CS 302.1 - Automata Theory

Lecture 06

Shantanav Chakraborty

Center for Quantum Science and Technology (CQST)

Center for Security, Theory and Algorithms (CSTAR)

IIIT Hyderabad

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

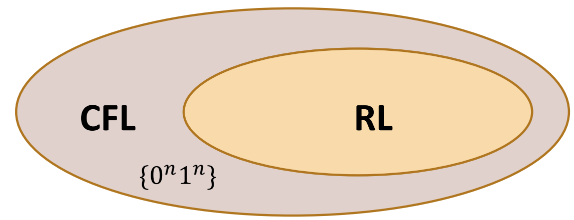
HYDERABAD

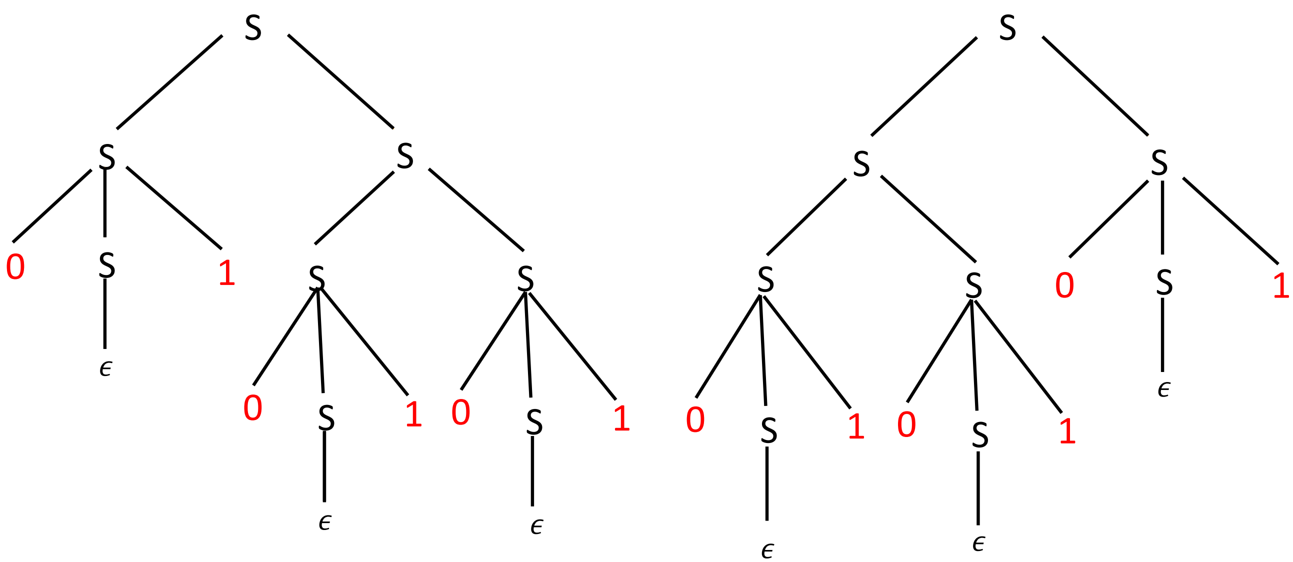
Quick Recap

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

$$\boldsymbol{V} \rightarrow (\boldsymbol{V} \cup \boldsymbol{T})^*$$

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





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

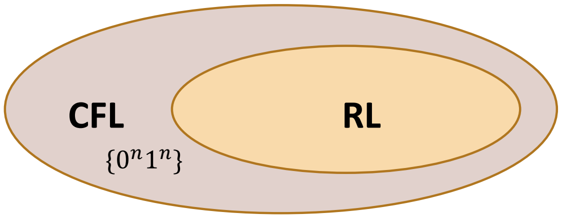
Ambiguous grammars: There exists $\boldsymbol{\omega} \in \boldsymbol{L}(\boldsymbol{G})$**,** such that there are **two or more leftmost derivations for $\boldsymbol{\omega}$** (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for $\boldsymbol{\omega}$. Ambiguity** may not be desirable

Quick Recap

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

$$\boldsymbol{V} \rightarrow (\boldsymbol{V} \cup \boldsymbol{T})^*$$

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 $\boldsymbol{\omega} \in \boldsymbol{L}(\boldsymbol{G})$**,** such that there are **two or more leftmost derivations for $\boldsymbol{\omega}$** (or equivalently two or more rightmost derivations) or equivalently **two or more parse trees for $\boldsymbol{\omega}$. Ambiguity** may not be desirable

Chomsky Normal Form: If every *rule* of the CFG is of the form

$\boldsymbol{A} \rightarrow \boldsymbol{BC}$ [B, C are not start variables]

$\boldsymbol{A} \rightarrow \boldsymbol{a}$ [a is a terminal]

$\boldsymbol{S} \rightarrow \boldsymbol{\epsilon}$ [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 for **deciding** whether w is generated by a CFG G.

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 ϵ 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 $S' \to S$

2. **Remove ϵ rules of the form $A \to \epsilon$**

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 $B \to uAvAw$ (u, v, w can be strings of variables and terminals)

Then new rules: $B \to uAvAw|uvAw|uAvw|uvw$

What if you had a rule such as $B \to A$? Then we would have needed to add a rule $B \to \epsilon$ (unless this rule has been already removed) as B is a **nullable variable.**

Repeat this procedure, until all ϵ-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 $S' \to S$

2. **Remove ϵ rules of the form $A \to \epsilon$**

E.g.:

$$S \to 0|X0|ZYZ$$
$$X \to Y|\epsilon$$
$$Y \to 1|X$$

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 $\boldsymbol{\epsilon}$ rules of the form $\boldsymbol{A \rightarrow \epsilon}$ (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 ϵ rules are removed)

E.g.:
$$S \rightarrow 0|X0|ZYZ$$
$$X \rightarrow Y|\epsilon$$
$$Y \rightarrow 1|X$$

To remove $X \rightarrow \epsilon$, we add new rules:
$$S \rightarrow 0|X0|ZYZ$$
$$X \rightarrow Y$$
$$Y \rightarrow 1|X|\epsilon$$

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 $\boldsymbol{\epsilon}$ rules of the form $\boldsymbol{A \to \epsilon}$ (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 ϵ rules are removed)

E.g.:

$$S \to 0|X0|ZYZ$$
$$X \to Y|\epsilon$$
$$Y \to 1|X$$

To remove $X \to \epsilon$, we add new rules:

$$S \to 0|X0|ZYZ$$
$$X \to Y$$
$$Y \to 1|X|\epsilon$$

To remove $Y \to \epsilon$, we add:

$$S \to 0|X0|ZYZ|ZZ$$
$$X \to Y$$
$$Y \to 1|X$$

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 $S' \to S$

2. Remove ϵ rules of the form $A \to \epsilon$

3. **Remove unit rules of the form $\boldsymbol{A \to B}$**

We **remove the rule $\boldsymbol{A \to B}$** and **whenever a rule $\boldsymbol{B \to u}$ appears** (u is a string of terminals and variables), we **add a new rule $\boldsymbol{A \to u}$**, 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 $\boldsymbol{A \to B}$ (Whenever a rule $B \to u$ appears, we add a new rule $A \to u$, unless this rule was already removed. Repeat these steps until all unit rules are removed.)

E.g.:

$$S \to A|11$$
$$A \to B|1$$
$$B \to S|0$$

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 $\boldsymbol{A \to B}$ (Whenever a rule $B \to u$ appears, we add a new rule $A \to u$, unless this rule was already removed. Repeat these steps until all unit rules are removed.)

E.g.:

$$S \to A|11$$
$$A \to B|1$$
$$B \to S|0$$

Remove $\boldsymbol{S \to A}$	Remove $\boldsymbol{A \to B}$	Remove $\boldsymbol{B \to S}$	Remove $\boldsymbol{B \to B}$	Remove $\boldsymbol{S \to B}$	Remove $\boldsymbol{A \to S}$
$S \to 11\|\boldsymbol{B}\|\mathbf{1}$	$S \to 11\|B\|1$	$S \to 11\|B\|1$	$S \to 11\|B\|1$	$S \to 11\|\mathbf{0}\|1$	$S \to 11\|\mathbf{0}\|1$
$A \to B\|1$	$A \to 1\|\boldsymbol{S}\|\mathbf{0}$	$A \to 1\|S\|0$	$A \to 1\|S\|0$	$A \to 1\|S\|0$	$A \to 1\|\mathbf{11}\|0$
$B \to S\|0$	$B \to S\|0$	$B \to 0\|\mathbf{11}\|\mathbf{1}\|\boldsymbol{B}$	$B \to 0\|11\|1$	$B \to 0\|11\|1$	$B \to 0\|11\|1$

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 $\boldsymbol{A \rightarrow B}$ (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.)

$$\begin{aligned} S &\rightarrow A|11 \\ A &\rightarrow B|1 \\ B &\rightarrow S|0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} \boldsymbol{S} &\rightarrow \mathbf{11|0|1} \\ \boldsymbol{A} &\rightarrow \mathbf{1|11|0} \\ \boldsymbol{B} &\rightarrow \mathbf{0|11|1} \end{aligned}$$

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 ϵ rules of the form $A \rightarrow \epsilon$

3. Remove unit rules of the form $A \rightarrow B$

4. **Remove long rules of the form $\boldsymbol{A \rightarrow u_1 u_2 \cdots u_k}$**

Note that each u_i could be a variable or a terminal. We do the following:

- Replace $A \rightarrow u_1 u_2 \cdots u_k$, $(k \geq 3)$ with the rules $A \rightarrow u_1 A_1,\ A_1 \rightarrow u_2 A_2, \cdots, A_{k-2} \rightarrow u_{k-1} u_k$

- 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 $\boldsymbol{S' \to S}$

Remove $\boldsymbol{\epsilon}$ rules of the form $\boldsymbol{A \to \epsilon}$ (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 ϵ rules are removed)**.**

Remove unit rules of the form $\boldsymbol{A \to B}$ (Whenever a rule $B \to u$ appears, we add a new rule $A \to u$, unless this rule was already removed. Repeat these steps until all unit rules are removed.)

Remove long rules of the form $\boldsymbol{A \to u_1 u_2 \cdots u_k}$ (Replace $A \to u_1 u_2 \cdots u_k$, $(k \geq 3)$ with the rules $A \to u_1 A_1$, $A_1 \to u_2 A_2, \cdots, A_{k-2} \to u_{k-1} u_k$; Replace any terminal u_i in the preceding rules with the new variable U_i and add the rule $U_i \to u_i$).

Chomsky Normal Form

CNF:

$\boldsymbol{A \rightarrow BC}$ [B, C are not start variables]
$\boldsymbol{A \rightarrow a}$ [a is a terminal]
$\boldsymbol{S \rightarrow \epsilon}$ [S is the Start Variable]

Convert the CFG

$$\boldsymbol{S \rightarrow ASA|aB}$$
$$\boldsymbol{A \rightarrow B|S}$$
$$\boldsymbol{B \rightarrow b|\epsilon}$$

to CNF.

1. Add a new start variable

$$\boldsymbol{S' \rightarrow S}$$
$$S \rightarrow ASA|aB$$
$$A \rightarrow B|S$$
$$B \rightarrow b|\epsilon$$

2a. Remove ϵ rules ($B \rightarrow \epsilon$)

$$S' \rightarrow S$$
$$S \rightarrow ASA|aB|\boldsymbol{a}$$
$$A \rightarrow B|S|\boldsymbol{\epsilon}$$
$$B \rightarrow b$$

2b. Remove ϵ rules (A$\rightarrow$ ϵ)

$$S' \rightarrow S$$
$$S \rightarrow ASA|aB|a|\boldsymbol{AS}|\boldsymbol{SA}|\boldsymbol{S}$$
$$A \rightarrow B|S$$
$$B \rightarrow b$$

Chomsky Normal Form

CNF:

$\boldsymbol{A \to BC}$ [B, C are not start variables]
$\boldsymbol{A \to a}$ [a is a terminal]
$\boldsymbol{S \to \epsilon}$ [S is the Start Variable]

Convert the CFG

$$\boldsymbol{S \to ASA | aB}$$
$$\boldsymbol{A \to B | S}$$
$$\boldsymbol{B \to b | \epsilon}$$

to CNF.

3a. Remove $\boldsymbol{S \to S}$

$$S' \to S$$
$$S \to ASA|aB|a|AS|SA$$
$$A \to B|S$$
$$B \to b$$

3b. Remove $\boldsymbol{S' \to S}$

$$S' \to \boldsymbol{ASA}|\boldsymbol{aB}|\boldsymbol{a}|\boldsymbol{AS}|\boldsymbol{SA}$$
$$S \to ASA|aB|a|AS|SA$$
$$A \to B|S$$
$$B \to b$$

3c. Remove A→ $\boldsymbol{B}$

$$S' \to ASA|aB|a|AS|SA$$
$$S \to ASA|aB|a|AS|SA$$
$$A \to S|\boldsymbol{b}$$
$$B \to b$$

3d. Remove A→ $\boldsymbol{S}$

$$S' \to ASA|aB|a|AS|SA$$
$$S \to ASA|aB|a|AS|SA$$
$$A \to b|\boldsymbol{ASA}|\boldsymbol{aB}|\boldsymbol{a}|\boldsymbol{AS}|\boldsymbol{SA}$$
$$B \to b$$

Chomsky Normal Form

CNF:

$\boldsymbol{A \to BC}$ [B, C are not start variables]
$\boldsymbol{A \to a}$ [a is a terminal]
$\boldsymbol{S \to \epsilon}$ [S is the Start Variable]

Convert the CFG

$$\boldsymbol{S \to ASA|aB}$$
$$\boldsymbol{A \to B|S}$$
$$\boldsymbol{B \to b|\epsilon}$$

to CNF.

3d. Remove A→ $\boldsymbol{S}$

$$S' \to ASA|aB|a|AS|SA$$
$$S \to ASA|aB|a|AS|SA$$
$$A \to b|ASA|aB|a|AS|SA$$
$$B \to b$$

4a. Remove long rules

$$S' \to A\boldsymbol{SA}|aB|a|AS|SA$$
$$S \to ASA|aB|a|AS|SA$$
$$A \to b|ASA|aB|a|AS|SA$$
$$B \to b$$

There are other rules of the form: Var → ASA

4b. Remove long rules

$$S' \to A\boldsymbol{U}|aB|a|AS|SA$$
$$S \to A\boldsymbol{U}|aB|a|AS|SA$$
$$A \to b|A\boldsymbol{U}|aB|a|AS|SA$$
$$U \to SA$$
$$B \to b$$

4c. Remove long rules

$$S' \to AU|\boldsymbol{V}B|a|AS|SA$$
$$S \to AU|\boldsymbol{V}B|a|AS|SA$$
$$A \to b|AU|VB|a|AS|SA$$
$$U \to SA$$
$$\boldsymbol{V \to a}$$
$$B \to b$$

Chomsky Normal Form

CNF:

$$\boldsymbol{A \to BC} \qquad [B, C \text{ are not start variables}]$$
$$\boldsymbol{A \to a} \qquad [a \text{ is a terminal}]$$
$$\boldsymbol{S \to \epsilon} \qquad [S \text{ is the Start Variable}]$$

Convert the CFG

$$\boldsymbol{S \to ASA|aB}$$
$$\boldsymbol{A \to B|S}$$
$$\boldsymbol{B \to b|\epsilon}$$

to CNF.

$$\boldsymbol{S' \to AU|VB|a|AS|SA}$$
$$\boldsymbol{S \to AU|VB|a|AS|SA}$$
$$\boldsymbol{A \to b|AU|VB|a|AS|SA}$$
$$\boldsymbol{U \to SA}$$
$$\boldsymbol{V \to a}$$
$$\boldsymbol{B \to b}$$

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 ω 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 ω 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 ω 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 ω 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 ω 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.

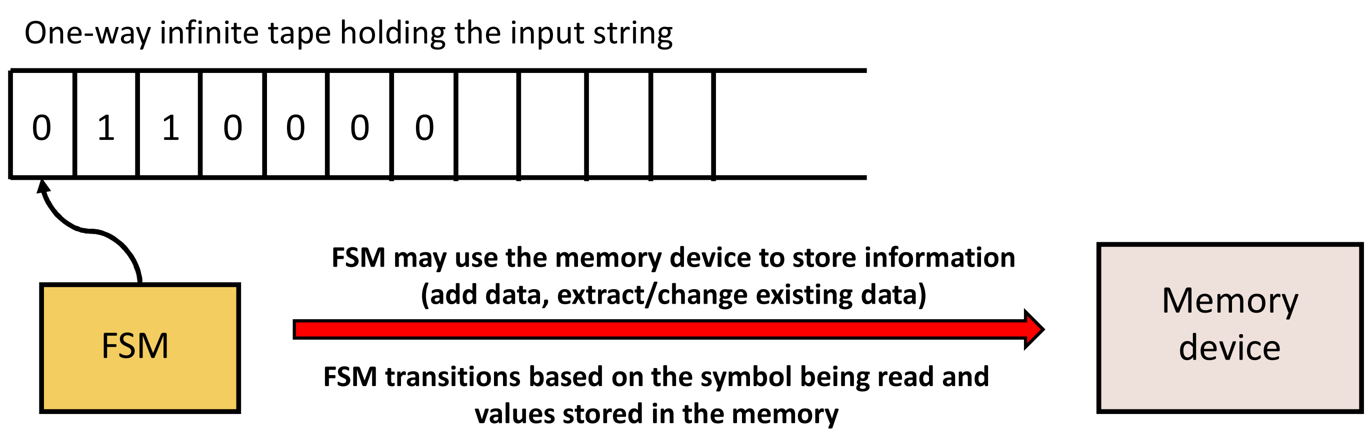
E.g.: $\{0^n 1^n, n \in \mathbb{N}\}$

Pushdown Automata

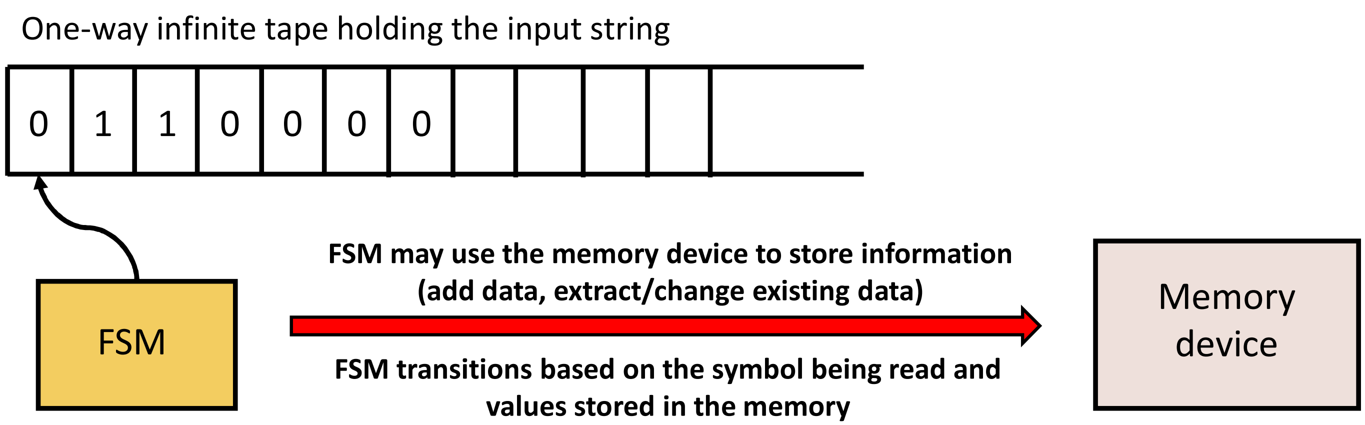
Intuition to build an Automata for CFL

- **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



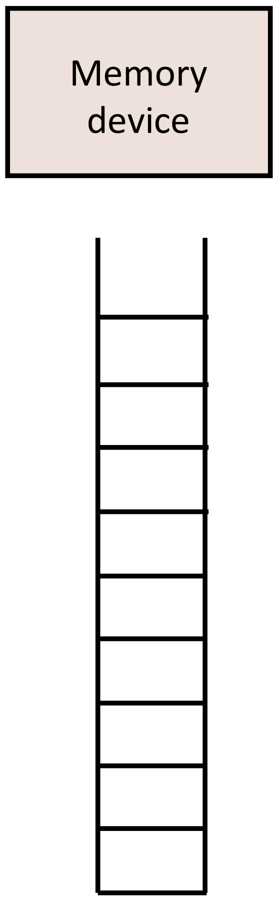
The memory device

- Simple memory device with unbounded memory.

Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, new elements can be added to the Stack (**PUSH**).
- At any stage, the element at the **top** of the STACK can be read by removing it from the stack (**POP**).



Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

PUSH

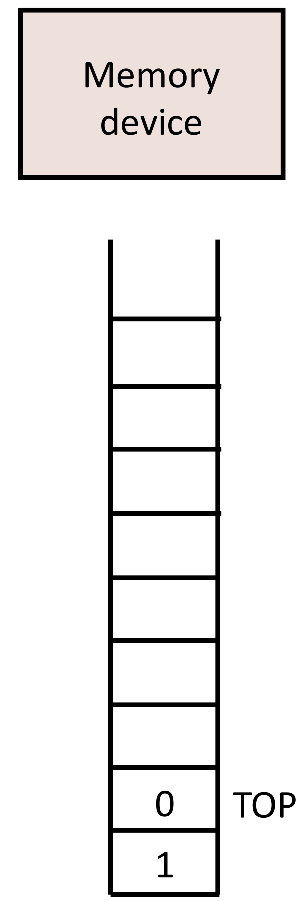
- New symbols can be **pushed** in to the STACK.

  E.g: PUSH 1

- The Top of the STACK now covers the old stack top, i.e.

  TOP = TOP + 1

- The size of the stack keeps growing.



Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

PUSH

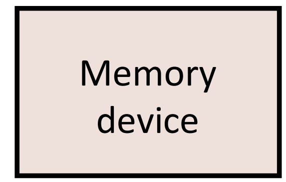
- New symbols can be **pushed** in to the STACK.

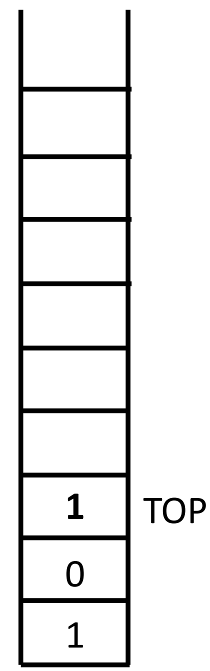
  E.g: PUSH 1

- The Top of the STACK now covers the old stack top, i.e.

  TOP = TOP + 1

- The size of the stack keeps growing.





Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

PUSH

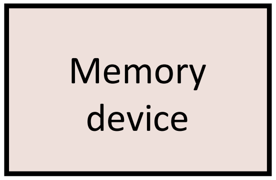
- New symbols can be **pushed** in to the STACK.

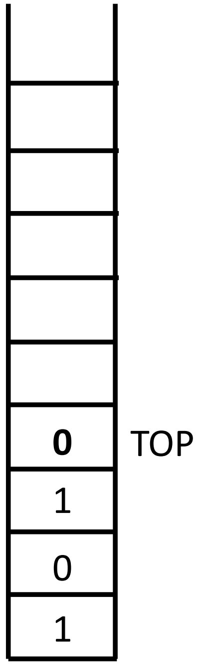
  E.g: **PUSH 0**

- The Top of the STACK now covers the old stack top, i.e.

  TOP = TOP + 1

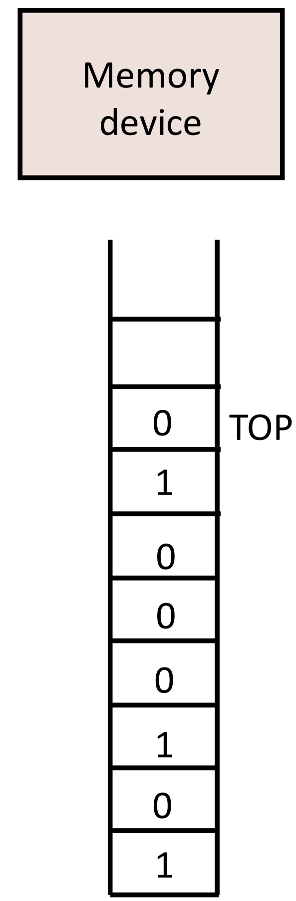
- The size of the stack keeps growing.





Pushdown Automata

The memory device



- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

PUSH

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

  TOP = TOP + 1

- The size of the stack keeps growing.

Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

POP

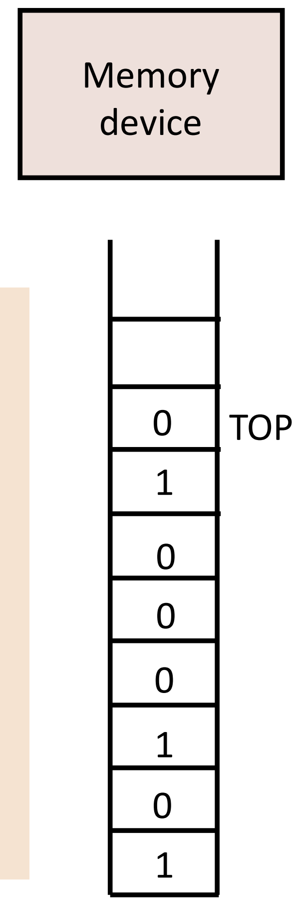
- The element from the TOP of the stack can be **popped** out

E.g.: **POP 0**

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

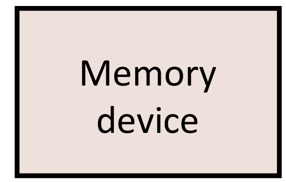
TOP = TOP - 1

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



Pushdown Automata

The memory device



- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

POP

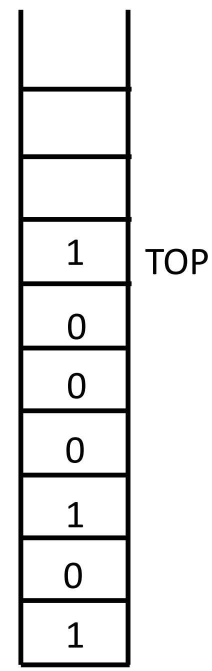
- The element from the TOP of the stack can be **popped** out

E.g.: **POP 0**

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

TOP = TOP - 1

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



Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

POP

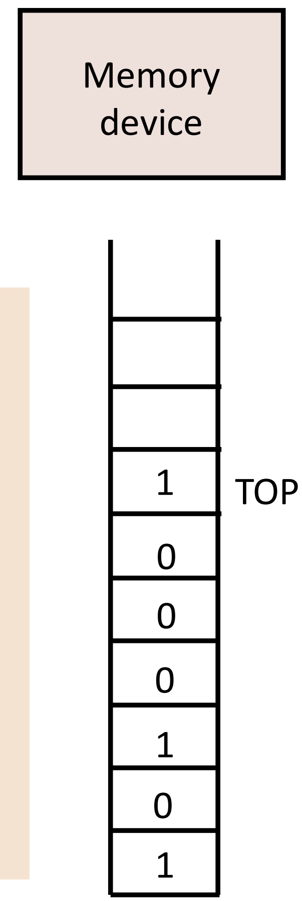
- The element from the TOP of the stack can be **popped** out

E.g.: **POP 1**

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

TOP = TOP - 1

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



Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, elements can be pushed or popped.

POP

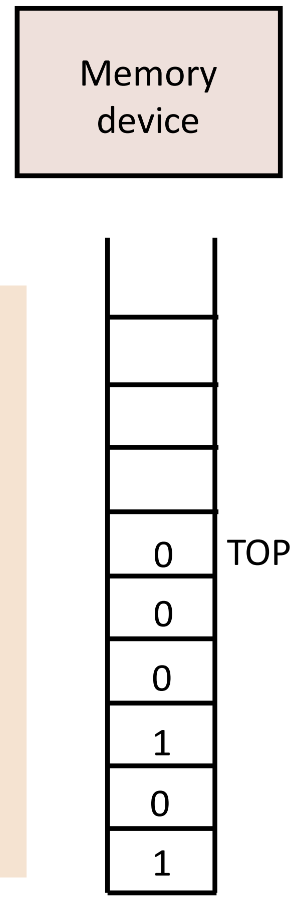
- The element from the TOP of the stack can be **popped** out

E.g.: **POP 1**

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

TOP = TOP - 1

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



Pushdown Automata

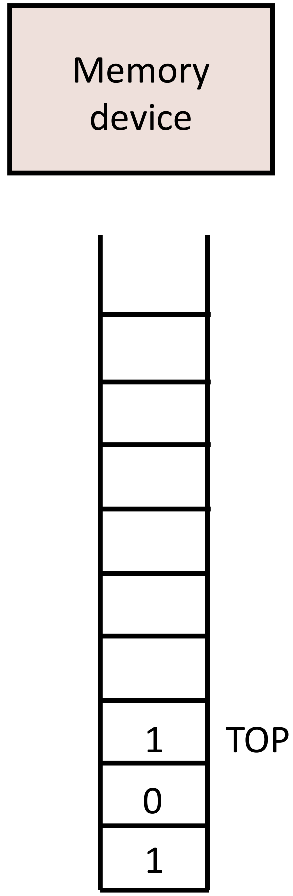
The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- Last In First Out (**LIFO**)

POP

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

- How would you know that the STACK is EMPTY?



Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- Last In First Out (**LIFO**)

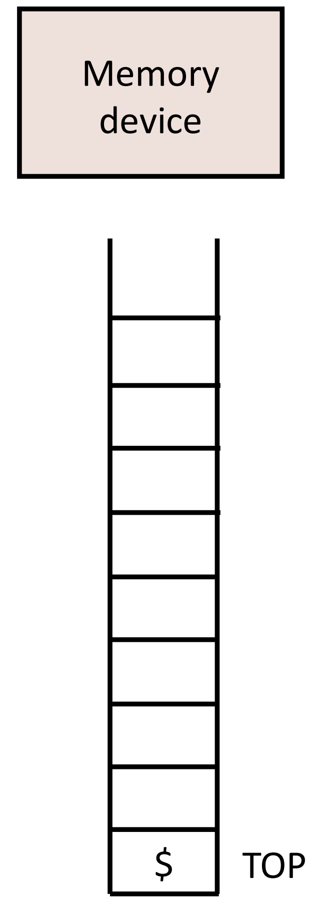
POP

- The element from the TOP of the stack can be **popped** out.
- TOP = 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 the popped element = $, 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:

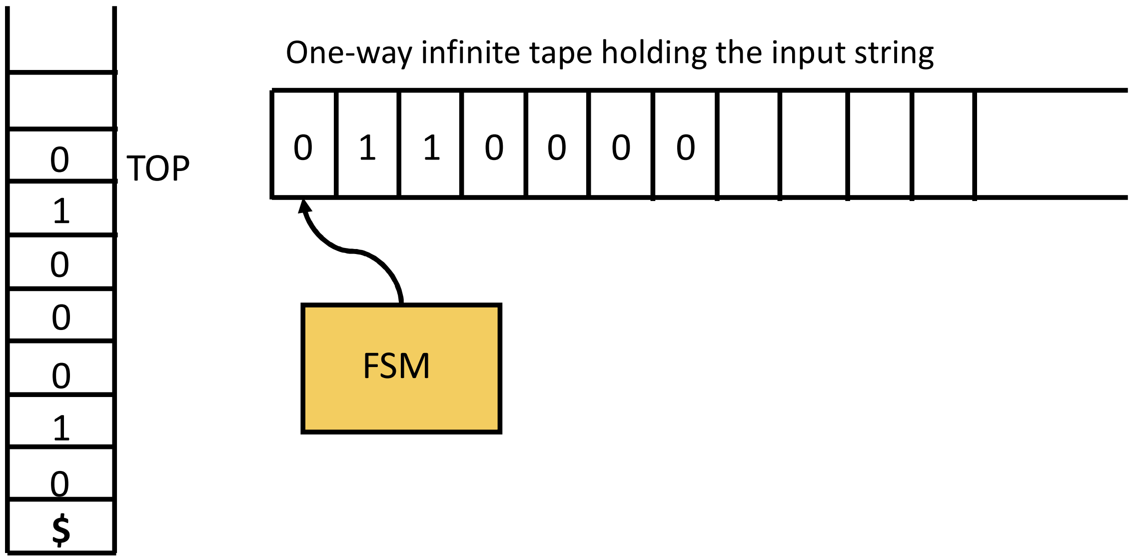
PUSH

- New symbols can be **pushed** in to the STACK.
- TOP = TOP + 1

POP

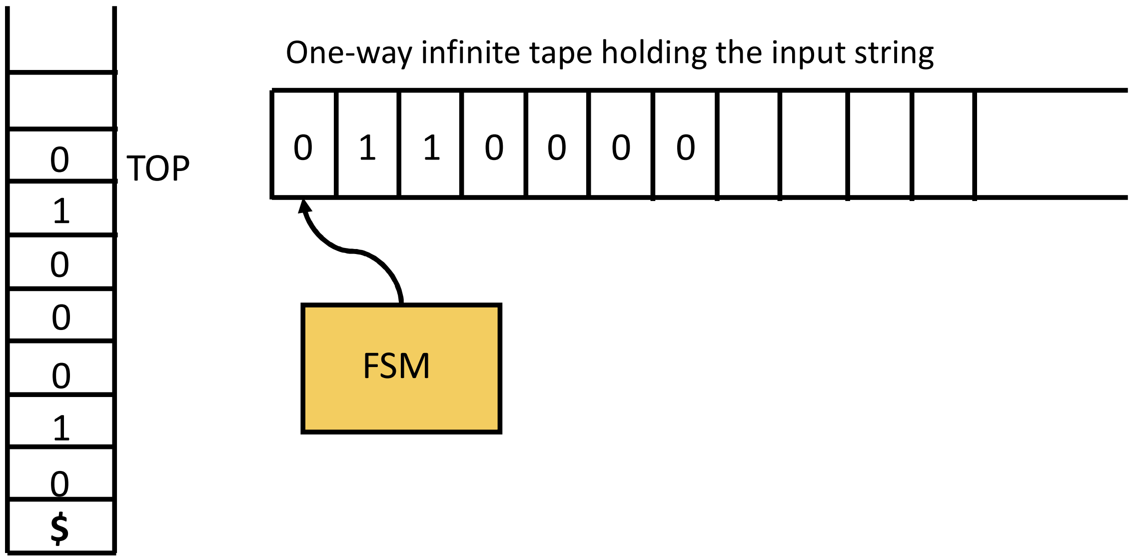
- The element from the TOP of the stack can be **popped** out.
- TOP = TOP - 1
- Elements can be popped until STACK is EMPTY.

Pushdown Automata



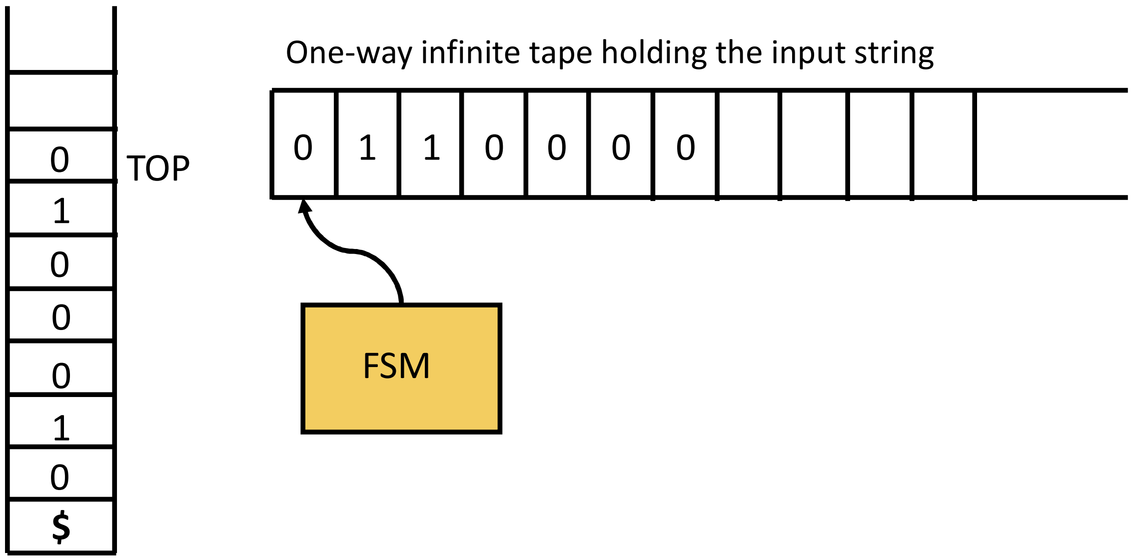
- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:

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 & 0 is popped, transition from i to j)

Pushdown Automata

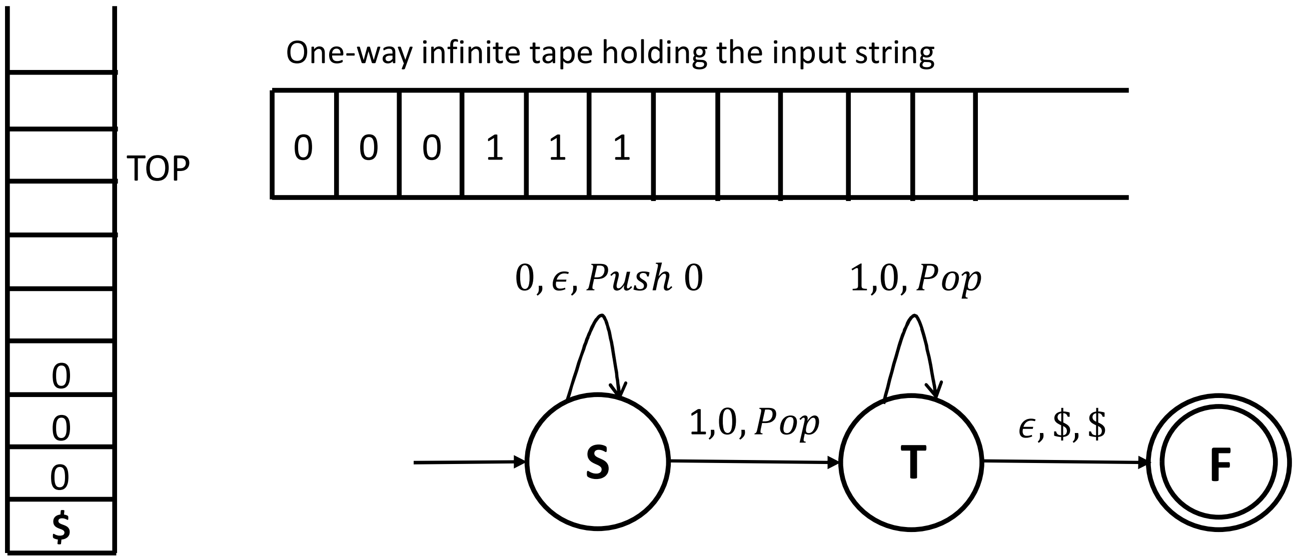


- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.

- The FSM:
  - Transitions based on the Input symbol and the element at the top of the stack (e.g.: If I/P symbol = 0 & POP 0, transition from i to j)

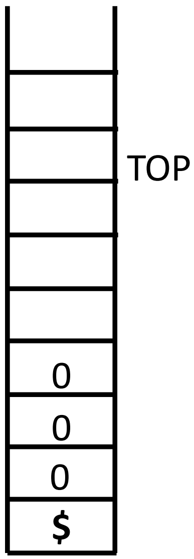
  - Pushes new elements into the Stack (e.g.: If I/P symbol = 0, PUSH 0, transition from i to j).

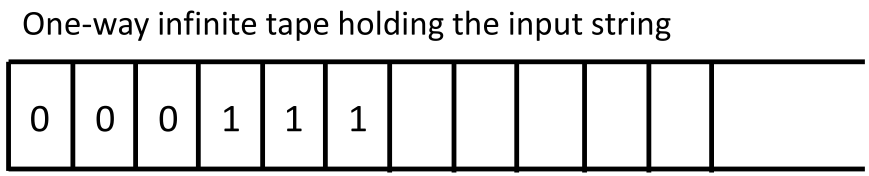
Pushdown Automata

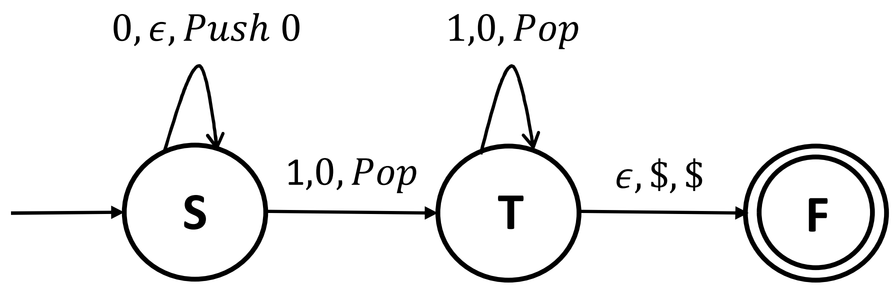


- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:
  - Transitions based on the Input symbol and the element at the top of the stack
  - Pops the element at the top of the Stack.
  - Pushes new elements into the Stack.

Pushdown Automata





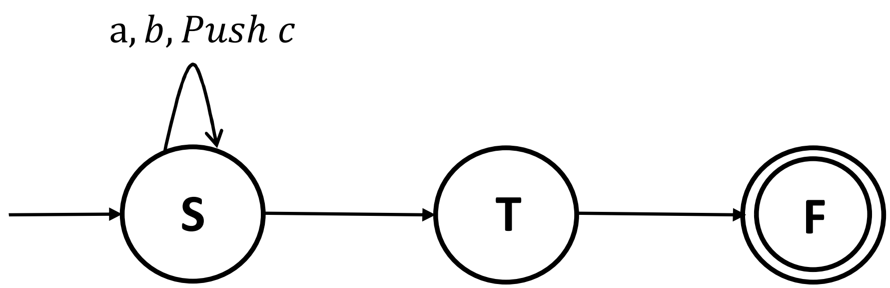


PDAs are **non-deterministic**.

- Missing transitions
- ϵ-transitions
- Multiple transitions/input symbol possible

Pushdown Automata

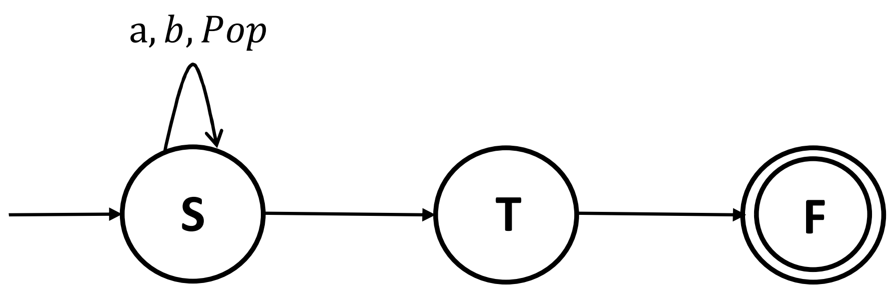
- How to represent a transition in a PDA?



If input symbol = a, Stack top = b (if b is popped), Push c onto the Stack a remain in S

Pushdown Automata

- How to represent a transition in a PDA?

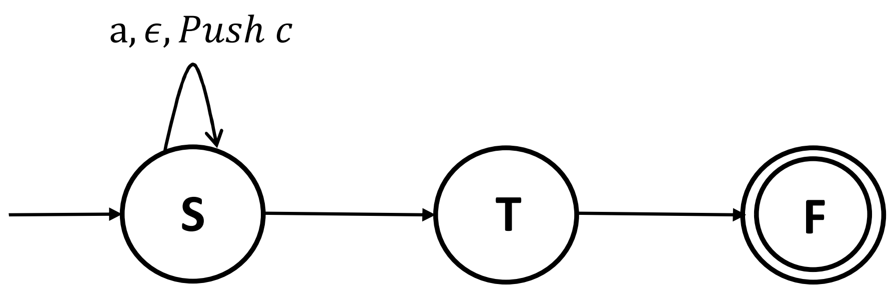


If input symbol = a, and b is popped, remain in S.

(If the symbol read is a and the element at the Stack TOP $= b$, then remain in S)

Pushdown Automata

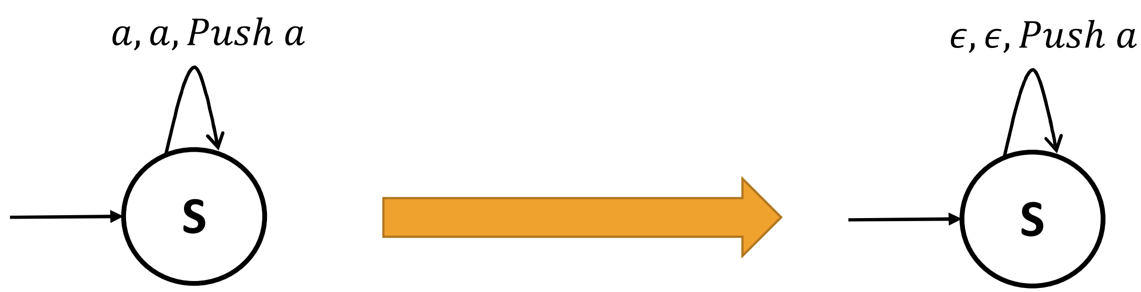
- How to represent a transition in a PDA?



If input symbol = a, then Push c

Pushdown Automata

- How to represent a transition in a PDA?

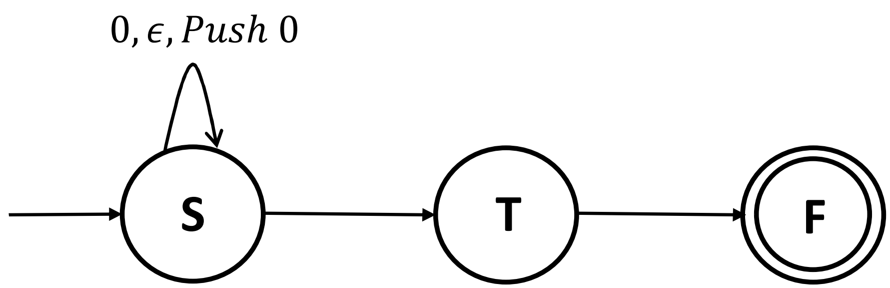


(i) If input symbol = a, and a is popped, then Push a and remain in S.
(ii) Push a on to the stack and remain in S.

Through Steps (i) and (ii), the PDA pushes a onto the stack if it reads a on the input tape and the element at the stack top = a.

Pushdown Automata

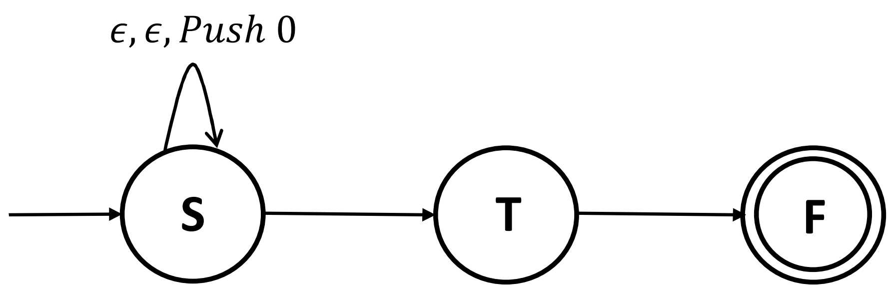
- How to represent a transition in a PDA?



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

Pushdown Automata

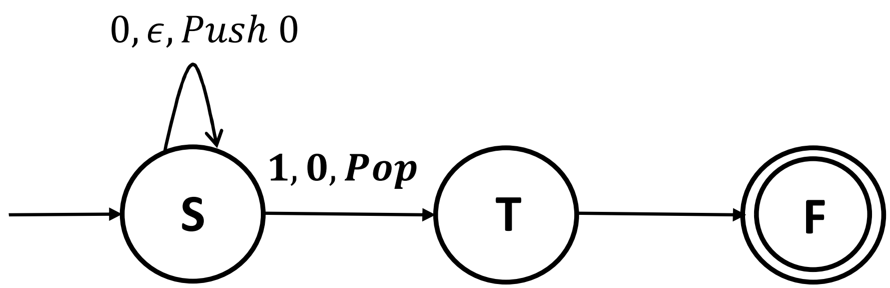
- How to represent a transition in a PDA?



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

Pushdown Automata

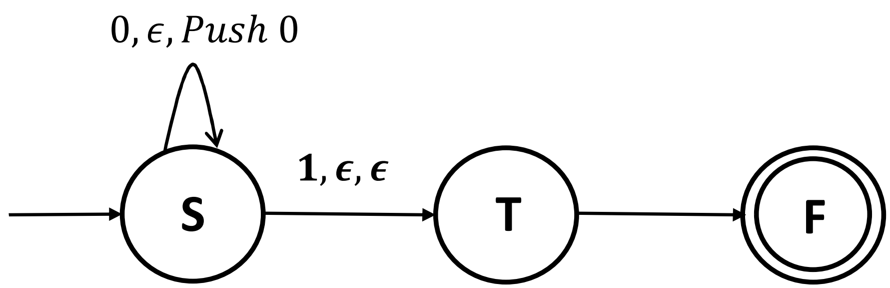
- How to represent a transition in a PDA?



If the input symbol is 1, and the element 0 is popped **(Pop 0)**, then transition from S to T

Pushdown Automata

- How to represent a transition in a PDA?

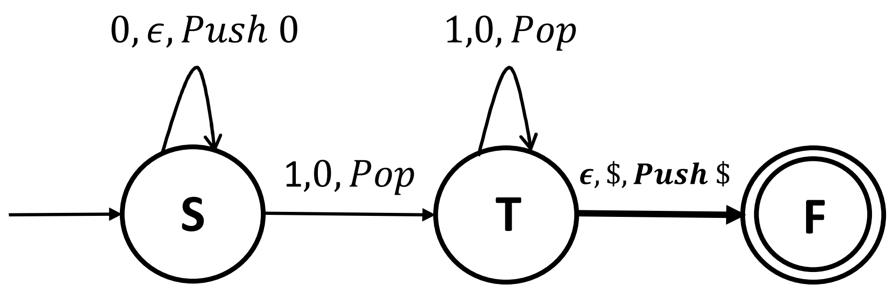


If the input symbol is 1, transition to T by ignoring the stack completely.

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

Pushdown Automata

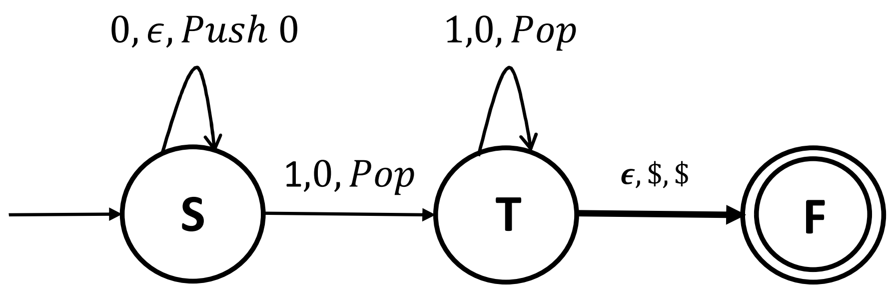
- How to represent a transition in a PDA?



Empty stack: If \$ is popped, push the \$ back onto the stack and transition to F from T, without reading the input

Pushdown Automata

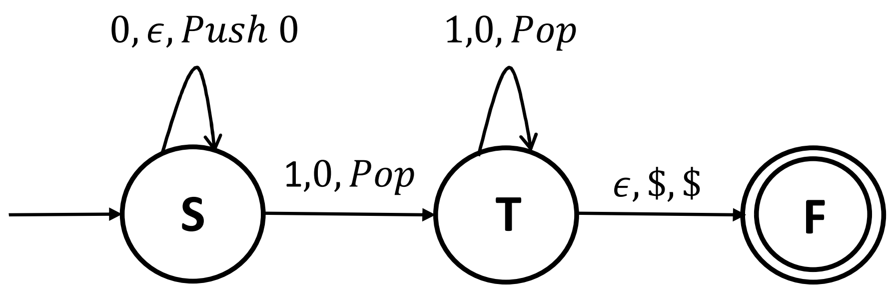
- How to represent a transition in a PDA?



Empty stack: If \$ is popped, push the \$ back onto the stack and transition to F from T, without reading the input

Pushdown Automata

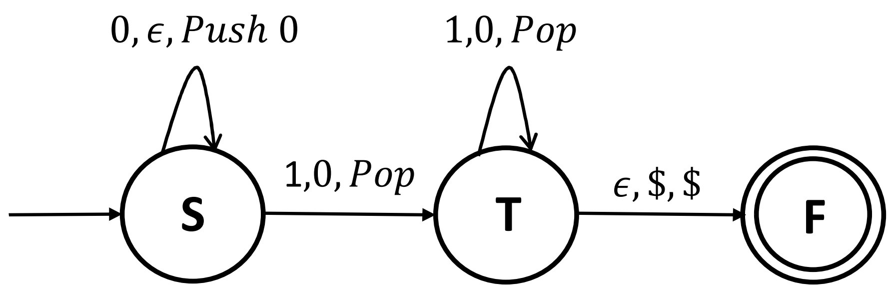
- How to represent a transition in a PDA?



What is the language recognized by this PDA?

Pushdown Automata

- How to represent a transition in a PDA?

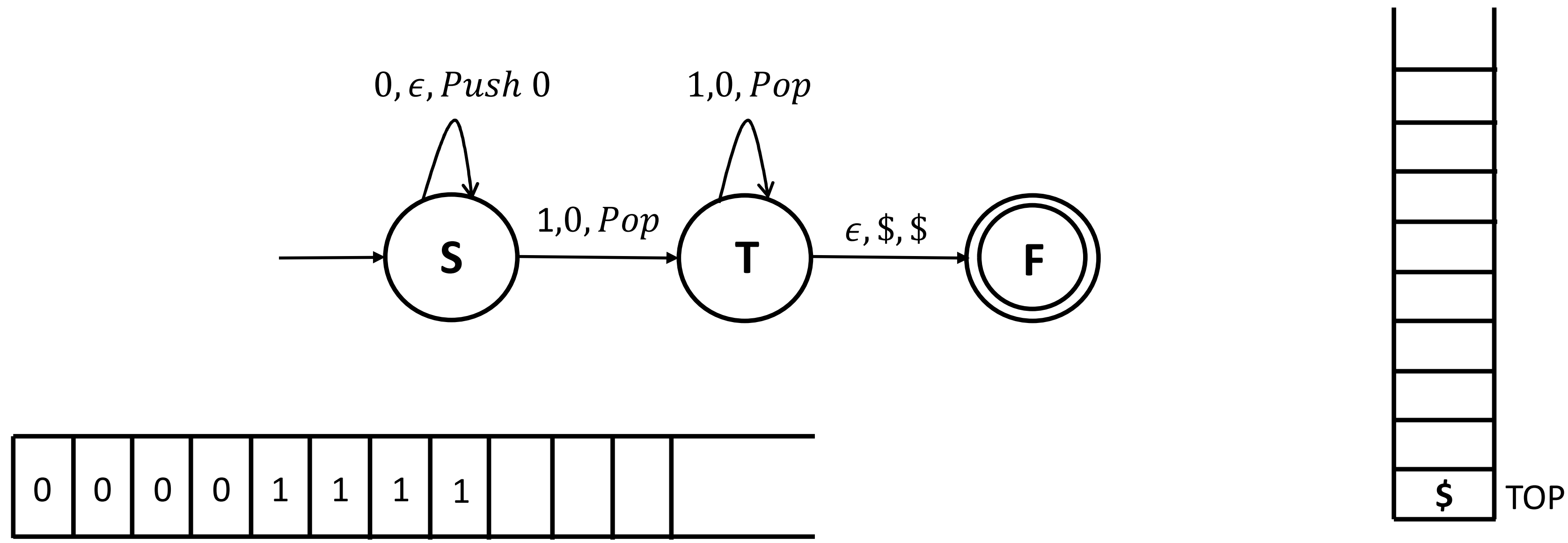


What is the language recognized by this PDA?

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

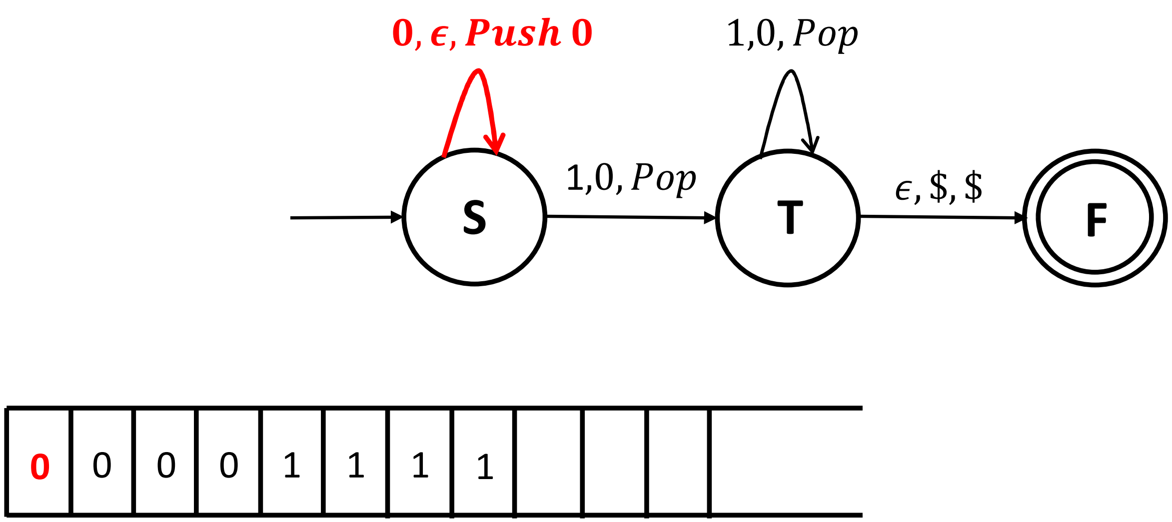
Pushdown Automata

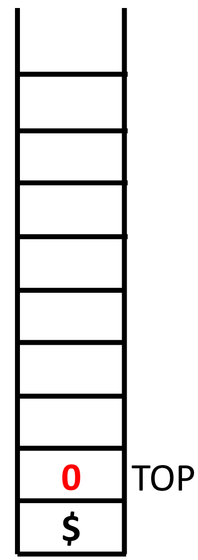
What is the language recognized by this PDA?



Pushdown Automata

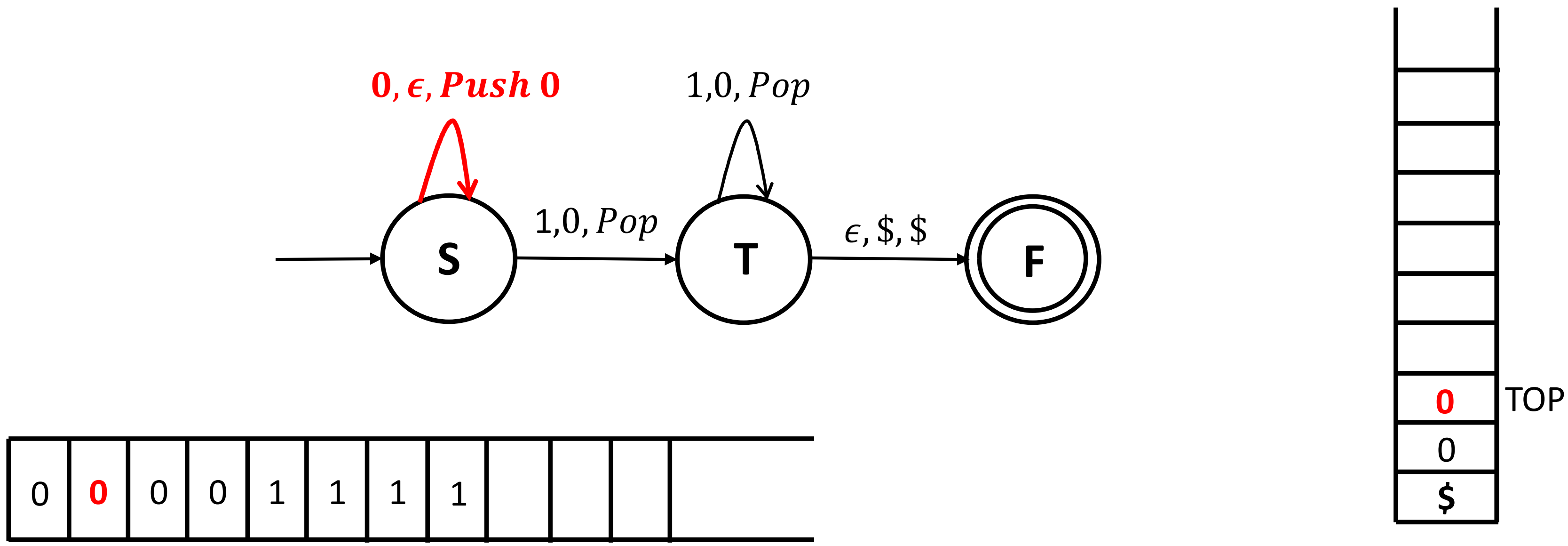
What is the language recognized by this PDA?





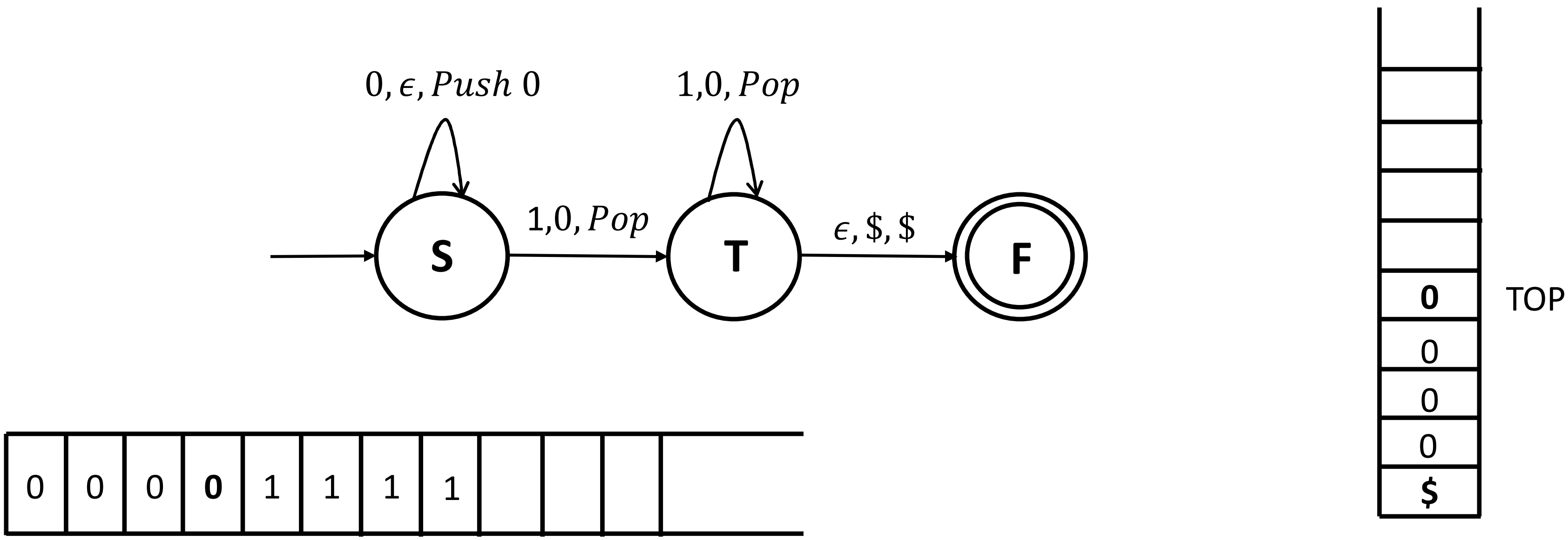
Pushdown Automata

What is the language recognized by this PDA?



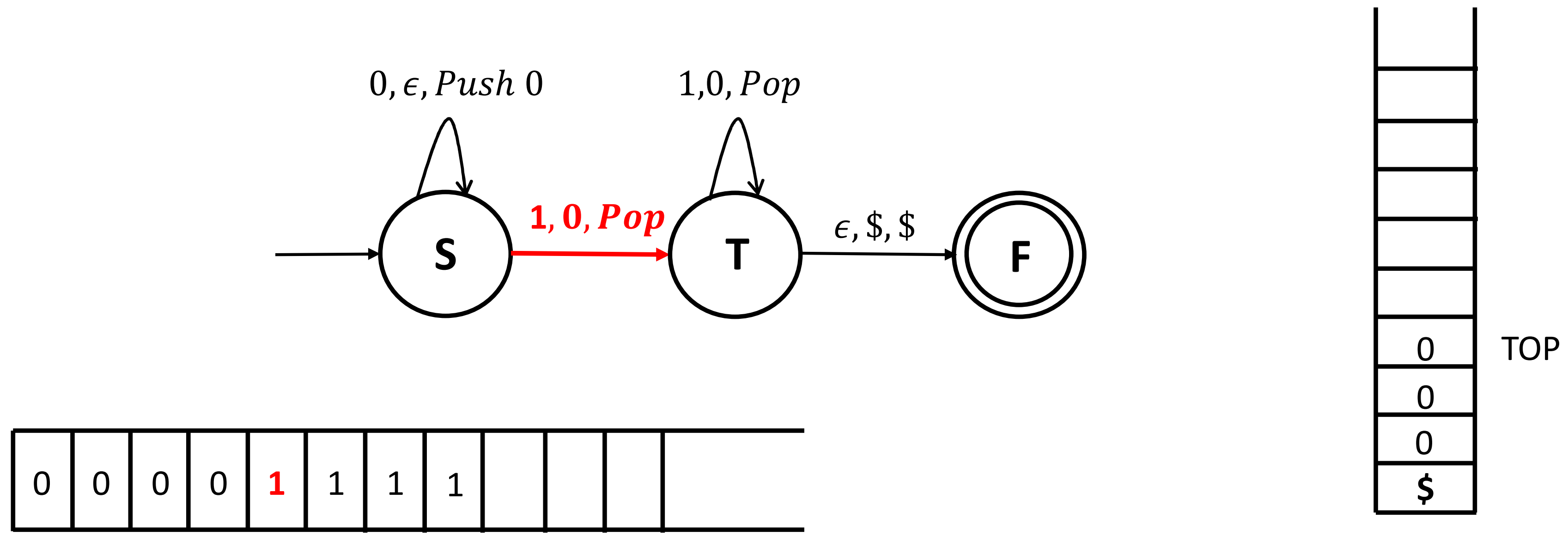
Pushdown Automata

What is the language recognized by this PDA?



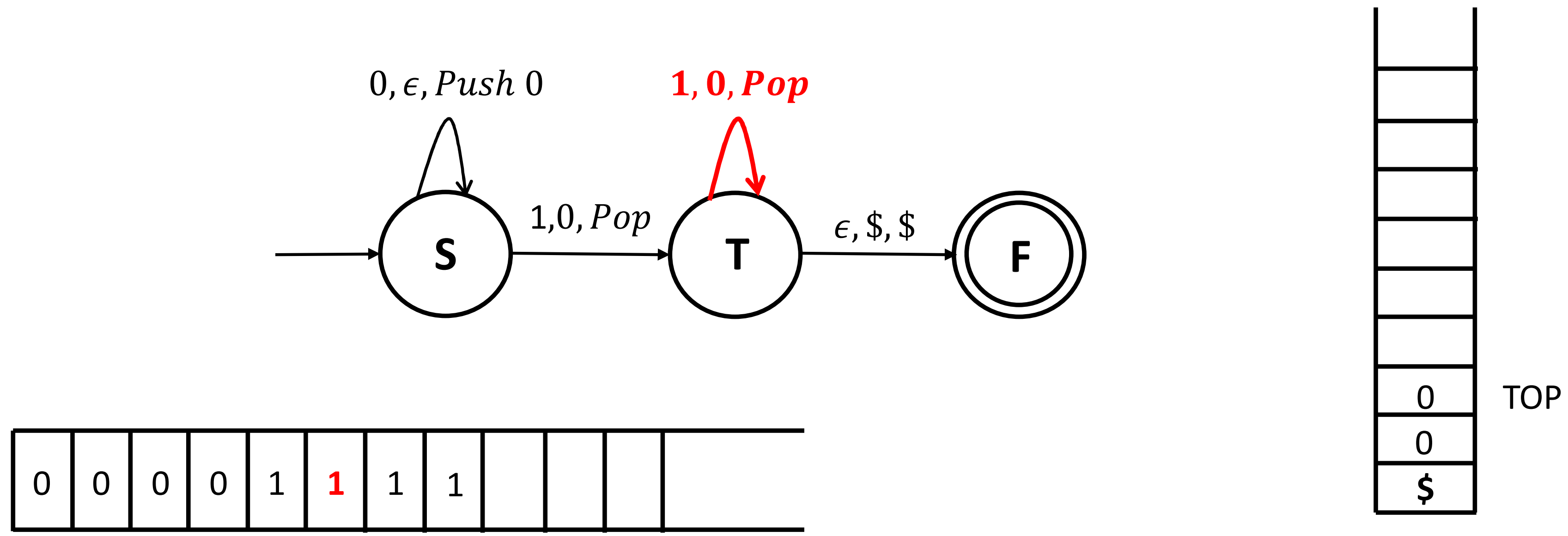
Pushdown Automata

What is the language recognized by this PDA?



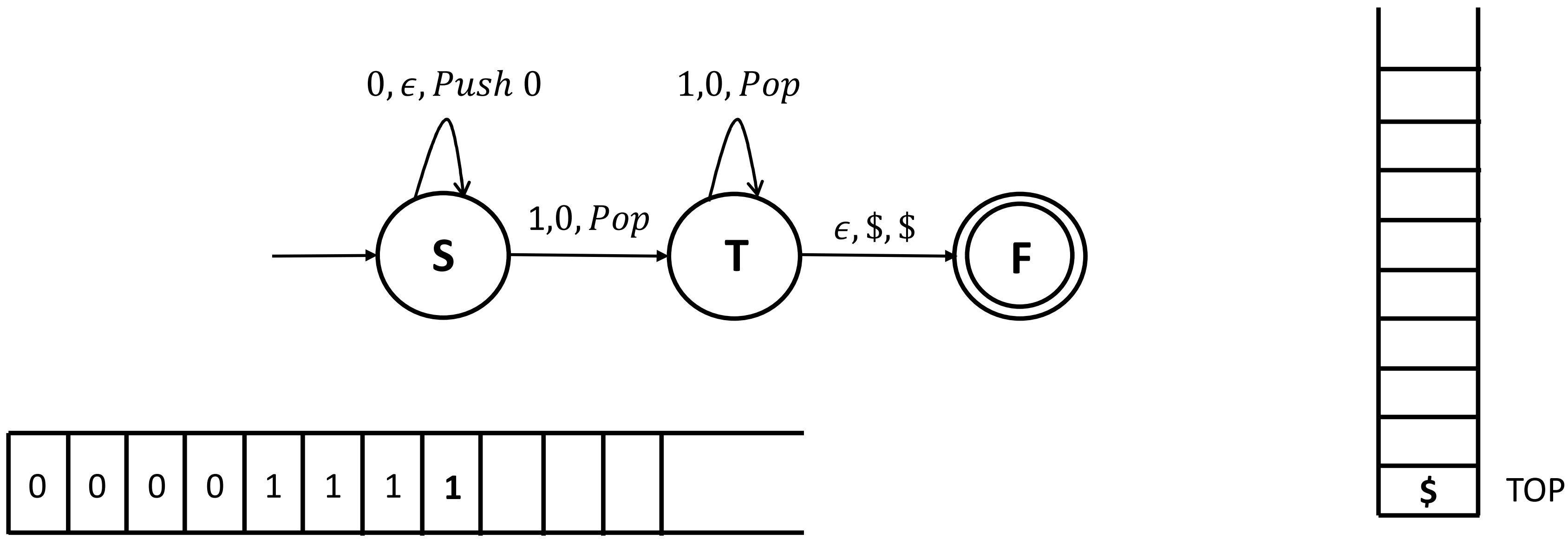
Pushdown Automata

What is the language recognized by this PDA?



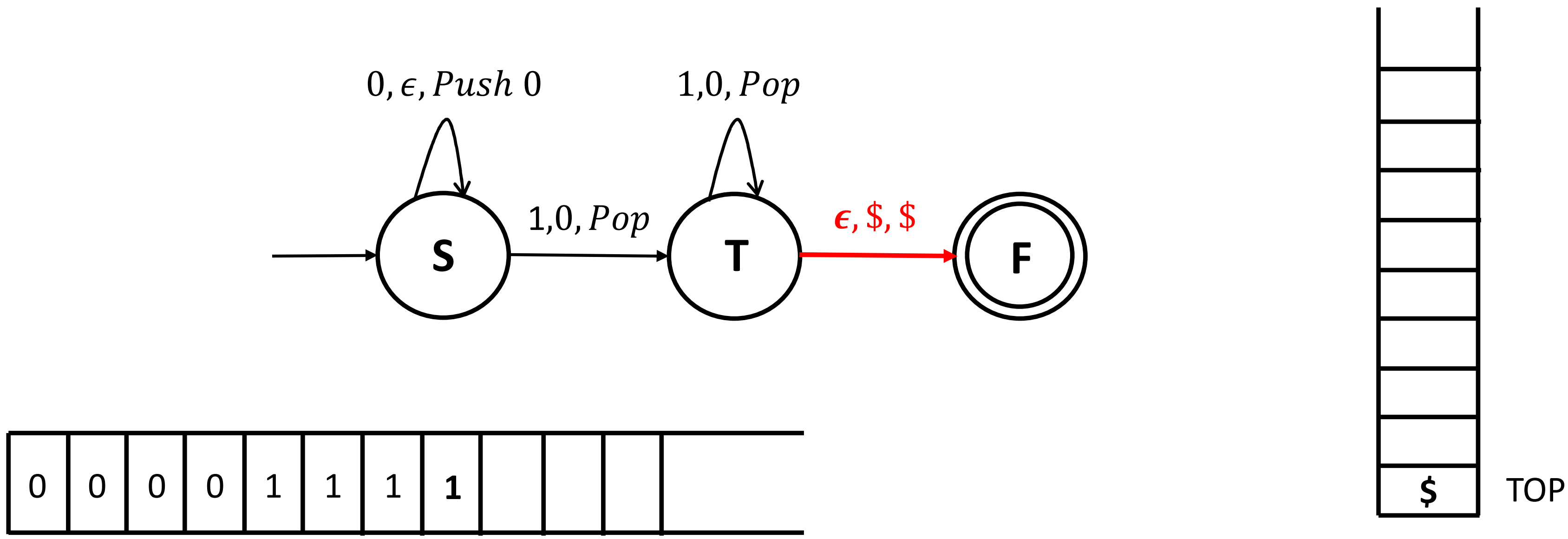
Pushdown Automata

What is the language recognized by this PDA?



Pushdown Automata

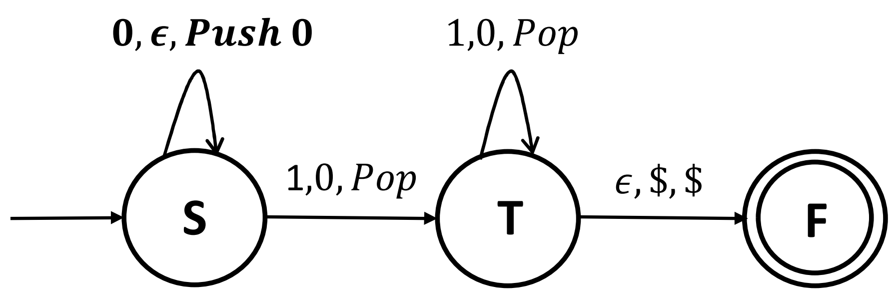
What is the language recognized by this PDA?



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

Pushdown Automata

What is the language recognized by this PDA?

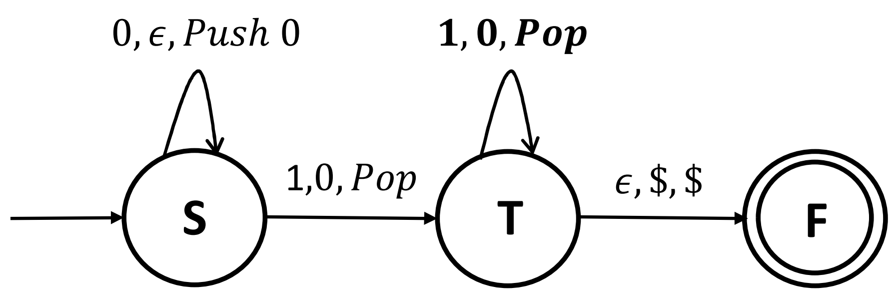


In some references (such as Sipser):

- The transitions of the PDA are labelled as "$\boldsymbol{a}, \boldsymbol{b} \rightarrow \boldsymbol{c}$", implying: If the input symbol read is $\boldsymbol{a}$, and the element at the top of the stack is $\boldsymbol{b}$ ($\boldsymbol{b}$ is popped), then push $\boldsymbol{c}$ on to the Stack.

Pushdown Automata

What is the language recognized by this PDA?

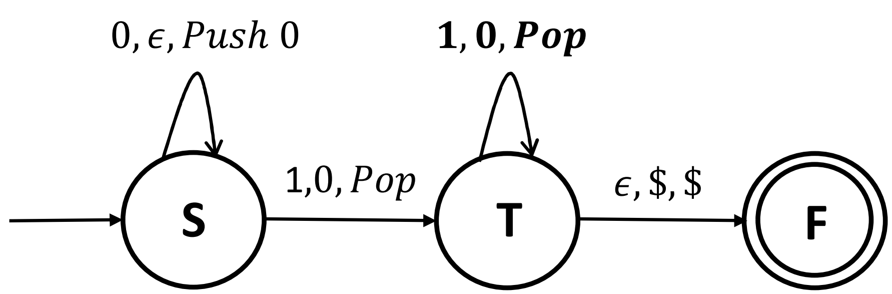


In some references (such as Sipser):

- The transitions of the PDA are labelled as "$\boldsymbol{a}, \boldsymbol{b} \rightarrow \boldsymbol{c}$", implying: If the input symbol read is $\boldsymbol{a}$, then pop $\boldsymbol{b}$ (the element at the top of the stack is $\boldsymbol{b}$) and push $\boldsymbol{c}$ on to the Stack.
- The label "$\boldsymbol{a}, \boldsymbol{b} \rightarrow \boldsymbol{\epsilon}$" implies that if the input symbol is $\boldsymbol{a}$ then **pop b**.

Pushdown Automata

What is the language recognized by this PDA?



In some references (such as Sipser):

- The transitions of the PDA are labelled as "$\boldsymbol{a}, \boldsymbol{b} \rightarrow \boldsymbol{c}$", implying: If the input symbol read is $\boldsymbol{a}$, the element at the top of the stack is $\boldsymbol{b}$, then pop $\boldsymbol{b}$ and push $\boldsymbol{c}$ on to the Stack.
- The label "$\boldsymbol{a}, \boldsymbol{b} \rightarrow \boldsymbol{\epsilon}$" implies that if the input symbol is $\boldsymbol{a}$ and $\boldsymbol{b}$ is **popped**.
- The symbol signifying the bottom of the Stack \$ is pushed at the very beginning.

Pushdown Automata

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

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

Pushdown Automata

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

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

A PDA accepts a string $w \in L$, if there exists a run such that

- It **reaches a final state** when the entire string is read.

OR

- The **stack is empty** when the entire string is read.

Pushdown Automata

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

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

A PDA accepts a string $w \in L$, if there exists a run such that

- It **reaches a final state** when the entire string is read.

OR

- The **stack is empty** when the entire string is read.

These two notions of acceptance are equivalent

Pushdown Automata

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

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

Transition function:

- $\boldsymbol{\delta(q_i, a, b) = (q_j, c)}$: If the input symbol read is a and b is popped, then push c onto the stack and transition from q_i to q_j

Pushdown Automata

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

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

Transition function:

- $\boldsymbol{\delta(q_i, a, b) = (q_j, c)}$: If the input symbol read is a and b is popped, then push c onto the stack and transition from q_i to q_j

- $\boldsymbol{\delta(q_i, a, \epsilon) = (q_j, c)}$:

Pushdown Automata

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

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

Transition function:

- $\boldsymbol{\delta(q_i, a, b) = (q_j, c)}$: If the input symbol read is a and b is popped, then push c onto the stack and transition from q_i to q_j

- $\boldsymbol{\delta(q_i, a, \epsilon) = (q_j, c)}$: 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 $(\mathrm{Q}, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

Transition function:

- $\boldsymbol{\delta(q_i, a, b) = (q_j, c)}$: If the input symbol read is a and b is popped, then push c onto the stack and transition from q_i to q_j

- $\boldsymbol{\delta(q_i, a, \epsilon) = (q_j, c)}$: If the input symbol read is a, then push c onto the stack and transition from q_i to q_j

- $\boldsymbol{\delta(q_i, a, b) = (q_j, \epsilon)}$:

Pushdown Automata

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

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

Transition function:

- $\boldsymbol{\delta(q_i, a, b) = (q_j, c)}$: If the input symbol read is a and b is popped, then push c onto the stack and transition from q_i to q_j

- $\boldsymbol{\delta(q_i, a, \epsilon) = (q_j, c)}$: If the input symbol read is a, then push c onto the stack and transition from q_i to q_j

- $\boldsymbol{\delta(q_i, a, b) = (q_j, \epsilon)}$: If the input symbol read is a, and the stack top = b (b is popped) then transition from q_i to q_j

- $\boldsymbol{\delta(q_i, \epsilon, \$) = (q_j, \$)}$:

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.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

Transition function:

- $\boldsymbol{\delta(q_i, a, b) = (q_j, c)}$: If the input symbol read is a and b is popped, then push c onto the stack and transition from q_i to q_j

- $\boldsymbol{\delta(q_i, a, \epsilon) = (q_j, c)}$: If the input symbol read is a, then push c onto the stack and transition from q_i to q_j

- $\boldsymbol{\delta(q_i, a, b) = (q_j, \epsilon)}$: If the input symbol read is a, and the stack top = b, then pop b and transition from q_i to q_j

- $\boldsymbol{\delta(q_i, \epsilon, \$) = (q_j, \$)}$: 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.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

Transition function:

- $\boldsymbol{\delta(q_i, a, b) = (q_j, c)}$: If the input symbol read is a and b is popped, then push c onto the stack and transition from q_i to q_j

- If the input symbol read is a and a is popped, then Push a and remain at q_i : ?

Pushdown Automata

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

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** $[\ \Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}\ ]$
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

Transition function:

- $\boldsymbol{\delta(q_i, a, b) = (q_j, c)}$: If the input symbol read is a and b is popped, then push c onto the stack and transition from q_i to q_j

- If the input symbol read is a and a is popped, then Push a and remain at q_i : $\boldsymbol{\delta(q_i, a, a) = (q_i, a)}$

Pushdown Automata

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

- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

- The Language of the PDA P is the set of strings the PDA accepts, i.e.

$$L = \{w | P \text{ accepts } w\}$$

There exists an accepting run for w on P

- If $\mathcal{L}(P) = L$, then the PDA P recognizes L

Pushdown Automata

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

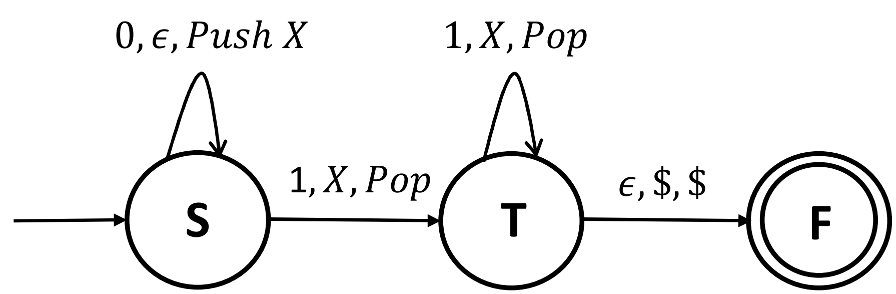
- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

- The Language of the PDA P is the set of strings the PDA accepts, i.e.

$$L = \{w | P \text{ accepts } w\}$$

- 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

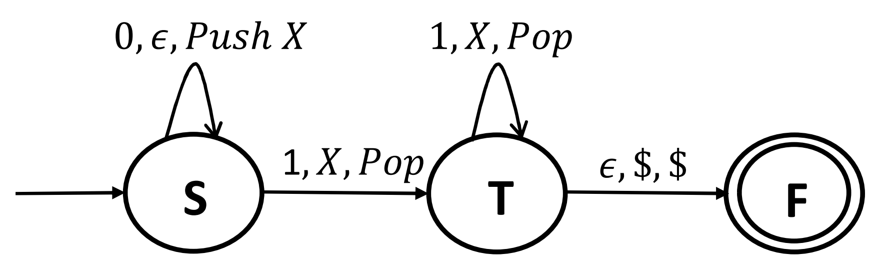
- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

- The Language of the PDA P is the set of strings the PDA accepts, i.e.

$$L = \{w | P \text{ accepts } w\}$$

- 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

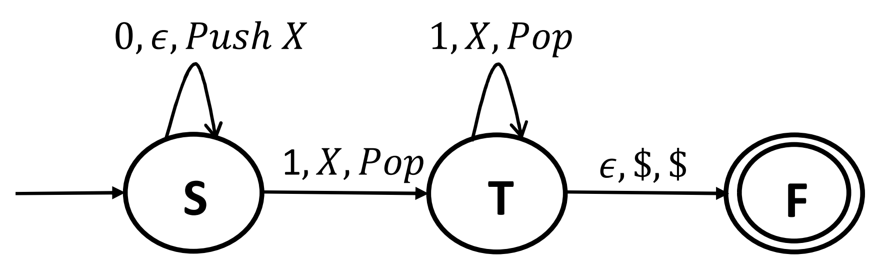
- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

- The Language of the PDA P is the set of strings the PDA accepts, i.e.

$$L = \{w | P \text{ accepts } w\}$$

- If $\mathcal{L}(P) = L$, then the PDA P recognizes L

- Stack alphabet **can be different** from the input alphabet



$$\delta(S, 0, \epsilon) = (S, X)$$
$$\delta(S, 1, X) = (T, \epsilon)$$

Pushdown Automata

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

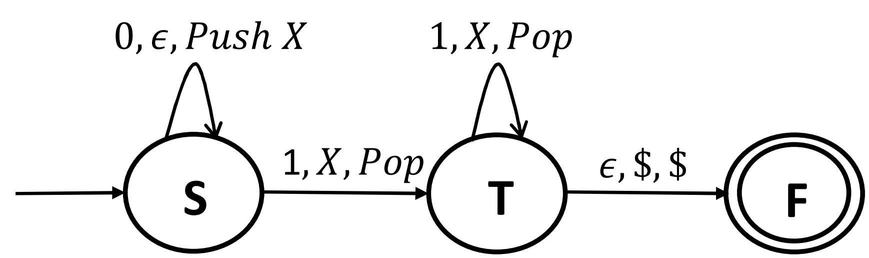
- Q is a finite set called the ***states.***
- Σ is the set of input ***alphabets.***
- Γ is the set of ***Stack alphabets***
- $\delta: Q \times \Sigma_\epsilon \times \Gamma_\epsilon \mapsto \mathcal{P}(Q \times \Gamma_\epsilon)$ is the ***transition function*** [$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ and $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$]
- $q_0 \in Q$ is the ***start state***.
- $F \subseteq Q$ is the set of ***accepting states***.

- The Language of the PDA P is the set of strings the PDA accepts, i.e.

$$L = \{w | P \text{ accepts } w\}$$

- If $\mathcal{L}(P) = L$, then the PDA P recognizes L

- Stack alphabet **can be different** from the input alphabet



$$\delta(S, 0, \epsilon) = (S, X)$$
$$\delta(S, 1, X) = (T, \epsilon)$$
$$\delta(T, 1, X) = (T, \epsilon)$$
$$\delta(T, \epsilon, \$) = (F, \$)$$

Thank You!