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

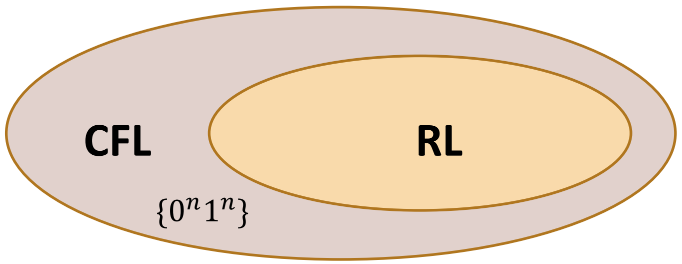
HYDERABAD

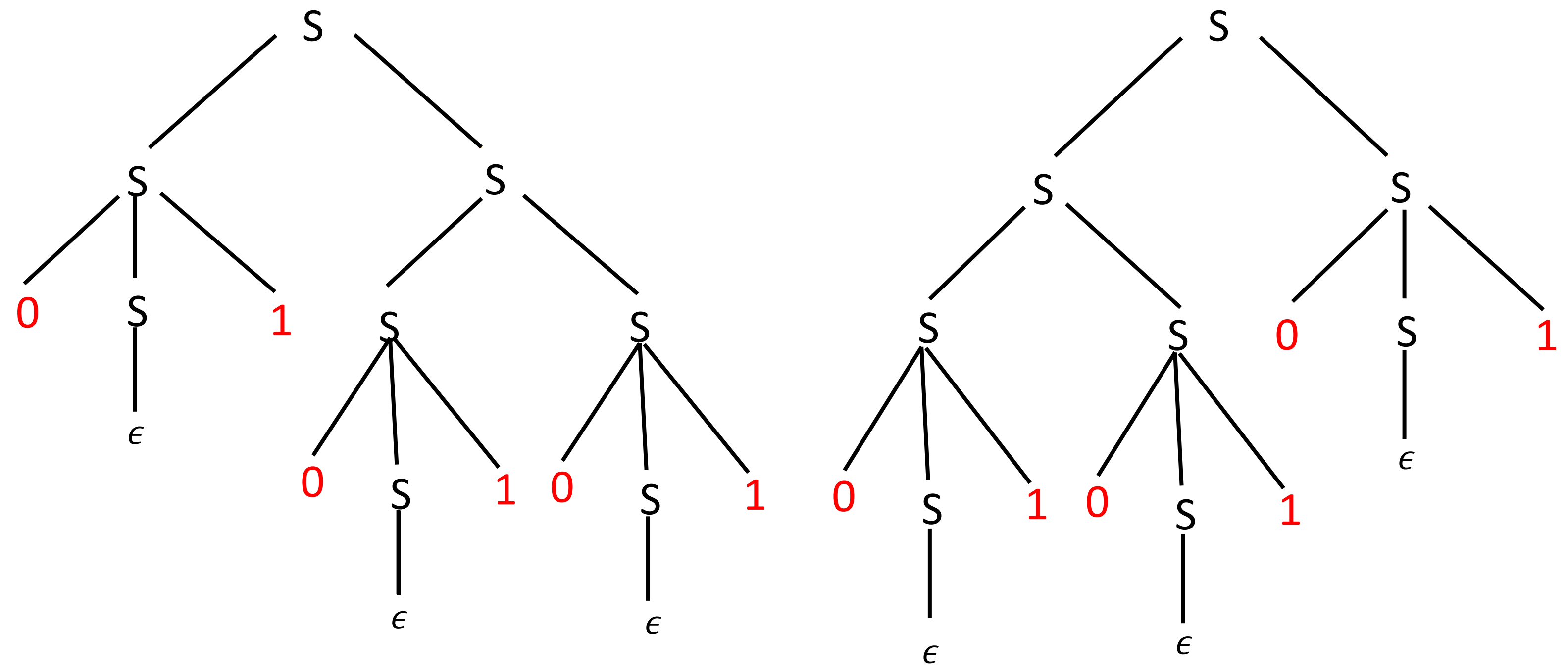
Quick Recap

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

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

$$\boldsymbol{Var \rightarrow Var\,Var}$$
$$\boldsymbol{Var \rightarrow ter}$$
$$\boldsymbol{Start\,Var \rightarrow \epsilon}$$

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

$$\boldsymbol{Var \rightarrow Var\,Var}$$
$$\boldsymbol{Var \rightarrow ter}$$
$$\boldsymbol{Start\,Var \rightarrow \epsilon}$$

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

$$\boldsymbol{Var \rightarrow Var\,Var}$$
$$\boldsymbol{Var \rightarrow ter}$$
$$\boldsymbol{Start\,Var \rightarrow \epsilon}$$

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 $\boldsymbol{G}$ generates $\boldsymbol{w}$**.

- 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

$$\boldsymbol{Var \rightarrow Var\, Var}$$
$$\boldsymbol{Var \rightarrow ter}$$
$$\boldsymbol{Start\, Var \rightarrow \epsilon}$$

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 $\boldsymbol{G}$ generates $\boldsymbol{w}$**. This problem appears to be **undecidable**.

Chomsky Normal Form

A CFG G is in **CNF** if every rule of G is of the form

$$\boldsymbol{Var \to Var\, Var}$$
$$\boldsymbol{Var \to ter}$$
$$\boldsymbol{Start\, Var \to \epsilon}$$

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 $\boldsymbol{G}$ generates $\boldsymbol{w}$**.

- 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 $\boldsymbol{2n-1}$ 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

$$\boldsymbol{Var \rightarrow Var\,Var}$$
$$\boldsymbol{Var \rightarrow ter}$$
$$\boldsymbol{Start\,Var \rightarrow \epsilon}$$

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 $\boldsymbol{G}$ generates $\boldsymbol{w}$**.

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

$$\boldsymbol{Var \rightarrow Var\,Var}$$
$$\boldsymbol{Var \rightarrow ter}$$
$$\boldsymbol{Start\,Var \rightarrow \epsilon}$$

where Var can be any variable, including the Start Variable, $Start\,Var$.

1) **A CFG in Chomsky Normal Form has derivations of $\boldsymbol{2n-1}$ steps for generating strings $\boldsymbol{w \in L(G)}$ of length $\boldsymbol{n}$.**

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 $\boldsymbol{2n-1}$ steps for generating strings $\boldsymbol{w} \in \boldsymbol{L(G)}$ of length $\boldsymbol{n}$.**

Proof: Note that any CFG in CNF can be written as:

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

We will prove this by **induction.**

(Basic step) Let $|w| = 1$. Then **one** application of the second rule would suffice. So any derivation of w would need $2|w| - 1 = 1$ step.

(Inductive hypothesis) Assume the statement of the theorem to be true for any string of length at most k where $k \geq 1$. Now we shall show that it holds for any $w \in L(G)$ such that $|w| = k + 1$.

Chomsky Normal Form

Prove that **a CFG in Chomsky Normal Form has derivations of $2n - 1$ steps for generating strings $w \in L(G)$ of length n.**

Proof: Note that any CFG in CNF can be written as:

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

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 $A \to BC$. So $\boldsymbol{w = xy}$**, where** $\boldsymbol{B \overset{*}{\Rightarrow} x}$**,** $\boldsymbol{|x| > 0}$ **and** $\boldsymbol{C \overset{*}{\Rightarrow} y}$, $\boldsymbol{|y| > 0}$. But since $|x|, |y| \leq k$, and we have that by the inductive hypothesis: (i) number of steps in the derivation $\boldsymbol{B \overset{*}{\Rightarrow} x}$ **is** $\boldsymbol{2|x| - 1}$ and (ii) number of steps in the derivation $\boldsymbol{C \overset{*}{\Rightarrow} y}$ **is** $\boldsymbol{2|y| - 1}$. So the number of steps in the derivation of w is

$$\boldsymbol{1 + (2|x| - 1) + (2|y| - 1) = 2(|x| + |y|) - 1 = 2|w| - 1 = 2(k + 1) - 1}.$$

Chomsky Normal Form

A CFG G is in **CNF** if every rule of G is of the form

$$\boldsymbol{Var \rightarrow Var\, Var}$$
$$\boldsymbol{Var \rightarrow ter}$$
$$\boldsymbol{Start\, Var \rightarrow \epsilon}$$

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 $S' \to S$

2. Remove ϵ rules of the form $A \to \epsilon$
   - Remove nullable symbols/rules

3. Remove unit (short) rules of the form $A \to B$
   - Remove useless symbols/rules

4. Remove long rules of the form $A \to u_1 u_2 \cdots u_k$
   - 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 $A \to \epsilon$**

We remove the rule $A \to \epsilon$. 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 ϵ 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 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 ϵ rules of the form $A \to \epsilon$ (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 $S' \rightarrow S$

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

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

We **remove the rule $\boldsymbol{A \rightarrow B}$** and **whenever a rule $\boldsymbol{B \rightarrow u}$ appears** (u is a string of terminals and variables), we **add a new rule $\boldsymbol{A \rightarrow u}$**, 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 $\boldsymbol{A \rightarrow B}$ (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.)

E.g.:

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

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.:

$$\begin{gathered} S \to A|11 \\ A \to B|1 \\ B \to S|0 \end{gathered}$$

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

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

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

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 $S' \to S$

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

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

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

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

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

- We 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

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 $\boldsymbol{S' \rightarrow S}$

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)**.**

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.)

Remove long rules of the form $\boldsymbol{A \rightarrow u_1 u_2 \cdots u_k}$ (Replace $A \rightarrow u_1 u_2 \cdots u_k$, $(k \geq 3)$ with the rules $A \rightarrow u_1 A_1$, $A_1 \rightarrow u_2 A_2, \cdots, A_{k-2} \rightarrow u_{k-1} u_k$; 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:

$\boldsymbol{A \rightarrow BC}$ [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 ($\boldsymbol{B \rightarrow \epsilon}$)

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

2b. Remove ϵ rules (A→ 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 $\boldsymbol{S' \to S}$

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

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→ $\boldsymbol{S}$

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

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

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

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.

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

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 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.

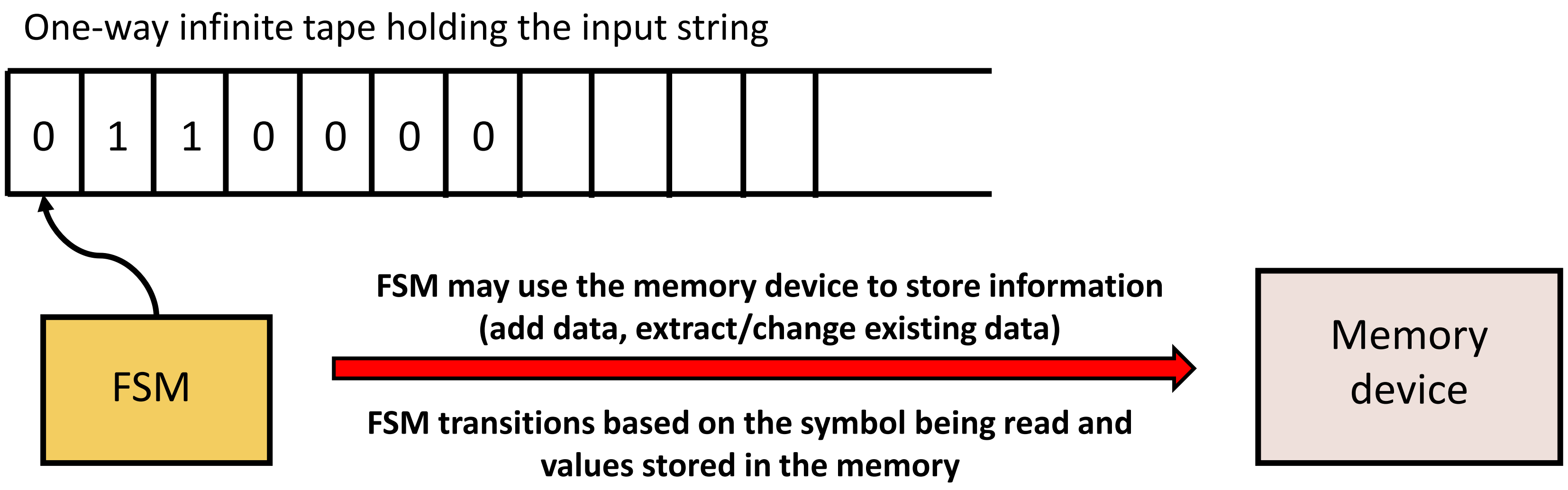
E.g.: $\{0^n 1^n, n \in \mathbb{N}\}$

Pushdown Automata

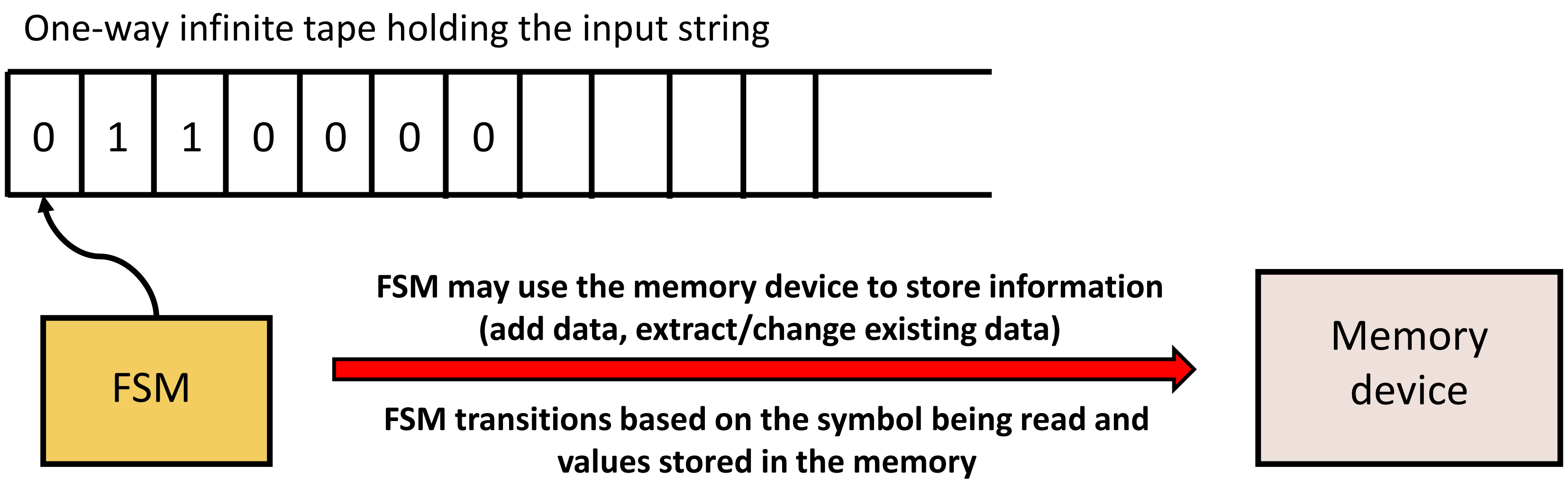
Intuition to build an Automata for CFL

- **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



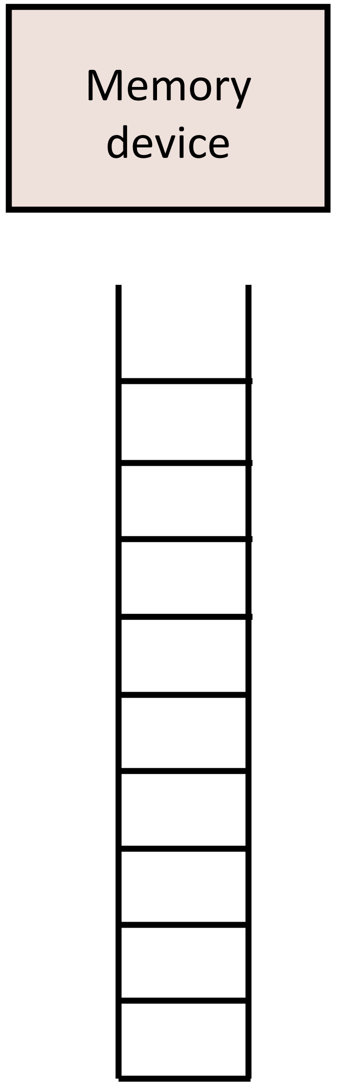
The memory device

- Simple memory device with unbounded memory.

Pushdown Automata

The memory device

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



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

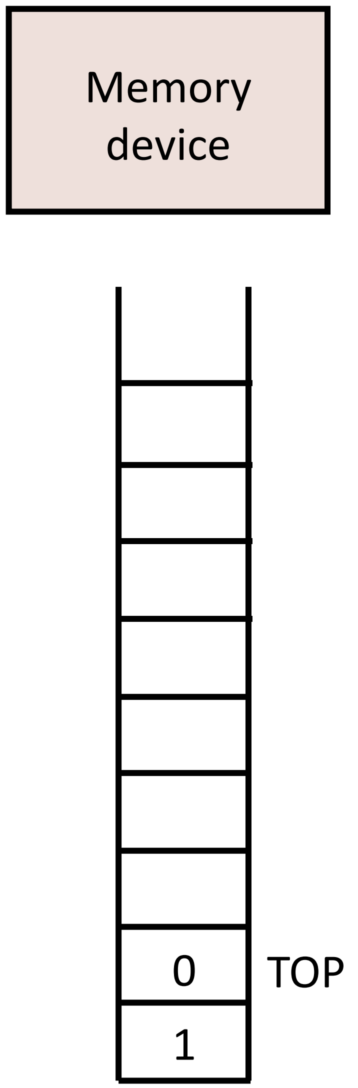
- New symbols can be **pushed** in to the STACK.

  E.g: If TOP of STACK = 0, PUSH 1

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

  TOP = TOP + 1

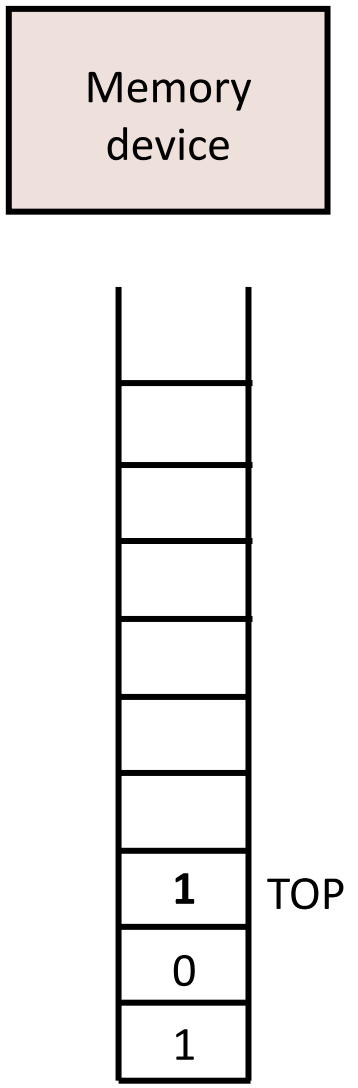
- The size of the stack keeps growing.



Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, 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.

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

PUSH

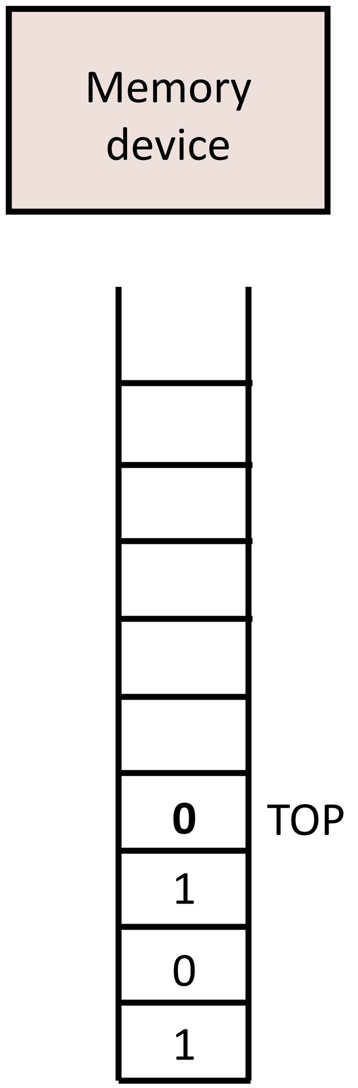
- New symbols can be **pushed** in to the STACK.

  E.g: **PUSH 0 (irrespective of the value of Top)**

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

  TOP = TOP + 1

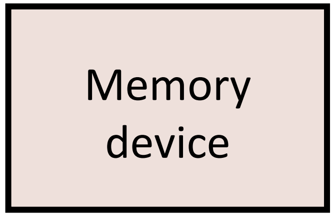
- The size of the stack keeps growing.

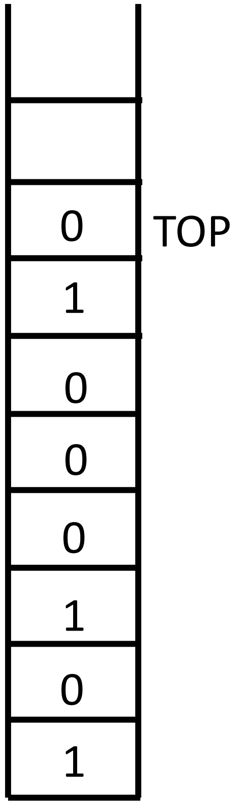


Pushdown Automata

The memory device

- Simple memory device with unbounded memory.
- Consider a **STACK**
- At any stage, 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.

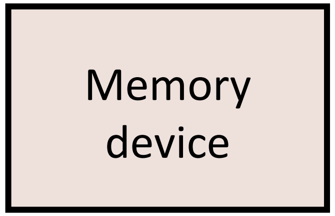
  TOP = TOP + 1

- The size of the stack keeps growing.

Pushdown Automata

The memory device

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



POP

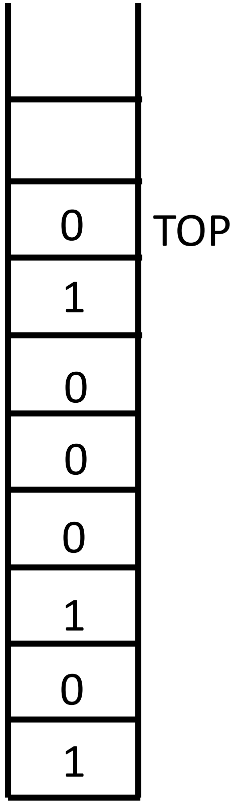
- The element from the TOP of the stack can be **popped** out

E.g.: **POP 0**

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

TOP = TOP - 1

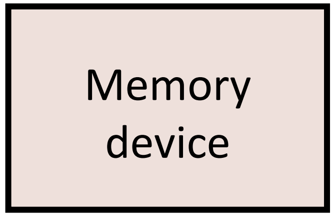
- Successive **POP** operations shrink the stack size. Elements can be popped until EMPTY.
- **Last In First Out (LIFO)**: The last element that was pushed is the first to be popped out



Pushdown Automata

The memory device

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



POP

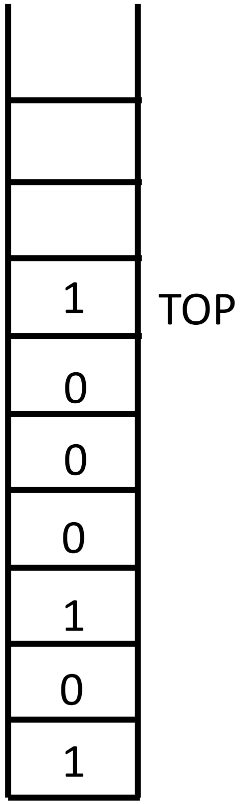
- The element from the TOP of the stack can be **popped** out

E.g.: **POP 0**

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

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

POP

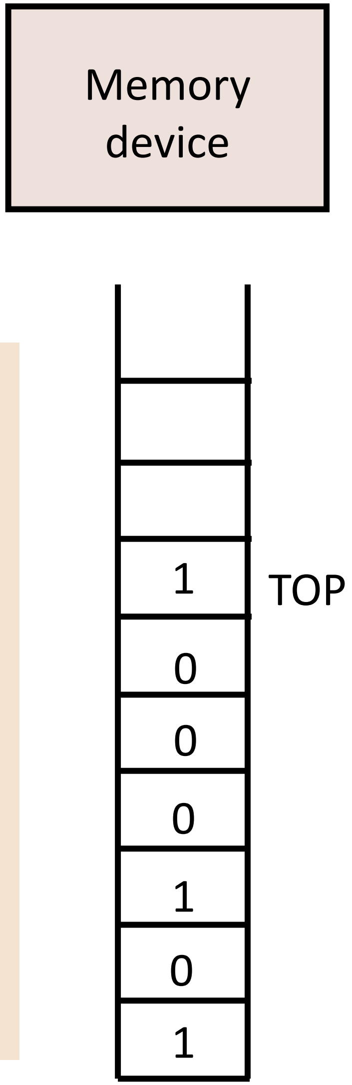
- The element from the TOP of the stack can be **popped** out

E.g.: **POP 1**

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

TOP = TOP - 1

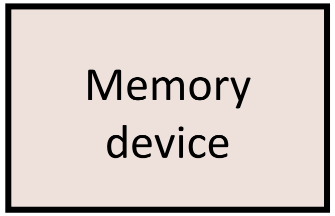
- Successive **POP** operations shrink the stack size. Elements can be popped until EMPTY.
- **Last In First Out (LIFO)**: The last element that was pushed is the first to be popped out



Pushdown Automata

The memory device

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



POP

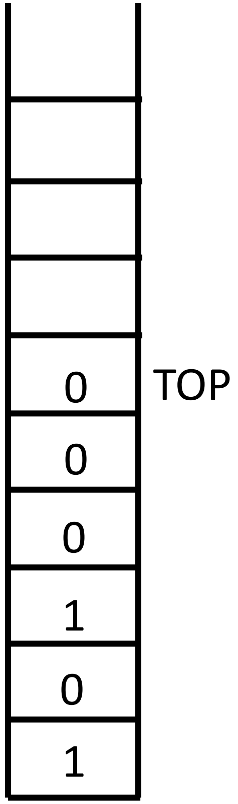
- The element from the TOP of the stack can be **popped** out

E.g.: **POP 1**

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

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

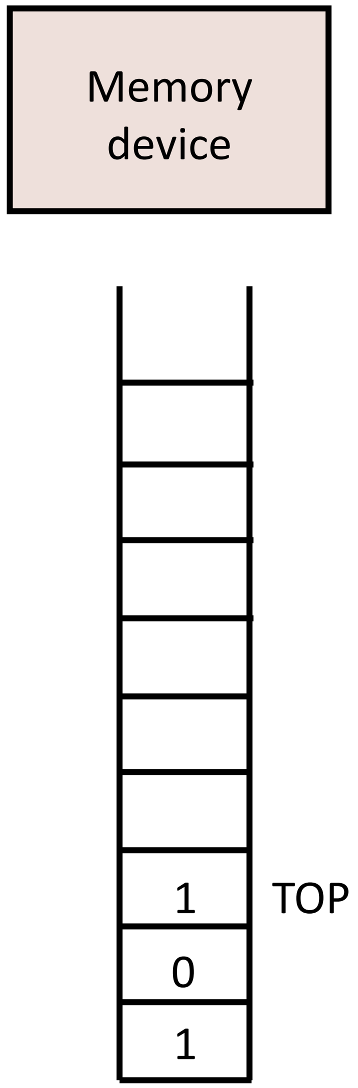
The memory device

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

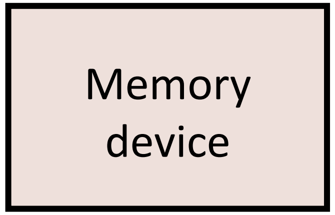
POP

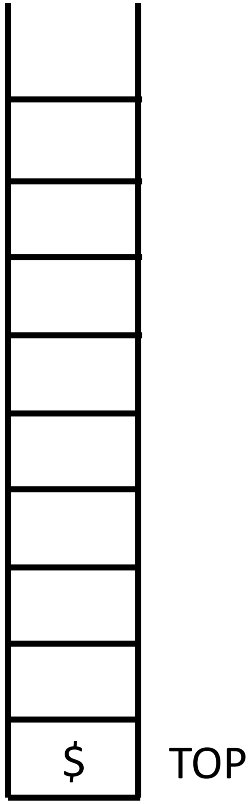
- The element from the TOP of the stack can be **popped** out.
- TOP = TOP - 1
- Elements can be popped until STACK is EMPTY.

- 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 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:

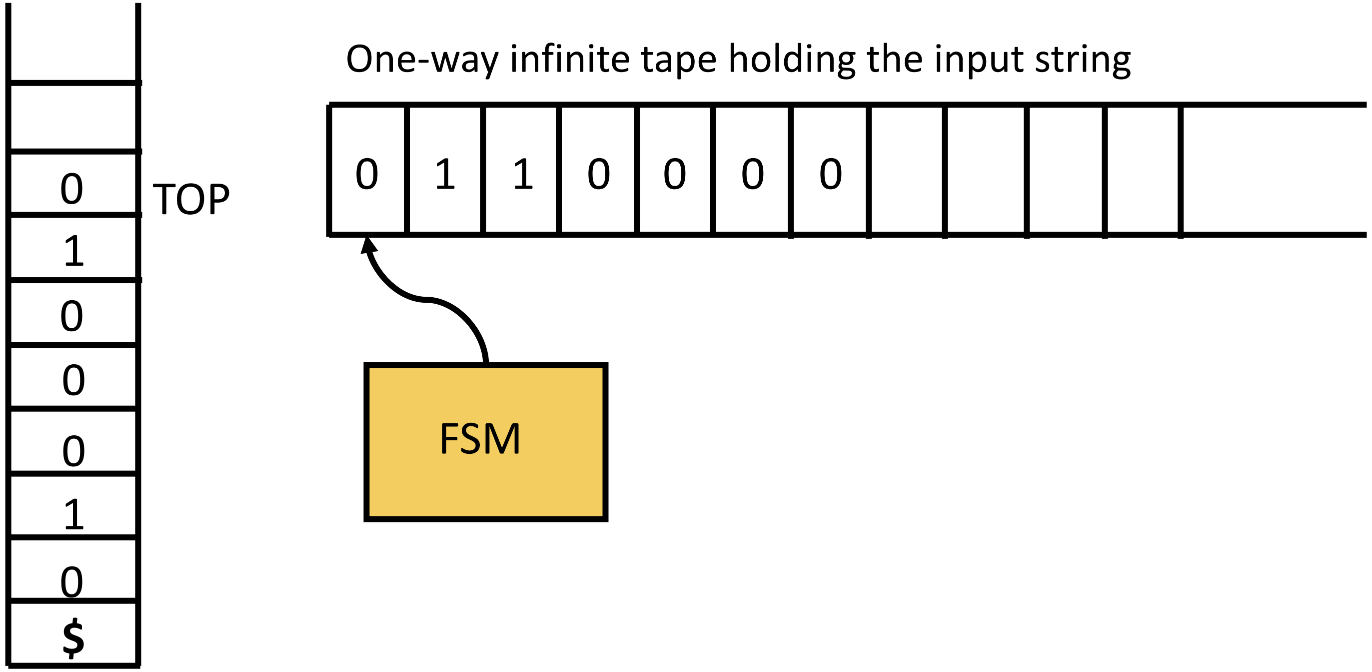
PUSH

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

POP

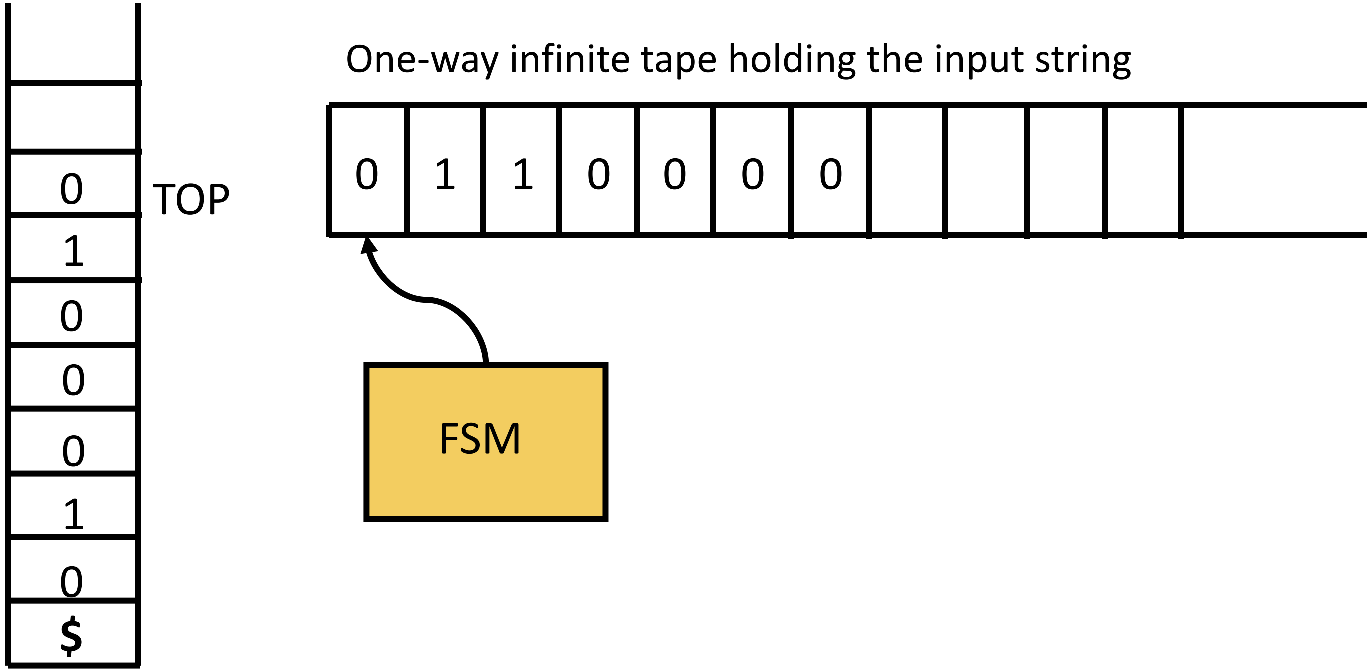
- The element from the TOP of the stack can be **popped** out.
- TOP = TOP - 1
- Elements can be popped until STACK is EMPTY.

Pushdown Automata



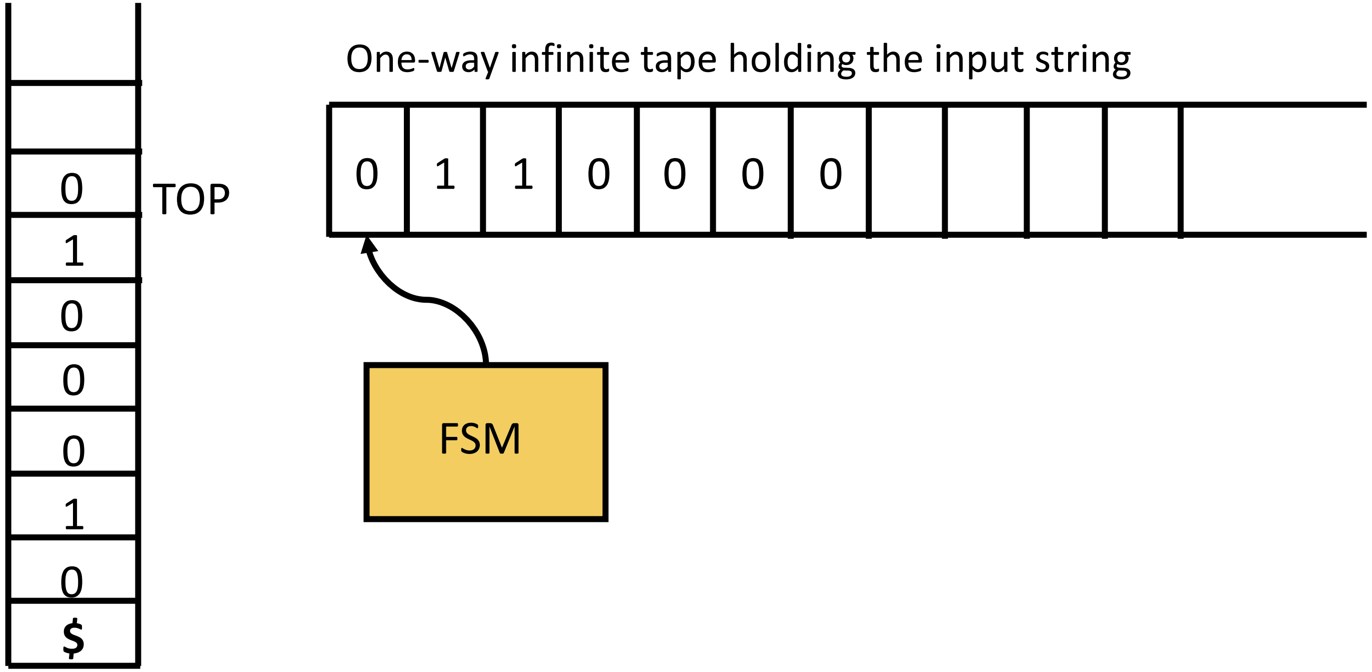
- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.
- The FSM:

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 & TOP = 0, transition from i to j)

Pushdown Automata

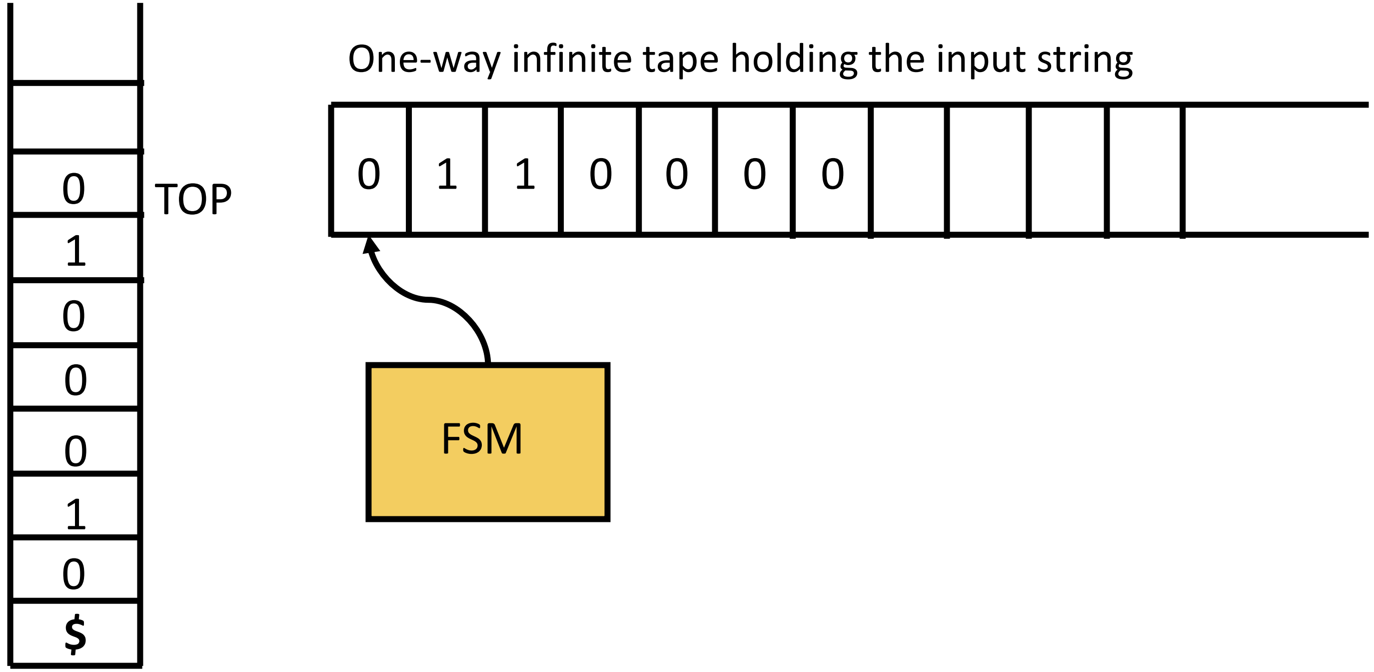


- A Pushdown Automata (PDA) is a finite automaton that has access to a stack.

- 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).

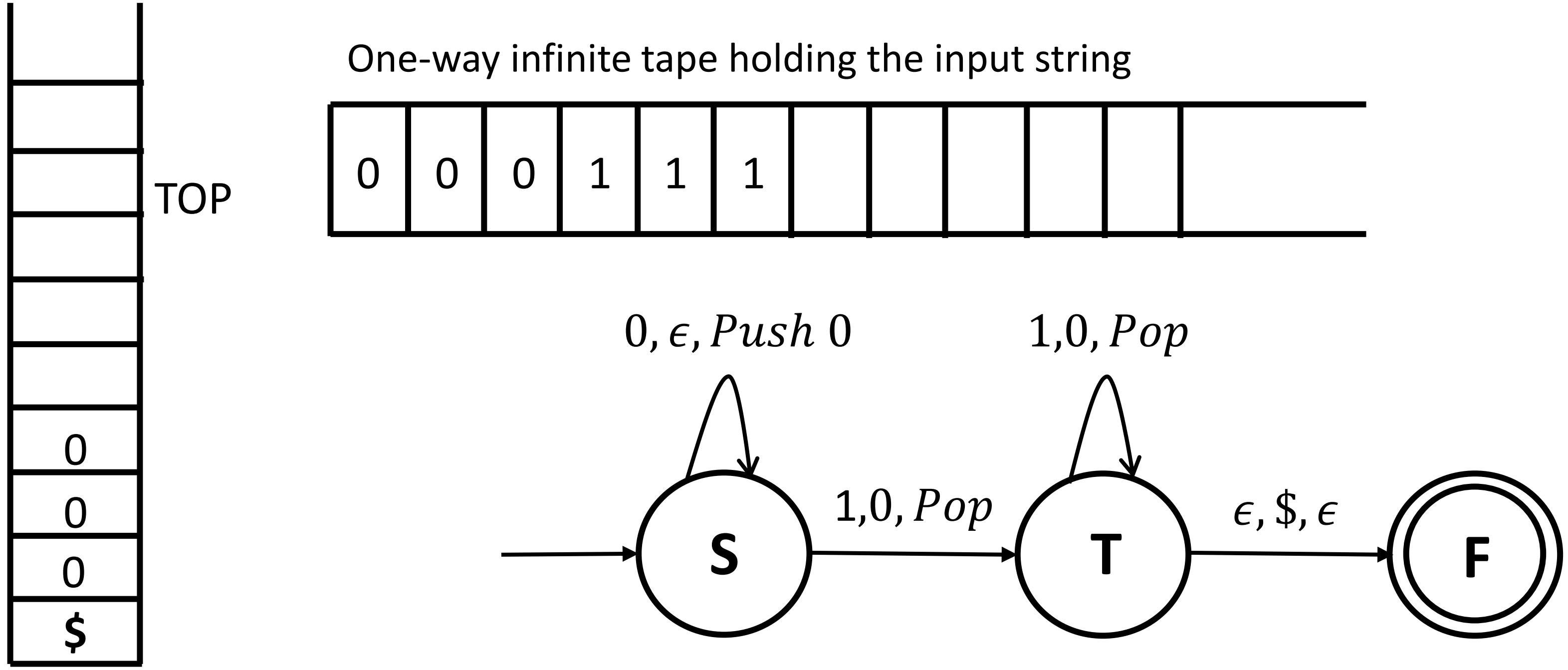
Pushdown Automata



- 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!