Unit - 2

Context Free Grammers :- is a set of recursive rewritting of rules (or productions) used to generate patterns of strings.

Components/Tuples of ~~CFG~~ CFG :- (V, Σ, R, S)

~~(V, T, P, S)~~ ~~Σ~~. where

$V \rightarrow$ is an alphabet

$\Sigma \rightarrow$ set of terminal & is subset of V.

$R \rightarrow$ The set of rules is finite subset of

$$(V - \Sigma) * V^*$$

$S \rightarrow$ start Symbol is an element of $(V - \Sigma)$. The members of S are called non terminals

example:- Let $G = \{V, \Sigma, R, S\}$ where V, Σ, R are as follows

$V = \{T, F, E\}$

$\Sigma = \{+, *, (,), id\}$

$R = \{E \rightarrow E+T,\ E \rightarrow T,\ T \rightarrow T*F,\ T \rightarrow F,\ F \rightarrow (E),\ F \rightarrow id\}$

$E \rightarrow E+T$ --- (i)
$E \rightarrow T$ --- ii
$T \rightarrow T*F$ --- (iii)
$T \rightarrow F$ --- (iv)
$F \rightarrow (E)$ --- (v)
$F \rightarrow id$ --- (vi)

- The Symbol E, T, F are expression, term & factor.
- Now Generate $\rightarrow (id * id + id) * (id + id)$

Ans:-

$E \to T$ --- (ii)
$\to T*F$ --- (iii)
$\to T*(E)$ --- (v)
$\to T*(E+T)$ --- (i)
$\to T*(T+T)$ --- (ii)
$\to T*(F+T)$ --- (iv)
$\to T*(id+T)$ --- (vi)
$\to T*(id+F)$ --- (iv)
$\to T*(id+id)$ --- (vi)
$\to F*(id+id)$ --- (iv)

$E \to (E)*(id+id)$ --- (v)
$\to (E+T)*(id+id)$ --- (i)
$\to (E+F)*(id+id)$ --- (iv)
$\to (E+id)*(id+id)$ --- (vi)
$\to (T+id)*(id+id)$ --- (ii)
$\to (T*F+id)*(id+id)$ --- (iii)
$\to (T*id+id)*(id+id)$ --- (vi)
$\to (F*id+id)*(id+id)$ --- (iv)
$\to (id*id+id)*(id+id)$ --- (vi) ✓

Que. - Construct a CFG that generate Language

$L = \{ w c w^R / w \in \{a,b\}^* \}$

$S \to AcA$ — (1)
$S \to BcB$ — (2)
$A \to aB/Ba$ — (3)
$B \to bA/Ab$ — (4)
$B \to \wedge$ — (5)
$A \to \wedge$ — (6)

eg:- $w = aab\,c\,baa$

$S \to AcA$ — (1)
$\to aBcBa$ — (3)
$\to aAbcbAa$ — (4)
$\to aaBbcbBaa$ — (5)
$\to aa\wedge bc b\wedge aa$ — (5)
$\to aab\,c\,b\wedge aa$ — (5)
$\to aab\,c\,baa$
$\to aab\,cbaa$

Que. Construct a CFG for generating an alternating sequence of 0's and 1's.

Soln Production Rules

$S \to 0T$
$S \to 1U$
$T \to 1U$
$U \to 0T$
$T \to 1$
$U \to 0$

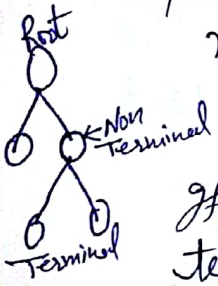
$\therefore$ CFG = $\{\{S,T,U\},\{0,1\},\{S \to 0T, S \to 1U, T \to 1U, U \to 0T, T \to 1, U \to 0\}, S\}$

for generating 0|0|01

$S \to 0T$
$\to 0|1U$
$\to 0|0T$
$\to 0|0|1U$
$\to 0|0|01$
$\to \boxed{0|0|0}$ ✓

Derivation Tree/Parse Tree :- is an ordered rooted tree that graphically represents the semantic information of a string derived from CFG.

Representation:- Root vertex – must be labeled as start non terminal Vertex – Interior node.



Terminal node – Leaf nodes / $\in$

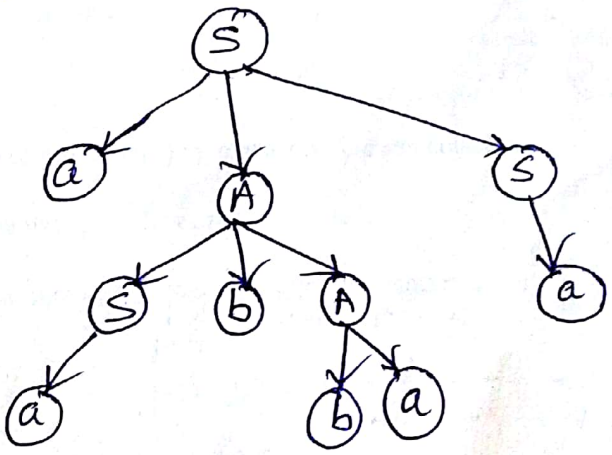
If a particular * NT is generating a string of terminals & NT of length n, then that will have n childeren. These childeren will be attached from left to right in sequence.

Example :- For the string "aabbaa" give the derivation tree using Grammer.

$G = \{\{S, A\}, \{a, b\}, \{S \to aAS,\ S \to a,\ A \to SbA,\ A \to SS,\ A \to ba\}\}$

Sol^n:-

$S \to aAS$
$\to a\,SbA\,S$
$\to aabAS$
$\to aabbaS$
$\to aabbaa$



Derivation Tree

Ex. Using $S \to aB/bA$
$A \to a/aS/bAA$
$B \to b/bS/aBB$

Draw the derivation Tree for the string
(i) aaabbabbba
(ii) aabbabb

Left most & Right most derivation ~~Tree of~~ a ~~string~~

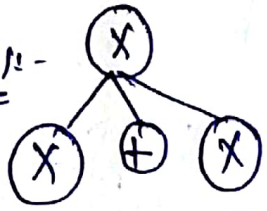
Leftmost Derivation:- A leftmost derivation is obtained by applying production to the left most variable in each step.

eg:- $X \to X+X \mid X*X \mid X \mid a$ over an alphabet $\{a\}$

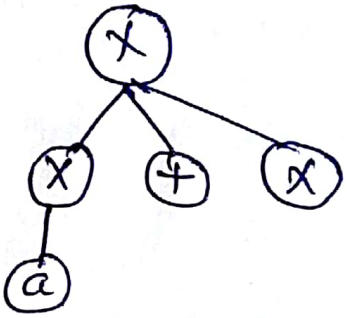
LMD:- $X \to X$ Findout LMD for "a+a*a" may be

$X \to X+X$
$\to a+X$
$\to a+X*X$
$\to a+a*X$
$\to a+a*a$

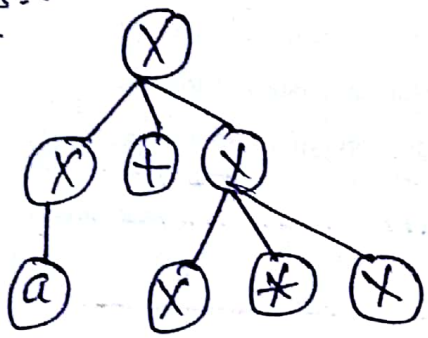
step 1:-



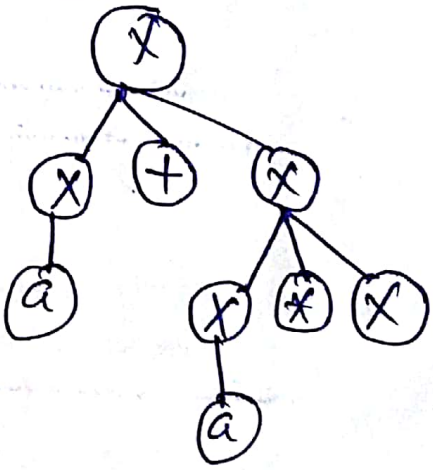
step 2:-



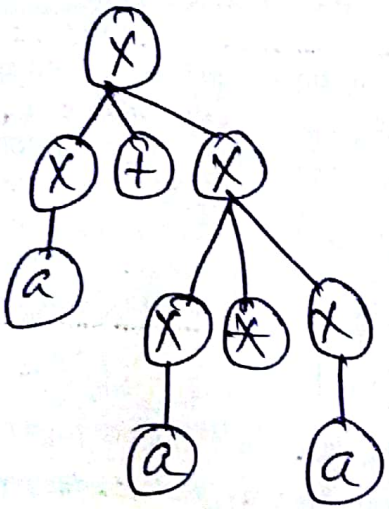
step 3:-



step 4:-



step ⑤



Rightmost Derivation:- A RMD is obtained by applying production to the RM variable in each step.

eg:- Take above example & Do your self

Ambiguous CFL :- A grammer of a Language is said to be Ambiguous if the same string can be generated in two different ways

OR

✓ A grammer is said to be ambiguous if two different derivation tree can be drawn for generating the same tree.

✓ eg①:- As in case of last example of LMD/RMD there were two derivation tree for X+X*X

example ②:- for grammer $S \to aSb/SS/\wedge$, Check that string "aabb" has ambiguous grammer.

Sol^n (i) $S \to aSb$
$\to aaSbb$
$\to aa\wedge bb$
$\to aabb$

(ii) $S \to SS$
$\to \wedge S$
$\to aSb$
$\to aaSbb$
$\to aa\wedge bb$
$\to aabb$

Draw Derivation tree for both production which show 2 Derivation tree for same string. Hence Grammer is ambiguous.

Removal of Ambiguity → Two methods

① Left Recursion
② Left factoring

① Left Recursion:-

$A \to A\alpha$
$\to A\alpha\alpha$
$\to A\alpha\alpha\alpha$
$\to A\alpha\alpha\alpha\alpha$ - - - - -

eg:- ~~$A \to A$~~ $A \to A\alpha_1/A\alpha_2/A\alpha_3 - - - A\alpha_n/\beta_1/\beta_n$

$A \to \beta_1 A'/\beta_2 A' - - - \beta_n A'$

$A' \to \alpha_1 A'/\alpha_2 A' - - - \alpha_n A'/\epsilon$

Here A is NT & α is a string of Grammer symbol

// $A \rightarrow A\alpha$

A NT A is said to be left recursive if there exist a ~~grammer~~ string of grammer symbol α such that

$A \xrightarrow{*} A\alpha$ // A is repeated every time.

eg:- $S \rightarrow S*S \mid S+S \mid a \mid b$ ← two left derivation

$S \rightarrow aS' \mid bS' \mid a \mid b$

$S' \rightarrow *SS' \mid +SS' \mid \in$

Now generate a*a+a

$S \rightarrow aS'$
$\rightarrow a*SS'$
$\rightarrow a*aS'$
$\rightarrow a*a+SS'$
$\rightarrow a*a+aS'$
$\rightarrow a*a+a$

(5) So Now we can generate only one derivation tree.

(2) Left Factoring :-

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid --- \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 --- \gamma_m$

Sol^n :-

$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2 --- \gamma_m$

$A' \rightarrow \beta_1 \mid \beta_2 \mid --- \mid \beta_n$

eg:- (1)

$A \rightarrow XY \mid XZ$

Sol^n :-

$A \rightarrow XA'$

$A' \rightarrow Y \mid Z$

(2) $A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid cdg \mid cdeB \mid cdfB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Simplification of CFGs :-

In a CFG, it may happens that all the production rules & symbols are not needed for the derivation of string. Also we may have

1. Some Non Terminals (NT) which doesn't derive any terminal string.
2. Symbols don't appear in any sentential form
3. Null production
4. Rules like NT → NT

elimination of these productions and symbols is called simplification of CFG.

steps/process to simplify CFG :-

1. Removal of Null Production.
2. Removal of Unit Production
3. Removal of Useless Symbols.

① Elimination/Removal of $\wedge$ production → A NT Symbol

A is a nullable variable if there is a production $A \to \wedge / \in$ or there is a derivation that starts at A & finally ends up with $\wedge / \in$

$\wedge / \in : A \to \text{----} \to \in / \wedge$

eg:- $S \to Xa$
$X \to aX / bX / \in$

The only null variable is X.
Substituting $\in$ in place X & rewriting the grammer we get

$S \to a$
$X \to a / b$

These productions are simply substituted in the grammer. The new set of productions after eliminating $\in$ are as

$S \to Xa / a$
$X \to aX / bX / a / b$

Example :- Consider Grammer G whose productions are

S → a | S | AB
S → aS | AB
A → ∈ | a
B → ∈ | c
D → b

} Construct a Grammer G without null production Generating L(G) - ∈.

Sol^{n} :- Here null ~~Product~~ variables are A, B.
- Now check for production containing AB or A or B on RHS.
- Here one more production S is also become ∈ because it (S) contains ~~AB~~ A & B in RHS as S → AB

Therefore ∈ variable are (S, A, B) & equivalent Grammer production are:-

S → aS ⇒ S → aS | a
S → AB ⇒ S → AB | A | B
A → a ⇒ A → a
B → c ⇒ B → c
D → b ⇒ D → b

} ⇒ S → aS | AB | A | B | a

---

(2) Unit Production :- Any production rule in the form A → B where A, B ∈ NT is called Unit production. i.e RHS production in RHS should have single NT.

eg:
S → A
B → B
B → c
c → D
D → a

} one eliminating the Unit production, we get reduced Grammer production

P_1 : S → a ✓

example:-

$S \to AB$
$A \to a$
$B \to C \mid b$
$C \to D$
$D \to E$
$E \to a$

[Here $S \to AB$ and $A \to a$ are not Unit Production.

Now consider " "

$B \to C$
$C \to D$
$D \to E$
$E \to a$

These productions after elimination & substituting a in place of E:

$B \to C$
$C \to D$
$D \to a$
$E \to a$

Similarly substitute for others we get

$B \to a$
$C \to a$
$D \to a$
$E \to a$

Hence final list of production is:

$S \to AB$
$A \to a$
$B \to a \mid b$
~~$C \to a$~~
~~$D \to a$~~
$E \to a$

(3) Removal of Useless Symbol (production) :- The production that can never take part in any derivation of any string are called useless Symbol/Production. Similarly, a variable that can never take part in derivation of any string is called a useless variable.

eg:-

$S \to abS \mid abA \mid abB$
$A \to cd$ (small c)
$B \to aB$
$C \to dc$

Note:- Production $C \to dc$ is useless

(1) because variable 'C' will never occure in any derivation

OR

The variable 'C' can not be reached from Starting Symbol S

(2) Production $B \to aB$ is also useless because there is no way it will ever terminate. So it can not produce/take part in any string generation.

So modified Grammer is

$S \to abS \mid abA$
$A \to cd$
$C \to dc$

Now if we check it again then C is not reachable. So final Grammer will be

$S \to abS \mid abA$
$A \to cd$

Note:- Remove all kinds of Productions mentioned above, first remove NULL, then Unit & finally remove useless Symbol. This order is very imp. to get correct result.

Normal Forms of CFG

In a CFG the RHS of a production can be any string of variable & Terminal (T & NT). when the production in G satisfy certain restrictions then G is said to be in NF.

Types :- (1) Chomsky Normal Form
(2) Greibach Normal form

(1) Chomsky NF :- A CFG ~~is in~~ G is in CNF if every production is of the form :-

$A \to a$
$A \to BC$
$S \to \Lambda$ is in 'G'

Note :- when 'Λ' $\in L(G)$ we assume that 'S' doesn't appear on R.H.S. of any production.

eg :- $S \to AB \mid \Lambda$, $A \to a$, $B \to b$ } is in CNF.

Algo. to convert CFG to CNF

Step (1) :- If the start symbol S occurs on some RHS, create a new start symbol S' and new production $S' \to S$.

Step (2) :- Remove NULL (Λ) Production

Step (3) :- Remove Unit production

Step (4) :- Replace each production $A \to B_1 \cdots B_n$ where $n \geq 2$ with $A \to B_1 C$ where $C \to B_2 B_3 \cdots B_n$. Repeat all the production having two or more symbol in RHS.

Step (5):- If the RHS of any production is in the form $A \to aB$ where 'a' is a terminal. A, B are NT then production is replaced by $A \to XB$ and $X \to a$. Repeat this step for every production which is in the form $A \to aB$.

example:-

$S \to ASA \mid aB$
$A \to B \mid S$
$B \to b \mid \in$, (A)

Step (1) Since S appear in RHS, we add a new state S_0 and $S_0 \to S$ is added to the production set & it becomes

$S_0 \to S,\ S \to ASA \mid aB,\ A \to B \mid S,\ B \to b \mid \Lambda$

Step (2) Now remove Λ production

$B \to \Lambda,\ A \to \Lambda$

After removing $B \to \Lambda$ we get

$S_0 \to S,\ S \to ASA \mid aB \mid a,\ A \to B \mid S \mid \Lambda,\ B \to b$

After removing $A \to \Lambda$ Production will be

$S_0 \to S,\ S \to ASA \mid aB \mid a \mid SA \mid AS \mid S,\ A \to B \mid S,\ B \to b$

Step (3) Now Remove Unit production.

After removing $S \to S$ we get

$S_0 \to S,\ S \to ASA \mid aB \mid a \mid SA \mid AS,\ A \to B \mid S,\ B \to b$

After removing $S_0 \to S$ the production set becomes:-

$S_0 \to ASA \mid aB \mid a \mid SA \mid AS,\ S \to ASA \mid aB \mid a \mid SA \mid AS$

$A \to B,\ B \to b$

After removing Unit production (A → B) we get.

$S_0 \to ASA|aB|a|AS|SA$, $S \to ASA|aB|a|AS|SA$

$A \to S|b$

$B \to b$

After removing (A → S) we get

$S_0 \to ASA|aB|a|AS|SA$, $S \to ASA|aB|a|AS|SA$

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

Step (4) :- Find more than two variable (NT) in RHS.

Here $S_0 \to ASA$, $S \to ASA$, $A \to ASA$. Now apply Step 4 & 5. Then

$S_0 \to AX|aB|a|AS|SA$

$S \to AX|aB|a|AS|SA$

$A \to b|AX|aB|a|AS|SA$

$B \to b$

$X \to SA$

Step (5) Now change the production $S_0 \to aB$, $S \to aB$, $A \to aB$ then

$S_0 \to AX|YB|a|AS|SA$

$S \to AX|YB|a|AS|SA$

$A \to b|AX|YB|a|AS|SA$

$B \to b$

$X \to SA$

$Y \to a$

Ut

Example :- CFG to CNF ⟹ $S \to aX \mid Yb \mid a$
$X \to S \mid \wedge$
$Y \to bY \mid b$

$A \to BC$
$A \to a$ ∈ CNF form

Step 1 :- $S_0 \to S$
$S \to aX \mid Yb \mid a$
$X \to S \mid \wedge$
$Y \to bY \mid b$

Step ② Remove Null Production
$S_0 \to S$
$S \to aX \mid Yb \mid a$
$X \to S$
$Y \to bY \mid b$

Step ③
$S_0 \to aX \mid Yb \mid a$
$S \to aX \mid Yb \mid a$
$X \to aX \mid Yb \mid a$
$Y \to bY \mid b$

Step ④
No more than 2 NT in R.H.S. So go to Step ⑤

Step ⑤
$S_0 \to AX \mid YB \mid a$
$S \to AX \mid YB \mid a$
$X \to AX \mid YB \mid a$
$Y \to BY \mid b$
$A \to a$
$B \to b$

CFG to GNF (Greibach NF) :-

A ~~grammar (G)~~ CFG is said to be in GNF if all production have the form:-

$A \to b$

$A \to bC_1C_2 \cdots C_n$

$S \to \epsilon$

where → $A, C_1, C_2 \cdots C_n$ are Non Terminal

→ b is Terminal

step ① :- If ~~all the Symbol~~ start symbol S occurs on some RHS, Create a new start Symbol S' & new production $S' \to S$.

Step ② :- Remove ϵ and unit production from given CFG (if any).

Step ③ :- Check whether the CFG is in CNF & Convert it to CNF if it is not.

Step ④ :- change the names of NT symbol into some A_i in ascending order of i

step ⑤ :- Alter the rules so that the NT are in ascending order such that, if the ~~production~~ production is of the form $A_i \to A_j X$ then $i < j$ and should never be $i \geq j$

Step ⑥ :- Remove Left Recursion.

check once again for GNF form we found problem once again

$A_4 \to b / bA_3A_4 / A_4A_4A_4$

$\therefore A_4 = A_4$ it produces left recursion so remove it;

step (6) Remove Left Recursion & introduce New variable

$A_4 \to b / bA_3A_4 / A_4A_4A_4$

$Z \to A_4A_4Z / A_4A_4$

$A_4 \to b / bA_3A_4 / bZ / bA_3A_4Z$

Now Grammer is :-

$A_1 \to A_2A_3 / A_4A_4$

$A_4 \to b / bA_3A_4 / bZ / bA_3A_4Z$

$Z \to A_4A_4Z / A_4A_4$

$A_2 \to b$

$A_3 \to a$

~~Now check for GNF or CNF~~

Now check / convert to GNF

Utility of NF :- every grammer is converted to some spacific form. These helps us to design some algorithms to answer certain questions.

Similarly if CFG is converted into CNF then one can answer whether the language generated by the grammer i.e $L(G)$ is Finite or not. for CFG converted into CNF the parse tree generated from CNF is always a Binary Tree same like if CFG is converted to GNF then PDA accepting the language generated by the grammer can easily be designed.

Pumping lemma for CFL :- It is used to proove that a Language is not context free.

If 'A' is a context free language then 'A' has a pumping length 'p' such that any string 's' where $|s| \geq p$ may be devided into 5 pieces $s = uvxyz$ such that following conditions must be true :-

1. uv^ixy^iz is in A for every $i \geq 0$
2. $|vy| > 0$ i.e v & y can not be $\in$ at same time.
3. $|vxy| \leq p$

Example:- Prove that, $L = \{a^n b^n c^n / n \geq 0\}$ is non context-free

Sol^n:- $L = \{abc, aabbcc, aaabbbccc, ----\}$

Let $w = aabbcc$

$w = uv^i xy^i z$

Let $u = \wedge$, $y = \wedge$

$v = aa$

$x = bb$

$z = cc$

So $w = v^i x z$

$= (aa)^i bb\ cc$

for $i = 1$, $w = aabbcc$ is in L

$i = 2$, $w = aaaabbcc$ is not in L So this is not CFL

---

ex.: Prove that $L = \{a^p / p \text{ is prime}\}$ is non CFL.

Sol^n:- p is prime means

$p = 2, 3, 5, 7, 11, ---$

So $L = \{aa, aaa, aaaaa, ----\}$ all are prime

Let $w = aa$

we know that $w = uv^i xy^i z$

$u = \wedge$
$v = a$
$x = \wedge$
$y = \wedge$
$z = a$

} So $\wedge a^i \wedge^i a \Rightarrow a^i a$

for $i = 1$, $w = aa$ is in L

$i = 2$, $w = aaa$ is in L

$i = 3$, $w = aaaa$ is not prime no.

So the given Language is not CFL.

Push Down Automata (PDA) :- PDA is

a m/c formate of CFL. If FSM is which is used to accept only CFL. A FSM/DFA can remember a finite amount of information, but PDA can remember an infinite amount of information.

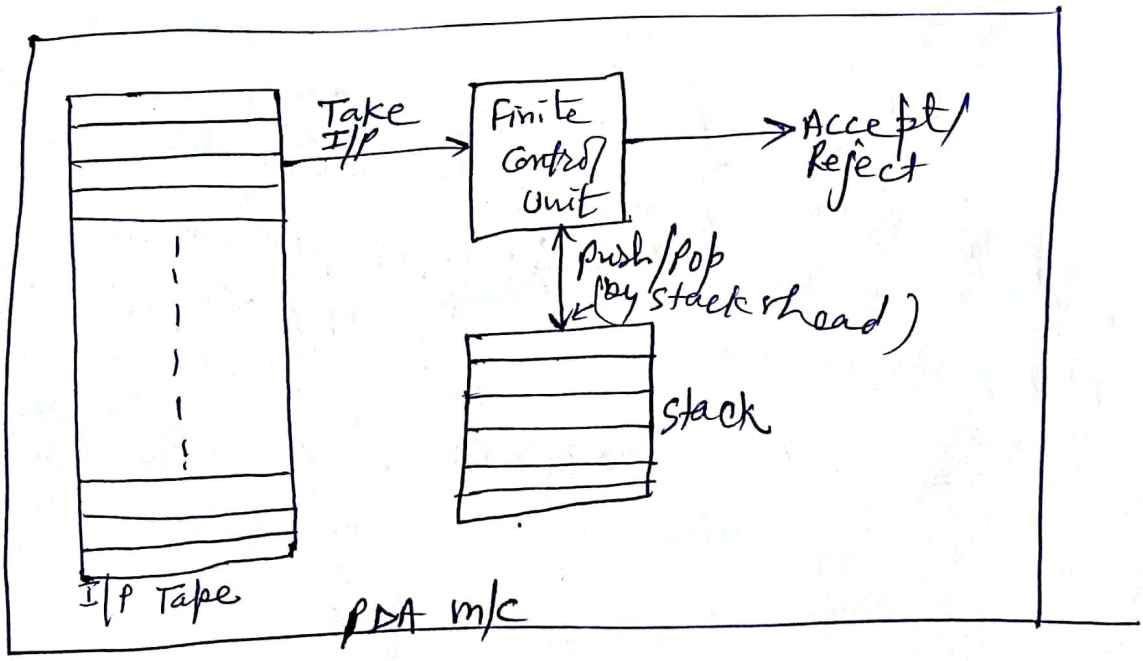
In short PDA is :-

"FSM/DFA" + "a stack"

Components of PDA :-

① I/P Tape ② Control Unit

③ stack with infinite size



- Stack head scans the top symbol of the stack.
- A stack does two operation
  - ① Push :- A new symbol is added at the top
  - ② Pop :- Top symbol is read/Remove from stack.

→ The PDA may or ~~may~~ may not read an i/p symbol, but it has to read the top of the stack in every transition.