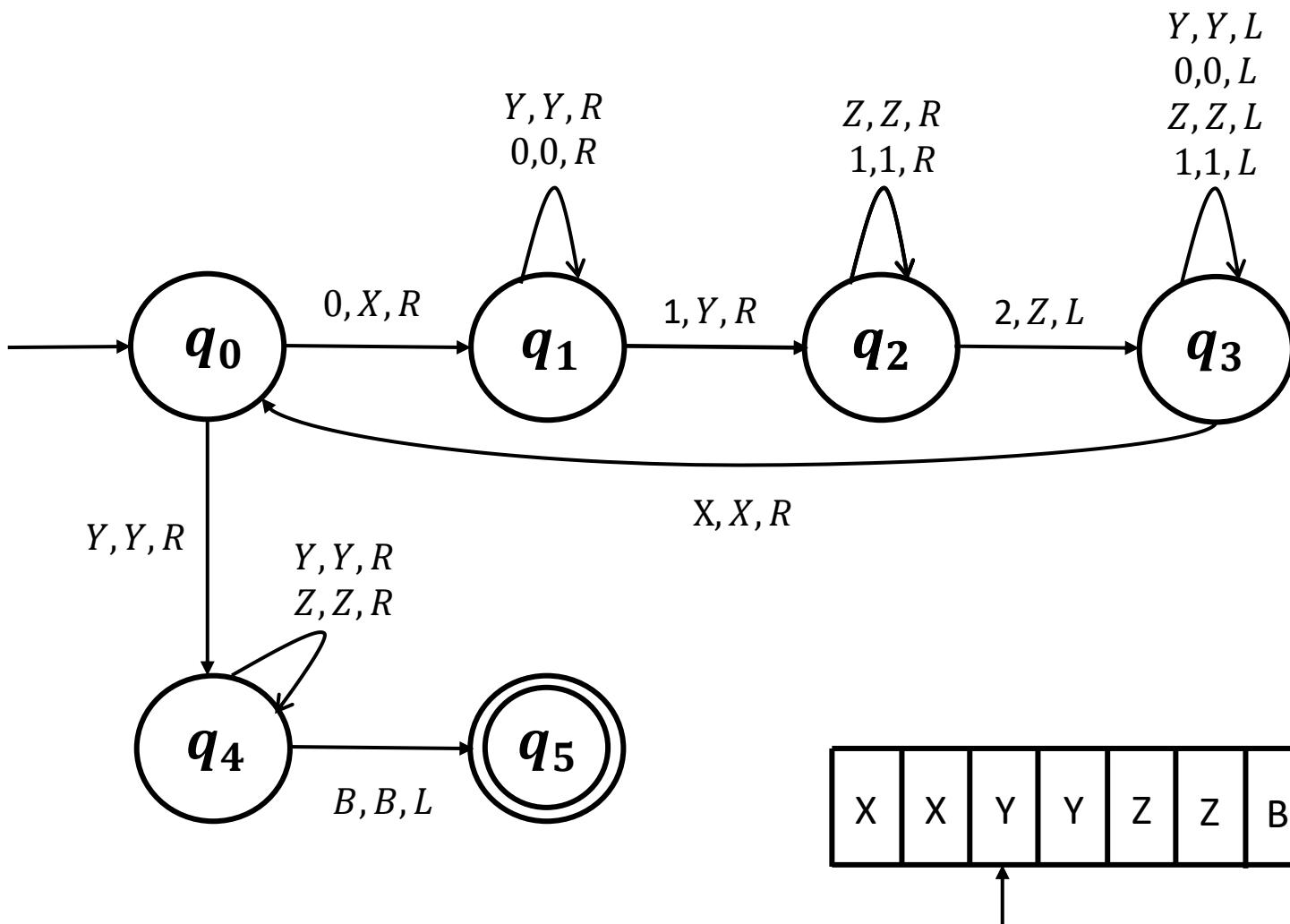
**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**



- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

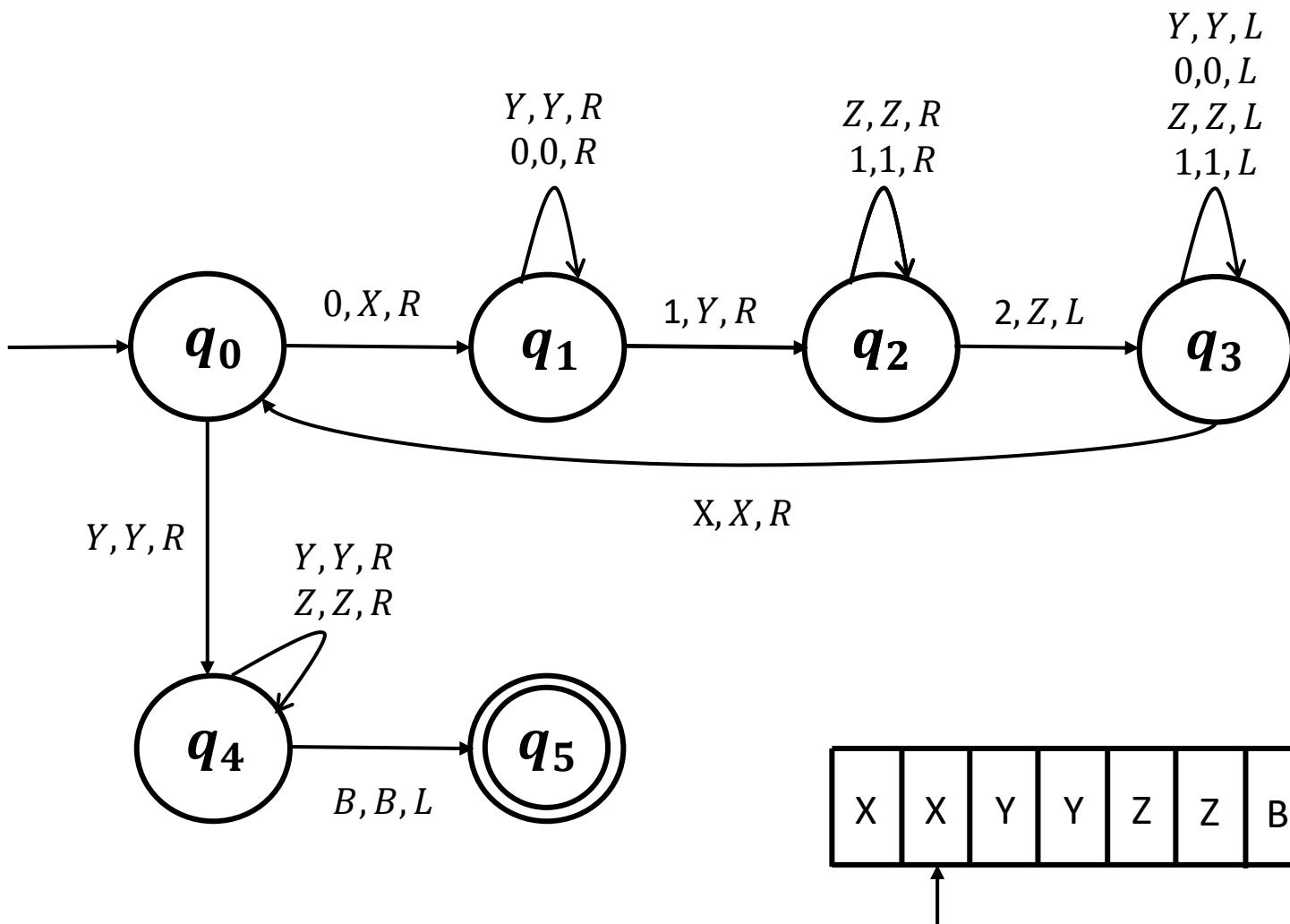
**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**



- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

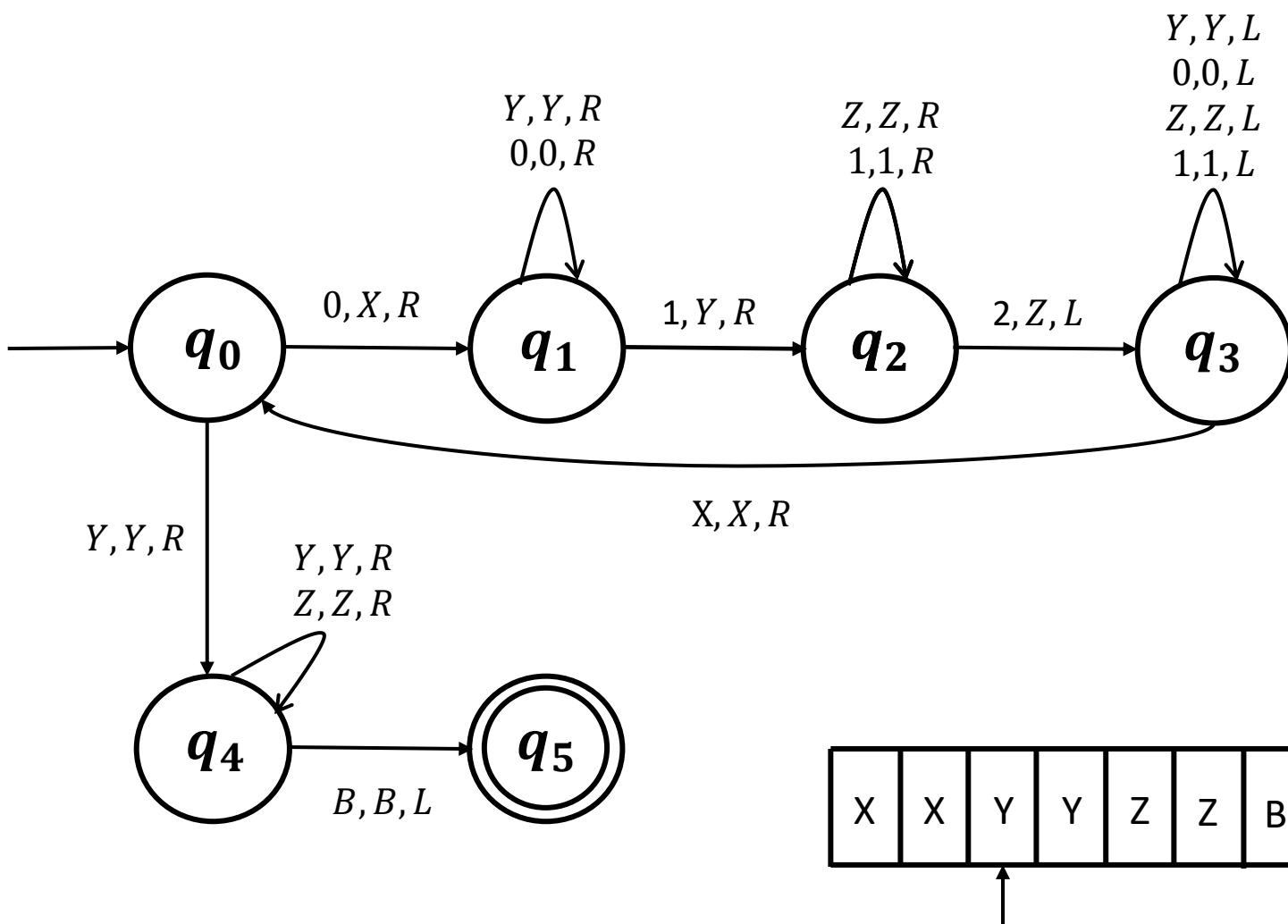
**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**



- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

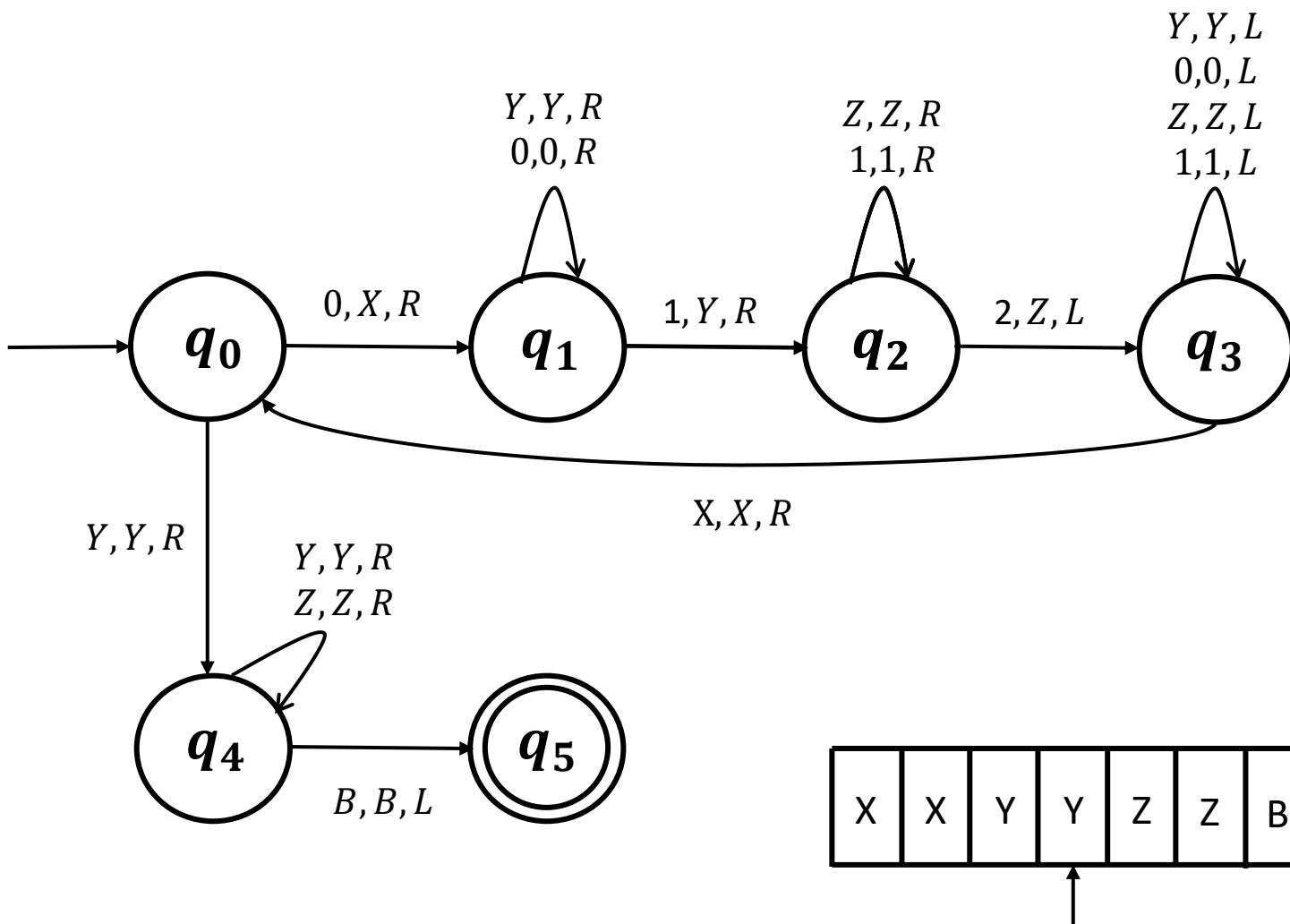
**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**



- Continue to go right to mark the next 2 with a  $Z$ .
- Skip all the 1's and move right/ Also, move across (to the right) the  $Z$ 's already marked.
- Mark a new 2 with a  $Z$  and start moving left.
- Keep moving left until an  $X$  is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the  $Y$ 's and  $Z$ 's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

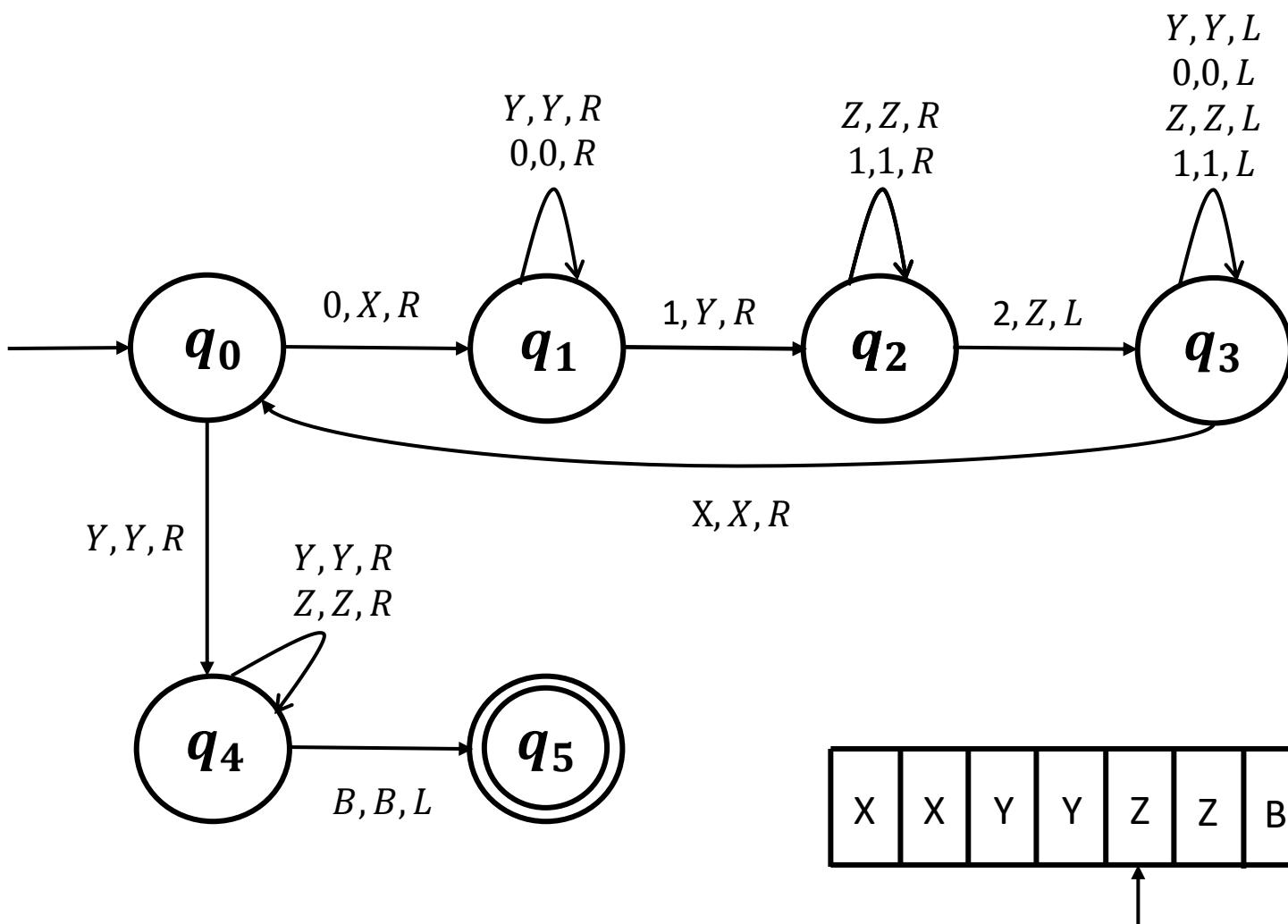
**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**



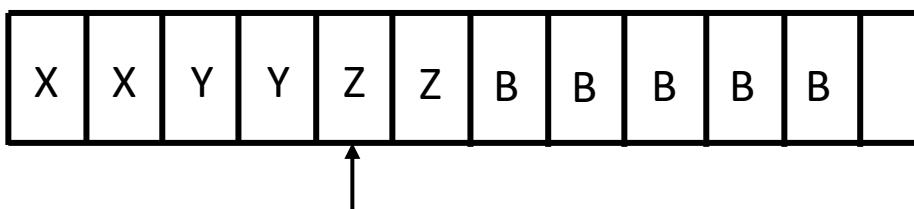
- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**

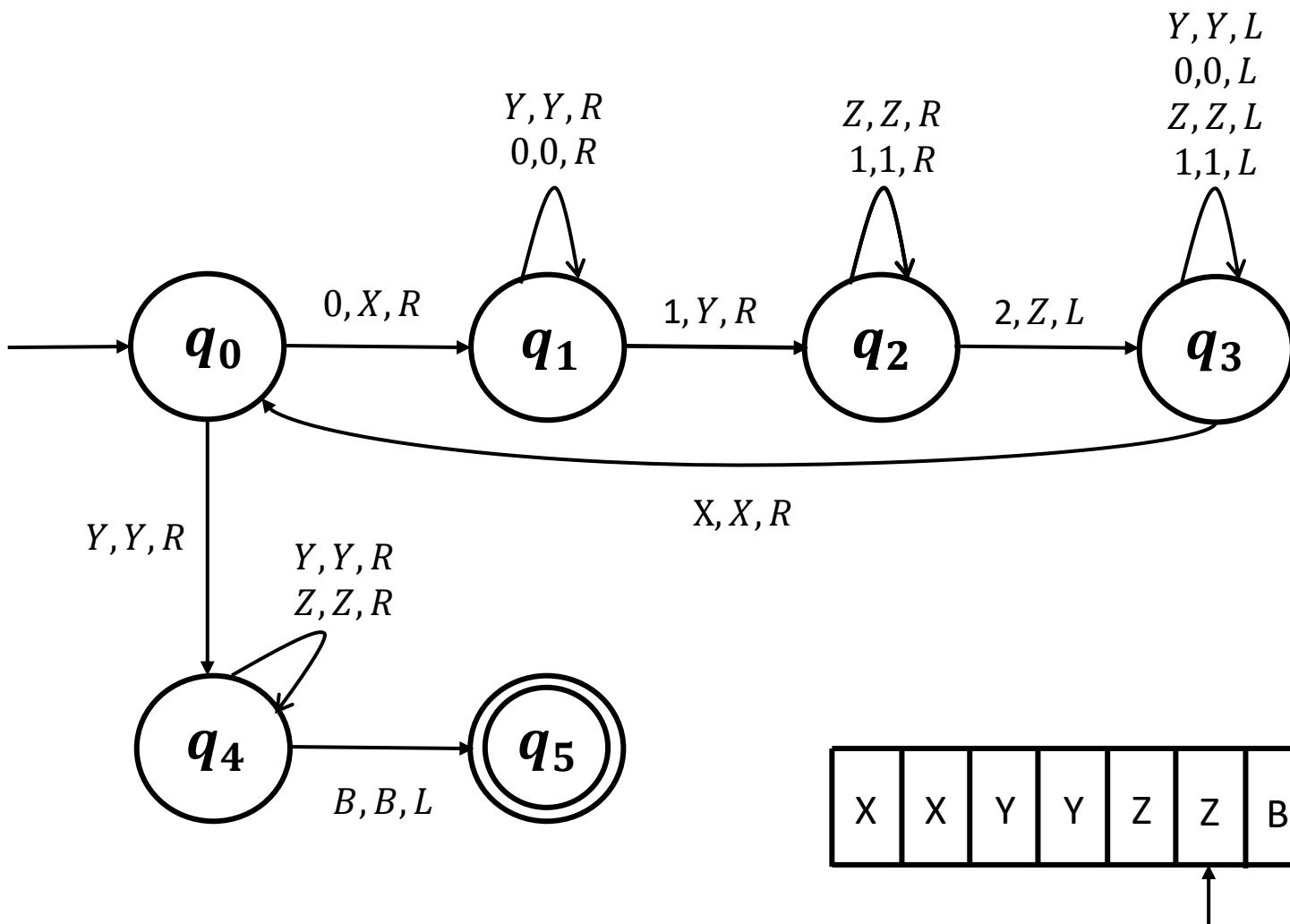


- Continue to go right to mark the next 2 with a  $Z$ .
- Skip all the  $1$ 's and move right/ Also, move across (to the right) the  $Z$ 's already marked.
- Mark a new  $2$  with a  $Z$  and start moving left.
- Keep moving left until an  $X$  is encountered.
- Either repeat this process if there are  $0$ 's,  $1$ 's or  $2$ 's left to mark  
or
- Skip across the  $Y$ 's and  $Z$ 's to the right end of the tape until a blank is found.
- Move left and accept the input string.



# Turing Machines

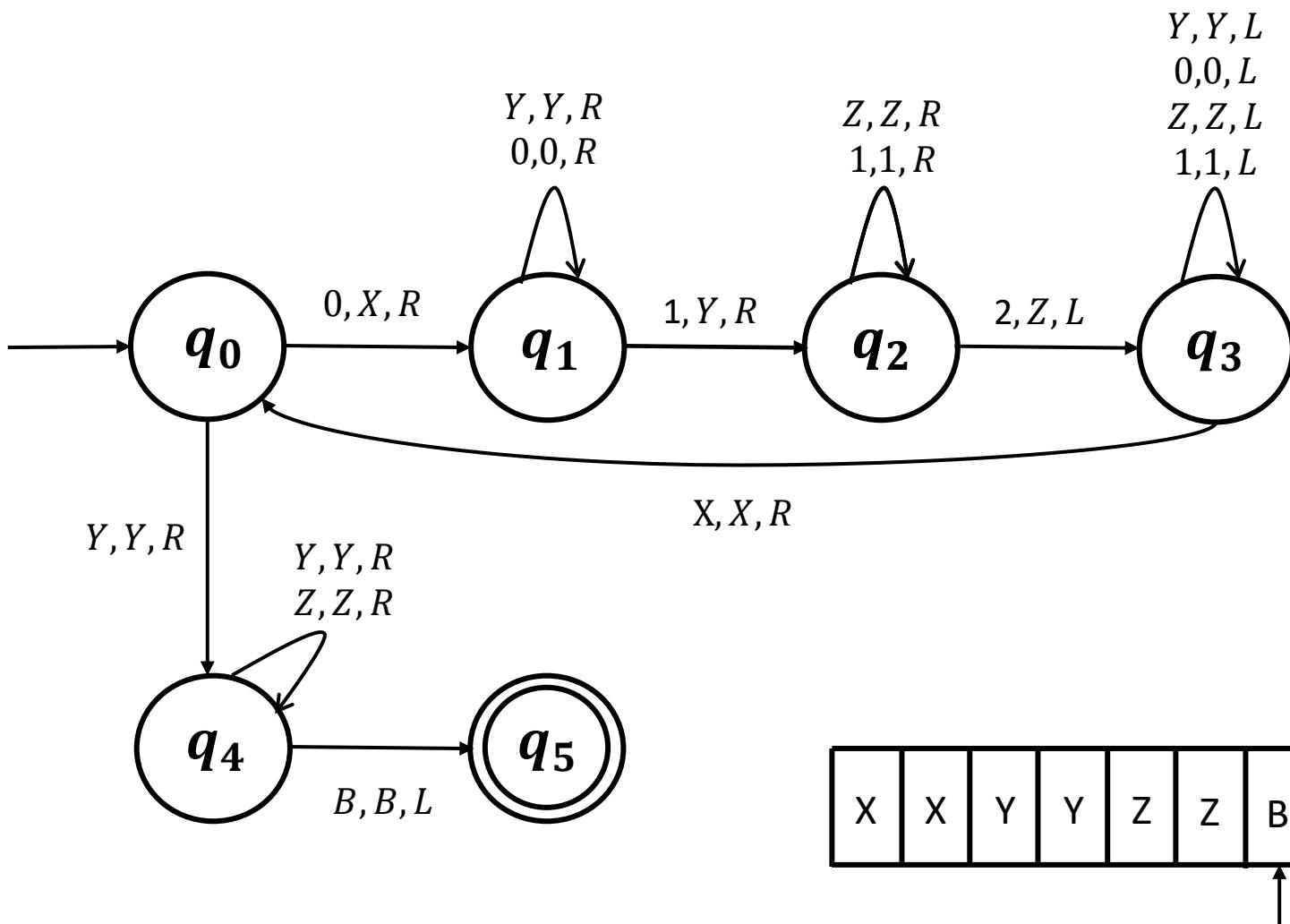
**Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$**



- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

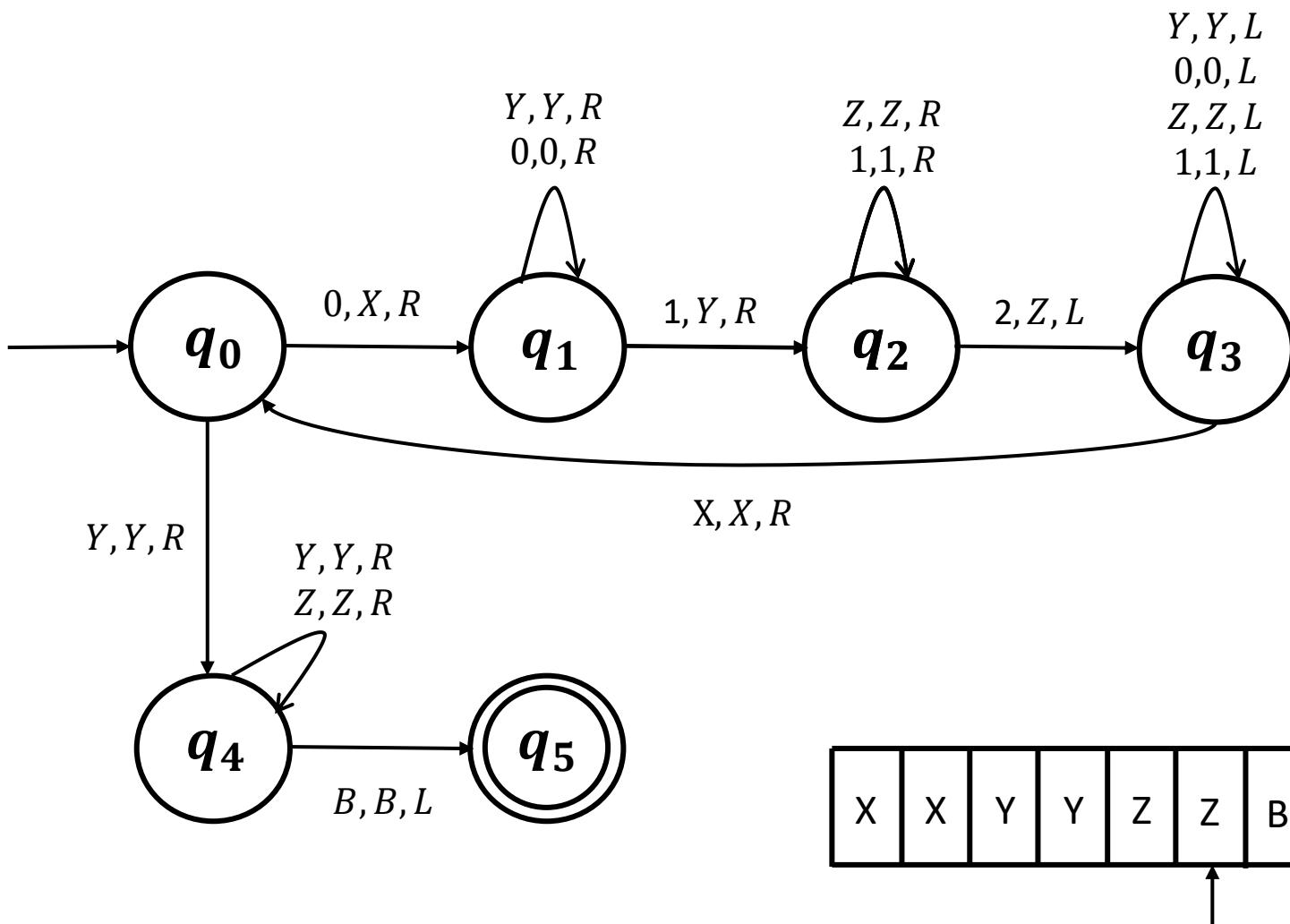
**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**



- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

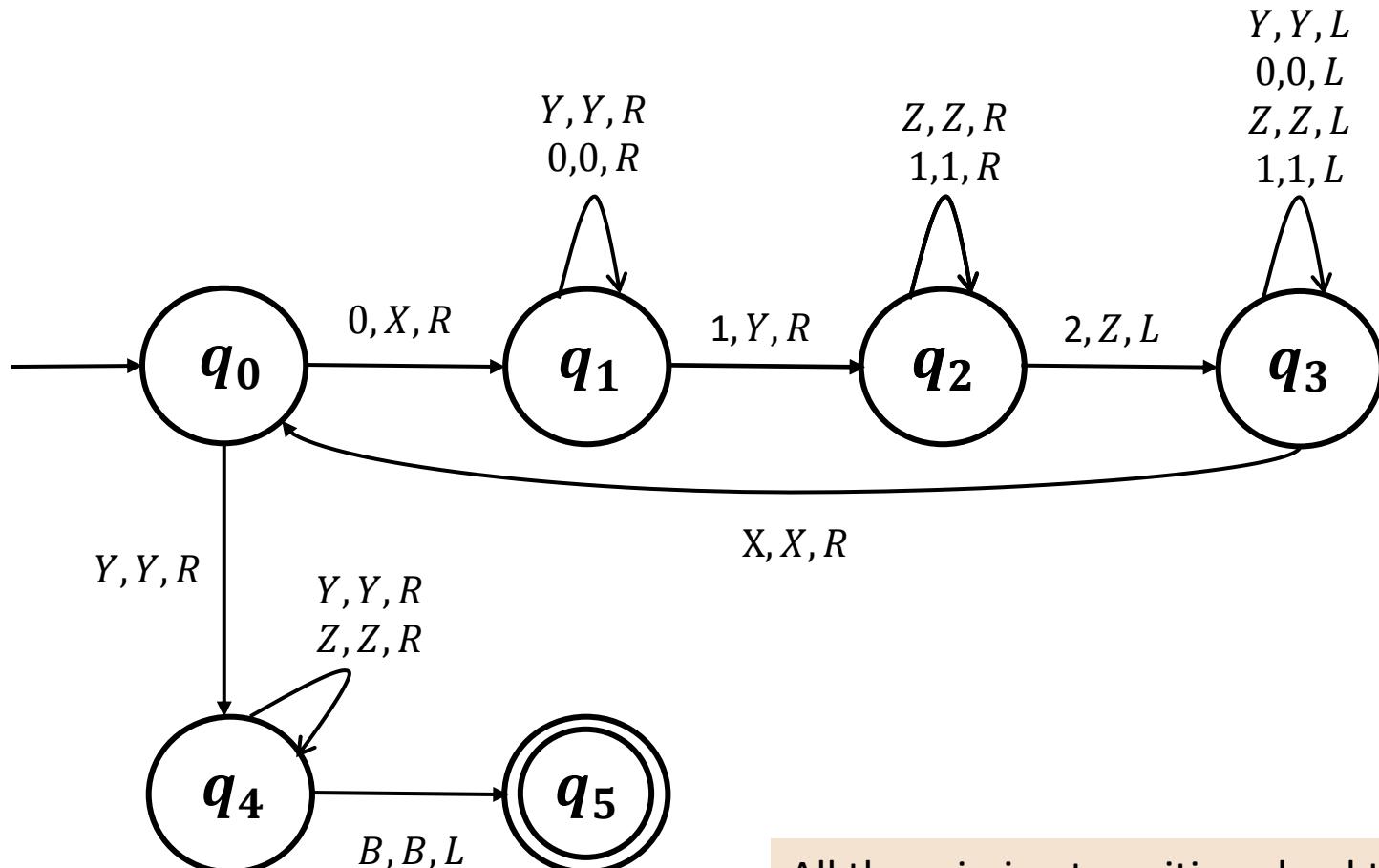
**Example: Let  $L = \{0^n 1^n 2^n | n \geq 1\}$**



- Continue to go right to mark the next 2 with a  $Z$ .
- Skip all the 1's and move right/ Also, move across (to the right) the  $Z$ 's already marked.
- Mark a new 2 with a  $Z$  and start moving left.
- Keep moving left until an  $X$  is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark  
or
- Skip across the  $Y$ 's and  $Z$ 's to the right end of the tape until a blank is found.
- Move left and accept the input string.

# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$

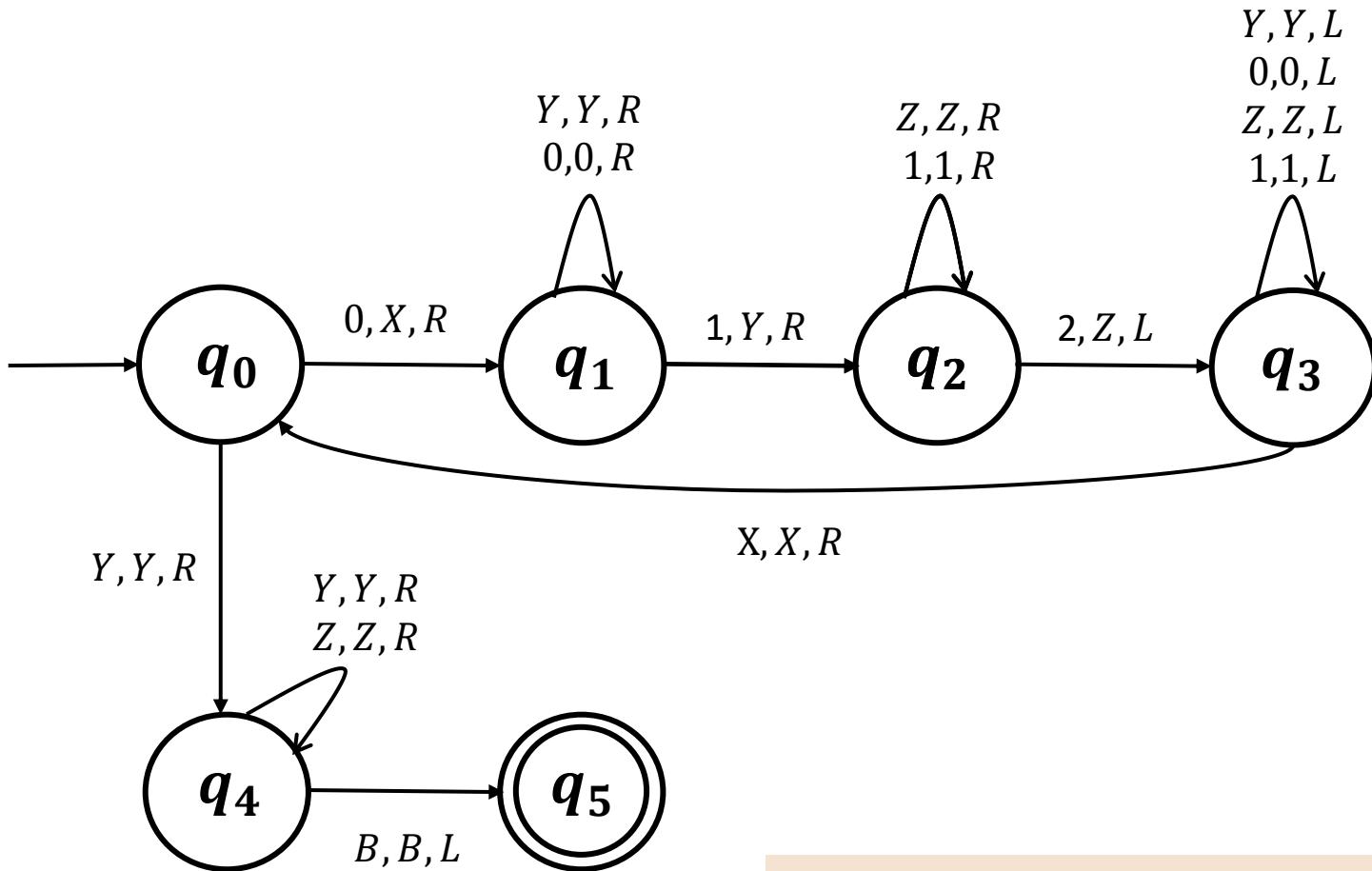


- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark
  - or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

All the missing transitions lead to a reject state and so any input not of the form  $\{0^n 1^n 2^n\}$  is rejected.

# Turing Machines

Example: Let  $L = \{0^n 1^n 2^n \mid n \geq 1\}$



- Continue to go right to mark the next 2 with a Z.
- Skip all the 1's and move right/ Also, move across (to the right) the Z's already marked.
- Mark a new 2 with a Z and start moving left.
- Keep moving left until an X is encountered.
- Either repeat this process if there are 0's, 1's or 2's left to mark
  - or
- Skip across the Y's and Z's to the right end of the tape until a blank is found.
- Move left and accept the input string.

$CFL \subseteq \text{Language recognized by TM}$

# Turing Machines

Formally, a Turing Machine is a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets** not containing the blank symbol  $B$ .
- $\Gamma$  is the **tape alphabet**, where  $B \subseteq \Gamma$  and  $\Sigma \subseteq \Gamma$ .
- $\delta: Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$  is the **transition function**
- $q_0 \in Q$  is the **start state**.
- $q_{accept} \in Q$  is the **accepting state**.
- $q_{reject} \in Q - \{q_{accept}\}$  is the **reject state**.

**Configuration of a TM:** Combination of the current tape contents, the current state and the current head location.

At each step, the Turing machine configuration changes. We say  $C_1$  **yields**  $C_2$  if the TM changes from  $C_1$  to  $C_2$  in one step.

**Start configuration:**

$0\ 0\ 0\ 1\ 1\ 1\ B\ B\ B\ B\ ...$   
↑  
 $q_0$

**Accept configuration:**

$X\ X\ X\ Y\ Y\ Y\ B\ B\ B\ B\ ...$   
↑  
 $q_4$

**Reject configuration:**

$X\ X\ X\ Y\ Y\ 0\ B\ B\ B\ B\ ...$   
↑  
 $q_{reject}$

A TM  $M$  **accepts**  $w$  if there exists a sequence of configurations  $C_1$  to  $C_k$ , where

- $C_1$  is the start configuration  $M$  on  $w$ .
- Each  $C_i$  yields  $C_{i+1}$ .
- $C_k$  is an accepting configuration

Language of a TM:

$$L = \{w | M \text{ accepts } w\}$$

$X\ 0\ 0\ 1\ 1\ 1\ B\ B\ B\ B\ ...$   
↑  
 $q_1$

# Turing Machines

Formally, a Turing Machine is a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$  where

- $Q$  is a finite set called the **states**.
- $\Sigma$  is the set of input **alphabets** not containing the blank symbol  $B$ .
- $\Gamma$  is the **tape alphabet**, where  $B \subseteq \Gamma$  and  $\Sigma \subseteq \Gamma$ .
- $\delta: Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$  is the **transition function**
- $q_0 \in Q$  is the **start state**.
- $q_{accept} \in Q$  is the **accepting state**.
- $q_{reject} \in Q - \{q_{accept}\}$  is the **reject state**.

A TM  $M$  accepts  $w$  if there exists a sequence of configurations  $C_1$  to  $C_k$ , where

- $C_1$  is the start configuration  $M$  on  $w$ .
- Each  $C_i$  yields  $C_{i+1}$ .
- $C_k$  is an accepting configuration

Language of a TM:

$$L = \{w \mid M \text{ accepts } w\}$$

**Configuration of a TM:**

- Combination of the current tape contents, the current state and the current head location.
- At each step, the Turing machine configuration changes. We say  $C_1$  **yields**  $C_2$  if the TM changes from  $C_1$  to  $C_2$  in one step.

**Next Lecture**

Languages **recognized** by TM vs Languages **decided** by TM  
Various TM model **variants**

**Thank You!**

# CS 302.1 - Automata Theory

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

**Recursive Language/Turing Decidable/Decidable:** A language  $L$  is called Recursive or Turing decidable or Decidable if there exists a Total Turing Machine  $M$  and

$$\left. \begin{array}{l} \forall \omega \in L, M(\omega) \text{ accepts} \\ \forall \omega \notin L, M(\omega) \text{ rejects} \end{array} \right\} \text{Halts on all inputs}$$

**The Church Turing thesis:** An algorithm can be written for a problem if and only if it is decidable, i.e. there exists a Total Turing machine that solves the problem. **Total TM  $\Leftrightarrow$  Algorithms!**

**Recursively Enumerable Language/Turing Recognizable (RE):** A language  $L$  is called Recursively Enumerable (RE) or Turing Recognizable if

$$\begin{array}{ll} \forall \omega \in L, M(\omega) \text{ accepts} & \\ \forall \omega \notin L, M(\omega) \text{ doesn't accept} & \text{(rejects or runs infinitely)} \end{array}$$

The standard TM model is quite robust. It can simulate other seemingly “powerful” variants such as

- Lazy TM
- Multi-tape TM
- Two-way infinite tape TM
- Non-deterministic TM

# Variants of TM models

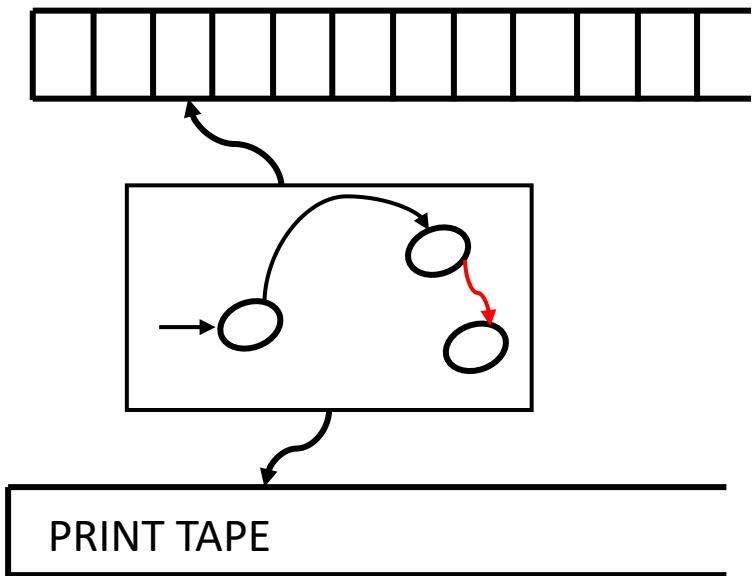
The standard TM model is quite robust. It can simulate other seemingly “powerful” variants such as

- Lazy TM
- Multi-tape TM
- Two-way infinite tape TM
- Non-deterministic TM

**Enumerators:** TM attached with a printer

- The Enumerator  $E$  uses the print tape to output strings
- The input tape is initially blank
- The language of  $E$  is the set of strings that it prints out
- If  $E$  does not halt, it may print infinitely many strings in some order

$$\mathcal{L}(E) = \{w \in \Sigma^* \mid w \text{ is printed by } E\}$$



# Variants of TM models

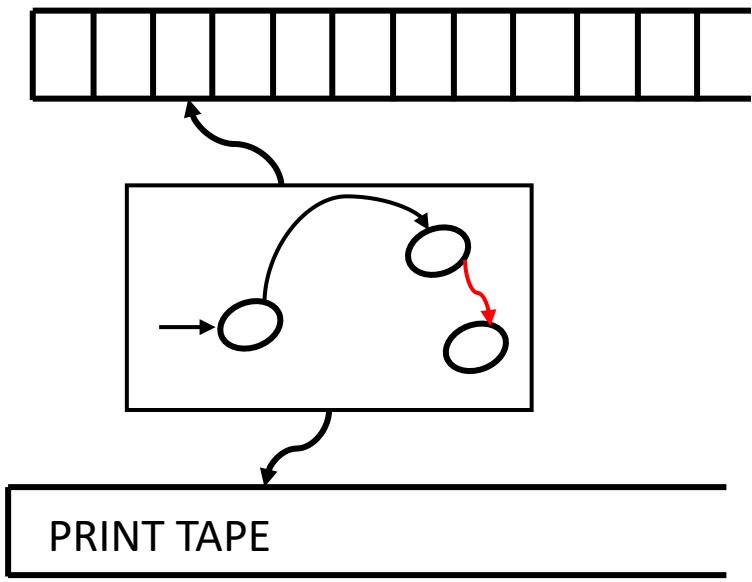
The standard TM model is quite robust. It can simulate other seemingly “powerful” variants such as

- Lazy TM
- Multi-tape TM
- Two-way infinite tape TM
- Non-deterministic TM

**Enumerators:** TM attached with a printer

- The Enumerator  $E$  uses the print tape to output strings
- The input tape is initially blank
- The language of  $E$  is the set of strings that it prints out
- If  $E$  does not halt, it may print infinitely many strings in some order

$$\mathcal{L}(E) = \{w \in \Sigma^* \mid w \text{ is printed by } E\}$$



- A language  $L$  is Recursively Enumerable or Turing-Recognizable iff some enumerator enumerates  $L$ .

# Variants of TM models

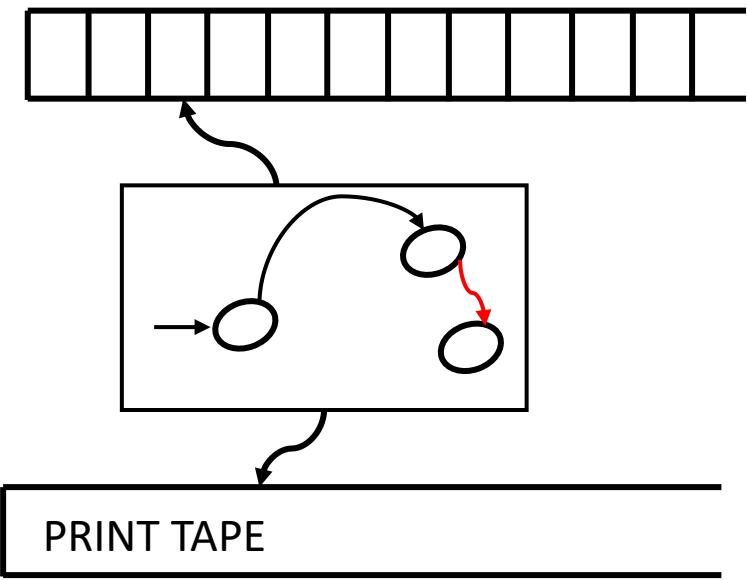
The standard TM model is quite robust. It can simulate other seemingly “powerful” variants such as

- Lazy TM
- Multi-tape TM
- Two-way infinite tape TM
- Non-deterministic TM

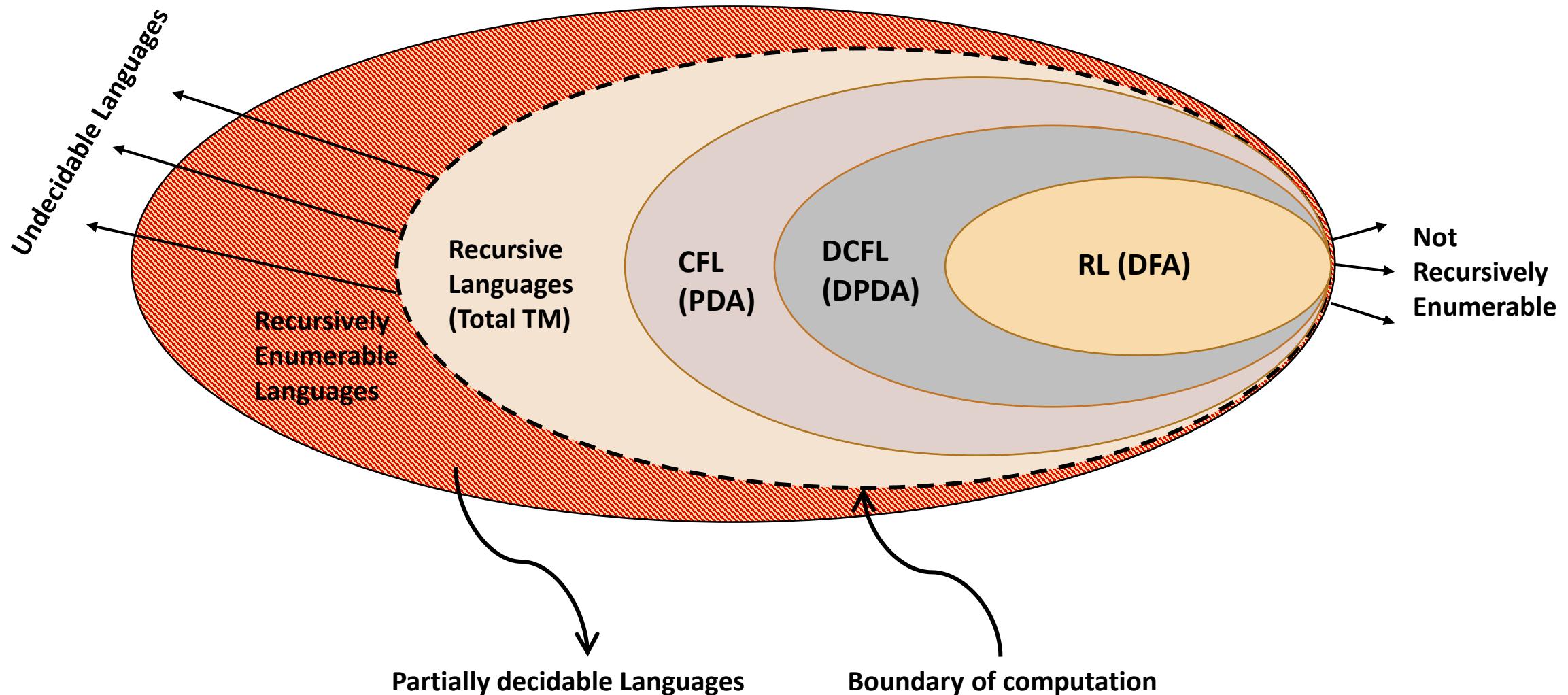
**Enumerators:** TM attached with a printer

- The Enumerator  $E$  uses the print tape to output strings
- The input tape is initially blank
- The language of  $E$  is the set of strings that it prints out
- If  $E$  does not halt, it may print infinitely many strings in some order

$$\mathcal{L}(E) = \{w \in \Sigma^* \mid w \text{ is printed by } E\}$$



- A language  $L$  is Recursively Enumerable or Turing-Recognizable iff some enumerator enumerates  $L$ .
- A language  $L$  is Recursive iff some enumerator enumerates  $L$  in lexicographic order.



# Encoding

The input to a TM are often strings/sequences of strings.

$M(w_1, w_2) =$  If  $w_1$  is a substring of  $w_2$ , ACCEPT  
Otherwise, REJECT.

Not just numbers, seemingly complicated objects such as a **graph, a DFA, a CFG and even a Turing Machine** itself can be encoded as a string – and hence can be an input to a TM.

# Encoding

The input to a TM are often strings/sequences of strings.

$M(w_1, w_2) =$  If  $w_1$  is a substring of  $w_2$ , ACCEPT  
Otherwise, REJECT.

Consider this example:

$M(\langle M_1 \rangle, w) =$  Run  $M_1$  on input  $w$ .  
If  $M_1(w)$  accepts, ACCEPT  
If  $M_1(w)$  rejects, REJECT

Not just numbers, seemingly complicated objects such as a **graph**, a **DFA**, a **CFG** and even a **Turing Machine** itself can be encoded as a string – and hence can be an input to a TM.

- $\langle M_1 \rangle$  is the encoding of TM  $M_1$  as a string.
- $M$  simulates the run of  $M_1$  on input  $w$ .
- Observe that  $M$  can accept a description of itself as input.

# Encoding

The input to a TM are often strings/sequences of strings.

$M(w_1, w_2) =$    If  $w_1$  is a substring of  $w_2$ , ACCEPT  
                    Otherwise, REJECT.

Consider this example:

$M(\langle M_1, w \rangle) =$    Run  $M_1$  on input  $w$ .  
                    If  $M_1(w)$  accepts, ACCEPT  
                    If  $M_1(w)$  rejects, REJECT

Not just numbers, seemingly complicated objects such as a **graph**, a **DFA**, a **CFG** and even a **Turing Machine** itself can be encoded as a string – and hence can be an input to a TM.

- $\langle M_1 \rangle$  is the encoding of TM  $M_1$  as a string.
- $M$  simulates the run of  $M_1$  on input  $w$ .
- Observe that  $M$  can accept a description of itself as input.

# Encoding

The input to a TM are often strings/sequences of strings.

$M(w_1, w_2) =$  If  $w_1$  is a substring of  $w_2$ , ACCEPT  
Otherwise, REJECT.

Consider this example:

$M(\langle M_1, w \rangle) =$  Run  $M_1$  on input  $w$ .  
If  $M_1(w)$  accepts, ACCEPT  
If  $M_1(w)$  rejects, REJECT

Not just numbers, seemingly complicated objects such as a **graph**, a **DFA**, a **CFG** and even a **Turing Machine** itself can be encoded as a string – and hence can be an input to a TM.

- $\langle M_1 \rangle$  is the encoding of TM  $M_1$  as a string.
- $M$  simulates the run of  $M_1$  on input  $w$ .
- Observe that  $M$  can accept a description of itself as input.

- Encoding objects such as TMs as strings will help define a Universal Turing Machine  $U_{TM}$  which is a DTM that accepts as input the encoding of a DTM  $M$  and an input string  $w$ , and simulates  $M(w)$ .
- To prove that problems related to regular languages, CFLs are decidable/undecidable, we need to provide encodings DFAs/CFGs as inputs to a TM.
- How can we encode objects as strings? We will show a simple encoding of a DTM into a binary string.

# Encoding a Turing Machine

- We will provide a simple mapping from a DTM to a **binary string**.
- Of course, this is not the only encoding.
- You can come up with your own encoding.

# Encoding a Turing Machine

Recall that a DTM  $M$  is a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ .

- Let  $Q = \{q_0, \dots, q_{m-1}\}$ ,  $\Sigma = \{0, 1, \dots, k-1\}$ ,  $\Gamma = \{0, 1, \dots, n-1\}$ . As  $\Sigma \subseteq \Gamma$ ,  $k < n$  and without loss of generality  $B$  corresponds to the last symbol  $n-1$  in  $\Gamma$ .

# Encoding a Turing Machine

Recall that a DTM  $M$  is a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ .

- Let  $Q = \{q_0, \dots, q_{m-1}\}$ ,  $\Sigma = \{0, 1, \dots, k-1\}$ ,  $\Gamma = \{0, 1, \dots, n-1\}$ . As  $\Sigma \subseteq \Gamma$ ,  $k < n$  and without loss of generality  $B$  corresponds to the last symbol  $n-1$  in  $\Gamma$ .

- Any state  $q_i \in Q$  can be encoded as a binary string, where

$$\langle q_0 \rangle = 0, \langle q_1 \rangle = 1, \langle q_2 \rangle = 10, \dots$$

- Any symbol in  $\Gamma$  (or  $\Sigma$ ) can be encoded as

$$\langle 0 \rangle = 0, \langle 1 \rangle = 1, \langle 2 \rangle = 10, \dots$$

- The directions  $\langle L \rangle = 0$  and  $\langle R \rangle = 1$ . So the transition function  $\delta(q_i, a) = (q_j, b, L/R)$  is just the sequence

$$\langle \langle q_i \rangle, \langle a \rangle, \langle q_j \rangle, \langle b \rangle, \langle L/R \rangle \rangle$$

- All such transitions are listed in lexicographic order into

$$\langle \delta \rangle = \langle \langle \delta_0 \rangle, \langle \delta_1 \rangle, \langle \delta_2 \rangle, \dots \rangle$$

# Encoding a Turing Machine

Recall that a DTM  $M$  is a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ .

Let  $Q = \{q_0, \dots, q_{m-1}\}$ ,  $\Sigma = \{0, 1, \dots, k - 1\}$ ,  $\Gamma = \{0, 1, \dots, n - 1\}$ .

Following these encodings we can simply encode the DTM  $M$  as

$$\begin{aligned} \langle M \rangle \\ = ( \langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle ) \end{aligned}$$

# Encoding a Turing Machine

Recall that a DTM  $M$  is a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ .

Let  $Q = \{q_0, \dots, q_{m-1}\}$ ,  $\Sigma = \{0, 1, \dots, k - 1\}$ ,  $\Gamma = \{0, 1, \dots, n - 1\}$ .

Following these encodings we can simply encode the DTM  $M$  as

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \langle q_0 \rangle, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

We are almost there but not quite. We have to find a way to combine this tuple of binary strings into one bigger binary string. Note that  $\langle \delta \rangle$  itself is a tuple of binary strings.

# Encoding a Turing Machine

Recall that a DTM  $M$  is a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ .

Let  $Q = \{q_0, \dots, q_{m-1}\}$ ,  $\Sigma = \{0, 1, \dots, k-1\}$ ,  $\Gamma = \{0, 1, \dots, n-1\}$ .

Following these encodings we can simply encode the DTM  $M$  as

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

We are almost there but not quite. We have to find a way to combine this tuple of binary strings into one bigger binary string. Note that  $\langle \delta \rangle$  itself is a tuple of binary strings.

We can combine multiple sequences of binary strings into one as follows. Consider the sequence

$$\langle \langle a_1 \rangle, \langle a_2 \rangle, \dots, \langle a_n \rangle \rangle = \langle \langle a_1 \rangle \# \langle a_2 \rangle \# \dots \# \langle a_n \rangle \rangle,$$

where  $a_i$  are binary strings of finite length.

We claim that using the following map suffices

$$0 \mapsto 00$$

$$1 \mapsto 01$$

$$\# \mapsto 1$$

# Encoding a Turing Machine

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, 0, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

We can combine multiple sequences of binary strings into one as follows. Consider the sequence

$$\langle \langle a_1 \rangle, \langle a_2 \rangle, \dots, \langle a_n \rangle \rangle = \langle \langle a_1 \rangle \# \langle a_2 \rangle \# \dots \# \langle a_n \rangle \rangle,$$

where  $a_i$  are binary strings of finite length.

We claim that using the following map suffices

$$\begin{aligned} 0 &\mapsto \textcolor{green}{00} \\ 1 &\mapsto \textcolor{red}{01} \\ \# &\mapsto \textcolor{blue}{1} \end{aligned}$$

Why does this work?

- For a 0 in an odd position, the symbol immediately following it corresponds to the symbol that was encoded
- We can identify the delimiter as the 1 that appears in an odd position.

# Encoding a Turing Machine

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, 0, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

We can combine multiple sequences of binary strings into one as follows. Consider the sequence

$$\langle \langle a_1 \rangle, \langle a_2 \rangle, \dots, \langle a_n \rangle \rangle = \langle \langle a_1 \rangle \# \langle a_2 \rangle \# \dots \# \langle a_n \rangle \rangle,$$

where  $a_i$  are binary strings of finite length.

We claim that using the following map suffices

$$\begin{aligned} 0 &\mapsto \textcolor{green}{00} \\ 1 &\mapsto \textcolor{red}{01} \\ \# &\mapsto \textcolor{blue}{1} \end{aligned}$$

E.g. Let  $a_1 = 1101$  and  $a_2 = 010$ . Then  $\langle 1101, 010 \rangle \mapsto \textcolor{red}{0101} \textcolor{green}{0001} \textcolor{blue}{1} \textcolor{green}{0001} \textcolor{red}{00}$

# Encoding a Turing Machine

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, 0, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

We can combine multiple sequences of binary strings into one as follows. Consider the sequence

$$\langle \langle a_1 \rangle, \langle a_2 \rangle, \dots, \langle a_n \rangle \rangle = \langle \langle a_1 \rangle \# \langle a_2 \rangle \# \dots \# \langle a_n \rangle \rangle,$$

where  $a_i$  are binary strings of finite length.

We claim that using the following map suffices

$$\begin{aligned} 0 &\mapsto \textcolor{green}{00} \\ 1 &\mapsto \textcolor{red}{01} \\ \# &\mapsto \textcolor{blue}{1} \end{aligned}$$

E.g. Let  $a_1 = 1101$  and  $a_2 = 010$ . Then  $\langle 1101, 010 \rangle \mapsto \textcolor{red}{0101} \textcolor{green}{0001} \textcolor{red}{1} \textcolor{green}{0001} \textcolor{red}{00}$

To recover  $a_1$  and  $a_2$  from the encoding:

- For any 0 in odd positions, the symbol that follow in the even positions, belong to  $a_1$ .
- If a 1 is obtained in an odd position, it corresponds to the delimiter/partition  $\Rightarrow a_1$  has been recovered, now  $a_2$  will be obtained similarly.
- This can be generalized to multiple tuples of binary strings which is what we need to encode  $M$ .

# Encoding a Turing Machine

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, 0, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

Every DTM corresponds to a binary string but the reverse is not necessarily true. Some binary strings are not valid descriptions of DTMs.

Can we make this a bijection?

# Encoding a Turing Machine

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, 0, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

Every DTM corresponds to a binary string but the reverse is not necessarily true. Some binary strings are not valid descriptions of DTMs.

Can we make this a bijection?

- Lexicographically generate binary strings.
- For any length  $k$ , there are  $2^k$  binary strings of length  $k$
- So any TM that can be described by a  $k$ -length binary string will be within this finite set.
- Some of these will not correspond to a valid DTM. Ignore them.
- Relabel the first binary string that corresponds to a valid TM as 0.
- Relabel the second binary string that corresponds to a valid TM as 1.

0  
1  
01  
...  
**00000011**  
...  
**0001100001**

Say this is the first binary string that corresponds to a valid TM.  
Relabel this as 0

If this is the next binary string, corresponding to a valid TM, relabel this as 1 and so on.

# Encoding a Turing Machine

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, 0, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

- Lexicographically generate binary strings.
- For any length  $k$ , there are  $2^k$  binary strings of length  $k$
- So any TM that can be described by a  $k$ -length binary string will be within this finite set.
- Some of these will not correspond to a valid DTM. Ignore them.
- Relabel the first binary string that corresponds to a valid TM as 0.
- Relabel the second binary string that corresponds to a valid TM as 1.

Every DTM corresponds to a binary string but the reverse is not necessarily true. Some binary strings are not valid descriptions of DTMs.

Can we make this a bijection?

0  
1  
01  
...  
**00000011**  
...  
**0001100001**

Say this is the first binary string that corresponds to a valid TM. Relabel this as 0

If this is the next binary string, corresponding to a valid TM, relabel this as 1 and so on.

Now we have a one-one mapping (bijective relationship) between the set of binary strings and DTMs.

# Universal Turing Machines

Now that we have shown how to encode objects including Turing Machines as binary strings, we can now define **Universal Turing Machines** – or Turing Machines that simulate other Turing Machines.

**Universal Turing Machine:** A Universal Turing Machine, denoted as  $U_{TM}$  accepts as input (i) the encoding of a Turing Machine  $M$  and (ii) an input string  $w$  and **simulates  $M$  running on  $w$** , i.e.

$$U_{TM}(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects} \\ \text{LOOPS INFINITELY, if } M(w) \text{ loops infinitely} \end{cases}$$

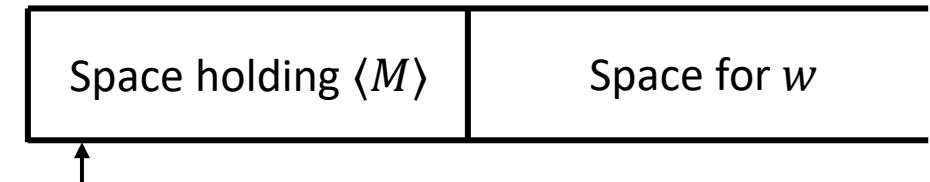
By the Church-Turing thesis, a  $U_{TM}$  can perform any computation on any feasible computational device.

So in principle using  $U_{TM}$ , Turing Machines can answer questions about Turing Machines!

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .



$U_{TM}$  checks

- the space for  $w$  to determine the symbol currently being read
- And the space containing  $\langle M \rangle$  for determining the transition function to be implemented

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

Examples: The following languages are decidable

- $A_{DFA} = \{\langle DFA, w \rangle | w \in L(DFA)\}$

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

$M$  = On input  $\langle DFA, w \rangle$ :

- Simulate the run of  $\langle DFA \rangle$  on  $w$ .
- If  $w$  is accepted, output *ACCEPT*
- If  $w$  is rejected, output *REJECT*

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

Examples: The following languages are decidable

- $A_{DFA} = \{\langle DFA, w \rangle | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

Examples: The following languages are decidable

- $A_{DFA} = \{\langle DFA, w \rangle | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

$M =$  On input  $\langle DFA \rangle$ :

- Mark the start state of  $\langle DFA \rangle$
- Repeat until no new states are marked
  - Mark any state that has an incoming transition from a marked state
- If the final state is unmarked, *ACCEPT*, else *REJECT*

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

Examples: The following languages are decidable

- $A_{DFA} = \{\langle DFA \rangle, w | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle | w \in L(CFG)\}$

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

Examples: The following languages are decidable

- $A_{DFA} = \{\langle DFA \rangle, w | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle | w \in L(CFG)\}$

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

$M$  = On input  $\langle CFG, w \rangle$ :

- Convert  $\langle CFG \rangle$  into CNF
- List all derivations of  $2|w| - 1$  steps
- If any of these derivations yield  $w$ , ACCEPT, else REJECT

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

Examples: The following languages are decidable

- $A_{DFA} = \{\langle DFA \rangle, w | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle | w \in L(CFG)\}$

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

$M$  = On input  $\langle CFG, w \rangle$ :

- Convert  $\langle CFG \rangle$  into CNF
- List all derivations of  $2|w| - 1$  steps
- If any of these derivations yield  $w$ , ACCEPT, else REJECT

Or, run the CYK algorithm

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

So for any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

Examples: The following languages are decidable

- $A_{DFA} = \{\langle DFA \rangle, w | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle | w \in L(CFG)\}$
- $E_{CFG} = \{\langle CFG, w \rangle | L(CFG) = \Phi\}$

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

For any DTM  $M$ , we obtain an encoding

$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

Examples: The following languages are decidable

- $A_{DFA} = \{\langle DFA \rangle, w | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle | w \in L(CFG)\}$
- $E_{CFG} = \{\langle CFG, w \rangle | L(CFG) = \Phi\}$

**Idea similar to DFAs:** Check if the Start Variable leads to any terminal

# Some Decidable Languages

Much like Turing Machines, DFAs, NFAs, CFGs can also be encoded as binary strings. In fact, a bijection can be established between binary strings and these objects.

This is useful as it helps answer the decidability of Languages related to them.

Examples: The following languages are decidable

- $A_{DFA} = \{\langle DFA \rangle, w | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle | w \in L(CFG)\}$
- $E_{CFG} = \{\langle CFG, w \rangle | L(CFG) = \Phi\}$

So for any DTM  $M$ , we obtain an encoding

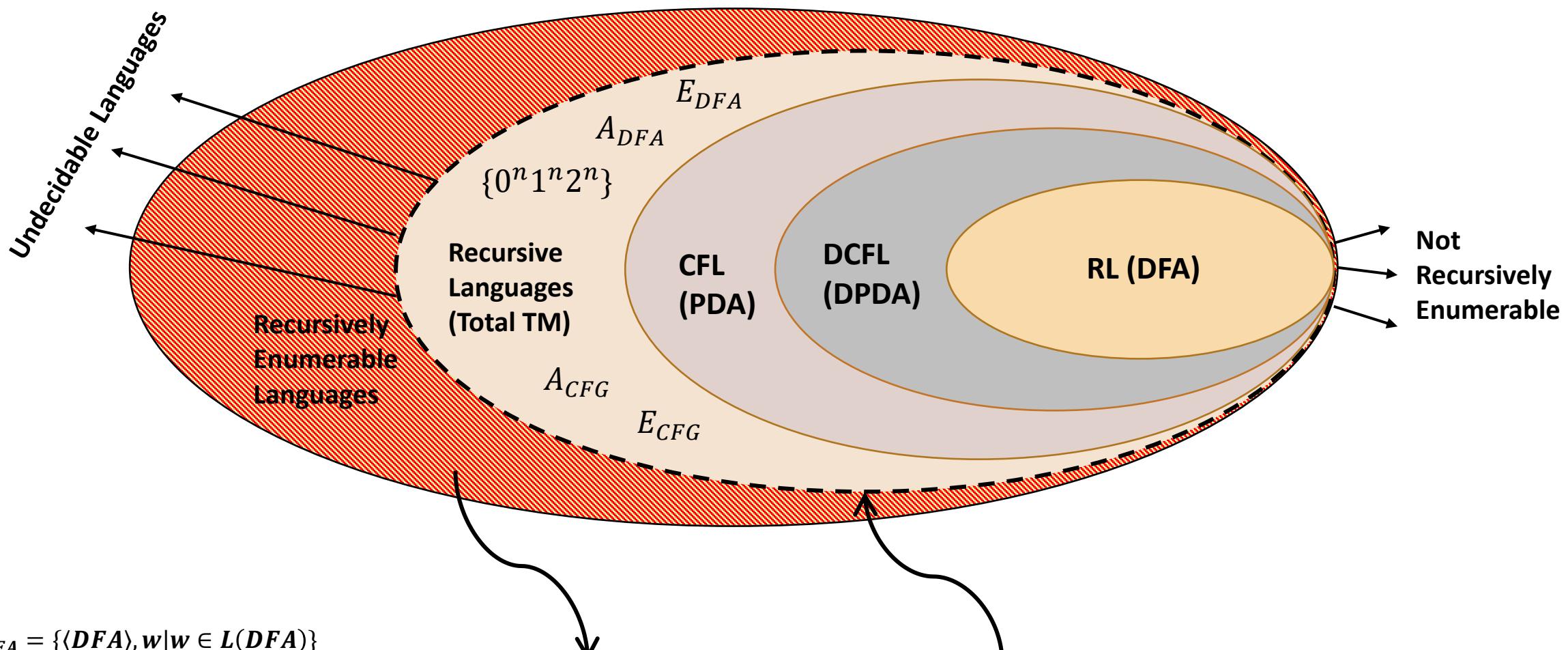
$$\langle M \rangle = (\langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \mathbf{0}, \langle q_{accept} \rangle, \langle q_{reject} \rangle)$$

such that  $\langle M \rangle \in \{0,1\}^*$ .

$M$  = On input  $\langle CFG \rangle$ :

- Mark all terminal symbols
- Repeat until no new variables are marked
  - Mark any  $V$ , s.t.  $V \rightarrow X_1X_2 \cdots X_l$ , and each  $X_k$  (variable or terminal) is already marked
- If  $S$  is unmarked, ACCEPT. Else REJECT

# Some Decidable Languages

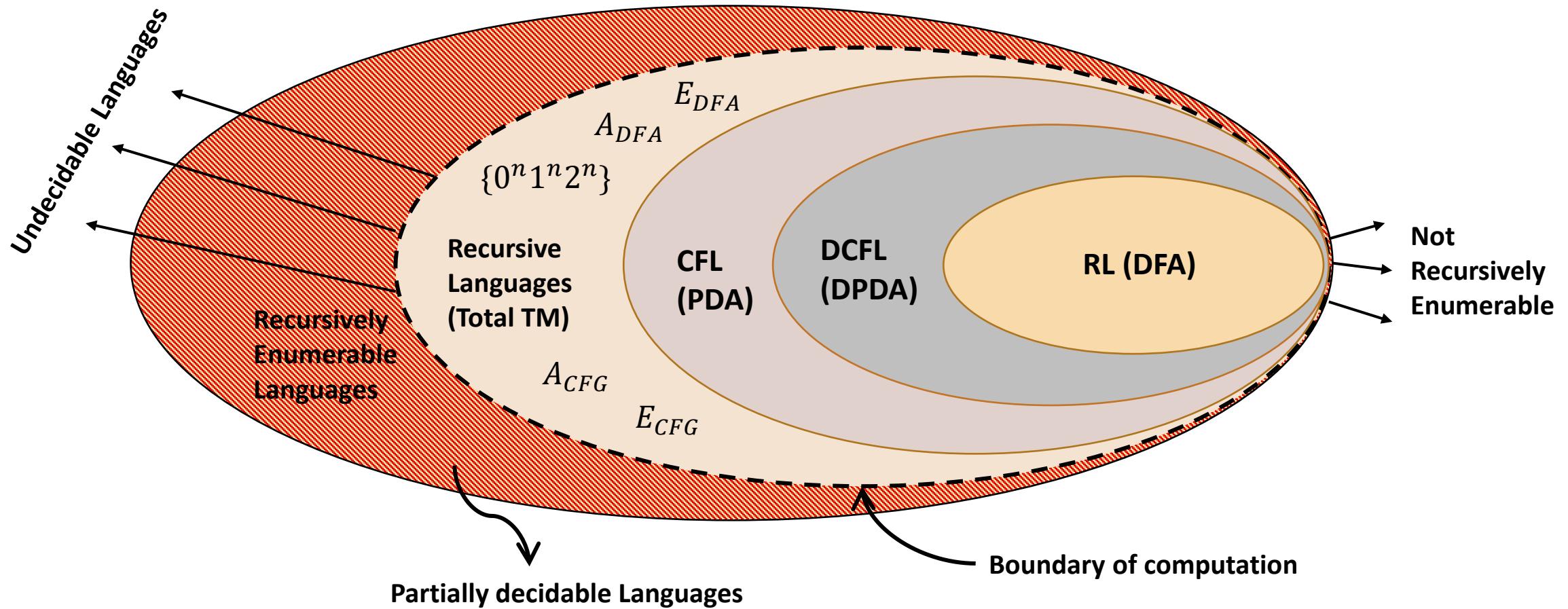


- $A_{DFA} = \{\langle DFA \rangle, w \mid w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle \mid L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle \mid w \in L(CFG)\}$
- $E_{CFG} = \{\langle CFG, w \rangle \mid L(CFG) = \Phi\}$

Partially decidable Languages

Boundary of computation

# Some Decidable Languages



- $A_{DFA} = \{\langle DFA \rangle, w | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle | w \in L(CFG)\}$
- $E_{CFG} = \{\langle CFG, w \rangle | L(CFG) = \Phi\}$

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$A_{TM}$ : Does there exist a Total Turing Machine  $A$  that accepts as input a Turing Machine  $M$  and an input string  $w$  and outputs ACCEPT, if  $M(w)$  accepts  $w$  and REJECT, if  $M(w)$  does not accept  $w$  (rejects or loops forever)?

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$A_{TM}$ : Does there exist a Total Turing Machine  $A$  that accepts as input a Turing Machine  $M$  and an input string  $w$  and outputs ACCEPT, if  $M(w)$  accepts  $w$  and REJECT, if  $M(w)$  does not accept  $w$  (rejects or loops forever)?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$A_{TM}$ : Does there exist a Total Turing Machine  $A$  that accepts as input a Turing Machine  $M$  and an input string  $w$  and outputs ACCEPT, if  $M(w)$  accepts  $w$  and REJECT, if  $M(w)$  does not accept  $w$  (rejects or loops forever)?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

Every binary string is a TM and vice versa. So the **input may have two copies of the same string (say  $w$ )**:

- The first copy corresponds the encoding of some TM  $w = \langle M_w \rangle$ .
- The second copy is the input string  $w = \langle M_w \rangle$ .

$$A(\langle w, w \rangle)$$

In this case,  $A$  simulates the run of TM  $M_w$  on the input string  $w$ , which is the binary encoding of  $M_w$  itself

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$A_{TM}$ : Does there exist a Total Turing Machine  $A$  that accepts as input a Turing Machine  $M$  and an input string  $w$  and outputs ACCEPT, if  $M(w)$  accepts  $w$  and REJECT, if  $M(w)$  does not accept  $w$  (rejects or loops forever)?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

There can be inputs such as  $A(\langle w, w \rangle)$

Let  $w = \langle M_w \rangle$

$$A(\langle w, w \rangle) = \begin{cases} \text{ACCEPTS, if } M_w(w) \text{ accepts} \\ \text{REJECTS, if } M_w(w) \text{ rejects or loops infinitely} \end{cases}$$

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$A_{TM}$ : Does there exist a Total Turing Machine  $A$  that accepts as input a Turing Machine  $M$  and an input string  $w$  and outputs ACCEPT, if  $M(w)$  accepts  $w$  and REJECT, if  $M(w)$  does not accept  $w$  (rejects or loops forever)?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

There can be inputs such as  $A(\langle w, w \rangle)$

Let  $w = \langle M_w \rangle$

$$A(\langle w, w \rangle) = \begin{cases} \text{ACCEPTS, if } M_w(w) \text{ accepts} \\ \text{REJECTS, if } M_w(w) \text{ rejects or loops infinitely} \end{cases}$$

≡

$$A(\langle w, w \rangle) = \begin{cases} \text{ACCEPTS, if } M_w(\langle M_w \rangle) \text{ accepts} \\ \text{REJECTS, if } M_w(\langle M_w \rangle) \text{ rejects or loops infinitely} \end{cases}$$

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$A_{TM}$ : Does there exist a Total Turing Machine  $A$  that accepts as input a Turing Machine  $M$  and an input string  $w$  and outputs ACCEPT, if  $M(w)$  accepts  $w$  and REJECT, if  $M(w)$  does not accept  $w$  (rejects or loops forever)?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

- We will show that if such a Total TM  $A$  exists, we run into the following contradiction

Using  $A$ , we can build a new Total TM for which there exists an instance for which the machine **both accepts and rejects!**

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

**Proof:** Let us assume that a Total Turing machine  $A$  exists. Then we can construct a special Total Turing Machine  $D$  that accepts an input  $w$  and uses  $A$  as a subroutine to simulate  $A(\langle w, w \rangle)$  in the following way:

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

**Proof:** Let us assume that a Total Turing machine  $A$  exists. Then we can construct a special Total Turing Machine  $D$  that accepts an input  $w$  and uses  $A$  as a subroutine to simulate  $A(\langle w, w \rangle)$  in the following way:

$$D(w) = \{ \text{Run } A(\langle w, w \rangle)$$

If  $A(\langle w, w \rangle)$  accepts, then  $D$  outputs REJECT  
If  $A(\langle w, w \rangle)$  rejects, then  $D$  outputs ACCEPT

}

Note: We have already established that any binary string  $w$  can be both a input string as well the encoding of a Turing Machine.

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

**Proof:** Let us assume that a Total Turing machine  $A$  exists. Then we can construct a special Total Turing Machine  $D$  that accepts an input  $w$  and uses  $A$  as a subroutine to simulate  $A(\langle w, w \rangle)$  in the following way:

$$D(w) = \{ \text{Run } A(\langle w, w \rangle)$$

If  $A(\langle w, w \rangle)$  accepts, then  $D$  outputs **REJECT**  
If  $A(\langle w, w \rangle)$  rejects, then  $D$  outputs **ACCEPT**  
}

Let  $w = \langle M_w \rangle$ , then

$$D(\langle M_w \rangle) = \begin{cases} \text{ACCEPTS, if } M_w(\langle M_w \rangle) \text{ doesn't accept} \\ \text{REJECTS, if } M_w(\langle M_w \rangle) \text{ accepts} \end{cases}$$

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

**Proof:** Let us assume that a Total Turing machine  $A$  exists. Then we can construct a special Total Turing Machine  $D$  that accepts an input  $w$  and uses  $A$  as a subroutine to simulate  $A(\langle w, w \rangle)$  in the following way:

$$D(w) = \{ \text{Run } A(\langle w, w \rangle)$$

If  $A(\langle w, w \rangle)$  accepts, then  $D$  outputs **REJECT**

If  $A(\langle w, w \rangle)$  rejects, then  $D$  outputs **ACCEPT**

}

Let  $w = \langle M_w \rangle$ , then

$$D(\langle M_w \rangle) = \begin{cases} \text{ACCEPTS, if } M_w(\langle M_w \rangle) \text{ doesn't accept} \\ \text{REJECTS, if } M_w(\langle M_w \rangle) \text{ accepts} \end{cases}$$

What happens when  $w = \langle D \rangle$  i.e.,  $M_w = D$ ?

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

**Proof:** Let us assume that a Total Turing machine  $A$  exists. Then we can construct a special Total Turing Machine  $D$  that accepts an input  $w$  and uses  $A$  as a subroutine to simulate  $A(\langle w, w \rangle)$  in the following way:

$D(w) = \{ \text{Run } A(\langle w, w \rangle)$

What happens when  $w = \langle D \rangle$  i.e.,  $M_w = D$ ?

If  $A(\langle w, w \rangle)$  accepts, then  $D$  outputs REJECT  
If  $A(\langle w, w \rangle)$  rejects, then  $D$  outputs ACCEPT  
}

$$D(\langle D \rangle) = \begin{cases} \text{ACCEPTS, if } D(\langle D \rangle) \text{ doesn't accept} \\ \text{REJECTS, if } D(\langle D \rangle) \text{ accepts} \end{cases}$$

Let  $w = \langle M_w \rangle$ , then  $D(\langle M_w \rangle) = \begin{cases} \text{ACCEPTS, if } M_w(\langle M_w \rangle) \text{ doesn't accept} \\ \text{REJECTS, if } M_w(\langle M_w \rangle) \text{ accepts} \end{cases}$

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

**Proof:** Let us assume that a Total Turing machine  $A$  exists. Then we can construct a special Total Turing Machine  $D$  that accepts an input  $w$  and uses  $A$  as a subroutine to simulate  $A(\langle w, w \rangle)$  in the following way:

$$D(w) = \{ \text{Run } A(\langle w, w \rangle)$$

What happens when  $w = \langle D \rangle$  i.e.,  $M_w = D$ ?

If  $A(\langle w, w \rangle)$  accepts, then  $D$  outputs **REJECT**

If  $A(\langle w, w \rangle)$  rejects, then  $D$  outputs **ACCEPT**

}

Let  $w = \langle M_w \rangle$ , then

$$D(\langle M_w \rangle) = \begin{cases} \text{ACCEPTS, if } M_w(\langle M_w \rangle) \text{ doesn't accept} \\ \text{REJECTS, if } M_w(\langle M_w \rangle) \text{ accepts} \end{cases}$$

- $D$  cannot be a Total TM as it cannot decide input  $\langle D \rangle$ .
- If a total TM  $A$  existed we could have constructed a total TM  $D$ .
- So a total TM  $A$  cannot exist and hence  $A_{TM}$  is not decidable.

CONTRADICTION!

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

Is  $A_{TM} \in RE$  ?

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

Of course,  $A_{TM} \in RE$  as  $A$  halts whenever  $M$  accepts  $w$  and so

$U$  = On input  $\langle M, w \rangle$ :

- Simulate  $M$  on input  $w$
- If  $M$  accepts  $w$ , ACCEPT; if  $M$  rejects  $w$ , REJECT

$U$  recognizes  $A_{TM}$

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ . Is  $A_{TM}$  decidable?

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

Of course,  $A_{TM} \in RE$  as  $A$  halts whenever  $M$  accepts  $w$  and so

$U$  = On input  $\langle M, w \rangle$ :

- Simulate  $M$  on input  $w$
- If  $M$  accepts  $w$ , ACCEPT; if  $M$  rejects  $w$ , REJECT

$U$  recognizes  $A_{TM}$

- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ .  $A_{TM}$  is undecidable

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

The proof uses a technique called **Diagonalization**.

First, recall that there exists a bijective map (one-one correspondence) between the set of all binary strings and Turing Machines.

We can list all the Turing Machines and write down the result of running any  $M_i$  on input  $\langle M_j \rangle$ .

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ .  $A_{TM}$  is undecidable

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

The proof uses a technique called **Diagonalization**.

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
$M_0$	Accept	Accept	Loops	Reject	Accept	$\cdots$
$M_1$	Accept	Reject	Reject	Accept	Reject	$\cdots$
$M_2$	Reject	Loops	Accept	Loops	Accept	$\cdots$
$M_3$	Accept	Reject	Reject	Accept	Reject	$\cdots$
$M_4$	Accept	Accept	Accept	Accept	Reject	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$

First, recall that there exists a bijective map (one-one correspondence) between the set of all binary strings and Turing Machines.

We can list all the Turing Machines and write down the result of running any  $M_i$  on input  $\langle M_j \rangle$ .

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ .  $A_{TM}$  is undecidable

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

The proof uses a technique called **Diagonalization**.

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...
$M_0$	Accept	Accept	Loops	Reject	Accept	...
$M_1$	Accept	Reject	Reject	Accept	Reject	...
$M_2$	Reject	Loops	Accept	Loops	Accept	...
$M_3$	Accept	Reject	Reject	Accept	Reject	...
$M_4$	Accept	Accept	Accept	Accept	Reject	...
:	:	:	:	:	:	...

How would this Table look for  $A$ ?

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ .  $A_{TM}$  is undecidable

$$A(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects or loops infinitely} \end{cases}$$

- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

The proof uses a technique called **Diagonalization**.

How would this Table look for  $A$ ?

	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...
$M_0$	Accept	Accept	Loops	Reject	Accept	...
$M_1$	Accept	Reject	Reject	Accept	Reject	...
$M_2$	Reject	Loops	Accept	Loops	Accept	...
$M_3$	Accept	Reject	Reject	Accept	Reject	...
$M_4$	Accept	Accept	Accept	Accept	Reject	...
:	:	:	:	:	:	...

$A$	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...
$M_0$	Accept	Accept	<b>Reject</b>	Reject	Accept	...
$M_1$	Accept	Reject	Reject	Accept	Reject	...
$M_2$	Reject	<b>Reject</b>	Accept	<b>Reject</b>	Accept	...
$M_3$	Accept	Reject	Reject	Accept	Reject	...
$M_4$	Accept	Accept	Accept	Accept	Reject	...
:	:	:	:	:	:	...

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ .  $A_{TM}$  is undecidable

The proof uses a technique called **Diagonalization**.

$A$	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...	$\langle D \rangle$	...
$M_0$	Accept	Accept	Reject	Reject	Accept	...	Accept	...
$M_1$	Accept	Reject	Reject	Accept	Reject	...	Accept	...
$M_2$	Reject	Reject	Accept	Reject	Accept	...	Accept	...
$M_3$	Accept	Reject	Reject	Accept	Reject	...	Reject	...
$M_4$	Accept	Accept	Accept	Accept	Reject	...	Reject	...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...
$D$						...		...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...

- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

$D(w) = \{ \text{ Run } A(\langle w, w \rangle)$

If  $A(\langle w, w \rangle)$  accepts, then REJECT  
If  $A(\langle w, w \rangle)$  rejects, then ACCEPT

}

$$D(\langle M_w \rangle) = \begin{cases} \text{ACCEPTS, if } M_w(\langle M_w \rangle) \text{ doesn't accept} \\ \text{REJECTS, if } M_w(\langle M_w \rangle) \text{ accepts} \end{cases}$$

- Somewhere we will also have the TM  $D$ .
- Note that  $D$  by definition computes the opposite of the diagonal entries of the table.

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ .  $A_{TM}$  is undecidable

The proof uses a technique called **Diagonalization**.

- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

Note that  $D$  by definition **computes the opposite of the diagonal entries** of the table.

<b><math>A</math></b>	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...	$\langle D \rangle$	...
$M_0$	Accept	Accept	Reject	Reject	Accept	...	Accept	...
$M_1$	Accept	Reject	Reject	Accept	Reject	...	Accept	...
$M_2$	Reject	Reject	Accept	Reject	Accept	...	Accept	...
$M_3$	Accept	Reject	Reject	Accept	Reject	...	Reject	...
$M_4$	Accept	Accept	Accept	Accept	Reject	...	Reject	...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...
<b><math>D</math></b>	Reject					...		...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ .  $A_{TM}$  is undecidable

The proof uses a technique called **Diagonalization**.

- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

Note that  $D$  by definition **computes the opposite of the diagonal entries** of the table.

<b><math>A</math></b>	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...	$\langle D \rangle$	...
$M_0$	Accept	Accept	Reject	Reject	Accept	...	Accept	...
$M_1$	Accept	Reject	Reject	Accept	Reject	...	Accept	...
$M_2$	Reject	Reject	Accept	Reject	Accept	...	Accept	...
$M_3$	Accept	Reject	Reject	Accept	Reject	...	Reject	...
$M_4$	Accept	Accept	Accept	Accept	Reject	...	Reject	...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...
<b><math>D</math></b>	Reject	Accept				...		...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ .  $A_{TM}$  is undecidable

The proof uses a technique called **Diagonalization**.

- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

Note that  $D$  by definition **computes the opposite of the diagonal entries** of the table.

<b><math>A</math></b>	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...	$\langle D \rangle$	...
$M_0$	Accept	Accept	Reject	Reject	Accept	...	Accept	...
$M_1$	Accept	Reject	Reject	Accept	Reject	...	Accept	...
$M_2$	Reject	Reject	Accept	Reject	Accept	...	Accept	...
$M_3$	Accept	Reject	Reject	Accept	Reject	...	Reject	...
$M_4$	Accept	Accept	Accept	Accept	Reject	...	Reject	...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...
<b><math>D</math></b>	Reject	Accept	Reject	Reject	Accept	...		...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...

What will be the  $D^{th}$  diagonal entry??

# An undecidable problem

$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$ .  $A_{TM}$  is undecidable

The proof uses a technique called **Diagonalization**.

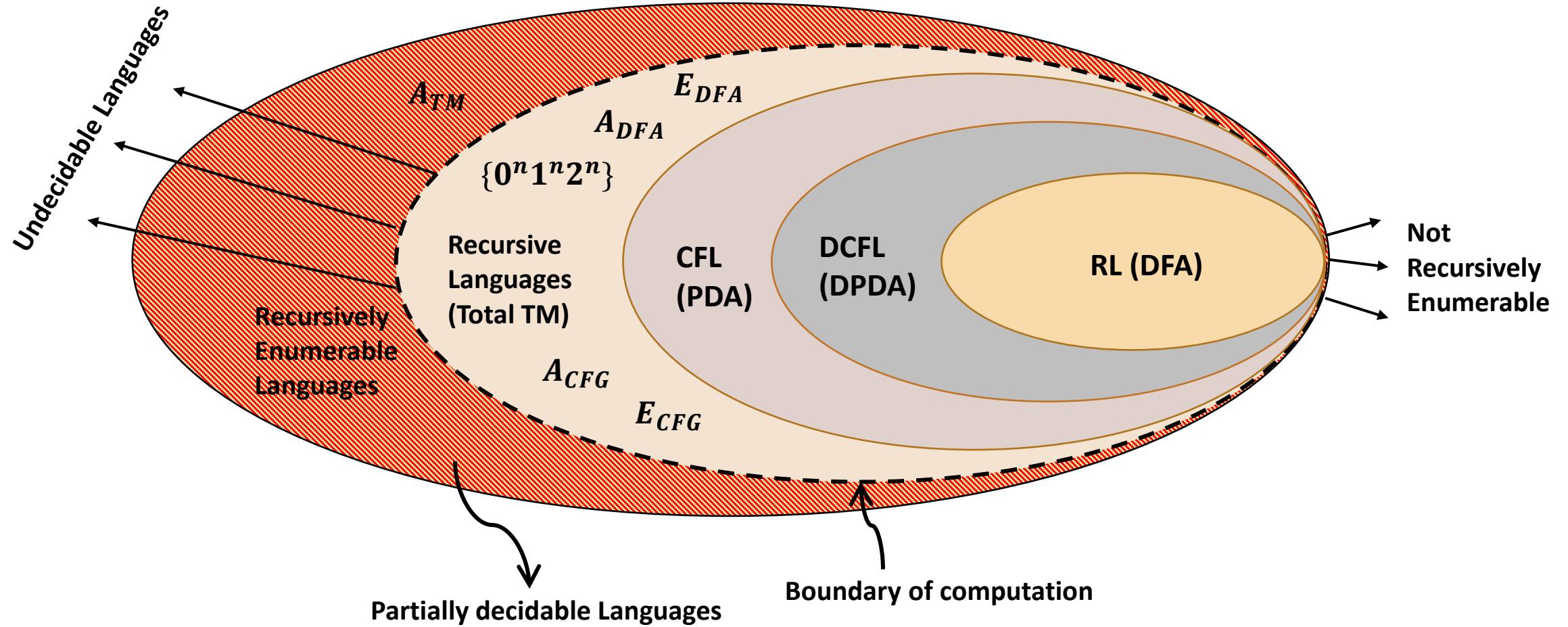
- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

Note that  $D$  by definition **computes the opposite of the diagonal entries** of the table.

<b><math>A</math></b>	$\langle M_0 \rangle$	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...	$\langle D \rangle$	...
$M_0$	Accept	Accept	Reject	Reject	Accept	...	Accept	...
$M_1$	Accept	Reject	Reject	Accept	Reject	...	Accept	...
$M_2$	Reject	Reject	Accept	Reject	Accept	...	Accept	...
$M_3$	Accept	Reject	Reject	Accept	Reject	...	Reject	...
$M_4$	Accept	Accept	Accept	Accept	Reject	...	Reject	...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...
<b><math>D</math></b>	Reject	Accept	Reject	Reject	Accept	...	??	...
⋮	⋮	⋮	⋮	⋮	⋮	...	⋮	...

What will be the  $D^{th}$  diagonal entry??

Contradiction!



### Next Lecture:

- Halting Problem
- More on Recursive & RE languages
- Completely undecidable languages
- Wrapping up

**Thank You!**

# CS 302.1 - Automata Theory

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

There exists a one-one mapping (bijective relationship) between the set of binary strings and TMs.

**Universal Turing Machine:** A Universal Turing Machine, denoted as  $U_{TM}$  accepts as input (i) the encoding of a Turing Machine  $M$  and (ii) an input string  $w$  and **simulates  $M$  running on  $w$** , i.e.

$$U_{TM}(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects} \\ \text{LOOPS INFINITELY, if } M(w) \text{ loops infinitely} \end{cases}$$

# Quick Recap

There exists a one-one mapping (bijective relationship) between the set of binary strings and TMs.

**Universal Turing Machine:** A Universal Turing Machine, denoted as  $U_{TM}$  accepts as input (i) the encoding of a Turing Machine  $M$  and (ii) an input string  $w$  and **simulates  $M$  running on  $w$** , i.e.

$$U_{TM}(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ accepts} \\ \text{REJECTS, if } M(w) \text{ rejects} \\ \text{LOOPS INFINITELY, if } M(w) \text{ loops infinitely} \end{cases}$$

Some examples of languages that are recursive/decidable:

- $A_{DFA} = \{\langle DFA \rangle, w | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle | w \in L(CFG)\}$
- $E_{CFG} = \{\langle CFG \rangle | L(CFG) = \Phi\}$

An undecidable language:

- $A_{TM} = \{\langle M, w \rangle | M \text{ accepts input } w\}$

# Quick Recap

There exists a one-one mapping (bijective relationship) between the set of binary strings and TMs.

Some examples of languages that are recursive/decidable:

- $A_{DFA} = \{\langle DFA \rangle, w | w \in L(DFA)\}$
- $E_{DFA} = \{\langle DFA \rangle | L(DFA) = \Phi\}$
- $A_{CFG} = \{\langle CFG, w \rangle | w \in L(CFG)\}$
- $E_{CFG} = \{\langle CFG, w \rangle | L(CFG) = \Phi\}$

An undecidable language:

- $A_{TM} = \{\langle M, w \rangle | M \text{ accepts input } w\}$

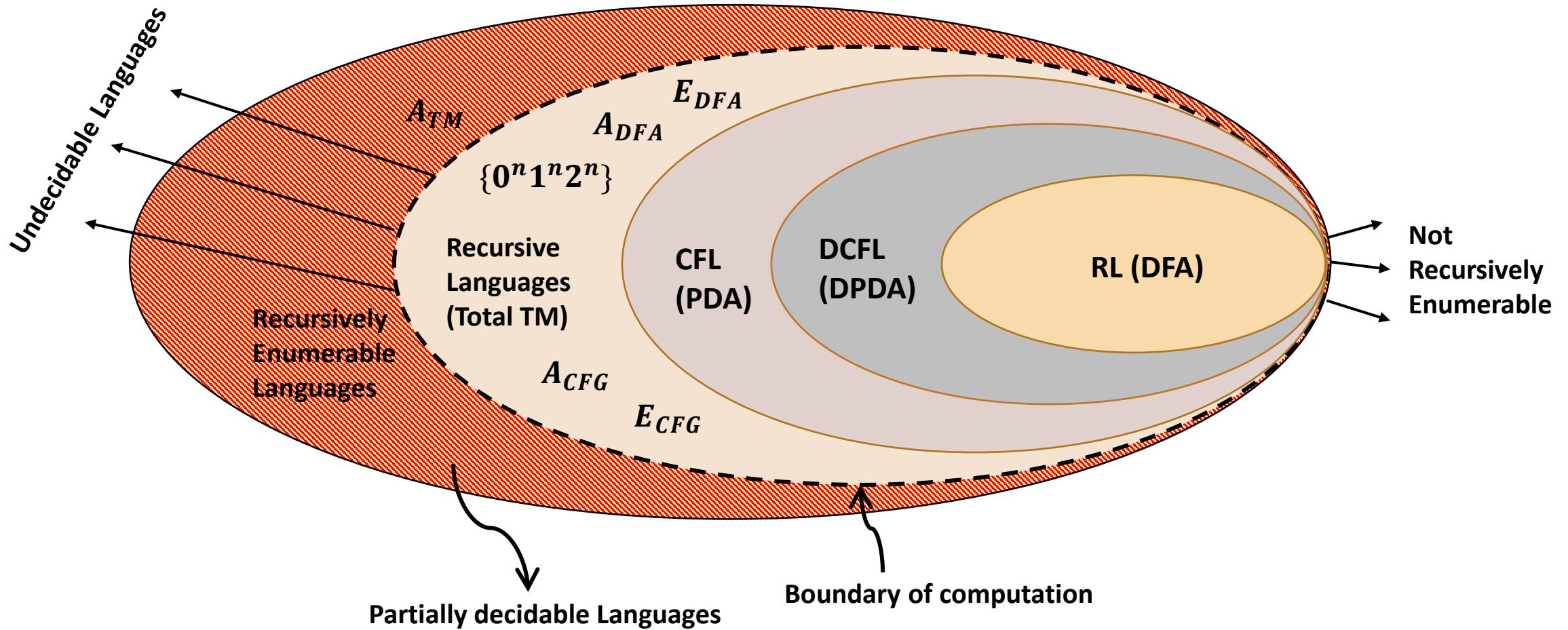
- $A_{TM}$  is undecidable
- $A_{TM} \in RE$  but not recursive
- $A_{TM}$  is partially decidable

**Proof strategy:** By contradiction. We assume that a Total TM  $A$  exists that decides  $A_{TM}$ .

We build a Total TM  $D$  that accepts  $w$  as input and calls  $A(\langle w, w \rangle)$  as a subroutine and outputs the opposite of  $A$ .

The TM  $D$  cannot decide the input  $w = \langle D \rangle$ . So  $A$  is undecidable.

# Quick Recap



# The Halting problem

$\text{HALT}_{TM} = \{\langle M, w \rangle | M \text{ halts on input } w\}$ . Is  $\text{HALT}_{TM}$  decidable?

**The Halting Problem:** Does there exist a Total Turing Machine  $H$  that accepts as input a Turing Machine  $M$  and an input string  $w$  and outputs YES, if  $M(w)$  halts (accepts or rejects) and NO, if  $M(w)$  does not halt (loops forever), i.e.

$$H(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ HALTS, i.e. accepts or rejects} \\ \text{REJECTS, if } M(w) \text{ does not HALT, i.e. loops infinitely} \end{cases}$$

# The Halting problem

$\text{HALT}_{TM} = \{\langle M, w \rangle | M \text{ halts on input } w\}$ . Is  $\text{HALT}_{TM}$  decidable?

**The Halting Problem:** Does there exist a Total Turing Machine  $H$  that accepts as input a Turing Machine  $M$  and an input string  $w$  and outputs YES, if  $M(w)$  halts (accepts or rejects) and NO, if  $M(w)$  does not halt (loops forever), i.e.

$$H(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ HALTS, i.e. accepts or rejects} \\ \text{REJECTS, if } M(w) \text{ does not HALT, i.e. loops infinitely} \end{cases}$$

- Turing stated the Halting problem and demonstrated its undecidability in his famous 1936 paper.
- This provided a negative answer to Hilbert's Entscheidungsproblem.
- **Proof strategy:** We will try to show that if we had such a Total TM  $H$ , we would be able to build a Total TM for  $A_{TM}$
- But since we proved that  $A_{TM}$  is undecidable, this would mean that  $H$  is undecidable.

# The Halting problem

$HALT_{TM} = \{\langle M, w \rangle | M \text{ halts on input } w\}$ . Is  $HALT_{TM}$  decidable?



**Proof idea:** We first assume that there exists such a Total Turing Machine  $H$ . Then, we use  $H$  as a subroutine to construct a Total Turing Machine  $A$  for  $A_{TM}$



But  $A$  cannot be Total and so a total TM that decides  $H$  cannot exist

# The Halting problem

$HALT_{TM} = \{\langle M, w \rangle | M \text{ halts on input } w\}$ . Is  $HALT_{TM}$  decidable?



**Proof idea:** We first assume that there exists such a Total Turing Machine  $H$ . Then, we use  $H$  as a subroutine to construct a Total Turing Machine  $A$  for  $A_{TM}$ .

**Outlining the steps for building  $A$  using  $H$ :**

- $A$  calls  $H(\langle M, w \rangle)$
- If  $H$  rejects, then we know that  $M(w)$  loops forever and so  $A$  would output **REJECT**.



# The Halting problem

$HALT_{TM} = \{\langle M, w \rangle | M \text{ halts on input } w\}$ . Is  $HALT_{TM}$  decidable?



**Proof idea:** We first assume that there exists such a Total Turing Machine  $H$ . Then, we use  $H$  as a subroutine to construct a Total Turing Machine  $A$  for  $A_{TM}$ .

**Outlining the steps for building  $A$  using  $H$ :**

- $A$  calls  $H(\langle M, w \rangle)$
- If  $H$  rejects, then we know that  $M(w)$  loops forever and so  $A$  would output **REJECT**.
- If  $H$  accepts,
  - $M(w)$  surely halts (either accepts or rejects).



# The Halting problem

$HALT_{TM} = \{\langle M, w \rangle | M \text{ halts on input } w\}$ . Is  $HALT_{TM}$  decidable?



**Proof idea:** We first assume that there exists such a Total Turing Machine  $H$ . Then, we use  $H$  as a subroutine to construct a Total Turing Machine  $A$  for  $A_{TM}$ .

**Outlining the steps for building  $A$  using  $H$ :**

- $A$  calls  $H(\langle M, w \rangle)$
- If  $H$  rejects, then we know that  $M(w)$  loops forever and so  $A$  would output **REJECT**.
- If  $H$  accepts,
  - $M(w)$  surely halts (either accepts or rejects).
  - Simply run  $M(w)$  and
    - **ACCEPT** if  $M(w)$  accepts
    - **REJECT** if  $M(w)$  rejects



# The Halting problem

$\text{HALT}_{TM} = \{\langle M, w \rangle | M \text{ halts on input } w\}$ . Is  $\text{HALT}_{TM}$  decidable?

$$H(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ HALTS, i.e. accepts or rejects} \\ \text{REJECTS, if } M(w) \text{ does not HALT, i.e. loops infinitely} \end{cases}$$

**Proof:** Assume that there exists such a Total Turing Machine  $H$ . Then, we use  $H$  as a subroutine to construct a Total Turing Machine  $A$  for  $A_{TM}$  as follows:

$A$  = On input  $\langle M, w \rangle$   
    Run  $H(\langle M, w \rangle)$   
    If  $H$  rejects, output *REJECT*  
    If  $H$  accepts,  
        Run  $M(w)$   
        If  $M(w)$  accepts, output *ACCEPT*  
        If  $M(w)$  rejects, output *REJECT*

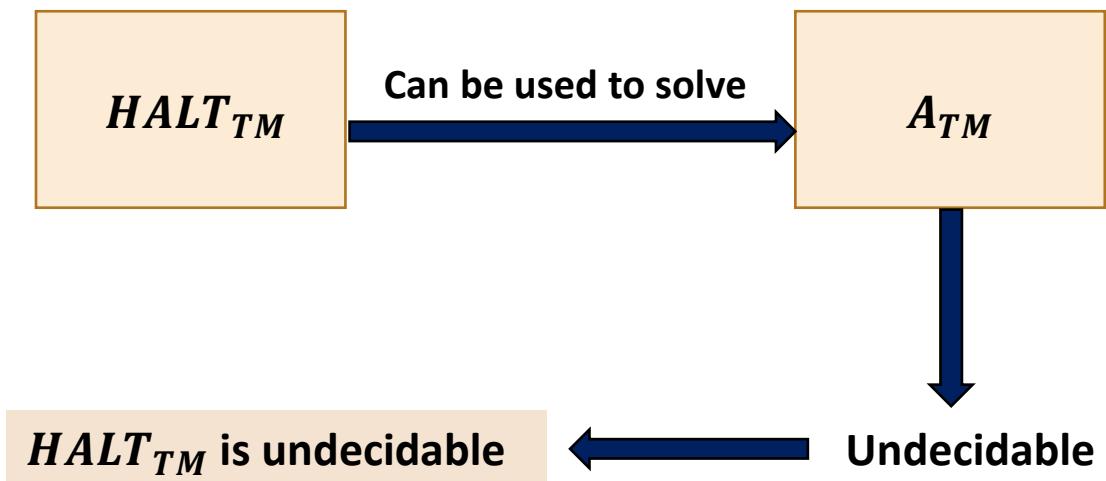
# The Halting problem

$HALT_{TM} = \{\langle M, w \rangle | M \text{ halts on input } w\}$ . Is  $HALT_{TM}$  decidable?

$$H(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ HALTS, i.e. accepts or rejects} \\ \text{REJECTS, if } M(w) \text{ does not HALT, i.e. loops infinitely} \end{cases}$$

**Proof:** Assume that there exists such a Total Turing Machine  $H$ . Then, we use  $H$  as a subroutine to construct a Total Turing Machine  $A$  for  $A_{TM}$  as follows:

$A$  = On input  $\langle M, w \rangle$   
Run  $H(\langle M, w \rangle)$   
If  $H$  rejects, output *REJECT*  
If  $H$  accepts,  
    Run  $M(w)$   
    If  $M(w)$  accepts, output *ACCEPT*  
    If  $M(w)$  rejects, output *REJECT*



# The Halting problem

$\text{HALT}_{TM} = \{\langle M, w \rangle | M \text{ halts on input } w\}$ . Is  $\text{HALT}_{TM}$  decidable?

$$H(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ HALTS, i.e. accepts or rejects} \\ \text{REJECTS, if } M(w) \text{ does not HALT, i.e. loops infinitely} \end{cases}$$

$\text{HALT}_{TM} \in RE$  as  $H$  halts whenever  $M$  accepts or rejects  $w$  and so

$Q =$  On input  $\langle M, w \rangle$ :

- Simulate  $M$  on input  $w$
- If  $M$  accepts  $w$ , ACCEPT; if  $M$  rejects  $w$ , REJECT

$Q$  recognizes  $\text{HALT}_{TM}$

# The Halting problem

$\text{HALT}_{TM} = \{\langle M, w \rangle \mid M \text{ halts on input } w\}$ . Is  $\text{HALT}_{TM}$  decidable?

$$H(\langle M, w \rangle) = \begin{cases} \text{ACCEPTS, if } M(w) \text{ HALTS, i.e. accepts or rejects} \\ \text{REJECTS, if } M(w) \text{ does not HALT, i.e. loops infinitely} \end{cases}$$

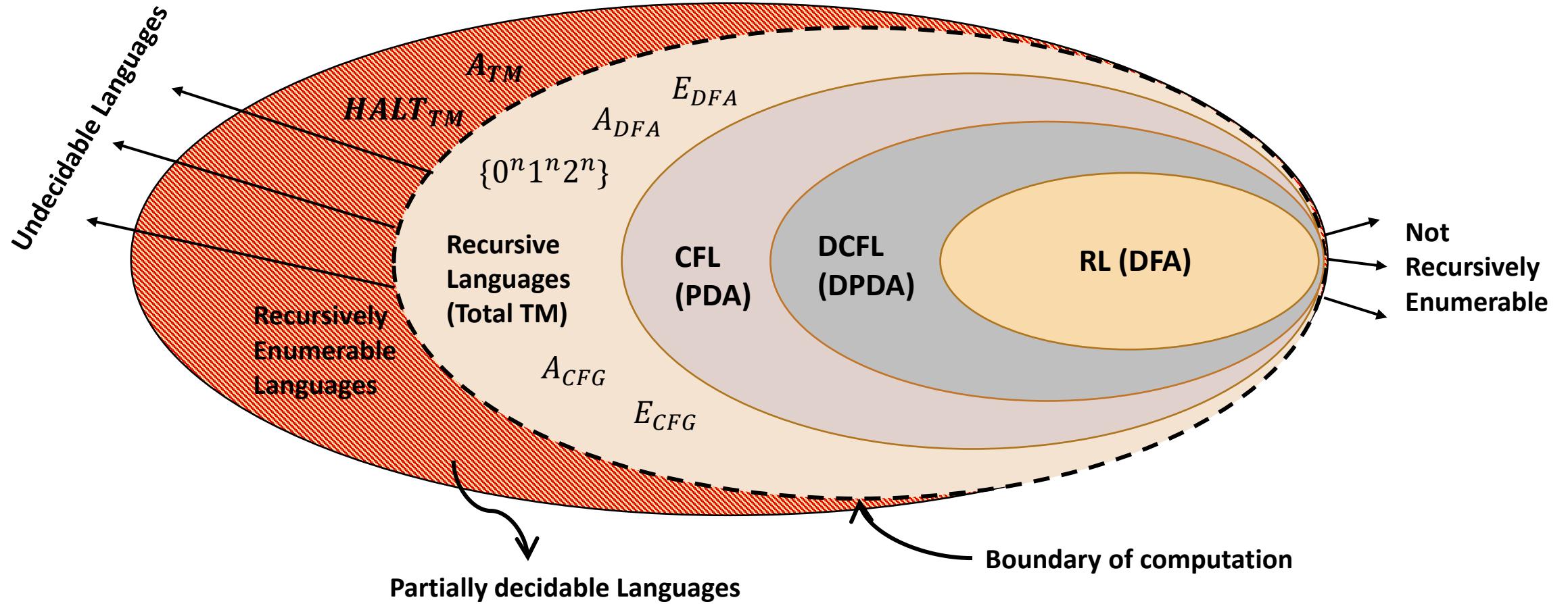
$\text{HALT}_{TM} \in RE$  as  $H$  halts whenever  $M$  accepts or rejects  $w$  and so

$Q$  = On input  $\langle M, w \rangle$ :

- Simulate  $M$  on input  $w$
- If  $M$  accepts  $w$ , ACCEPT; if  $M$  rejects  $w$ , REJECT

$Q$  recognizes  $\text{HALT}_{TM}$

- $\text{HALT}_{TM}$  is undecidable
- $\text{HALT}_{TM} \in RE$  but not recursive
- $\text{HALT}_{TM}$  is partially decidable

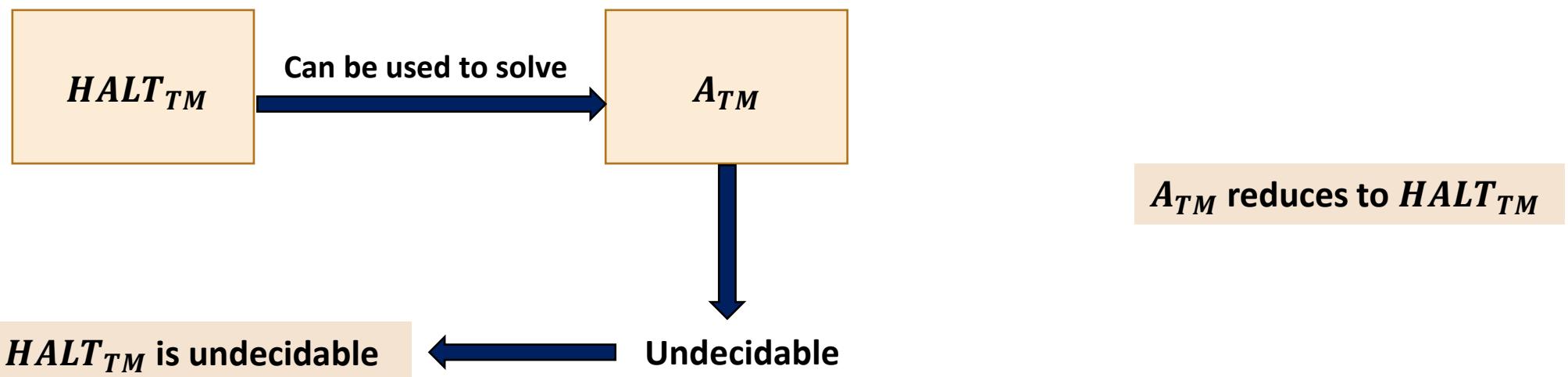


# Reduction

Recall the proof of the undecidability of the Halting Problem

**What did we do there?**

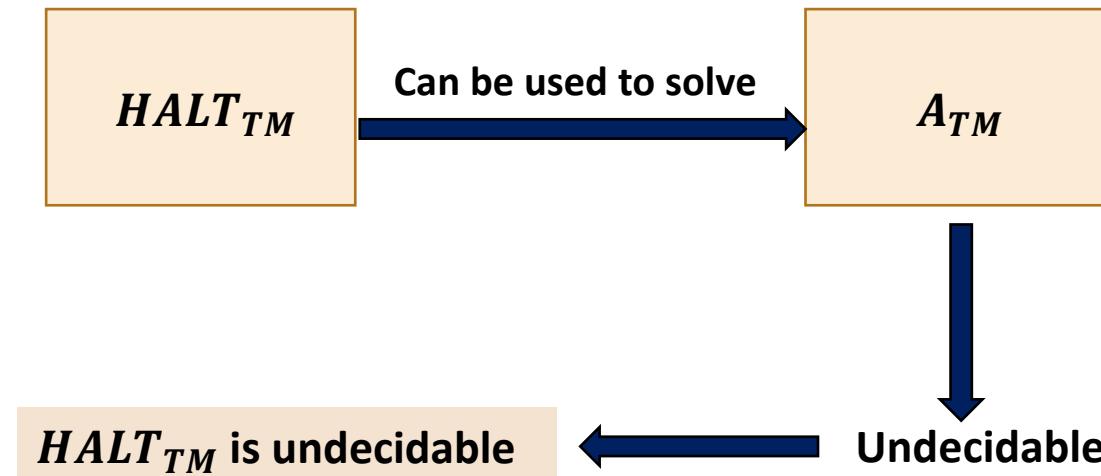
- We used a (supposed) decider for the Halting Problem to build a decider for  $A_{TM}$ .
- This established that  $\text{HALT}_{TM}$  can be used to solve  $A_{TM}$ .
- As  $A_{TM}$  is undecidable, this established that  $\text{HALT}_{TM}$  is undecidable too.
- The key underlying concept is an idea called **Reduction**.



# Reduction

Generally,

- A language  $A$  **reduces to** another language  $B$  ( $A \leq B$ ) iff we can build a **solver** for  $A$  using a **solver** for  $B$
- In terms of computability, suppose using  $B$  we can compute  $A$ . Then, if  $A$  is **undecidable** then so is  $B$ .
- So, in the last proof we showed:  $A_{TM} \leq \text{HALT}_{TM}$  to prove that  $\text{HALT}_{TM}$  is undecidable.



# Reduction

Generally,

- A language  $A$  reduces to another language  $B$  ( $A \leq B$ ) iff we can build a solver for  $A$  using solver for  $B$ .
- In terms of computability, suppose using  $B$  we can compute  $A$ . Then, if  $A$  is undecidable then so is  $B$ .
- So, in the last proof we showed:  $A_{TM} \leq \text{HALT}_{TM}$  to prove that  $\text{HALT}_{TM}$  is undecidable.
- This is a common technique to show that certain problems are decidable/undecidable.

Suppose,  $A \leq B$  and

- $B$  is decidable. Then  $A$  is decidable.
- $A$  is undecidable. Then  $B$  is undecidable.
- If  $B \in RE$ , then  $A \in RE$ .

For example, we can prove that a problem  $P$  is undecidable by reducing the Halting problem to  $P$ .

# Reduction

Generally,

- A language  $A$  reduces to another language  $B$  ( $A \leq B$ ) iff we can build a solver for  $A$  using solver for  $B$ .
- In terms of computability, suppose using  $B$  we can compute  $A$ . Then, if  $A$  is undecidable then so is  $B$ .
- So, in the last proof we showed:  $A_{TM} \leq \text{HALT}_{TM}$  to prove that  $\text{HALT}_{TM}$  is undecidable.
- This is a common technique to show that certain problems are decidable/undecidable.

We can prove that several problems are undecidable by the reducing  $\text{HALT}_{TM}$  to these problems.

## Examples:

- $E_{TM} = \{\langle M \rangle | M \text{ is a Turing Machine and } L(M) = \Phi\}$
  - $EQ_{TM} = \{\langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are Turing Machines having } L(M_1) = L(M_2)\}$
  - $ALL_{TM} = \{\langle M \rangle | M \text{ halts on all inputs}\}$
- ⋮

For any such problem (say  $P$ ),

$$\text{HALT}_{TM} \leq P$$

# Reduction

We can prove that several problems are undecidable by the **reduction of these problems** to the Halting Problem.

## Examples:

- $E_{TM} = \{\langle M \rangle | M \text{ is a Turing Machine and } L(M) = \Phi\}$
- $EQ_{TM} = \{\langle M_1, M_2 \rangle | M_1 \text{ and } M_2 \text{ are Turing Machines having } L(M_1) = L(M_2)\}$
- $ALL_{TM} = \{\langle M \rangle | M \text{ halts on all inputs}\}$   
⋮

For any of these problems  $P$ ,

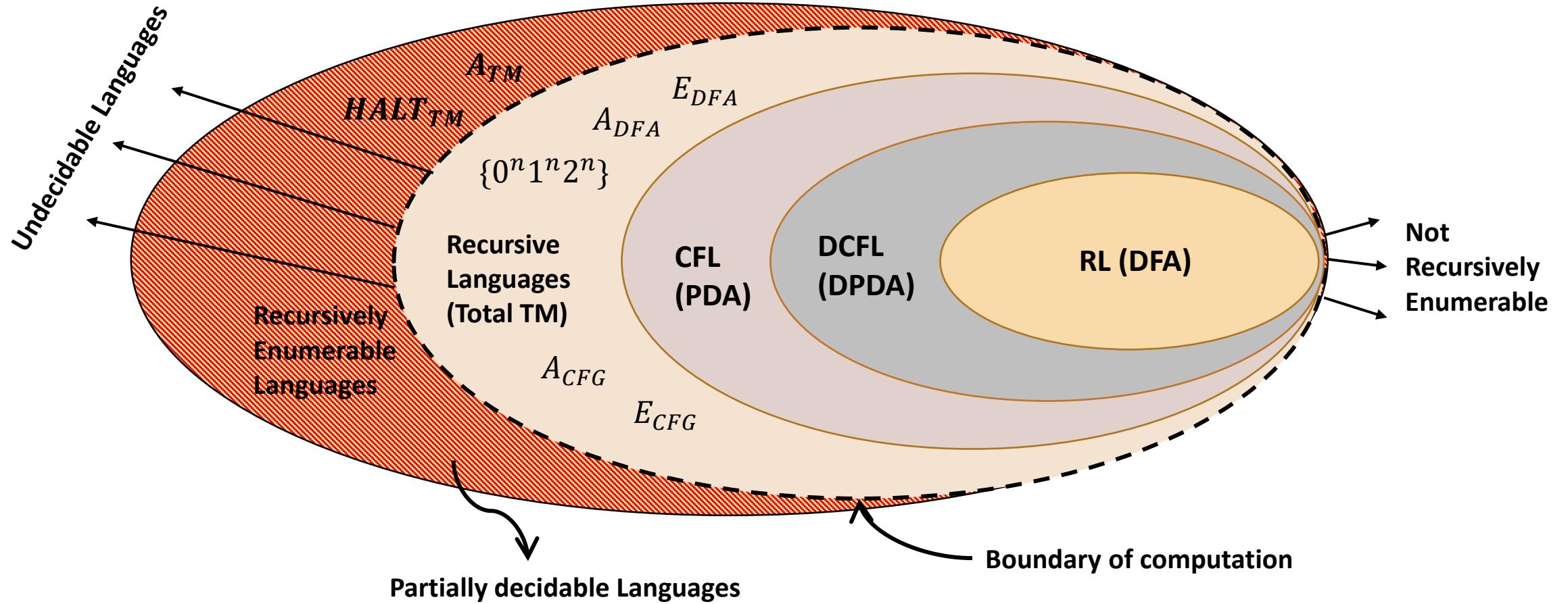
$$HALT_{TM} \leq P$$

## Generic strategy for proof:

- Consider that a Total TM for the given problem exists (say  $T_M$ ).
- Show that can build a total TM  $H$  that can decide  $HALT_{TM}$  using  $T_M$  as a subroutine.

- $HALT_{TM} \leq E_{TM}$
- $HALT_{TM} \leq EQ_{TM}$
- $HALT_{TM} \leq ALL_{TM}$

This reduction would prove that  $E_{TM}$ ,  $EQ_{TM}$ ,  $ALL_{TM}$  are undecidable.



# Closure properties

Are Recursive languages **closed under Union**? If  $R_1$  and  $R_2$  are recursive, is  $R_1 \cup R_2$  recursive?

**Proof:**

- Let  $M_1$  and  $M_2$  be the Total Turing Machines corresponding to  $R_1$  and  $R_2$  respectively.
- Using  $M$ , we construct a Total Turing Machine  $M'$  such that  $M'$  decides  $R_1 \cup R_2$ .

# Closure properties

Are Recursive languages **closed under Union**? If  $R_1$  and  $R_2$  are recursive, is  $R_1 \cup R_2$  recursive?

**Proof:**

- Let  $M_1$  and  $M_2$  be the Total Turing Machines corresponding to  $R_1$  and  $R_2$  respectively.
- Using  $M$ , we construct a Total Turing Machine  $M'$  such that  $M'$  decides  $R_1 \cup R_2$ .

$M'$  = On input  $w$

Run  $M_1(w)$

Run  $M_2(w)$

If either of them accept, *ACCEPT*

If both rejects, *REJECT*

Recursive languages are **CLOSED under Union**.

# Closure properties

Are Recursive languages **closed under intersection**? If  $R_1$  and  $R_2$  are recursive, is  $R_1 \cap R_2$  recursive?

**Proof:**

- Let  $M_1$  and  $M_2$  be the Total Turing Machines corresponding to  $R_1$  and  $R_2$  respectively.
- Using  $M$ , we construct a Total Turing Machine  $M'$  such that  $M'$  decides  $R_1 \cap R_2$ .

$M' =$  On input  $w$   
Run  $M_1(w)$   
Run  $M_2(w)$   
If both of them accept, *ACCEPT*  
If either rejects, *REJECT*

Recursive languages are **CLOSED under Intersection**.

# Closure properties

Are Recursive languages **closed under complementation**? If some language  $R$  is recursive, is  $\bar{R}$  also recursive?

**Proof:**

- If  $R$  is recursive then there exists a total TM  $M$  for  $R$ .
- Using  $M$ , we construct a Total Turing Machine  $\bar{M}$  such that  $\bar{M}$  decides  $\bar{R}$ .

$\bar{M}$  = On input  $w$

Run  $M(w)$ .

If  $M$  accepts, *REJECT*

If  $M$  rejects, *ACCEPT*

Recursive languages are **CLOSED under complementation**

# Closure properties

**$L$  and  $\bar{L}$  are both Recursively Enumerable if and only if  $L$  is Recursive.**

# Closure properties

**$L$  and  $\bar{L}$  are both Recursively Enumerable if and only if  $L$  is Recursive.**

**Proof:** In one direction, the proof is trivial. That is, if  $L$  is Recursive then so is  $\bar{L}$ . As Recursive Languages  $R \subseteq RE$ , we have that for  $L$  and  $\bar{L}$  are in  $RE$ .

# Closure properties

**$L$  and  $\bar{L}$  are both Recursively Enumerable if and only if  $L$  is Recursive.**

**Proof:** In one direction, the proof is trivial. That is, if  $L$  is Recursive then so is  $\bar{L}$ . As Recursive Languages  $R \subseteq RE$ , we have that for  $L$  and  $\bar{L}$  are in  $RE$ .

For the other direction: Let  $M_1$  be the  $TM$  for  $L$  and  $M_2$  be the TM for  $\bar{L}$ . Then, if  $w \in L$ ,  $M_1(w)$  accepts and if  $w \notin L$ ,  $M_2(w)$  accepts.

# Closure properties

**$L$  and  $\bar{L}$  are both Recursively Enumerable if and only if  $L$  is Recursive.**

**Proof:** In one direction, the proof is trivial. That is, if  $L$  is Recursive then so is  $\bar{L}$ . As Recursive Languages  $R \subseteq RE$ , we have that for  $L$  and  $\bar{L}$  are in  $RE$ .

For the other direction: Let  $M_1$  be the *TM* for  $L$  and  $M_2$  be the *TM* for  $\bar{L}$ . Then, if  $w \in L$ ,  $M_1(w)$  accepts and if  $w \notin L$ ,  $M_2(w)$  accepts.

How do we build a Total Turing Machine for  $L$ ? There is one problem: We can't run  $M_1$  and  $M_2$  one after the other as for some input  $M_1$  gets stuck in an infinite loop and  $M_2$  never gets control.

# Closure properties

**$L$  and  $\bar{L}$  are both Recursively Enumerable if and only if  $L$  is Recursive.**

**Proof:** In one direction, the proof is trivial. That is, if  $L$  is Recursive then so is  $\bar{L}$ . As Recursive Languages  $R \subseteq RE$ , we have that for  $L$  and  $\bar{L}$  are in  $RE$ .

For the other direction: Let  $M_1$  be the *TM* for  $L$  and  $M_2$  be the *TM* for  $\bar{L}$ . Then, if  $w \in L$ ,  $M_1(w)$  accepts and if  $w \notin L$ ,  $M_2(w)$  accepts.

How do we build a Total Turing Machine for  $L$ ? There is one problem: We can't run  $M_1$  and  $M_2$  one after the other as for some input  $M_1$  gets stuck in an infinite loop and  $M_2$  never gets control.

**Idea:** We use a time sharing technique – also known as **Dovetailing** to build a Total *TM*  $M'$  for  $L$ .

# Closure properties

**$L$  and  $\bar{L}$  are both Recursively Enumerable if and only if  $L$  is Recursive.**

**Proof:** In one direction, the proof is trivial. That is, if  $L$  is Recursive then so is  $\bar{L}$ . As Recursive Languages  $R \subseteq RE$ , we have that for  $L$  and  $\bar{L}$  are in  $RE$ .

For the other direction: Let  $M_1$  be the *TM* for  $L$  and  $M_2$  be the *TM* for  $\bar{L}$ . Then, if  $w \in L$ ,  $M_1(w)$  accepts and if  $w \notin L$ ,  $M_2(w)$  accepts.

How do we build a Total Turing Machine for  $L$ ? There is one problem: We can't run  $M_1$  and  $M_2$  one after the other as for some input  $M_1$  gets stuck in an infinite loop and  $M_2$  never gets control.

**Idea:** We use a time sharing technique – also known as **Dovetailing** to build a Total *TM*  $M'$  for  $L$ .

$M' =$

For  $i = 1, 2, \dots$

Run  $M_1(w)$  for  $i$  steps.

Run  $M_2(w)$  for  $i$  steps.

If  $M_1$  accepts,  $M'$  outputs *ACCEPT*.

If  $M_2$  accepts,  $M'$  outputs *REJECT*.

# Closure properties

**$L$  and  $\bar{L}$  are both Recursively Enumerable if and only if  $L$  is Recursive.**

**Proof:** In one direction, the proof is trivial. That is, if  $L$  is Recursive then so is  $\bar{L}$ . As Recursive Languages  $R \subseteq RE$ , we have that for  $L$  and  $\bar{L}$  are in  $RE$ .

For the other direction: Let  $M_1$  be the *TM* for  $L$  and  $M_2$  be the *TM* for  $\bar{L}$ . Then, if  $w \in L$ ,  $M_1(w)$  accepts and if  $w \notin L$ ,  $M_2(w)$  accepts.

How do we build a Total Turing Machine for  $L$ ? There is one problem: We can't run  $M_1$  and  $M_2$  one after the other as for some input  $M_1$  gets stuck in an infinite loop and  $M_2$  never gets control.

**Idea:** We use a time sharing technique – also known as **Dovetailing** to build a Total *TM*  $M'$  for  $L$ .

$M' =$

For  $i = 1, 2, \dots$

Run  $M_1(w)$  for  $i$  steps.

Run  $M_2(w)$  for  $i$  steps.

If  $M_1$  accepts,  $M'$  outputs *ACCEPT*.

If  $M_2$  accepts,  $M'$  outputs *REJECT*.

**$L$  is Recursive**

# Closure properties

Using Dovetailing it is easy to prove that:

- RE languages are closed under **union and intersection**
  - On input  $w$ , run  $M_1(w)$  and  $M_2(w)$  in parallel using dovetailing
  - **For union:** If either  $M_1(w)$  or  $M_2(w)$  accepts, *ACCEPT*
  - **For intersection:** If both  $M_1(w)$  and  $M_2(w)$  accept, *ACCEPT*

# Closure properties

Using Dovetailing it is easy to prove that:

- RE languages are closed under **union and intersection**
  - On input  $w$ , run  $M_1(w)$  and  $M_2(w)$  in parallel using dovetailing
  - **For union:** If either  $M_1(w)$  or  $M_2(w)$  accepts, *ACCEPT*
  - **For intersection:** If both  $M_1(w)$  and  $M_2(w)$  accept, *ACCEPT*

RE languages are **NOT closed** under complementation. Why?

# Closure properties

Using Dovetailing it is easy to prove that:

- RE languages are **closed under union and intersection**
  - On input  $w$ , run  $M_1(w)$  and  $M_2(w)$  in parallel using dovetailing
  - **For union:** If either  $M_1(w)$  or  $M_2(w)$  accepts, *ACCEPT*
  - **For intersection:** If both  $M_1(w)$  and  $M_2(w)$  accept, *ACCEPT*

RE languages are **NOT closed** under complementation. Why?

- Well, we just proved that  $L$  and  $\bar{L}$  are both Recursively Enumerable, **iff**  $L$  is Recursive.
- But we know that there exists problems that are in  $RE$  but are not Recursive (e.g:  $A_{TM}, HALT_{TM}, \dots$ ).
- So the complement of such problems are not in  $RE$  (e.g.:  $\overline{A_{TM}}, \overline{HALT}_{TM}, \dots$ ).

# Closure properties

RE languages are **NOT closed** under complementation. Why?

- Well, we just proved that  $L$  and  $\bar{L}$  are both Recursively Enumerable, iff  $L$  is Recursive.
- But we know that there exists problems that are in  $RE$  but not Recursive (e.g.:  $A_{TM}$ ,  $HALT_{TM}, \dots$ ).
- So the complement of such problems are not in  $RE$  (e.g.:  $\overline{HALT}_{TM}, \dots$ ).

Suppose  $L \in RE$  and  $M$  be the  $TM$  which recognizes  $L$ , i.e.  $\mathcal{L}(M) = L$ .

What if we try to build a TM  $\bar{M}$  that outputs the opposite of  $M$ ?

# Closure properties

RE languages are **NOT closed** under complementation. Why?

- Well, we just proved that  $L$  and  $\bar{L}$  are both Recursively Enumerable, iff  $L$  is Recursive.
- But we know that there exists problems that are in  $RE$  but not Recursive (e.g.:  $A_{TM}$ ,  $HALT_{TM}, \dots$ ).
- So the complement of such problems are not in  $RE$  (e.g.:  $\overline{HALT}_{TM}, \dots$ ).

Suppose  $L \in RE$  and  $M$  be the  $TM$  which recognizes  $L$ , i.e.  $\mathcal{L}(M) = L$ .

What if we try to build a TM  $\bar{M}$  that outputs the opposite of  $M$ ?

$\bar{M}$  = On input  $w$

Run  $M(w)$ .

If  $M(w)$  accepts, output *REJECT*

If  $M(w)$  rejects, output *ACCEPT*

If  $M(w)$  loops, .....

# Closure properties

RE languages are **NOT closed** under complementation. Why?

- Well, we just proved that  $L$  and  $\bar{L}$  are both Recursively Enumerable, iff  $L$  is Recursive.
- But we know that there exists problems that are in  $RE$  but not Recursive (e.g.:  $A_{TM}$ ,  $HALT_{TM}, \dots$ ).
- So the complement of such problems are not in  $RE$  (e.g.:  $\overline{HALT}_{TM}, \dots$ ).

Suppose  $L \in RE$  and  $M$  be the  $TM$  which recognizes  $L$ , i.e.  $\mathcal{L}(M) = L$ .

What if we try to build a TM  $\bar{M}$  that outputs the opposite of  $M$ ?

$\bar{M}$  = On input  $w$

Run  $M(w)$ .

If  $M(w)$  accepts, output *REJECT*

If  $M(w)$  rejects, output *ACCEPT*

If  $M(w)$  loops, .....

So if,

$w \notin L$ , i.e.,  $w \in \bar{L}$ ,  $\bar{M}$  outputs *ACCEPT* or loops forever

$w \in L$ , i.e.,  $w \notin \bar{L}$ ,  $\bar{M}$  outputs *REJECT*

# Closure properties

RE languages are **NOT closed** under complementation. Why?

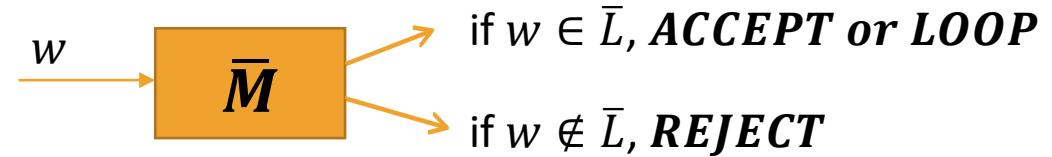
$\bar{M}$  = On input  $w$

Run  $M(w)$ .

If  $M(w)$  accepts, output REJECT

If  $M(w)$  rejects, output ACCEPT

If  $M(w)$  loops, .....



# Closure properties

RE languages are **NOT closed** under complementation. Why?

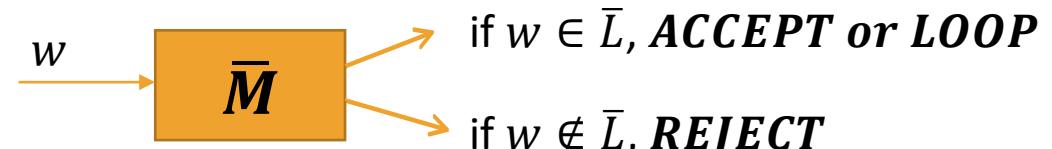
$\bar{M}$  = On input  $w$

Run  $M(w)$ .

If  $M(w)$  accepts, output REJECT

If  $M(w)$  rejects, output ACCEPT

If  $M(w)$  loops, .....



**Co-Recursively Enumerable Language/co-Turing Recognizable (Co-RE/ $\overline{RE}$ /nRE):** A language  $C$  is **Co-Recursively Enumerable (co-RE/ $\overline{RE}$ /nRE)** or **Co-Turing Recognizable** if

$\forall \omega \in C, M(\omega)$  doesn't reject    (accepts or loops forever)

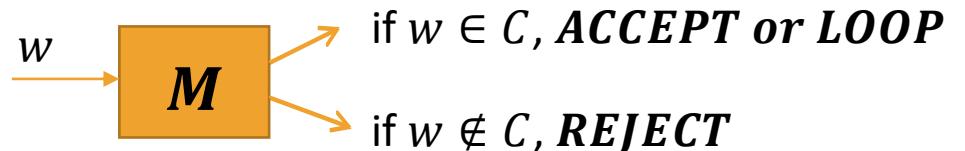
$\forall \omega \notin C, M(\omega)$  rejects

If  $L \in RE$ ,  $\bar{L} \in co\text{-}RE$  and vice versa

# *CO-RE*

**Co-Recursively Enumerable Language (Co-RE/ $\overline{RE}$ /nRE):**  $C \in \text{Co-RE}$  if

$\forall \omega \in C, M(\omega)$  **doesn't reject**  
 $\forall \omega \notin C, M(\omega)$  **rejects**

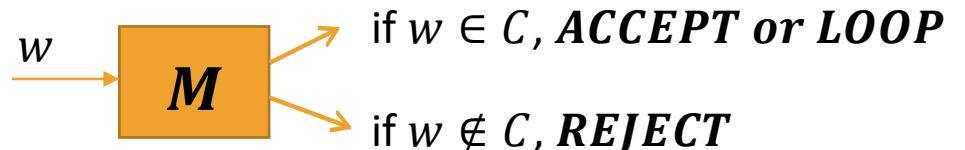


- **RE:** Halts and accepts if  $w \in L$ . May loop when  $w \notin L$ .
- **co-RE:** May loop when  $w \in L$ . Halts and rejects if  $w \notin L$ .

# *CO-RE*

**Co-Recursively Enumerable Language (Co-RE/ $\overline{RE}$ /nRE):**  $C \in \text{Co-RE}$  if

$\forall \omega \in C, M(\omega)$  **doesn't reject**  
 $\forall \omega \notin C, M(\omega)$  **rejects**



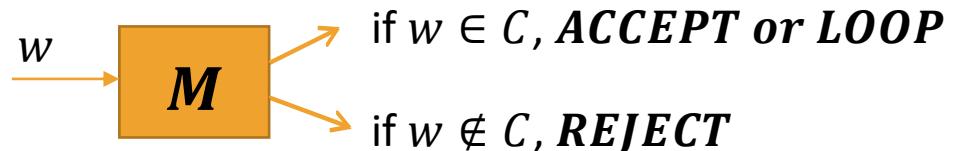
- **RE:** Halts and accepts if  $w \in L$ . May loop when  $w \notin L$ .
- **co-RE:** May loop when  $w \in L$ . Halts and rejects if  $w \notin L$ .

**Note:** Every Recursive Language  $R$ , is both in  $RE$  and  $co\text{-}RE$ , i.e.  $R \subseteq RE \cap co\text{-}RE$

# *CO-RE*

**Co-Recursively Enumerable Language (Co-RE/ $\overline{RE}$ /nRE):**  $C \in \text{Co-RE}$  if

$\forall \omega \in C, M(\omega)$  **doesn't reject**  
 $\forall \omega \notin C, M(\omega)$  **rejects**



- **RE:** Halts and accepts if  $w \in L$ . May loop when  $w \notin L$ .
- **co-RE:** May loop when  $w \in L$ . Halts and rejects if  $w \notin L$ .

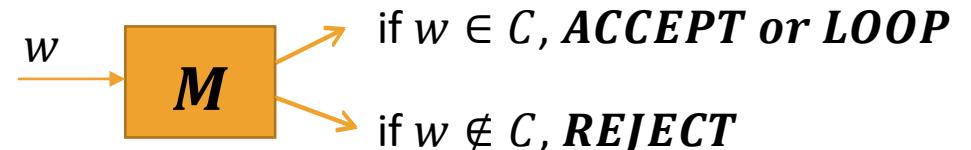
**Note:** Every Recursive Language  $R$ , is both in  $RE$  and  $co\text{-}RE$ , i.e.  $R \subseteq RE \cap co\text{-}RE$

Is  $R = RE \cap co\text{-}RE$ ?

# *CO-RE*

**Co-Recursively Enumerable Language (Co-RE/ $\overline{RE}$ /nRE):**  $C \in \text{Co-RE}$  if

- $\forall \omega \in C, M(\omega)$  **doesn't reject**
- $\forall \omega \notin C, M(\omega)$  **rejects**



- **RE:** Halts and accepts if  $w \in L$ . May loop when  $w \notin L$ .
- **co-RE:** May loop when  $w \in L$ . Halts and rejects if  $w \notin L$ .

**Note: Every Recursive Language  $R$ , is both in  $RE$  and  $co\text{-}RE$ , i.e.  $R \subseteq RE \cap co\text{-}RE$**

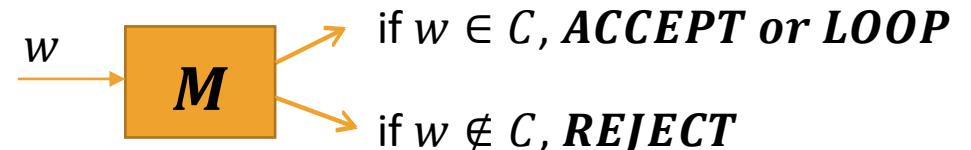
**Is  $R = RE \cap co\text{-}RE$ ?**

We have to prove the following: **If  $L \in RE$  and  $L \in co\text{-}RE$ , then  $L$  is Recursive**

# *CO-RE*

**Co-Recursively Enumerable Language (Co-RE/ $\overline{RE}$ /nRE):**  $C \in \text{Co-RE}$  if

- $\forall \omega \in C, M(\omega) \text{ doesn't reject}$
- $\forall \omega \notin C, M(\omega) \text{ rejects}$



- **RE:** Halts and accepts if  $w \in L$ . May loop when  $w \notin L$ .
- **co-RE:** May loop when  $w \in L$ . Halts and rejects if  $w \notin L$ .

**Note: Every Recursive Language  $R$ , is both in  $RE$  and  $co\text{-}RE$ , i.e.  $R \subseteq RE \cap co\text{-}RE$**

**Is  $R = RE \cap co\text{-}RE$ ?**

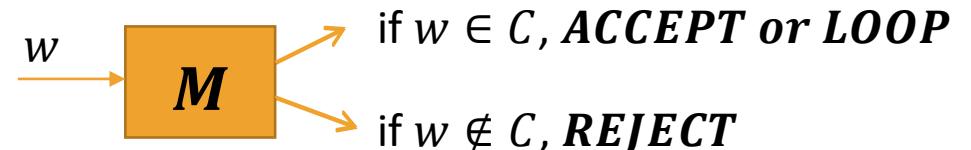
We have to prove the following: **If  $L \in RE$  and  $L \in co\text{-}RE$ , then  $L$  is Recursive**

**Proof:** Let  $M$  be a TM such that  $L(M) = L$ . As  $L \in co\text{-}RE$ , there also exists a  $\bar{M}$  (that outputs the opposite to  $M$ ) that halts and outputs reject whenever  $w \notin L$ .

# *CO-RE*

**Co-Recursively Enumerable Language (Co-RE/ $\overline{RE}$ /nRE):**  $C \in \text{Co-RE}$  if

- $\forall \omega \in C, M(\omega) \text{ doesn't reject}$
- $\forall \omega \notin C, M(\omega) \text{ rejects}$



- **RE:** Halts and accepts if  $w \in L$ . May loop when  $w \notin L$ .
- **co-RE:** May loop when  $w \in L$ . Halts and rejects if  $w \notin L$ .

**Note: Every Recursive Language  $R$ , is both in  $RE$  and  $co\text{-}RE$ , i.e.  $R \subseteq RE \cap co\text{-}RE$**

**Is  $R = RE \cap co\text{-}RE$ ?**

We have to prove the following: **If  $L \in RE$  and  $L \in co\text{-}RE$ , then  $L$  is Recursive**

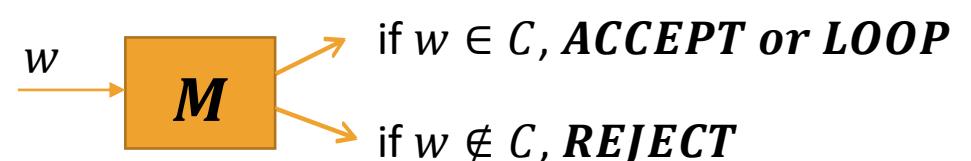
**Proof:** Let  $M$  be a TM such that  $L(M) = L$ . As  $L \in co\text{-}RE$ , there also exists a  $\bar{M}$  (that outputs the opposite to  $M$ ) that halts and outputs reject whenever  $w \notin L$ . We shall construct a Total Turing Machine  $D$  by using  $M$  and  $\bar{M}$ .

We need to run  $M$  and  $\bar{M}$  in parallel using dovetailing.

# *CO-RE*

**Co-Recursively Enumerable Language (Co-RE/ $\overline{RE}$ /nRE):**  $C \in \text{Co-RE}$  if

- $\forall w \in C, M(w)$  **doesn't reject**
- $\forall w \notin C, M(w)$  **rejects**



- **RE:** Halts and accepts if  $w \in L$ . May loop when  $w \notin L$ .
- **co-RE:** May loop when  $w \in L$ . Halts and rejects if  $w \notin L$ .

**Note:** Every Recursive Language  $R$ , is both in **RE** and **co-RE**, i.e.  $R \subseteq RE \cap co-RE$

Is  $R = RE \cap co-RE$ ?

We have to prove the following: If  $L \in RE$  and  $L \in co-RE$ , then  $L$  is Recursive

**Proof:** Let  $M$  be a TM such that  $L(M) = L$ . As  $L \in co-RE$ , there also exists a  $\bar{M}$  (that outputs the opposite to  $M$ ) that halts and outputs reject whenever  $w \notin L$ . We shall construct a Total Turing Machine  $D$  by using  $M$  and  $\bar{M}$ .

$D$  = On input  $w$

Run  $M(w)$  and  $\bar{M}(w)$  in parallel

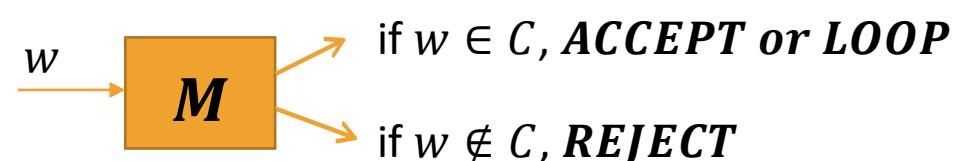
If  $M(w)$  accepts, output **ACCEPT**

If  $\bar{M}(w)$  rejects, output **REJECT**

# *CO-RE*

**Co-Recursively Enumerable Language (Co-RE/ $\overline{RE}$ /nRE):**  $C \in \text{Co-RE}$  if

- $\forall \omega \in C, M(\omega) \text{ doesn't reject}$
- $\forall \omega \notin C, M(\omega) \text{ rejects}$



- **RE:** Halts and accepts if  $w \in L$ . May loop when  $w \notin L$ .
- **co-RE:** May loop when  $w \in L$ . Halts and rejects if  $w \notin L$ .

**Note:** Every Recursive Language  $R$ , is both in **RE** and **co-RE**, i.e.  $R \subseteq RE \cap co-RE$

Is  $R = RE \cap co-RE$ ?

We have to prove the following: If  $L \in RE$  and  $L \in co-RE$ , then  $L$  is Recursive

**Proof:** Let  $M$  be a TM such that  $L(M) = L$ . As  $L \in co-RE$ , there also exists a  $\bar{M}$  (that outputs the opposite of  $M$ ) that halts and outputs reject whenever  $w \notin L$ . We shall construct a Total Turing Machine  $D$  by using  $M$  and  $\bar{M}$ .

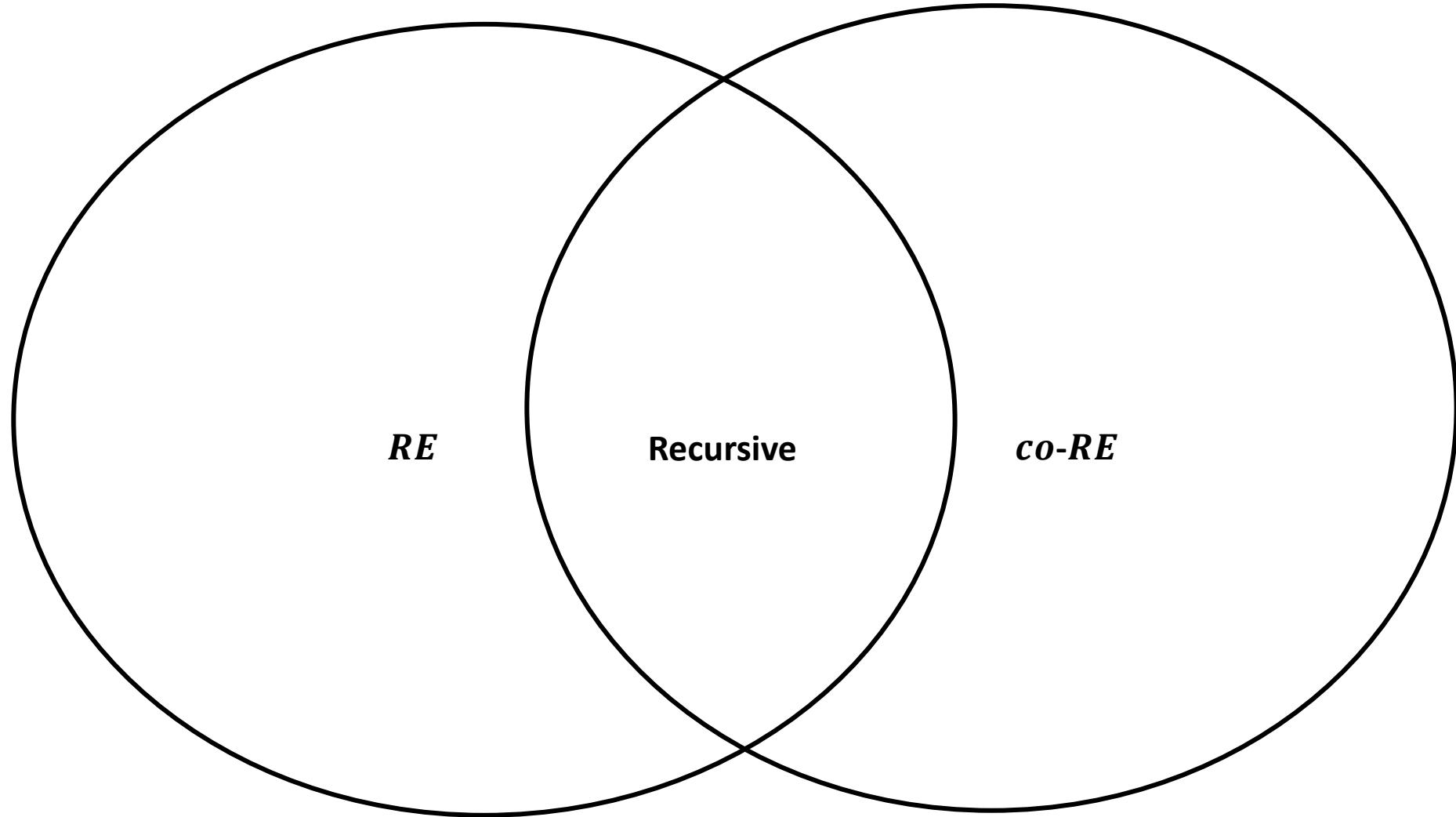
$D$  = On input  $w$

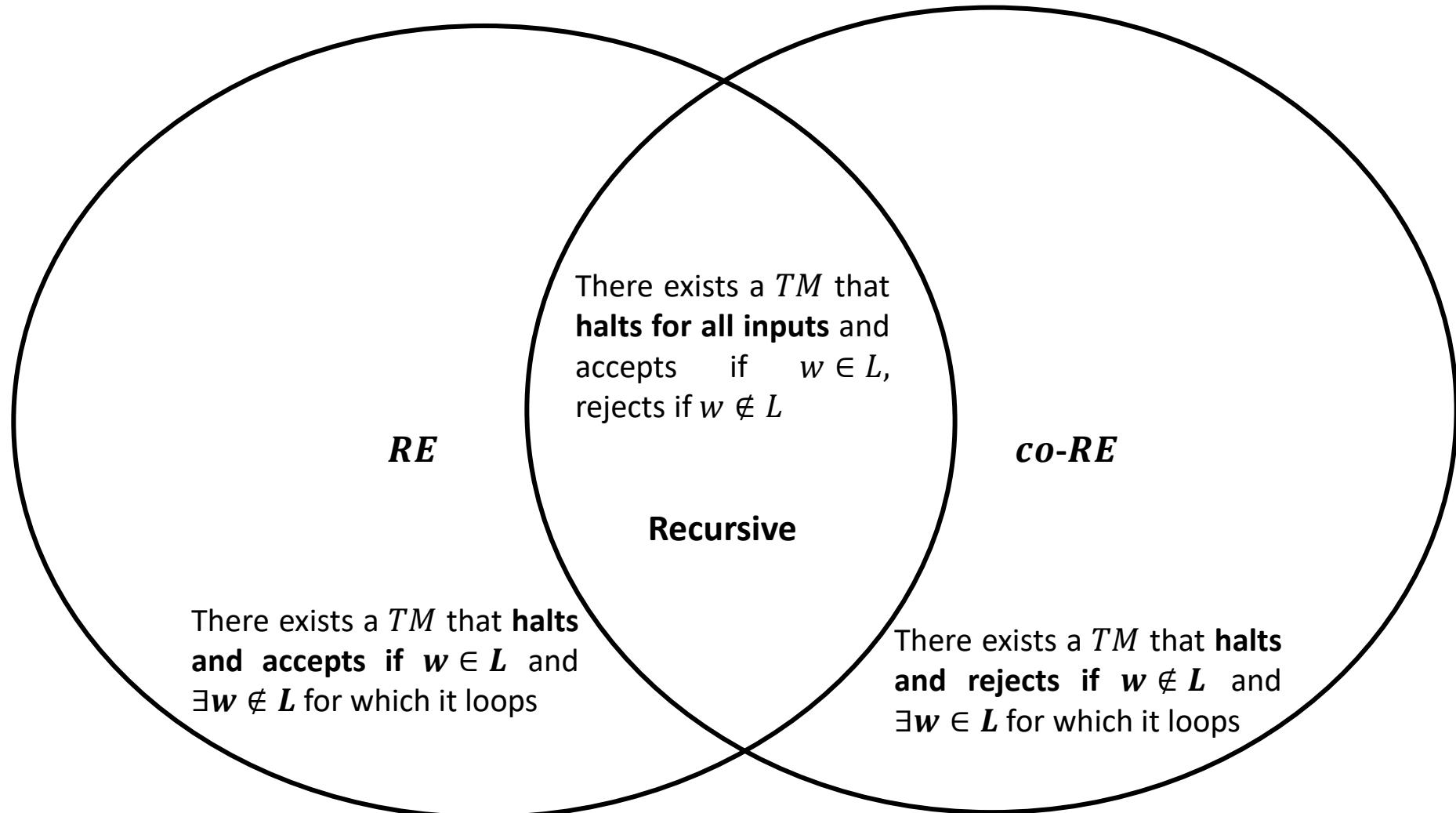
Run  $M(w)$  and  $\bar{M}(w)$  in parallel

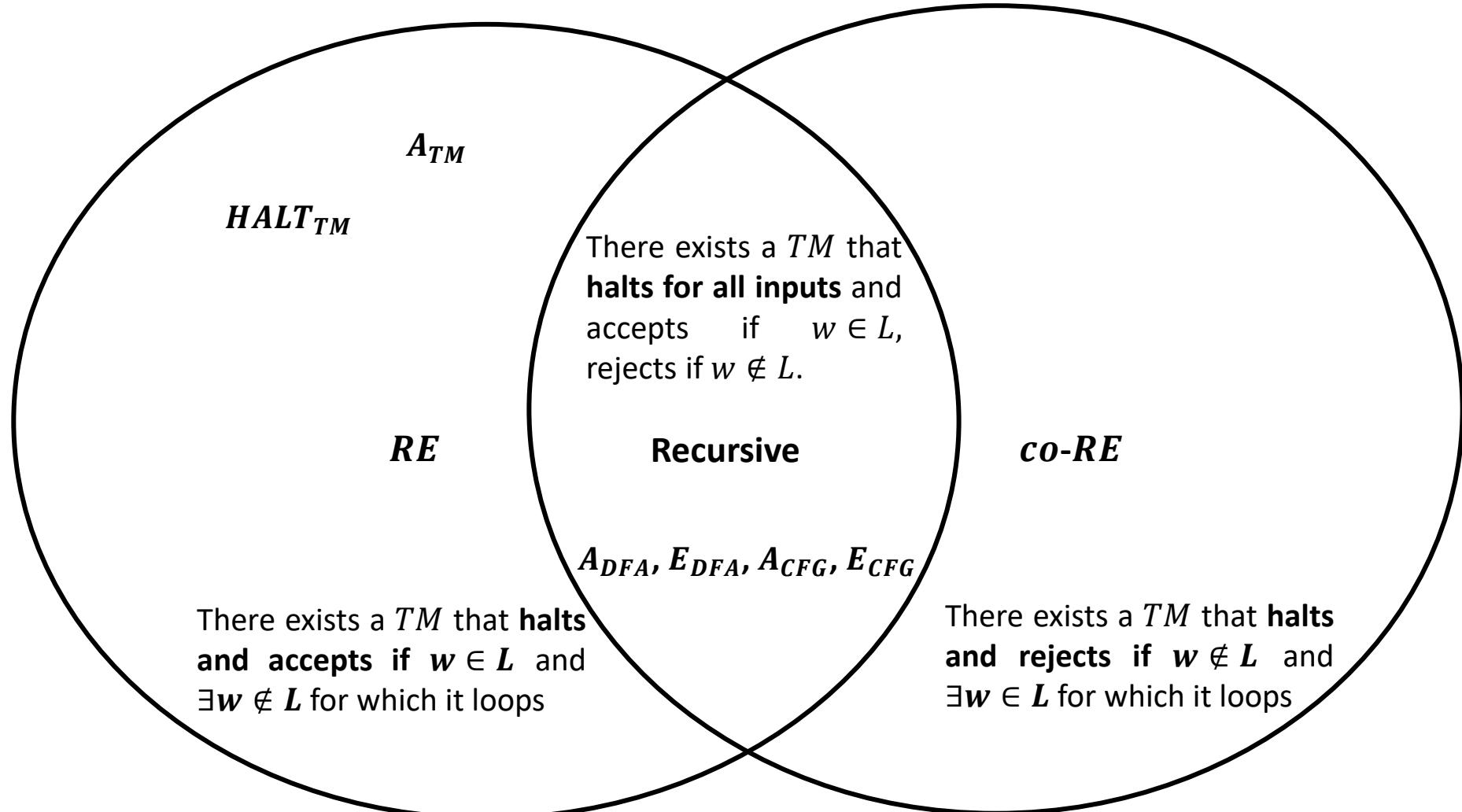
If  $M(w)$  accepts, output *ACCEPT*

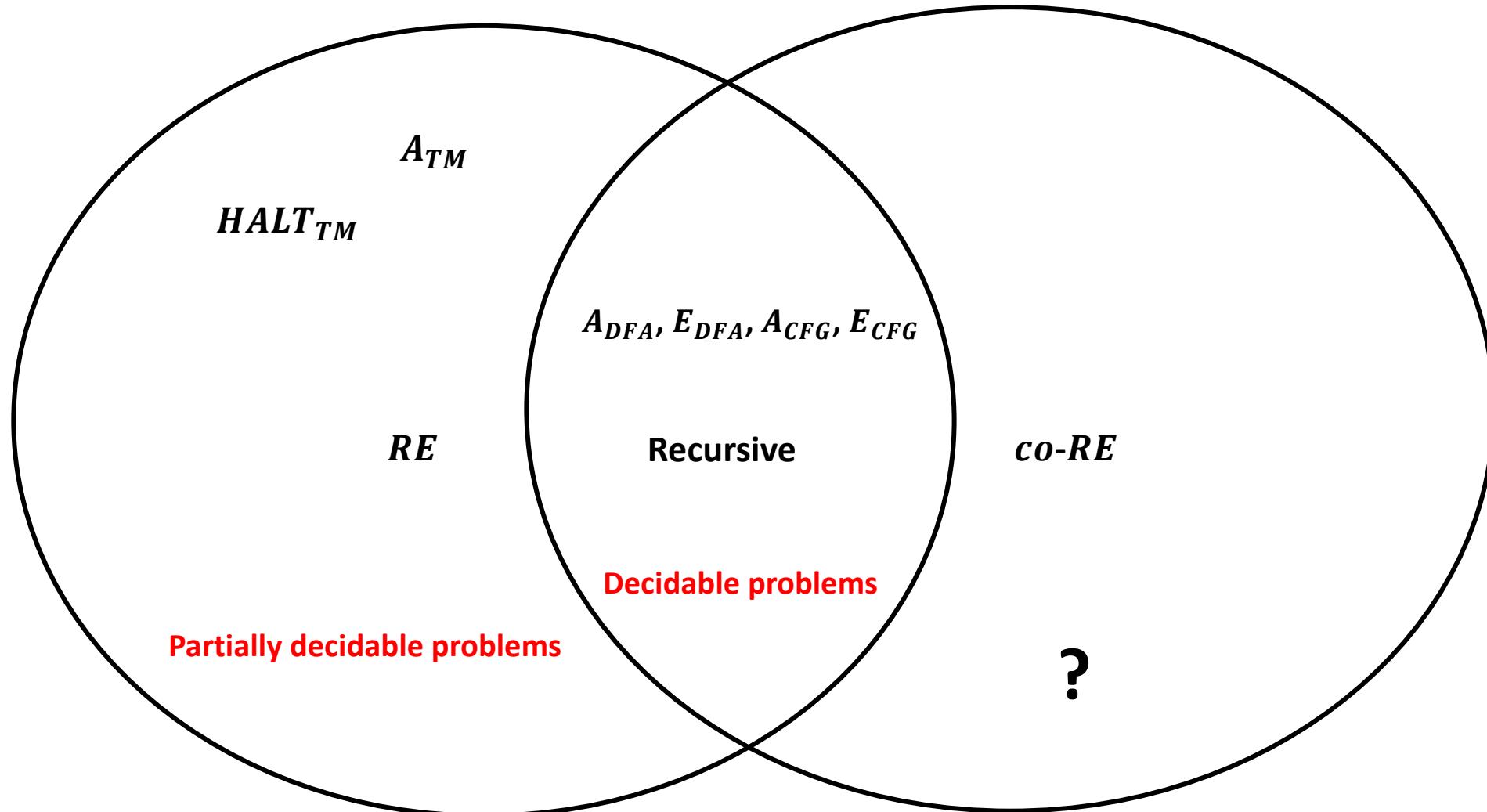
If  $\bar{M}(w)$  rejects, output *REJECT*

So  $R = RE \cap co-RE$





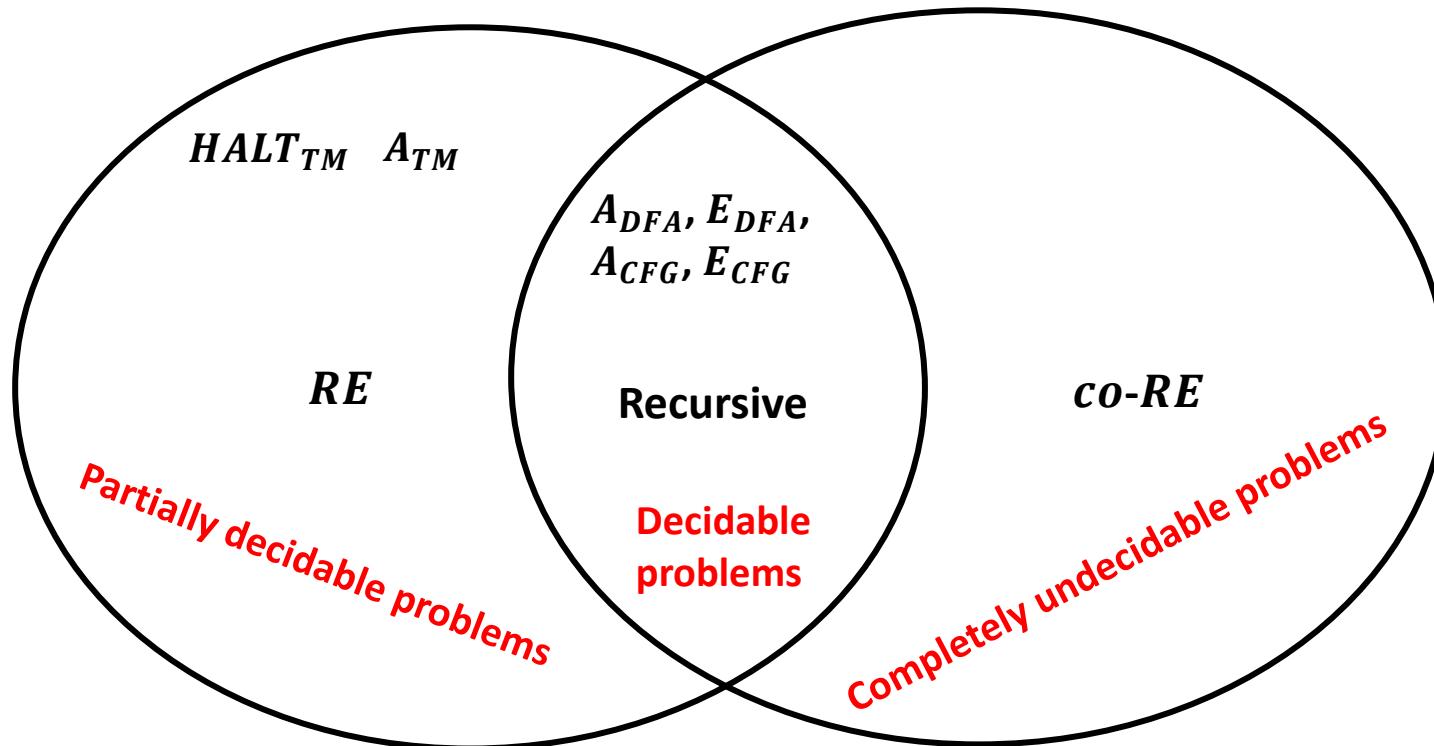




# Closure properties

**Completely undecidable languages:** Languages  $L$  for which there exists at least one instance  $w \in L$ , for which the TM enters into an infinite loop.

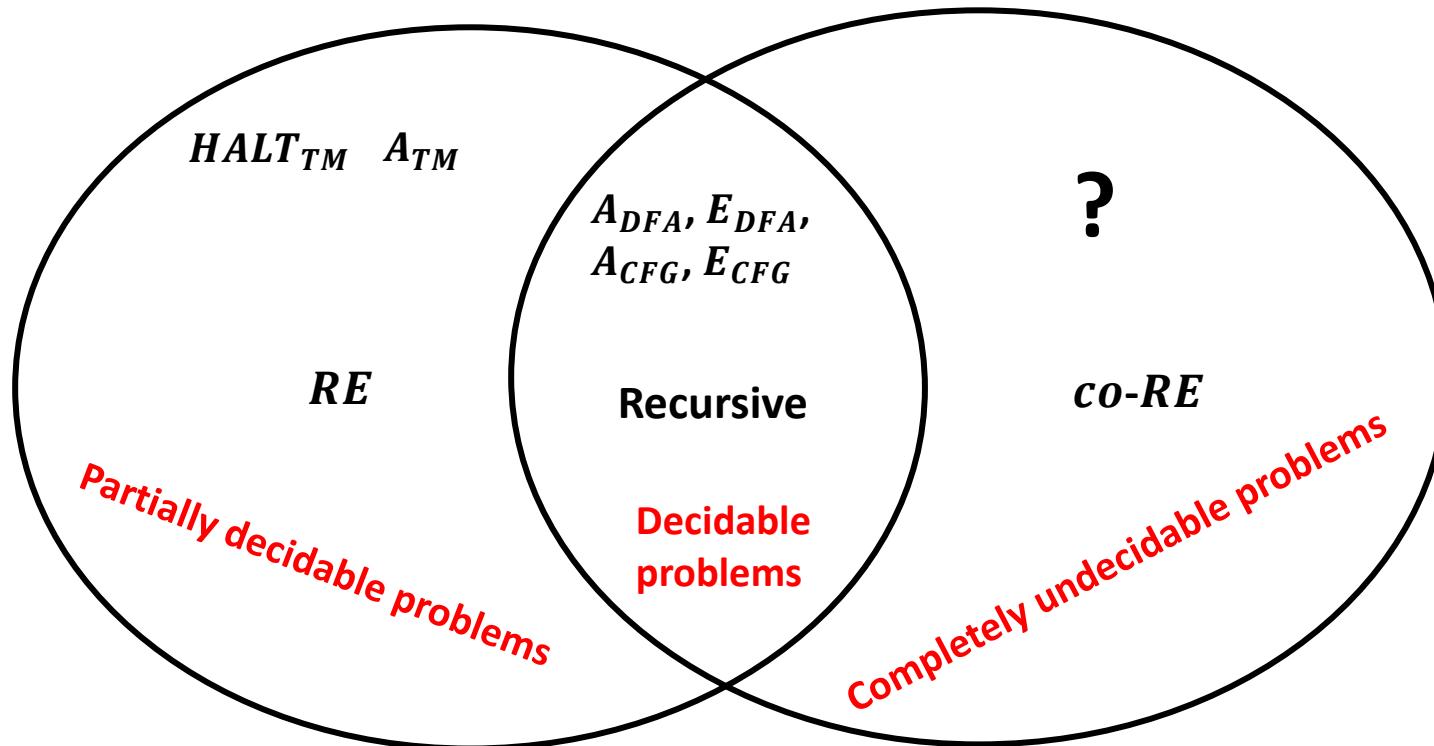
So, languages **that are in  $co\text{-}RE$  but are not recursive** are **completely undecidable**.



# Closure properties

**Completely undecidable languages:** Languages  $L$  for which there exists at least one instance  $w \in L$ , for which the TM enters into an infinite loop.

So, languages **that are in  $co\text{-}RE$  but are not recursive** are **completely undecidable**.



# Closure properties

**Completely undecidable languages:** Languages  $L$  for which there exists at least one instance  $w \in L$ , for which the TM enters into an infinite loop.

If  $L \in RE$  but is not Recursive (partially decidable), then  $\bar{L} \in co\text{-}RE$  but is not recursive. So Complement of any partially decidable language is completely undecidable

- E.g.:  $A_{TM} \in RE$  and so  $\overline{A_{TM}} \in co\text{-}RE$  and is **completely undecidable**

$$A_{TM} = \{\langle M, w \rangle \mid M \text{ accepts input } w\}$$

$$\overline{A_{TM}} = \{\langle M, w \rangle \mid M \text{ doesn't accept input } w\}$$

# Closure properties

**Completely undecidable languages:** Languages  $L$  for which there exists at least one instance  $w \in L$ , for which the TM enters into an infinite loop.

If  $L \in RE$  but is not Recursive (partially decidable), then  $\bar{L} \in co\text{-}RE$  but is not recursive. So Complement of any partially decidable language is completely undecidable

- E.g.:  $A_{TM} \in RE$  and so  $\overline{A_{TM}} \in co\text{-}RE$  and is **completely undecidable**

$$A_{TM} = \{\langle M, w \rangle | M \text{ accepts input } w\}$$

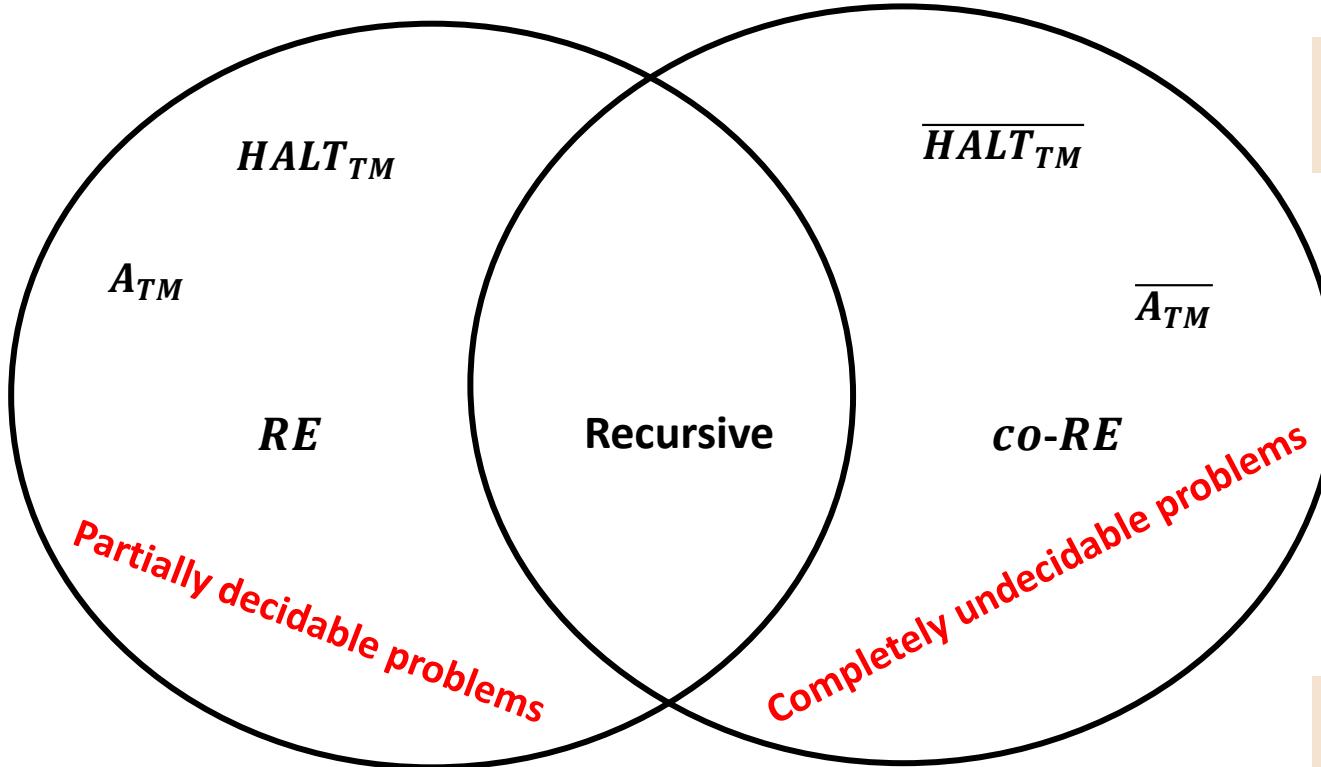
$$\overline{A_{TM}} = \{\langle M, w \rangle | M \text{ doesn't accept input } w\}$$

- Similarly,  $\overline{HALT_{TM}}$  is also completely undecidable

$$\overline{HALT_{TM}} = \{\langle M, w \rangle | M \text{ doesn't halt on input } w\}$$

# Closure properties

**Completely undecidable languages:** Languages  $L$  for which there exists at least one instance  $w \in L$ , for which the TM enters into an infinite loop.



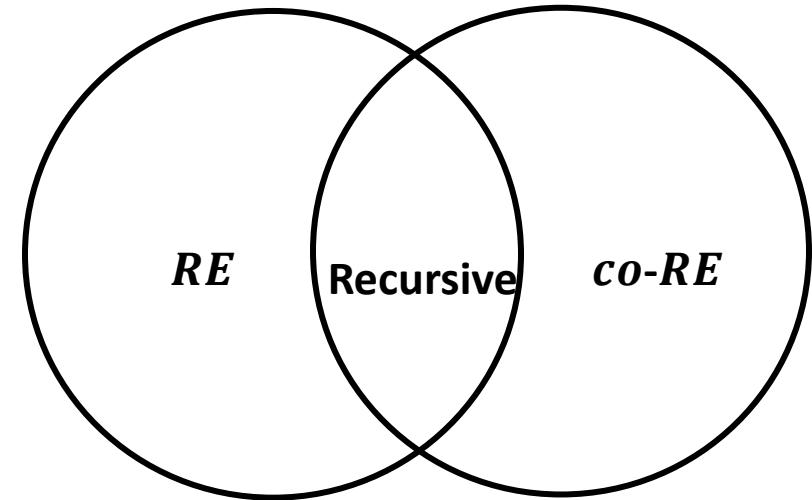
Languages that are in  $co\text{-}RE$  but are not recursive are **completely undecidable**.

Complement of any partially decidable language is completely undecidable.

# Summing up

We have the following:

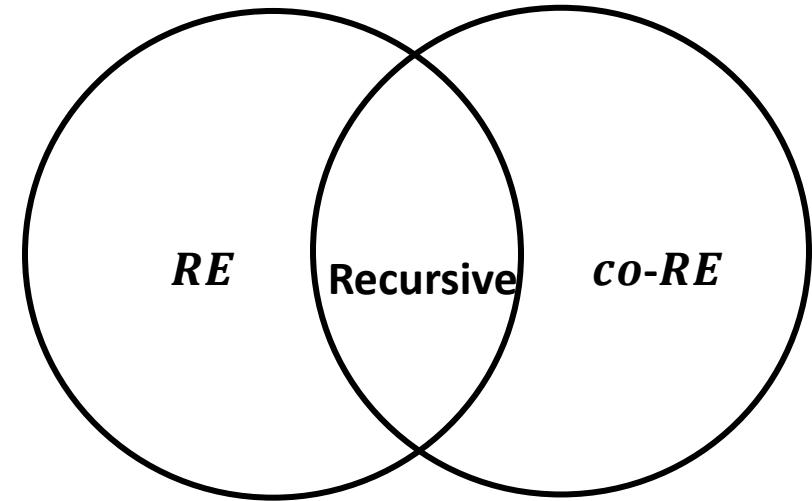
- Recursive Languages are closed under complement, union & intersection
- *RE* is closed under union & intersection but not complement



# Summing up

We have the following:

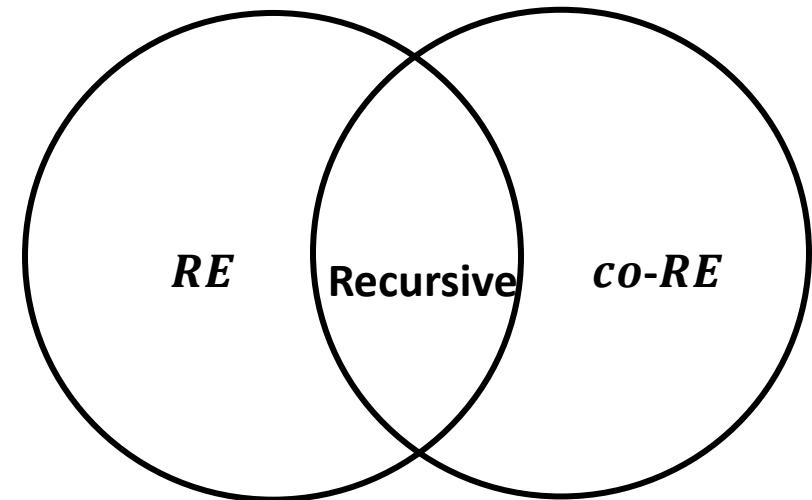
- Recursive Languages are closed under complement, union & intersection
- $RE$  is closed under union & intersection but not complement
- $L \in RE$  and  $\bar{L} \in RE$ , iff  $L$  is Recursive.
- If  $L \in RE$  then  $\bar{L} \in co\text{-}RE$ .
- If  $L \in co\text{-}RE$  then  $\bar{L} \in RE$ .



# Summing up

We have the following:

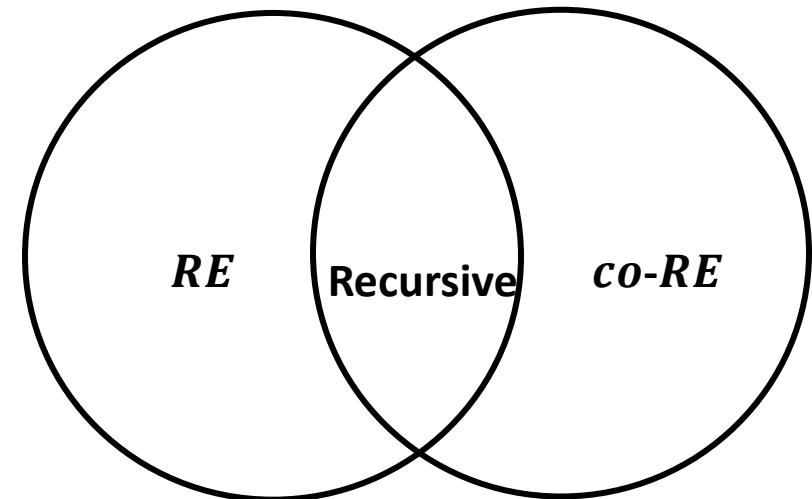
- Recursive Languages are closed under complement, union & intersection
- $RE$  is closed under union & intersection but not complement
- $L \in RE$  and  $\bar{L} \in RE$ , iff  $L$  is Recursive.
- If  $L \in RE$  then  $\bar{L} \in co\text{-}RE$ .
- If  $L \in co\text{-}RE$  then  $\bar{L} \in RE$ .
- $R = RE \cap co\text{-}RE$



# Summing up

We have the following:

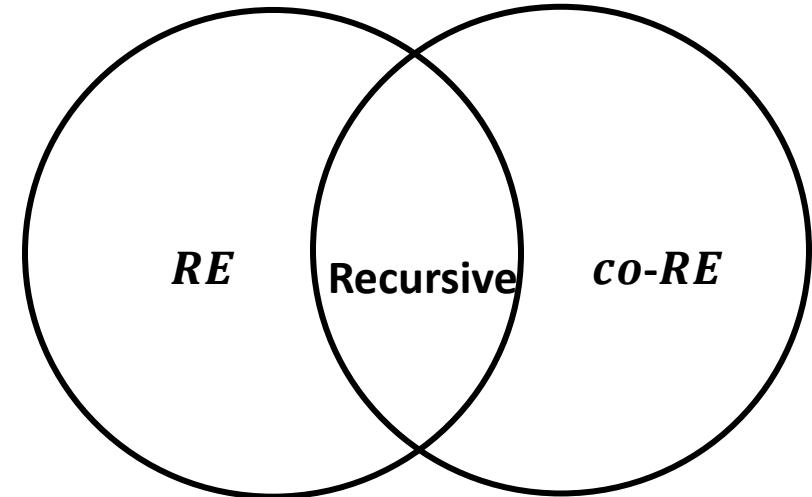
- Recursive Languages are closed under complement, union & intersection
- $RE$  is closed under union & intersection but not complement
- $L \in RE$  and  $\bar{L} \in RE$ , iff  $L$  is Recursive.
- If  $L \in RE$  then  $\bar{L} \in co\text{-}RE$ .
- If  $L \in co\text{-}RE$  then  $\bar{L} \in RE$ .
- $R = RE \cap co\text{-}RE$
- If  $L \in RE$  but is not Recursive, then  $L$  is partially decidable
- If  $L \in co\text{-}RE$  but is not Recursive, then  $L$  is completely undecidable.



# Summing up

We have the following:

- Recursive Languages are closed under complement, union & intersection
- $RE$  is closed under union & intersection but not complement
- $L \in RE$  and  $\bar{L} \in RE$ , iff  $L$  is Recursive.
- If  $L \in RE$  then  $\bar{L} \in co\text{-}RE$ .
- If  $L \in co\text{-}RE$  then  $\bar{L} \in RE$ .
- $R = RE \cap co\text{-}RE$
- If  $L \in RE$  but is not Recursive, then  $L$  is partially decidable
- If  $L \in co\text{-}RE$  but is not Recursive, then  $L$  is completely undecidable.



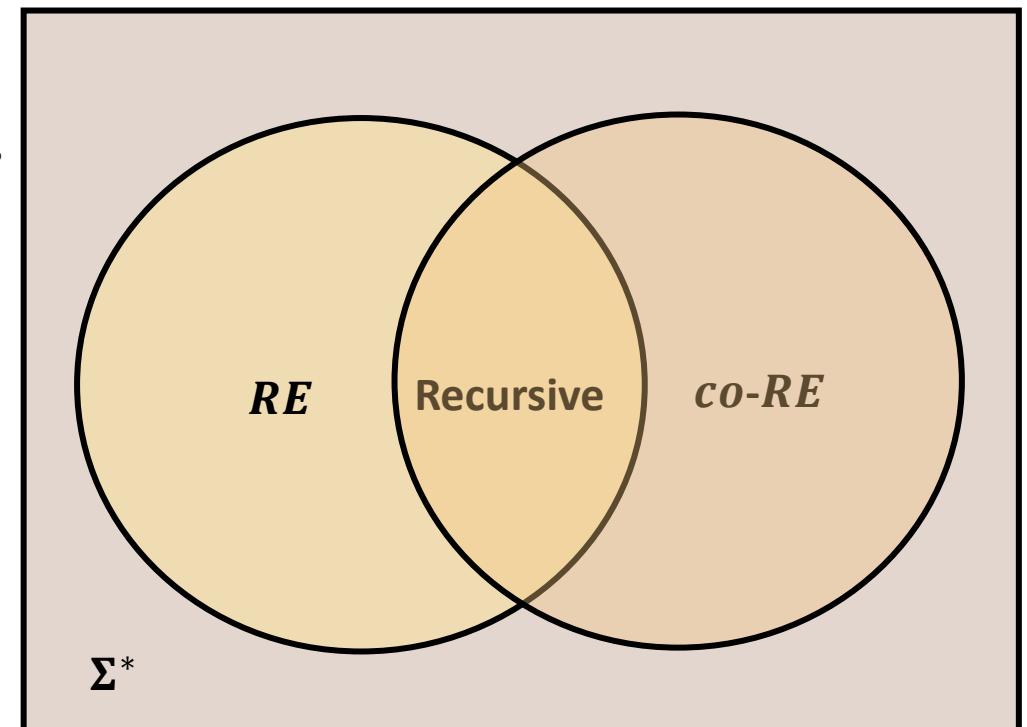
Note that there are languages outside of  $RE \cup co\text{-}RE$ .

# Summing up

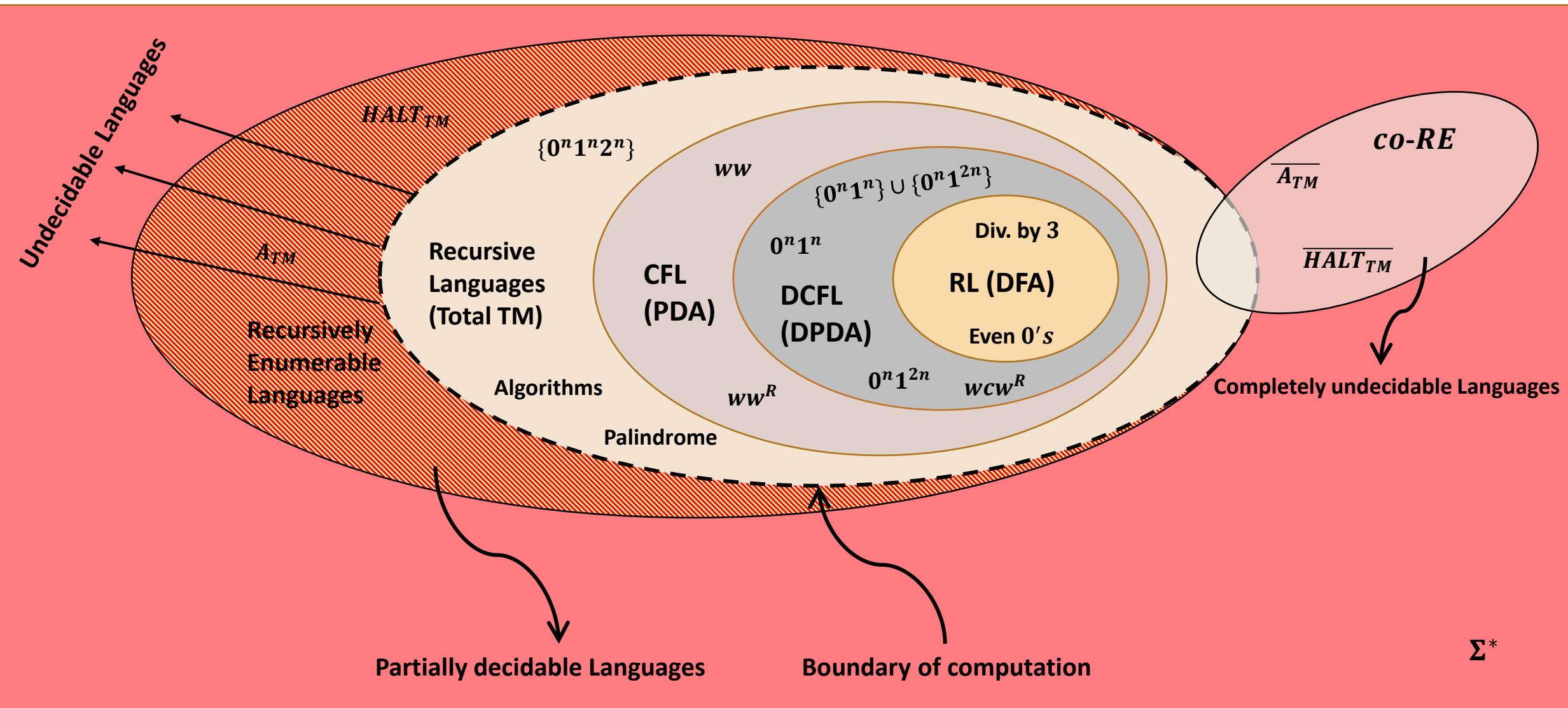
We have the following:

- Recursive Languages are closed under complement, union & intersection
- $RE$  is closed under union & intersection but not complement
- $L \in RE$  and  $\bar{L} \in RE$ , iff  $L$  is Recursive.
- If  $L \in RE$  then  $\bar{L} \in co\text{-}RE$ .
- If  $L \in co\text{-}RE$  then  $\bar{L} \in RE$ .
- $R = RE \cap co\text{-}RE$
- If  $L \in RE$  but is not Recursive, then  $L$  is partially decidable
- If  $L \in co\text{-}RE$  but is not Recursive, then  $L$  is completely undecidable.

Note that there are languages outside of  $RE \cup co\text{-}RE$ .



# Everything in one slide



# The Road ahead to Complexity Theory...

- We finished up by looking at problems that are decidable/undecidable.
- There are many things that I couldn't cover:
  - Several cool problems that can be proven to be decidable/undecidable and classified to be in  $R$ ,  $RE$ ,  $co\text{-}RE$  etc
  - Mapping reduction, Recursion Theorem, Rice's Theorem
- Problems that are not computable are highly likely to never be solved on feasible computational devices.
- In how much time/space can **computable problems** be solved in? Complexity Theory: classify problems according to their hardness.
- Million dollar problems waiting to be solved!
- E.g.: Quantum computers model how nature computes at the fundamental level: provably faster than classical machines on several problems and most likely violates the Extended Church Turing Thesis.

**Thank You!**